

# The L<sup>A</sup>T<sub>E</sub>X3 Sources

The L<sup>A</sup>T<sub>E</sub>X Project\*

Released 2023-01-16

## Abstract

This is the reference documentation for the `expl3` programming environment. The `expl3` modules set up an experimental naming scheme for L<sup>A</sup>T<sub>E</sub>X commands, which allow the L<sup>A</sup>T<sub>E</sub>X programmer to systematically name functions and variables, and specify the argument types of functions.

The T<sub>E</sub>X and  $\varepsilon$ -T<sub>E</sub>X primitives are all given a new name according to these conventions. However, in the main direct use of the primitives is not required or encouraged: the `expl3` modules define an independent low-level L<sup>A</sup>T<sub>E</sub>X3 programming language.

The `expl3` modules are designed to be loaded on top of L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$ . With an up-to-date L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$  kernel, this material is loaded as part of the format. The fundamental programming code can also be loaded with other T<sub>E</sub>X formats, subject to restrictions on the full range of functionality.

---

\*E-mail: [latex-team@latex-project.org](mailto:latex-team@latex-project.org)

# Contents

<b>I Introduction</b>	<b>1</b>
<b>1 Introduction to <code>expl3</code> and this document</b>	<b>2</b>
1.1 Naming functions and variables . . . . .	2
1.1.1 Scratch variables . . . . .	5
1.1.2 Terminological inexactitude . . . . .	5
1.2 Documentation conventions . . . . .	5
1.3 Formal language conventions which apply generally . . . . .	7
1.4 $\text{\LaTeX}$ concepts not supported by $\text{\LaTeX3}$ . . . . .	7
<b>II Bootstrapping</b>	<b>8</b>
<b>2 The <code>l3bootstrap</code> package: Bootstrap code</b>	<b>9</b>
2.1 Using the $\text{\LaTeX3}$ modules . . . . .	9
<b>3 The <code>l3names</code> package: Namespace for primitives</b>	<b>11</b>
3.1 Setting up the $\text{\LaTeX3}$ programming language . . . . .	11
<b>III Programming Flow</b>	<b>12</b>
<b>4 The <code>l3basics</code> package: Basic definitions</b>	<b>13</b>
4.1 No operation functions . . . . .	13
4.2 Grouping material . . . . .	13
4.3 Control sequences and functions . . . . .	14
4.3.1 Defining functions . . . . .	14
4.3.2 Defining new functions using parameter text . . . . .	15
4.3.3 Defining new functions using the signature . . . . .	17
4.3.4 Copying control sequences . . . . .	19
4.3.5 Deleting control sequences . . . . .	20
4.3.6 Showing control sequences . . . . .	20
4.3.7 Converting to and from control sequences . . . . .	20
4.4 Analysing control sequences . . . . .	22
4.5 Using or removing tokens and arguments . . . . .	23
4.5.1 Selecting tokens from delimited arguments . . . . .	25
4.6 Predicates and conditionals . . . . .	25
4.6.1 Tests on control sequences . . . . .	27
4.6.2 Primitive conditionals . . . . .	27
4.7 Starting a paragraph . . . . .	28
4.8 Debugging support . . . . .	29

<b>5</b>	<b>The <i>l3expn</i> package: Argument expansion</b>	<b>30</b>
5.1	Defining new variants . . . . .	30
5.2	Methods for defining variants . . . . .	31
5.3	Introducing the variants . . . . .	32
5.4	Manipulating the first argument . . . . .	34
5.5	Manipulating two arguments . . . . .	36
5.6	Manipulating three arguments . . . . .	36
5.7	Unbraced expansion . . . . .	37
5.8	Preventing expansion . . . . .	38
5.9	Controlled expansion . . . . .	39
5.10	Internal functions . . . . .	42
<b>6</b>	<b>The <i>l3sort</i> package: Sorting functions</b>	<b>43</b>
6.1	Controlling sorting . . . . .	43
<b>7</b>	<b>The <i>l3tl-analysis</i> package: Analysing token lists</b>	<b>45</b>
<b>8</b>	<b>The <i>l3regex</i> package: Regular expressions in <i>T<small>E</small>X</i></b>	<b>46</b>
8.1	Syntax of regular expressions . . . . .	47
8.1.1	Regular expression examples . . . . .	47
8.1.2	Characters in regular expressions . . . . .	48
8.1.3	Characters classes . . . . .	48
8.1.4	Structure: alternatives, groups, repetitions . . . . .	49
8.1.5	Matching exact tokens . . . . .	50
8.1.6	Miscellaneous . . . . .	52
8.2	Syntax of the replacement text . . . . .	52
8.3	Pre-compiling regular expressions . . . . .	54
8.4	Matching . . . . .	55
8.5	Submatch extraction . . . . .	56
8.6	Replacement . . . . .	57
8.7	Scratch regular expressions . . . . .	59
8.8	Bugs, misfeatures, future work, and other possibilities . . . . .	59
<b>9</b>	<b>The <i>l3prg</i> package: Control structures</b>	<b>62</b>
9.1	Defining a set of conditional functions . . . . .	62
9.2	The boolean data type . . . . .	64
9.2.1	Scratch booleans . . . . .	66
9.3	Boolean expressions . . . . .	66
9.4	Logical loops . . . . .	68
9.5	Producing multiple copies . . . . .	69
9.6	Detecting <i>T<small>E</small>X</i> 's mode . . . . .	70
9.7	Primitive conditionals . . . . .	70
9.8	Nestable recursions and mappings . . . . .	70
9.8.1	Simple mappings . . . . .	71
9.9	Internal programming functions . . . . .	71

<b>10 The <code>I3sys</code> package: System/runtime functions</b>	<b>72</b>
10.1 The name of the job . . . . .	72
10.2 Date and time . . . . .	72
10.3 Engine . . . . .	73
10.4 Output format . . . . .	73
10.5 Platform . . . . .	74
10.6 Random numbers . . . . .	74
10.7 Access to the shell . . . . .	74
10.8 Loading configuration data . . . . .	75
10.8.1 Final settings . . . . .	76
<b>11 The <code>I3msg</code> package: Messages</b>	<b>77</b>
11.1 Creating new messages . . . . .	77
11.2 Customizable information for message modules . . . . .	78
11.3 Contextual information for messages . . . . .	78
11.4 Issuing messages . . . . .	80
11.4.1 Messages for showing material . . . . .	83
11.4.2 Expandable error messages . . . . .	83
11.5 Redirecting messages . . . . .	83
<b>12 The <code>I3file</code> package: File and I/O operations</b>	<b>85</b>
12.1 Input–output stream management . . . . .	85
12.1.1 Reading from files . . . . .	87
12.1.2 Writing to files . . . . .	90
12.1.3 Wrapping lines in output . . . . .	92
12.1.4 Constant input–output streams, and variables . . . . .	93
12.1.5 Primitive conditionals . . . . .	93
12.2 File operation functions . . . . .	93
<b>13 The <code>I3luatex</code> package: LuaTeX-specific functions</b>	<b>98</b>
13.1 Breaking out to Lua . . . . .	98
13.2 Lua interfaces . . . . .	99
<b>14 The <code>I3legacy</code> package: Interfaces to legacy concepts</b>	<b>101</b>
<b>IV Data types</b>	<b>102</b>
<b>15 The <code>I3tl</code> package: Token lists</b>	<b>103</b>
15.1 Creating and initialising token list variables . . . . .	103
15.2 Adding data to token list variables . . . . .	104
15.3 Token list conditionals . . . . .	105
15.3.1 Testing the first token . . . . .	107
15.4 Working with token lists as a whole . . . . .	108
15.4.1 Using token lists . . . . .	108
15.4.2 Counting and reversing token lists . . . . .	109
15.4.3 Viewing token lists . . . . .	110
15.5 Manipulating items in token lists . . . . .	111
15.5.1 Mapping over token lists . . . . .	111
15.5.2 Head and tail of token lists . . . . .	113

15.5.3	Items and ranges in token lists . . . . .	115
15.5.4	Sorting token lists . . . . .	117
15.6	Manipulating tokens in token lists . . . . .	117
15.6.1	Replacing tokens . . . . .	117
15.6.2	Reassigning category codes . . . . .	118
15.7	Constant token lists . . . . .	119
15.8	Scratch token lists . . . . .	119
<b>16</b>	<b>The <code>I3str</code> package: Strings</b>	<b>121</b>
16.1	Creating and initialising string variables . . . . .	122
16.2	Adding data to string variables . . . . .	123
16.3	String conditionals . . . . .	123
16.4	Mapping over strings . . . . .	125
16.5	Working with the content of strings . . . . .	127
16.6	Modifying string variables . . . . .	130
16.7	String manipulation . . . . .	131
16.8	Viewing strings . . . . .	132
16.9	Constant strings . . . . .	133
16.10	Scratch strings . . . . .	133
16.11	Deprecated functions . . . . .	133
<b>17</b>	<b>The <code>I3str-convert</code> package: string encoding conversions</b>	<b>134</b>
17.1	Encoding and escaping schemes . . . . .	134
17.2	Conversion functions . . . . .	136
17.3	Conversion by expansion (for PDF contexts) . . . . .	136
17.4	Possibilities, and things to do . . . . .	136
<b>18</b>	<b>The <code>I3quark</code> package: Quarks</b>	<b>138</b>
18.1	Quarks . . . . .	138
18.2	Defining quarks . . . . .	139
18.3	Quark tests . . . . .	139
18.4	Recursion . . . . .	140
18.4.1	An example of recursion with quarks . . . . .	141
18.5	Scan marks . . . . .	142
<b>19</b>	<b>The <code>I3seq</code> package: Sequences and stacks</b>	<b>143</b>
19.1	Creating and initialising sequences . . . . .	143
19.2	Appending data to sequences . . . . .	145
19.3	Recovering items from sequences . . . . .	145
19.4	Recovering values from sequences with branching . . . . .	146
19.5	Modifying sequences . . . . .	148
19.6	Sequence conditionals . . . . .	148
19.7	Mapping over sequences . . . . .	149
19.8	Using the content of sequences directly . . . . .	151
19.9	Sequences as stacks . . . . .	152
19.10	Sequences as sets . . . . .	153
19.11	Constant and scratch sequences . . . . .	154
19.12	Viewing sequences . . . . .	155

<b>20 The <code>I3int</code> package: Integers</b>	<b>156</b>
20.1 Integer expressions . . . . .	156
20.2 Creating and initialising integers . . . . .	158
20.3 Setting and incrementing integers . . . . .	159
20.4 Using integers . . . . .	159
20.5 Integer expression conditionals . . . . .	160
20.6 Integer expression loops . . . . .	161
20.7 Integer step functions . . . . .	163
20.8 Formatting integers . . . . .	164
20.9 Converting from other formats to integers . . . . .	165
20.10 Random integers . . . . .	166
20.11 Viewing integers . . . . .	167
20.12 Constant integers . . . . .	167
20.13 Scratch integers . . . . .	167
20.14 Direct number expansion . . . . .	168
20.15 Primitive conditionals . . . . .	168
<b>21 The <code>I3flag</code> package: Expandable flags</b>	<b>170</b>
21.1 Setting up flags . . . . .	170
21.2 Expandable flag commands . . . . .	171
<b>22 The <code>I3clist</code> package: Comma separated lists</b>	<b>172</b>
22.1 Creating and initialising comma lists . . . . .	173
22.2 Adding data to comma lists . . . . .	174
22.3 Modifying comma lists . . . . .	175
22.4 Comma list conditionals . . . . .	176
22.5 Mapping over comma lists . . . . .	176
22.6 Using the content of comma lists directly . . . . .	178
22.7 Comma lists as stacks . . . . .	179
22.8 Using a single item . . . . .	180
22.9 Viewing comma lists . . . . .	181
22.10 Constant and scratch comma lists . . . . .	181
<b>23 The <code>I3token</code> package: Token manipulation</b>	<b>182</b>
23.1 Creating character tokens . . . . .	183
23.2 Manipulating and interrogating character tokens . . . . .	184
23.3 Generic tokens . . . . .	187
23.4 Converting tokens . . . . .	187
23.5 Token conditionals . . . . .	188
23.6 Peeking ahead at the next token . . . . .	191
23.7 Description of all possible tokens . . . . .	195
23.8 Deprecated functions . . . . .	198

<b>24 The <code>I3prop</code> package: Property lists</b>	<b>199</b>
24.1 Creating and initialising property lists . . . . .	199
24.2 Adding and updating property list entries . . . . .	200
24.3 Recovering values from property lists . . . . .	201
24.4 Modifying property lists . . . . .	202
24.5 Property list conditionals . . . . .	202
24.6 Recovering values from property lists with branching . . . . .	203
24.7 Mapping over property lists . . . . .	204
24.8 Viewing property lists . . . . .	205
24.9 Scratch property lists . . . . .	206
24.10 Constants . . . . .	206
<b>25 The <code>I3skip</code> package: Dimensions and skips</b>	<b>207</b>
25.1 Creating and initialising <code>dim</code> variables . . . . .	207
25.2 Setting <code>dim</code> variables . . . . .	208
25.3 Utilities for dimension calculations . . . . .	208
25.4 Dimension expression conditionals . . . . .	209
25.5 Dimension expression loops . . . . .	211
25.6 Dimension step functions . . . . .	212
25.7 Using <code>dim</code> expressions and variables . . . . .	213
25.8 Viewing <code>dim</code> variables . . . . .	214
25.9 Constant dimensions . . . . .	215
25.10 Scratch dimensions . . . . .	215
25.11 Creating and initialising <code>skip</code> variables . . . . .	215
25.12 Setting <code>skip</code> variables . . . . .	216
25.13 Skip expression conditionals . . . . .	217
25.14 Using <code>skip</code> expressions and variables . . . . .	217
25.15 Viewing <code>skip</code> variables . . . . .	217
25.16 Constant skips . . . . .	218
25.17 Scratch skips . . . . .	218
25.18 Inserting skips into the output . . . . .	218
25.19 Creating and initialising <code>muskip</code> variables . . . . .	219
25.20 Setting <code>muskip</code> variables . . . . .	219
25.21 Using <code>muskip</code> expressions and variables . . . . .	220
25.22 Viewing <code>muskip</code> variables . . . . .	220
25.23 Constant muskips . . . . .	221
25.24 Scratch muskips . . . . .	221
25.25 Primitive conditional . . . . .	221
<b>26 The <code>I3keys</code> package: Key–value interfaces</b>	<b>222</b>
26.1 Creating keys . . . . .	223
26.2 Sub-dividing keys . . . . .	228
26.3 Choice and multiple choice keys . . . . .	228
26.4 Key usage scope . . . . .	231
26.5 Setting keys . . . . .	231
26.6 Handling of unknown keys . . . . .	232
26.7 Selective key setting . . . . .	232
26.8 Digesting keys . . . . .	233
26.9 Utility functions for keys . . . . .	234
26.10 Low-level interface for parsing key–val lists . . . . .	234

<b>27 The <code>I3intarray</code> package: fast global integer arrays</b>	<b>237</b>
27.1 <code>I3intarray</code> documentation . . . . .	237
27.1.1 Implementation notes . . . . .	238
<b>28 The <code>I3fp</code> package: Floating points</b>	<b>239</b>
28.1 Creating and initialising floating point variables . . . . .	241
28.2 Setting floating point variables . . . . .	241
28.3 Using floating points . . . . .	242
28.4 Floating point conditionals . . . . .	243
28.5 Floating point expression loops . . . . .	245
28.6 Some useful constants, and scratch variables . . . . .	247
28.7 Floating point exceptions . . . . .	248
28.8 Viewing floating points . . . . .	249
28.9 Floating point expressions . . . . .	250
28.9.1 Input of floating point numbers . . . . .	250
28.9.2 Precedence of operators . . . . .	251
28.9.3 Operations . . . . .	251
28.10 Disclaimer and roadmap . . . . .	258
<b>29 The <code>I3fparry</code> package: fast global floating point arrays</b>	<b>261</b>
29.1 <code>I3fparry</code> documentation . . . . .	261
<b>30 The <code>I3cctab</code> package: Category code tables</b>	<b>262</b>
30.1 Creating and initialising category code tables . . . . .	262
30.2 Using category code tables . . . . .	262
30.3 Category code table conditionals . . . . .	263
30.4 Constant category code tables . . . . .	263
<b>V Text manipulation</b>	<b>264</b>
<b>31 The <code>I3unicode</code> package: Unicode support functions</b>	<b>265</b>
<b>32 The <code>I3text</code> package: text processing</b>	<b>267</b>
32.1 Expanding text . . . . .	267
32.2 Case changing . . . . .	268
32.3 Removing formatting from text . . . . .	270
32.4 Control variables . . . . .	270
32.5 Mapping to graphemes . . . . .	271
<b>VI Typesetting</b>	<b>272</b>

<b>33 The <code>I3box</code> package: Boxes</b>	<b>273</b>
33.1 Creating and initialising boxes . . . . .	273
33.2 Using boxes . . . . .	274
33.3 Measuring and setting box dimensions . . . . .	275
33.4 Box conditionals . . . . .	276
33.5 The last box inserted . . . . .	276
33.6 Constant boxes . . . . .	276
33.7 Scratch boxes . . . . .	276
33.8 Viewing box contents . . . . .	277
33.9 Boxes and color . . . . .	277
33.10 Horizontal mode boxes . . . . .	277
33.11 Vertical mode boxes . . . . .	278
33.12 Using boxes efficiently . . . . .	280
33.13 Affine transformations . . . . .	281
33.14 Primitive box conditionals . . . . .	284
<b>34 The <code>I3coffins</code> package: Coffin code layer</b>	<b>285</b>
34.1 Creating and initialising coffins . . . . .	285
34.2 Setting coffin content and poles . . . . .	286
34.3 Coffin affine transformations . . . . .	287
34.4 Joining and using coffins . . . . .	287
34.5 Measuring coffins . . . . .	288
34.6 Coffin diagnostics . . . . .	288
34.7 Constants and variables . . . . .	289
<b>35 The <code>I3color</code> package: Color support</b>	<b>291</b>
35.1 Color in boxes . . . . .	291
35.2 Color models . . . . .	291
35.3 Color expressions . . . . .	293
35.4 Named colors . . . . .	294
35.5 Selecting colors . . . . .	294
35.6 Colors for fills and strokes . . . . .	295
35.6.1 Coloring math mode material . . . . .	295
35.7 Multiple color models . . . . .	295
35.8 Exporting color specifications . . . . .	296
35.9 Creating new color models . . . . .	297
35.9.1 Color profiles . . . . .	298
<b>36 The <code>I3pdf</code> package: Core PDF support</b>	<b>299</b>
36.1 Objects . . . . .	299
36.2 Version . . . . .	300
36.3 Page (media) size . . . . .	301
36.4 Compression . . . . .	301
36.5 Destinations . . . . .	301
36.6 Deprecated functions . . . . .	303
<b>VII Additions and removals</b>	<b>304</b>

<b>37 The <i>I3candidates</i> package: Experimental additions to <i>I3kernel</i></b>	<b>305</b>
37.1 Important notice . . . . .	305
37.2 Additions to <i>I3box</i> . . . . .	306
37.3 Additions to <i>I3expan</i> . . . . .	306
37.4 Additions to <i>I3fp</i> . . . . .	306
37.5 Additions to <i>I3file</i> . . . . .	307
37.6 Additions to <i>I3flag</i> . . . . .	307
37.7 Additions to <i>I3intarray</i> . . . . .	307
37.8 Additions to <i>I3msg</i> . . . . .	308
37.9 Additions to <i>I3prg</i> . . . . .	308
37.10 Additions to <i>I3prop</i> . . . . .	309
37.11 Additions to <i>I3seq</i> . . . . .	310
37.12 Additions to <i>I3sys</i> . . . . .	311
37.13 Additions to <i>I3tl</i> . . . . .	312
37.14 Additions to <i>I3token</i> . . . . .	313
<b>VIII Implementation</b>	<b>315</b>
<b>38 <i>I3bootstrap</i> implementation</b>	<b>316</b>
38.1 Lua $\text{\TeX}$ -specific code . . . . .	316
38.2 The <code>\pdfstrcmp</code> primitive in X $\text{\TeX}$ . . . . .	316
38.3 Loading support Lua code . . . . .	317
38.4 Engine requirements . . . . .	317
38.5 Extending allocators . . . . .	318
38.6 The $\text{\LaTeX}3$ code environment . . . . .	319
<b>39 <i>I3names</i> implementation</b>	<b>322</b>
<b>40 <i>I3kernel-functions</i>: kernel-reserved functions</b>	<b>349</b>
40.1 Internal kernel functions . . . . .	349
40.2 Kernel backend functions . . . . .	356
<b>41 <i>I3basics</i> implementation</b>	<b>358</b>
41.1 Renaming some $\text{\TeX}$ primitives (again) . . . . .	358
41.2 Defining some constants . . . . .	360
41.3 Defining functions . . . . .	360
41.4 Selecting tokens . . . . .	361
41.5 Gobbling tokens from input . . . . .	363
41.6 Debugging and patching later definitions . . . . .	363
41.7 Conditional processing and definitions . . . . .	364
41.8 Dissecting a control sequence . . . . .	370
41.9 Exist or free . . . . .	372
41.10 Preliminaries for new functions . . . . .	374
41.11 Defining new functions . . . . .	376
41.12 Copying definitions . . . . .	377
41.13 Undefining functions . . . . .	378
41.14 Generating parameter text from argument count . . . . .	378
41.15 Defining functions from a given number of arguments . . . . .	379
41.16 Using the signature to define functions . . . . .	380

41.17	Checking control sequence equality . . . . .	382
41.18	Diagnostic functions . . . . .	382
41.19	Decomposing a macro definition . . . . .	384
41.20	Doing nothing functions . . . . .	385
41.21	Breaking out of mapping functions . . . . .	385
41.22	Starting a paragraph . . . . .	386
<b>42</b>	<b>l3expan implementation</b>	<b>387</b>
42.1	General expansion . . . . .	387
42.2	Hand-tuned definitions . . . . .	391
42.3	Last-unbraced versions . . . . .	394
42.4	Preventing expansion . . . . .	396
42.5	Controlled expansion . . . . .	397
42.6	Emulating e-type expansion . . . . .	398
42.7	Defining function variants . . . . .	405
42.8	Definitions with the automated technique . . . . .	415
<b>43</b>	<b>l3sort implementation</b>	<b>417</b>
43.1	Variables . . . . .	417
43.2	Finding available \toks registers . . . . .	418
43.3	Protected user commands . . . . .	420
43.4	Merge sort . . . . .	422
43.5	Expandable sorting . . . . .	425
43.6	Messages . . . . .	430
<b>44</b>	<b>l3tl-analysis implementation</b>	<b>433</b>
44.1	Internal functions . . . . .	433
44.2	Internal format . . . . .	433
44.3	Variables and helper functions . . . . .	434
44.4	Plan of attack . . . . .	436
44.5	Disabling active characters . . . . .	437
44.6	First pass . . . . .	438
44.7	Second pass . . . . .	443
44.8	Mapping through the analysis . . . . .	446
44.9	Showing the results . . . . .	447
44.10	Peeking ahead . . . . .	449
44.11	Messages . . . . .	456
<b>45</b>	<b>l3regex implementation</b>	<b>458</b>
45.1	Plan of attack . . . . .	458
45.2	Helpers . . . . .	459
45.2.1	Constants and variables . . . . .	461
45.2.2	Testing characters . . . . .	462
45.2.3	Internal auxiliaries . . . . .	463
45.2.4	Character property tests . . . . .	466
45.2.5	Simple character escape . . . . .	468
45.3	Compiling . . . . .	473
45.3.1	Variables used when compiling . . . . .	474
45.3.2	Generic helpers used when compiling . . . . .	476
45.3.3	Mode . . . . .	477

45.3.4	Framework . . . . .	479
45.3.5	Quantifiers . . . . .	482
45.3.6	Raw characters . . . . .	485
45.3.7	Character properties . . . . .	487
45.3.8	Anchoring and simple assertions . . . . .	488
45.3.9	Character classes . . . . .	488
45.3.10	Groups and alternations . . . . .	492
45.3.11	Catcodes and csnames . . . . .	494
45.3.12	Raw token lists with \u . . . . .	498
45.3.13	Other . . . . .	502
45.3.14	Showing regexes . . . . .	502
45.4	Building . . . . .	509
45.4.1	Variables used while building . . . . .	509
45.4.2	Framework . . . . .	510
45.4.3	Helpers for building an NFA . . . . .	513
45.4.4	Building classes . . . . .	514
45.4.5	Building groups . . . . .	516
45.4.6	Others . . . . .	520
45.5	Matching . . . . .	522
45.5.1	Variables used when matching . . . . .	522
45.5.2	Matching: framework . . . . .	525
45.5.3	Using states of the NFA . . . . .	528
45.5.4	Actions when matching . . . . .	529
45.6	Replacement . . . . .	531
45.6.1	Variables and helpers used in replacement . . . . .	531
45.6.2	Query and brace balance . . . . .	533
45.6.3	Framework . . . . .	534
45.6.4	Submatches . . . . .	537
45.6.5	Csnames in replacement . . . . .	539
45.6.6	Characters in replacement . . . . .	540
45.6.7	An error . . . . .	544
45.7	User functions . . . . .	544
45.7.1	Variables and helpers for user functions . . . . .	547
45.7.2	Matching . . . . .	549
45.7.3	Extracting submatches . . . . .	550
45.7.4	Replacement . . . . .	555
45.7.5	Peeking ahead . . . . .	558
45.8	Messages . . . . .	564
45.9	Code for tracing . . . . .	570
<b>46</b>	<b>I3prg implementation</b>	<b>572</b>
46.1	Primitive conditionals . . . . .	572
46.2	Defining a set of conditional functions . . . . .	572
46.3	The boolean data type . . . . .	572
46.4	Internal auxiliaries . . . . .	574
46.5	Boolean expressions . . . . .	575
46.6	Logical loops . . . . .	580
46.7	Producing multiple copies . . . . .	581
46.8	Detecting T <sub>E</sub> X's mode . . . . .	583
46.9	Internal programming functions . . . . .	583

<b>47 l3sys implementation</b>	<b>585</b>
47.1 Kernel code . . . . .	585
47.1.1 Detecting the engine . . . . .	585
47.1.2 Randomness . . . . .	587
47.1.3 Platform . . . . .	587
47.1.4 Configurations . . . . .	587
47.1.5 Access to the shell . . . . .	589
47.2 Dynamic (every job) code . . . . .	592
47.2.1 The name of the job . . . . .	592
47.2.2 Time and date . . . . .	593
47.2.3 Random numbers . . . . .	593
47.2.4 Access to the shell . . . . .	595
47.2.5 Held over from l3file . . . . .	595
47.3 Last-minute code . . . . .	596
47.3.1 Detecting the output . . . . .	596
47.3.2 Configurations . . . . .	596
<b>48 l3msg implementation</b>	<b>598</b>
48.1 Internal auxiliaries . . . . .	598
48.2 Creating messages . . . . .	598
48.3 Messages: support functions and text . . . . .	600
48.4 Showing messages: low level mechanism . . . . .	601
48.5 Displaying messages . . . . .	603
48.6 Kernel-specific functions . . . . .	612
48.7 Internal messages . . . . .	613
48.8 Expandable errors . . . . .	619
48.9 Message formatting . . . . .	620
<b>49 l3file implementation</b>	<b>621</b>
49.1 Input operations . . . . .	621
49.1.1 Variables and constants . . . . .	621
49.1.2 Stream management . . . . .	622
49.1.3 Reading input . . . . .	625
49.2 Output operations . . . . .	628
49.2.1 Variables and constants . . . . .	628
49.2.2 Internal auxiliaries . . . . .	629
49.3 Stream management . . . . .	629
49.3.1 Deferred writing . . . . .	631
49.3.2 Immediate writing . . . . .	632
49.3.3 Special characters for writing . . . . .	632
49.3.4 Hard-wrapping lines to a character count . . . . .	633
49.4 File operations . . . . .	642
49.4.1 Internal auxiliaries . . . . .	644
49.5 GetIdInfo . . . . .	659
49.6 Checking the version of kernel dependencies . . . . .	660
49.7 Messages . . . . .	662
49.8 Functions delayed from earlier modules . . . . .	663

<b>50 <i>I3luatex</i> implementation</b>	<b>665</b>
50.1 Breaking out to Lua . . . . .	665
50.2 Messages . . . . .	666
50.3 Lua functions for internal use . . . . .	667
50.4 Preserving iniTeX Lua data for runs . . . . .	672
<b>51 <i>I3legacy</i> Implementation</b>	<b>674</b>
<b>52 <i>I3tl</i> implementation</b>	<b>676</b>
52.1 Functions . . . . .	676
52.2 Constant token lists . . . . .	678
52.3 Adding to token list variables . . . . .	678
52.4 Internal quarks and quark-query functions . . . . .	681
52.5 Reassigning token list category codes . . . . .	681
52.6 Modifying token list variables . . . . .	684
52.7 Token list conditionals . . . . .	688
52.8 Mapping over token lists . . . . .	693
52.9 Using token lists . . . . .	695
52.10 Working with the contents of token lists . . . . .	696
52.11 The first token from a token list . . . . .	699
52.12 Token by token changes . . . . .	704
52.13 Using a single item . . . . .	706
52.14 Viewing token lists . . . . .	709
52.15 Internal scan marks . . . . .	711
52.16 Scratch token lists . . . . .	711
<b>53 <i>I3str</i> implementation</b>	<b>712</b>
53.1 Internal auxiliaries . . . . .	712
53.2 Creating and setting string variables . . . . .	713
53.3 Modifying string variables . . . . .	714
53.4 String comparisons . . . . .	715
53.5 Mapping over strings . . . . .	718
53.6 Accessing specific characters in a string . . . . .	720
53.7 Counting characters . . . . .	725
53.8 The first character in a string . . . . .	726
53.9 String manipulation . . . . .	727
53.10 Viewing strings . . . . .	730
<b>54 <i>I3str-convert</i> implementation</b>	<b>732</b>
54.1 Helpers . . . . .	732
54.1.1 Variables and constants . . . . .	732
54.2 String conditionals . . . . .	734
54.3 Conversions . . . . .	735
54.3.1 Producing one byte or character . . . . .	735
54.3.2 Mapping functions for conversions . . . . .	736
54.3.3 Error-reporting during conversion . . . . .	737
54.3.4 Framework for conversions . . . . .	738
54.3.5 Byte unescape and escape . . . . .	742
54.3.6 Native strings . . . . .	743
54.3.7 <i>clist</i> . . . . .	744

54.3.8	8-bit encodings . . . . .	744
54.4	Messages . . . . .	747
54.5	Escaping definitions . . . . .	748
54.5.1	Unescape methods . . . . .	749
54.5.2	Escape methods . . . . .	753
54.6	Encoding definitions . . . . .	755
54.6.1	UTF-8 support . . . . .	755
54.6.2	UTF-16 support . . . . .	760
54.6.3	UTF-32 support . . . . .	765
54.7	PDF names and strings by expansion . . . . .	768
54.7.1	ISO 8859 support . . . . .	769
<b>55</b>	<b>I3quark implementation</b>	<b>786</b>
55.1	Quarks . . . . .	786
55.2	Scan marks . . . . .	794
<b>56</b>	<b>I3seq implementation</b>	<b>796</b>
56.1	Allocation and initialisation . . . . .	797
56.2	Appending data to either end . . . . .	800
56.3	Modifying sequences . . . . .	801
56.4	Sequence conditionals . . . . .	803
56.5	Recovering data from sequences . . . . .	805
56.6	Mapping over sequences . . . . .	809
56.7	Using sequences . . . . .	813
56.8	Sequence stacks . . . . .	814
56.9	Viewing sequences . . . . .	815
56.10	Scratch sequences . . . . .	816
<b>57</b>	<b>I3int implementation</b>	<b>817</b>
57.1	Integer expressions . . . . .	818
57.2	Creating and initialising integers . . . . .	820
57.3	Setting and incrementing integers . . . . .	822
57.4	Using integers . . . . .	823
57.5	Integer expression conditionals . . . . .	823
57.6	Integer expression loops . . . . .	827
57.7	Integer step functions . . . . .	828
57.8	Formatting integers . . . . .	830
57.9	Converting from other formats to integers . . . . .	835
57.10	Viewing integer . . . . .	838
57.11	Random integers . . . . .	839
57.12	Constant integers . . . . .	839
57.13	Scratch integers . . . . .	839
57.14	Integers for earlier modules . . . . .	840
<b>58</b>	<b>I3flag implementation</b>	<b>841</b>
58.1	Non-expandable flag commands . . . . .	841
58.2	Expandable flag commands . . . . .	842

<b>59 <i>I3clist</i> implementation</b>	<b>844</b>
59.1 Removing spaces around items . . . . .	845
59.2 Allocation and initialisation . . . . .	846
59.3 Adding data to comma lists . . . . .	848
59.4 Comma lists as stacks . . . . .	849
59.5 Modifying comma lists . . . . .	851
59.6 Comma list conditionals . . . . .	854
59.7 Mapping over comma lists . . . . .	855
59.8 Using comma lists . . . . .	859
59.9 Using a single item . . . . .	861
59.10 Viewing comma lists . . . . .	863
59.11 Scratch comma lists . . . . .	864
<b>60 <i>I3token</i> implementation</b>	<b>865</b>
60.1 Internal auxiliaries . . . . .	865
60.2 Manipulating and interrogating character tokens . . . . .	865
60.3 Creating character tokens . . . . .	868
60.4 Generic tokens . . . . .	874
60.5 Token conditionals . . . . .	875
60.6 Peeking ahead at the next token . . . . .	884
<b>61 <i>I3prop</i> implementation</b>	<b>892</b>
61.1 Internal auxiliaries . . . . .	893
61.2 Allocation and initialisation . . . . .	894
61.3 Accessing data in property lists . . . . .	896
61.4 Property list conditionals . . . . .	901
61.5 Recovering values from property lists with branching . . . . .	902
61.6 Mapping over property lists . . . . .	903
61.7 Viewing property lists . . . . .	904
<b>62 <i>I3skip</i> implementation</b>	<b>906</b>
62.1 Length primitives renamed . . . . .	906
62.2 Internal auxiliaries . . . . .	906
62.3 Creating and initialising <i>dim</i> variables . . . . .	906
62.4 Setting <i>dim</i> variables . . . . .	907
62.5 Utilities for dimension calculations . . . . .	908
62.6 Dimension expression conditionals . . . . .	909
62.7 Dimension expression loops . . . . .	911
62.8 Dimension step functions . . . . .	912
62.9 Using <i>dim</i> expressions and variables . . . . .	914
62.10 Viewing <i>dim</i> variables . . . . .	916
62.11 Constant dimensions . . . . .	916
62.12 Scratch dimensions . . . . .	916
62.13 Creating and initialising <i>skip</i> variables . . . . .	916
62.14 Setting <i>skip</i> variables . . . . .	918
62.15 Skip expression conditionals . . . . .	918
62.16 Using <i>skip</i> expressions and variables . . . . .	919
62.17 Inserting skips into the output . . . . .	919
62.18 Viewing <i>skip</i> variables . . . . .	920
62.19 Constant skips . . . . .	920

62.20	Scratch skips . . . . .	920
62.21	Creating and initialising <code>muskip</code> variables . . . . .	920
62.22	Setting <code>muskip</code> variables . . . . .	921
62.23	Using <code>muskip</code> expressions and variables . . . . .	922
62.24	Viewing <code>muskip</code> variables . . . . .	922
62.25	Constant muskips . . . . .	923
62.26	Scratch muskips . . . . .	923
<b>63</b>	<b>l3keys Implementation</b>	<b>924</b>
63.1	Low-level interface . . . . .	924
63.2	Constants and variables . . . . .	931
63.2.1	Internal auxiliaries . . . . .	933
63.3	The key defining mechanism . . . . .	934
63.4	Turning properties into actions . . . . .	936
63.5	Creating key properties . . . . .	943
63.6	Setting keys . . . . .	949
63.7	Utilities . . . . .	958
63.8	Messages . . . . .	960
<b>64</b>	<b>l3intarray implementation</b>	<b>962</b>
64.1	Lua implementation . . . . .	962
64.1.1	Allocating arrays . . . . .	962
64.1.2	Array items . . . . .	965
64.1.3	Working with contents of integer arrays . . . . .	967
64.2	Font dimension based implementation . . . . .	968
64.2.1	Allocating arrays . . . . .	969
64.2.2	Array items . . . . .	970
64.2.3	Working with contents of integer arrays . . . . .	972
64.3	Common parts . . . . .	974
64.3.1	Random arrays . . . . .	974
<b>65</b>	<b>l3fp implementation</b>	<b>976</b>
<b>66</b>	<b>l3fp-aux implementation</b>	<b>977</b>
66.1	Access to primitives . . . . .	977
66.2	Internal representation . . . . .	977
66.3	Using arguments and semicolons . . . . .	978
66.4	Constants, and structure of floating points . . . . .	979
66.5	Overflow, underflow, and exact zero . . . . .	982
66.6	Expanding after a floating point number . . . . .	982
66.7	Other floating point types . . . . .	983
66.8	Packing digits . . . . .	986
66.9	Decimate (dividing by a power of 10) . . . . .	989
66.10	Functions for use within primitive conditional branches . . . . .	991
66.11	Integer floating points . . . . .	992
66.12	Small integer floating points . . . . .	993
66.13	Fast string comparison . . . . .	994
66.14	Name of a function from its l3fp-parse name . . . . .	994
66.15	Messages . . . . .	994

<b>67 I3fp-traps Implementation</b>	<b>995</b>
67.1 Flags . . . . .	995
67.2 Traps . . . . .	995
67.3 Errors . . . . .	999
67.4 Messages . . . . .	999
<b>68 I3fp-round implementation</b>	<b>1001</b>
68.1 Rounding tools . . . . .	1001
68.2 The round function . . . . .	1005
<b>69 I3fp-parse implementation</b>	<b>1010</b>
69.1 Work plan . . . . .	1010
69.1.1 Storing results . . . . .	1011
69.1.2 Precedence and infix operators . . . . .	1012
69.1.3 Prefix operators, parentheses, and functions . . . . .	1015
69.1.4 Numbers and reading tokens one by one . . . . .	1016
69.2 Main auxiliary functions . . . . .	1018
69.3 Helpers . . . . .	1019
69.4 Parsing one number . . . . .	1020
69.4.1 Numbers: trimming leading zeros . . . . .	1026
69.4.2 Number: small significand . . . . .	1027
69.4.3 Number: large significand . . . . .	1029
69.4.4 Number: beyond 16 digits, rounding . . . . .	1031
69.4.5 Number: finding the exponent . . . . .	1034
69.5 Constants, functions and prefix operators . . . . .	1037
69.5.1 Prefix operators . . . . .	1037
69.5.2 Constants . . . . .	1040
69.5.3 Functions . . . . .	1041
69.6 Main functions . . . . .	1042
69.7 Infix operators . . . . .	1044
69.7.1 Closing parentheses and commas . . . . .	1045
69.7.2 Usual infix operators . . . . .	1047
69.7.3 Juxtaposition . . . . .	1048
69.7.4 Multi-character cases . . . . .	1048
69.7.5 Ternary operator . . . . .	1049
69.7.6 Comparisons . . . . .	1049
69.8 Tools for functions . . . . .	1051
69.9 Messages . . . . .	1054
<b>70 I3fp-assign implementation</b>	<b>1055</b>
70.1 Assigning values . . . . .	1055
70.2 Updating values . . . . .	1056
70.3 Showing values . . . . .	1056
70.4 Some useful constants and scratch variables . . . . .	1057

<b>71 I3fp-logic Implementation</b>	<b>1058</b>
71.1 Syntax of internal functions . . . . .	1058
71.2 Tests . . . . .	1058
71.3 Comparison . . . . .	1059
71.4 Floating point expression loops . . . . .	1062
71.5 Extrema . . . . .	1065
71.6 Boolean operations . . . . .	1067
71.7 Ternary operator . . . . .	1068
<b>72 I3fp-basics Implementation</b>	<b>1070</b>
72.1 Addition and subtraction . . . . .	1070
72.1.1 Sign, exponent, and special numbers . . . . .	1071
72.1.2 Absolute addition . . . . .	1073
72.1.3 Absolute subtraction . . . . .	1075
72.2 Multiplication . . . . .	1079
72.2.1 Signs, and special numbers . . . . .	1079
72.2.2 Absolute multiplication . . . . .	1081
72.3 Division . . . . .	1083
72.3.1 Signs, and special numbers . . . . .	1083
72.3.2 Work plan . . . . .	1084
72.3.3 Implementing the significand division . . . . .	1087
72.4 Square root . . . . .	1092
72.5 About the sign and exponent . . . . .	1099
72.6 Operations on tuples . . . . .	1100
<b>73 I3fp-extended implementation</b>	<b>1102</b>
73.1 Description of fixed point numbers . . . . .	1102
73.2 Helpers for numbers with extended precision . . . . .	1103
73.3 Multiplying a fixed point number by a short one . . . . .	1104
73.4 Dividing a fixed point number by a small integer . . . . .	1104
73.5 Adding and subtracting fixed points . . . . .	1105
73.6 Multiplying fixed points . . . . .	1106
73.7 Combining product and sum of fixed points . . . . .	1107
73.8 Extended-precision floating point numbers . . . . .	1110
73.9 Dividing extended-precision numbers . . . . .	1112
73.10 Inverse square root of extended precision numbers . . . . .	1116
73.11 Converting from fixed point to floating point . . . . .	1118
<b>74 I3fp-expo implementation</b>	<b>1120</b>
74.1 Logarithm . . . . .	1120
74.1.1 Work plan . . . . .	1120
74.1.2 Some constants . . . . .	1121
74.1.3 Sign, exponent, and special numbers . . . . .	1121
74.1.4 Absolute ln . . . . .	1121
74.2 Exponential . . . . .	1128
74.2.1 Sign, exponent, and special numbers . . . . .	1128
74.3 Power . . . . .	1133
74.4 Factorial . . . . .	1139

<b>75 l3fp-trig Implementation</b>	<b>1142</b>
75.1 Direct trigonometric functions . . . . .	1143
75.1.1 Filtering special cases . . . . .	1143
75.1.2 Distinguishing small and large arguments . . . . .	1146
75.1.3 Small arguments . . . . .	1147
75.1.4 Argument reduction in degrees . . . . .	1147
75.1.5 Argument reduction in radians . . . . .	1148
75.1.6 Computing the power series . . . . .	1156
75.2 Inverse trigonometric functions . . . . .	1158
75.2.1 Arctangent and arccotangent . . . . .	1159
75.2.2 Arcsine and arccosine . . . . .	1164
75.2.3 Arccosecant and arcsecant . . . . .	1166
<b>76 l3fp-convert implementation</b>	<b>1168</b>
76.1 Dealing with tuples . . . . .	1168
76.2 Trimming trailing zeros . . . . .	1168
76.3 Scientific notation . . . . .	1169
76.4 Decimal representation . . . . .	1170
76.5 Token list representation . . . . .	1172
76.6 Formatting . . . . .	1173
76.7 Convert to dimension or integer . . . . .	1173
76.8 Convert from a dimension . . . . .	1174
76.9 Use and eval . . . . .	1175
76.10 Convert an array of floating points to a comma list . . . . .	1176
<b>77 l3fp-random Implementation</b>	<b>1178</b>
77.1 Engine support . . . . .	1178
77.2 Random floating point . . . . .	1182
77.3 Random integer . . . . .	1182
<b>78 l3farray implementation</b>	<b>1188</b>
78.1 Allocating arrays . . . . .	1188
78.2 Array items . . . . .	1189
<b>79 l3cctab implementation</b>	<b>1193</b>
79.1 Variables . . . . .	1193
79.2 Allocating category code tables . . . . .	1194
79.3 Saving category code tables . . . . .	1195
79.4 Using category code tables . . . . .	1196
79.5 Category code table conditionals . . . . .	1201
79.6 Constant category code tables . . . . .	1202
79.7 Messages . . . . .	1204
<b>80 l3unicode implementation</b>	<b>1206</b>
80.1 User functions . . . . .	1206
80.2 Data loader . . . . .	1210

<b>81 I3text implementation</b>	<b>1220</b>
81.1 Internal auxiliaries . . . . .	1220
81.2 Utilities . . . . .	1221
81.3 Codepoint utilities . . . . .	1224
81.4 Configuration variables . . . . .	1227
81.5 Expansion to formatted text . . . . .	1228
<b>82 I3text-case implementation</b>	<b>1238</b>
82.1 Case changing . . . . .	1238
<b>83 I3text-map implementation</b>	<b>1267</b>
83.1 Mapping to text . . . . .	1267
<b>84 I3text-purify implementation</b>	<b>1275</b>
84.1 Purifying text . . . . .	1275
84.2 Accent and letter-like data for purifying text . . . . .	1281
<b>85 I3box implementation</b>	<b>1289</b>
85.1 Support code . . . . .	1289
85.2 Creating and initialising boxes . . . . .	1289
85.3 Measuring and setting box dimensions . . . . .	1290
85.4 Using boxes . . . . .	1291
85.5 Box conditionals . . . . .	1292
85.6 The last box inserted . . . . .	1292
85.7 Constant boxes . . . . .	1292
85.8 Scratch boxes . . . . .	1293
85.9 Viewing box contents . . . . .	1293
85.10 Horizontal mode boxes . . . . .	1294
85.11 Vertical mode boxes . . . . .	1296
85.12 Affine transformations . . . . .	1299
<b>86 I3coffins Implementation</b>	<b>1309</b>
86.1 Coffins: data structures and general variables . . . . .	1309
86.2 Basic coffin functions . . . . .	1310
86.3 Measuring coffins . . . . .	1316
86.4 Coffins: handle and pole management . . . . .	1316
86.5 Coffins: calculation of pole intersections . . . . .	1320
86.6 Affine transformations . . . . .	1322
86.7 Aligning and typesetting of coffins . . . . .	1330
86.8 Coffin diagnostics . . . . .	1335
86.9 Messages . . . . .	1341

<b>87 <i>I3color</i> Implementation</b>	<b>1342</b>
87.1 Basics . . . . .	1342
87.2 Predefined color names . . . . .	1343
87.3 Setup . . . . .	1344
87.4 Utility functions . . . . .	1344
87.5 Model conversion . . . . .	1345
87.6 Color expressions . . . . .	1346
87.7 Selecting colors (and color models) . . . . .	1355
87.8 Math color . . . . .	1357
87.9 Fill and stroke color . . . . .	1360
87.10 Defining named colors . . . . .	1360
87.11 Exporting colors . . . . .	1363
87.12 Additional color models . . . . .	1365
87.13 Applying profiles . . . . .	1379
87.14 Diagnostics . . . . .	1380
87.15 Messages . . . . .	1381
<b>88 Deprecated functions</b>	<b>1384</b>
<b>89 <i>I3pdf</i> implementation</b>	<b>1385</b>
89.1 Compression . . . . .	1385
89.2 Objects . . . . .	1386
89.3 Version . . . . .	1386
89.4 Page size . . . . .	1388
89.5 Destinations . . . . .	1388
89.6 PDF Page size (media box) . . . . .	1388
89.7 Deprecated functions . . . . .	1389
<b>90 <i>I3candidates</i> Implementation</b>	<b>1390</b>
90.1 Additions to <i>I3box</i> . . . . .	1390
90.1.1 Viewing part of a box . . . . .	1390
90.2 Additions to <i>I3flag</i> . . . . .	1392
90.3 Additions to <i>I3msg</i> . . . . .	1393
90.4 Additions to <i>I3prg</i> . . . . .	1394
90.5 Additions to <i>I3prop</i> . . . . .	1395
90.6 Additions to <i>I3seq</i> . . . . .	1395
90.7 Additions to <i>I3sys</i> . . . . .	1400
90.8 Additions to <i>I3file</i> . . . . .	1401
90.9 Additions to <i>I3tl</i> . . . . .	1401
90.9.1 Building a token list . . . . .	1401
90.9.2 Other additions to <i>I3tl</i> . . . . .	1405
90.10 Additions to <i>I3token</i> . . . . .	1405

<b>91 I3deprecation implementation</b>	<b>1408</b>
91.1 Patching definitions to deprecate . . . . .	1408
91.2 Removed functions . . . . .	1410
91.3 Deprecated I3basics functions . . . . .	1414
91.4 Deprecated I3str functions . . . . .	1414
91.5 Deprecated I3seq functions . . . . .	1415
91.6 Deprecated I3sys functions . . . . .	1415
91.7 Deprecated I3tl functions . . . . .	1416
91.8 Deprecated I3token functions . . . . .	1416
<b>Index</b>	<b>1419</b>

**Part I**  
**Introduction**

# Chapter 1

## Introduction to `expl3` and this document

This document is intended to act as a comprehensive reference manual for the `expl3` language. A general guide to the L<sup>A</sup>T<sub>E</sub>X3 programming language is found in [expl3.pdf](#).

### 1.1 Naming functions and variables

L<sup>A</sup>T<sub>E</sub>X3 does not use @ as a “letter” for defining internal macros. Instead, the symbols \_ and : are used in internal macro names to provide structure. The name of each *function* is divided into logical units using \_, while : separates the *name* of the function from the *argument specifier* (“arg-spec”). This describes the arguments expected by the function. In most cases, each argument is represented by a single letter. The complete list of arg-spec letters for a function is referred to as the *signature* of the function.

Each function name starts with the *module* to which it belongs. Thus apart from a small number of very basic functions, all `expl3` function names contain at least one underscore to divide the module name from the descriptive name of the function. For example, all functions concerned with comma lists are in module `clist` and begin `\clist_`.

Every function must include an argument specifier. For functions which take no arguments, this will be blank and the function name will end :. Most functions take one or more arguments, and use the following argument specifiers:

**N** and **n** These mean *no manipulation*, of a single token for N and of a set of tokens given in braces for n. Both pass the argument through exactly as given. Usually, if you use a single token for an n argument, all will be well.

**c** This means *csname*, and indicates that the argument will be turned into a csname before being used. So `\foo:c {ArgumentOne}` will act in the same way as `\foo:N \ArgumentOne`. All macros that appear in the argument are expanded. An internal error will occur if the result of expansion inside a c-type argument is not a series of character tokens.

**V** and **v** These mean *value of variable*. The V and v specifiers are used to get the content of a variable without needing to worry about the underlying T<sub>E</sub>X structure containing the data. A V argument will be a single token (similar to N), for example

`\foo:V \MyVariable`; on the other hand, using `v` a csname is constructed first, and then the value is recovered, for example `\foo:v {MyVariable}`.

- This means *expansion once*. In general, the `V` and `v` specifiers are favoured over `o` for recovering stored information. However, `o` is useful for correctly processing information with delimited arguments.
- ✖ The `x` specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The `\edef` primitive carries out this type of expansion. Functions which feature an `x`-type argument are *not* expandable.
- ✖ The `e` specifier is in many respects identical to `x`, but with a very different implementation. Functions which feature an `e`-type argument may be expandable. The drawback is that `e` is extremely slow (often more than 200 times slower) in older engines, more precisely in non-LuaTeX engines older than 2019.
- ✖ The `f` specifier stands for *full expansion*, and in contrast to `x` stops at the first non-expandable token (reading the argument from left to right) without trying to expand it. If this token is a *<space token>*, it is gobbled, and thus won't be part of the resulting argument. For example, when setting a token list variable (a macro used for storage), the sequence

```
\tl_set:Nn \l_mya_tl { A }
\tl_set:Nn \l_myb_tl { B }
\tl_set:Nf \l_mya_tl { \l_mya_tl \l_myb_tl }
```

will leave `\l_mya_tl` with the content `A\l_myb_tl`, as `A` cannot be expanded and so terminates expansion before `\l_myb_tl` is considered.

**T and F** For logic tests, there are the branch specifiers `T` (*true*) and `F` (*false*). Both specifiers treat the input in the same way as `n` (no change), but make the logic much easier to see.

- ✖ The letter `p` indicates `\TeX` *parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- ✖ Finally, there is the `w` specifier for *weird* arguments. This covers everything else, but mainly applies to delimited values (where the argument must be terminated by some specified string).
- ✖ The `D` stands for **Do not use**. All of the `\TeX` primitives are initially `\let` to a `D` name, and some are then given a second name. These functions have no standardized syntax, they are engine dependent and their name can change without warning, thus their use is *strongly discouraged* in package code: programmers should instead use the interfaces documented in [interface3.pdf](#)<sup>1</sup>.

Notice that the argument specifier describes how the argument is processed prior to being passed to the underlying function. For example, `\foo:c` will take its argument, convert it to a control sequence and pass it to `\foo:N`.

---

<sup>1</sup>If a primitive offers a functionality not yet in the kernel, programmers and users are encouraged to write to the `\TeX-L` mailing list (<mailto:LATEX-L@listserv.uni-heidelberg.de>) describing their use-case and intended behaviour, so that a possible interface can be discussed. Temporarily, while an interface is not provided, programmers may use the procedure described in the [l3styleguide.pdf](#).

Variables are named in a similar manner to functions, but begin with a single letter to define the type of variable:

**c** Constant: global parameters whose value should not be changed.

**g** Parameters whose value should only be set globally.

**l** Parameters whose value should only be set locally.

Each variable name is then build up in a similar way to that of a function, typically starting with the module<sup>2</sup> name and then a descriptive part. Variables end with a short identifier to show the variable type:

**clist** Comma separated list.

**dim** “Rigid” lengths.

**fp** Floating-point values;

**int** Integer-valued count register.

**muskip** “Rubber” lengths for use in mathematics.

**seq** “Sequence”: a data-type used to implement lists (with access at both ends) and stacks.

**skip** “Rubber” lengths.

**str** String variables: contain character data.

**t1** Token list variables: placeholder for a token list.

Applying V-type or v-type expansion to variables of one of the above types is supported, while it is not supported for the following variable types:

**bool** Either true or false.

**box** Box register.

**coffin** A “box with handles” — a higher-level data type for carrying out **box** alignment operations.

**flag** Integer that can be incremented expandably.

**fparray** Fixed-size array of floating point values.

**intarray** Fixed-size array of integers.

**ior/iow** An input or output stream, for reading from or writing to, respectively.

**prop** Property list: analogue of dictionary or associative arrays in other languages.

**regex** Regular expression.

---

<sup>2</sup>The module names are not used in case of generic scratch registers defined in the data type modules, e.g., the **int** module contains some scratch variables called **\l\_tmpa\_int**, **\l\_tmpb\_int**, and so on. In such a case adding the module name up front to denote the module and in the back to indicate the type, as in **\l\_int\_tmpa\_int** would be very unreadable.

### 1.1.1 Scratch variables

Modules focussed on variable usage typically provide four scratch variables, two local and two global, with names of the form  $\backslash\langle scope\rangle\_tmpa\_<type>/\backslash\langle scope\rangle\_tmpb\_<type>$ . These are never used by the core code. The nature of TeX grouping means that as with any other scratch variable, these should only be set and used with no intervening third-party code.

### 1.1.2 Terminological inexactitude

A word of warning. In this document, and others referring to the `expl3` programming modules, we often refer to “variables” and “functions” as if they were actual constructs from a real programming language. In truth, TeX is a macro processor, and functions are simply macros that may or may not take arguments and expand to their replacement text. Many of the common variables are *also* macros, and if placed into the input stream will simply expand to their definition as well — a “function” with no arguments and a “token list variable” are almost the same.<sup>3</sup> On the other hand, some “variables” are actually registers that must be initialised and their values set and retrieved with specific functions.

The conventions of the `expl3` code are designed to clearly separate the ideas of “macros that contain data” and “macros that contain code”, and a consistent wrapper is applied to all forms of “data” whether they be macros or actually registers. This means that sometimes we will use phrases like “the function returns a value”, when actually we just mean “the macro expands to something”. Similarly, the term “execute” might be used in place of “expand” or it might refer to the more specific case of “processing in TeX’s stomach” (if you are familiar with the TeXbook parlance).

If in doubt, please ask; chances are we’ve been hasty in writing certain definitions and need to be told to tighten up our terminology.

## 1.2 Documentation conventions

This document is typeset with the experimental `!3doc` class; several conventions are used to help describe the features of the code. A number of conventions are used here to make the documentation clearer.

Each group of related functions is given in a box. For a function with a “user” name, this might read:

---

```
\ExplSyntaxOn \ExplSyntaxOn ... \ExplSyntaxOff  
\ExplSyntaxOff
```

The textual description of how the function works would appear here. The syntax of the function is shown in mono-spaced text to the right of the box. In this example, the function takes no arguments and so the name of the function is simply reprinted.

For programming functions, which use `_` and `:` in their name there are a few additional conventions: If two related functions are given with identical names but different argument specifiers, these are termed *variants* of each other, and the latter functions are printed in grey to show this more clearly. They will carry out the same function but will take different types of argument:

---

<sup>3</sup>TeXnically, functions with no arguments are `\long` while token list variables are not.

---

`\seq_new:N \seq_new:N <sequence>`

`\seq_new:c`

When a number of variants are described, the arguments are usually illustrated only for the base function. Here, *<sequence>* indicates that `\seq_new:N` expects the name of a sequence. From the argument specifier, `\seq_new:c` also expects a sequence name, but as a name rather than as a control sequence. Each argument given in the illustration should be described in the following text.

**Fully expandable functions** Some functions are fully expandable, which allows them to be used within an x-type or e-type argument (in plain TeX terms, inside an `\edef` or `\expanded`), as well as within an f-type argument. These fully expandable functions are indicated in the documentation by a star:

---

`\cs_to_str:N ∗ \cs_to_str:N <cs>`

As with other functions, some text should follow which explains how the function works. Usually, only the star will indicate that the function is expandable. In this case, the function expects a *<cs>*, shorthand for a *<control sequence>*.

**Restricted expandable functions** A few functions are fully expandable but cannot be fully expanded within an f-type argument. In this case a hollow star is used to indicate this:

---

`\seq_map_function:NN ∗ \seq_map_function:NN <seq> <function>`

**Conditional functions** Conditional (*if*) functions are normally defined in three variants, with T, F and TF argument specifiers. This allows them to be used for different “true”/“false” branches, depending on which outcome the conditional is being used to test. To indicate this without repetition, this information is given in a shortened form:

---

`\sys_if_engine_xetex:TF ∗ \sys_if_engine_xetex:TF {<true code>} {<false code>}`

The underlining and italic of TF indicates that three functions are available:

- `\sys_if_engine_xetex:T`
- `\sys_if_engine_xetex:F`
- `\sys_if_engine_xetex:TF`

Usually, the illustration will use the TF variant, and so both *<true code>* and *<false code>* will be shown. The two variant forms T and F take only *<true code>* and *<false code>*, respectively. Here, the star also shows that this function is expandable. With some minor exceptions, *all* conditional functions in the `expl3` modules should be defined in this way.

Variables, constants and so on are described in a similar manner:

---

`\l_tmpa_t1` A short piece of text will describe the variable: there is no syntax illustration in this case.

In some cases, the function is similar to one in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> or plain TeX. In these cases, the text will include an extra “**TeXhackers note**” section:

---

`\token_to_str:N`  $\star$  `\token_to_str:N`  $\langle token \rangle$

---

The normal description text.

**TeXhackers note:** Detail for the experienced TeX or L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  programmer. In this case, it would point out that this function is the TeX primitive `\string`.

**Changes to behaviour** When new functions are added to expl3, the date of first inclusion is given in the documentation. Where the documented behaviour of a function changes after it is first introduced, the date of the update will also be given. This means that the programmer can be sure that any release of expl3 after the date given will contain the function of interest with expected behaviour as described. Note that changes to code internals, including bug fixes, are not recorded in this way *unless* they impact on the expected behaviour.

### 1.3 Formal language conventions which apply generally

As this is a formal reference guide for L<sup>A</sup>T<sub>E</sub>X3 programming, the descriptions of functions are intended to be reasonably “complete”. However, there is also a need to avoid repetition. Formal ideas which apply to general classes of function are therefore summarised here.

For tests which have a TF argument specification, the test if evaluated to give a logically TRUE or FALSE result. Depending on this result, either the  $\langle true\ code \rangle$  or the  $\langle false\ code \rangle$  will be left in the input stream. In the case where the test is expandable, and a predicate (\_p) variant is available, the logical value determined by the test is left in the input stream: this will typically be part of a larger logical construct.

### 1.4 TeX concepts not supported by L<sup>A</sup>T<sub>E</sub>X3

The TeX concept of an “\outer” macro is *not supported* at all by L<sup>A</sup>T<sub>E</sub>X3. As such, the functions provided here may break when used on top of L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  if \outer tokens are used in the arguments.

**Part II**  
**Bootstrapping**

## Chapter 2

# The `\l3bootstrap` package Bootstrap code

### 2.1 Using the L<sup>A</sup>T<sub>E</sub>X3 modules

The modules documented in `source3` are designed to be used on top of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> and are loaded all as one with the usual `\usepackage{expl3}` or `\RequirePackage{expl3}` instructions.

As the modules use a coding syntax different from standard L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> it provides a few functions for setting it up.

---

`\ExplSyntaxOn`    `\ExplSyntaxOn <code>` `\ExplSyntaxOff`

`\ExplSyntaxOff`

Updated: 2011-08-13

The `\ExplSyntaxOn` function switches to a category code regime in which spaces and new lines are ignored, and in which the colon (`:`) and underscore (`_`) are treated as “letters”, thus allowing access to the names of code functions and variables. Within this environment, `~` is used to input a space. The `\ExplSyntaxOff` reverts to the document category code regime.

**TEXhackers note:** Spaces introduced by `~` behave much in the same way as normal space characters in the standard category code regime: they are ignored after a control word or at the start of a line, and multiple consecutive `~` are equivalent to a single one. However, `~` is *not* ignored at the end of a line.

---

`\ProvidesExplPackage`    `\RequirePackage{expl3}`

`\ProvidesExplClass`

`\ProvidesExplFile`

Updated: 2017-03-19

`\ProvidesExplPackage {<package>} {<date>} {<version>} {<description>}`

These functions act broadly in the same way as the corresponding L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> kernel functions `\ProvidesPackage`, `\ProvidesClass` and `\ProvidesFile`. However, they also implicitly switch `\ExplSyntaxOn` for the remainder of the code with the file. At the end of the file, `\ExplSyntaxOff` will be called to reverse this. (This is the same concept as L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> provides in turning on `\makeatletter` within package and class code.) The `<date>` should be given in the format `<year>/<month>/<day>` or in the ISO date format `<year>-<month>-<day>`. If the `<version>` is given then it will be prefixed with `v` in the package identifier line.

---

```
\GetIdInfo           \RequirePackage{l3bootstrap}
\GetIdInfo $Id: <SVN info field> $ {\<description>}
```

---

Updated: 2012-06-04

Extracts all information from a SVN field. Spaces are not ignored in these fields. The information pieces are stored in separate control sequences with `\ExplFileName` for the part of the file name leading up to the period, `\ExplFileDate` for date, `\ExplFileVersion` for version and `\ExplFileDescription` for the description.

To summarize: Every single package using this syntax should identify itself using one of the above methods. Special care is taken so that every package or class file loaded with `\RequirePackage` or similar are loaded with usual L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  category codes and the L<sup>A</sup>T<sub>E</sub>X3 category code scheme is reloaded when needed afterwards. See implementation for details. If you use the `\GetIdInfo` command you can use the information when loading a package with

```
\ProvidesExplPackage{\ExplFileName}
{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

## Chapter 3

# The **I3names** package Namespace for primitives

### 3.1 Setting up the L<sup>A</sup>T<sub>E</sub>X3 programming language

This module is at the core of the L<sup>A</sup>T<sub>E</sub>X3 programming language. It performs the following tasks:

- defines new names for all T<sub>E</sub>X primitives;
- emulate required primitives not provided by default in LuaT<sub>E</sub>X;
- switches to the category code régime for programming;

This module is entirely dedicated to primitives (and emulations of these), which should not be used directly within L<sup>A</sup>T<sub>E</sub>X3 code (outside of “kernel-level” code). As such, the primitives are not documented here: *The T<sub>E</sub>Xbook*, *T<sub>E</sub>X by Topic* and the manuals for pdfT<sub>E</sub>X, X<sub>E</sub>T<sub>E</sub>X, LuaT<sub>E</sub>X, pT<sub>E</sub>X and upT<sub>E</sub>X should be consulted for details of the primitives. These are named `\tex_{name}:D`, typically based on the primitive’s `{name}` in pdfT<sub>E</sub>X and omitting a leading pdf when the primitive is not related to pdf output.

**Part III**  
**Programming Flow**

## Chapter 4

# The **I3basics** package

## Basic definitions

As the name suggest this package holds some basic definitions which are needed by most or all other packages in this set.

Here we describe those functions that are used all over the place. With that we mean functions dealing with the construction and testing of control sequences. Furthermore the basic parts of conditional processing are covered; conditional processing dealing with specific data types is described in the modules specific for the respective data types.

### 4.1 No operation functions

---

`\prg_do_nothing: * \prg_do_nothing:`

An expandable function which does nothing at all: leaves nothing in the input stream after a single expansion.

---

`\scan_stop: \scan_stop:`

A non-expandable function which does nothing. Does not vanish on expansion but produces no typeset output.

### 4.2 Grouping material

---

`\group_begin: \group_begin:  
\group_end: \group_end:`

These functions begin and end a group for definition purposes. Assignments are local to groups unless carried out in a global manner. (A small number of exceptions to this rule will be noted as necessary elsewhere in this document.) Each `\group_begin:` must be matched by a `\group_end:,` although this does not have to occur within the same function. Indeed, it is often necessary to start a group within one function and finish it within another, for example when seeking to use non-standard category codes.

---

`\group_insert_after:N \group_insert_after:N <token>`

Adds `<token>` to the list of `<tokens>` to be inserted when the current group level ends. The list of `<tokens>` to be inserted is empty at the beginning of a group: multiple applications of `\group_insert_after:N` may be used to build the inserted list one `<token>` at a time. The current group level may be closed by a `\group_end:` function or by a token with category code 2 (close-group), namely a `}` if standard category codes apply.

---

`\group_show_list: \group_show_list:`  
`\group_log_list: \group_log_list:`

---

New: 2021-05-11 Display (to the terminal or log file) a list of the groups that are currently opened. This is intended for tracking down problems.

**TeXhackers note:** This is a wrapper around the `\showgroups` primitive.

## 4.3 Control sequences and functions

As `TeX` is a macro language, creating new functions means creating macros. At point of use, a function is replaced by the replacement text (“code”) in which each parameter in the code (`#1, #2, etc.`) is replaced the appropriate arguments absorbed by the function. In the following, `<code>` is therefore used as a shorthand for “replacement text”.

Functions which are not “protected” are fully expanded inside an `x` expansion. In contrast, “protected” functions are not expanded within `x` expansions.

### 4.3.1 Defining functions

Functions can be created with no requirement that they are declared first (in contrast to variables, which must always be declared). Declaring a function before setting up the code means that the name chosen is checked and an error raised if it is already in use. The name of a function can be checked at the point of definition using the `\cs_new...` functions: this is recommended for all functions which are defined for the first time.

There are three ways to define new functions. All classes define a function to expand to the substitution text. Within the substitution text the actual parameters are substituted for the formal parameters (`#1, #2, ...`).

**new** Create a new function with the `new` scope, such as `\cs_new:Npn`. The definition is global and results in an error if it is already defined.

**set** Create a new function with the `set` scope, such as `\cs_set:Npn`. The definition is restricted to the current `TeX` group and does not result in an error if the function is already defined.

**gset** Create a new function with the `gset` scope, such as `\cs_gset:Npn`. The definition is global and does not result in an error if the function is already defined.

Within each set of scope there are different ways to define a function. The differences depend on restrictions on the actual parameters and the expandability of the resulting function.

**nopar** Create a new function with the `nopar` restriction, such as `\cs_set_nopar:Npn`. The parameter may not contain `\par` tokens.

**protected** Create a new function with the **protected** restriction, such as `\cs_set_protected:Npn`. The parameter may contain `\par` tokens but the function will not expand within an `x`-type or `e`-type expansion.

Finally, the functions in Subsections 4.3.2 and 4.3.3 are primarily meant to define *base functions* only. Base functions can only have the following argument specifiers:

**N and n** No manipulation.

**T and F** Functionally equivalent to `n` (you are actually encouraged to use the family of `\prg_new_conditional:` functions described in Section 9.1).

**p and w** These are special cases.

The `\cs_new:` functions below (and friends) do not stop you from using other argument specifiers in your function names, but they do not handle expansion for you. You should define the base function and then use `\cs_generate_variant:Nn` to generate custom variants as described in Section 5.2.

### 4.3.2 Defining new functions using parameter text

---

`\cs_new:Npn \cs_new:Npn <function> <parameters> {{code}}`

`\cs_new:cpx` Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1, #2, etc.`) will be replaced by those absorbed by the function. The definition is global and an error results if the `<function>` is already defined.

---

`\cs_new_nopar:Npn \cs_new_nopar:Npn <function> <parameters> {{code}}`

`\cs_new_nopar:cpx` Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1, #2, etc.`) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The definition is global and an error results if the `<function>` is already defined.

---

`\cs_new_protected:Npn \cs_new_protected:Npn <function> <parameters> {{code}}`

`\cs_new_protected:cpx` Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1, #2, etc.`) will be replaced by those absorbed by the function. The `<function>` will not expand within an `x`-type or `e`-type argument. The definition is global and an error results if the `<function>` is already defined.

---

`\cs_new_protected_nopar:Npn \cs_new_protected_nopar:Npn <function> <parameters> {{code}}`

`\cs_new_protected_nopar:cpx` Creates `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (`#1, #2, etc.`) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The `<function>` will not expand within an `x`-type or `e`-type argument. The definition is global and an error results if the `<function>` is already defined.

---

`\cs_set:Npn` `\cs_set:Npn <function> <parameters> {<code>}`  
`\cs_set:cpx` Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

---

`\cs_set_nopar:Npn` `\cs_set_nopar:Npn <function> <parameters> {<code>}`  
`\cs_set_nopar:cpx` Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is restricted to the current TeX group level.

---

`\cs_set_protected:Npn` `\cs_set_protected:Npn <function> <parameters> {<code>}`  
`\cs_set_protected:cpx` Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is restricted to the current TeX group level. The `<function>` will not expand within an x-type or e-type argument.

---

`\cs_set_protected_nopar:Npn` `\cs_set_protected_nopar:Npn <function> <parameters> {<code>}`  
`\cs_set_protected_nopar:cpx`

Sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is restricted to the current TeX group level. The `<function>` will not expand within an x-type or e-type argument.

---

`\cs_gset:Npn` `\cs_gset:Npn <function> <parameters> {<code>}`  
`\cs_gset:cpx` Globally sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is *not* restricted to the current TeX group level: the assignment is global.

---

`\cs_gset_nopar:Npn` `\cs_gset_nopar:Npn <function> <parameters> {<code>}`  
`\cs_gset_nopar:cpx` Globally sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. When the `<function>` is used the `<parameters>` absorbed cannot contain `\par` tokens. The assignment of a meaning to the `<function>` is *not* restricted to the current TeX group level: the assignment is global.

---

`\cs_gset_protected:Npn` `\cs_gset_protected:Npn <function> <parameters> {<code>}`  
`\cs_gset_protected:cpx` Globally sets `<function>` to expand to `<code>` as replacement text. Within the `<code>`, the `<parameters>` (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the `<function>` is *not* restricted to the current TeX group level: the assignment is global. The `<function>` will not expand within an x-type or e-type argument.

---

<code>\cs_gset_protected_nopar:Npn</code>	<code>\cs_gset_protected_nopar:Npn &lt;function&gt; &lt;parameters&gt; {&lt;code&gt;}</code>
<code>\cs_gset_protected_nopar:cpx</code>	
<code>\cs_gset_protected_nopar:Npx</code>	
<code>\cs_gset_protected_nopar:cpn</code>	

---

Globally sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain \par tokens. The assignment of a meaning to the *<function>* is *not* restricted to the current TeX group level: the assignment is global. The *<function>* will not expand within an x-type or e-type argument.

#### 4.3.3 Defining new functions using the signature

---

<code>\cs_new:Nn</code>	<code>\cs_new:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_new:(cn Nx cx)</code>	

---

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The definition is global and an error results if the *<function>* is already defined.

---

<code>\cs_new_nopar:Nn</code>	<code>\cs_new_nopar:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_new_nopar:(cn Nx cx)</code>	

---

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain \par tokens. The definition is global and an error results if the *<function>* is already defined.

---

<code>\cs_new_protected:Nn</code>	<code>\cs_new_protected:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_new_protected:(cn Nx cx)</code>	

---

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The *<function>* will not expand within an x-type or e-type argument. The definition is global and an error results if the *<function>* is already defined.

---

<code>\cs_new_protected_nopar:Nn</code>	<code>\cs_new_protected_nopar:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_new_protected_nopar:(cn Nx cx)</code>	

---

Creates *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. When the *<function>* is used the *<parameters>* absorbed cannot contain \par tokens. The *<function>* will not expand within an x-type or e-type argument. The definition is global and an error results if the *<function>* is already defined.

---

<code>\cs_set:Nn</code>	<code>\cs_set:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_set:(cn Nx cx)</code>	

---

Sets *<function>* to expand to *<code>* as replacement text. Within the *<code>*, the number of *<parameters>* is detected automatically from the function signature. These *<parameters>* (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the *<function>* is restricted to the current TeX group level.

<code>\cs_set_nopar:Nn</code>	<code>\cs_set_nopar:Nn &lt;function&gt; {{code}}</code>	Sets <i>&lt;function&gt;</i> to expand to <i>&lt;code&gt;</i> as replacement text. Within the <i>&lt;code&gt;</i> , the number of <i>&lt;parameters&gt;</i> is detected automatically from the function signature. These <i>&lt;parameters&gt;</i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i>&lt;function&gt;</i> is used the <i>&lt;parameters&gt;</i> absorbed cannot contain \par tokens. The assignment of a meaning to the <i>&lt;function&gt;</i> is restricted to the current T <sub>E</sub> X group level.
<code>\cs_set_protected:Nn</code>	<code>\cs_set_protected:Nn &lt;function&gt; {{code}}</code>	Sets <i>&lt;function&gt;</i> to expand to <i>&lt;code&gt;</i> as replacement text. Within the <i>&lt;code&gt;</i> , the number of <i>&lt;parameters&gt;</i> is detected automatically from the function signature. These <i>&lt;parameters&gt;</i> (#1, #2, etc.) will be replaced by those absorbed by the function. The <i>&lt;function&gt;</i> will not expand within an x-type or e-type argument. The assignment of a meaning to the <i>&lt;function&gt;</i> is restricted to the current T <sub>E</sub> X group level.
<code>\cs_set_protected_nopar:Nn</code>	<code>\cs_set_protected_nopar:Nn &lt;function&gt; {{code}}</code>	Sets <i>&lt;function&gt;</i> to expand to <i>&lt;code&gt;</i> as replacement text. Within the <i>&lt;code&gt;</i> , the number of <i>&lt;parameters&gt;</i> is detected automatically from the function signature. These <i>&lt;parameters&gt;</i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i>&lt;function&gt;</i> is used the <i>&lt;parameters&gt;</i> absorbed cannot contain \par tokens. The <i>&lt;function&gt;</i> will not expand within an x-type or e-type argument. The assignment of a meaning to the <i>&lt;function&gt;</i> is restricted to the current T <sub>E</sub> X group level.
<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn &lt;function&gt; {{code}}</code>	Sets <i>&lt;function&gt;</i> to expand to <i>&lt;code&gt;</i> as replacement text. Within the <i>&lt;code&gt;</i> , the number of <i>&lt;parameters&gt;</i> is detected automatically from the function signature. These <i>&lt;parameters&gt;</i> (#1, #2, etc.) will be replaced by those absorbed by the function. The assignment of a meaning to the <i>&lt;function&gt;</i> is global.
<code>\cs_gset_nopar:Nn</code>	<code>\cs_gset_nopar:Nn &lt;function&gt; {{code}}</code>	Sets <i>&lt;function&gt;</i> to expand to <i>&lt;code&gt;</i> as replacement text. Within the <i>&lt;code&gt;</i> , the number of <i>&lt;parameters&gt;</i> is detected automatically from the function signature. These <i>&lt;parameters&gt;</i> (#1, #2, etc.) will be replaced by those absorbed by the function. When the <i>&lt;function&gt;</i> is used the <i>&lt;parameters&gt;</i> absorbed cannot contain \par tokens. The assignment of a meaning to the <i>&lt;function&gt;</i> is global.
<code>\cs_gset_protected:Nn</code>	<code>\cs_gset_protected:Nn &lt;function&gt; {{code}}</code>	Sets <i>&lt;function&gt;</i> to expand to <i>&lt;code&gt;</i> as replacement text. Within the <i>&lt;code&gt;</i> , the number of <i>&lt;parameters&gt;</i> is detected automatically from the function signature. These <i>&lt;parameters&gt;</i> (#1, #2, etc.) will be replaced by those absorbed by the function. The <i>&lt;function&gt;</i> will not expand within an x-type or e-type argument. The assignment of a meaning to the <i>&lt;function&gt;</i> is global.

---

```
\cs_gset_protected_nopar:Nn      \cs_gset_protected_nopar:Nn <function> {<code>}
\cs_gset_protected_nopar:(cn|Nx|cx)
```

Sets  $\langle function \rangle$  to expand to  $\langle code \rangle$  as replacement text. Within the  $\langle code \rangle$ , the number of  $\langle parameters \rangle$  is detected automatically from the function signature. These  $\langle parameters \rangle$  ( $\#1, \#2, \dots$ ) will be replaced by those absorbed by the function. When the  $\langle function \rangle$  is used the  $\langle parameters \rangle$  absorbed cannot contain  $\backslash par$  tokens. The  $\langle function \rangle$  will not expand within an x-type or e-type argument. The assignment of a meaning to the  $\langle function \rangle$  is global.

---

```
\cs_generate_from_arg_count:NNnn   \cs_generate_from_arg_count:NNnn <function> <creator> {<number>}
\cs_generate_from_arg_count:(cNnn|Ncnn) {<code>}
```

Updated: 2012-01-14

Uses the  $\langle creator \rangle$  function (which should have signature Npn, for example  $\backslash cs\_new:Npn$ ) to define a  $\langle function \rangle$  which takes  $\langle number \rangle$  arguments and has  $\langle code \rangle$  as replacement text. The  $\langle number \rangle$  of arguments is an integer expression, evaluated as detailed for  $\backslash int\_eval:n$ .

#### 4.3.4 Copying control sequences

Control sequences (not just functions as defined above) can be set to have the same meaning using the functions described here. Making two control sequences equivalent means that the second control sequence is a *copy* of the first (rather than a pointer to it). Thus the old and new control sequence are not tied together: changes to one are not reflected in the other.

In the following text “cs” is used as an abbreviation for “control sequence”.

---

```
\cs_new_eq:NN      \cs_new_eq:NN <cs1> <cs2>
\cs_new_eq:(Nc|cN|cc) \cs_new_eq:NN <cs1> <token>
```

Globally creates  $\langle control\ sequence_1 \rangle$  and sets it to have the same meaning as  $\langle control\ sequence_2 \rangle$  or  $\langle token \rangle$ . The second control sequence may subsequently be altered without affecting the copy.

---

```
\cs_set_eq:NN      \cs_set_eq:NN <cs1> <cs2>
\cs_set_eq:(Nc|cN|cc) \cs_set_eq:NN <cs1> <token>
```

Sets  $\langle control\ sequence_1 \rangle$  to have the same meaning as  $\langle control\ sequence_2 \rangle$  (or  $\langle token \rangle$ ). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the  $\langle control\ sequence_1 \rangle$  is restricted to the current TeX group level.

---

```
\cs_gset_eq:NN      \cs_gset_eq:NN <cs1> <cs2>
\cs_gset_eq:(Nc|cN|cc) \cs_gset_eq:NN <cs1> <token>
```

Globally sets  $\langle control\ sequence_1 \rangle$  to have the same meaning as  $\langle control\ sequence_2 \rangle$  (or  $\langle token \rangle$ ). The second control sequence may subsequently be altered without affecting the copy. The assignment of a meaning to the  $\langle control\ sequence_1 \rangle$  is *not* restricted to the current TeX group level: the assignment is global.

### 4.3.5 Deleting control sequences

There are occasions where control sequences need to be deleted. This is handled in a very simple manner.

<code>\cs_undefine:N</code>	<code>\cs_undefine:N &lt;control sequence&gt;</code>
<code>\cs_undefine:c</code>	Sets <code>&lt;control sequence&gt;</code> to be globally undefined.
Updated: 2011-09-15	

### 4.3.6 Showing control sequences

<code>\cs_meaning:N</code>	<code>\cs_meaning:N &lt;control sequence&gt;</code>
<code>\cs_meaning:c</code>	This function expands to the <i>meaning</i> of the <code>&lt;control sequence&gt;</code> control sequence. For a macro, this includes the <code>&lt;replacement text&gt;</code> .
Updated: 2011-12-22	

**TeXhackers note:** This is TeX's `\meaning` primitive. For tokens that are not control sequences, it is more logical to use `\token_to_meaning:N`. The `c` variant correctly reports undefined arguments.

<code>\cs_show:N</code>	<code>\cs_show:N &lt;control sequence&gt;</code>
<code>\cs_show:c</code>	Displays the definition of the <code>&lt;control sequence&gt;</code> on the terminal.
Updated: 2017-02-14	

**TeXhackers note:** This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

<code>\cs_log:N</code>	<code>\cs_log:N &lt;control sequence&gt;</code>
<code>\cs_log:c</code>	Writes the definition of the <code>&lt;control sequence&gt;</code> in the log file. See also <code>\cs_show:N</code> which displays the result in the terminal.
New: 2014-08-22 Updated: 2017-02-14	

### 4.3.7 Converting to and from control sequences

<code>\use:c</code>	<code>\use:c {&lt;control sequence name&gt;}</code>
---------------------	---

Expands the `<control sequence name>` until only characters remain, and then converts this into a control sequence. This process requires two expansions. As in other `c`-type arguments the `<control sequence name>` must, when fully expanded, consist of character tokens, typically a mixture of category code 10 (space), 11 (letter) and 12 (other).

As an example of the `\use:c` function, both

```
\use:c { a b c }
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\use:c { \tl_use:N \l_my_tl }
```

would be equivalent to

\abc

after two expansions of \use:c.

---

\cs\_if\_exist\_use:N \* \cs\_if\_exist\_use:N *(control sequence)*  
\cs\_if\_exist\_use:c \* \cs\_if\_exist\_use:NTF *(control sequence)* {*true code*} {*false code*}  
\cs\_if\_exist\_use:NTF \* Tests whether the *(control sequence)* is currently defined according to the conditional  
\cs\_if\_exist\_use:cTF \* \cs\_if\_exist:NTF (whether as a function or another control sequence type), and if it is  
New: 2012-11-10 inserts the *(control sequence)* into the input stream followed by the *(true code)*. Otherwise  
the *(false code)* is used.

---

\cs:w \* \cs:w *(control sequence name)* \cs\_end:  
\cs\_end: \* Converts the given *(control sequence name)* into a single control sequence token. This  
process requires one expansion. The content for *(control sequence name)* may be literal  
material or from other expandable functions. The *(control sequence name)* must, when  
fully expanded, consist of character tokens which are not active: typically of category  
code 10 (space), 11 (letter) or 12 (other), or a mixture of these.

**TeXhackers note:** These are the TeX primitives \csname and \endcsname.

As an example of the \cs:w and \cs\_end: functions, both

\cs:w a b c \cs\_end:

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { a b c }
\cs:w \tl_use:N \l_my_tl \cs_end:
```

would be equivalent to

\abc

after one expansion of \cs:w.

---

\cs\_to\_str:N \* \cs\_to\_str:N *(control sequence)*

Converts the given *(control sequence)* into a series of characters with category code 12  
(other), except spaces, of category code 10. The result does *not* include the current  
escape token, contrarily to \token\_to\_str:N. Full expansion of this function requires  
exactly 2 expansion steps, and so an x-type or e-type expansion, or two o-type expansions  
are required to convert the *(control sequence)* to a sequence of characters in the input  
stream. In most cases, an f-expansion is correct as well, but this loses a space at the  
start of the result.

## 4.4 Analysing control sequences

---

\cs\_split\_function:N ∗ \cs\_split\_function:N *function*

New: 2018-04-06

Splits the *function* into the *name* (*i.e.* the part before the colon) and the *signature* (*i.e.* after the colon). This information is then placed in the input stream in three parts: the *name*, the *signature* and a logic token indicating if a colon was found (to differentiate variables from function names). The *name* does not include the escape character, and both the *name* and *signature* are made up of tokens with category code 12 (other).

The next three functions decompose TeX macros into their constituent parts: if the *token* passed is not a macro then no decomposition can occur. In the latter case, all three functions leave \scan\_stop: in the input stream.

---

\cs\_prefix\_spec:N ∗ \cs\_prefix\_spec:N *token*

New: 2019-02-27

If the *token* is a macro, this function leaves the applicable TeX prefixes in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_prefix_spec:N \next:nn
```

leaves \long in the input stream. If the *token* is not a macro then \scan\_stop: is left in the input stream.

**TeXhackers note:** The prefix can be empty, \long, \protected or \protected\long with backslash replaced by the current escape character.

---

\cs\_parameter\_spec:N ∗ \cs\_parameter\_spec:N *token*

New: 2022-06-24

If the *token* is a macro, this function leaves the primitive TeX parameter specification in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1 y #2 }
\cs_parameter_spec:N \next:nn
```

leaves #1#2 in the input stream. If the *token* is not a macro then \scan\_stop: is left in the input stream.

**TeXhackers note:** If the parameter specification contains the string ->, then the function produces incorrect results.

---

```
\cs_replacement_spec:N ∗ \cs_replacement_spec:N ⟨token⟩
```

New: 2019-02-27

If the ⟨token⟩ is a macro, this function leaves the replacement text in input stream as a string of character tokens of category code 12 (with spaces having category code 10). Thus for example

```
\cs_set:Npn \next:nn #1#2 { x #1~y #2 }
\cs_replacement_spec:N \next:nn
```

leaves x#1~y#2 in the input stream. If the ⟨token⟩ is not a macro then \scan\_stop: is left in the input stream.

**TeXhackers note:** If the parameter specification contains the string →, then the function produces incorrect results.

## 4.5 Using or removing tokens and arguments

Tokens in the input can be read and used or read and discarded. If one or more tokens are wrapped in braces then when absorbing them the outer set is removed. At the same time, the category code of each token is set when the token is read by a function (if it is read more than once, the category code is determined by the situation in force when first function absorbs the token).

---

```
\use:n   ∗ \use:n    {⟨group1⟩}
\use:nn   ∗ \use:nn   {⟨group1⟩} {⟨group2⟩}
\use:nnn  ∗ \use:nnn  {⟨group1⟩} {⟨group2⟩} {⟨group3⟩}
\use:nnnn ∗ \use:nnnn {⟨group1⟩} {⟨group2⟩} {⟨group3⟩} {⟨group4⟩}
```

As illustrated, these functions absorb between one and four arguments, as indicated by the argument specifier. The braces surrounding each argument are removed and the remaining tokens are left in the input stream. The category code of these tokens is also fixed by this process (if it has not already been by some other absorption). All of these functions require only a single expansion to operate, so that one expansion of

```
\use:nn { abc } { { def } }
```

results in the input stream containing

```
abc { def }
```

i.e. only the outer braces are removed.

**TeXhackers note:** The \use:n function is equivalent to L<sup>A</sup>T<sub>E</sub>X 2ε's \@firstofone.

---

`\use_i:nn` \* `\use_i:nn {⟨arg1⟩} {⟨arg2⟩}`

`\use_ii:nn` \* These functions absorb two arguments from the input stream. The function `\use_i:nn` discards the second argument, and leaves the content of the first argument in the input stream. `\use_ii:nn` discards the first argument and leaves the content of the second argument in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

**TeXhackers note:** These are equivalent to L<sup>A</sup>T<sub>E</sub>X 2<sub><</sub>'s `\@firstoftwo` and `\@secondoftwo`.

---

`\use_i:nnn` \* `\use_i:nnn {⟨arg1⟩} {⟨arg2⟩} {⟨arg3⟩}`

`\use_ii:nnn` \* `\use_iii:nnn` \* These functions absorb three arguments from the input stream. The function `\use_i:nnn` discards the second and third arguments, and leaves the content of the first argument in the input stream. `\use_ii:nnn` and `\use_iii:nnn` work similarly, leaving the content of second or third arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

---

`\use_i:nnnn` \* `\use_i:nnnn {⟨arg1⟩} {⟨arg2⟩} {⟨arg3⟩} {⟨arg4⟩}`

`\use_ii:nnnn` \* `\use_iii:nnnn` \* `\use_iv:nnnn` \* These functions absorb four arguments from the input stream. The function `\use_i:nnnn` discards the second, third and fourth arguments, and leaves the content of the first argument in the input stream. `\use_i:nnnn`, `\use_ii:nnnn` and `\use_iv:nnnn` work similarly, leaving the content of second, third or fourth arguments in the input stream, respectively. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the functions to take effect.

---

`\use_i_ii:nnn` \* `\use_i_ii:nnn {⟨arg1⟩} {⟨arg2⟩} {⟨arg3⟩}`

This function absorbs three arguments and leaves the content of the first and second in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect. An example:

```
\use_i_ii:nnn { abc } { { def } } { ghi }
```

results in the input stream containing

```
abc { def }
```

i.e. the outer braces are removed and the third group is removed.

---

`\use_ii_i:nn` \* `\use_ii_i:nn {⟨arg1⟩} {⟨arg2⟩}`

New: 2019-06-02 This function absorbs two arguments and leaves the content of the second and first in the input stream. The category code of these tokens is also fixed (if it has not already been by some other absorption). A single expansion is needed for the function to take effect.

---

\use_none:n	* \use_none:n {<group <sub>1</sub> >}
\use_none:nn	* These functions absorb between one and nine groups from the input stream, leaving nothing on the resulting input stream. These functions work after a single expansion.
\use_none:nmm	* One or more of the n arguments may be an unbraced single token ( <i>i.e.</i> an N argument).
\use_none:nmmn	*
\use_none:nmmnn	*
\use_none:nmmnnn	*
\use_none:nmmnnnn	*
\use_none:nmmnnnnn	*
\use_none:nmmnnnnnn	*

---

\use:e \* \use:e {<expandable tokens>}

New: 2018-06-18 Fully expands the <token list> in an x-type manner, *but* the function remains fully expandable, and parameter character (usually #) need not be doubled.

**TEXhackers note:** \use:e is a wrapper around the primitive \expanded where it is available: it requires two expansions to complete its action. When \expanded is not available this function is very slow.

---

\use:x	\use:x {<expandable tokens>}
--------	------------------------------

---

Updated: 2011-12-31 Fully expands the <expandable tokens> and inserts the result into the input stream at the current location. Any hash characters (#) in the argument must be doubled.

#### 4.5.1 Selecting tokens from delimited arguments

A different kind of function for selecting tokens from the token stream are those that use delimited arguments.

---

\use_none_delimit_by_q_nil:w	* \use_none_delimit_by_q_nil:w {balanced text} \q_nil
\use_none_delimit_by_q_stop:w	* \use_none_delimit_by_q_stop:w {balanced text} \q_stop
\use_none_delimit_by_q_recursion_stop:w	* \use_none_delimit_by_q_recursion_stop:w {balanced text} \q_recursion_stop

---

Absorb the <balanced text> from the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

---

\use_i_delimit_by_q_nil:nw	* \use_i_delimit_by_q_nil:nw {<inserted tokens>} {balanced text}
\use_i_delimit_by_q_stop:nw	* \q_nil
\use_i_delimit_by_q_recursion_stop:nw	* \use_i_delimit_by_q_stop:nw {<inserted tokens>} {balanced text} \q_stop
	\use_i_delimit_by_q_recursion_stop:nw {<inserted tokens>} {balanced text} \q_recursion_stop

---

Absorb the <balanced text> from the input stream delimited by the marker given in the function name, leaving <inserted tokens> in the input stream for further processing.

## 4.6 Predicates and conditionals

LATEX3 has three concepts for conditional flow processing:

**Branching conditionals** Functions that carry out a test and then execute, depending on its result, either the code supplied as the *<true code>* or the *<false code>*. These arguments are denoted with T and F, respectively. An example would be

```
\cs_if_free:cTF {abc} {\langle true code \rangle} {\langle false code \rangle}
```

a function that turns the first argument into a control sequence (since it's marked as c) then checks whether this control sequence is still free and then depending on the result carries out the code in the second argument (true case) or in the third argument (false case).

These type of functions are known as “conditionals”; whenever a TF function is defined it is usually accompanied by T and F functions as well. These are provided for convenience when the branch only needs to go a single way. Package writers are free to choose which types to define but the kernel definitions always provide all three versions.

Important to note is that these branching conditionals with *<true code>* and/or *<false code>* are always defined in a way that the code of the chosen alternative can operate on following tokens in the input stream.

These conditional functions may or may not be fully expandable, but if they are expandable they are accompanied by a “predicate” for the same test as described below.

**Predicates** “Predicates” are functions that return a special type of boolean value which can be tested by the boolean expression parser. All functions of this type are expandable and have names that end with \_p in the description part. For example,

```
\cs_if_free_p:N
```

would be a predicate function for the same type of test as the conditional described above. It would return “true” if its argument (a single token denoted by N) is still free for definition. It would be used in constructions like

```
\bool_if:nTF {
    \cs_if_free_p:N \l_tmpz_t1 || \cs_if_free_p:N \g_tmpz_t1
} {\langle true code \rangle} {\langle false code \rangle}
```

For each predicate defined, a “branching conditional” also exists that behaves like a conditional described above.

**Primitive conditionals** There is a third variety of conditional, which is the original concept used in plain TeX and L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub>. Their use is discouraged in expl3 (although still used in low-level definitions) because they are more fragile and in many cases require more expansion control (hence more code) than the two types of conditionals described above.

---

\c\_true\_bool Constants that represent **true** and **false**, respectively. Used to implement predicates.  
\c\_false\_bool

### 4.6.1 Tests on control sequences

---

`\cs_if_eq_p:NN` \* `\cs_if_eq_p:NN`  $\langle cs_1 \rangle \langle cs_2 \rangle$   
`\cs_if_eq:NNTF` \* `\cs_if_eq:NNTF`  $\langle cs_1 \rangle \langle cs_2 \rangle \{ \langle true code \rangle \} \{ \langle false code \rangle \}$

---

Compares the definition of two *(control sequences)* and is logically `true` if they are the same, *i.e.* if they have exactly the same definition when examined with `\cs_show:N`.

---

`\cs_if_exist_p:N` \* `\cs_if_exist_p:N` *(control sequence)*  
`\cs_if_exist_p:c` \* `\cs_if_exist:NTF` *(control sequence)*  $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$   
`\cs_if_exist:NTF` \* Tests whether the *(control sequence)* is currently defined (whether as a function or another  
`\cs_if_exist:cTF` \* control sequence type). Any definition of *(control sequence)* other than `\relax` evaluates as `true`.

---



---

`\cs_if_free_p:N` \* `\cs_if_free_p:N` *(control sequence)*  
`\cs_if_free_p:c` \* `\cs_if_free:NTF` *(control sequence)*  $\{ \langle true code \rangle \} \{ \langle false code \rangle \}$   
`\cs_if_free:NTF` \* Tests whether the *(control sequence)* is currently free to be defined. This test is `false` if  
`\cs_if_free:cTF` \* the *(control sequence)* currently exists (as defined by `\cs_if_exist:NTF`).

---

### 4.6.2 Primitive conditionals

The  $\varepsilon$ -TeX engine itself provides many different conditionals. Some expand whatever comes after them and others don't. Hence the names for these underlying functions often contains a `:w` part but higher level functions are often available. See for instance `\int_compare_p:nNn` which is a wrapper for `\if_int_compare:w`.

Certain conditionals deal with specific data types like boxes and fonts and are described there. The ones described below are either the universal conditionals or deal with control sequences. We prefix primitive conditionals with `\if_`.

---

`\if_true:` \* `\if_true: <true code>` `\else: <false code>` `\fi:`  
`\if_false:` \* `\if_false: <true code>` `\else: <false code>` `\fi:`  
`\else:` \* `\reverse_if:N` *(primitive conditional)*  
`\fi:` \* `\if_true:` always executes *(true code)*, while `\if_false:` always executes *(false code)*.  
`\reverse_if:N` \* `\reverse_if:N` reverses any two-way primitive conditional. `\else:` and `\fi:` delimit the branches of the conditional. The function `\or:` is documented in `\3int` and used in case switches.

**TeXhackers note:** These are equivalent to their corresponding TeX primitive conditionals; `\reverse_if:N` is  $\varepsilon$ -TeX's `\unless`.

---

`\if_meaning:w` \* `\if_meaning:w`  $\langle arg_1 \rangle \langle arg_2 \rangle$  `\else: <false code>` `\fi:`  
`\if_meaning:w` executes *(true code)* when  $\langle arg_1 \rangle$  and  $\langle arg_2 \rangle$  are the same, otherwise it executes *(false code)*.  $\langle arg_1 \rangle$  and  $\langle arg_2 \rangle$  could be functions, variables, tokens; in all cases the *unexpanded* definitions are compared.

**TeXhackers note:** This is TeX's `\ifx`.

---

```
\if:w      * \if:w <token1> <token2> <true code> \else: <false code> \fi:  
\if_charcode:w * \if_catcode:w <token1> <token2> <true code> \else: <false code> \fi:  
\if_catcode:w * These conditionals expand any following tokens until two unexpandable tokens are left.  
If you wish to prevent this expansion, prefix the token in question with \exp_not:N.  
\if_catcode:w tests if the category codes of the two tokens are the same whereas \if:w tests if the character codes are identical. \if_charcode:w is an alternative name for \if:w.
```

---

```
\if_cs_exist:N * \if_cs_exist:N <cs> <true code> \else: <false code> \fi:  
\if_cs_exist:w * \if_cs_exist:w <tokens> \cs_end: <true code> \else: <false code> \fi:
```

Check if *<cs>* appears in the hash table or if the control sequence that can be formed from *<tokens>* appears in the hash table. The latter function does not turn the control sequence in question into \scan\_stop:. This can be useful when dealing with control sequences which cannot be entered as a single token.

---

```
\if_mode_horizontal: * \if_mode_horizontal: <true code> \else: <false code> \fi:  
\if_mode_vertical: * Execute <true code> if currently in horizontal mode, otherwise execute <false code>. Sim-  
\if_mode_math:     * ilar for the other functions.  
\if_mode_inner:   *
```

## 4.7 Starting a paragraph

---

```
\mode_leave_vertical: \mode_leave_vertical:
```

---

New: 2017-07-04 Ensures that TeX is not in vertical (inter-paragraph) mode. In horizontal or math mode this command has no effect, in vertical mode it switches to horizontal mode, and inserts a box of width \parindent, followed by the \everypar token list.

**TeXhackers note:** This results in the contents of the \everypar token register being inserted, after \mode\_leave\_vertical: is complete. Notice that in contrast to the LATEX 2 $\epsilon$  \leavevmode approach, no box is used by the method implemented here.

## 4.8 Debugging support

---

```
\debug_on:n  \debug_on:n { <comma-separated list> }
\debug_off:n \debug_off:n { <comma-separated list> }
```

New: 2017-07-16 Turn on and off within a group various debugging code, some of which is also available  
Updated: 2017-08-02 as `expl3` load-time options. The items that can be used in the `<list>` are

- `check-declarations` that checks all `expl3` variables used were previously declared and that local/global variables (based on their name or on their first assignment) are only locally/globally assigned;
- `check-expressions` that checks integer, dimension, skip, and muskip expressions are not terminated prematurely;
- `deprecation` that makes soon-to-be-deprecated commands produce errors;
- `log-functions` that logs function definitions;
- `all` that does all of the above.

Providing these as switches rather than options allows testing code even if it relies on other packages: load all other packages, call `\debug_on:n`, and load the code that one is interested in testing. These functions can only be used in L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package mode loaded with `enable-debug` or another option implying it.

---

```
\debug_suspend: \debug_suspend: ... \debug_resume:
```

`\debug_resume:` Suppress (locally) errors and logging from `debug` commands, except for the `deprecation` errors or warnings. These pairs of commands can be nested. This can be used around pieces of code that are known to fail checks, if such failures should be ignored. See for instance `|3coffins`.

## Chapter 5

# The `\3exp` package Argument expansion

This module provides generic methods for expanding TeX arguments in a systematic manner. The functions in this module all have prefix `exp`.

Not all possible variations are implemented for every base function. Instead only those that are used within the L<sup>A</sup>T<sub>E</sub>X3 kernel or otherwise seem to be of general interest are implemented. Consult the module description to find out which functions are actually defined. The next section explains how to define missing variants.

### 5.1 Defining new variants

The definition of variant forms for base functions may be necessary when writing new functions or when applying a kernel function in a situation that we haven't thought of before.

Internally preprocessing of arguments is done with functions of the form `\exp_...` . They all look alike, an example would be `\exp_args:NNo`. This function has three arguments, the first and the second are a single tokens, while the third argument should be given in braces. Applying `\exp_args:NNo` expands the content of third argument once before any expansion of the first and second arguments. If `\seq_gpush:N` was not defined it could be coded in the following way:

```
\exp_args:NNo \seq_gpush:Nn
  \g_file_name_stack
  { \l_tmpa_t1 }
```

In other words, the first argument to `\exp_args:NNo` is the base function and the other arguments are preprocessed and then passed to this base function. In the example the first argument to the base function should be a single token which is left unchanged while the second argument is expanded once. From this example we can also see how the variants are defined. They just expand into the appropriate `\exp_` function followed by the desired base function, *e.g.*

```
\cs_generate_variant:Nn \seq_gpush:Nn { No }
```

results in the definition of `\seq_gpush:N`

```
\cs_new:Npn \seq_gpush:No { \exp_args:NNo \seq_gpush:Nn }
```

Providing variants in this way in style files is safe as the `\cs_generate_variant:Nn` function will only create new definitions if there is not already one available. Therefore adding such definition to later releases of the kernel will not make such style files obsolete.

The steps above may be automated by using the function `\cs_generate_variant:Nn`, described next.

## 5.2 Methods for defining variants

We recall the set of available argument specifiers.

- `N` is used for single-token arguments while `c` constructs a control sequence from its name and passes it to a parent function as an `N`-type argument.
- Many argument types extract or expand some tokens and provide it as an `n`-type argument, namely a braced multiple-token argument: `V` extracts the value of a variable, `v` extracts the value from the name of a variable, `n` uses the argument as it is, `o` expands once, `f` expands fully the front of the token list, `e` and `x` expand fully all tokens (differences are explained later).
- A few odd argument types remain: `T` and `F` for conditional processing, otherwise identical to `n`-type arguments, `p` for the parameter text in definitions, `w` for arguments with a specific syntax, and `D` to denote primitives that should not be used directly.

---

```
\cs_generate_variant:Nn \cs_generate_variant:Nn <parent control sequence> {<variant argument specifiers>}
```

---

Updated: 2017-11-28

This function is used to define argument-specifier variants of the *<parent control sequence>* for L<sup>A</sup>T<sub>E</sub>X3 code-level macros. The *<parent control sequence>* is first separated into the *<base name>* and *<original argument specifier>*. The comma-separated list of *<variant argument specifiers>* is then used to define variants of the *<original argument specifier>* if these are not already defined. For each *<variant>* given, a function is created that expands its arguments as detailed and passes them to the *<parent control sequence>*. So for example

```
\cs_set:Npn \foo:Nn #1#2 { code here }
\cs_generate_variant:Nn \foo:Nn { c }
```

creates a new function `\foo:c` which expands its first argument into a control sequence name and passes the result to `\foo:Nn`. Similarly

```
\cs_generate_variant:Nn \foo:Nn { NV , cV }
```

generates the functions `\foo:NV` and `\foo:cV` in the same way. The `\cs_generate_variant:Nn` function can only be applied if the *<parent control sequence>* is already defined. If the *<parent control sequence>* is protected or if the *<variant>* involves any `x` argument, then the *<variant control sequence>* is also protected. The *<variant>* is created globally, as is any `\exp_args:N<variant>` function needed to carry out the expansion.

Only `n` and `N` arguments can be changed to other types. The only allowed changes are

- `c` variant of an `N` parent;
- `o`, `V`, `v`, `f`, `e`, or `x` variant of an `n` parent;
- `N`, `n`, `T`, `F`, or `p` argument unchanged.

This means the *<parent>* of a *<variant>* form is always unambiguous, even in cases where both an `n`-type parent and an `N`-type parent exist, such as for `\tl_count:n` and `\tl_count:N`.

For backward compatibility it is currently possible to make `n`, `o`, `V`, `v`, `f`, `e`, or `x`-type variants of an `N`-type argument or `N` or `c`-type variants of an `n`-type argument. Both are deprecated. The first because passing more than one token to an `N`-type argument will typically break the parent function's code. The second because programmers who use that most often want to access the value of a variable given its name, hence should use a `V`-type or `v`-type variant instead of `c`-type. In those cases, using the lower-level `\exp_args:No` or `\exp_args:Nc` functions explicitly is preferred to defining confusing variants.

### 5.3 Introducing the variants

The `V` type returns the value of a register, which can be one of `t1`, `clist`, `int`, `skip`, `dim`, `muskip`, or built-in T<sub>E</sub>X registers. The `v` type is the same except it first creates a control sequence out of its argument before returning the value.

In general, the programmer should not need to be concerned with expansion control. When simply using the content of a variable, functions with a `V` specifier should be used. For those referred to by (cs)name, the `v` specifier is available for the same purpose. Only

when specific expansion steps are needed, such as when using delimited arguments, should the lower-level functions with `o` specifiers be employed.

The `e` type expands all tokens fully, starting from the first. More precisely the expansion is identical to that of TeX's `\message` (in particular `#` needs not be doubled). It was added in May 2018. In recent enough engines (starting around 2019) it relies on the primitive `\expanded` hence is fast. In older engines it is very much slower. As a result it should only be used in performance critical code if typical users will have a recent installation of the TeX ecosystem.

The `x` type expands all tokens fully, starting from the first. In contrast to `e`, all macro parameter characters `#` must be doubled, and omitting this leads to low-level errors. In addition this type of expansion is not expandable, namely functions that have `x` in their signature do not themselves expand when appearing inside `x` or `e` expansion.

The `f` type is so special that it deserves an example. It is typically used in contexts where only expandable commands are allowed. Then `x`-expansion cannot be used, and `f`-expansion provides an alternative that expands the front of the token list as much as can be done in such contexts. For instance, say that we want to evaluate the integer expression `3 + 4` and pass the result `7` as an argument to an expandable function `\example:n`. For this, one should define a variant using `\cs_generate_variant:Nn \example:n { f }`, then do

```
\example:f { \int_eval:n { 3 + 4 } }
```

Note that `x`-expansion would also expand `\int_eval:n` fully to its result `7`, but the variant `\example:x` cannot be expandable. Note also that `o`-expansion would not expand `\int_eval:n` fully to its result since that function requires several expansions. Besides the fact that `x`-expansion is protected rather than expandable, another difference between `f`-expansion and `x`-expansion is that `f`-expansion expands tokens from the beginning and stops as soon as a non-expandable token is encountered, while `x`-expansion continues expanding further tokens. Thus, for instance

```
\example:f { \int_eval:n { 1 + 2 } , \int_eval:n { 3 + 4 } }
```

results in the call

```
\example:n { 3 , \int_eval:n { 3 + 4 } }
```

while using `\example:x` or `\example:e` instead results in

```
\example:n { 3 , 7 }
```

at the cost of being protected (for `x` type) or very much slower in old engines (for `e` type). If you use `f` type expansion in conditional processing then you should stick to using TF type functions only as the expansion does not finish any `\if... \fi:` itself!

It is important to note that both `f`- and `o`-type expansion are concerned with the expansion of tokens from left to right in their arguments. In particular, `o`-type expansion applies to the first *token* in the argument it receives: it is conceptually similar to

```
\exp_after:wN <base function> \exp_after:wN { <argument> }
```

At the same time, `f`-type expansion stops at the *first* non-expandable token. This means for example that both

```
\tl_set:No \l_tmpa_tl { \g_tmpb_tl }
```

and

```
\tl_set:Nf \l_tmpa_tl { { \g_tmpb_tl } }
```

leave `\g_tmpb_tl` unchanged: `{` is the first token in the argument and is non-expandable. It is usually best to keep the following in mind when using variant forms.

- Variants with `x`-type arguments (that are fully expanded before being passed to the `n`-type base function) are never expandable even when the base function is. Such variants cannot work correctly in arguments that are themselves subject to expansion. Consider using `f` or `e` expansion.
- In contrast, `e` expansion (full expansion, almost like `x` except for the treatment of `#`) does not prevent variants from being expandable (if the base function is). The drawback is that `e` expansion is very much slower in old engines (before 2019). Consider using `f` expansion if that type of expansion is sufficient to perform the required expansion, or `x` expansion if the variant will not itself need to be expandable.
- Finally `f` expansion only expands the front of the token list, stopping at the first non-expandable token. This may fail to fully expand the argument.

When speed is essential (for functions that do very little work and whose variants are used numerous times in a document) the following considerations apply because internal functions for argument expansion come in two flavours, some faster than others.

- Arguments that might need expansion should come first in the list of arguments.
- Arguments that should consist of single tokens `N`, `c`, `V`, or `v` should come first among these.
- Arguments that appear after the first multi-token argument `n`, `f`, `e`, or `o` require slightly slower special processing to be expanded. Therefore it is best to use the optimized functions, namely those that contain only `N`, `c`, `V`, and `v`, and, in the last position, `o`, `f`, `e`, with possible trailing `N` or `n` or `T` or `F`, which are not expanded. Any `x`-type argument causes slightly slower processing.

## 5.4 Manipulating the first argument

These functions are described in detail: expansion of multiple tokens follows the same rules but is described in a shorter fashion.

---

```
\exp_args:Nc * \exp_args:Nc <function> {<tokens>}
```

`\exp_args:cc *` This function absorbs two arguments (the `<function>` name and the `<tokens>`). The `<tokens>` are expanded until only characters remain, and are then turned into a control sequence. The result is inserted into the input stream *after* reinsertion of the `<function>`. Thus the `<function>` may take more than one argument: all others are left unchanged.

The `:cc` variant constructs the `<function>` name in the same manner as described for the `<tokens>`.

---

`\exp_args:N`  $\star$  `\exp_args:N`  $\langle function \rangle \{ \langle tokens \rangle \} \dots$

This function absorbs two arguments (the  $\langle function \rangle$  name and the  $\langle tokens \rangle$ ). The  $\langle tokens \rangle$  are expanded once, and the result is inserted in braces into the input stream *after* reinsertion of the  $\langle function \rangle$ . Thus the  $\langle function \rangle$  may take more than one argument: all others are left unchanged.

---

`\exp_args:NV`  $\star$  `\exp_args:NV`  $\langle function \rangle \langle variable \rangle$

This function absorbs two arguments (the names of the  $\langle function \rangle$  and the  $\langle variable \rangle$ ). The content of the  $\langle variable \rangle$  are recovered and placed inside braces into the input stream *after* reinsertion of the  $\langle function \rangle$ . Thus the  $\langle function \rangle$  may take more than one argument: all others are left unchanged.

---

`\exp_args:Nv`  $\star$  `\exp_args:Nv`  $\langle function \rangle \{ \langle tokens \rangle \}$

This function absorbs two arguments (the  $\langle function \rangle$  name and the  $\langle tokens \rangle$ ). The  $\langle tokens \rangle$  are expanded until only characters remain, and are then turned into a control sequence. This control sequence should be the name of a  $\langle variable \rangle$ . The content of the  $\langle variable \rangle$  are recovered and placed inside braces into the input stream *after* reinsertion of the  $\langle function \rangle$ . Thus the  $\langle function \rangle$  may take more than one argument: all others are left unchanged.

---

`\exp_args:Ne`  $\star$  `\exp_args:Ne`  $\langle function \rangle \{ \langle tokens \rangle \}$

New: 2018-05-15 This function absorbs two arguments (the  $\langle function \rangle$  name and the  $\langle tokens \rangle$ ) and exhaustively expands the  $\langle tokens \rangle$ . The result is inserted in braces into the input stream *after* reinsertion of the  $\langle function \rangle$ . Thus the  $\langle function \rangle$  may take more than one argument: all others are left unchanged.

**TeXhackers note:** This relies on the `\expanded` primitive when available (in LuaTeX and starting around 2019 in other engines). Otherwise it uses some fall-back code that is very much slower. As a result it should only be used in performance-critical code if typical users have a recent installation of the TeX ecosystem.

---

`\exp_args:Nf`  $\star$  `\exp_args:Nf`  $\langle function \rangle \{ \langle tokens \rangle \}$

This function absorbs two arguments (the  $\langle function \rangle$  name and the  $\langle tokens \rangle$ ). The  $\langle tokens \rangle$  are fully expanded until the first non-expandable token is found (if that is a space it is removed), and the result is inserted in braces into the input stream *after* reinsertion of the  $\langle function \rangle$ . Thus the  $\langle function \rangle$  may take more than one argument: all others are left unchanged.

---

`\exp_args:Nx` `\exp_args:Nx`  $\langle function \rangle \{ \langle tokens \rangle \}$

This function absorbs two arguments (the  $\langle function \rangle$  name and the  $\langle tokens \rangle$ ) and exhaustively expands the  $\langle tokens \rangle$ . The result is inserted in braces into the input stream *after* reinsertion of the  $\langle function \rangle$ . Thus the  $\langle function \rangle$  may take more than one argument: all others are left unchanged.

## 5.5 Manipulating two arguments

---

```
\exp_args:Nnc * \exp_args:Nnc <token1> <token2> {<tokens>}
\exp_args:Nno * These optimized functions absorb three arguments and expand the second and third as
\exp_args:NNv * detailed by their argument specifier. The first argument of the function is then the next
\exp_args:NNv * item on the input stream, followed by the expansion of the second and third arguments.
\exp_args:Nne *
```

```
\exp_args:NNf *
\exp_args:Ncc *
\exp_args:Nco *
\exp_args:Ncv *
\exp_args:Ncv *
\exp_args:Ncf *
\exp_args:NVV *
```

---

Updated: 2018-05-15

---

```
\exp_args:Nnc * \exp_args:Noo <token> {<tokens1>} {<tokens2>}
\exp_args:Nno * These functions absorb three arguments and expand the second and third as detailed by
\exp_args:NnV * their argument specifier. The first argument of the function is then the next item on
\exp_args:Nnv * the input stream, followed by the expansion of the second and third arguments. These
\exp_args:Nne * functions need slower processing.
\exp_args:Nnf *
```

```
\exp_args:Noc *
\exp_args:Noo *
\exp_args:Nof *
\exp_args:Nvo *
\exp_args:Nfo *
\exp_args:Nff *
\exp_args:Nee *
```

---

Updated: 2018-05-15

---

```
\exp_args:NNx \exp_args:NNx <token1> <token2> {<tokens>}
\exp_args:Ncx *
\exp_args:Nnx *
\exp_args:Nox *
\exp_args:Nxo *
\exp_args:Nxx *
```

These functions absorb three arguments and expand the second and third as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments. These functions are not expandable due to their x-type argument.

## 5.6 Manipulating three arguments

---

```
\exp_args>NNNo * \exp_args>NNNNo <token1> <token2> <token3> {<tokens>}
\exp_args:NNNV * These optimized functions absorb four arguments and expand the second, third and
\exp_args:NNNV * fourth as detailed by their argument specifier. The first argument of the function is then
\exp_args:Nccc * the next item on the input stream, followed by the expansion of the second argument,
\exp_args:NcNc * etc.
\exp_args:NcNo *
\exp_args:Ncco *
```

---

```
\exp_args:NNcf * \exp_args:NNoo <token1> <token2> {<token3>} {<tokens>}
\exp_args:NNno * These functions absorb four arguments and expand the second, third and fourth as de-
\exp_args:NNnV * tailed by their argument specifier. The first argument of the function is then the next
\exp_args:NNoo * item on the input stream, followed by the expansion of the second argument, etc. These
\exp_args:NNVV * functions need slower processing.
\exp_args:Ncno *
```

```
\exp_args:NcnV *
\exp_args:Ncoo *
\exp_args:NcVV *
\exp_args:NnnC *
\exp_args:Nnno *
\exp_args:Nnnf *
\exp_args:Nnff *
\exp_args:Nooo *
\exp_args:Noof *
\exp_args:Nffo *
\exp_args:Neee *
```

---

```
\exp_args:NNNx \exp_args:NNnx <token1> <token2> {<tokens1>} {<tokens2>}
```

```
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox
```

These functions absorb four arguments and expand the second, third and fourth as detailed by their argument specifier. The first argument of the function is then the next item on the input stream, followed by the expansion of the second argument, *etc.*

---

New: 2015-08-12

---

## 5.7 Unbraced expansion

---

```
\exp_last_unbraced:No * \exp_last_unbraced:Nno <token> {<tokens1>} {<tokens2>}
\exp_last_unbraced:NV *
\exp_last_unbraced:Nv *
\exp_last_unbraced:Ne *
\exp_last_unbraced:Nf *
\exp_last_unbraced:NNo *
\exp_last_unbraced:NNV *
\exp_last_unbraced:NNF *
\exp_last_unbraced:Nco *
\exp_last_unbraced:Ncv *
\exp_last_unbraced:Nno *
\exp_last_unbraced:Noo *
\exp_last_unbraced:Nfo *
\exp_last_unbraced:NNNNo *
\exp_last_unbraced:NNNV *
\exp_last_unbraced:NNNF *
\exp_last_unbraced:NnNo *
\exp_last_unbraced:NNNNNo *
\exp_last_unbraced:NNNNf *
```

\* These functions absorb the number of arguments given by their specification, carry out the expansion indicated and leave the results in the input stream, with the last argument not surrounded by the usual braces. Of these, the :Nno, :Noo, :Nfo and :NnNo variants need slower processing.

\* **TEXhackers note:** As an optimization, the last argument is unbraced by some of those functions before expansion. This can cause problems if the argument is empty: for instance, \*\exp\_last\_unbraced:Nf \foo\_bar:w {} \q\_stop leads to an infinite loop, as the quark is f-expanded.

---

Updated: 2018-05-15

---

---

```
\exp_last_unbraced:Nx \exp_last_unbraced:Nx <function> {<tokens>}
```

This function fully expands the  $\langle tokens \rangle$  and leaves the result in the input stream after reinsertion of the  $\langle function \rangle$ . This function is not expandable.

---

```
\exp_last_two_unbraced:Noo * \exp_last_two_unbraced:Noo <token> {<tokens1>} {<tokens2>}
```

This function absorbs three arguments and expands the second and third once. The first argument of the function is then the next item on the input stream, followed by the expansion of the second and third arguments, which are not wrapped in braces. This function needs special (slower) processing.

---

```
\exp_after:wN * \exp_after:wN <token1> <token2>
```

Carries out a single expansion of  $\langle token_2 \rangle$  (which may consume arguments) prior to the expansion of  $\langle token_1 \rangle$ . If  $\langle token_2 \rangle$  has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that  $\langle token_1 \rangle$  may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal TeX category codes). Unless specifically required this should be avoided: expansion should be carried out using an appropriate argument specifier variant or the appropriate `\exp_arg:N` function.

**TeXhackers note:** This is the TeX primitive `\expandafter` renamed.

## 5.8 Preventing expansion

Despite the fact that the following functions are all about preventing expansion, they're designed to be used in an expandable context and hence are all marked as being 'expandable' since they themselves disappear after the expansion has completed.

---

```
\exp_not:N * \exp_not:N <token>
```

Prevents expansion of the  $\langle token \rangle$  in a context where it would otherwise be expanded, for example an `x`-type argument or the first token in an `o` or `e` or `f` argument.

**TeXhackers note:** This is the TeX `\noexpand` primitive. It only prevents expansion. At the beginning of an `f`-type argument, a space  $\langle token \rangle$  is removed even if it appears as `\exp_not:N c_space_token`. In an `x`-expanding definition (`\cs_new:Npx`), a macro parameter introduces an argument even if it appears as `\exp_not:N # 1`. This differs from `\exp_not:n`.

---

```
\exp_not:c * \exp_not:c {<tokens>}
```

Expands the  $\langle tokens \rangle$  until only characters remain, and then converts this into a control sequence. Further expansion of this control sequence is then inhibited using `\exp_not:N`.

---

`\exp_not:n * \exp_not:n {⟨tokens⟩}`

Prevents expansion of the  $⟨tokens⟩$  in an **e** or **x**-type argument. In all other cases the  $⟨tokens⟩$  continue to be expanded, for example in the input stream or in other types of arguments such as **c**, **f**, **v**. The argument of `\exp_not:n` *must* be surrounded by braces.

**TeXhackers note:** This is the  $\varepsilon$ -**TeX** `\unexpanded` primitive. In an **x**-expanding definition (`\cs_new:Npx`), `\exp_not:n #1` is equivalent to `##1` rather than to `#1`, namely it inserts the two characters `#` and `1`. In an **e**-type argument `\exp_not:n #` is equivalent to `#`, namely it inserts the character `#`.

---

`\exp_not:o * \exp_not:o {⟨tokens⟩}`

Expands the  $⟨tokens⟩$  once, then prevents any further expansion in **x**-type or **e**-type arguments using `\exp_not:n`.

---

`\exp_not:V * \exp_not:V {⟨variable⟩}`

Recovers the content of the  $⟨variable⟩$ , then prevents expansion of this material in **x**-type or **e**-type arguments using `\exp_not:n`.

---

`\exp_not:v * \exp_not:v {⟨tokens⟩}`

Expands the  $⟨tokens⟩$  until only characters remains, and then converts this into a control sequence which should be a  $⟨variable⟩$  name. The content of the  $⟨variable⟩$  is recovered, and further expansion in **x**-type or **e**-type arguments is prevented using `\exp_not:n`.

---

`\exp_not:e * \exp_not:e {⟨tokens⟩}`

Expands  $⟨tokens⟩$  exhaustively, then protects the result of the expansion (including any tokens which were not expanded) from further expansion in **e** or **x**-type arguments using `\exp_not:n`. This is very rarely useful but is provided for consistency.

---

`\exp_not:f * \exp_not:f {⟨tokens⟩}`

Expands  $⟨tokens⟩$  fully until the first unexpandable token is found (if it is a space it is removed). Expansion then stops, and the result of the expansion (including any tokens which were not expanded) is protected from further expansion in **x**-type or **e**-type arguments using `\exp_not:n`.

---

`\exp_stop_f: * \foo_bar:f { ⟨tokens⟩ \exp_stop_f: {⟨more tokens⟩} }`

Updated: 2011-06-03 This function terminates an **f**-type expansion. Thus if a function `\foo_bar:f` starts an **f**-type expansion and all of  $⟨tokens⟩$  are expandable `\exp_stop_f:` terminates the expansion of tokens even if  $⟨more tokens⟩$  are also expandable. The function itself is an implicit space token. Inside an **x**-type or **e**-type expansion, it retains its form, but when typeset it produces the underlying space ( $\text{\_}$ ).

## 5.9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of **TeX** expansion from the programmer by providing concepts that evaluate/expand arguments of functions prior to

calling the “base” functions. Thus, instead of using many `\expandafter` calls and other trickery it is usually a matter of choosing the right variant of a function to achieve a desired result.

Of course, deep down TeX is using expansion as always and there are cases where a programmer needs to control that expansion directly; typical situations are basic data manipulation tools. This section documents the functions for that level. These commands are used throughout the kernel code, but we hope that outside the kernel there will be little need to resort to them. Instead the argument manipulation methods document above should usually be sufficient.

While `\exp_after:wN` expands one token (out of order) it is sometimes necessary to expand several tokens in one go. The next set of commands provide this functionality. Be aware that it is absolutely required that the programmer has full control over the tokens to be expanded, i.e., it is not possible to use these functions to expand unknown input as part of *<expandable-tokens>* as that will break badly if unexpandable tokens are encountered in that place!

---

`\exp:w` \* `\exp:w <expandable tokens> \exp_end:`

`\exp_end:` \*  
Expands *<expandable-tokens>* until reaching `\exp_end:` at which point expansion stops.  
The full expansion of *<expandable tokens>* has to be empty. If any token in *<expandable tokens>* or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end:` will be misinterpreted later on.<sup>4</sup>

In typical use cases the `\exp_end:` is hidden somewhere in the replacement text of *<expandable-tokens>* rather than being on the same expansion level than `\exp:w`, e.g., you may see code such as

```
\exp:w \@@_case:NnTF #1 {#2} {} {}
```

where somewhere during the expansion of `\@@_case:NnTF` the `\exp_end:` gets generated.

**TeXhackers note:** The current implementation uses `\romannumeral` hence ignores space tokens and explicit signs + and - in the expansion of the *<expandable tokens>*, but this should not be relied upon.

---

<sup>4</sup>Due to the implementation you might get the character in position 0 in the current font (typically ‘‘’’) in the output without any error message!

---

`\exp:w` \* `\exp:w <expandable-tokens> \exp_end_continue_f:w <further-tokens>`  
`\exp_end_continue_f:w` \* Expands *<expandable-tokens>* until reaching `\exp_end_continue_f:w` at which point expansion continues as an f-type expansion expanding *<further-tokens>* until an unexpandable token is encountered (or the f-type expansion is explicitly terminated by `\exp_stop_f:`). As with all f-type expansions a space ending the expansion gets removed.

---

New: 2015-08-23

The full expansion of *<expandable-tokens>* has to be empty. If any token in *<expandable-tokens>* or any token generated by expanding the tokens therein is not expandable the expansion will end prematurely and as a result `\exp_end_continue_f:w` will be misinterpreted later on.<sup>5</sup>

In typical use cases *<expandable-tokens>* contains no tokens at all, e.g., you will see code such as

```
\exp_after:wN { \exp:w \exp_end_continue_f:w #2 }
```

where the `\exp_after:wN` triggers an f-expansion of the tokens in `#2`. For technical reasons this has to happen using two tokens (if they would be hidden inside another command `\exp_after:wN` would only expand the command but not trigger any additional f-expansion).

You might wonder why there are two different approaches available, after all the effect of

```
\exp:w <expandable-tokens> \exp_end:
```

can be alternatively achieved through an f-type expansion by using `\exp_stop_f:`, i.e.

```
\exp:w \exp_end_continue_f:w <expandable-tokens> \exp_stop_f:
```

The reason is simply that the first approach is slightly faster (one less token to parse and less expansion internally) so in places where such performance really matters and where we want to explicitly stop the expansion at a defined point the first form is preferable.

---

`\exp:w` \* `\exp:w <expandable-tokens> \exp_end_continue_f:nw <further-tokens>`  
`\exp_end_continue_f:nw` \* The difference to `\exp_end_continue_f:w` is that we first we pick up an argument which is then returned to the input stream. If *<further-tokens>* starts with space tokens then these space tokens are removed while searching for the argument. If it starts with a brace group then the braces are removed. Thus such spaces or braces will not terminate the f-type expansion.

---

<sup>5</sup>In this particular case you may get a character into the output as well as an error message.

## 5.10 Internal functions

---

```
\:::n \cs_new:Npn \exp_args:Ncof { ::c ::o ::f ::: }
```

\:::N Internal forms for the base expansion types. These names do *not* conform to the general L<sup>A</sup>T<sub>E</sub>X3 approach as this makes them more readily visible in the log and so forth. They \:::c should not be used outside this module.

\:::o

\:::e

\:::f

\:::x

\:::v

\:::V

\:::

---

```
\:::o_unbraced \cs_new:Npn \exp_last_unbraced:Nno { ::n ::o_unbraced ::: }
```

\:::e\_unbraced Internal forms for the expansion types which leave the terminal argument unbraced.  
\:::f\_unbraced These names do *not* conform to the general L<sup>A</sup>T<sub>E</sub>X3 approach as this makes them more readily visible in the log and so forth. They should not be used outside this module.  
\:::x\_unbraced  
\:::v\_unbraced  
\:::V\_unbraced

# Chapter 6

## The `I3sort` package

### Sorting functions

#### 6.1 Controlling sorting

L<sup>A</sup>T<sub>E</sub>X3 comes with a facility to sort list variables (sequences, token lists, or comma-lists) according to some user-defined comparison. For instance,

```
\clist_set:Nn \l_foo_clist { 3 , 01 , -2 , 5 , +1 }
\clist_sort:Nn \l_foo_clist
{
    \int_compare:nNnTF { #1 } > { #2 }
        { \sort_return_swapped: }
        { \sort_return_same: }
}
```

results in `\l_foo_clist` holding the values { -2 , 01 , +1 , 3 , 5 } sorted in non-decreasing order.

The code defining the comparison should call `\sort_return_swapped:` if the two items given as `#1` and `#2` are not in the correct order, and otherwise it should call `\sort_return_same:` to indicate that the order of this pair of items should not be changed.

For instance, a `<comparison code>` consisting only of `\sort_return_same:` with no test yields a trivial sort: the final order is identical to the original order. Conversely, using a `<comparison code>` consisting only of `\sort_return_swapped:` reverses the list (in a fairly inefficient way).

**TeXhackers note:** The current implementation is limited to sorting approximately 20000 items (40000 in LuaT<sub>E</sub>X), depending on what other packages are loaded.

Internally, the code from `I3sort` stores items in `\toks` registers allocated locally. Thus, the `<comparison code>` should not call `\newtoks` or other commands that allocate new `\toks` registers. On the other hand, altering the value of a previously allocated `\toks` register is not a problem.

---

```
\sort_return_same: \seq_sort:Nn seq var
\sort_return_swapped: { ... \sort_return_same: or \sort_return_swapped: ... }
```

---

New: 2017-02-06 Indicates whether to keep the order or swap the order of two items that are compared in the sorting code. Only one of the `\sort_return_...` functions should be used by the code, according to the results of some tests on the items #1 and #2 to be compared.

# Chapter 7

## The `\tl_analysis` package: Analysing token lists

This module provides functions that are particularly useful in the `\l3regex` module for mapping through a token list one `\langle token \rangle` at a time (including begin-group/end-group tokens). For `\tl_analysis_map_inline:Nn` or `\tl_analysis_map_inline:nn`, the token list is given as an argument; the analogous function `\peek_analysis_map_inline:n` documented in `\l3token` finds tokens in the input stream instead. In both cases the user provides `\langle inline code \rangle` that receives three arguments for each `\langle token \rangle`:

- `\langle tokens \rangle`, which both o-expand and x-expand to the `\langle token \rangle`. The detailed form of `\langle tokens \rangle` may change in later releases.
- `\langle char code \rangle`, a decimal representation of the character code of the `\langle token \rangle`, -1 if it is a control sequence.
- `\langle catcode \rangle`, a capital hexadecimal digit which denotes the category code of the `\langle token \rangle` (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing "`\langle catcode \rangle`".

In addition, there is a debugging function `\tl_analysis_show:n`, very similar to the `\ShowTokens` macro from the `ted` package.

---

`\tl_analysis_show:N` `\tl_analysis_show:n` `\{<token list>\}`

`\tl_analysis_show:n` `\tl_analysis_log:n` `\{<token list>\}`

`\tl_analysis_log:N`

`\tl_analysis_log:n`

New: 2021-05-11

Displays to the terminal (or log) the detailed decomposition of the `\langle token list \rangle` into tokens, showing the category code of each character token, the meaning of control sequences and active characters, and the value of registers.

---

`\tl_analysis_map_inline:nn` `\tl_analysis_map_inline:nn` `\{<token list>\} {<inline function>}`

`\tl_analysis_map_inline:Nn`

New: 2018-04-09

Updated: 2022-03-26

Applies the `\langle inline function \rangle` to each individual `\langle token \rangle` in the `\langle token list \rangle`. The `\langle inline function \rangle` receives three arguments as explained above. As all other mappings the mapping is done at the current group level, i.e. any local assignments made by the `\langle inline function \rangle` remain in effect after the loop.

## Chapter 8

# The `\l3regex` package: Regular expressions in `TEX`

The `\l3regex` package provides regular expression testing, extraction of submatches, splitting, and replacement, all acting on token lists. The syntax of regular expressions is mostly a subset of the PCRE syntax (and very close to POSIX), with some additions due to the fact that `TEX` manipulates tokens rather than characters. For performance reasons, only a limited set of features are implemented. Notably, back-references are not supported.

Let us give a few examples. After

```
\tl_set:Nn \l_my_tl { That~cat. }
\regex_replace_once:nnN { at } { is } \l_my_tl
```

the token list variable `\l_my_tl` holds the text “This cat.”, where the first occurrence of “at” was replaced by “is”. A more complicated example is a pattern to emphasize each word and add a comma after it:

```
\regex_replace_all:nnN { \w+ } { \c{emph}\cB{\0 }\cE\, } \l_my_tl
```

The `\w` sequence represents any “word” character, and `+` indicates that the `\w` sequence should be repeated as many times as possible (at least once), hence matching a word in the input token list. In the replacement text, `\0` denotes the full match (here, a word). The command `\emph` is inserted using `\c{emph}`, and its argument `\0` is put between braces `\cB\{` and `\cE\}`.

If a regular expression is to be used several times, it can be compiled once, and stored in a regex variable using `\regex_set:Nn`. For example,

```
\regex_new:N \l_foo_regex
\regex_set:Nn \l_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\l_foo_regex` a regular expression which matches the starting marker for an environment: `\begin`, followed by a begin-group token (`\cB.`), then any number of tokens which are neither begin-group nor end-group character tokens (`\c[^BE].*`), ending with an end-group token (`\cE.`). As explained in the next section, the parentheses “capture” the result of `\c[^BE].*`, giving us access to the name of the environment when doing replacements.

## 8.1 Syntax of regular expressions

### 8.1.1 Regular expression examples

We start with a few examples, and encourage the reader to apply `\regex_show:n` to these regular expressions.

- `Cat` matches the word “Cat” capitalized in this way, but also matches the beginning of the word “Cattle”: use `\bCat\b` to match a complete word only.
- `[abc]` matches one letter among “a”, “b”, “c”; the pattern `(a|b|c)` matches the same three possible letters (but see the discussion of submatches below).
- `[A-Za-z]*` matches any number (due to the quantifier `*`) of Latin letters (not accented).
- `\c{[A-Za-z]*}` matches a control sequence made of Latin letters.
- `\_[^\_]*\_` matches an underscore, any number of characters other than underscore, and another underscore; it is equivalent to `\_.*?\_` where `.` matches arbitrary characters and the lazy quantifier `*?` means to match as few characters as possible, thus avoiding matching underscores.
- `[\+\\-]?\d+` matches an explicit integer with at most one sign.
- `[\+\\-\\_]*\d+\_*` matches an explicit integer with any number of `+` and `-` signs, with spaces allowed except within the mantissa, and surrounded by spaces.
- `[\+\\-\\_]*(\d+|\d*\.\d+)\_*` matches an explicit integer or decimal number; using `[.,]` instead of `\.` would allow the comma as a decimal marker.
- `[\+\\-\\_]*(\d+|\d*\.\d+)\_*((?i)pt|in|[cm]m|ex|[bs]p|[dn]d|[pcn]c)\_*` matches an explicit dimension with any unit that `TeX` knows, where `(?i)` means to treat lowercase and uppercase letters identically.
- `[\+\\-\\_]*((?i)nan|inf|(\d+|\d*\.\d+)(\_*e[\+\\-\\_]*\d+)?)\_*` matches an explicit floating point number or the special values `nan` and `inf` (with signs and spaces allowed).
- `[\+\\-\\_]*(\d+|\c{C.})\_*` matches an explicit integer or control sequence (without checking whether it is an integer variable).
- `\G.*?\K` at the beginning of a regular expression matches and discards (due to `\K`) everything between the end of the previous match (`\G`) and what is matched by the rest of the regular expression; this is useful in `\regex_replace_all:nnN` when the goal is to extract matches or submatches in a finer way than with `\regex_extract_all:nnN`.

While it is impossible for a regular expression to match only integer expressions, `[\+\\-\\_]*\d+\_*(\[\+\\-*\]/[\+\\-\\_]*\d+\_*)*` matches among other things all valid integer expressions (made only with explicit integers). One should follow it with further testing.

### 8.1.2 Characters in regular expressions

Most characters match exactly themselves, with an arbitrary category code. Some characters are special and must be escaped with a backslash (*e.g.*, `\*` matches a star character). Some escape sequences of the form backslash–letter also have a special meaning (for instance `\d` matches any digit). As a rule,

- every alphanumeric character (`A–Z`, `a–z`, `0–9`) matches exactly itself, and should not be escaped, because `\A`, `\B`, ... have special meanings;
- non-alphanumeric printable ascii characters can (and should) always be escaped: many of them have special meanings (*e.g.*, use `\(`, `\)`, `\?`, `\.`, `\^`);
- spaces should always be escaped (even in character classes);
- any other character may be escaped or not, without any effect: both versions match exactly that character.

Note that these rules play nicely with the fact that many non-alphanumeric characters are difficult to input into TeX under normal category codes. For instance, `\abc\%` matches the characters `\abc%` (with arbitrary category codes), but does not match the control sequence `\abc` followed by a percent character. Matching control sequences can be done using the `\c{\langle regex\rangle}` syntax (see below).

Any special character which appears at a place where its special behaviour cannot apply matches itself instead (for instance, a quantifier appearing at the beginning of a string), after raising a warning.

Characters.

`\x{hh...}` Character with hex code `hh...`

`\xhh` Character with hex code `hh`.

`\a` Alarm (hex 07).

`\e` Escape (hex 1B).

`\f` Form-feed (hex 0C).

`\n` New line (hex 0A).

`\r` Carriage return (hex 0D).

`\t` Horizontal tab (hex 09).

### 8.1.3 Characters classes

Character properties.

. A single period matches any token.

`\d` Any decimal digit.

`\h` Any horizontal space character, equivalent to `[\ \^\^I]`: space and tab.

`\s` Any space character, equivalent to `[\ \^\^I\^\^J\^\^L\^\^M]`.

\v Any vertical space character, equivalent to `[\^J\^K\^L\^M]`. Note that `\^K` is a vertical space, but not a space, for compatibility with Perl.

\w Any word character, *i.e.*, alphanumerics and underscore, equivalent to the explicit class `[A-Za-z0-9\_]`.

\D Any token not matched by \d.

\H Any token not matched by \h.

\N Any token other than the \n character (hex 0A).

\S Any token not matched by \s.

\V Any token not matched by \v.

\W Any token not matched by \w.

Of those, ., \D, \H, \N, \S, \V, and \W match arbitrary control sequences.

Character classes match exactly one token in the subject.

[...] Positive character class. Matches any of the specified tokens.

[^...] Negative character class. Matches any token other than the specified characters.

x-y Within a character class, this denotes a range (can be used with escaped characters).

[:(*name*):] Within a character class (one more set of brackets), this denotes the POSIX character class *name*, which can be `alnum`, `alpha`, `ascii`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `word`, or `xdigit`.

[:^(*name*):] Negative POSIX character class.

For instance, `[a-oq-z\cC.]` matches any lowercase latin letter except p, as well as control sequences (see below for a description of \c).

In character classes, only [, ^, -, ], \ and spaces are special, and should be escaped. Other non-alphanumeric characters can still be escaped without harm. Any escape sequence which matches a single character (\d, \D, etc.) is supported in character classes. If the first character is ^, then the meaning of the character class is inverted; ^ appearing anywhere else in the range is not special. If the first character (possibly following a leading ^) is ] then it does not need to be escaped since ending the range there would make it empty. Ranges of characters can be expressed using -, for instance, [\D 0-5] and [^6-9] are equivalent.

#### 8.1.4 Structure: alternatives, groups, repetitions

Quantifiers (repetition).

? 0 or 1, greedy.

?? 0 or 1, lazy.

\* 0 or more, greedy.

\*? 0 or more, lazy.

+ 1 or more, greedy.

$+?$  1 or more, lazy.

$\{n\}$  Exactly  $n$ .

$\{n,\}$   $n$  or more, greedy.

$\{n,\}?$   $n$  or more, lazy.

$\{n,m\}$  At least  $n$ , no more than  $m$ , greedy.

$\{n,m\}?$  At least  $n$ , no more than  $m$ , lazy.

For greedy quantifiers the regex code will first investigate matches that involve as many repetitions as possible, while for lazy quantifiers it investigates matches with as few repetitions as possible first.

Alternation and capturing groups.

$A|B|C$  Either one of A, B, or C, investigating A first.

$(\dots)$  Capturing group.

$(?:\dots)$  Non-capturing group.

$(?| \dots)$  Non-capturing group which resets the group number for capturing groups in each alternative. The following group is numbered with the first unused group number.

Capturing groups are a means of extracting information about the match. Parenthesized groups are labelled in the order of their opening parenthesis, starting at 1. The contents of those groups corresponding to the “best” match (leftmost longest) can be extracted and stored in a sequence of token lists using for instance `\regex_extract_once:nnNTF`.

The `\K` escape sequence resets the beginning of the match to the current position in the token list. This only affects what is reported as the full match. For instance,

```
\regex_extract_all:nnN { a \K . } { a123aaxyz } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{1}` and `{a}`: the true matches are `{a1}` and `{aa}`, but they are trimmed by the use of `\K`. The `\K` command does not affect capturing groups: for instance,

```
\regex_extract_once:nnN { (. \K c)+ \d } { acbc3 } \l_foo_seq
```

results in `\l_foo_seq` containing the items `{c3}` and `{bc}`: the true match is `{acbc3}`, with first submatch `{bc}`, but `\K` resets the beginning of the match to the last position where it appears.

### 8.1.5 Matching exact tokens

The `\c` escape sequence allows to test the category code of tokens, and match control sequences. Each character category is represented by a single uppercase letter:

- `C` for control sequences;
- `B` for begin-group tokens;
- `E` for end-group tokens;

- M for math shift;
- T for alignment tab tokens;
- P for macro parameter tokens;
- U for superscript tokens (up);
- D for subscript tokens (down);
- S for spaces;
- L for letters;
- O for others; and
- A for active characters.

The `\c` escape sequence is used as follows.

`\c{<regex>}` A control sequence whose csname matches the `<regex>`, anchored at the beginning and end, so that `\c{begin}` matches exactly `\begin`, and nothing else.

`\cX` Applies to the next object, which can be a character, escape character sequence such as `\x{0A}`, character class, or group, and forces this object to only match tokens with category X (any of CBEMTPUDSLOA). For instance, `\cL[A-Z\d]` matches uppercase letters and digits of category code letter, `\cC.` matches any control sequence, and `\cO(abc)` matches abc where each character has category other.<sup>6</sup>

`\c[XYZ]` Applies to the next object, and forces it to only match tokens with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[LSO](..)` matches two tokens of category letter, space, or other.

`\c[^XYZ]` Applies to the next object and prevents it from matching any token with category X, Y, or Z (each being any of CBEMTPUDSLOA). For instance, `\c[^O]\d` matches digits which have any category different from other.

The category code tests can be used inside classes; for instance, `[\cO\d \c[LO][A-F]]` matches what TeX considers as hexadecimal digits, namely digits with category other, or uppercase letters from A to F with category either letter or other. Within a group affected by a category code test, the outer test can be overridden by a nested test: for instance, `\cL(ab\cO\*cd)` matches ab\*cd where all characters are of category letter, except \* which has category other.

The `\u` escape sequence allows to insert the contents of a token list directly into a regular expression or a replacement, avoiding the need to escape special characters. Namely, `\u{<var name>}` matches the exact contents (both character codes and category codes) of the variable `\<var name>`, which are obtained by applying `\exp_not:v {<var name>}` at the time the regular expression is compiled. Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\t1_to_str:v`. Quantifiers are supported.

The `\ur` escape sequence allows to insert the contents of a `regex` variable into a larger regular expression. For instance, `A\ur{l_tmpa_regex}D` matches the tokens A and

---

<sup>6</sup>This last example also captures “abc” as a regex group; to avoid this use a non-capturing group `\cO(?:abc)`.

D separated by something that matches the regular expression `\l_tmpa_regex`. This behaves as if a non-capturing group were surrounding `\l_tmpa_regex`, and any group contained in `\l_tmpa_regex` is converted to a non-capturing group. Quantifiers are supported.

For instance, if `\l_tmpa_regex` has value `B|C`, then `A\ur{\l_tmpa_regex}D` is equivalent to `A(?:B|C)D` (matching `ABD` or `ACD`) and not to `AB|CD` (matching `AB` or `CD`). To get the latter effect, it is simplest to use TeX's expansion machinery directly: if `\l_mymodule_BC_t1` contains `B|C` then the following two lines show the same result:

```
\regex_show:n { A \u{l_mymodule_BC_t1} D }
\regex_show:n { A B | C D }
```

### 8.1.6 Miscellaneous

Anchors and simple assertions.

`\b` Word boundary: either the previous token is matched by `\w` and the next by `\W`, or the opposite. For this purpose, the ends of the token list are considered as `\W`.

`\B` Not a word boundary: between two `\w` tokens or two `\W` tokens (including the boundary).

`^` or `\A` Start of the subject token list.

`$`, `\Z` or `\z` End of the subject token list.

`\G` Start of the current match. This is only different from `^` in the case of multiple matches: for instance `\regex_count:nnN { \G a } { aaba } \l_tmpa_int` yields 2, but replacing `\G` by `^` would result in `\l_tmpa_int` holding the value 1.

The option `(?i)` makes the match case insensitive (treating `A-Z` and `a-z` as equivalent, with no support yet for Unicode case changing). This applies until the end of the group in which it appears, and can be reverted using `(?-i)`. For instance, in `(?i)(a(?-i)b|c)d`, the letters `a` and `d` are affected by the `i` option. Characters within ranges and classes are affected individually: `(?i)[\?-B]` is equivalent to `[\?@ABab]` (and differs from the much larger class `[\?-b]`), and `(?i)[^aeiou]` matches any character which is not a vowel. The `i` option has no effect on `\c{...}`, on `\u{...}`, on character properties, or on character classes, for instance it has no effect at all in `(?i)\u{l_foo_t1}\d\d[:lower:]`.

## 8.2 Syntax of the replacement text

Most of the features described in regular expressions do not make sense within the replacement text. Backslash introduces various special constructions, described further below:

- `\0` is the whole match;
- `\1` is the submatch that was matched by the first (capturing) group `(...)`; similarly for `\2, ..., \9` and `\g{<number>}`;
- `\u` inserts a space (spaces are ignored when not escaped);

- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\x{hhh}` correspond to single characters as in regular expressions;
- `\c{<cs name>}` inserts a control sequence;
- `\c<category><character>` (see below);
- `\u{<tl var name>}` inserts the contents of the `<tl var>` (see below).

Characters other than backslash and space are simply inserted in the result (but since the replacement text is first converted to a string, one should also escape characters that are special for T<sub>E</sub>X, for instance use `\#`). Non-alphanumeric characters can always be safely escaped with a backslash.

For instance,

```
\tl_set:Nn \l_my_t1 { Hello,\~world! }
\regex_replace_all:nnN { ([er]?\l{o}) . } { (\O--\l{1}) } \l_my_t1
```

results in `\l_my_t1` holding `H(e1l--el)(o,--o) w(or--o)(1d--1)!`

The submatches are numbered according to the order in which the opening parenthesis of capturing groups appear in the regular expression to match. The *n*-th submatch is empty if there are fewer than *n* capturing groups or for capturing groups that appear in alternatives that were not used for the match. In case a capturing group matches several times during a match (due to quantifiers) only the last match is used in the replacement text. Submatches always keep the same category codes as in the original token list.

By default, the category code of characters inserted by the replacement are determined by the prevailing category code regime at the time where the replacement is made, with two exceptions:

- space characters (with character code 32) inserted with `\u` or `\x20` or `\x{20}` have category code 10 regardless of the prevailing category code regime;
- if the category code would be 0 (escape), 5 (newline), 9 (ignore), 14 (comment) or 15 (invalid), it is replaced by 12 (other) instead.

The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\cX(...)` Produces the characters “...” with category *X*, which must be one of CBEMTPUDSLOA as in regular expressions. Parentheses are optional for a single character (which can be an escape sequence). When nested, the innermost category code applies, for instance `\cL(Hello\cS\ world)!` gives this text with standard category codes.

`\c{<text>}` Produces the control sequence with csname *<text>*. The *<text>* may contain references to the submatches `\0`, `\1`, and so on, as in the example for `\u` below.

The escape sequence `\u{<var name>}` allows to insert the contents of the variable with name *<var name>* directly into the replacement, giving an easier control of category codes. When nested in `\c{...}` and `\u{...}` constructions, the `\u` and `\c` escape sequences perform `\tl_to_str:v`, namely extract the value of the control sequence and turn it into a string. Matches can also be used within the arguments of `\c` and `\u`. For instance,

```
\tl_set:Nn \l_my_one_t1 { first }
\tl_set:Nn \l_my_two_t1 { \emph{second} }
\tl_set:Nn \l_my_t1 { one , two , one , one }
\regex_replace_all:nnN { [^,]+ } { \u{\l_my_\l{0}_t1} } \l_my_t1
```

results in `\l_my_t1` holding `first,\emph{second},first,first`.

Regex replacement is also a convenient way to produce token lists with arbitrary category codes. For instance

```
\tl_clear:N \l_tmpa_tl  
\regex_replace_all:nnN { } { \cU\% \cA\~ } \l_tmpa_tl
```

results in `\l_tmpa_tl` containing the percent character with category code 7 (superscript) and an active tilde character.

## 8.3 Pre-compiling regular expressions

If a regular expression is to be used several times, it is better to compile it once rather than doing it each time the regular expression is used. The compiled regular expression is stored in a variable. All of the `\l3regex` module's functions can be given their regular expression argument either as an explicit string or as a compiled regular expression.

---

```
\regex_new:N \regex_new:N <regex var>
```

New: 2017-05-26 Creates a new `<regex var>` or raises an error if the name is already taken. The declaration is global. The `<regex var>` is initially such that it never matches.

---

```
\regex_set:Nn \regex_set:Nn <regex var> {<regex>}
```

New: 2017-05-26 Stores a compiled version of the `<regular expression>` in the `<regex var>`. The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. For instance, this function can be used as

```
\regex_new:N \l_my_regex  
\regex_set:Nn \l_my_regex { my\ (simple\ )? reg(ex|ular\ expression) }
```

---

```
\regex_const:Nn \regex_const:Nn <regex var> {<regex>}
```

New: 2017-05-26 Creates a new constant `<regex var>` or raises an error if the name is already taken. The value of the `<regex var>` is set globally to the compiled version of the `<regular expression>`.

---

```
\regex_show:N \regex_show:n {<regex>}  
\regex_show:n \regex_log:n {<regex>}
```

New: 2021-04-26 Updated: 2021-04-29 Displays in the terminal or writes in the log file (respectively) how `\l3regex` interprets the `<regex>`. For instance, `\regex_show:n {\A X|Y}` shows

```
+--branch  
    anchor at start (\A)  
    char code 88 (X)  
+--branch  
    char code 89 (Y)
```

indicating that the anchor `\A` only applies to the first branch: the second branch is not anchored to the beginning of the match.

## 8.4 Matching

All regular expression functions are available in both :n and :N variants. The former require a “standard” regular expression, while the later require a compiled expression as generated by \regex\_set:Nn.

---

\regex\_match:nnTF \regex\_match:nnTF {⟨regular expression⟩} {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}  
\regex\_match:NnTF

New: 2017-05-26

Tests whether the ⟨regular expression⟩ matches any part of the ⟨token list⟩. For instance,

```
\regex_match:nnTF { b [cde]* } { abecdex } { TRUE } { FALSE }
\regex_match:nnTF { [b-dq-w] } { example } { TRUE } { FALSE }
```

leaves TRUE then FALSE in the input stream.

---

\regex\_count:nnN \regex\_count:nnN {⟨regular expression⟩} {⟨token list⟩} <int var>

New: 2017-05-26

Sets <int var> within the current T<sub>E</sub>X group level equal to the number of times ⟨regular expression⟩ appears in ⟨token list⟩. The search starts by finding the left-most longest match, respecting greedy and lazy (non-greedy) operators. Then the search starts again from the character following the last character of the previous match, until reaching the end of the token list. Infinite loops are prevented in the case where the regular expression can match an empty token list: then we count one match between each pair of characters. For instance,

```
\int_new:N \l_foo_int
\regex_count:nnN { (b+|c) } { abbababcbb } \l_foo_int
```

results in \l\_foo\_int taking the value 5.

---

\regex\_match\_case:nn \regex\_match\_case:nnTF

New: 2022-01-10

```
\regex_match_case:nnTF
{
  {⟨regex1⟩} {⟨code case1⟩}
  {⟨regex2⟩} {⟨code case2⟩}
  ...
  {⟨regexn⟩} {⟨code casen⟩}
} {⟨token list⟩}
{⟨true code⟩} {⟨false code⟩}
```

Determines which of the ⟨regular expressions⟩ matches at the earliest point in the ⟨token list⟩, and leaves the corresponding ⟨code<sub>i</sub>⟩ followed by the ⟨true code⟩ in the input stream. If several ⟨regex⟩ match starting at the same point, then the first one in the list is selected and the others are discarded. If none of the ⟨regex⟩ match, the ⟨false code⟩ is left in the input stream. Each ⟨regex⟩ can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the ⟨token list⟩, each of the ⟨regex⟩ is searched in turn. If one of them matches then the corresponding ⟨code⟩ is used and everything else is discarded, while if none of the ⟨regex⟩ match at a given position then the next starting position is attempted. If none of the ⟨regex⟩ match anywhere in the ⟨token list⟩ then nothing is left in the input stream. Note that this differs from nested \regex\_match:nnTF statements since all ⟨regex⟩ are attempted at each position rather than attempting to match ⟨regex<sub>1</sub>⟩ at every position before moving on to ⟨regex<sub>2</sub>⟩.

## 8.5 Submatch extraction

---

```
\regex_extract_once:nnN  \regex_extract_once:nnN {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩
\regex_extract_once:nnNTF \regex_extract_once:nnNTF {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩ {⟨true code⟩} {⟨false
\regex_extract_once:NnN   code⟩}
```

**\regex\_extract\_once:NnNTF** Finds the first match of the *⟨regular expression⟩* in the *⟨token list⟩*. If it exists, the match is stored as the first item of the *⟨seq var⟩*, and further items are the contents of capturing groups, in the order of their opening parenthesis. The *⟨seq var⟩* is assigned locally. If there is no match, the *⟨seq var⟩* is cleared. The testing versions insert the *⟨true code⟩* into the input stream if a match was found, and the *⟨false code⟩* otherwise.

For instance, assume that you type

```
\regex_extract_once:nnNTF { \A(La)?TeX(!*)\Z } { LaTeX!!! } \l_foo_seq
{ true } { false }
```

Then the regular expression (anchored at the start with `\A` and at the end with `\Z`) must match the whole token list. The first capturing group, `(La)?`, matches `La`, and the second capturing group, `(!*)`, matches `!!!`. Thus, `\l_foo_seq` contains as a result the items `{LaTeX!!!}`, `{La}`, and `{!!!}`, and the `true` branch is left in the input stream. Note that the *n*-th item of `\l_foo_seq`, as obtained using `\seq_item:Nn`, correspond to the submatch numbered *(n - 1)* in functions such as `\regex_replace_once:nnN`.

---

```
\regex_extract_all:nnN  \regex_extract_all:nnN {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩
\regex_extract_all:nnNTF \regex_extract_all:nnNTF {⟨regex⟩} {⟨token list⟩} ⟨seq var⟩ {⟨true code⟩} {⟨false
\regex_extract_all:NnN   code⟩}
```

**\regex\_extract\_all:NnNTF** Finds all matches of the *⟨regular expression⟩* in the *⟨token list⟩*, and stores all the submatch information in a single sequence (concatenating the results of multiple `\regex_extract_once:nnN` calls). The *⟨seq var⟩* is assigned locally. If there is no match, the *⟨seq var⟩* is cleared. The testing versions insert the *⟨true code⟩* into the input stream if a match was found, and the *⟨false code⟩* otherwise. For instance, assume that you type

```
\regex_extract_all:nnNTF { \w+ } { Hello,~world! } \l_foo_seq
{ true } { false }
```

Then the regular expression matches twice, the resulting sequence contains the two items `{Hello}` and `{world}`, and the `true` branch is left in the input stream.

---

`\regex_split:nnN`    \regex\_split:nnN {*regular expression*} {*token list*} {*seq var*}  
`\regex_split:nnNTF` \regex\_split:nnNTF {*regular expression*} {*token list*} {*seq var*} {*true code*}  
`\regex_split:Nnn`    {*false code*}  
`\regex_split:NnNTF` Splits the *token list* into a sequence of parts, delimited by matches of the *regular expression*. If the *regular expression* has capturing groups, then the token lists that they match are stored as items of the sequence as well. The assignment to *seq var* is local. If no match is found the resulting *seq var* has the *token list* as its sole item. If the *regular expression* matches the empty token list, then the *token list* is split into single tokens. The testing versions insert the *true code* into the input stream if a match was found, and the *false code* otherwise. For example, after  
 New: 2017-05-26

```
\seq_new:N \l_path_seq
\regex_split:nnNTF { / } { the/path/for/this/file.tex } \l_path_seq
{ true } { false }
```

the sequence `\l_path_seq` contains the items `{the}`, `{path}`, `{for}`, `{this}`, and `{file.tex}`, and the `true` branch is left in the input stream.

## 8.6 Replacement

---

`\regex_replace_once:nnN`    \regex\_replace\_once:nnN {*regular expression*} {*replacement*} {*tl var*}  
`\regex_replace_once:nnNTF` \regex\_replace\_once:nnNTF {*regular expression*} {*replacement*} {*tl var*} {*true code*} {*false code*}  
`\regex_replace_once:Nnn`  
`\regex_replace_once:NnNTF` Searches for the *regular expression* in the contents of the *tl var* and replaces the first match with the *replacement*. In the *replacement*, `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc*. The result is assigned locally to *tl var*.  
 New: 2017-05-26

---

`\regex_replace_all:nnN`    \regex\_replace\_all:nnN {*regular expression*} {*replacement*} {*tl var*}  
`\regex_replace_all:nnNTF` \regex\_replace\_all:nnNTF {*regular expression*} {*replacement*} {*tl var*} {*true code*} {*false code*}  
`\regex_replace_all:Nnn`  
`\regex_replace_all:NnNTF` Replaces all occurrences of the *regular expression* in the contents of the *tl var* by the *replacement*, where `\0` represents the full match, `\1` represent the contents of the first capturing group, `\2` of the second, *etc*. Every match is treated independently, and matches cannot overlap. The result is assigned locally to *tl var*.  
 New: 2017-05-26

---

```
\regex_replace_case_once:nN  \regex_replace_case_once:nNTF
\regex_replace_case_once:nNTF {
  New: 2022-01-10
  {<regex1>} {<replacement1>}
  {<regex2>} {<replacement2>}
  ...
  {<regexn>} {<replacementn>}
} <tl var>
{<true code>} {<false code>}
```

---

Replaces the earliest match of the regular expression  $(?|<regex_1>|\dots|<regex_n>)$  in the  $\langle token\ list\ variable\rangle$  by the  $\langle replacement\rangle$  corresponding to which  $\langle regex_i\rangle$  matched, then leaves the  $\langle true\ code\rangle$  in the input stream. If none of the  $\langle regex\rangle$  match, then the  $\langle tl\ var\rangle$  is not modified, and the  $\langle false\ code\rangle$  is left in the input stream. Each  $\langle regex\rangle$  can either be given as a regex variable or as an explicit regular expression.

In detail, for each starting position in the  $\langle token\ list\rangle$ , each of the  $\langle regex\rangle$  is searched in turn. If one of them matches then it is replaced by the corresponding  $\langle replacement\rangle$  as described for  $\backslash\text{regex\_replace\_once:nnN}$ . This is equivalent to checking with  $\backslash\text{regex\_match\_case:nn}$  which  $\langle regex\rangle$  matches, then performing the replacement with  $\backslash\text{regex\_replace\_once:nnN}$ .

---

```
\regex_replace_case_all:nN  \regex_replace_case_all:nNTF
\regex_replace_case_all:nNTF {
  New: 2022-01-10
  {<regex1>} {<replacement1>}
  {<regex2>} {<replacement2>}
  ...
  {<regexn>} {<replacementn>}
} <tl var>
{<true code>} {<false code>}
```

---

Replaces all occurrences of all  $\langle regex\rangle$  in the  $\langle token\ list\rangle$  by the corresponding  $\langle replacement\rangle$ . Every match is treated independently, and matches cannot overlap. The result is assigned locally to  $\langle tl\ var\rangle$ , and the  $\langle true\ code\rangle$  or  $\langle false\ code\rangle$  is left in the input stream depending on whether any replacement was made or not.

In detail, for each starting position in the  $\langle token\ list\rangle$ , each of the  $\langle regex\rangle$  is searched in turn. If one of them matches then it is replaced by the corresponding  $\langle replacement\rangle$ , and the search resumes at the position that follows this match (and replacement). For instance

```
\tl_set:Nn \l_tmpa_tl { Hello,~world! }
\regex_replace_case_all:nN
{
  { [A-Za-z]+ } { ``\0'' }
  { \b } { --- }
  { . } { [\0] }
} \l_tmpa_tl
```

results in  $\backslash\l_\text{tmpa}_\text{tl}$  having the contents ‘‘Hello’’---[,] [!] ‘‘world’’---[!]. Note in particular that the word-boundary assertion  $\backslash\b$  did not match at the start of words because the case  $[A-Za-z]^+$  matched at these positions. To change this, one could simply swap the order of the two cases in the argument of  $\backslash\text{regex\_replace\_case\_all:nn}$ .

## 8.7 Scratch regular expressions

---

`\l_tmpa_regex` Scratch regex for local assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_regex` Scratch regex for global assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 8.8 Bugs, misfeatures, future work, and other possibilities

The following need to be done now.

- Rewrite the documentation in a more ordered way, perhaps add a BNF?
  - Additional error-checking to come.
- Clean up the use of messages.
- Cleaner error reporting in the replacement phase.
- Add tracing information.
- Detect attempts to use back-references and other non-implemented syntax.
- Test for the maximum register `\c_max_register_int`.
- Find out whether the fact that `\W` and friends match the end-marker leads to bugs.  
Possibly update `\__regex_item_reverse:n`.
- The empty cs should be matched by `\c{}`, not by `\c{csname.?endcsname\s?}`.
- Code improvements to come.
  - Shift arrays so that the useful information starts at position 1.
  - Only build `\c{...}` once.
  - Use arrays for the left and right state stacks when compiling a regex.
  - Should `\__regex_action_free_group:n` only be used for greedy `{n,}` quantifier?  
(I think not.)
  - Quantifiers for `\u` and assertions.
  - When matching, keep track of an explicit stack of `curr_state` and `curr_submatches`.
  - If possible, when a state is reused by the same thread, kill other subthreads.

- Use an array rather than `\g__regex_balance_t1` to build the function `\__regex-replacement_balance_one_match:n`.
- Reduce the number of epsilon-transitions in alternatives.
- Optimize simple strings: use less states (`abcade` should give two states, for `abc` and `ade`). [Does that really make sense?]
- Optimize groups with no alternative.
- Optimize states with a single `\__regex_action_free:n`.
- Optimize the use of `\__regex_action_success`: by inserting it in state 2 directly instead of having an extra transition.
- Optimize the use of `\int_step_...` functions.
- Groups don't capture within regexes for csnames; optimize and document.
- Better "show" for anchors, properties, and catcode tests.
- Does `\K` really need a new state for itself?
- When compiling, use a boolean `in_cs` and less magic numbers.
- Instead of checking whether the character is special or alphanumeric using its character code, check if it is special in regexes with `\cs_if_exist` tests.

The following features are likely to be implemented at some point in the future.

- General look-ahead/behind assertions.
- Regex matching on external files.
- Conditional subpatterns with look ahead/behind: "if what follows is [...], then [...]".
- `(*..)` and `(?..)` sequences to set some options.
- UTF-8 mode for pdftEX.
- Newline conventions are not done. In particular, we should have an option for `.` not to match newlines. Also, `\A` should differ from `^`, and `\Z`, `\z` and `$` should differ.
- Unicode properties: `\p{..}` and `\P{..}`; `\X` which should match any "extended" Unicode sequence. This requires to manipulate a lot of data, probably using tree-boxes.

The following features of PCRE or Perl may or may not be implemented.

- Callout with `(?C...)` or other syntax: some internal code changes make that possible, and it can be useful for instance in the replacement code to stop a regex replacement when some marker has been found; this raises the question of a potential `\regex_break`: and then of playing well with `\t1_map_break`: called from within the code in a regex. It also raises the question of nested calls to the regex machinery, which is a problem since `\fontdimen` are global.

- Conditional subpatterns (other than with a look-ahead or look-behind condition): this is non-regular, isn't it?
- Named subpatterns:  $\text{\TeX}$  programmers have lived so far without any need for named macro parameters.

The following features of PCRE or Perl will definitely not be implemented.

- Back-references: non-regular feature, this requires backtracking, which is prohibitively slow.
- Recursion: this is a non-regular feature.
- Atomic grouping, possessive quantifiers: those tools, mostly meant to fix catastrophic backtracking, are unnecessary in a non-backtracking algorithm, and difficult to implement.
- Subroutine calls: this syntactic sugar is difficult to include in a non-backtracking algorithm, in particular because the corresponding group should be treated as atomic.
- Backtracking control verbs: intrinsically tied to backtracking.
- $\backslash\text{ddd}$ , matching the character with octal code ddd: we already have  $\backslash\text{x}\{\dots\}$  and the syntax is confusingly close to what we could have used for backreferences ( $\backslash 1$ ,  $\backslash 2$ , ...), making it harder to produce useful error message.
- $\backslash\text{cx}$ , similar to  $\text{\TeX}$ 's own  $\backslash\text{^^x}$ .
- Comments:  $\text{\TeX}$  already has its own system for comments.
- $\backslash\text{Q}\dots\backslash\text{E}$  escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- $\backslash\text{C}$  single byte in UTF-8 mode: Xe $\text{\TeX}$  and Lua $\text{\TeX}$  serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

# Chapter 9

## The `\l3prg` package Control structures

Conditional processing in L<sup>A</sup>T<sub>E</sub>X3 is defined as something that performs a series of tests, possibly involving assignments and calling other functions that do not read further ahead in the input stream. After processing the input, a *state* is returned. The states returned are `\true` and `\false`.

L<sup>A</sup>T<sub>E</sub>X3 has two forms of conditional flow processing based on these states. The first form is predicate functions that turn the returned state into a boolean `\true` or `\false`. For example, the function `\cs_if_free_p:N` checks whether the control sequence given as its argument is free and then returns the boolean `\true` or `\false` values to be used in testing with `\if_predicate:w` or in functions to be described below. The second form is the kind of functions choosing a particular argument from the input stream based on the result of the testing as in `\cs_if_free:NTF` which also takes one argument (the `N`) and then executes either `true` or `false` depending on the result.

**TEXhackers note:** The arguments are executed after exiting the underlying `\if... \fi:` structure.

### 9.1 Defining a set of conditional functions

---

```
\prg_new_conditional:Npnn \prg_new_conditional:Npnn \langle name\rangle:\langle arg spec\rangle \langle parameters\rangle {\langle conditions\rangle} {\langle code\rangle}
\prg_set_conditional:Npnn \prg_new_conditional:Nnn \langle name\rangle:\langle arg spec\rangle {\langle conditions\rangle} {\langle code\rangle}
\prg_gset_conditional:Npnn \prg_new_conditional:Nnn \langle name\rangle:\langle arg spec\rangle {\langle conditions\rangle} {\langle code\rangle}
```

---

These functions create a family of conditionals using the same `{⟨code⟩}` to perform the test created. Those conditionals are expandable if `⟨code⟩` is. The `new` versions check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the `set` versions do no check and perform assignments locally (*cf.* `\cs_set:Npn`). The conditionals created are dependent on the comma-separated list of `⟨conditions⟩`, which should be one or more of `p`, `T`, `F` and `TF`.

---

Updated: 2022-11-01

---

```
\prg_new_protected_conditional:Npnn \prg_new_protected_conditional:Npnn \langle name\rangle:\langle arg spec\rangle
\prg_set_protected_conditional:Npnn <parameters> {\{conditions\}} {\{code\}}
\prg_gset_protected_conditional:Npnn \prg_new_protected_conditional:Nnn \langle name\rangle:\langle arg spec\rangle
\prg_new_protected_conditional:Nnn {\{conditions\}} {\{code\}}
\prg_set_protected_conditional:Nnn
\prg_gset_protected_conditional:Nnn
```

---

Updated: 2012-02-06

---

These functions create a family of protected conditionals using the same  $\{\langle code \rangle\}$  to perform the test created. The  $\langle code \rangle$  does not need to be expandable. The **new** version check for existing definitions and perform assignments globally (*cf.* `\cs_new:Npn`) whereas the **set** version do not (*cf.* `\cs_set:Npn`). The conditionals created are depended on the comma-separated list of  $\langle conditions \rangle$ , which should be one or more of T, F and TF (not p).

The conditionals are defined by `\prg_new_conditional:Npnn` and friends as:

- $\langle name \rangle\_p:\langle arg spec \rangle$  — a predicate function which will supply either a logical **true** or logical **false**. This function is intended for use in cases where one or more logical tests are combined to lead to a final outcome. This function cannot be defined for **protected** conditionals.
- $\langle name \rangle:\langle arg spec \rangle T$  — a function with one more argument than the original  $\langle arg spec \rangle$  demands. The  $\langle true branch \rangle$  code in this additional argument will be left on the input stream only if the test is **true**.
- $\langle name \rangle:\langle arg spec \rangle F$  — a function with one more argument than the original  $\langle arg spec \rangle$  demands. The  $\langle false branch \rangle$  code in this additional argument will be left on the input stream only if the test is **false**.
- $\langle name \rangle:\langle arg spec \rangle TF$  — a function with two more arguments than the original  $\langle arg spec \rangle$  demands. The  $\langle true branch \rangle$  code in the first additional argument will be left on the input stream if the test is **true**, while the  $\langle false branch \rangle$  code in the second argument will be left on the input stream if the test is **false**.

The  $\langle code \rangle$  of the test may use  $\langle parameters \rangle$  as specified by the second argument to `\prg_set_conditional:Npnn`: this should match the  $\langle argument specification \rangle$  but this is not enforced. The Nnn versions infer the number of arguments from the argument specification given (*cf.* `\cs_new:Nn`, etc.). Within the  $\langle code \rangle$ , the functions `\prg_return_true:` and `\prg_return_false:` are used to indicate the logical outcomes of the test.

An example can easily clarify matters here:

```
\prg_set_conditional:Npnn \foo_if_bar:NN #1#2 { p , T , TF }
{
  \if_meaning:w \l_tmpa_tl #1
    \prg_return_true:
  \else:
    \if_meaning:w \l_tmpa_tl #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  \fi:
}
```

This defines the function `\foo_if_bar_p:N`, `\foo_if_bar:NNTF` and `\foo_if_bar:NNT` but not `\foo_if_bar:NNF` (because F is missing from the *(conditions)* list). The return statements take care of resolving the remaining `\else:` and `\fi:` before returning the state. There must be a return statement for each branch; failing to do so will result in erroneous output if that branch is executed.

---

```
\prg_new_eq_conditional:Nnn \prg_new_eq_conditional:Nnn \⟨name⟩:(arg spec) \⟨name⟩:(arg spec)
\prg_set_eq_conditional:Nnn {⟨conditions⟩}
```

---

These functions copy a family of conditionals. The `new` version checks for existing definitions (*cf.* `\cs_new_eq:N`) whereas the `set` version does not (*cf.* `\cs_set_eq:N`). The conditionals copied are depended on the comma-separated list of *(conditions)*, which should be one or more of p, T, F and TF.

---

```
\prg_return_true: * \prg_return_true:
\prg_return_false: * \prg_return_false:
```

---

These “return” functions define the logical state of a conditional statement. They appear within the code for a conditional function generated by `\prg_set_conditional:Npnn`, *etc*, to indicate when a true or false branch should be taken. While they may appear multiple times each within the code of such conditionals, the execution of the conditional must result in the expansion of one of these two functions *exactly once*.

The return functions trigger what is internally an f-expansion process to complete the evaluation of the conditional. Therefore, after `\prg_return_true:` or `\prg_return_false:` there must be no non-expandable material in the input stream for the remainder of the expansion of the conditional code. This includes other instances of either of these functions.

---

```
\prg_generate_conditional_variant:Nnn \prg_generate_conditional_variant:Nnn \⟨name⟩:(arg spec)
                                         {⟨variant argument specifiers⟩} {⟨condition specifiers⟩}
New: 2017-12-12
```

---

Defines argument-specifier variants of conditionals. This is equivalent to running `\cs_generate_variant:Nn` *⟨conditional⟩* {⟨variant argument specifiers⟩} on each *(conditional)* described by the *(condition specifiers)*. These base-form *(conditionals)* are obtained from the *(name)* and *(arg spec)* as described for `\prg_new_conditional:Npnn`, and they should be defined.

## 9.2 The boolean data type

This section describes a boolean data type which is closely connected to conditional processing as sometimes you want to execute some code depending on the value of a switch (*e.g.*, draft/final) and other times you perhaps want to use it as a predicate function in an `\if_predicate:w` test. The problem of the primitive `\if_false:` and `\if_true:` tokens is that it is not always safe to pass them around as they may interfere with scanning for termination of primitive conditional processing. Therefore, we employ two canonical booleans: `\c_true_bool` or `\c_false_bool`. Besides preventing problems as described above, it also allows us to implement a simple boolean parser supporting the logical operations And, Or, Not, *etc.* which can then be used on both the boolean type and predicate functions.

All conditional \bool\_ functions except assignments are expandable and expect the input to also be fully expandable (which generally means being constructed from predicate functions and booleans, possibly nested).

**TeXhackers note:** The `bool` data type is not implemented using the `\iffalse/\iftrue` primitives, in contrast to `\newif`, etc., in plain TeX, L<sup>A</sup>T<sub>E</sub>X 2<sub><</sub> and so on. Programmers should not base use of `bool` switches on any particular expectation of the implementation.

---

`\bool_new:N \bool_new:N <boolean>`

`\bool_new:c` Creates a new `<boolean>` or raises an error if the name is already taken. The declaration is global. The `<boolean>` is initially `false`.

---

`\bool_const:Nn \bool_const:Nn <boolean> {<boolexpr>}`

`\bool_const:cn` Creates a new constant `<boolean>` or raises an error if the name is already taken. The value of the `<boolean>` is set globally to the result of evaluating the `<boolexpr>`.  
New: 2017-11-28

---

`\bool_set_false:N \bool_set_false:N <boolean>`

`\bool_set_false:c` Sets `<boolean>` logically `false`.  
`\bool_gset_false:N`

---

`\bool_gset_false:c`

---

`\bool_set_true:N \bool_set_true:N <boolean>`

`\bool_set_true:c` Sets `<boolean>` logically `true`.  
`\bool_gset_true:N`

---

`\bool_gset_true:c`

---

`\bool_set_eq:NN` `\bool_set_eq:NN <boolean> <boolean>`

`\bool_set_eq:(cN|Nc|cc)` Sets `<boolean>` to the current value of `<boolean>`.  
`\bool_gset_eq:NN`

---

`\bool_gset_eq:(cN|Nc|cc)`

---

`\bool_set:Nn \bool_set:Nn <boolean> {<boolexpr>}`

`\bool_set:cn` Evaluates the `<boolean expression>` as described for `\bool_if:nTF`, and sets the `<boolean>` variable to the logical truth of this evaluation.  
`\bool_gset:Nn`

---

`\bool_gset:cn`

---

Updated: 2017-07-15

---

`\bool_if_p:N * \bool_if_p:N <boolean>`

`\bool_if_p:c * \bool_if:NTF <boolean> {<true code>} {<false code>}`

`\bool_if:NTF *` Tests the current truth of `<boolean>`, and continues expansion based on this result.  
`\bool_if:cTF *`

---

Updated: 2017-07-15

---

`\bool_to_str:N * \bool_to_str:N <boolean>`

`\bool_to_str:c * \bool_to_str:n <boolean expression>`

`\bool_to_str:n *` Expands to the letters `true` or `false` depending on the logical truth of the `<boolean>` or  
New: 2021-11-01 `<boolean expression>`.

<code>\bool_show:N</code>	<code>\bool_show:N &lt;boolean&gt;</code>
<code>\bool_show:c</code>	Displays the logical truth of the <i>&lt;boolean&gt;</i> on the terminal.
New: 2012-02-09	
Updated: 2021-04-29	
<code>\bool_show:n</code>	<code>\bool_show:n {&lt;boolean expression&gt;}</code>
New: 2012-02-09	Displays the logical truth of the <i>&lt;boolean expression&gt;</i> on the terminal.
Updated: 2017-07-15	
<code>\bool_log:N</code>	<code>\bool_log:N &lt;boolean&gt;</code>
<code>\bool_log:c</code>	Writes the logical truth of the <i>&lt;boolean&gt;</i> in the log file.
New: 2014-08-22	
Updated: 2021-04-29	
<code>\bool_log:n</code>	<code>\bool_log:n {&lt;boolean expression&gt;}</code>
New: 2014-08-22	Writes the logical truth of the <i>&lt;boolean expression&gt;</i> in the log file.
Updated: 2017-07-15	
<code>\bool_if_exist_p:N</code>	<code>\bool_if_exist_p:N &lt;boolean&gt;</code>
<code>\bool_if_exist_p:c</code>	<code>\bool_if_exist:cTF &lt;boolean&gt; {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\bool_if_exist:cTF</code>	Tests whether the <i>&lt;boolean&gt;</i> is currently defined. This does not check that the <i>&lt;boolean&gt;</i> really is a boolean variable.
New: 2012-03-03	

### 9.2.1 Scratch booleans

`\l_tmpa_bool` A scratch boolean for local assignment. It is never used by the kernel code, and so is safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

`\g_tmpa_bool` A scratch boolean for global assignment. It is never used by the kernel code, and so is safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, it may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 9.3 Boolean expressions

As we have a boolean datatype and predicate functions returning boolean *<true>* or *<false>* values, it seems only fitting that we also provide a parser for *<boolean expressions>*.

A boolean expression is an expression which given input in the form of predicate functions and boolean variables, return boolean *<true>* or *<false>*. It supports the logical operations And, Or and Not as the well-known infix operators `&&` and `||` and prefix `!` with their usual precedences (namely, `&&` binds more tightly than `||`). In addition to this, parentheses can be used to isolate sub-expressions. For example,

```
\int_compare_p:n { 1 = 1 } &&
(
  \int_compare_p:n { 2 = 3 } ||
  \int_compare_p:n { 4 <= 4 } ||
  \str_if_eq_p:nn { abc } { def }
) &&
! \int_compare_p:n { 2 = 4 }
```

is a valid boolean expression.

Contrarily to some other programming languages, the operators `&&` and `||` evaluate both operands in all cases, even when the first operand is enough to determine the result. This “eager” evaluation should be contrasted with the “lazy” evaluation of `\bool_lazy_...` functions.

**TeXhackers note:** The eager evaluation of boolean expressions is unfortunately necessary in TeX. Indeed, a lazy parser can get confused if `&&` or `||` or parentheses appear as (unbraced) arguments of some predicates. For instance, the innocuous-looking expression below would break (in a lazy parser) if `#1` were a closing parenthesis and `\l_tmpa_bool` were `true`.

```
( \l_tmpa_bool || \token_if_eq_meaning_p:NN X #1 )
```

Minimal (lazy) evaluation can be obtained using the conditionals `\bool_lazy_all:nTF`, `\bool_lazy_and:nnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:nnTF`, which only evaluate their boolean expression arguments when they are needed to determine the resulting truth value. For example, when evaluating the boolean expression

```
\bool_lazy_and_p:nn
{
  \bool_lazy_any_p:n
  {
    { \int_compare_p:n { 2 = 3 } }
    { \int_compare_p:n { 4 <= 4 } }
    { \int_compare_p:n { 1 = \error } } % skipped
  }
}
{ ! \int_compare_p:n { 2 = 4 } }
```

the line marked with `skipped` is not expanded because the result of `\bool_lazy_any_p:n` is known once the second boolean expression is found to be logically `true`. On the other hand, the last line is expanded because its logical value is needed to determine the result of `\bool_lazy_and_p:nn`.

---

`\bool_if_p:n * \bool_if_p:n {<boolean expression>}`  
`\bool_if:nTF * \bool_if:nTF {<boolean expression>} {<true code>} {<false code>}`

---

Updated: 2017-07-15 Tests the current truth of `<boolean expression>`, and continues expansion based on this result. The `<boolean expression>` should consist of a series of predicates or boolean variables with the logical relationship between these defined using `&&` (“And”), `||` (“Or”), `!` (“Not”) and parentheses. The logical Not applies to the next predicate or group.

<code>\bool_lazy_all_p:n</code>	<code>\bool_lazy_all_p:n { {(boolexpr<sub>1</sub>)} {(boolexpr<sub>2</sub>)} ... {(boolexpr<sub>N</sub>)} }</code>
<code>\bool_lazy_all:nTF</code>	<code>\bool_lazy_all:nTF { {(boolexpr<sub>1</sub>)} {(boolexpr<sub>2</sub>)} ... {(boolexpr<sub>N</sub>)} } {&lt;true code&gt;} {&lt;false code&gt;}</code>
New: 2015-11-15	
Updated: 2017-07-15	Implements the “And” operation on the <i>&lt;boolean expressions&gt;</i> , hence is <b>true</b> if all of them are <b>true</b> and <b>false</b> if any of them is <b>false</b> . Contrarily to the infix operator <b>&amp;&amp;</b> , only the <i>&lt;boolean expressions&gt;</i> which are needed to determine the result of <code>\bool_lazy_all:nTF</code> are evaluated. See also <code>\bool_lazy_and:nnTF</code> when there are only two <i>&lt;boolean expressions&gt;</i> .
<code>\bool_lazy_and_p:nn</code>	<code>\bool_lazy_and_p:nn {&lt;boolexpr<sub>1</sub>&gt;} {&lt;boolexpr<sub>2</sub>&gt;}</code>
<code>\bool_lazy_and:nnTF</code>	<code>\bool_lazy_and:nnTF {&lt;boolexpr<sub>1</sub>&gt;} {&lt;boolexpr<sub>2</sub>&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
New: 2015-11-15	
Updated: 2017-07-15	Implements the “And” operation between two boolean expressions, hence is <b>true</b> if both are <b>true</b> . Contrarily to the infix operator <b>&amp;&amp;</b> , the <i>&lt;boolexpr<sub>2</sub>&gt;</i> is only evaluated if it is needed to determine the result of <code>\bool_lazy_and:nnTF</code> . See also <code>\bool_lazy_all:nTF</code> when there are more than two <i>&lt;boolean expressions&gt;</i> .
<code>\bool_lazy_any_p:n</code>	<code>\bool_lazy_any_p:n { {(boolexpr<sub>1</sub>)} {(boolexpr<sub>2</sub>)} ... {(boolexpr<sub>N</sub>)} }</code>
<code>\bool_lazy_any:nTF</code>	<code>\bool_lazy_any:nTF { {(boolexpr<sub>1</sub>)} {(boolexpr<sub>2</sub>)} ... {(boolexpr<sub>N</sub>)} } {&lt;true code&gt;} {&lt;false code&gt;}</code>
New: 2015-11-15	
Updated: 2017-07-15	Implements the “Or” operation on the <i>&lt;boolean expressions&gt;</i> , hence is <b>true</b> if any of them is <b>true</b> and <b>false</b> if all of them are <b>false</b> . Contrarily to the infix operator <b>  </b> , only the <i>&lt;boolean expressions&gt;</i> which are needed to determine the result of <code>\bool_lazy_any:nTF</code> are evaluated. See also <code>\bool_lazy_or:nnTF</code> when there are only two <i>&lt;boolean expressions&gt;</i> .
<code>\bool_lazy_or_p:nn</code>	<code>\bool_lazy_or_p:nn {&lt;boolexpr<sub>1</sub>&gt;} {&lt;boolexpr<sub>2</sub>&gt;}</code>
<code>\bool_lazy_or:nnTF</code>	<code>\bool_lazy_or:nnTF {&lt;boolexpr<sub>1</sub>&gt;} {&lt;boolexpr<sub>2</sub>&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
New: 2015-11-15	
Updated: 2017-07-15	Implements the “Or” operation between two boolean expressions, hence is <b>true</b> if either one is <b>true</b> . Contrarily to the infix operator <b>  </b> , the <i>&lt;boolexpr<sub>2</sub>&gt;</i> is only evaluated if it is needed to determine the result of <code>\bool_lazy_or:nnTF</code> . See also <code>\bool_lazy_any:nTF</code> when there are more than two <i>&lt;boolean expressions&gt;</i> .
<code>\bool_not_p:n</code>	<code>\bool_not_p:n {&lt;boolean expression&gt;}</code>
Updated: 2017-07-15	Function version of <code>!(&lt;boolean expression&gt;)</code> within a boolean expression.
<code>\bool_xor_p:nn</code>	<code>\bool_xor_p:nn {&lt;boolexpr<sub>1</sub>&gt;} {&lt;boolexpr<sub>2</sub>&gt;}</code>
<code>\bool_xor:nnTF</code>	<code>\bool_xor:nnTF {&lt;boolexpr<sub>1</sub>&gt;} {&lt;boolexpr<sub>2</sub>&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
New: 2018-05-09	Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

## 9.4 Logical loops

Loops using either boolean expressions or stored boolean values.

<code>\bool_do_until:Nn</code>	<code>\bool_do_until:Nn &lt;boolean&gt; {&lt;code&gt;}</code>
<code>\bool_do_until:cn</code>	Places the <i>&lt;code&gt;</i> in the input stream for T <sub>E</sub> X to process, and then checks the logical value of the <i>&lt;boolean&gt;</i> . If it is <b>false</b> then the <i>&lt;code&gt;</i> is inserted into the input stream again and the process loops until the <i>&lt;boolean&gt;</i> is <b>true</b> .
Updated: 2017-07-15	

<code>\bool_do_while:Nn</code> ☆	<code>\bool_do_while:Nn &lt;boolean&gt; {&lt;code&gt;}</code>
<code>\bool_do_while:cn</code> ☆	Places the <code>&lt;code&gt;</code> in the input stream for T <sub>E</sub> X to process, and then checks the logical value of the <code>&lt;boolean&gt;</code> . If it is <code>true</code> then the <code>&lt;code&gt;</code> is inserted into the input stream again and the process loops until the <code>&lt;boolean&gt;</code> is <code>false</code> .
Updated: 2017-07-15	
<code>\bool_until_do:Nn</code> ☆	<code>\bool_until_do:Nn &lt;boolean&gt; {&lt;code&gt;}</code>
<code>\bool_until_do:cn</code> ☆	This function firsts checks the logical value of the <code>&lt;boolean&gt;</code> . If it is <code>false</code> the <code>&lt;code&gt;</code> is placed in the input stream and expanded. After the completion of the <code>&lt;code&gt;</code> the truth of the <code>&lt;boolean&gt;</code> is re-evaluated. The process then loops until the <code>&lt;boolean&gt;</code> is <code>true</code> .
Updated: 2017-07-15	
<code>\bool_while_do:Nn</code> ☆	<code>\bool_while_do:Nn &lt;boolean&gt; {&lt;code&gt;}</code>
<code>\bool_while_do:cn</code> ☆	This function firsts checks the logical value of the <code>&lt;boolean&gt;</code> . If it is <code>true</code> the <code>&lt;code&gt;</code> is placed in the input stream and expanded. After the completion of the <code>&lt;code&gt;</code> the truth of the <code>&lt;boolean&gt;</code> is re-evaluated. The process then loops until the <code>&lt;boolean&gt;</code> is <code>false</code> .
Updated: 2017-07-15	
<code>\bool_do_until:nn</code> ☆	<code>\bool_do_until:nn {&lt;boolean expression&gt;} {&lt;code&gt;}</code>
Updated: 2017-07-15	Places the <code>&lt;code&gt;</code> in the input stream for T <sub>E</sub> X to process, and then checks the logical value of the <code>&lt;boolean expression&gt;</code> as described for <code>\bool_if:nTF</code> . If it is <code>false</code> then the <code>&lt;code&gt;</code> is inserted into the input stream again and the process loops until the <code>&lt;boolean expression&gt;</code> evaluates to <code>true</code> .
<code>\bool_do_while:nn</code> ☆	<code>\bool_do_while:nn {&lt;boolean expression&gt;} {&lt;code&gt;}</code>
Updated: 2017-07-15	Places the <code>&lt;code&gt;</code> in the input stream for T <sub>E</sub> X to process, and then checks the logical value of the <code>&lt;boolean expression&gt;</code> as described for <code>\bool_if:nTF</code> . If it is <code>true</code> then the <code>&lt;code&gt;</code> is inserted into the input stream again and the process loops until the <code>&lt;boolean expression&gt;</code> evaluates to <code>false</code> .
<code>\bool_until_do:nn</code> ☆	<code>\bool_until_do:nn {&lt;boolean expression&gt;} {&lt;code&gt;}</code>
Updated: 2017-07-15	This function firsts checks the logical value of the <code>&lt;boolean expression&gt;</code> (as described for <code>\bool_if:nTF</code> ). If it is <code>false</code> the <code>&lt;code&gt;</code> is placed in the input stream and expanded. After the completion of the <code>&lt;code&gt;</code> the truth of the <code>&lt;boolean expression&gt;</code> is re-evaluated. The process then loops until the <code>&lt;boolean expression&gt;</code> is <code>true</code> .
<code>\bool_while_do:nn</code> ☆	<code>\bool_while_do:nn {&lt;boolean expression&gt;} {&lt;code&gt;}</code>
Updated: 2017-07-15	This function firsts checks the logical value of the <code>&lt;boolean expression&gt;</code> (as described for <code>\bool_if:nTF</code> ). If it is <code>true</code> the <code>&lt;code&gt;</code> is placed in the input stream and expanded. After the completion of the <code>&lt;code&gt;</code> the truth of the <code>&lt;boolean expression&gt;</code> is re-evaluated. The process then loops until the <code>&lt;boolean expression&gt;</code> is <code>false</code> .

## 9.5 Producing multiple copies

<code>\prg_replicate:nn</code> *	<code>\prg_replicate:nn {&lt;integer expression&gt;} {&lt;tkenes&gt;}</code>
Updated: 2011-07-04	Evaluates the <code>&lt;integer expression&gt;</code> (which should be zero or positive) and creates the resulting number of copies of the <code>&lt;tkenes&gt;</code> . The function is both expandable and safe for nesting. It yields its result after two expansion steps.

## 9.6 Detecting T<sub>E</sub>X's mode

---

```
\mode_if_horizontal_p: * \mode_if_horizontal_p:  
\mode_if_horizontal:TF * \mode_if_horizontal:TF {\langle true code\rangle} {\langle false code\rangle}
```

---

Detects if T<sub>E</sub>X is currently in horizontal mode.

---

```
\mode_if_inner_p: * \mode_if_inner_p:  
\mode_if_inner:TF * \mode_if_inner:TF {\langle true code\rangle} {\langle false code\rangle}
```

---

Detects if T<sub>E</sub>X is currently in inner mode.

---

```
\mode_if_math_p: * \mode_if_math:TF {\langle true code\rangle} {\langle false code\rangle}  
\mode_if_math:TF * Detects if TEX is currently in maths mode.
```

---

Updated: 2011-09-05

---

```
\mode_if_vertical_p: * \mode_if_vertical_p:  
\mode_if_vertical:TF * \mode_if_vertical:TF {\langle true code\rangle} {\langle false code\rangle}
```

---

Detects if T<sub>E</sub>X is currently in vertical mode.

## 9.7 Primitive conditionals

---

```
\if_predicate:w * \if_predicate:w {\langle predicate\rangle} {\langle true code\rangle} {\langle false code\rangle} \fi:
```

---

This function takes a predicate function and branches according to the result. (In practice this function would also accept a single boolean variable in place of the *<predicate>* but to make the coding clearer this should be done through *\if\_bool:N*.)

---

```
\if_bool:N * \if_bool:N {\langle boolean\rangle} {\langle true code\rangle} {\langle false code\rangle} \fi:
```

---

This function takes a boolean variable and branches according to the result.

## 9.8 Nestable recursions and mappings

There are a number of places where recursion or mapping constructs are used in `expl3`. At a low-level, these typically require insertion of tokens at the end of the content to allow “clean up”. To support such mappings in a nestable form, the following functions are provided.

---

```
\prg_break_point:Nn * \prg_break_point:Nn \langle type\rangle_map_break: {\langle code\rangle}
```

---

New: 2018-03-26

Used to mark the end of a recursion or mapping: the functions *\langle type\rangle\_map\_break:* and *\langle type\rangle\_map\_break:n* use this to break out of the loop (see *\prg\_map\_break:Nn* for how to set these up). After the loop ends, the *<code>* is inserted into the input stream. This occurs even if the break functions are *not* applied: *\prg\_break\_point:Nn* is functionally-equivalent in these cases to *\use\_ii:nn*.

---

```
\prg_map_break:Nn * \prg_map_break:Nn \⟨type⟩_map_break: {⟨user code⟩}
...  
New: 2018-03-26 \prg_break_point:Nn \⟨type⟩_map_break: {⟨ending code⟩}
```

---

Breaks a recursion in mapping contexts, inserting in the input stream the *⟨user code⟩* after the *⟨ending code⟩* for the loop. The function breaks loops, inserting their *⟨ending code⟩*, until reaching a loop with the same *⟨type⟩* as its first argument. This *⟨type⟩\_map\_break:* argument must be defined; it is simply used as a recognizable marker for the *⟨type⟩*.

For types with mappings defined in the kernel, *⟨type⟩\_map\_break:* and *⟨type⟩\_map\_break:n* are defined as *\prg\_map\_break:Nn \⟨type⟩\_map\_break: {}* and the same with {} omitted.

### 9.8.1 Simple mappings

In addition to the more complex mappings above, non-nestable mappings are used in a number of locations and support is provided for these.

---

```
\prg_break_point: * This copy of \prg_do_nothing: is used to mark the end of a fast short-term recursion:  
New: 2018-03-27 the function \prg_break:n uses this to break out of the loop.
```

---

---

```
\prg_break: * \prg_break:n {⟨code⟩} ... \prg_break_point:  
\prg_break:n * Breaks a recursion which has no ⟨ending code⟩ and which is not a user-breakable mapping  
New: 2018-03-27 (see for instance \prop_get:Nn), and inserts the ⟨code⟩ in the input stream.
```

---

## 9.9 Internal programming functions

---

```
\group_align_safe_begin: * \group_align_safe_begin:  
\group_align_safe_end: * ...
Updated: 2011-08-11 \group_align_safe_end:
```

---

These functions are used to enclose material in a TEX alignment environment within a specially-constructed group. This group is designed in such a way that it does not add brace groups to the output but does act as a group for the & token inside *\halign*. This is necessary to allow grabbing of tokens for testing purposes, as TEX uses group level to determine the effect of alignment tokens. Without the special grouping, the use of a function such as *\peek\_after:Nw* would result in a forbidden comparison of the internal *\endtemplate* token, yielding a fatal error. Each *\group\_align\_safe\_begin:* must be matched by a *\group\_align\_safe\_end:*, although this does not have to occur within the same function.

# Chapter 10

## The `\I3sys` package: System/runtime functions

### 10.1 The name of the job

---

`\c_sys_jobname_str` Constant that gets the “job name” assigned when `\TeX` starts.

New: 2015-09-19

Updated: 2019-10-27

**\TeXhackers note:** This copies the contents of the primitive `\jobname`. For technical reasons, the string here is not of the same internal form as other, but may be manipulated using normal string functions.

### 10.2 Date and time

---

`\c_sys_minute_int` The date and time at which the current job was started: these are all reported as integers.

`\c_sys_hour_int`

`\c_sys_day_int`

`\c_sys_month_int`

`\c_sys_year_int`

**\TeXhackers note:** Whilst the underlying primitives can be altered by the user, this interface to the time and date is intended to be the “real” values.

---

New: 2015-09-22

## 10.3 Engine

---

```
\sys_if_engine_luatex_p: * \sys_if_engine_pdftex:TF {\(true code)} {\(false code)}
\sys_if_engine_luatex:TF * Conditionals which allow engine-specific code to be used. The names follow naturally
\sys_if_engine_pdftex_p: * from those of the engine binaries: note that the (u)ptex tests are for ε-pTEX and ε-upTEX
\sys_if_engine_pdftex:TF * as expl3 requires the ε-TEX extensions. Each conditional is true for exactly one supported
\sys_if_engine_ptex_p:   * engine. In particular, \sys_if_engine_ptex_p: is true for ε-pTEX but false for ε-upTEX.
\sys_if_engine_ptex:TF *
\sys_if_engine_uptex_p: *
\sys_if_engine_uptex:TF *
\sys_if_engine_xetex_p: *
\sys_if_engine_xetex:TF *
```

---

New: 2015-09-07

---

**\c\_sys\_engine\_str** The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

---

New: 2015-09-19

---

**\c\_sys\_engine\_exec\_str** The name of the standard executable for the current T<sub>E</sub>X engine given as a lower case string: one of `luatex`, `luahbtex`, `pdftex`, `eptex`, `euptex` or `xetex`.

---

New: 2020-08-20

---

**\c\_sys\_engine\_format\_str** The name of the preloaded format for the current T<sub>E</sub>X run given as a lower case string: one of `lualatex` (or `dviulualatex`), `pdflatex` (or `latex`), `platex`, `uplatex` or `xelatex` for L<sup>A</sup>T<sub>E</sub>X, similar names for plain T<sub>E</sub>X (except pdftEX in DVI mode yields `etex`), and `cont-en` for ConT<sub>E</sub>Xt (i.e. the `\fmtname`).

---

**\sys\_timer:** \* `\sys_timer:`

---

New: 2020-09-24

Expands to the current value of the engine's timer clock, a non-negative integer. This function is only defined for engines with timer support. This command measures not just CPU time but real time (including time waiting for user input). The unit are scaled seconds ( $2^{-16}$  seconds).

## 10.4 Output format

---

```
\sys_if_output_dvi_p: * \sys_if_output_dvi:TF {\(true code)} {\(false code)}
\sys_if_output_dvi:TF * Conditionals which give the current output mode the TEX run is operating in. This is
\sys_if_output_pdf_p: * always one of two outcomes, DVI mode or PDF mode. The two sets of conditionals are
\sys_if_output_pdf:TF * thus complementary and are both provided to allow the programmer to emphasise the
New: 2015-09-19 most appropriate case.
```

---

---

**\c\_sys\_output\_str** The current output mode given as a lower case string: one of `dvi` or `pdf`.

---

New: 2015-09-19

## 10.5 Platform

---

```
\sys_if_platform_unix_p: * \sys_if_platform_unix:TF {{true code}} {{false code}}
\sys_if_platform_unix:TF *
\sys_if_platform_windows_p: *
\sys_if_platform_windows:TF *
```

---

New: 2018-07-27

Conditionals which allow platform-specific code to be used. The names follow the Lua `os.type()` function, *i.e.* all Unix-like systems are `unix` (including Linux and MacOS).

---

`\c_sys_platform_str` The current platform given as a lower case string: one of `unix`, `windows` or `unknown`.

---

New: 2018-07-27

## 10.6 Random numbers

---

```
\sys_rand_seed: * \sys_rand_seed:
```

---

New: 2017-05-27 Expands to the current value of the engine's random seed, a non-negative integer. In engines without random number support this expands to 0.

---

```
\sys_gset_rand_seed:n \sys_gset_rand_seed:n {{integer}}
```

---

New: 2017-05-27 Globally sets the seed for the engine's pseudo-random number generator to the *<integer expression>*. This random seed affects all `\..._rand` functions (such as `\int_rand:nn` or `\clist_rand_item:n`) as well as other packages relying on the engine's random number generator. In engines without random number support this produces an error.

**TEXhackers note:** While a 32-bit (signed) integer can be given as a seed, only the absolute value is used and any number beyond  $2^{28}$  is divided by an appropriate power of 2. We recommend using an integer in  $[0, 2^{28} - 1]$ .

## 10.7 Access to the shell

---

```
\sys_get_shell:nnN \sys_get_shell:nnN {{shell command}} {{setup}} {tl var}
\sys_get_shell:nnNTF \sys_get_shell:nnNTF {{shell command}} {{setup}} {tl var} {{true code}} {{false code}}
```

---

New: 2019-09-20

Defines `{tl var}` to the text returned by the `{shell command}`. The `{shell command}` is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the `{setup}` argument, which is run just before running the `{shell command}` (in a group). If shell escape is disabled, the `{tl var}` will be set to `\q_no_value` in the non-branching version. Note that quote characters ("") *cannot* be used inside the `{shell command}`. The `\sys_get_shell:nnNTF` conditional inserts the `true code` if the shell is available and no quote is detected, and the `false code` otherwise.

---

`\c_sys_shell_escape_int` This variable exposes the internal triple of the shell escape status. The possible values are

- 
- 0** Shell escape is disabled
  - 1** Unrestricted shell escape is enabled
  - 2** Restricted shell escape is enabled

---

`\sys_if_shell_p: * \sys_if_shell_p:`  
`\sys_if_shell:TF * \sys_if_shell:TF {<true code>} {<false code>}`

---

New: 2017-05-27 Performs a check for whether shell escape is enabled. This returns true if either of restricted or unrestricted shell escape is enabled.

---

`\sys_if_shell_unrestricted_p: * \sys_if_shell_unrestricted_p:`  
`\sys_if_shell_unrestricted:TF * \sys_if_shell_unrestricted:TF {<true code>} {<false code>}`

---

New: 2017-05-27

Performs a check for whether *unrestricted* shell escape is enabled.

---

`\sys_if_shell_restricted_p: * \sys_if_shell_restricted_p:`  
`\sys_if_shell_restricted:TF * \sys_if_shell_restricted:TF {<true code>} {<false code>}`

---

New: 2017-05-27

Performs a check for whether *restricted* shell escape is enabled. This returns false if unrestricted shell escape is enabled. Unrestricted shell escape is not considered a superset of restricted shell escape in this case. To find whether any shell escape is enabled use `\sys_if_shell:`.

---

`\sys_shell_now:n \sys_shell_now:n {<tokens>}`  
`\sys_shell_now:x` Execute *<tokens>* through shell escape immediately.

---

New: 2017-05-27

---

`\sys_shell_shipout:n \sys_shell_shipout:n {<tokens>}`  
`\sys_shell_shipout:x` Execute *<tokens>* through shell escape at shipout.

---

New: 2017-05-27

## 10.8 Loading configuration data

---

`\sys_load_backend:n \sys_load_backend:n {<backend>}`

---

New: 2019-09-12 Loads the additional configuration file needed for backend support. If the *<backend>* is empty, the standard backend for the engine in use will be loaded. This command may only be used once.

---

`\sys_ensure_backend: \sys_ensure_backend:`

---

New: 2022-07-29 Ensures that a backend has been loaded by calling `\sys_load_backend:n` if required.

---

**\c\_sys\_backend\_str** Set to the name of the backend in use by `\sys_load_backend:n` when issued. Possible values are

- `pdftex`
- `luatex`
- `xetex`
- `dvips`
- `dvipdfmx`
- `dvisvgm`

---

**\sys\_load\_debug: \sys\_load\_debug:**

New: 2019-09-12 Load the additional configuration file for debugging support.

---

### 10.8.1 Final settings

---

**\sys\_finalise: \sys\_finalise:**

New: 2019-10-06 Finalises all system-dependent functionality: required before loading a backend.

---

# Chapter 11

## The `l3msg` package

### Messages

Messages need to be passed to the user by modules, either when errors occur or to indicate how the code is proceeding. The `l3msg` module provides a consistent method for doing this (as opposed to writing directly to the terminal or log).

The system used by `l3msg` to create messages divides the process into two distinct parts. Named messages are created in the first part of the process; at this stage, no decision is made about the type of output that the message will produce. The second part of the process is actually producing a message. At this stage a choice of message *class* has to be made, for example `error`, `warning` or `info`.

By separating out the creation and use of messages, several benefits are available. First, the messages can be altered later without needing details of where they are used in the code. This makes it possible to alter the language used, the detail level and so on. Secondly, the output which results from a given message can be altered. This can be done on a message class, module or message name basis. In this way, message behaviour can be altered and messages can be entirely suppressed.

### 11.1 Creating new messages

All messages have to be created before they can be used. The text of messages is automatically wrapped to the length available in the console. As a result, formatting is only needed where it helps to show meaning. In particular, `\\\` may be used to force a new line and `\_` forces an explicit space. Additionally, `\{`, `\#`, `\}`, `\%` and `\~` can be used to produce the corresponding character.

Messages may be subdivided *by one level* using the `/` character. This is used within the message filtering system to allow for example the L<sup>A</sup>T<sub>E</sub>X kernel messages to belong to the module L<sup>A</sup>T<sub>E</sub>X while still being filterable at a more granular level. Thus for example

```
\msg_new:nnn { mymodule } { submodule / message } ...
```

will allow to filter out specifically messages from the `submodule`.

---

`\msg_new:nnnn` `\msg_new:nnnn {<module>} {<message>} {<text>} {<more text>}`  
`\msg_new:nnn` Creates a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (#1 to #4) can be used: these will be supplied at the time the message is used. An error is raised if the *<message>* already exists.

---

`\msg_set:nnnn` `\msg_set:nnnn {<module>} {<message>} {<text>} {<more text>}`  
`\msg_set:nnn`  
`\msg_gset:nnnn`  
`\msg_gset:nnn` Sets up the text for a *<message>* for a given *<module>*. The message is defined to first give *<text>* and then *<more text>* if the user requests it. If no *<more text>* is available then a standard text is given instead. Within *<text>* and *<more text>* four parameters (#1 to #4) can be used: these will be supplied at the time the message is used.

---

`\msg_if_exist_p:nn` \* `\msg_if_exist_p:nn {<module>} {<message>}`  
`\msg_if_exist:nnTF` \* `\msg_if_exist:nnTF {<module>} {<message>} {<true code>} {<false code>}`

---

New: 2012-03-03 Tests whether the *<message>* for the *<module>* is currently defined.

## 11.2 Customizable information for message modules

---

`\msg_module_name:n` \* `\msg_module_name:n {<module>}`  
New: 2018-10-10 Expands to the public name of the *<module>* as defined by `\g_msg_module_name_prop` (or otherwise leaves the *<module>* unchanged).

---

`\msg_module_type:n` \* `\msg_module_type:n {<module>}`  
New: 2018-10-10 Expands to the description which applies to the *<module>*, for example a `Package` or `Class`. The information here is defined in `\g_msg_module_type_prop`, and will default to `Package` if an entry is not present.

---

`\g_msg_module_name_prop` Provides a mapping between the module name used for messages, and that for documentation.  
New: 2018-10-10

---

`\g_msg_module_type_prop` Provides a mapping between the module name used for messages, and that type of module. For example, for L<sup>A</sup>T<sub>E</sub>X3 core messages, an empty entry is set here meaning that they are not described using the standard `Package` text.  
New: 2018-10-10

## 11.3 Contextual information for messages

---

`\msg_line_context:` ★ `\msg_line_context:`  
Prints the current line number when a message is given, and thus suitable for giving context to messages. The number itself is proceeded by the text `on line`.

---

`\msg_line_number: * \msg_line_number:`

Prints the current line number when a message is given.

---

`\msg_fatal_text:n * \msg_fatal_text:n {<module>}`

Produces the standard text

**Fatal Package <module> Error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

---

`\msg_critical_text:n * \msg_critical_text:n {<module>}`

Produces the standard text

**Critical Package <module> Error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

---

`\msg_error_text:n * \msg_error_text:n {<module>}`

Produces the standard text

**Package <module> Error**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included.

---

`\msg_warning_text:n * \msg_warning_text:n {<module>}`

Produces the standard text

**Package <module> Warning**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included. The *<type>* of *<module>* may be adjusted: **Package** is the standard outcome: see `\msg_module_type:n`.

---

`\msg_info_text:n * \msg_info_text:n {<module>}`

Produces the standard text:

**Package <module> Info**

This function can be redefined to alter the language in which the message is given, using #1 as the name of the *<module>* to be included. The *<type>* of *<module>* may be adjusted: **Package** is the standard outcome: see `\msg_module_type:n`.

---

```
\msg_see_documentation_text:n ∗ \msg_see_documentation_text:n {⟨module⟩}
```

Updated: 2018-09-30

---

Produces the standard text

See the ⟨module⟩ documentation for further information.

This function can be redefined to alter the language in which the message is given, using #1 as the name of the ⟨module⟩ to be included. The name of the ⟨module⟩ is produced using \msg\_module\_name:n.

## 11.4 Issuing messages

Messages behave differently depending on the message class. In all cases, the message may be issued supplying 0 to 4 arguments. If the number of arguments supplied here does not match the number in the definition of the message, extra arguments are ignored, or empty arguments added (of course the sense of the message may be impaired). The four arguments are converted to strings before being added to the message text: the x-type variants should be used to expand material. Note that this expansion takes place with the standard definitions in effect, which means that shorthands such as \~ or \\ are *not* available; instead one should use \iow\_char:N \~ and \iow\_newline:, respectively. The following message classes exist:

- **fatal**, ending the TeX run;
- **critical**, ending the file being input;
- **error**, interrupting the TeX run without ending it;
- **warning**, written to terminal and log file, for important messages that may require corrections by the user;
- **note** (less common than **info**) for important information messages written to the terminal and log file;
- **info** for normal information messages written to the log file only;
- **term** and **log** for un-decorated messages written to the terminal and log file, or to the log file only;
- **none** for suppressed messages.

---

```
\msg_fatal:nnnnn \msg_fatal:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩}  
\msg_fatal:nnxxx {⟨arg four⟩}
```

```
\msg_fatal:nnnn  
\msg_fatal:nnxx  
\msg_fatal:nnn  
\msg_fatal:nnx  
\msg_fatal:nn  
\msg_fatal:nnx  
\msg_fatal:nn
```

Issues ⟨module⟩ error ⟨message⟩, passing ⟨arg one⟩ to ⟨arg four⟩ to the text-creating functions. After issuing a fatal error the TeX run halts. No PDF file will be produced in this case (DVI mode runs may produce a truncated DVI file).

---

Updated: 2012-08-11

---

```
\msg_critical:nnnnn \msg_critical:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
\msg_critical:nnxxx \msg_critical:nnnnn {\arg four}
\msg_critical:nnnnn Issues \module error \message, passing \arg one to \arg four to the text-creating
\msg_critical:nnxx functions. After issuing a critical error, TeX stops reading the current input file. This
\msg_critical:nnnn may halt the TeX run (if the current file is the main file) or may abort reading a sub-file.
\msg_critical:nnn \msg_critical:nnx
\msg_critical:nnx
\msg_critical:nn
```

**TeXhackers note:** The TeX \endinput primitive is used to exit the file. In particular, the rest of the current line remains in the input stream.

Updated: 2012-08-11

---

```
\msg_error:nnnnn \msg_error:nnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
\msg_error:nnxxx \msg_error:nnnnn {\arg four}
\msg_error:nnnnn Issues \module error \message, passing \arg one to \arg four to the text-creating
\msg_error:nnxx functions. The error interrupts processing and issues the text at the terminal. After user
\msg_error:nnnn input, the run continues.
\msg_error:nnx
\msg_error:nnx
\msg_error:nn
```

---

Updated: 2012-08-11

---

```
\msg_warning:nnnnn \msg_warning:nnxxxx {\module} {\message} {\arg one} {\arg two} {\arg three}
\msg_warning:nnxxx \msg_warning:nnnnn {\arg four}
\msg_warning:nnnnn Issues \module warning \message, passing \arg one to \arg four to the text-creating
\msg_warning:nnxx functions. The warning text is added to the log file and the terminal, but the TeX run
\msg_warning:nnnn is not interrupted.
\msg_warning:nnx
\msg_warning:nnx
\msg_warning:nn
```

---

Updated: 2012-08-11

---

```
\msg_note:nnnnnn \msg_note:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_note:nnxxxx four}
\msg_note:nnnnn \msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_note:nnxxx four}
\msg_note:nnnn \msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_note:nnxx four}
\msg_note:nnn \msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_note:nnx four}
\msg_note:nn \msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_note:nn four}
```

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. For the more common `\msg_info:nnnnnn`, the information text is added to the log file only, while `\msg_note:nnnnnn` adds the info text to both the log file and the terminal. The TeX run is not interrupted.

```
\msg_info:nnnnnn
\msg_info:nnxxxx
\msg_info:nnnnn
\msg_info:nnxxx
\msg_info:nnnn
\msg_info:nnx
\msg_info:nn
\msg_info:nx
\msg_info:nn
```

---

New: 2021-05-18

---

```
\msg_term:nnnnnn \msg_term:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_term:nnxxxx four}
\msg_term:nnnnn \msg_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_term:nnxxx four}
\msg_term:nnnn \msg_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_term:nnxx four}
\msg_term:nnn \msg_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_term:nnx four}
\msg_term:nn \msg_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_term:nn four}
```

Issues *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The output is briefer than `\msg_info:nnnnnn`, omitting for instance the module name. It is added to the log file by `\msg_log:nnnnnnn` while `\msg_term:nnnnnnn` also prints it on the terminal.

```
\msg_log:nnnnnn
\msg_log:nnxxxx
\msg_log:nnnnn
\msg_log:nnxxx
\msg_log:nnnn
\msg_log:nnxx
\msg_log:nnn
\msg_log:nnx
\msg_log:nn
```

---

Updated: 2012-08-11

---

```
\msg_none:nnnnnn \msg_none:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg
\msg_none:nnxxxx four}
\msg_none:nnnnn Does nothing: used as a message class to prevent any output at all (see the discussion of
\msg_none:nnxxx message redirection).
\msg_none:nnnn
\msg_none:nnxx
\msg_none:nnn
\msg_none:nnx
\msg_none:nn
```

---

Updated: 2012-08-11

### 11.4.1 Messages for showing material

---

```
\msg_show:nnnnn \msg_show:nnnnn {{module}} {{message}} {{arg one}} {{arg two}} {{arg three}} {{arg four}}
```

`\msg_show:nnnnn` Issues `<module>` information `<message>`, passing `<arg one>` to `<arg four>` to the text-creating functions. The information text is shown on the terminal and the TeX run is interrupted in a manner similar to `\t1_show:n`. This is used in conjunction with `\msg_show_item:n` and similar functions to print complex variable contents completely. If the formatted text does not contain `>~` at the start of a line, an additional line `>~.` will be put at the end. In addition, a final period is added if not present.

---

New: 2017-12-04

### 11.4.2 Expandable error messages

In very rare cases it may be necessary to produce errors in an expansion-only context. The functions in this section should only be used if there is no alternative approach using `\msg_error:nnnnn` or other non-expandable commands from the previous section. Despite having a similar interface as non-expandable messages, expandable errors must be handled internally very differently from normal error messages, as none of the tools to print to the terminal or the log file are expandable. As a result, short-hands such as `\{` or `\\"` do not work, and messages must be very short (with default settings, they are truncated after approximately 50 characters). It is advisable to ensure that the message is understandable even when truncated, by putting the most important information up front. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

---

```
\msg_expandable_error:nnnnn * \msg_expandable_error:nnnnn {{module}} {{message}} {{arg one}} {{arg two}} {{arg three}} {{arg four}}
```

`\msg_expandable_error:nnnnn` \*

`\msg_expandable_error:nnfff` \*

`\msg_expandable_error:nnnn` \*

`\msg_expandable_error:nnff` \*

`\msg_expandable_error:nnn` \*

`\msg_expandable_error:nnf` \*

`\msg_expandable_error:nn` \*

---

New: 2015-08-06

Updated: 2019-02-28

Issues an “Undefined error” message from TeX itself using the undefined control sequence `\:::error` then prints “! `<module>`: ”`<error message>`, which should be short. With default settings, anything beyond approximately 60 characters long (or bytes in some engines) is cropped. A leading space might be removed as well.

## 11.5 Redirecting messages

Each message has a “name”, which can be used to alter the behaviour of the message when it is given. Thus we might have

```
\msg_new:nnnn { module } { my-message } { Some~text } { Some~more~text }
```

to define a message, with

```
\msg_error:nn { module } { my-message }
```

when it is used. With no filtering, this raises an error. However, we could alter the behaviour with

```
\msg_redirect_class:nn { error } { warning }
```

to turn all errors into warnings, or with

```
\msg_redirect_module:nnn { module } { error } { warning }
```

to alter only messages from that module, or even

```
\msg_redirect_name:nnn { module } { my-message } { warning }
```

to target just one message. Redirection applies first to individual messages, then to messages from one module and finally to messages of one class. Thus it is possible to select out an individual message for special treatment even if the entire class is already redirected.

Multiple redirections are possible. Redirections can be cancelled by providing an empty argument for the target class. Redirection to a missing class raises an error immediately. Infinite loops are prevented by eliminating the redirection starting from the target of the redirection that caused the loop to appear. Namely, if redirections are requested as  $A \rightarrow B$ ,  $B \rightarrow C$  and  $C \rightarrow A$  in this order, then the  $A \rightarrow B$  redirection is cancelled.

---

```
\msg_redirect_class:nn \msg_redirect_class:nn {<class one>} {{<class two>}}
```

Updated: 2012-04-27

Changes the behaviour of messages of  $\langle class\ one\rangle$  so that they are processed using the code for those of  $\langle class\ two\rangle$ . Each  $\langle class\rangle$  can be one of **fatal**, **critical**, **error**, **warning**, **note**, **info**, **term**, **log**, **none**.

---

```
\msg_redirect_module:nnn \msg_redirect_module:nnn {<module>} {<class one>} {<class two>}
```

Updated: 2012-04-27

Redirects message of  $\langle class\ one\rangle$  for  $\langle module\rangle$  to act as though they were from  $\langle class\ two\rangle$ . Messages of  $\langle class\ one\rangle$  from sources other than  $\langle module\rangle$  are not affected by this redirection. This function can be used to make some messages “silent” by default. For example, all of the **warning** messages of  $\langle module\rangle$  could be turned off with:

```
\msg_redirect_module:nnn { module } { warning } { none }
```

---

```
\msg_redirect_name:nnn \msg_redirect_name:nnn {<module>} {<message>} {{<class>}}
```

Updated: 2012-04-27

Redirects a specific  $\langle message\rangle$  from a specific  $\langle module\rangle$  to act as a member of  $\langle class\rangle$  of messages. No further redirection is performed. This function can be used to make a selected message “silent” without changing global parameters:

```
\msg_redirect_name:nnn { module } { annoying-message } { none }
```

## Chapter 12

# The `I3file` package

## File and I/O operations

This module provides functions for working with external files. Some of these functions apply to an entire file, and have prefix `\file_...`, while others are used to work with files on a line by line basis and have prefix `\ior_...` (reading) or `\iow_...` (writing).

It is important to remember that when reading external files `TEX` attempts to locate them using both the operating system path and entries in the `TEX` file database (most `TEX` systems use such a database). Thus the “current path” for `TEX` is somewhat broader than that for other programs.

For functions which expect a `<file name>` argument, this argument may contain both literal items and expandable content, which should on full expansion be the desired file name. Active characters (as declared in `\l_char_active_seq`) are *not* expanded, allowing the direct use of these in file names. Quote tokens ("") are not permitted in file names as they are reserved for internal use by some `TEX` primitives.

Spaces are trimmed at the beginning and end of the file name: this reflects the fact that some file systems do not allow or interact unpredictably with spaces in these positions. When no extension is given, this will trim spaces from the start of the name only.

### 12.1 Input–output stream management

As `TEX` engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in `LATEX3`. Instead, an internal pool of streams is maintained, and these are allocated and deallocated as needed by other modules. As a result, the programmer should close streams when they are no longer needed, to release them for other processes.

Note that I/O operations are global: streams should all be declared with global names and treated accordingly.

---

\ior\_new:N      \ior\_new:N *stream*  
\ior\_new:c    \iow\_new:N *stream*  
\iow\_new:N     Globally reserves the name of the *stream*, either for reading or for writing as appropriate. The *stream* is not opened until the appropriate \...\_open:Nn function is used.  
New: 2011-09-26   Attempting to use a *stream* which has not been opened is an error, and the *stream* will behave as the corresponding \c\_term\_....  
Updated: 2011-12-27

---

\ior\_open:Nn    \ior\_open:Nn *stream* {{*file name*}}  
\ior\_open:cn    Opens *file name* for reading using *stream* as the control sequence for file access. If the *stream* was already open it is closed before the new operation begins. The *stream* is available for access immediately and will remain allocated to *file name* until a \ior\_close:N instruction is given or the T<sub>E</sub>X run ends. If the file is not found, an error is raised.  
Updated: 2012-02-10

---

\ior\_open:NnTF \ior\_open:NnTF *stream* {{*file name*}} {{*true code*}} {{*false code*}}  
\ior\_open:cnTF   Opens *file name* for reading using *stream* as the control sequence for file access. If the *stream* was already open it is closed before the new operation begins. The *stream* is available for access immediately and will remain allocated to *file name* until a \ior\_close:N instruction is given or the T<sub>E</sub>X run ends. The *true code* is then inserted into the input stream. If the file is not found, no error is raised and the *false code* is inserted into the input stream.  
New: 2013-01-12

---

\iow\_open:Nn    \iow\_open:Nn *stream* {{*file name*}}  
\iow\_open:cn    Opens *file name* for writing using *stream* as the control sequence for file access. If the *stream* was already open it is closed before the new operation begins. The *stream* is available for access immediately and will remain allocated to *file name* until a \iow\_close:N instruction is given or the T<sub>E</sub>X run ends. Opening a file for writing clears any existing content in the file (*i.e.* writing is *not* additive).  
Updated: 2012-02-09

---

\ior\_close:N    \ior\_close:N *stream*  
\ior\_close:c   \iow\_close:N *stream*  
\iow\_close:N   Closes the *stream*. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.  
\iow\_close:c  
Updated: 2012-07-31

---

\ior\_show:N \ior\_show:N *stream*  
\ior\_show:c \ior\_log:N *stream*  
\ior\_log:N \iow\_show:N *stream*  
\ior\_log:c \iow\_log:N *stream*  
\iow\_show:N   Display (to the terminal or log file) the file name associated to the (read or write) *stream*.  
\iow\_show:c  
\iow\_log:N  
\iow\_log:c  
New: 2021-05-11

---

```
\ior_show_list: \ior_show_list:  
\ior_log_list: \ior_log_list:  
\iow_show_list: \iow_show_list:  
\iow_log_list: \iow_log_list:
```

---

New: 2017-06-27 Display (to the terminal or log file) a list of the file names associated with each open (read or write) stream. This is intended for tracking down problems.

### 12.1.1 Reading from files

Reading from files and reading from the terminal are separate processes in `expl3`. The functions `\ior_get:NN` and `\ior_str_get:NN`, and their branching equivalents, are designed to work with *files*.

---

```
\ior_get:NN \ior_get:NN <stream> <token list variable>  
\ior_get:NNTF \ior_get:NNTF <stream> <token list variable> <true code> <false code>
```

---

New: 2012-06-24  
Updated: 2019-03-23 Function that reads one or more lines (until an equal number of left and right braces are found) from the file input `<stream>` and stores the result locally in the `<token list>` variable. The material read from the `<stream>` is tokenized by TeX according to the category codes and `\endlinechar` in force when the function is used. Assuming normal settings, any lines which do not end in a comment character % have the line ending converted to a space, so for example input

```
a b c
```

results in a token list `a\b\c`. Any blank line is converted to the token `\par`. Therefore, blank lines can be skipped by using a test such as

```
\ior_get:NN \l_my_stream \l_tmpa_tl  
\tl_set:Nn \l_tmpb_tl { \par }  
\tl_if_eq:NNF \l_tmpa_tl \l_tmpb_tl  
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens. In the non-branching version, where the `<stream>` is not open the `<tl var>` is set to `\q_no_value`.

**TeXhackers note:** This protected macro is a wrapper around the TeX primitive `\read`. Regardless of settings, TeX replaces trailing space and tab characters (character codes 32 and 9) in each line by an end-of-line character (character code `\endlinechar`, omitted if `\endlinechar` is negative or too large) before turning characters into tokens according to current category codes. With default settings, spaces appearing at the beginning of lines are also ignored.

---

```
\ior_str_get:NN \ior_str_get:NN <stream> <token list variable>
\ior_str_get:NNTF \ior_str_get:NNTF <stream> <token list variable> <true code> <false code>
```

---

New: 2016-12-04 Function that reads one line from the file input  $\langle stream \rangle$  and stores the result locally in the  $\langle token list \rangle$  variable. The material is read from the  $\langle stream \rangle$  as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the  $\langle token list variable \rangle$  being empty. Unlike  $\text{\ior_get:NN}$ , line ends do not receive any special treatment. Thus input

```
a b c
```

results in a token list  $a$   $b$   $c$  with the letters  $a$ ,  $b$ , and  $c$  having category code 12. In the non-branching version, where the  $\langle stream \rangle$  is not open the  $\langle tl var \rangle$  is set to  $\text{\q_no_value}$ .

**TeXhackers note:** This protected macro is a wrapper around the  $\varepsilon$ -TeX primitive  $\text{\readline}$ . Regardless of settings,  $\text{\TeX}$  removes trailing space and tab characters (character codes 32 and 9). However, the end-line character normally added by this primitive is not included in the result of  $\text{\ior_str_get:NN}$ .

All mappings are done at the current group level, *i.e.* any local assignments made by the  $\langle function \rangle$  or  $\langle code \rangle$  discussed below remain in effect after the loop.

---

```
\ior_map_inline:Nn \ior_map_inline:Nn <stream> {<inline function>}
```

---

New: 2012-02-11 Applies the  $\langle inline function \rangle$  to each set of  $\langle lines \rangle$  obtained by calling  $\text{\ior_get:NN}$  until reaching the end of the file.  $\text{\TeX}$  ignores any trailing new-line marker from the file it reads. The  $\langle inline function \rangle$  should consist of code which receives the  $\langle line \rangle$  as #1.

---

```
\ior_str_map_inline:Nn \ior_str_map_inline:Nn <stream> {<inline function>}
```

---

New: 2012-02-11 Applies the  $\langle inline function \rangle$  to every  $\langle line \rangle$  in the  $\langle stream \rangle$ . The material is read from the  $\langle stream \rangle$  as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The  $\langle inline function \rangle$  should consist of code which receives the  $\langle line \rangle$  as #1. Note that  $\text{\TeX}$  removes trailing space and tab characters (character codes 32 and 9) from every line upon input.  $\text{\TeX}$  also ignores any trailing new-line marker from the file it reads.

---

```
\ior_map_variable>NNn \ior_map_variable>NNn <stream> <tl var> {<code>}
```

---

New: 2019-01-13 For each set of  $\langle lines \rangle$  obtained by calling  $\text{\ior_get:NN}$  until reaching the end of the file, stores the  $\langle lines \rangle$  in the  $\langle tl var \rangle$  then applies the  $\langle code \rangle$ . The  $\langle code \rangle$  will usually make use of the  $\langle variable \rangle$ , but this is not enforced. The assignments to the  $\langle variable \rangle$  are local. Its value after the loop is the last set of  $\langle lines \rangle$ , or its original value if the  $\langle stream \rangle$  is empty.  $\text{\TeX}$  ignores any trailing new-line marker from the file it reads. This function is typically faster than  $\text{\ior_map_inline:Nn}$ .

---

```
\ior_str_map_variable:Nn \ior_str_map_variable:Nn <stream> <variable> {{code}}
```

New: 2019-01-13

For each *line* in the *stream*, stores the *line* in the *variable* then applies the *code*. The material is read from the *stream* as a series of tokens with category code 12 (other), with the exception of space characters which are given category code 10 (space). The *code* will usually make use of the *variable*, but this is not enforced. The assignments to the *variable* are local. Its value after the loop is the last *line*, or its original value if the *stream* is empty. Note that TeX removes trailing space and tab characters (character codes 32 and 9) from every line upon input. TeX also ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_str_map_inline:Nn`.

---

```
\ior_map_break: \ior_map_break:
```

New: 2012-06-29

Used to terminate a `\ior_map_...` function before all lines from the *stream* have been processed. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_iор
{
  \str_if_eq:nnTF { #1 } { bingo }
  {
    \ior_map_break:
  }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map_...` scenario leads to low level TeX errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

---

```
\ior_map_break:n \ior_map_break:n {{code}}
```

New: 2012-06-29

Used to terminate a `\ior_map_...` function before all lines in the *stream* have been processed, inserting the *code* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\ior_map_inline:Nn \l_my_iор
{
  \str_if_eq:nnTF { #1 } { bingo }
  {
    \ior_map_break:n { <code> }
  }
  {
    % Do something useful
  }
}
```

Use outside of a `\ior_map_...` scenario leads to low level TeX errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before the *code* is inserted into the input stream. This depends on the design of the mapping function.

---

```
\ior_if_eof_p:N * \ior_if_eof_p:N <stream>
\ior_if_eof:NTF * \ior_if_eof:NTF <stream> {\<true code>} {\<false code>}
```

---

Updated: 2012-02-10 Tests if the end of a file *<stream>* has been reached during a reading operation. The test also returns a **true** value if the *<stream>* is not open.

### 12.1.2 Writing to files

---

```
\iow_now:Nn \iow_now:Nn <stream> {\<tokens>}
```

---

Updated: 2012-06-05 This functions writes *<tokens>* to the specified *<stream>* immediately (*i.e.* the write operation is called on expansion of `\iow_now:Nn`).

---

```
\iow_log:n \iow_log:n {\<tokens>}
```

---

Updated: 2012-06-05 This function writes the given *<tokens>* to the log (transcript) file immediately: it is a dedicated version of `\iow_now:Nn`.

---

```
\iow_term:n \iow_term:n {\<tokens>}
```

---

Updated: 2012-06-05 This function writes the given *<tokens>* to the terminal file immediately: it is a dedicated version of `\iow_now:Nn`.

---

```
\iow_shipout:Nn \iow_shipout:Nn <stream> {\<tokens>}
```

---

Updated: 2012-09-08 This functions writes *<tokens>* to the specified *<stream>* when the current page is finalised (*i.e.* at shipout). The **x**-type variants expand the *<tokens>* at the point where the function is used but *not* when the resulting tokens are written to the *<stream>* (*cf.* `\iow_shipout_x:Nn`).

**TeXhackers note:** When using `expl3` with a format other than L<sup>A</sup>T<sub>E</sub>X, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

---

```
\iow_shipout_x:Nn \iow_shipout_x:Nn <stream> {\<tokens>}
```

---

Updated: 2012-09-08 This functions writes *<tokens>* to the specified *<stream>* when the current page is finalised (*i.e.* at shipout). The *<tokens>* are expanded at the time of writing in addition to any expansion when the function is used. This makes these functions suitable for including material finalised during the page building process (such as the page number integer).

**TeXhackers note:** This is a wrapper around the TeX primitive `\write`. When using `expl3` with a format other than L<sup>A</sup>T<sub>E</sub>X, new line characters inserted using `\iow_newline:` or using the line-wrapping code `\iow_wrap:nnnN` are not recognized in the argument of `\iow_shipout:Nn`. This may lead to the insertion of additional unwanted line-breaks.

---

`\iow_char:N ∗ \iow_char:N ⟨char⟩`

Inserts *⟨char⟩* into the output stream. Useful when trying to write difficult characters such as %, {, }, etc. in messages, for example:

```
\iow_now:Nx \g_my_iow { \iow_char:N \{ text \iow_char:N \} }
```

The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

---

`\iow_newline: ∗ \iow_newline:`

Function to add a new line within the *⟨tokens⟩* written to a file. The function has no effect if writing is taking place without expansion (*e.g.* in the second argument of `\iow_now:Nn`).

**TeXhackers note:** When using `expl3` with a format other than L<sup>A</sup>T<sub>E</sub>X, the character inserted by `\iow_newline:` is not recognized by TeX, which may lead to the insertion of additional unwanted line-breaks. This issue only affects `\iow_shipout:Nn`, `\iow_shipout_x:Nn` and direct uses of primitive operations.

### 12.1.3 Wrapping lines in output

---

`\iow_wrap:nnN \iow_wrap:nnN {\langle text\rangle} {\langle run-on text\rangle} {\langle set up\rangle} {⟨function⟩}`  
`\iow_wrap:nxnN`

New: 2012-06-28  
Updated: 2017-12-04

This function wraps the *⟨text⟩* to a fixed number of characters per line. At the start of each line which is wrapped, the *⟨run-on text⟩* is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the *⟨run-on text⟩* for all lines except the first, for which the target number of characters is simply `\l_iow_line_count_int` since there is no run-on text. The *⟨text⟩* and *⟨run-on text⟩* are exhaustively expanded by the function, with the following substitutions:

- `\\" or \iow_newline`: may be used to force a new line,
- `\_` may be used to represent a forced space (for example after a control sequence),
- `\#, \%, \{, \}, \~` may be used to represent the corresponding character,
- `\iow_allow_break`: may be used to allow a line-break without inserting a space (this is experimental),
- `\iow_indent:n` may be used to indent a part of the *⟨text⟩* (not the *⟨run-on text⟩*).

Additional functions may be added to the wrapping by using the *⟨set up⟩*, which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the *⟨text⟩* which is not to be expanded on wrapping should be converted to a string using `\token_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N`, etc.

The result of the wrapping operation is passed as a braced argument to the *⟨function⟩*, which is typically a wrapper around a write operation. The output of `\iow_wrap:nnN` (*i.e.* the argument passed to the *⟨function⟩*) consists of characters of category “other” (category code 12), with the exception of spaces which have category “space” (category code 10). This means that the output does *not* expand further when written to a file.

**TeXhackers note:** Internally, `\iow_wrap:nnN` carries out an x-type expansion on the *⟨text⟩* to expand it. This is done in such a way that `\exp_not:N` or `\exp_not:n` could be used to prevent expansion of material. However, this is less conceptually clear than conversion to a string, which is therefore the supported method for handling expandable material in the *⟨text⟩*.

---

`\iow_indent:n \iow_indent:n {\langle text\rangle}`

New: 2011-09-21

In the first argument of `\iow_wrap:nnN` (for instance in messages), indents *⟨text⟩* by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the *⟨text⟩*. In case the indented *⟨text⟩* should appear on separate lines from the surrounding text, use `\\"` to force line breaks.

---

`\l_iow_line_count_int`

New: 2012-06-24

The maximum number of characters in a line to be written by the `\iow_wrap:nnN` function. This value depends on the TeX system in use: the standard value is 78, which is typically correct for unmodified TeX Live and MiKTeX systems.

### 12.1.4 Constant input–output streams, and variables

---

`\g_tmpa_iow` Scratch input stream for global use. These are never used by the kernel code, and so  
`\g_tmrb_iow` are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
New: 2017-12-11 other non-kernel code and so should only be used for short-term storage.

---

`\c_log_iow` Constant output streams for writing to the log and to the terminal (plus the log), respectively.  
`\c_term_iow`

---

`\g_tmra_iow` Scratch output stream for global use. These are never used by the kernel code, and so  
`\g_tmrb_iow` are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
New: 2017-12-11 other non-kernel code and so should only be used for short-term storage.

### 12.1.5 Primitive conditionals

---

```
\if_eof:w * \if_eof:w <stream>
    <true code>
\else:
    <false code>
\fi:
```

Tests if the `<stream>` returns “end of file”, which is true for non-existent files. The `\else:` branch is optional.

**TeXhackers note:** This is the T<sub>E</sub>X primitive `\ifeof`.

## 12.2 File operation functions

---

`\g_file_curr_dir_str` Contain the directory, name and extension of the current file. The directory is empty if  
`\g_file_curr_name_str` the file was loaded without an explicit path (*i.e.* if it is in the T<sub>E</sub>X search path), and does  
`\g_file_curr_ext_str` *not* end in / other than the case that it is exactly equal to the root directory. The `<name>`  
New: 2017-06-21 and `<ext>` parts together make up the file name, thus the `<name>` part may be thought of as the “job name” for the current file. Note that T<sub>E</sub>X does not provide information on the `<ext>` part for the main (top level) file and that this file always has an empty `<dir>` component. Also, the `<name>` here will be equal to `\c_sys_jobname_str`, which may be different from the real file name (if set using `--jobname`, for example).

---

**\l\_file\_search\_path\_seq** Each entry is the path to a directory which should be searched when seeking a file. Each path can be relative or absolute, and should not include the trailing slash. The entries are not expanded when used so may contain active characters but should not feature any variable content. Spaces need not be quoted.

New: 2017-06-18

---

**TeXhackers note:** When working as a package in L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$ , `expl3` will automatically append the current `\input@path` to the set of values from `\l_file_search_path_seq`.

---

**\file\_if\_exist:nTF** `\file_if_exist:nTF {<file name>} {<true code>} {<false code>}`

Updated: 2012-02-10 Searches for `<file name>` using the current T<sub>E</sub>X search path and the additional paths controlled by `\l_file_search_path_seq`.

---

**\file\_get:nnN** `\file_get:nnN {<filename>} {<setup>} <t1>`  
**\file\_get:nnNTF** `\file_get:nnNTF {<filename>} {<setup>} <t1> {<true code>} {<false code>}`

New: 2019-01-16 Defines `<t1>` to the contents of `<filename>`. Category codes may need to be set appropriately via the `<setup>` argument. The non-branching version sets the `<t1>` to `\q_no_value` if the file is not found. The branching version runs the `<true code>` after the assignment to `<t1>` if the file is found, and `<false code>` otherwise.

Updated: 2019-02-16

---

**\file\_get\_full\_name:nN** `\file_get_full_name:nN {<file name>} <t1>`  
**\file\_get\_full\_name:VN** `\file_get_full_name:vnTF {<file name>} <t1> {<true code>} {<false code>}`  
**\file\_get\_full\_name:nNTF** `\file_get_full_name:nNTF {<file name>} <t1> {<true code>} {<false code>}`  
**\file\_get\_full\_name:VNTF** Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found sets the `<t1 var>` the fully-qualified name of the file, *i.e.* the path and file name. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. In the non-branching version, the `<t1 var>` will be set to `\q_no_value` in the case that the file does not exist.

Updated: 2019-02-16

---

**\file\_full\_name:n**  $\star$  `\file_full_name:n {<file name>}`

**\file\_full\_name:v**  $\star$  `\file_full_name:v {<file name>}`

New: 2019-09-03 Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found leaves the fully-qualified name of the file, *i.e.* the path and file name, in the input stream. This includes an extension `.tex` when the given `<file name>` has no extension but the file found has that extension. If the file is not found on the path, the expansion is empty.

---

\file\_parse\_full\_name:nNNN \file\_parse\_full\_name:nNNN {<full name>} <dir> <name> <ext>  
\file\_parse\_full\_name:VNNN  
Parses the <full name> and splits it into three parts, each of which is returned by setting the appropriate local string variable:  
New: 2017-06-23  
Updated: 2020-06-24

---

- The <dir>: everything up to the last / (path separator) in the <file path>. As with system PATH variables and related functions, the <dir> does *not* include the trailing / unless it points to the root directory. If there is no path (only a file name), <dir> is empty.
- The <name>: everything after the last / up to the last ., where both of those characters are optional. The <name> may contain multiple . characters. It is empty if <full name> consists only of a directory name.
- The <ext>: everything after the last . (including the dot). The <ext> is empty if there is no . after the last /.

Before parsing, the <full name> is expanded until only non-expandable tokens remain, except that active characters are also not expanded. Quotes ("") are invalid in file names and are discarded from the input.

---

\file\_parse\_full\_name:n \* \file\_parse\_full\_name:n {<full name>}  
Parses the <full name> as described for \file\_parse\_full\_name:nNNN, and leaves <dir>, <name>, and <ext> in the input stream, each inside a pair of braces.  
New: 2020-06-24

---

\file\_parse\_full\_name\_apply:nN \* \file\_parse\_full\_name\_apply:nN {<full name>} <function>  
New: 2020-06-24

---

Parses the <full name> as described for \file\_parse\_full\_name:nNNN, and passes <dir>, <name>, and <ext> as arguments to <function>, as an n-type argument each, in this order.

---

\file\_hex\_dump:n ☆ \file\_hex\_dump:n {<file name>}  
\file\_hex\_dump:nnn ☆ \file\_hex\_dump:nnn {<file name>} {<start index>} {<end index>}  
New: 2019-11-19

---

Searches for <file name> using the current TeX search path and the additional paths controlled by \l\_file\_search\_path\_seq. It then expands to leave the hexadecimal dump of the file content in the input stream. The file is read as bytes, which means that in contrast to most TeX behaviour there will be a difference in result depending on the line endings used in text files. The same file will produce the same result between different engines: the algorithm used is the same in all cases. When the file is not found, the result of expansion is empty. The {<start index>} and {<end index>} values work as described for \str\_range:nnn.

---

\file\_get\_hex\_dump:nN \file\_get\_hex\_dump:nN {<file name>} <tl var>  
\file\_get\_hex\_dump:nTF \file\_get\_hex\_dump:nnnN {<file name>} {<start index>} {<end index>} <tl var>  
\file\_get\_hex\_dump:nnnN \file\_get\_hex\_dump:nnnNTF Sets the <tl var> to the result of applying \file\_hex\_dump:n/\file\_hex\_dump:nnn to the <file>. If the file is not found, the <tl var> will be set to \q\_no\_value.  
New: 2019-11-19

---

<code>\file_mdfive_hash:n</code>	<code>\file_mdfive_hash:n {&lt;file name&gt;}</code>	
<small>New: 2019-09-03</small>	Sets the <code>&lt;file name&gt;</code> to the result of applying <code>\file_mdfive_hash:n</code> to the <code>&lt;file&gt;</code> . If the file is not found, the <code>&lt;tl var&gt;</code> will be set to <code>\q_no_value</code> .	
<code>\file_get_mdfive_hash:nN</code>	<code>\file_get_mdfive_hash:nN {&lt;file name&gt;} &lt;tl var&gt;</code>	
<code>\file_get_mdfive_hash:nNTF</code>	Sets the <code>&lt;tl var&gt;</code> to the result of applying <code>\file_mdfive_hash:n</code> to the <code>&lt;file&gt;</code> . If the file is not found, the <code>&lt;tl var&gt;</code> will be set to <code>\q_no_value</code> .	
<small>New: 2017-07-11</small>		
<small>Updated: 2019-02-16</small>		
<code>\file_size:n</code>	<code>\file_size:n {&lt;file name&gt;}</code>	
<small>New: 2019-09-03</small>	Sets the <code>&lt;file name&gt;</code> to the result of applying <code>\file_size:n</code> to the <code>&lt;file&gt;</code> . If the file is not found, the <code>&lt;tl var&gt;</code> will be set to <code>\q_no_value</code> . This is not available in older versions of X <sub>E</sub> T <sub>E</sub> X.	
<code>\file_get_size:nN</code>	<code>\file_get_size:nN {&lt;file name&gt;} &lt;tl var&gt;</code>	
<code>\file_get_size:nNTF</code>	Sets the <code>&lt;tl var&gt;</code> to the result of applying <code>\file_size:n</code> to the <code>&lt;file&gt;</code> . If the file is not found, the <code>&lt;tl var&gt;</code> will be set to <code>\q_no_value</code> . This is not available in older versions of X <sub>E</sub> T <sub>E</sub> X.	
<small>New: 2017-07-09</small>		
<small>Updated: 2019-02-16</small>		
<code>\file_timestamp:n</code>	<code>\file_timestamp:n {&lt;file name&gt;}</code>	
<small>New: 2019-09-03</small>	Sets the <code>&lt;file name&gt;</code> to the result of applying <code>\file_timestamp:n</code> to the <code>&lt;file&gt;</code> . If the file is not found, the <code>&lt;tl var&gt;</code> will be set to <code>\q_no_value</code> . This is not available in older versions of X <sub>E</sub> T <sub>E</sub> X.	
<code>\file_get_timestamp:nN</code>	<code>\file_get_timestamp:nN {&lt;file name&gt;} &lt;tl var&gt;</code>	
<code>\file_get_timestamp:nNTF</code>	Sets the <code>&lt;tl var&gt;</code> to the result of applying <code>\file_timestamp:n</code> to the <code>&lt;file&gt;</code> . If the file is not found, the <code>&lt;tl var&gt;</code> will be set to <code>\q_no_value</code> . This is not available in older versions of X <sub>E</sub> T <sub>E</sub> X.	
<small>New: 2017-07-09</small>		
<small>Updated: 2019-02-16</small>		

---

```
\file_compare_timestamp_p:nNn * \file_compare_timestamp:nNn {\langle file-1\rangle} {comparator} {\langle file-2\rangle} {\langle true
\file_compare_timestamp:nNnTF * code\rangle} {\langle false code\rangle}
```

---

New: 2019-05-13

Updated: 2019-09-20

---

Compares the file stamps on the two *⟨files⟩* as indicated by the *⟨comparator⟩*, and inserts either the *⟨true code⟩* or *⟨false case⟩* as required. A file which is not found is treated as older than any file which is found. This allows for example the construct

```
\file_compare_timestamp:nNnT { source-file } > { derived-file }
{
    % Code to regenerate derived file
}
```

to work when the derived file is entirely absent. The timestamp of two absent files is regarded as different. This is not available in older versions of X<sub>H</sub>T<sub>E</sub>X.

---

```
\file_input:n \file_input:n {\langle file name\rangle}
```

---

Updated: 2017-06-26

Searches for *⟨file name⟩* in the path as detailed for `\file_if_exist:nTF`, and if found reads in the file as additional L<sub>A</sub>T<sub>E</sub>X source. All files read are recorded for information and the file name stack is updated by this function. An error is raised if the file is not found.

---

```
\file_if_exist_input:n \file_if_exist_input:n {\langle file name\rangle}
\file_if_exist_input:nF \file_if_exist_input:nF {\langle file name\rangle} {\langle false code\rangle}
```

---

New: 2014-07-02

Searches for *⟨file name⟩* using the current T<sub>E</sub>X search path and the additional paths included in `\l_file_search_path_seq`. If found then reads in the file as additional L<sub>A</sub>T<sub>E</sub>X source as described for `\file_input:n`, otherwise inserts the *⟨false code⟩*. Note that these functions do not raise an error if the file is not found, in contrast to `\file_input:n`.

---

```
\file_input_stop: \file_input_stop:
```

---

New: 2017-07-07

Ends the reading of a file started by `\file_input:n` or similar before the end of the file is reached. Where the file reading is being terminated due to an error, `\msg_critical:nn(nn)` should be preferred.

**T<sub>E</sub>Xhackers note:** This function must be used on a line on its own: T<sub>E</sub>X reads files line-by-line and so any additional tokens in the “current” line will still be read.

This is also true if the function is hidden inside another function (which will be the normal case), i.e., all tokens on the same line in the source file are still processed. Putting it on a line by itself in the definition doesn’t help as it is the line where it is used that counts!

---

```
\file_show_list: \file_show_list:
\file_log_list: \file_log_list:
```

---

These functions list all files loaded by L<sub>A</sub>T<sub>E</sub>X 2<sub>E</sub> commands that populate `\@filelist` or by `\file_input:n`. While `\file_show_list:` displays the list in the terminal, `\file_log_list:` outputs it to the log file only.

# Chapter 13

## The `\I3luateX` package: LuaTeX-specific functions

The LuaTeX engine provides access to the Lua programming language, and with it access to the “internals” of TeX. In order to use this within the framework provided here, a family of functions is available. When used with pdfTeX, pTeX, upTeX or XeTeX these raise an error: use `\sys_if_engine_luatex:T` to avoid this. Details on using Lua with the LuaTeX engine are given in the LuaTeX manual.

### 13.1 Breaking out to Lua

---

```
\lua_now:n * \lua_now:n {⟨token list⟩}
```

---

```
\lua_now:e *
```

New: 2018-06-18

The `⟨token list⟩` is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter for processing. Each `\lua_now:n` block is treated by Lua as a separate chunk. The Lua interpreter executes the `⟨Lua input⟩` immediately, and in an expandable manner.

**TeXhackers note:** `\lua_now:e` is a macro wrapper around `\directlua:` when LuaTeX is in use two expansions are required to yield the result of the Lua code.

---

```
\lua_shipout_e:n \lua_shipout:n {⟨token list⟩}
```

---

```
\lua_shipout:n
```

---

New: 2018-06-18

The `⟨token list⟩` is first tokenized by TeX, which includes converting line ends to spaces in the usual TeX manner and which respects currently-applicable TeX category codes. The resulting `⟨Lua input⟩` is passed to the Lua interpreter when the current page is finalised (*i.e.* at shipout). Each `\lua_shipout:n` block is treated by Lua as a separate chunk. The Lua interpreter will execute the `⟨Lua input⟩` during the page-building routine: no TeX expansion of the `⟨Lua input⟩` will occur at this stage.

In the case of the `\lua_shipout_e:n` version the input is fully expanded by TeX in an e-type manner during the shipout operation.

**TeXhackers note:** At a TeX level, the `⟨Lua input⟩` is stored as a “whatsit”.

---

`\lua_escape:n` \* `\lua_escape:n {<token list>}`

`\lua_escape:e` \*

New: 2015-06-29

Converts the *<token list>* such that it can safely be passed to Lua: embedded backslashes, double and single quotes, and newlines and carriage returns are escaped. This is done by prepending an extra token consisting of a backslash with category code 12, and for the line endings, converting them to `\n` and `\r`, respectively.

**TeXhackers note:** `\lua_escape:e` is a macro wrapper around `\luaescapestring`: when LuaTeX is in use two expansions are required to yield the result of the Lua code.

---

`\lua_load_module:n` `\lua_load_module:n {<Lua module name>}`

New: 2022-05-14

Loads a Lua module into the Lua interpreter.

`\lua_now:n` passes its *{<token list>}* argument to the Lua interpreter as a single line, with characters interpreted under the current catcode regime. These two facts mean that `\lua_now:n` rarely behaves as expected for larger pieces of code. Therefore, package authors should **not** write significant amounts of Lua code in the arguments to `\lua_now:n`. Instead, it is strongly recommended that they write the majority of their Lua code in a separate file, and then load it using `\lua_load_module:n`.

**TeXhackers note:** This is a wrapper around the Lua call `require '<module>'`.

## 13.2 Lua interfaces

As well as interfaces for TeX, there are a small number of Lua functions provided here.

---

`ltx.utils` Most public interfaces provided by the module are stored within the `ltx.utils` table.

---

`ltx.utils.filereadump` `<dump> = ltx.utils.filereadump(<file>,<offset>,<length>)`

Returns the uppercase hexadecimal representation of the content of the *<file>* read as bytes. If the *<length>* is given, only this part of the file is returned; similarly, one may specify the *<offset>* from the start of the file. If the *<length>* is not given, the entire file is read starting at the *<offset>*.

---

`ltx.utils.filemd5sum` `<hash> = ltx.utils.filemd5sum(<file>)`

Returns the MD5 sum of the file contents read as bytes; note that the result will depend on the nature of the line endings used in the file, in contrast to normal TeX behaviour. If the *<file>* is not found, nothing is returned with *no error raised*.

---

`ltx.utils.filemoddate` `<date> = ltx.utils.filemoddate(<file>)`

Returns the date/time of last modification of the *<file>* in the format

`D:<year><month><day><hour><minute><second><offset>`

where the latter may be Z (UTC) or *<plus-minus><hours>'<minutes>*'. If the *<file>* is not found, nothing is returned with *no error raised*.

---

```
ltx.utils.filesize size = ltx.utils.filesize(<file>)
```

Returns the size of the *<file>* in bytes. If the *<file>* is not found, nothing is returned with *no error raised*.

## Chapter 14

# The `I3legacy` package

## Interfaces to legacy concepts

There are a small number of TeX or L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  concepts which are not used in `expl3` code but which need to be manipulated when working as a L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package. To allow these to be integrated cleanly into `expl3` code, a set of legacy interfaces are provided here.

---

`\legacy_if_p:n` \* `\legacy_if:nTF {<name>} {<true code>} {<false code>}`  
`\legacy_if:nTF` \* Tests if the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub> /plain TeX conditional (generated by `\newif`) if `true` or `false` and branches accordingly. The `<name>` of the conditional should *omit* the leading `if`.

---

`\legacy_if_set_true:n` `\legacy_if_set_true:n {<name>}`  
`\legacy_if_set_false:n` `\legacy_if_set_false:n {<name>}`  
`\legacy_if_gset_true:n` Sets the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub> /plain TeX conditional `\if<name>` (generated by `\newif`) to be `true` or `false`.  
`\legacy_if_gset_false:n`

New: 2021-05-10

---

`\legacy_if_set:nn` `\legacy_if_set:nn {<name>} {<boolexpr>}`  
`\legacy_if_gset:nn` Sets the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub> /plain TeX conditional `\if<name>` (generated by `\newif`) to the result of evaluating the `<boolean expression>`.  
New: 2021-05-10

**Part IV**  
**Data types**

# Chapter 15

## The `\I3tl` package

### Token lists

`\TeX` works with tokens, and `\LaTeX3` therefore provides a number of functions to deal with lists of tokens. Token lists may be present directly in the argument to a function:

```
\foo:n { a collection of \tokens }
```

or may be stored in a so-called “token list variable”, which have the suffix `tl`: a token list variable can also be used as the argument to a function, for example

```
\foo:N \l_some_tl
```

In both cases, functions are available to test and manipulate the lists of tokens, and these have the module prefix `tl`. In many cases, functions which can be applied to token list variables are paired with similar functions for application to explicit lists of tokens: the two “views” of a token list are therefore collected together here.

A token list (explicit, or stored in a variable) can be seen either as a list of “items”, or a list of “tokens”. An item is whatever `\use:n` would grab as its argument: a single non-space token or a brace group, with optional leading explicit space characters (each item is thus itself a token list). A token is either a normal `N` argument, or `,`, `{`, or `}` (assuming normal `\TeX` category codes). Thus for example

```
{ Hello } ~ world
```

contains six items (`Hello`, `w`, `o`, `r`, `l` and `d`), but thirteen tokens (`{`, `H`, `e`, `l`, `l`, `o`, `}`, `,`, `w`, `o`, `r`, `l` and `d`). Functions which act on items are often faster than their analogue acting directly on tokens.

### 15.1 Creating and initialising token list variables

---

```
\tl_new:N \tl_new:N <tl var>
```

`\tl_new:c` Creates a new `<tl var>` or raises an error if the name is already taken. The declaration is global. The `<tl var>` is initially empty.

<u>\tl_const:Nn</u>	\tl_const:Nn <i>(tl var)</i> { <i>(token list)</i> }
<u>\tl_const:(Nx cn cx)</u>	Creates a new constant <i>(tl var)</i> or raises an error if the name is already taken. The value of the <i>(tl var)</i> is set globally to the <i>(token list)</i> .
<u>\tl_clear:N</u>	\tl_clear:N <i>(tl var)</i>
<u>\tl_clear:c</u>	Clears all entries from the <i>(tl var)</i> .
<u>\tl_gclear:N</u>	\tl_gclear:N
<u>\tl_gclear:c</u>	
<u>\tl_clear_new:N</u>	\tl_clear_new:N <i>(tl var)</i>
<u>\tl_clear_new:c</u>	Ensures that the <i>(tl var)</i> exists globally by applying \tl_new:N if necessary, then applies \tl_(g)clear:N to leave the <i>(tl var)</i> empty.
<u>\tl_gclear_new:N</u>	\tl_gclear_new:N
<u>\tl_gclear_new:c</u>	
<u>\tl_set_eq:NN</u>	\tl_set_eq:NN <i>(tl var<sub>1</sub>)</i> <i>(tl var<sub>2</sub>)</i>
<u>\tl_set_eq:(cN Nc cc)</u>	Sets the content of <i>(tl var<sub>1</sub>)</i> equal to that of <i>(tl var<sub>2</sub>)</i> .
<u>\tl_gset_eq:NN</u>	\tl_gset_eq:NN
<u>\tl_gset_eq:(cN Nc cc)</u>	
<u>\tl_concat:NNN</u>	\tl_concat:NNN <i>(tl var<sub>1</sub>)</i> <i>(tl var<sub>2</sub>)</i> <i>(tl var<sub>3</sub>)</i>
<u>\tl_concat:ccc</u>	Concatenates the content of <i>(tl var<sub>2</sub>)</i> and <i>(tl var<sub>3</sub>)</i> together and saves the result in <i>(tl var<sub>1</sub>)</i> . The <i>(tl var<sub>2</sub>)</i> is placed at the left side of the new token list.
<u>\tl_gconcat:NNN</u>	\tl_gconcat:NNN
<u>\tl_gconcat:ccc</u>	
New: 2012-05-18	
<u>\tl_if_exist_p:N *</u>	\tl_if_exist_p:N <i>(tl var)</i>
<u>\tl_if_exist_p:c *</u>	\tl_if_exist:cTF <i>(tl var)</i> { <i>(true code)</i> } { <i>(false code)</i> }
<u>\tl_if_exist:cTF *</u>	Tests whether the <i>(tl var)</i> is currently defined. This does not check that the <i>(tl var)</i> really is a token list variable.
New: 2012-03-03	

## 15.2 Adding data to token list variables

<u>\tl_set:Nn</u>	\tl_set:Nn <i>(tl var)</i> { <i>(tokens)</i> }
<u>\tl_set:(NV Nv No Nf Nx cn cV cv co cf cx)</u>	
<u>\tl_gset:Nn</u>	
<u>\tl_gset:(NV Nv No Nf Nx cn cV cv co cf cx)</u>	
Sets <i>(tl var)</i> to contain <i>(tokens)</i> , removing any previous content from the variable.	
<u>\tl_put_left:Nn</u>	\tl_put_left:Nn <i>(tl var)</i> { <i>(tokens)</i> }
<u>\tl_put_left:(NV No Nx cn cV co cx)</u>	
<u>\tl_gput_left:Nn</u>	
<u>\tl_gput_left:(NV No Nx cn cV co cx)</u>	
Appends <i>(tokens)</i> to the left side of the current content of <i>(tl var)</i> .	

---

```
\tl_put_right:Nn          \tl_put_right:Nn <tl var> {<tokens>}
\tl_put_right:(NV|No|Nx|cn|cV|co|cx)
\tl_gput_right:Nn
\tl_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

---

Appends *<tokens>* to the right side of the current content of *<tl var>*.

### 15.3 Token list conditionals

---

```
\tl_if_blank_p:n      * \tl_if_blank_p:n {<token list>}
\tl_if_blank_p:(e|V|o) * \tl_if_blank:nTF {<token list>} {<true code>} {<false code>}
\tl_if_blank:nTF      * Tests if the <token list> consists only of blank spaces (i.e. contains no item). The test is
\tl_if_blank:(e|V|o)TF * true if <token list> is zero or more explicit space characters (explicit tokens with character
Updated: 2019-09-04   code 32 and category code 10), and is false otherwise.
```

---



---

```
\tl_if_empty_p:N * \tl_if_empty_p:N <tl var>
\tl_if_empty_p:c * \tl_if_empty:NTF <tl var> {<true code>} {<false code>}
\tl_if_empty:NTF * Tests if the <token list variable> is entirely empty (i.e. contains no tokens at all).
\tl_if_empty:cTF *
```

---



---

```
\tl_if_empty_p:n      * \tl_if_empty_p:n {<token list>}
\tl_if_empty_p:(V|o) * \tl_if_empty:nTF {<token list>} {<true code>} {<false code>}
\tl_if_empty:nTF      * Tests if the <token list> is entirely empty (i.e. contains no tokens at all).
\tl_if_empty:(V|o)TF *
```

---

New: 2012-05-24

Updated: 2012-06-05

---



---

```
\tl_if_eq_p:NN      * \tl_if_eq_p:NN <tl var1> <tl var2>
\tl_if_eq_p:(Nc|cN|cc) * \tl_if_eq:NNTF <tl var1> <tl var2> {<true code>} {<false code>}
\tl_if_eq:NNTF      * Compares the content of two <token list variables> and is logically true if the two contain
\tl_if_eq:(Nc|cN|cc)TF * the same list of tokens (i.e. identical in both the list of characters they contain and the
category codes of those characters). Thus for example
```

```
  \tl_set:Nn \l_tmpa_tl { abc }
  \tl_set:Nx \l_tmpb_tl { \tl_to_str:n { abc } }
  \tl_if_eq:NNTF \l_tmpa_tl \l_tmpb_tl { true } { false }
```

yields **false**. See also `\str_if_eq:nnTF` for a comparison that ignores category codes.

---

```
\tl_if_eq:NnTF \tl_if_eq:NnTF <tl var1> {<token list2>} {<true code>} {<false code>}
```

---

`\tl_if_eq:cnTF` Tests if the *<token list variable<sub>1</sub>>* and the *<token list<sub>2</sub>>* contain the same list of tokens, both  
 New: 2020-07-14 in respect of character codes and category codes. This conditional is not expandable: see  
     `\tl_if_eq:NNTF` for an expandable version when both token lists are stored in variables,  
     or `\str_if_eq:nnTF` if category codes are not important.

---

**\tl\_if\_eq:nnTF** **\tl\_if\_eq:nnTF** {*token list<sub>1</sub>*} {*token list<sub>2</sub>*} {*true code*} {*false code*}

Tests if *token list<sub>1</sub>* and *token list<sub>2</sub>* contain the same list of tokens, both in respect of character codes and category codes. This conditional is not expandable: see **\tl\_if\_eq:NNTF** for an expandable version when token lists are stored in variables, or **\str\_if\_eq:nnTF** if category codes are not important.

---

**\tl\_if\_in:NnTF** **\tl\_if\_in:NnTF** {*tl var*} {*token list*} {*true code*} {*false code*}

---

**\tl\_if\_in:cNTF** Tests if the *token list* is found in the content of the *tl var*. The *token list* cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

---

**\tl\_if\_in:nnTF** **\tl\_if\_in:nnTF** {*token list<sub>1</sub>*} {*token list<sub>2</sub>*} {*true code*} {*false code*}

---

**\tl\_if\_in:(Vn|on|no)TF** Tests if *token list<sub>2</sub>* is found inside *token list<sub>1</sub>*. The *token list<sub>2</sub>* cannot contain the tokens {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does *not* enter brace (category code 1/2) groups.

---

**\tl\_if\_novalue\_p:n** \* **\tl\_if\_novalue\_p:n** {*token list*}

---

**\tl\_if\_novalue:nTF** \* **\tl\_if\_novalue:nTF** {*token list*} {*true code*} {*false code*}

---

**New: 2017-11-14** Tests if the *token list* is exactly equal to the special **\c\_novalue\_t1** marker. This function is intended to allow construction of flexible document interface structures in which missing optional arguments are detected.

---

**\tl\_if\_single\_p:N** \* **\tl\_if\_single\_p:N** {*tl var*}

---

**\tl\_if\_single\_p:c** \* **\tl\_if\_single:NTF** {*tl var*} {*true code*} {*false code*}

---

**\tl\_if\_single:NTF** \* Tests if the content of the *tl var* consists of a single *item*, i.e. is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group,

---

**Updated: 2011-08-13** surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to **\tl\_count:N**.

---

**\tl\_if\_single\_p:n** \* **\tl\_if\_single\_p:n** {*token list*}

---

**\tl\_if\_single:nTF** \* **\tl\_if\_single:nTF** {*token list*} {*true code*} {*false code*}

---

**Updated: 2011-08-13** Tests if the *token list* has exactly one *item*, i.e. is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to **\tl\_count:n**.

---

**\tl\_if\_single\_token\_p:n** \* **\tl\_if\_single\_token\_p:n** {*token list*}

---

**\tl\_if\_single\_token:nTF** \* **\tl\_if\_single\_token:nTF** {*token list*} {*true code*} {*false code*}

---

Tests if the token list consists of exactly one token, i.e. is either a single space character or a single normal token. Token groups ({}...) are not single tokens.

---

```
\tl_case:Nn  * \tl_case:NnTF <test token list variable>
\tl_case:cn  *
\tl_case:NnTF *
\tl_case:cnTF *
  <token list variable case1> {{code case1}}
  <token list variable case2> {{code case2}}
  ...
  <token list variable casen> {{code casen}}
}
{{true code}}
{{false code}}
```

New: 2013-07-24

---

This function compares the *<test token list variable>* in turn with each of the *<token list variable cases>*. If the two are equal (as described for `\tl_if_eq:NNTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\tl_case:Nn`, which does nothing if there is no match, is also available.

### 15.3.1 Testing the first token

---

```
\tl_if_head_eq_catcode_p:nN * \tl_if_head_eq_catcode_p:nN {{token list}} <test token>
\tl_if_head_eq_catcode_p:oN * \tl_if_head_eq_catcode:nNTF {{token list}} <test token>
\tl_if_head_eq_catcode:nNTF * {{true code}} {{false code}}
\tl_if_head_eq_catcode:oNTF *
```

Updated: 2012-07-09

---

Tests if the first *<token>* in the *<token list>* has the same category code as the *<test token>*. In the case where the *<token list>* is empty, the test is always **false**.

---

```
\tl_if_head_eq_charcode_p:nN * \tl_if_head_eq_charcode_p:nN {{token list}} <test token>
\tl_if_head_eq_charcode_p:fN * \tl_if_head_eq_charcode:nNTF {{token list}} <test token>
\tl_if_head_eq_charcode:nNTF * {{true code}} {{false code}}
\tl_if_head_eq_charcode:fNTF *
```

Updated: 2012-07-09

---

Tests if the first *<token>* in the *<token list>* has the same character code as the *<test token>*. In the case where the *<token list>* is empty, the test is always **false**.

---

```
\tl_if_head_eq_meaning_p:nN * \tl_if_head_eq_meaning_p:nN {{token list}} <test token>
\tl_if_head_eq_meaning:nNTF * \tl_if_head_eq_meaning:nNTF {{token list}} <test token>
  {{true code}} {{false code}}
```

Updated: 2012-07-09

---

Tests if the first *<token>* in the *<token list>* has the same meaning as the *<test token>*. In the case where *<token list>* is empty, the test is always **false**.

---

```
\tl_if_head_is_group_p:n * \tl_if_head_is_group_p:n {{token list}}
\tl_if_head_is_group:nTF * \tl_if_head_is_group:nTF {{token list}} {{true code}} {{false code}}
```

New: 2012-07-08

---

Tests if the first *<token>* in the *<token list>* is an explicit begin-group character (with category code 1 and any character code), in other words, if the *<token list>* starts with a brace group. In particular, the test is **false** if the *<token list>* starts with an implicit token such as `\c_group_begin_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

---

```
\tl_if_head_is_N_type_p:n ∗ \tl_if_head_is_N_type_p:n {⟨token list⟩}
\tl_if_head_is_N_type:nTF ∗ \tl_if_head_is_N_type:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

---

New: 2012-07-08

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is a normal N-type argument. In other words, it is neither an explicit space character (explicit token with character code 32 and category code 10) nor an explicit begin-group character (with category code 1 and any character code). An empty argument yields **false**, as it does not have a normal first token. This function is useful to implement actions on token lists on a token by token basis.

---

```
\tl_if_head_is_space_p:n ∗ \tl_if_head_is_space_p:n {⟨token list⟩}
\tl_if_head_is_space:nTF ∗ \tl_if_head_is_space:nTF {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

---

Updated: 2012-07-08

Tests if the first ⟨*token*⟩ in the ⟨*token list*⟩ is an explicit space character (explicit token with character code 32 and category code 10). In particular, the test is **false** if the ⟨*token list*⟩ starts with an implicit token such as `\c_space_token`, or if it is empty. This function is useful to implement actions on token lists on a token by token basis.

## 15.4 Working with token lists as a whole

### 15.4.1 Using token lists

---

```
\tl_to_str:n ∗ \tl_to_str:n {⟨token list⟩}
\tl_to_str:V ∗
```

---

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, leaving the resulting character tokens in the input stream. A ⟨*string*⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This function requires only a single expansion. Its argument *must* be braced.

**TeXhackers note:** This is the  $\varepsilon$ -TeX primitive `\detokenize`. Converting a ⟨*token list*⟩ to a ⟨*string*⟩ yields a concatenation of the string representations of every token in the ⟨*token list*⟩. The string representation of a control sequence is

- an escape character, whose character code is given by the internal parameter `\escapechar`, absent if the `\escapechar` is negative or greater than the largest character code;
- the control sequence name, as defined by `\cs_to_str:N`;
- a space, unless the control sequence name is a single character whose category at the time of expansion of `\tl_to_str:n` is not “letter”.

The string representation of an explicit character token is that character, doubled in the case of (explicit) macro parameter characters (normally #). In particular, the string representation of a token list may depend on the category codes in effect when it is evaluated, and the value of the `\escapechar`: for instance `\tl_to_str:n f\{a}` normally produces the three character “backslash”, “lower-case a”, “space”, but it may also produce a single “lower-case a” if the escape character is negative and `a` is currently not a letter.

---

```
\tl_to_str:N ∗ \tl_to_str:N ⟨tl var⟩
\tl_to_str:c ∗
```

---

Converts the content of the ⟨*tl var*⟩ into a series of characters with category code 12 (other) with the exception of spaces, which retain category code 10 (space). This ⟨*string*⟩ is then left in the input stream. For low-level details, see the notes given for `\tl_to_str:n`.

---

`\tl_use:N` \* `\tl_use:N` *{tl var}*

`\tl_use:c` \*

Recovers the content of a *{tl var}* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a *{tl var}* directly without an accessor function.

### 15.4.2 Counting and reversing token lists

---

`\tl_count:n` \* `\tl_count:n` *{tokens}*

`\tl_count:(V|o)` \*

Counts the number of *{items}* in *{tokens}* and leaves this information in the input stream.

New: 2012-05-13

Unbraced tokens count as one element as do each token group *{...}*. This process ignores any unprotected spaces within *{tokens}*. See also `\tl_count:N`. This function requires three expansions, giving an *{integer denotation}*.

---

`\tl_count:N` \* `\tl_count:N` *{tl var}*

`\tl_count:c` \*

Counts the number of *{items}* in the *{tl var}* and leaves this information in the input stream. Unbraced tokens count as one element as do each token group *{...}*. This process ignores any unprotected spaces within the *{tl var}*. See also `\tl_count:n`. This function requires three expansions, giving an *{integer denotation}*.

---

`\tl_count_tokens:n` \* `\tl_count_tokens:n` *{tokens}*

New: 2019-02-25

Counts the number of TeX tokens in the *{tokens}* and leaves this information in the input stream. Every token, including spaces and braces, contributes one to the total; thus for instance, the token count of `a~{bc}` is 6.

---

`\tl_reverse:n` \* `\tl_reverse:n` *{token list}*

`\tl_reverse:(V|o)` \*

Reverses the order of the *{items}* in the *{token list}*, so that *{item}\_1*{item}\_2*{item}\_3* ...*{item}\_n** becomes *{item}\_n*{item}\_3*{item}\_2*{item}\_1**. This process preserves unprotected space within the *{token list}*. Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider `\tl_reverse_items:n`. See also `\tl_reverse:N`.***

**TeXhackers note:** The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

`\tl_reverse:N` \* `\tl_reverse:N` *{tl var}*

`\tl_reverse:c`

`\tl_greverse:N`

`\tl_greverse:c`

Updated: 2012-01-08

Sets the *{tl var}* to contain the result of reversing the order of its *{items}*, so that *{item}\_1*{item}\_2*{item}\_3* ...*{item}\_n** becomes *{item}\_n*{item}\_3*{item}\_2*{item}\_1**. This process preserves unprotected spaces within the *{token list variable}*. Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. This is equivalent to a combination of an assignment and `\tl_reverse:V`. See also `\tl_reverse_items:n` for improved performance.***

---

`\tl_reverse_items:n` \* `\tl_reverse_items:n {<token list>}`

New: 2012-01-08

Reverses the order of the  $\langle items \rangle$  stored in  $\langle tl var \rangle$ , so that  $\{ \langle item_1 \rangle \} \{ \langle item_2 \rangle \} \{ \langle item_3 \rangle \} \dots \{ \langle item_n \rangle \}$  becomes  $\{ \langle item_n \rangle \} \dots \{ \langle item_3 \rangle \} \{ \langle item_2 \rangle \} \{ \langle item_1 \rangle \}$ . This process removes any unprotected space within the  $\langle token list \rangle$ . Braced token groups are copied without reversing the order of tokens, and keep the outer set of braces. Items which are initially not braced are copied with braces in the result. In cases where preserving spaces is important, consider the slower function `\tl_reverse:n`.

**TeXhackers note:** The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

`\tl_trim_spaces:n` \* `\tl_trim_spaces:n {<token list>}`

`\tl_trim_spaces:o` \*

New: 2011-07-09

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the  $\langle token list \rangle$  and leaves the result in the input stream.

Updated: 2012-06-25

**TeXhackers note:** The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

`\tl_trim_spaces_apply:nN` \* `\tl_trim_spaces_apply:nN {<token list>} <function>`

`\tl_trim_spaces_apply:on` \*

New: 2018-04-12

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the  $\langle token list \rangle$  and passes the result to the  $\langle function \rangle$  as an `n`-type argument.

---

`\tl_trim_spaces:N` `\tl_trim_spaces:N {tl var}`

`\tl_trim_spaces:c`

Sets the  $\langle tl var \rangle$  to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

`\tl_gtrim_spaces:N`

`\tl_gtrim_spaces:c`

New: 2011-07-09

### 15.4.3 Viewing token lists

---

`\tl_show:N` `\tl_show:N {tl var}`

`\tl_show:c`

Displays the content of the  $\langle tl var \rangle$  on the terminal.

Updated: 2021-04-29

**TeXhackers note:** This is similar to the T<sub>E</sub>X primitive `\show`, wrapped to a fixed number of characters per line.

---

`\tl_show:n` `\tl_show:n {<token list>}`

Updated: 2015-08-07

Displays the  $\langle token list \rangle$  on the terminal.

**TeXhackers note:** This is similar to the ε-T<sub>E</sub>X primitive `\showtokens`, wrapped to a fixed number of characters per line.

---

`\tl_log:N` `\tl_log:N <tl var>`  
`\tl_log:c` Writes the content of the `<tl var>` in the log file. See also `\tl_show:N` which displays the result in the terminal.  
New: 2014-08-22  
Updated: 2021-04-29

---

---

`\tl_log:n` `\tl_log:n {<token list>}`  
New: 2014-08-22 Writes the `<token list>` in the log file. See also `\tl_show:n` which displays the result in the terminal.  
Updated: 2015-08-07

---

## 15.5 Manipulating items in token lists

### 15.5.1 Mapping over token lists

All mappings are done at the current group level, *i.e.* any local assignments made by the `<function>` or `<code>` discussed below remain in effect after the loop.

---

`\tl_map_function:NN`  $\star$  `\tl_map_function:NN <tl var> <function>`  
`\tl_map_function:cN`  $\star$  Applies `<function>` to every `<item>` in the `<tl var>`. The `<function>` receives one argument for each iteration. This may be a number of tokens if the `<item>` was stored within braces. Hence the `<function>` should anticipate receiving n-type arguments. See also `\tl_map_function:nN`.  
Updated: 2012-06-29

---

---

`\tl_map_function:nN`  $\star$  `\tl_map_function:nN {<token list>} <function>`  
Updated: 2012-06-29 Applies `<function>` to every `<item>` in the `<token list>`. The `<function>` receives one argument for each iteration. This may be a number of tokens if the `<item>` was stored within braces. Hence the `<function>` should anticipate receiving n-type arguments. See also `\tl_map_function:NN`.

---

---

`\tl_map_inline:Nn` `\tl_map_inline:Nn <tl var> {<inline function>}`  
`\tl_map_inline:cn` Applies the `<inline function>` to every `<item>` stored within the `<tl var>`. The `<inline function>` should consist of code which receives the `<item>` as #1. See also `\tl_map_function:NN`.  
Updated: 2012-06-29

---

---

`\tl_map_inline:nn` `\tl_map_inline:nn {<token list>} {<inline function>}`  
Updated: 2012-06-29 Applies the `<inline function>` to every `<item>` stored within the `<token list>`. The `<inline function>` should consist of code which receives the `<item>` as #1. See also `\tl_map_function:nN`.

---

---

\tl\_map\_tokens:Nn  $\star$  \tl\_map\_tokens:Nn  $\langle tl\ var \rangle \{ \langle code \rangle \}$   
\tl\_map\_tokens:cn  $\star$  \tl\_map\_tokens:nn  $\{ \langle tokens \rangle \} \{ \langle code \rangle \}$   
\tl\_map\_tokens:nn  $\star$  Analogue of \tl\_map\_function:NN which maps several tokens instead of a single function.  
New: 2019-09-02  
group. For instance,

```
\tl_map_tokens:Nn \l_my_t1 { \prg_replicate:nn { 2 } }
```

expands to twice each  $\langle item \rangle$  in the  $\langle tl\ var \rangle$ : for each  $\langle item \rangle$  in \l\_my\_t1 the function \prg\_replicate:nn receives 2 and  $\langle item \rangle$  as its two arguments. The function \tl\_map\_inline:Nn is typically faster but is not expandable.

---

\tl\_map\_variable>NNn \tl\_map\_variable:NNn  $\langle tl\ var \rangle \langle variable \rangle \{ \langle code \rangle \}$   
\tl\_map\_variable:cNn Stores each  $\langle item \rangle$  of the  $\langle tl\ var \rangle$  in turn in the (token list)  $\langle variable \rangle$  and applies the  $\langle code \rangle$ . The  $\langle code \rangle$  will usually make use of the  $\langle variable \rangle$ , but this is not enforced. The assignments to the  $\langle variable \rangle$  are local. Its value after the loop is the last  $\langle item \rangle$  in the  $\langle tl\ var \rangle$ , or its original value if the  $\langle tl\ var \rangle$  is blank. See also \tl\_map\_inline:Nn.  
Updated: 2012-06-29

---

\tl\_map\_variable:nNn \tl\_map\_variable:nNn  $\{ \langle token\ list \rangle \} \langle variable \rangle \{ \langle code \rangle \}$   
Updated: 2012-06-29 Stores each  $\langle item \rangle$  of the  $\langle token\ list \rangle$  in turn in the (token list)  $\langle variable \rangle$  and applies the  $\langle code \rangle$ . The  $\langle code \rangle$  will usually make use of the  $\langle variable \rangle$ , but this is not enforced. The assignments to the  $\langle variable \rangle$  are local. Its value after the loop is the last  $\langle item \rangle$  in the  $\langle tl\ var \rangle$ , or its original value if the  $\langle tl\ var \rangle$  is blank. See also \tl\_map\_inline:nn.

---

\tl\_map\_break:  $\star$  \tl\_map\_break:  
Updated: 2012-06-29 Used to terminate a \tl\_map\_... function before all entries in the  $\langle token\ list\ variable \rangle$  have been processed. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_t1
{
    \str_if_eq:nnT { #1 } { bingo } { \tl_map_break: }
    % Do something useful
}
```

See also \tl\_map\_break:n. Use outside of a \tl\_map\_... scenario leads to low level TeX errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before the  $\langle tokens \rangle$  are inserted into the input stream. This depends on the design of the mapping function.

---

`\tl_map_break:n`  $\star$  `\tl_map_break:n {<code>}`

Updated: 2012-06-29

Used to terminate a `\tl_map_...` function before all entries in the *<token list variable>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\tl_map_inline:Nn \l_my_tl
{
    \str_if_eq:nnT { #1 } { bingo }
        { \tl_map_break:n { <code> } }
    % Do something useful
}
```

Use outside of a `\tl_map_...` scenario leads to low level T<sub>E</sub>X errors.

**T<sub>E</sub>Xhackers note:** When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

### 15.5.2 Head and tail of token lists

Functions which deal with either only the very first item (balanced text or single normal token) in a token list, or the remaining tokens.

---

`\tl_head:N`  $\star$  `\tl_head:n {<token list>}`

`\tl_head:n`  $\star$

`\tl_head:(V|v|f)`  $\star$

Updated: 2012-09-09

Leaves in the input stream the first *<item>* in the *<token list>*, discarding the rest of the *<token list>*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\tl_head:n { abc }
```

and

```
\tl_head:n { ~ abc }
```

both leave `a` in the input stream. If the “head” is a brace group, rather than a single token, the braces are removed, and so

```
\tl_head:n { ~ { ~ ab } c }
```

yields `ab`. A blank *<token list>* (see `\tl_if_blank:nTF`) results in `\tl_head:n` leaving nothing in the input stream.

**T<sub>E</sub>Xhackers note:** The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

```
\tl_head:w * \tl_head:w <token list> { } \q_stop
```

Leaves in the input stream the first *<item>* in the *<token list>*, discarding the rest of the *<token list>*. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded. A blank *<token list>* (which consists only of space characters) results in a low-level TeX error, which may be avoided by the inclusion of an empty group in the input (as shown), without the need for an explicit test. Alternatively, `\tl_if_blank:nF` may be used to avoid using the function with a “blank” argument. This function requires only a single expansion, and thus is suitable for use within an o-type expansion. In general, `\tl_head:n` should be preferred if the number of expansions is not critical.

---

```
\tl_tail:N      * \tl_tail:n <token list>
```

```
\tl_tail:n      * Discards all leading explicit space characters (explicit tokens with character code 32 and
```

```
\tl_tail:(V|v|f) * category code 10) and the first <item> in the <token list>, and leaves the remaining tokens  
Updated: 2012-09-01 in the input stream. Thus for example
```

---

```
\tl_tail:n { a ~ {bc} d }
```

and

```
\tl_tail:n { ~ a ~ {bc} d }
```

both leave `\{bc\}d` in the input stream. A blank *<token list>* (see `\tl_if_blank:nTF`) results in `\tl_tail:n` leaving nothing in the input stream.

**TeXhackers note:** The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an x-type or e-type argument expansion.

If you wish to handle token lists where the first token may be a space, and this needs to be treated as the head/tail, this can be accomplished using `\tl_if_head_is_space:nTF`, for example

```
\exp_last_unbraced:NNo
  \cs_new:Npn \__mypkg_gobble_space:w \c_space_tl { }
\cs_new:Npn \mypkg_tl_head_keep_space:n #1
{
  \tl_if_head_is_space:nTF {#1}
  { ~ }
  { \tl_head:n {#1} }
}
\cs_new:Npn \mypkg_tl_tail_keep_space:n #1
{
  \tl_if_head_is_space:nTF {#1}
  { \exp_not:o { \__mypkg_gobble_space:w #1 } }
  { \tl_tail:n {#1} }
}
```

### 15.5.3 Items and ranges in token lists

---

`\tl_item:nn` \* `\tl_item:nn {⟨token list⟩} {⟨integer expression⟩}`

`\tl_item:Nn` \* Indexing items in the *⟨token list⟩* from 1 on the left, this function evaluates the *⟨integer expression⟩* and leaves the appropriate item from the *⟨token list⟩* in the input stream.

New: 2014-07-17 If the *⟨integer expression⟩* is negative, indexing occurs from the right of the token list, starting at  $-1$  for the right-most item. If the index is out of bounds, then the function expands to nothing.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *⟨item⟩* does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

`\tl_rand_item:N` \* `\tl_rand_item:N ⟨t1 var⟩`  
`\tl_rand_item:c` \* `\tl_rand_item:n {⟨token list⟩}`

`\tl_rand_item:n` \* Selects a pseudo-random item of the *⟨token list⟩*. If the *⟨token list⟩* is blank, the result is empty. This is not available in older versions of X<sub>E</sub>T<sub>E</sub>X.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *⟨item⟩* does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

```
\tl_range:Nnn * \tl_range:Nnn {tl var} {\langle start index\rangle} {\langle end index\rangle}
\tl_range:nnn * \tl_range:nnn {\langle token list\rangle} {\langle start index\rangle} {\langle end index\rangle}
```

---

New: 2017-02-17 Leaves in the input stream the items from the  $\langle start index \rangle$  to the  $\langle end index \rangle$  inclusive.  
Updated: 2017-07-15 Spaces and braces are preserved between the items returned (but never at either end of the list). Here  $\langle start index \rangle$  and  $\langle end index \rangle$  should be  $\langle integer expressions \rangle$ . For describing in detail the functions' behavior, let  $m$  and  $n$  be the start and end index respectively. If either is 0, the result is empty. A positive index means ‘start counting from the left end’, and a negative index means ‘from the right end’. Let  $l$  be the count of the token list.

The *actual start point* is determined as  $M = m$  if  $m > 0$  and as  $M = l + m + 1$  if  $m < 0$ . Similarly the *actual end point* is  $N = n$  if  $n > 0$  and  $N = l + n + 1$  if  $n < 0$ . If  $M > N$ , the result is empty. Otherwise it consists of all items from position  $M$  to position  $N$  inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions  $s$  for  $s \leq 0$  or  $s > l$ .

Spaces in between items in the actual range are preserved. Spaces at either end of the token list will be removed anyway (think to the token list being passed to `\tl_trim_spaces:n` to begin with).

Thus, with  $l = 7$  as in the examples below, all of the following are equivalent and result in the whole token list

```
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { 1 } { 12 }
\tl_range:nnn { abcd~{e{}}fg } { -7 } { 7 }
\tl_range:nnn { abcd~{e{}}fg } { -12 } { 7 }
```

Here are some more interesting examples. The calls

```
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd{e{}}` on the terminal; similarly

```
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { 2 } { -3 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { 5 } }
\iow_term:x { \tl_range:nnn { abcd~{e{}}fg } { -6 } { -3 } }
```

are all equivalent and will print `bcd {e{}}` on the terminal (note the space in the middle). To the contrary,

```
\tl_range:nnn { abcd~{e{}}f } { 2 } { 4 }
```

will discard the space after ‘d’.

If we want to get the items from, say, the third to the last in a token list `<tl>`, the call is `\tl_range:nnn { <tl> } { 3 } { -1 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

For better performance, see `\tl_range_braced:nnn` and `\tl_range_unbraced:nnn`.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the  $\langle item \rangle$  does not expand further when appearing in an `x`-type or `e`-type argument expansion.

#### 15.5.4 Sorting token lists

---

```
\tl_sort:Nn \tl_sort:Nn {tl var} {\langle comparison code\rangle}
```

`\tl_sort:cn`  
`\tl_gsort:Nn`  
`\tl_gsort:cn`

Sorts the items in the `\langle tl var\rangle` according to the `\langle comparison code\rangle`, and assigns the result to `\langle tl var\rangle`. The details of sorting comparison are described in Section 6.1.

New: 2017-02-06

---

---

```
\tl_sort:nN * \tl_sort:nN {\langle token list\rangle} {\langle conditional\rangle}
```

New: 2017-02-06

---

Sorts the items in the `\langle token list\rangle`, using the `\langle conditional\rangle` to compare items, and leaves the result in the input stream. The `\langle conditional\rangle` should have signature :nnTF, and return `true` if the two items being compared should be left in the same order, and `false` if the items should be swapped. The details of sorting comparison are described in Section 6.1.

**TeXhackers note:** The result is returned within `\exp_not:n`, which means that the token list does not expand further when appearing in an `x`-type or `e`-type argument expansion.

### 15.6 Manipulating tokens in token lists

#### 15.6.1 Replacing tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a category code 1/2 pair).

---

```
\tl_replace_once:Nnn \tl_replace_once:Nnn {tl var} {\langle old tokens\rangle} {\langle new tokens\rangle}
```

`\tl_replace_once:cnn`  
`\tl_greplace_once:Nnn`  
`\tl_greplace_once:cnn`

Replaces the first (leftmost) occurrence of `\langle old tokens\rangle` in the `\langle tl var\rangle` with `\langle new tokens\rangle`. `\langle Old tokens\rangle` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

Updated: 2011-08-11

---

---

```
\tl_replace_all:Nnn \tl_replace_all:Nnn {tl var} {\langle old tokens\rangle} {\langle new tokens\rangle}
```

`\tl_replace_all:cnn`  
`\tl_greplace_all:Nnn`  
`\tl_greplace_all:cnn`

Replaces all occurrences of `\langle old tokens\rangle` in the `\langle tl var\rangle` with `\langle new tokens\rangle`. `\langle Old tokens\rangle` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern `\langle old tokens\rangle` may remain after the replacement (see `\tl_remove_all:Nn` for an example).

Updated: 2011-08-11

---

---

```
\tl_remove_once:Nn \tl_remove_once:Nn {tl var} {\langle tokens\rangle}
```

`\tl_remove_once:cn`  
`\tl_gremove_once:Nn`  
`\tl_gremove_once:cn`

Removes the first (leftmost) occurrence of `\langle tokens\rangle` from the `\langle tl var\rangle`. `\langle Tokens\rangle` cannot contain `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

Updated: 2011-08-11

---

---

```
\tl_remove_all:Nn \tl_remove_all:Nn <tl var> {{tokens}}
```

```
\tl_remove_all:cn
```

```
\tl_gremove_all:Nn
```

```
\tl_gremove_all:cn
```

Updated: 2011-08-11

Removes all occurrences of *<tokens>* from the *<tl var>*. *(Tokens)* cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern *<tokens>* may remain after the removal, for instance,

```
\tl_set:Nn \l_tmpa_tl {abcccd} \tl_remove_all:Nn \l_tmpa_tl {bc}
```

results in *\l\_tmpa\_tl* containing abcd.

### 15.6.2 Reassigning category codes

These functions allow the rescanning of tokens: re-apply TeX's tokenization process to apply category codes different from those in force when the tokens were absorbed. Whilst this functionality is supported, it is often preferable to find alternative approaches to achieving outcomes rather than rescanning tokens (for example construction of token lists token-by-token with intervening category code changes or using `\char_generate:nn`).

---

```
\tl_set_rescan:Nnn \tl_set_rescan:Nnn <tl var> {{setup}} {{tokens}}
```

```
\tl_set_rescan:(Nno|Nnx|cnn|cno|cnx)
```

```
\tl_gset_rescan:Nnn
```

```
\tl_gset_rescan:(Nno|Nnx|cnn|cno|cnx)
```

Updated: 2015-08-11

Sets *<tl var>* to contain *<tokens>*, applying the category code régime specified in the *<setup>* before carrying out the assignment. (Category codes applied to tokens not explicitly covered by the *<setup>* are those in force at the point of use of `\tl_set_rescan:Nnn`.) This allows the *<tl var>* to contain material with category codes other than those that apply when *<tokens>* are absorbed. The *<setup>* is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_rescan:nn`.

**TEXhackers note:** The *<tokens>* are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user *<setup>*), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

---

`\tl_rescan:nn` `\tl_rescan:nn {\langle setup\rangle} {\langle tokens\rangle}`

Updated: 2015-08-11

Rescans  $\langle tokens \rangle$  applying the category code régime specified in the  $\langle setup \rangle$ , and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the  $\langle setup \rangle$  are those in force at the point of use of `\tl_rescan:nn`.) The  $\langle setup \rangle$  is run within a group and may contain any valid input, although only changes in category codes, such as uses of `\cctab_select:N`, are relevant. See also `\tl_set_rescan:Nnn`, which is more robust than using `\tl_set:Nn` in the  $\langle tokens \rangle$  argument of `\tl_rescan:nn`.

**TeXhackers note:** The  $\langle tokens \rangle$  are first turned into a string (using `\tl_to_str:n`). If the string contains one or more characters with character code `\newlinechar` (set equal to `\endlinechar` unless that is equal to 32, before the user  $\langle setup \rangle$ ), then it is split into lines at these characters, then read as if reading multiple lines from a file, ignoring spaces (catcode 10) at the beginning and spaces and tabs (character code 32 or 9) at the end of every line. Otherwise, spaces (and tabs) are retained at both ends of the single-line string, as if it appeared in the middle of a line read from a file.

Contrarily to the `\scantokens` primitive, `\tl_rescan:nn` tokenizes the whole string in the same category code regime rather than one token at a time, so that directives such as `\verb` that rely on changing category codes will not function properly.

## 15.7 Constant token lists

---

`\c_empty_tl` Constant that is always empty.

---

`\c_novalue_tl` A marker for the absence of an argument. This constant `tl` can safely be typeset (*cf.* `\q_nil`), with the result being `-NoValue-`. It is important to note that `\c_novalue_tl` is constructed such that it will *not* match the simple text input `-NoValue-`, *i.e.* that

`\tl_if_eq:NnTF \c_novalue_tl { -NoValue- }`

is logically `false`. The `\c_novalue_tl` marker is intended for use in creating document-level interfaces, where it serves as an indicator that an (optional) argument was omitted. In particular, it is distinct from a simple empty `tl`.

---

`\c_space_tl` An explicit space character contained in a token list (compare this with `\c_space_token`). For use where an explicit space is required.

## 15.8 Scratch token lists

---

`\l_tmpa_tl` Scratch token lists for local assignment. These are never used by the kernel code, and so  
`\l_tmpb_tl` are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_t1` Scratch token lists for global assignment. These are never used by the kernel code, and  
`\g_tmpb_t1` so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten  
by other non-kernel code and so should only be used for short-term storage.

## Chapter 16

# The `I3str` package: Strings

`TEX` associates each character with a category code: as such, there is no concept of a “string” as commonly understood in many other programming languages. However, there are places where we wish to manipulate token lists while in some sense “ignoring” category codes: this is done by treating token lists as strings in a `TEX` sense.

A `TEX` string (and thus an `expl3` string) is a series of characters which have category code 12 (“other”) with the exception of space characters which have category code 10 (“space”). Thus at a technical level, a `TEX` string is a token list with the appropriate category codes. In this documentation, these are simply referred to as strings.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module which accept a token list argument first convert it to a string using `\tl_to_str:n` for internal processing, and do not treat a token list or the corresponding string representation differently.

As a string is a subset of the more general token list, it is sometimes unclear when one should be used over the other. Use a string variable for data that isn’t primarily intended for typesetting and for which a level of protection from unwanted expansion is suitable. This data type simplifies comparison of variables since there are no concerns about expansion of their contents.

The functions `\cs_to_str:N`, `\tl_to_str:n`, `\tl_to_str:N` and `\token_to_str:N` (and variants) generate strings from the appropriate input: these are documented in `I3basics`, `I3tl` and `I3token`, respectively.

Most expandable functions in this module come in three flavours:

- `\str_...:N`, which expect a token list or string variable as their argument;
- `\str_...:n`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n`, which ignores any space encountered during the operation: these functions are typically faster than those which take care of escaping spaces appropriately.

## 16.1 Creating and initialising string variables

---

\str\_new:N \str\_new:N *str var*  
\str\_new:c  
Creates a new *str var* or raises an error if the name is already taken. The declaration  
New: 2015-09-18 is global. The *str var* is initially empty.

---

\str\_const:Nn \str\_const:Nn *str var* {*token list*}  
\str\_const:(NV|Nx|cn|cV|cx)  
Creates a new constant *str var* or raises an error if the name is already taken. The  
New: 2015-09-18 value of the *str var* is set globally to the *token list*, converted to a string.  
Updated: 2018-07-28

---

\str\_clear:N \str\_clear:N *str var*  
\str\_clear:c  
Clears the content of the *str var*.  
\str\_gclear:N  
\str\_gclear:c  
New: 2015-09-18

---

\str\_clear\_new:N \str\_clear\_new:N *str var*  
\str\_clear\_new:c  
Ensures that the *str var* exists globally by applying \str\_new:N if necessary, then  
New: 2015-09-18 applies \str\_(g)clear:N to leave the *str var* empty.

---

\str\_set\_eq:NN \str\_set\_eq:NN *str var*<sub>1</sub> *str var*<sub>2</sub>  
\str\_set\_eq:(cN|Nc|cc)  
\str\_gset\_eq:NN  
\str\_gset\_eq:(cN|Nc|cc)  
Sets the content of *str var*<sub>1</sub> equal to that of *str var*<sub>2</sub>.  
New: 2015-09-18

---

\str\_concat:NNN \str\_concat:NNN *str var*<sub>1</sub> *str var*<sub>2</sub> *str var*<sub>3</sub>  
\str\_concat:ccc  
\str\_gconcat:NNN  
\str\_gconcat:ccc  
Concatenates the content of *str var*<sub>2</sub> and *str var*<sub>3</sub> together and saves the result in  
New: 2017-10-08 *str var*<sub>1</sub>. The *str var*<sub>2</sub> is placed at the left side of the new string variable. The  
*str var*<sub>2</sub> and *str var*<sub>3</sub> must indeed be strings, as this function does not convert their  
contents to a string.

---

\str\_if\_exist\_p:N \* \str\_if\_exist\_p:N *str var*  
\str\_if\_exist\_p:c \* \str\_if\_exist:NTF *str var* {{true code}} {{false code}}  
\str\_if\_exist:NTF \* Tests whether the *str var* is currently defined. This does not check that the *str var*  
\str\_if\_exist:cTF \* really is a string.  
New: 2015-09-18

---

## 16.2 Adding data to string variables

---

\str\_set:Nn \str\_set:(NV|Nx|cn|cV|cx) \str\_gset:Nn \str\_gset:(NV|Nx|cn|cV|cx)

Converts the *token list* to a *string*, and stores the result in *str var*.  
New: 2015-09-18  
Updated: 2018-07-28

---

\str\_put\_left:Nn \str\_put\_left:(NV|Nx|cn|cV|cx) \str\_gput\_left:Nn \str\_gput\_left:(NV|Nx|cn|cV|cx)

Converts the *token list* to a *string*, and prepends the result to *str var*. The current contents of the *str var* are not automatically converted to a string.  
New: 2015-09-18  
Updated: 2018-07-28

---

\str\_put\_right:Nn \str\_put\_right:(NV|Nx|cn|cV|cx) \str\_gput\_right:Nn \str\_gput\_right:(NV|Nx|cn|cV|cx)

Converts the *token list* to a *string*, and appends the result to *str var*. The current contents of the *str var* are not automatically converted to a string.  
New: 2015-09-18  
Updated: 2018-07-28

---

## 16.3 String conditionals

---

\str\_if\_empty\_p:N \* \str\_if\_empty\_p:N (str var)  
\str\_if\_empty\_p:c \* \str\_if\_empty:NTF (str var) {<true code>} {<false code>}  
\str\_if\_empty:NTF \* Tests if the *string variable* is entirely empty (*i.e.* contains no characters at all).  
\str\_if\_empty:cTF \*  
\str\_if\_empty\_p:n \*  
\str\_if\_empty:nTF \*

New: 2015-09-18  
Updated: 2022-03-21

---

\str\_if\_eq\_p:NN \* \str\_if\_eq\_p:NN (str var<sub>1</sub>) (str var<sub>2</sub>)  
\str\_if\_eq\_p:(Nc|cN|cc) \* \str\_if\_eq:NNTF (str var<sub>1</sub>) (str var<sub>2</sub>) {<true code>} {<false code>}  
\str\_if\_eq:NNTF \* Compares the content of two *str variables* and is logically **true** if the two contain the same characters in the same order. See \tl\_if\_eq:NNTF to compare tokens (including their category codes) rather than characters.  
New: 2015-09-18

---

---

```
\str_if_eq_p:nn          * \str_if_eq_p:nn {⟨tl1⟩} {⟨tl2⟩}
\str_if_eq_p:(Vn|on|no|nV|VV|vn|nv|ee) * \str_if_eq:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
\str_if_eq:nnTF          *
\str_if_eq:(Vn|on|no|nV|VV|vn|nv|ee)TF *
```

---

Updated: 2018-06-18

Compares the two *⟨token lists⟩* on a character by character basis (namely after converting them to strings), and is **true** if the two *⟨strings⟩* contain the same characters in the same order. Thus for example

```
\str_if_eq_p:no { abc } { \tl_to_str:n { abc } }
```

is logically **true**. See `\tl_if_eq:nnTF` to compare tokens (including their category codes) rather than characters.

---

```
\str_if_in:NnTF \str_if_in:NnTF <str var> {⟨token list⟩} {⟨true code⟩} {⟨false code⟩}
```

---

`\str_if_in:cnTF` Converts the *⟨token list⟩* to a *⟨string⟩* and tests if that *⟨string⟩* is found in the content of  
New: 2017-10-08 the *⟨str var⟩*.

---

---

```
\str_if_in:nnTF \str_if_in:nnTF {⟨tl1⟩} {⟨tl2⟩} {⟨true code⟩} {⟨false code⟩}
```

---

New: 2017-10-08 Converts both *⟨token lists⟩* to *⟨strings⟩* and tests whether *⟨string<sub>2</sub>⟩* is found inside  
⟨string<sub>1</sub>⟩.

---

```
\str_case:nn          * \str_case:nnTF {⟨test string⟩}
\str_case:(Vn|on|nV|nv) * {
  {⟨string case1⟩} {⟨code case1⟩}
\str_case:nnTF          *   {⟨string case2⟩} {⟨code case2⟩}
\str_case:(Vn|on|nV|nv)TF * ...
\str_case:Nn            *   {⟨string casen⟩} {⟨code casen⟩}
\str_case:NnTF          *
```

---

New: 2013-07-24  
Updated: 2022-03-21

Compares the *⟨test string⟩* in turn with each of the *⟨string cases⟩* (all token lists are converted to strings). If the two are equal (as described for `\str_if_eq:nnTF`) then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\str_case:nn`, which does nothing if there is no match, is also available.

This set of functions performs no expansion on each *⟨string case⟩* argument, so any variable in there will be compared as a string. If expansion is needed in the *⟨string cases⟩*, then `\str_case_e:nn(TF)` should be used instead.

---

```
\str_case_e:nn    * \str_case_e:nnTF {{test string}}
\str_case_e:nnTF *  {
  New: 2018-06-19
    {<string case1>} {{code case1}}
    {<string case2>} {{code case2}}
    ...
    {<string casen>} {{code casen}}
}
{{true code}}
{{false code}}
```

---

Compares the full expansion of the *<test string>* in turn with the full expansion of the *<string cases>* (all token lists are converted to strings). If the two full expansions are equal (as described for `\str_if_eq:nnTF`) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The function `\str_case_e:nn`, which does nothing if there is no match, is also available. The *<test string>* is expanded in each comparison, and must always yield the same result: for example, random numbers must not be used within this string.

---

```
\str_compare_p:nNn * \str_compare_p:nNn {{t1}} <relation> {{t2}}
\str_compare_p:eNe * \str_compare:nNnTF {{t1}} <relation> {{t2}} {{true code}} {{false code}}
\str_compare:nNnTF * Compares the two <token lists> on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The <relation> can be <, =, or > and the test is true under the following conditions:
New: 2021-05-17
```

---

- for <, if the first string is earlier than the second in lexicographic order;
- for =, if the two strings have exactly the same characters;
- for >, if the first string is later than the second in lexicographic order.

Thus for example the following is logically **true**:

```
\str_compare_p:nNn { ab } < { abc }
```

**TEXhackers note:** This is a wrapper around the `\(pdf)strcmp`. It is meant for programming and not for sorting textual contents, as it simply considers character codes and not more elaborate considerations of grapheme clusters, locale, etc.

## 16.4 Mapping over strings

All mappings are done at the current group level, *i.e.* any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

---

```
\str_map_function:nN ☆ \str_map_function:nN {{token list}} <function>
\str_map_function:NN ☆ \str_map_function:NN <str var> <function>
\str_map_function:cN ☆ Converts the <token list> to a <string> then applies <function> to every <character> in the <string> including spaces.
New: 2017-11-14
```

---

---

\str\_map\_inline:nn \str\_map\_inline:nn {⟨token list⟩} {⟨inline function⟩}  
\str\_map\_inline:Nn \str\_map\_inline:Nn ⟨str var⟩ {⟨inline function⟩}  
\str\_map\_inline:cN Converts the ⟨token list⟩ to a ⟨string⟩ then applies the ⟨inline function⟩ to every ⟨character⟩ in the ⟨str var⟩ including spaces. The ⟨inline function⟩ should consist of code which receives the ⟨character⟩ as #1.  
New: 2017-11-14

---

\str\_map\_tokens:nn ☆ \str\_map\_tokens:nn {⟨token list⟩} {⟨code⟩}  
\str\_map\_tokens:Nn ☆ \str\_map\_tokens:Nn ⟨str var⟩ {⟨code⟩}  
\str\_map\_tokens:cN Converts the ⟨token list⟩ to a ⟨string⟩ then applies ⟨code⟩ to every ⟨character⟩ in the ⟨string⟩ including spaces. The ⟨code⟩ receives each character as a trailing brace group. This is equivalent to \str\_map\_function:nN if the ⟨code⟩ consists of a single function.  
New: 2021-05-05

---

\str\_map\_variable:nNn \str\_map\_variable:nNn {⟨token list⟩} ⟨variable⟩ {⟨code⟩}  
\str\_map\_variable:NNn \str\_map\_variable:NNn ⟨str var⟩ ⟨variable⟩ {⟨code⟩}  
\str\_map\_variable:cNn Converts the ⟨token list⟩ to a ⟨string⟩ then stores each ⟨character⟩ in the ⟨string⟩ (including spaces) in turn in the (string or token list) ⟨variable⟩ and applies the ⟨code⟩. The ⟨code⟩ will usually make use of the ⟨variable⟩, but this is not enforced. The assignments to the ⟨variable⟩ are local. Its value after the loop is the last ⟨character⟩ in the ⟨string⟩, or its original value if the ⟨string⟩ is empty. See also \str\_map\_inline:Nn.  
New: 2017-11-14

---

\str\_map\_break: ☆ \str\_map\_break:  
New: 2017-10-08 Used to terminate a \str\_map\_... function before all characters in the ⟨string⟩ have been processed. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
    \str_if_eq:nnT {#1} {bingo} { \str_map_break: }
    % Do something useful
}
```

See also \str\_map\_break:n. Use outside of a \str\_map\_... scenario leads to low level TeX errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before continuing with the code that follows the loop. This depends on the design of the mapping function.

---

`\str_map_break:n`  $\star$  `\str_map_break:n {<code>}`

New: 2017-10-08 Used to terminate a `\str_map_...` function before all characters in the  $\langle string \rangle$  have been processed, inserting the  $\langle code \rangle$  after the mapping has ended. This normally takes place within a conditional statement, for example

```
\str_map_inline:Nn \l_my_str
{
    \str_if_eq:nnT { #1 } { bingo }
        { \str_map_break:n { <code> } }
    % Do something useful
}
```

Use outside of a `\str_map_...` scenario leads to low level T<sub>E</sub>X errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before the  $\langle code \rangle$  is inserted into the input stream. This depends on the design of the mapping function.

## 16.5 Working with the content of strings

---

`\str_use:N`  $\star$  `\str_use:N {str var}`

New: 2015-09-18 Recovers the content of a  $\langle str var \rangle$  and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a  $\langle str \rangle$  directly without an accessor function.

---

`\str_count:N`  $\star$  `\str_count:n {<token list>}`

`\str_count:c`  $\star$

`\str_count:n`  $\star$

`\str_count_ignore_spaces:n`  $\star$

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of  $\langle token list \rangle$ , as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_count:N` and `\str_count:n`, all characters including spaces are counted. The `\str_count_ignore_spaces:n` function leaves the number of non-space characters in the input stream.

---

`\str_count_spaces:N`  $\star$  `\str_count_spaces:n {<token list>}`

New: 2015-09-18 `\str_count_spaces:c`  $\star$  Leaves in the input stream the number of space characters in the string representation of  $\langle token list \rangle$ , as an integer denotation. Of course, this function has no `_ignore_spaces` variant.

---

```
\str_head:N          * \str_head:n {\<token list>}
\str_head:c          *
\str_head:n          *
\str_head_ignore_spaces:n *
```

---

New: 2015-09-18

Converts the  $\langle token\ list\rangle$  into a  $\langle string\rangle$ . The first character in the  $\langle string\rangle$  is then left in the input stream, with category code “other”. The functions differ if the first character is a space: `\str_head:N` and `\str_head:n` return a space token with category code 10 (blank space), while the `\str_head_ignore_spaces:n` function ignores this space character and leaves the first non-space character in the input stream. If the  $\langle string\rangle$  is empty (or only contains spaces in the case of the `_ignore_spaces` function), then nothing is left on the input stream.

---

```
\str_tail:N          * \str_tail:n {\<token list>}
\str_tail:c          *
\str_tail:n          *
\str_tail_ignore_spaces:n *
```

---

New: 2015-09-18

Converts the  $\langle token\ list\rangle$  to a  $\langle string\rangle$ , removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` only trim that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the  $\langle token\ list\rangle$  is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

---

```
\str_item:Nn         * \str_item:nn {\<token list>} {\<integer expression>}
\str_item:nn         *
\str_item_ignore_spaces:nn *
```

---

New: 2015-09-18

Converts the  $\langle token\ list\rangle$  to a  $\langle string\rangle$ , and leaves in the input stream the character in position  $\langle integer\ expression\rangle$  of the  $\langle string\rangle$ , starting at 1 for the first (left-most) character. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces when counting characters. If the  $\langle integer\ expression\rangle$  is negative, characters are counted from the end of the  $\langle string\rangle$ . Hence,  $-1$  is the right-most character, etc.

---

```

\str_range:Nnn          * \str_range:nnn {\langle token list\rangle} {\langle start index\rangle} {\langle end index\rangle}
\str_range:cnn          *
\str_range:nnn          *
\str_range_ignore_spaces:nnn *

```

---

New: 2015-09-18

---

Converts the  $\langle token\ list\rangle$  to a  $\langle string\rangle$ , and leaves in the input stream the characters from the  $\langle start\ index\rangle$  to the  $\langle end\ index\rangle$  inclusive. Spaces are preserved and counted as items (contrast this with  $\backslash tl\_range:nnn$  where spaces are not counted as items and are possibly discarded from the output).

Here  $\langle start\ index\rangle$  and  $\langle end\ index\rangle$  should be integer denotations. For describing in detail the functions' behavior, let  $m$  and  $n$  be the start and end index respectively. If either is 0, the result is empty. A positive index means 'start counting from the left end', a negative index means 'start counting from the right end'. Let  $l$  be the count of the token list.

The *actual start point* is determined as  $M = m$  if  $m > 0$  and as  $M = l + m + 1$  if  $m < 0$ . Similarly the *actual end point* is  $N = n$  if  $n > 0$  and  $N = l + n + 1$  if  $n < 0$ . If  $M > N$ , the result is empty. Otherwise it consists of all items from position  $M$  to position  $N$  inclusive; for the purpose of this rule, we can imagine that the token list extends at infinity on either side, with void items at positions  $s$  for  $s \leq 0$  or  $s > l$ . For instance,

```

\iow_term:x { \str_range:nnn { abcdef } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdef } { -4 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { -2 } { -1 } }
\iow_term:x { \str_range:nnn { abcdef } { 0 } { -1 } }

```

prints **bcd**, **cde**, **ef**, and an empty line to the terminal. The  $\langle start\ index\rangle$  must always be smaller than or equal to the  $\langle end\ index\rangle$ : if this is not the case then no output is generated. Thus

```

\iow_term:x { \str_range:nnn { abcdef } { 5 } { 2 } }
\iow_term:x { \str_range:nnn { abcdef } { -1 } { -4 } }

```

both yield empty strings.

The behavior of  $\backslash str\_range\_ignore\_spaces:nnn$  is similar, but spaces are removed before starting the job. The input

```

\iow_term:x { \str_range:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abcdefg } { -6 } { -3 } }

\iow_term:x { \str_range:nnn { abc~efg } { 2 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { 2 } { -3 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { 5 } }
\iow_term:x { \str_range:nnn { abc~efg } { -6 } { -3 } }

\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcdefg } { -6 } { -3 } }

```

```
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { 2 } { -3 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { 5 } }
\iow_term:x { \str_range_ignore_spaces:nnn { abcd~efg } { -6 } { -3 } }
```

will print four instances of `bcde`, four instances of `bc e` and eight instances of `bcde`.

## 16.6 Modifying string variables

---

`\str_replace_once:Nnn`  
`\str_replace_once:cnn`  
`\str_greplace_once:Nnn`  
`\str_greplace_once:cnn`

`\str_replace_once:Nnn <str var> {<old>} {<new>}`

Converts the `<old>` and `<new>` token lists to strings, then replaces the first (leftmost) occurrence of `<old string>` in the `<str var>` with `<new string>`.

New: 2017-10-08

---



---

`\str_replace_all:Nnn`  
`\str_replace_all:cnn`  
`\str_greplace_all:Nnn`  
`\str_greplace_all:cnn`

`\str_replace_all:Nnn <str var> {<old>} {<new>}`

Converts the `<old>` and `<new>` token lists to strings, then replaces all occurrences of `<old string>` in the `<str var>` with `<new string>`. As this function operates from left to right, the pattern `<old string>` may remain after the replacement (see `\str_remove_all:Nn` for an example).

New: 2017-10-08

---



---

`\str_remove_once:Nn`  
`\str_remove_once:cn`  
`\str_gremove_once:Nn`  
`\str_gremove_once:cn`

`\str_remove_once:Nn <str var> {<token list>}`

Converts the `<token list>` to a `<string>` then removes the first (leftmost) occurrence of `<string>` from the `<str var>`.

New: 2017-10-08

---



---

`\str_remove_all:Nn`  
`\str_remove_all:cn`  
`\str_gremove_all:Nn`  
`\str_gremove_all:cn`

`\str_remove_all:Nn <str var> {<token list>}`

Converts the `<token list>` to a `<string>` then removes all occurrences of `<string>` from the `<str var>`. As this function operates from left to right, the pattern `<string>` may remain after the removal, for instance,

New: 2017-10-08

---

`\str_set:Nn \l_tmpa_str {abbccd} \str_remove_all:Nn \l_tmpa_str {bc}`

results in `\l_tmpa_str` containing `abcd`.

## 16.7 String manipulation

---

```
\str_lowercase:n * \str_lowercase:n {\langle tokens\rangle}
\str_lowercase:f * \str_uppercase:n {\langle tokens\rangle}
\str_uppercase:n * Converts the input \langle tokens\rangle to their string representation, as described for \tl_to_-
\str_uppercase:f * str:n, and then to the lower or upper case representation using a one-to-one mapping
New: 2019-11-26 as described by the Unicode Consortium file UnicodeData.txt.
```

---

These functions are intended for case changing programmatic data in places where upper/lower case distinctions are meaningful. One example would be automatically generating a function name from user input where some case changing is needed. In this situation the input is programmatic, not textual, case does have meaning and a language-independent one-to-one mapping is appropriate. For example

```
\cs_new_protected:Npn \myfunc:nn #1#2
{
    \cs_set_protected:cpx
    {
        user
        \str_uppercase:f { \tl_head:n {\#1} }
        \str_lowercase:f { \tl_tail:n {\#1} }
    }
    { #2 }
}
```

would be used to generate a function with an auto-generated name consisting of the upper case equivalent of the supplied name followed by the lower case equivalent of the rest of the input.

These functions should *not* be used for

- Caseless comparisons: use `\str_casfold:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\text_lowercase:n(n)`, `\text_uppercase:n(n)` and `\text_titlecase:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

---

`\str_casefold:n` \* `\str_casefold:n {⟨tokens⟩}`

`\str_casefold:v` \* Converts the input `⟨tokens⟩` to their string representation, as described for `\tl_to_str:n`,

New: 2022-10-16 and then folds the case of the resulting `⟨string⟩` to remove case information. The result of this process is left in the input stream.

String folding is a process used for material such as identifiers rather than for “text”. The folding provided by `\str_casefold:n` follows the mappings provided by the [Unicode Consortium](#), who [state](#):

Case folding is primarily used for caseless comparison of text, such as identifiers in a computer program, rather than actual text transformation. Case folding in Unicode is based on the lowercase mapping, but includes additional changes to the source text to help make it language-insensitive and consistent. As a result, case-folded text should be used solely for internal processing and generally should not be stored or displayed to the end user.

The folding approach implemented by `\str_casefold:n` follows the “full” scheme defined by the Unicode Consortium (*e.g.* SSfolds to SS). As case-folding is a language-insensitive process, there is no special treatment of Turkic input (*i.e.* I always folds to i and not to ı).

## 16.8 Viewing strings

---

`\str_show:N`      `\str_show:N ⟨str var⟩`

`\str_show:c`      Displays the content of the `⟨str var⟩` on the terminal.

`\str_show:n`

New: 2015-09-18

Updated: 2021-04-29

---

---

`\str_log:N`      `\str_log:N ⟨str var⟩`

`\str_log:c`      Writes the content of the `⟨str var⟩` in the log file.

`\str_log:n`

New: 2019-02-15

Updated: 2021-04-29

---

## 16.9 Constant strings

---

\c\_ampersand\_str Constant strings, containing a single character token, with category code 12.  
\c\_atsign\_str  
\c\_backslash\_str  
\c\_left\_brace\_str  
\c\_right\_brace\_str  
\c\_circumflex\_str  
\c\_colon\_str  
\c\_dollar\_str  
\c\_hash\_str  
\c\_percent\_str  
\c\_tilde\_str  
\c\_underscore\_str  
\c\_zero\_str

---

New: 2015-09-19  
Updated: 2020-12-22

## 16.10 Scratch strings

---

\l\_tmpa\_str Scratch strings for local assignment. These are never used by the kernel code, and so  
\l\_tmpb\_str are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
other non-kernel code and so should only be used for short-term storage.

---

\g\_tmpa\_str Scratch strings for global assignment. These are never used by the kernel code, and so  
\g\_tmpb\_str are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
other non-kernel code and so should only be used for short-term storage.

## 16.11 Deprecated functions

---

\str\_foldcase:n \* \str\_foldcase:n {\langle tokens \rangle}  
\str\_foldcase:V \* A previous name for the functionally-identical \str\_casefold:n.  
New: 2019-11-26

---

# Chapter 17

## The `I3str-convert` package: string encoding conversions

### 17.1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.<sup>7</sup>
- Bytes are translated to `TEx` tokens through a given “escaping”. Those are defined for the most part by the `pdf` file format. See Table 2 for a list of escaping methods supported.<sup>7</sup>

---

<sup>7</sup>Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

$\langle Encoding \rangle$	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
<code>clist</code>	comma-list of integers
<code>{empty}</code>	native (Unicode) string
<code>default</code>	like <code>utf8</code> with 8-bit engines, and like native with unicode-engines

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

$\langle Escaping \rangle$	description
<code>bytes, or empty</code>	arbitrary bytes
<code>hex, hexadecimal</code>	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescape name</code>
<code>string</code>	see <code>\pdfescape string</code>
<code>url</code>	encoding used in URLs

## 17.2 Conversion functions

---

```
\str_set_convert:Nnnn \str_set_convert:Nnnn <str var> {<string>} {<name 1>} {<name 2>}
```

---

```
\str_gset_convert:Nnnn
```

This function converts the *<string>* from the encoding given by *<name 1>* to the encoding given by *<name 2>*, and stores the result in the *<str var>*. Each *<name>* can have the form *<encoding>* or *<encoding>/<escaping>*, where the possible values of *<encoding>* and *<escaping>* are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty *<name>* indicates the use of “native” strings, 8-bit for pdfTEX, and Unicode strings for the other two engines.

For example,

```
\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }
```

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark “`FEFF`”, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the *<string>* is not valid according to the *<escaping 1>* and *<encoding 1>*, or if it cannot be reencoded in the *<encoding 2>* and *<escaping 2>* (for instance, if a character does not exist in the *<encoding 2>*). Erroneous input is replaced by the Unicode replacement character “`FFFD`”, and characters which cannot be reencoded are replaced by either the replacement character “`FFFD`” if it exists in the *<encoding 2>*, or an encoding-specific replacement character, or the question mark character.

---

```
\str_set_convert:NnnnTF \str_set_convert:NnnnTF <str var> {<string>} {<name 1>} {<name 2>} {<true code>}
```

---

```
\str_gset_convert:NnnnTF {<false code>}
```

As `\str_set_convert:Nnnn`, converts the *<string>* from the encoding given by *<name 1>* to the encoding given by *<name 2>*, and assigns the result to *<str var>*. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the *<string>* is not valid according to the *<name 1>* encoding, or cannot be expressed in the *<name 2>* encoding. Instead, the *<false code>* is performed.

## 17.3 Conversion by expansion (for PDF contexts)

A small number of expandable functions are provided for use in PDF string/name contexts. These *assume UTF-8* and *no escaping* in the input.

---

```
\str_convert_pdfname:n * \str_convert_pdfname:n <string>
```

As `\str_set_convert:Nnnn`, converts the *<string>* on a byte-by-byte basis with non-ASCII codepoints escaped using hashes.

## 17.4 Possibilities, and things to do

Encoding/escaping-related tasks.

- In X<sub>H</sub>T<sub>E</sub>X/LuaT<sub>E</sub>X, would it be better to use the `^~~~....` approach to build a string from a given list of character codes? Namely, within a group, assign `0-9a-f` and all characters we want to category “other”, then assign `^` the category superscript, and use `\scantokens`.
- Change `\str_set_convert:Nnnn` to expand its last two arguments.
- Describe the internal format in the code comments. Refuse code points in `["D800, "DFFF]` in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps the characters `! ' ()*-./0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko’s `stringenc`). CESU?
- More escapings: ASCII85, shell escapes, lua escapes, *etc.*?

# Chapter 18

## The **I3quark** package Quarks

Two special types of constants in I<sup>A</sup>T<sub>E</sub>X3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

### 18.1 Quarks

Quarks are control sequences (and in fact, token lists) that expand to themselves and should therefore *never* be executed directly in the code. This would result in an endless loop!

They are meant to be used as delimiter in weird functions, the most common use case being the ‘stop token’ (*i.e.* `\q_stop`). For example, when writing a macro to parse a user-defined date

```
\date_parse:n {19/June/1981}
```

one might write a command such as

```
\cs_new:Npn \date_parse:n #1 { \date_parse_aux:w #1 \q_stop }
\cs_new:Npn \date_parse_aux:w #1 / #2 / #3 \q_stop
{ <do something with the date> }
```

Quarks are sometimes also used as error return values for functions that receive erroneous input. For example, in the function `\prop_get:NnN` to retrieve a value stored in some key of a property list, if the key does not exist then the return value is the quark `\q_no_value`. As mentioned above, such quarks are extremely fragile and it is imperative when using such functions that code is carefully written to check for pathological cases to avoid leakage of a quark into an uncontrolled environment.

Quarks also permit the following ingenious trick when parsing tokens: when you pick up a token in a temporary variable and you want to know whether you have picked up a particular quark, all you have to do is compare the temporary variable to the quark using `\tl_if_eq:NNTF`. A set of special quark testing functions is set up below. All the quark testing functions are expandable although the ones testing only single tokens are much faster.

## 18.2 Defining quarks

---

`\quark_new:N \quark_new:N <quark>`

Creates a new  $\langle \text{quark} \rangle$  which expands only to  $\langle \text{quark} \rangle$ . The  $\langle \text{quark} \rangle$  is defined globally, and an error message is raised if the name was already taken.

---

`\q_stop` Used as a marker for delimited arguments, such as

```
\cs_set:Npn \tmp:w #1#2 \q_stop {#1}
```

---

`\q_mark` Used as a marker for delimited arguments when `\q_stop` is already in use.

---

`\q_nil` Quark to mark a null value in structured variables or functions. Used as an end delimiter when this may itself need to be tested (in contrast to `\q_stop`, which is only ever used as a delimiter).

---

`\q_no_value` A canonical value for a missing value, when one is requested from a data structure. This is therefore used as a “return” value by functions such as `\prop_get:NnN` if there is no data to return.

## 18.3 Quark tests

The method used to define quarks means that the single token (N) tests are faster than the multi-token (n) tests. The latter should therefore only be used when the argument can definitely take more than a single token.

---

`\quark_if_nil_p:N * \quark_if_nil_p:N <token>`  
`\quark_if_nil:nTF * \quark_if_nil:nTF <token> {\<true code>} {\<false code>}`

Tests if the  $\langle \text{token} \rangle$  is equal to `\q_nil`.

---

`\quark_if_nil_p:n * \quark_if_nil_p:n {\<token list>}`  
`\quark_if_nil_p:(o|v) * \quark_if_nil:nTF {\<token list>} {\<true code>} {\<false code>}`  
`\quark_if_nil:nTF * Tests if the <token list> contains only \q_nil (distinct from <token list> being empty or`  
`\quark_if_nil:(o|v)TF * containing \q_nil plus one or more other tokens).`

---

`\quark_if_no_value_p:N * \quark_if_no_value_p:N <token>`  
`\quark_if_no_value_p:c * \quark_if_no_value:nTF <token> {\<true code>} {\<false code>}`  
`\quark_if_no_value:nTF * Tests if the <token> is equal to \q_no_value.`  
`\quark_if_no_value:cTF *`

---

`\quark_if_no_value_p:n * \quark_if_no_value_p:n {\<token list>}`  
`\quark_if_no_value:nTF * \quark_if_no_value:nTF {\<token list>} {\<true code>} {\<false code>}`

Tests if the  $\langle \text{token list} \rangle$  contains only `\q_no_value` (distinct from  $\langle \text{token list} \rangle$  being empty or containing `\q_no_value` plus one or more other tokens).

## 18.4 Recursion

This module provides a uniform interface to intercepting and terminating loops as when one is doing tail recursion. The building blocks follow below and an example is shown in Section 18.4.1.

---

**\q\_recursion\_tail** This quark is appended to the data structure in question and appears as a real element there. This means it gets any list separators around it.

---

**\q\_recursion\_stop** This quark is added *after* the data structure. Its purpose is to make it possible to terminate the recursion at any point easily.

---

**\quark\_if\_recursion\_tail\_stop:N** \* **\quark\_if\_recursion\_tail\_stop:N** *(token)*

Tests if *(token)* contains only the marker **\q\_recursion\_tail**, and if so uses **\use\_none\_delimit\_by\_q\_recursion\_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q\_recursion\_tail** and **\q\_recursion\_stop** as the last two items.

---

**\quark\_if\_recursion\_tail\_stop:n** \* **\quark\_if\_recursion\_tail\_stop:n** {*(token list)*}  
**\quark\_if\_recursion\_tail\_stop:o** \*

---

Updated: 2011-09-06

Tests if the *(token list)* contains only **\q\_recursion\_tail**, and if so uses **\use\_none\_delimit\_by\_q\_recursion\_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q\_recursion\_tail** and **\q\_recursion\_stop** as the last two items.

---

**\quark\_if\_recursion\_tail\_stop\_do:Nn** \* **\quark\_if\_recursion\_tail\_stop\_do:Nn** *(token)* {{*insertion*}}

Tests if *(token)* contains only the marker **\q\_recursion\_tail**, and if so uses **\use\_i\_delimit\_by\_q\_recursion\_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q\_recursion\_tail** and **\q\_recursion\_stop** as the last two items. The *(insertion)* code is then added to the input stream after the recursion has ended.

---

**\quark\_if\_recursion\_tail\_stop\_do:nn** \* **\quark\_if\_recursion\_tail\_stop\_do:nn** {*(token list)*} {{*insertion*}}  
**\quark\_if\_recursion\_tail\_stop\_do:on** \*

---

Updated: 2011-09-06

Tests if the *(token list)* contains only **\q\_recursion\_tail**, and if so uses **\use\_i\_delimit\_by\_q\_recursion\_stop:w** to terminate the recursion that this belongs to. The recursion input must include the marker tokens **\q\_recursion\_tail** and **\q\_recursion\_stop** as the last two items. The *(insertion)* code is then added to the input stream after the recursion has ended.

---

```
\quark_if_recursion_tail_break:NN * \quark_if_recursion_tail_break:nN {\langle token list\rangle}
\quark_if_recursion_tail_break:nN * \langle type\rangle_map_break:
```

---

New: 2018-04-10

Tests if *⟨token list⟩* contains only `\q_recursion_tail`, and if so terminates the recursion using `\⟨type⟩_map_break:`. The recursion end should be marked by `\prg_break_point:Nn \⟨type⟩_map_break:`.

#### 18.4.1 An example of recursion with quarks

Quarks are mainly used internally in the `expl3` code to define recursion functions such as `\tl_map_inline:nn` and so on. Here is a small example to demonstrate how to use quarks in this fashion. We shall define a command called `\my_map dbl:nn` which takes a token list and applies an operation to every *pair* of tokens. For example, `\my_map dbl:nn {abcd} {[--#1--#2--]~}` would produce “[*-a-b-*] [*-c-d-*]”. Using quarks to define such functions simplifies their logic and ensures robustness in many cases.

Here’s the definition of `\my_map dbl:nn`. First of all, define the function that does the processing based on the inline function argument #2. Then initiate the recursion using an internal function. The token list #1 is terminated using `\q_recursion_tail`, with delimiters according to the type of recursion (here a pair of `\q_recursion_tail`), concluding with `\q_recursion_stop`. These quarks are used to mark the end of the token list being operated upon.

```
\cs_new:Npn \my_map dbl:nn #1#2
{
    \cs_set:Npn \__my_map dbl_fn:nn ##1 ##2 {\#2}
    \__my_map dbl:nn #1 \q_recursion_tail \q_recursion_tail
    \q_recursion_stop
}
```

The definition of the internal recursion function follows. First check if either of the input tokens are the termination quarks. Then, if not, apply the inline function to the two arguments.

```
\cs_new:Nn \__my_map dbl:nn
{
    \quark_if_recursion_tail_stop:n {\#1}
    \quark_if_recursion_tail_stop:n {\#2}
    \__my_map dbl_fn:nn {\#1} {\#2}
```

Finally, recurse:

```
\__my_map dbl:nn
```

Note that contrarily to L<sup>A</sup>T<sub>E</sub>X3 built-in mapping functions, this mapping function cannot be nested, since the second map would overwrite the definition of `\__my_map dbl_fn:nn`.

## 18.5 Scan marks

Scan marks are control sequences set equal to `\scan_stop:`, hence never expand in an expansion context and are (largely) invisible if they are encountered in a typesetting context.

Like quarks, they can be used as delimiters in weird functions and are often safer to use for this purpose. Since they are harmless when executed by TeX in non-expandable contexts, they can be used to mark the end of a set of instructions. This allows to skip to that point if the end of the instructions should not be performed (see `\3regex`).

---

`\scan_new:N` `\scan_new:N <scan mark>`

New: 2018-04-01 Creates a new `<scan mark>` which is set equal to `\scan_stop:`. The `<scan mark>` is defined globally, and an error message is raised if the name was already taken by another scan mark.

---

`\s_stop` Used at the end of a set of instructions, as a marker that can be jumped to using `\use_-none_delimit_by_s_stop:w`.

New: 2018-04-01

---

`\use_none_delimit_by_s_stop:w * \use_none_delimit_by_s_stop:w <tokens> \s_stop`

New: 2018-04-01

Removes the `<tokens>` and `\s_stop` from the input stream. This leads to a low-level TeX error if `\s_stop` is absent.

# Chapter 19

## The `\seq` package Sequences and stacks

L<sup>A</sup>T<sub>E</sub>X3 implements a “sequence” data type, which contain an ordered list of entries which may contain any *<balanced text>*. It is possible to map functions to sequences such that the function is applied to every item in the sequence.

Sequences are also used to implement stack functions in L<sup>A</sup>T<sub>E</sub>X3. This is achieved using a number of dedicated stack functions.

### 19.1 Creating and initialising sequences

---

`\seq_new:N \seq_new:N <sequence>`

`\seq_new:c` Creates a new *<sequence>* or raises an error if the name is already taken. The declaration is global. The *<sequence>* initially contains no items.

---

`\seq_clear:N \seq_clear:N <sequence>`

`\seq_clear:c` Clears all items from the *<sequence>*.  
`\seq_gclear:N`  
`\seq_gclear:c`

---

`\seq_clear_new:N \seq_clear_new:N <sequence>`

`\seq_clear_new:c`  
`\seq_gclear_new:N`  
`\seq_gclear_new:c` Ensures that the *<sequence>* exists globally by applying `\seq_new:N` if necessary, then applies `\seq_(g)clear:N` to leave the *<sequence>* empty.

---

`\seq_set_eq:NN \seq_set_eq:NN <sequence1> <sequence2>`

`\seq_set_eq:(cN|Nc|cc)` Sets the content of *<sequence<sub>1</sub>>* equal to that of *<sequence<sub>2</sub>>*.

`\seq_gset_eq:NN`

`\seq_gset_eq:(cN|Nc|cc)`

---

```
\seq_set_from_clist:NN          \seq_set_from_clist:NN <sequence> <comma-list>
\seq_set_from_clist:(cN|Nc|cc)
\seq_set_from_clist:Nn
\seq_set_from_clist:cn
\seq_gset_from_clist:NN
\seq_gset_from_clist:(cN|Nc|cc)
\seq_gset_from_clist:Nn
\seq_gset_from_clist:cn
```

---

New: 2014-07-17

Converts the data in the *<comma list>* into a *<sequence>*: the original *<comma list>* is unchanged.

---

```
\seq_const_from_clist:Nn \seq_const_from_clist:Nn <seq var> {{<comma-list>}}
```

---

\seq\_const\_from\_clist:cn Creates a new constant *<seq var>* or raises an error if the name is already taken. The *<seq var>* is set globally to contain the items in the *<comma list>*.

---

```
\seq_set_split:Nnn \seq_set_split:Nnn <sequence> {{<delimiter>}} {{<token list>}}
```

---

\seq\_set\_split:NnV  
\seq\_gset\_split:Nnn  
\seq\_gset\_split:NnV

Splits the *<token list>* into *<items>* separated by *<delimiter>*, and assigns the result to the *<sequence>*. Spaces on both sides of each *<item>* are ignored, then one set of outer braces is removed (if any); this space trimming behaviour is identical to that of `I3clist` functions.

New: 2011-08-15

Updated: 2012-07-02

Empty *<items>* are preserved by `\seq_set_split:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <sequence> {}`. The *<delimiter>* may not contain {, } or # (assuming TeX's normal category code régime). If the *<delimiter>* is empty, the *<token list>* is split into *<items>* as a *<token list>*. See also `\seq_set_split_keep_spaces:Nnn`, which omits space stripping.

---

```
\seq_set_split_keep_spaces:Nnn \seq_set_split_keep_spaces:Nnn <sequence> {{<delimiter>}} {{<token list>}}
```

---

```
\seq_set_split_keep_spaces:NnV
```

---

```
\seq_gset_split_keep_spaces:Nnn
```

---

```
\seq_gset_split_keep_spaces:NnV
```

---

New: 2021-03-24

Splits the *<token list>* into *<items>* separated by *<delimiter>*, and assigns the result to the *<sequence>*. One set of outer braces is removed (if any) but any surrounding spaces are retained: any braces *inside* one or more spaces are therefore kept. Empty *<items>* are preserved by `\seq_set_split_keep_spaces:Nnn`, and can be removed afterwards using `\seq_remove_all:Nn <sequence> {}`. The *<delimiter>* may not contain {, } or # (assuming TeX's normal category code régime). If the *<delimiter>* is empty, the *<token list>* is split into *<items>* as a *<token list>*. See also `\seq_set_split:Nnn`, which removes spaces around the delimiters.

---

```
\seq_concat:NNN \seq_concat:NNN <sequence1> <sequence2> <sequence3>
```

---

```
\seq_concat:ccc
```

---

```
\seq_gconcat:NNN
```

---

```
\seq_gconcat:ccc
```

Concatenates the content of *<sequence<sub>2</sub>>* and *<sequence<sub>3</sub>>* together and saves the result in *<sequence<sub>1</sub>>*. The items in *<sequence<sub>2</sub>>* are placed at the left side of the new sequence.

---

\seq\_if\_exist\_p:N \* \seq\_if\_exist\_p:N *sequence*  
\seq\_if\_exist\_p:c \* \seq\_if\_exist:NTF *sequence* {\i{true code}} {\i{false code}}  
\seq\_if\_exist:NTF \* Tests whether the *sequence* is currently defined. This does not check that the *sequence* really is a sequence variable.

---

New: 2012-03-03

## 19.2 Appending data to sequences

---

\seq\_put\_left:Nn \seq\_put\_left:Nn *sequence* {\i{item}}  
\seq\_put\_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)  
\seq\_gput\_left:Nn  
\seq\_gput\_left:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

---

Appends the *item* to the left of the *sequence*.

---

\seq\_put\_right:Nn \seq\_put\_right:Nn *sequence* {\i{item}}  
\seq\_put\_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)  
\seq\_gput\_right:Nn  
\seq\_gput\_right:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

---

Appends the *item* to the right of the *sequence*.

## 19.3 Recovering items from sequences

Items can be recovered from either the left or the right of sequences. For implementation reasons, the actions at the left of the sequence are faster than those acting on the right. These functions all assign the recovered material locally, *i.e.* setting the *token list variable* used with \tl\_set:Nn and *never* \tl\_gset:Nn.

---

\seq\_get\_left>NN \seq\_get\_left>NN *sequence* <token list variable>  
\seq\_get\_left:cN Stores the left-most item from a *sequence* in the *token list variable* without removing it from the *sequence*. The *token list variable* is assigned locally. If *sequence* is empty the *token list variable* is set to the special marker \q\_no\_value.  
Updated: 2012-05-14

---

\seq\_get\_right>NN \seq\_get\_right>NN *sequence* <token list variable>  
\seq\_get\_right:cN Stores the right-most item from a *sequence* in the *token list variable* without removing it from the *sequence*. The *token list variable* is assigned locally. If *sequence* is empty the *token list variable* is set to the special marker \q\_no\_value.  
Updated: 2012-05-19

---

\seq\_pop\_left>NN \seq\_pop\_left>NN *sequence* <token list variable>  
\seq\_pop\_left:cN Pops the left-most item from a *sequence* into the *token list variable*, *i.e.* removes the item from the sequence and stores it in the *token list variable*. Both of the variables are assigned locally. If *sequence* is empty the *token list variable* is set to the special marker \q\_no\_value.  
Updated: 2012-05-14

---

`\seq_gpop_left:NN` `\seq_gpop_left:NN` *(sequence)* *(token list variable)*  
`\seq_gpop_left:cN` Pops the left-most item from a *(sequence)* into the *(token list variable)*, i.e. removes  
Updated: 2012-05-14 the item from the sequence and stores it in the *(token list variable)*. The *(sequence)* is modified globally, while the assignment of the *(token list variable)* is local. If *(sequence)* is empty the *(token list variable)* is set to the special marker `\q_no_value`.

---

`\seq_pop_right:NN` `\seq_pop_right:NN` *(sequence)* *(token list variable)*  
`\seq_pop_right:cN` Pops the right-most item from a *(sequence)* into the *(token list variable)*, i.e. removes the  
Updated: 2012-05-19 item from the sequence and stores it in the *(token list variable)*. Both of the variables are assigned locally. If *(sequence)* is empty the *(token list variable)* is set to the special marker `\q_no_value`.

---

`\seq_gpop_right:NN` `\seq_gpop_right:NN` *(sequence)* *(token list variable)*  
`\seq_gpop_right:cN` Pops the right-most item from a *(sequence)* into the *(token list variable)*, i.e. removes  
Updated: 2012-05-19 the item from the sequence and stores it in the *(token list variable)*. The *(sequence)* is modified globally, while the assignment of the *(token list variable)* is local. If *(sequence)* is empty the *(token list variable)* is set to the special marker `\q_no_value`.

---

`\seq_item:Nn *` `\seq_item:Nn` *(sequence)* {*integer expression*}  
`\seq_item:cN *` Indexing items in the *(sequence)* from 1 at the top (left), this function evaluates the  
New: 2014-07-17 *(integer expression)* and leaves the appropriate item from the sequence in the input stream. If the *(integer expression)* is negative, indexing occurs from the bottom (right) of the sequence. If the *(integer expression)* is larger than the number of items in the *(sequence)* (as calculated by `\seq_count:N`) then the function expands to nothing.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *(item)* does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

`\seq_rand_item:N *` `\seq_rand_item:N` *(seq var)*  
`\seq_rand_item:c *` Selects a pseudo-random item of the *(sequence)*. If the *(sequence)* is empty the result is empty. This is not available in older versions of X<sub>H</sub>TeX.  
New: 2016-12-06

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *(item)* does not expand further when appearing in an `x`-type or `e`-type argument expansion.

## 19.4 Recovering values from sequences with branching

The functions in this section combine tests for non-empty sequences with recovery of an item from the sequence. They offer increased readability and performance over separate testing and recovery phases.

<code>\seq_get_left:NNTF</code>	<code>\seq_get_left:NNTF &lt;sequence&gt; &lt;token list variable&gt; {{true code}} {{false code}}</code>
<code>\seq_get_left:CNTF</code>	If the <i>&lt;sequence&gt;</i> is empty, leaves the <i>&lt;false code&gt;</i> in the input stream. The value of the <i>&lt;token list variable&gt;</i> is not defined in this case and should not be relied upon. If the <i>&lt;sequence&gt;</i> is non-empty, stores the left-most item from the <i>&lt;sequence&gt;</i> in the <i>&lt;token list variable&gt;</i> without removing it from the <i>&lt;sequence&gt;</i> , then leaves the <i>&lt;true code&gt;</i> in the input stream. The <i>&lt;token list variable&gt;</i> is assigned locally.
<small>New: 2012-05-14</small>	
<small>Updated: 2012-05-19</small>	
<code>\seq_get_right:NNTF</code>	<code>\seq_get_right:NNTF &lt;sequence&gt; &lt;token list variable&gt; {{true code}} {{false code}}</code>
<code>\seq_get_right:CNTF</code>	If the <i>&lt;sequence&gt;</i> is empty, leaves the <i>&lt;false code&gt;</i> in the input stream. The value of the <i>&lt;token list variable&gt;</i> is not defined in this case and should not be relied upon. If the <i>&lt;sequence&gt;</i> is non-empty, stores the right-most item from the <i>&lt;sequence&gt;</i> in the <i>&lt;token list variable&gt;</i> without removing it from the <i>&lt;sequence&gt;</i> , then leaves the <i>&lt;true code&gt;</i> in the input stream. The <i>&lt;token list variable&gt;</i> is assigned locally.
<small>New: 2012-05-19</small>	
<code>\seq_pop_left:NNTF</code>	<code>\seq_pop_left:NNTF &lt;sequence&gt; &lt;token list variable&gt; {{true code}} {{false code}}</code>
<code>\seq_pop_left:CNTF</code>	If the <i>&lt;sequence&gt;</i> is empty, leaves the <i>&lt;false code&gt;</i> in the input stream. The value of the <i>&lt;token list variable&gt;</i> is not defined in this case and should not be relied upon. If the <i>&lt;sequence&gt;</i> is non-empty, pops the left-most item from the <i>&lt;sequence&gt;</i> in the <i>&lt;token list variable&gt;</i> , i.e. removes the item from the <i>&lt;sequence&gt;</i> , then leaves the <i>&lt;true code&gt;</i> in the input stream. Both the <i>&lt;sequence&gt;</i> and the <i>&lt;token list variable&gt;</i> are assigned locally.
<small>New: 2012-05-14</small>	
<small>Updated: 2012-05-19</small>	
<code>\seq_gpop_left:NNTF</code>	<code>\seq_gpop_left:NNTF &lt;sequence&gt; &lt;token list variable&gt; {{true code}} {{false code}}</code>
<code>\seq_gpop_left:CNTF</code>	If the <i>&lt;sequence&gt;</i> is empty, leaves the <i>&lt;false code&gt;</i> in the input stream. The value of the <i>&lt;token list variable&gt;</i> is not defined in this case and should not be relied upon. If the <i>&lt;sequence&gt;</i> is non-empty, pops the left-most item from the <i>&lt;sequence&gt;</i> in the <i>&lt;token list variable&gt;</i> , i.e. removes the item from the <i>&lt;sequence&gt;</i> , then leaves the <i>&lt;true code&gt;</i> in the input stream. The <i>&lt;sequence&gt;</i> is modified globally, while the <i>&lt;token list variable&gt;</i> is assigned locally.
<small>New: 2012-05-14</small>	
<small>Updated: 2012-05-19</small>	
<code>\seq_pop_right:NNTF</code>	<code>\seq_pop_right:NNTF &lt;sequence&gt; &lt;token list variable&gt; {{true code}} {{false code}}</code>
<code>\seq_pop_right:CNTF</code>	If the <i>&lt;sequence&gt;</i> is empty, leaves the <i>&lt;false code&gt;</i> in the input stream. The value of the <i>&lt;token list variable&gt;</i> is not defined in this case and should not be relied upon. If the <i>&lt;sequence&gt;</i> is non-empty, pops the right-most item from the <i>&lt;sequence&gt;</i> in the <i>&lt;token list variable&gt;</i> , i.e. removes the item from the <i>&lt;sequence&gt;</i> , then leaves the <i>&lt;true code&gt;</i> in the input stream. Both the <i>&lt;sequence&gt;</i> and the <i>&lt;token list variable&gt;</i> are assigned locally.
<small>New: 2012-05-19</small>	
<code>\seq_gpop_right:NNTF</code>	<code>\seq_gpop_right:NNTF &lt;sequence&gt; &lt;token list variable&gt; {{true code}} {{false code}}</code>
<code>\seq_gpop_right:CNTF</code>	If the <i>&lt;sequence&gt;</i> is empty, leaves the <i>&lt;false code&gt;</i> in the input stream. The value of the <i>&lt;token list variable&gt;</i> is not defined in this case and should not be relied upon. If the <i>&lt;sequence&gt;</i> is non-empty, pops the right-most item from the <i>&lt;sequence&gt;</i> in the <i>&lt;token list variable&gt;</i> , i.e. removes the item from the <i>&lt;sequence&gt;</i> , then leaves the <i>&lt;true code&gt;</i> in the input stream. The <i>&lt;sequence&gt;</i> is modified globally, while the <i>&lt;token list variable&gt;</i> is assigned locally.
<small>New: 2012-05-19</small>	

## 19.5 Modifying sequences

While sequences are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update sequences, while retaining the order of the unaffected entries.

---

```
\seq_remove_duplicates:N \seq_remove_duplicates:N <sequence>
```

`\seq_remove_duplicates:c` `\seq_gremove_duplicates:N`  
`\seq_gremove_duplicates:c`

Removes duplicate items from the *<sequence>*, leaving the left most copy of each item in the *<sequence>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

**TeXhackers note:** This function iterates through every item in the *<sequence>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large sequences.

---

```
\seq_remove_all:Nn \seq_remove_all:Nn <sequence> {<item>}
```

`\seq_remove_all:cn` `\seq_gremove_all:Nn`  
`\seq_gremove_all:cn`

Removes every occurrence of *<item>* from the *<sequence>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

---

```
\seq_reverse:N \seq_reverse:N <sequence>
```

`\seq_reverse:c` `\seq_greverse:N`  
`\seq_greverse:c`

New: 2014-07-18

---

```
\seq_sort:Nn \seq_sort:Nn <sequence> {<comparison code>}
```

`\seq_sort:cn` `\seq_gsort:Nn`  
`\seq_gsort:cn`

Sorts the items in the *<sequence>* according to the *<comparison code>*, and assigns the result to *<sequence>*. The details of sorting comparison are described in Section 6.1.

New: 2017-02-06

---

```
\seq_shuffle:N \seq_shuffle:N <seq var>
```

`\seq_shuffle:c` `\seq_gshuffle:N`  
`\seq_gshuffle:c`

Sets the *<seq var>* to the result of placing the items of the *<seq var>* in a random order. Each item is (roughly) as likely to end up in any given position.

New: 2018-04-29

**TeXhackers note:** For sequences with more than 13 items or so, only a small proportion of all possible permutations can be reached, because the random seed `\sys_rand_seed`: only has 28-bits. The use of `\toks` internally means that sequences with more than 32767 or 65535 items (depending on the engine) cannot be shuffled.

## 19.6 Sequence conditionals

---

```
\seq_if_empty_p:N * \seq_if_empty_p:N <sequence>
```

```
\seq_if_empty_p:c * \seq_if_empty:NTF <sequence> {<true code>} {<false code>}
```

```
\seq_if_empty:NTF * Tests if the <sequence> is empty (containing no items).
```

```
\seq_if_empty:cTF *
```

---

`\seq_if_in:NnTF` `\seq_if_in:NnTF <sequence> {<item>} {<true code>} {<false code>}`  
`\seq_if_in:(NV|Nv|No|Nx|cn|cV|cv|co|cx)TF`

Tests if the *<item>* is present in the *<sequence>*.

## 19.7 Mapping over sequences

All mappings are done at the current group level, *i.e.* any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

---

`\seq_map_function>NN`  $\star$  `\seq_map_function>NN <sequence> <function>`

`\seq_map_function:cN`  $\star$

Updated: 2012-06-29

Applies *<function>* to every *<item>* stored in the *<sequence>*. The *<function>* will receive one argument for each iteration. The *<items>* are returned from left to right. To pass further arguments to the *<function>*, see `\seq_map_tokens:Nn`. The function `\seq_map_inline:Nn` is faster than `\seq_map_function>NN` for sequences with more than about 10 items.

---

`\seq_map_inline:Nn` `\seq_map_inline:Nn <sequence> {<inline function>}`

`\seq_map_inline:cN`

Updated: 2012-06-29

Applies *<inline function>* to every *<item>* stored within the *<sequence>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. The *<items>* are returned from left to right.

---

`\seq_map_tokens:Nn`  $\star$  `\seq_map_tokens:Nn <sequence> {<code>}`

`\seq_map_tokens:cN`  $\star$

New: 2019-08-30

Analogue of `\seq_map_function>NN` which maps several tokens instead of a single function. The *<code>* receives each item in the *<sequence>* as a trailing brace group. For instance,

```
\seq_map_tokens:Nn \l_my_seq { \prg_replicate:nn { 2 } }
```

expands to twice each item in the *<sequence>*: for each item in `\l_my_seq` the function `\prg_replicate:nn` receives 2 and *<item>* as its two arguments. The function `\seq_map_inline:Nn` is typically faster but it is not expandable.

---

`\seq_map_variable:NNn` `\seq_map_variable:NNn <sequence> <variable> {<code>}`

`\seq_map_variable:(Ncn|cNn|ccn)`

Updated: 2012-06-29

Stores each *<item>* of the *<sequence>* in turn in the (token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<item>* in the *<sequence>*, or its original value if the *<sequence>* is empty. The *<items>* are returned from left to right.

---

`\seq_map_indexed_function:NN`  $\star$  `\seq_map_indexed_function:NN <seq var> <function>`

New: 2018-05-03

Applies *<function>* to every entry in the *<sequence variable>*. The *<function>* should have signature `:nn`. It receives two arguments for each iteration: the *<index>* (namely 1 for the first entry, then 2 and so on) and the *<item>*.

---

```
\seq_map_indexed_inline:Nn \seq_map_indexed_inline:Nn <seq var> {{inline function}}
```

New: 2018-05-03

Applies *<inline function>* to every entry in the *<sequence variable>*. The *<inline function>* should consist of code which receives the *<index>* (namely 1 for the first entry, then 2 and so on) as #1 and the *<item>* as #2.

---

```
\seq_map_break: ☆ \seq_map_break:
```

Updated: 2012-06-29

Used to terminate a `\seq_map_...` function before all entries in the *<sequence>* have been processed. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \seq_map_break: }
    {
        % Do something useful
    }
}
```

Use outside of a `\seq_map_...` scenario leads to low level T<sub>E</sub>X errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

---

```
\seq_map_break:n ☆ \seq_map_break:n {<code>}
```

Updated: 2012-06-29

Used to terminate a `\seq_map_...` function before all entries in the *<sequence>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\seq_map_inline:Nn \l_my_seq
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \seq_map_break:n { <code> } }
    {
        % Do something useful
    }
}
```

Use outside of a `\seq_map_...` scenario leads to low level T<sub>E</sub>X errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

---

`\seq_set_map:NNn`    `\seq_set_map:NNn <sequence1> <sequence2> {<inline function>}`  
`\seq_gset_map:NNn`    Applies *<inline function>* to every *<item>* stored within the *<sequence<sub>2</sub>>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. The sequence resulting from applying *<inline function>* to each *<item>* is assigned to *<sequence<sub>1</sub>>*.

---

New: 2011-12-22    Updated: 2020-07-16

**TeXhackers note:** Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

---

`\seq_set_map_x:NNn`    `\seq_set_map_x:NNn <sequence1> <sequence2> {<inline function>}`  
`\seq_gset_map_x:NNn`    Applies *<inline function>* to every *<item>* stored within the *<sequence<sub>2</sub>>*. The *<inline function>* should consist of code which will receive the *<item>* as #1. The sequence resulting from x-expanding *<inline function>* applied to each *<item>* is assigned to *<sequence<sub>1</sub>>*. As such, the code in *<inline function>* should be expandable.

---

New: 2020-07-16

**TeXhackers note:** Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

---

`\seq_count:N * \seq_count:N <sequence>`  
`\seq_count:c *` Leaves the number of items in the *<sequence>* in the input stream as an *<integer denotation>*. The total number of items in a *<sequence>* includes those which are empty and duplicates, i.e. every item in a *<sequence>* is unique.

---

New: 2012-07-13

## 19.8 Using the content of sequences directly

---

`\seq_use:Nnnn * \seq_use:Nnnn <seq var> {<separator between two>}`  
`\seq_use:cnnn * {<separator between more than two>} {<separator between final two>}`

---

New: 2013-05-26

Places the contents of the *<seq var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the sequence has more than two items, the *<separator between more than two>* is placed between each pair of items except the last, for which the *<separator between final two>* is used. If the sequence has exactly two items, then they are placed in the input stream separated by the *<separator between two>*. If the sequence has a single item, it is placed in the input stream, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
```

inserts “a, b, c, de, and f” in the input stream. The first separator argument is not used in this case because the sequence has more than 2 items.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an x-type or e-type argument expansion.

---

`\seq_use:Nn` \* `\seq_use:Nn <seq var> {<separator>}`

`\seq_use:cN` \* Places the contents of the `<seq var>` in the input stream, with the `<separator>` between the items. If the sequence has a single item, it is placed in the input stream with no `<separator>`, and an empty sequence produces no output. An error is raised if the variable does not exist or if it is invalid.

New: 2013-05-26

For example,

```
\seq_set_split:Nnn \l_tmpa_seq { | } { a | b | c | {de} | f }
\seq_use:Nn \l_tmpa_seq { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the `<items>` do not expand further when appearing in an `x-type` or `e-type` argument expansion.

## 19.9 Sequences as stacks

Sequences can be used as stacks, where data is pushed to and popped from the top of the sequence. (The left of a sequence is the top, for performance reasons.) The stack functions for sequences are not intended to be mixed with the general ordered data functions detailed in the previous section: a sequence should either be used as an ordered data type or as a stack, but not in both ways.

---

`\seq_get:NN` `\seq_get:NN <sequence> <token list variable>`

`\seq_get:cN` Updated: 2012-05-14 Reads the top item from a `<sequence>` into the `<token list variable>` without removing it from the `<sequence>`. The `<token list variable>` is assigned locally. If `<sequence>` is empty the `<token list variable>` is set to the special marker `\q_no_value`.

---

`\seq_pop:NN` `\seq_pop:NN <sequence> <token list variable>`

`\seq_pop:cN` Updated: 2012-05-14 Pops the top item from a `<sequence>` into the `<token list variable>`. Both of the variables are assigned locally. If `<sequence>` is empty the `<token list variable>` is set to the special marker `\q_no_value`.

---

`\seq_gpop:NN` `\seq_gpop:NN <sequence> <token list variable>`

`\seq_gpop:cN` Updated: 2012-05-14 Pops the top item from a `<sequence>` into the `<token list variable>`. The `<sequence>` is modified globally, while the `<token list variable>` is assigned locally. If `<sequence>` is empty the `<token list variable>` is set to the special marker `\q_no_value`.

---

`\seq_get:NNTF` `\seq_get:NNTF <sequence> <token list variable> {<true code>} {<false code>}`

`\seq_get:cNTF` New: 2012-05-14 Updated: 2012-05-19 If the `<sequence>` is empty, leaves the `<false code>` in the input stream. The value of the `<token list variable>` is not defined in this case and should not be relied upon. If the `<sequence>` is non-empty, stores the top item from a `<sequence>` in the `<token list variable>` without removing it from the `<sequence>`. The `<token list variable>` is assigned locally.

\seq\_pop:NNTF    \seq\_pop:NNTF *<sequence>* *<token list variable>* {{*true code*}} {{*false code*}}  
\seq\_pop:cNTF    If the *<sequence>* is empty, leaves the *<false code>* in the input stream. The value of  
the *<token list variable>* is not defined in this case and should not be relied upon. If  
the *<sequence>* is non-empty, pops the top item from the *<sequence>* in the *<token list  
variable>*, i.e. removes the item from the *<sequence>*. Both the *<sequence>* and the *<token  
list variable>* are assigned locally.  
New: 2012-05-14  
Updated: 2012-05-19

\seq\_gpop:NNTF \seq\_gpop:NNTF *sequence* *token list variable* {*true code*} {*false code*}  
\seq\_gpop:cNTF If the *sequence* is empty, leaves the *false code* in the input stream. The value of the *token list variable* is not defined in this case and should not be relied upon. If the *sequence* is non-empty, pops the top item from the *sequence* in the *token list variable*, i.e. removes the item from the *sequence*. The *sequence* is modified globally, while the *token list variable* is assigned locally.

```
\seq_push:Nn                                \seq_push:Nn  <sequence> {(item)}  
\seq_push:(NV|Nv|No|Nx|cn|cV|cv|co|cx)  
\seq_gpush:Nn  
\seq_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)
```

Adds the {*<item>*} to the top of the *<sequence>*.

## 19.10 Sequences as sets

Sequences can also be used as sets, such that all of their items are distinct. Usage of sequences as sets is not currently widespread, hence no specific set function is provided. Instead, it is explained here how common set operations can be performed by combining several functions described in earlier sections. When using sequences to implement sets, one should be careful not to rely on the order of items in the sequence representing the set.

Sets should not contain several occurrences of a given item. To make sure that a `\seq_remove_duplicates:N` *sequence variable*) only has distinct items, use `\seq_remove_duplicates:N` *sequence variable*). This function is relatively slow, and to avoid performance issues one should only use it when necessary.

Some operations on a set  $\langle seq\ var \rangle$  are straightforward. For instance, `\seq_count:N`  $\langle seq\ var \rangle$  expands to the number of items, while `\seq_if_in:NnTF`  $\langle seq\ var \rangle$  { $\langle item \rangle$ } tests if the  $\langle item \rangle$  is in the set.

Adding an  $\langle item \rangle$  to a set  $\langle seq\ var \rangle$  can be done by appending it to the  $\langle seq\ var \rangle$  if it is not already in the  $\langle seq\ var \rangle$ :

```
\seq_if_in:NnF seq var {item}  
{ \seq_put_right:Nn seq var {item} }
```

Removing an  $\langle item \rangle$  from a set  $\langle seq\ var \rangle$  can be done using `\seq_remove_all:Nn`,

\seq\_remove\_all:Nn \seq var {*item*}

The intersection of two sets  $\langle \text{seq } var_1 \rangle$  and  $\langle \text{seq } var_2 \rangle$  can be stored into  $\langle \text{seq } var_3 \rangle$  by collecting items of  $\langle \text{seq } var_1 \rangle$  which are in  $\langle \text{seq } var_2 \rangle$ .

```
\seq_clear:N <seq var3>
\seq_map_inline:Nn <seq var1>
{
\seq_if_in:NnT <seq var2> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}
```

The code as written here only works if  $\langle \text{seq } \text{var}_3 \rangle$  is different from the other two sequence variables. To cover all cases, items should first be collected in a sequence  $\backslash \text{l\_}\langle \text{pkg} \rangle\text{\_internal\_seq}$ , then  $\langle \text{seq } \text{var}_3 \rangle$  should be set equal to this internal sequence. The same remark applies to other set functions.

The union of two sets  $\langle \text{seq } \text{var}_1 \rangle$  and  $\langle \text{seq } \text{var}_2 \rangle$  can be stored into  $\langle \text{seq } \text{var}_3 \rangle$  through

```
\seq_concat:NNN <seq var3> <seq var1> <seq var2>
\seq_remove_duplicates:N <seq var3>
```

or by adding items to (a copy of)  $\langle \text{seq } \text{var}_1 \rangle$  one by one

```
\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{
\seq_if_in:NnF <seq var3> {#1}
{ \seq_put_right:Nn <seq var3> {#1} }
}
```

The second approach is faster than the first when the  $\langle \text{seq } \text{var}_2 \rangle$  is short compared to  $\langle \text{seq } \text{var}_1 \rangle$ .

The difference of two sets  $\langle \text{seq } \text{var}_1 \rangle$  and  $\langle \text{seq } \text{var}_2 \rangle$  can be stored into  $\langle \text{seq } \text{var}_3 \rangle$  by removing items of the  $\langle \text{seq } \text{var}_2 \rangle$  from (a copy of) the  $\langle \text{seq } \text{var}_1 \rangle$  one by one.

```
\seq_set_eq:NN <seq var3> <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn <seq var3> {#1} }
```

The symmetric difference of two sets  $\langle \text{seq } \text{var}_1 \rangle$  and  $\langle \text{seq } \text{var}_2 \rangle$  can be stored into  $\langle \text{seq } \text{var}_3 \rangle$  by computing the difference between  $\langle \text{seq } \text{var}_1 \rangle$  and  $\langle \text{seq } \text{var}_2 \rangle$  and storing the result as  $\backslash \text{l\_}\langle \text{pkg} \rangle\text{\_internal\_seq}$ , then the difference between  $\langle \text{seq } \text{var}_2 \rangle$  and  $\langle \text{seq } \text{var}_1 \rangle$ , and finally concatenating the two differences to get the symmetric differences.

```
\seq_set_eq:NN \l_\langle \text{pkg} \rangle\text{\_internal\_seq} <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \l_\langle \text{pkg} \rangle\text{\_internal\_seq} {#1} }
\seq_set_eq:NN <seq var3> <seq var2>
\seq_map_inline:Nn <seq var1>
{ \seq_remove_all:Nn <seq var3> {#1} }
\seq_concat:NNN <seq var3> <seq var3> \l_\langle \text{pkg} \rangle\text{\_internal\_seq}
```

## 19.11 Constant and scratch sequences

---

`\c_empty_seq` Constant that is always empty.

New: 2012-07-02

---

**\l\_tmpa\_seq** Scratch sequences for local assignment. These are never used by the kernel code, and so  
**\l\_tmpb\_seq** are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
New: 2012-04-26 other non-kernel code and so should only be used for short-term storage.

---

**\g\_tmpa\_seq** Scratch sequences for global assignment. These are never used by the kernel code, and so  
**\g\_tmpb\_seq** are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
New: 2012-04-26 other non-kernel code and so should only be used for short-term storage.

## 19.12 Viewing sequences

---

**\seq\_show:N**      \seq\_show:N *sequence*  
**\seq\_show:c**      Displays the entries in the *sequence* in the terminal.  
Updated: 2021-04-29

---

**\seq\_log:N**      \seq\_log:N *sequence*  
**\seq\_log:c**      Writes the entries in the *sequence* in the log file.  
New: 2014-08-12  
Updated: 2021-04-29

# Chapter 20

## The `I3int` package Integers

Calculation and comparison of integer values can be carried out using literal numbers, `int` registers, constants and integers stored in token list variables. The standard operators `+`, `-`, `/` and `*` and parentheses can be used within such expressions to carry arithmetic operations. This module carries out these functions on *integer expressions* (“`intexpr`”).

### 20.1 Integer expressions

Throughout this module, (almost) all `n`-type argument allow for an `<intexpr>` argument with the following syntax. The `<integer expression>` should consist, after expansion, of `+`, `-`, `*`, `/`, `(`, `)` and of course integer operands. The result is calculated by applying standard mathematical rules with the following peculiarities:

- `/` denotes division rounded to the closest integer with ties rounded away from zero;
- there is an error and the overall expression evaluates to zero whenever the absolute value of any intermediate result exceeds  $2^{31} - 1$ , except in the case of scaling operations  $a*b/c$ , for which  $a*b$  may be arbitrarily large (but the operands  $a$ ,  $b$ ,  $c$  are still constrained to an absolute value at most  $2^{31} - 1$ );
- parentheses may not appear after unary `+` or `-`, namely placing `+()` or `-()` at the start of an expression or after `+`, `-`, `*`, `/` or `(` leads to an error.

Each integer operand can be either an integer variable (with no need for `\int_use:N`) or an integer denotation. For example both

```
\int_show:n { 5 + 4 * 3 - ( 3 + 4 * 5 ) }
```

and

```
\tl_new:N \l_my_tl
\tl_set:Nn \l_my_tl { 5 }
\int_new:N \l_my_int
\int_set:Nn \l_my_int { 4 }
\int_show:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

show the same result  $-6$  because  $\backslash l\_my\_t1$  expands to the integer denotation  $5$  while the integer variable  $\backslash l\_my\_int$  takes the value  $4$ . As the  $\langle integer\ expression \rangle$  is fully expanded from left to right during evaluation, fully expandable and restricted-expandable functions can both be used, and  $\backslash exp\_not:n$  and its variants have no effect while  $\backslash exp\_not:N$  may incorrectly interrupt the expression.

**TeXhackers note:** Exactly two expansions are needed to evaluate  $\backslash int\_eval:n$ . The result is *not* an  $\langle internal\ integer \rangle$ , and therefore should be terminated by a space if used in  $\backslash int\_value:w$  or in a TeX-style integer assignment.

As all TeX integers, integer operands can also be:  $\backslash value\{\langle LATEX 2\varepsilon\ counter \rangle\}$ ; dimension or skip variables, converted to integers in  $sp$ ; the character code of some character given as ' $\langle char \rangle$ ' or ' $\backslash \langle char \rangle$ '; octal numbers given as ' followed by digits from  $0$  to  $7$ ; or hexadecimal numbers given as " followed by digits and upper case letters from  $A$  to  $F$ .

---

**`\int_eval:n`**  $\star$  `\int_eval:n {<integer expression>}`

Evaluates the  $\langle integer\ expression \rangle$  and leaves the result in the input stream as an integer denotation: for positive results an explicit sequence of decimal digits not starting with  $0$ , for negative results – followed by such a sequence, and  $0$  for zero.

---

**`\int_eval:w`**  $\star$  `\int_eval:w {<integer expression>}`

New: 2018-03-30 Evaluates the  $\langle integer\ expression \rangle$  as described for  $\backslash int\_eval:n$ . The end of the expression is the first token encountered that cannot form part of such an expression. If that token is  $\backslash scan\_stop$ : it is removed, otherwise not. Spaces do *not* terminate the expression. However, spaces terminate explicit integers, and this may terminate the expression: for instance,  $\backslash int\_eval:w 1\underline{u}+1\underline{u}9$  (with explicit space tokens inserted using  $\sim$  in a code setting) expands to  $29$  since the digit  $9$  is not part of the expression.

---

**`\int_sign:n`**  $\star$  `\int_sign:n {<intexpr>}`

New: 2018-11-03 Evaluates the  $\langle integer\ expression \rangle$  then leaves  $1$  or  $0$  or  $-1$  in the input stream according to the sign of the result.

---

**`\int_abs:n`**  $\star$  `\int_abs:n {<integer expression>}`

Updated: 2012-09-26 Evaluates the  $\langle integer\ expression \rangle$  as described for  $\backslash int\_eval:n$  and leaves the absolute value of the result in the input stream as an  $\langle integer\ denotation \rangle$  after two expansions.

---

**`\int_div_round:nn`**  $\star$  `\int_div_round:nn {<intexpr1>} {<intexpr2>}`

Updated: 2012-09-26 Evaluates the two  $\langle integer\ expressions \rangle$  as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using  $/$  directly in an  $\langle integer\ expression \rangle$ . The result is left in the input stream as an  $\langle integer\ denotation \rangle$  after two expansions.

---

**`\int_div_truncate:nn`**  $\star$  `\int_div_truncate:nn {<intexpr1>} {<intexpr2>}`

Updated: 2012-02-09 Evaluates the two  $\langle integer\ expressions \rangle$  as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using  $/$  rounds to the closest integer instead. The result is left in the input stream as an  $\langle integer\ denotation \rangle$  after two expansions.

---

```
\int_max:nn * \int_max:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}
\int_min:nn * \int_min:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}
```

---

Updated: 2012-09-26 Evaluates the *<integer expressions>* as described for `\int_eval:n` and leaves either the larger or smaller value in the input stream as an *<integer denotation>* after two expansions.

---

```
\int_mod:nn * \int_mod:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}
```

---

Updated: 2012-09-26 Evaluates the two *<integer expressions>* as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting `\int_div_truncate:nn {\langle intexpr_1\rangle} {\langle intexpr_2\rangle}` times *<intexpr\_2>* from *<intexpr\_1>*. Thus, the result has the same sign as *<intexpr\_1>* and its absolute value is strictly less than that of *<intexpr\_2>*. The result is left in the input stream as an *<integer denotation>* after two expansions.

## 20.2 Creating and initialising integers

---

```
\int_new:N \int_new:N <integer>
```

---

`\int_new:c` Creates a new *<integer>* or raises an error if the name is already taken. The declaration is global. The *<integer>* is initially equal to 0.

---

```
\int_const:Nn \int_const:Nn <integer> {\langle integer expression\rangle}
```

---

Updated: 2011-10-22 `\int_const:cn` Creates a new constant *<integer>* or raises an error if the name is already taken. The value of the *<integer>* is set globally to the *<integer expression>*.

---

```
\int_zero:N \int_zero:N <integer>
```

---

`\int_zero:c` Sets *<integer>* to 0.  
`\int_gzero:N`  
`\int_gzero:c`

---

```
\int_zero_new:N \int_zero_new:N <integer>
```

---

`\int_zero_new:c` Ensures that the *<integer>* exists globally by applying `\int_new:N` if necessary, then applies `\int_(g)zero:N` to leave the *<integer>* set to zero.

New: 2011-12-13

---

```
\int_set_eq:NN \int_set_eq:NN <integer_1> <integer_2>
```

---

`\int_set_eq:(cN|Nc|cc)` Sets the content of *<integer\_1>* equal to that of *<integer\_2>*.

`\int_gset_eq:NN`

`\int_gset_eq:(cN|Nc|cc)`

---

```
\int_if_exist_p:N * \int_if_exist_p:N <int>
\int_if_exist_p:c * \int_if_exist:NTF <int> {\langle true code\rangle} {\langle false code\rangle}
```

---

`\int_if_exist:NTF` \* Tests whether the *<int>* is currently defined. This does not check that the *<int>* really is an integer variable.

New: 2012-03-03

## 20.3 Setting and incrementing integers

---

\int\_add:Nn \int\_add:Nn *integer* {*integer expression*}  
\int\_add:cn Adds the result of the *integer expression* to the current content of the *integer*.  
\int\_gadd:Nn  
\int\_gadd:cn

---

Updated: 2011-10-22

---

\int\_decr:N \int\_decr:N *integer*  
\int\_decr:c Decreases the value stored in *integer* by 1.  
\int\_gdecr:N  
\int\_gdecr:c

---

---

\int\_incr:N \int\_incr:N *integer*  
\int\_incr:c Increases the value stored in *integer* by 1.  
\int\_gincr:N  
\int\_gincr:c

---

---

\int\_set:Nn \int\_set:Nn *integer* {*integer expression*}  
\int\_set:cn Sets *integer* to the value of *integer expression*, which must evaluate to an integer (as described for \int\_eval:n).  
\int\_gset:Nn  
\int\_gset:cn

---

Updated: 2011-10-22

---

\int\_sub:Nn \int\_sub:Nn *integer* {*integer expression*}  
\int\_sub:cn Subtracts the result of the *integer expression* from the current content of the *integer*.  
\int\_gsub:Nn  
\int\_gsub:cn

---

Updated: 2011-10-22

## 20.4 Using integers

---

\int\_use:N \* \int\_use:N *integer*  
\int\_use:c \* Recovers the content of an *integer* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an *integer* is required (such as in the first and third arguments of \int\_compare:nNnTF).  
Updated: 2011-10-22

**TeXhackers note:** \int\_use:N is the TeX primitive \the: this is one of several L<sup>A</sup>T<sub>E</sub>X3 names for this primitive.

## 20.5 Integer expression conditionals

---

```
\int_compare_p:nNn * \int_compare_p:nNn {\langle intexpr_1\rangle} {relation} {\langle intexpr_2\rangle}
\int_compare:nNnTF * \int_compare:nNnTF
    {\langle intexpr_1\rangle} {relation} {\langle intexpr_2\rangle}
    {\langle true code\rangle} {\langle false code\rangle}
```

---

This function first evaluates each of the *<integer expressions>* as described for `\int_eval:n`. The two results are then compared using the *<relation>*:

Equal	=
Greater than	>
Less than	<

This function is less flexible than `\int_compare:nTF` but around 5 times faster.

---

```
\int_compare_p:n * \int_compare_p:n
\int_compare:nTF * {
    ...
    {\langle intexpr_1\rangle} {\langle relation_1\rangle}
    ...
    {\langle intexpr_N\rangle} {\langle relation_N\rangle}
    {\langle intexpr_{N+1}\rangle}
}
\int_compare:nTF
{
    ...
    {\langle intexpr_1\rangle} {\langle relation_1\rangle}
    ...
    {\langle intexpr_N\rangle} {\langle relation_N\rangle}
    {\langle intexpr_{N+1}\rangle}
}
{\langle true code\rangle} {\langle false code\rangle}
```

---

Updated: 2013-01-13

This function evaluates the *<integer expressions>* as described for `\int_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<intexpr<sub>1</sub>>* and *<intexpr<sub>2</sub>>* using the *<relation<sub>1</sub>>*, then *<intexpr<sub>2</sub>>* and *<intexpr<sub>3</sub>>* using the *<relation<sub>2</sub>>*, until finally comparing *<intexpr<sub>N</sub>>* and *<intexpr<sub>N+1</sub>>* using the *<relation<sub>N</sub>>*. The test yields `true` if all comparisons are `true`. Each *<integer expression>* is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other *<integer expression>* is evaluated and no other comparison is performed. The *<relations>* can be any of the following:

Equal	= or ==
Greater than or equal to	<code>&gt;=</code>
Greater than	<code>&gt;</code>
Less than or equal to	<code>&lt;=</code>
Less than	<code>&lt;</code>
Not equal	<code>!=</code>

This function is more flexible than `\int_compare:nNnTF` but around 5 times slower.

---

```
\int_case:nn   * \int_case:nnTF {\test integer expression}
\int_case:nnTF *   {
  New: 2013-07-24
    {<intexpr case1>} {<code case1>}
    {<intexpr case2>} {<code case2>}
    ...
    {<intexpr casen>} {<code casen>}
}
{<true code>}
{<false code>}
```

---

This function evaluates the  $\langle test \ integer \ expression \rangle$  and compares this in turn to each of the  $\langle integer \ expression \ cases \rangle$ . If the two are equal then the associated  $\langle code \rangle$  is left in the input stream and other cases are discarded. If any of the cases are matched, the  $\langle true \ code \rangle$  is also inserted into the input stream (after the code for the appropriate case), while if none match then the  $\langle false \ code \rangle$  is inserted. The function  $\backslash int\_case:nn$ , which does nothing if there is no match, is also available. For example

```
\int_case:nnF
  { 2 * 5 }
{
  { 5 }      { Small }
  { 4 + 6 }  { Medium }
  { -2 * 10 } { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

---

```
\int_if_even_p:n * \int_if_odd_p:n {\integer expression}
\int_if_even:nTF * \int_if_odd:nTF {\integer expression}
\int_if_odd_p:n *  {<true code>} {<false code>}
```

---

$\backslash int\_if\_odd:nTF$  This function first evaluates the  $\langle integer \ expression \rangle$  as described for  $\backslash int\_eval:n$ . It then evaluates if this is odd or even, as appropriate.

## 20.6 Integer expression loops

---

```
\int_do_until:nNnn ☆ \int_do_until:nNnn {\<intexpr1>} {<relation>} {\<intexpr2>} {<code>}
```

---

Places the  $\langle code \rangle$  in the input stream for TeX to process, and then evaluates the relationship between the two  $\langle integer \ expressions \rangle$  as described for  $\backslash int\_compare:nNnTF$ . If the test is **false** then the  $\langle code \rangle$  is inserted into the input stream again and a loop occurs until the  $\langle relation \rangle$  is **true**.

---

```
\int_do_while:nNnn ☆ \int_do_while:nNnn {\<intexpr1>} {<relation>} {\<intexpr2>} {<code>}
```

---

Places the  $\langle code \rangle$  in the input stream for TeX to process, and then evaluates the relationship between the two  $\langle integer \ expressions \rangle$  as described for  $\backslash int\_compare:nNnTF$ . If the test is **true** then the  $\langle code \rangle$  is inserted into the input stream again and a loop occurs until the  $\langle relation \rangle$  is **false**.

---

`\int_until_do:nNnn`  $\star$  `\int_until_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}`

Evaluates the relationship between the two  $\langle integer\ expressions \rangle$  as described for `\int_compare:nNnTF`, and then places the  $\langle code \rangle$  in the input stream if the  $\langle relation \rangle$  is `false`. After the  $\langle code \rangle$  has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

---

`\int_while_do:nNnn`  $\star$  `\int_while_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}`

Evaluates the relationship between the two  $\langle integer\ expressions \rangle$  as described for `\int_compare:nNnTF`, and then places the  $\langle code \rangle$  in the input stream if the  $\langle relation \rangle$  is `true`. After the  $\langle code \rangle$  has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

---

`\int_do_until:nn`  $\star$  `\int_do_until:nn {\langle integer\ relation \rangle} {\langle code \rangle}`

Updated: 2013-01-13

Places the  $\langle code \rangle$  in the input stream for TeX to process, and then evaluates the  $\langle integer\ relation \rangle$  as described for `\int_compare:nTF`. If the test is `false` then the  $\langle code \rangle$  is inserted into the input stream again and a loop occurs until the  $\langle relation \rangle$  is `true`.

---

`\int_do_while:nn`  $\star$  `\int_do_while:nn {\langle integer\ relation \rangle} {\langle code \rangle}`

Updated: 2013-01-13

Places the  $\langle code \rangle$  in the input stream for TeX to process, and then evaluates the  $\langle integer\ relation \rangle$  as described for `\int_compare:nTF`. If the test is `true` then the  $\langle code \rangle$  is inserted into the input stream again and a loop occurs until the  $\langle relation \rangle$  is `false`.

---

`\int_until_do:nn`  $\star$  `\int_until_do:nn {\langle integer\ relation \rangle} {\langle code \rangle}`

Updated: 2013-01-13

Evaluates the  $\langle integer\ relation \rangle$  as described for `\int_compare:nTF`, and then places the  $\langle code \rangle$  in the input stream if the  $\langle relation \rangle$  is `false`. After the  $\langle code \rangle$  has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

---

`\int_while_do:nn`  $\star$  `\int_while_do:nn {\langle integer\ relation \rangle} {\langle code \rangle}`

Updated: 2013-01-13

Evaluates the  $\langle integer\ relation \rangle$  as described for `\int_compare:nTF`, and then places the  $\langle code \rangle$  in the input stream if the  $\langle relation \rangle$  is `true`. After the  $\langle code \rangle$  has been processed by TeX the test is repeated, and a loop occurs until the test is `false`.

## 20.7 Integer step functions

---

```
\int_step_function:nN  ☆ \int_step_function:nN {<final value>} <function>
\int_step_function:nnN ☆ \int_step_function:nnN {<initial value>} {<final value>} <function>
\int_step_function:nnnN ☆ \int_step_function:nnnN {<initial value>} {<step>} {<final value>} <function>
```

---

New: 2012-06-04

Updated: 2018-04-22

This function first evaluates the  $\langle initial\ value\rangle$ ,  $\langle step\rangle$  and  $\langle final\ value\rangle$ , all of which should be integer expressions. The  $\langle function\rangle$  is then placed in front of each  $\langle value\rangle$  from the  $\langle initial\ value\rangle$  to the  $\langle final\ value\rangle$  in turn (using  $\langle step\rangle$  between each  $\langle value\rangle$ ). The  $\langle step\rangle$  must be non-zero. If the  $\langle step\rangle$  is positive, the loop stops when the  $\langle value\rangle$  becomes larger than the  $\langle final\ value\rangle$ . If the  $\langle step\rangle$  is negative, the loop stops when the  $\langle value\rangle$  becomes smaller than the  $\langle final\ value\rangle$ . The  $\langle function\rangle$  should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }
\int_step_function:nnnN { 1 } { 1 } { 5 } \my_func:n
```

would print

```
[I saw 1] [I saw 2] [I saw 3] [I saw 4] [I saw 5]
```

The functions  $\text{\int\_step\_function:nN}$  and  $\text{\int\_step\_function:nnN}$  both use a fixed  $\langle step\rangle$  of 1, and in the case of  $\text{\int\_step\_function:nN}$  the  $\langle initial\ value\rangle$  is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

---

```
\int_step_inline:nn \int_step_inline:nn {<final value>} <code>
\int_step_inline:nnn \int_step_inline:nnn {<initial value>} {<final value>} <code>
\int_step_inline:nnnn \int_step_inline:nnnn {<initial value>} {<step>} {<final value>} <code>
```

---

New: 2012-06-04

Updated: 2018-04-22

This function first evaluates the  $\langle initial\ value\rangle$ ,  $\langle step\rangle$  and  $\langle final\ value\rangle$ , all of which should be integer expressions. Then for each  $\langle value\rangle$  from the  $\langle initial\ value\rangle$  to the  $\langle final\ value\rangle$  in turn (using  $\langle step\rangle$  between each  $\langle value\rangle$ ), the  $\langle code\rangle$  is inserted into the input stream with  $\#1$  replaced by the current  $\langle value\rangle$ . Thus the  $\langle code\rangle$  should define a function of one argument ( $\#1$ ).

The functions  $\text{\int\_step\_inline:nn}$  and  $\text{\int\_step\_inline:nnn}$  both use a fixed  $\langle step\rangle$  of 1, and in the case of  $\text{\int\_step\_inline:nn}$  the  $\langle initial\ value\rangle$  is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

---

```
\int_step_variable:nNn \int_step_variable:nNn {<final value>} <tl var> <code>
\int_step_variable:nnNn \int_step_variable:nnNn {<initial value>} {<final value>} <tl var> <code>
\int_step_variable:nnnNn \int_step_variable:nnnNn {<initial value>} {<step>} {<final value>} <tl var>
<code>
```

---

New: 2012-06-04

Updated: 2018-04-22

This function first evaluates the  $\langle initial\ value\rangle$ ,  $\langle step\rangle$  and  $\langle final\ value\rangle$ , all of which should be integer expressions. Then for each  $\langle value\rangle$  from the  $\langle initial\ value\rangle$  to the  $\langle final\ value\rangle$  in turn (using  $\langle step\rangle$  between each  $\langle value\rangle$ ), the  $\langle code\rangle$  is inserted into the input stream, with the  $\langle tl\ var\rangle$  defined as the current  $\langle value\rangle$ . Thus the  $\langle code\rangle$  should make use of the  $\langle tl\ var\rangle$ .

The functions  $\text{\int\_step\_variable:nNn}$  and  $\text{\int\_step\_variable:nnNn}$  both use a fixed  $\langle step\rangle$  of 1, and in the case of  $\text{\int\_step\_variable:nNn}$  the  $\langle initial\ value\rangle$  is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

## 20.8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

---

```
\int_to_arabic:n * \int_to_arabic:n {\langle integer expression\rangle}
```

Updated: 2011-10-22 Places the value of the *<integer expression>* in the input stream as digits, with category code 12 (other).

---

```
\int_to_alpha:n * \int_to_alpha:n {\langle integer expression\rangle}
```

\int\_to\_Alph:n \* Updated: 2011-09-17 Evaluates the *<integer expression>* and converts the result into a series of letters, which are then left in the input stream. The conversion rule uses the 26 letters of the English alphabet, in order, adding letters when necessary to increase the total possible range of representable numbers. Thus

```
\int_to_alpha:n { 1 }
```

places **a** in the input stream,

```
\int_to_alpha:n { 26 }
```

is represented as **z** and

```
\int_to_alpha:n { 27 }
```

is converted to **aa**. For conversions using other alphabets, use `\int_to_symbols:nnn` to define an alphabet-specific function. The basic `\int_to_alpha:n` and `\int_to_Alph:n` functions should not be modified. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

---

```
\int_to_symbols:nnn * \int_to_symbols:nnn
  {\langle integer expression\rangle} {\langle total symbols\rangle}
  {\langle value to symbol mapping\rangle}
```

Updated: 2011-09-17 This is the low-level function for conversion of an *<integer expression>* into a symbolic form (often letters). The *<total symbols>* available should be given as an integer expression. Values are actually converted to symbols according to the *<value to symbol mapping>*. This should be given as *<total symbols>* pairs of entries, a number and the appropriate symbol. Thus the `\int_to_alpha:n` function is defined as

```
\cs_new:Npn \int_to_alpha:n #1
{
  \int_to_symbols:nnn {#1} { 26 }
  {
    { 1 } { a }
    { 2 } { b }
    ...
    { 26 } { z }
  }
}
```

---

```
\int_to_bin:n ∗ \int_to_bin:n {⟨integer expression⟩}
```

New: 2014-02-11 Calculates the value of the ⟨integer expression⟩ and places the binary representation of the result in the input stream.

---

```
\int_to_hex:n ∗ \int_to_hex:n {⟨integer expression⟩}
```

\int\_to\_Hex:n ∗ Calculates the value of the ⟨integer expression⟩ and places the hexadecimal (base 16) representation of the result in the input stream. Letters are used for digits beyond 9: lower case letters for \int\_to\_hex:n and upper case ones for \int\_to\_Hex:n. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

---

```
\int_to_oct:n ∗ \int_to_oct:n {⟨integer expression⟩}
```

New: 2014-02-11 Calculates the value of the ⟨integer expression⟩ and places the octal (base 8) representation of the result in the input stream. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

---

```
\int_to_base:nn ∗ \int_to_base:nn {⟨integer expression⟩} {⟨base⟩}
```

\int\_to\_Base:nn ∗ Calculates the value of the ⟨integer expression⟩ and converts it into the appropriate representation in the ⟨base⟩; the latter may be given as an integer expression. For bases greater than 10 the higher “digits” are represented by letters from the English alphabet: lower case letters for \int\_to\_base:n and upper case ones for \int\_to\_Base:n. The maximum ⟨base⟩ value is 36. The resulting tokens are digits with category code 12 (other) and letters with category code 11 (letter).

**TeXhackers note:** This is a generic version of \int\_to\_bin:n, etc.

---

```
\int_to_roman:n ∗ \int_to_roman:n {⟨integer expression⟩}
```

\int\_to\_Roman:n ∗ Places the value of the ⟨integer expression⟩ in the input stream as Roman numerals, either lower case (\int\_to\_roman:n) or upper case (\int\_to\_Roman:n). If the value is negative or zero, the output is empty. The Roman numerals are letters with category code 11 (letter). The letters used are mdclxvi, repeated as needed: the notation with bars (such as  $\bar{v}$  for 5000) is not used. For instance \int\_to\_roman:n { 8249 } expands to mmmmmmmccxlxx.

## 20.9 Converting from other formats to integers

---

```
\int_from_alpha:n ∗ \int_from_alpha:n {⟨letters⟩}
```

Updated: 2014-08-25 Converts the ⟨letters⟩ into the integer (base 10) representation and leaves this in the input stream. The ⟨letters⟩ are first converted to a string, with no expansion. Lower and upper case letters from the English alphabet may be used, with “a” equal to 1 through to “z” equal to 26. The function also accepts a leading sign, made of + and -. This is the inverse function of \int\_to\_alpha:n and \int\_to\_Alpha:n.

---

```
\int_from_bin:n ∗ \int_from_bin:n {⟨binary number⟩}
```

New: 2014-02-11 Converts the ⟨binary number⟩ into the integer (base 10) representation and leaves this in

Updated: 2014-08-25 the input stream. The ⟨binary number⟩ is first converted to a string, with no expansion.

The function accepts a leading sign, made of + and −, followed by binary digits. This is the inverse function of \int\_to\_bin:n.

---

```
\int_from_hex:n ∗ \int_from_hex:n {⟨hexadecimal number⟩}
```

New: 2014-02-11 Converts the ⟨hexadecimal number⟩ into the integer (base 10) representation and leaves this in

Updated: 2014-08-25 the input stream. Digits greater than 9 may be represented in the ⟨hexadecimal number⟩ by upper or lower case letters. The ⟨hexadecimal number⟩ is first converted to a string, with no expansion. The function also accepts a leading sign, made of + and −. This is the inverse function of \int\_to\_hex:n and \int\_to\_Hex:n.

---

```
\int_from_oct:n ∗ \int_from_oct:n {⟨octal number⟩}
```

New: 2014-02-11 Converts the ⟨octal number⟩ into the integer (base 10) representation and leaves this in

Updated: 2014-08-25 the input stream. The ⟨octal number⟩ is first converted to a string, with no expansion.

The function accepts a leading sign, made of + and −, followed by octal digits. This is the inverse function of \int\_to\_oct:n.

---

```
\int_from_roman:n ∗ \int_from_roman:n {⟨roman numeral⟩}
```

Updated: 2014-08-25 Converts the ⟨roman numeral⟩ into the integer (base 10) representation and leaves this in

the input stream. The ⟨roman numeral⟩ is first converted to a string, with no expansion. The ⟨roman numeral⟩ may be in upper or lower case; if the numeral contains characters besides mdclxvi or MDCLXVI then the resulting value is −1. This is the inverse function of \int\_to\_roman:n and \int\_to\_Roman:n.

---

```
\int_from_base:nn ∗ \int_from_base:nn {⟨number⟩} {⟨base⟩}
```

Updated: 2014-08-25 Converts the ⟨number⟩ expressed in ⟨base⟩ into the appropriate value in base 10. The

⟨number⟩ is first converted to a string, with no expansion. The ⟨number⟩ should consist of digits and letters (either lower or upper case), plus optionally a leading sign. The maximum ⟨base⟩ value is 36. This is the inverse function of \int\_to\_base:nn and \int\_to\_Base:nn.

## 20.10 Random integers

---

```
\int_rand:nn ∗ \int_rand:nn {⟨intexpr1⟩} {⟨intexpr2⟩}
```

New: 2016-12-06 Evaluates the two ⟨integer expressions⟩ and produces a pseudo-random number between

Updated: 2018-04-27 the two (with bounds included). This is not available in older versions of X<sub>E</sub>T<sub>E</sub>X.

---

```
\int_rand:n ∗ \int_rand:n {⟨intexpr⟩}
```

New: 2018-05-05 Evaluates the ⟨integer expression⟩ then produces a pseudo-random number between 1

and the ⟨intexpr⟩ (included). This is not available in older versions of X<sub>E</sub>T<sub>E</sub>X.

## 20.11 Viewing integers

---

`\int_show:N \int_show:N <integer>`

`\int_show:c` Displays the value of the *<integer>* on the terminal.

---

`\int_show:n \int_show:n {<integer expression>}`

New: 2011-11-22 Displays the result of evaluating the *<integer expression>* on the terminal.

Updated: 2015-08-07

---

---

`\int_log:N \int_log:N <integer>`

`\int_log:c` Writes the value of the *<integer>* in the log file.

New: 2014-08-22

Updated: 2015-08-03

---

---

`\int_log:n \int_log:n {<integer expression>}`

New: 2014-08-22 Writes the result of evaluating the *<integer expression>* in the log file.

Updated: 2015-08-07

---

## 20.12 Constant integers

---

`\c_zero_int` Integer values used with primitive tests and assignments: their self-terminating nature  
`\c_one_int` makes these more convenient and faster than literal numbers.

New: 2018-05-07

---

---

`\c_max_int` The maximum value that can be stored as an integer.

---

`\c_max_register_int` Maximum number of registers.

---

`\c_max_char_int` Maximum character code completely supported by the engine.

## 20.13 Scratch integers

---

`\l_tmpa_int` Scratch integer for local assignment. These are never used by the kernel code, and so  
`\l_tmpb_int` are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_int` Scratch integer for global assignment. These are never used by the kernel code, and so  
`\g_tmpb_int` are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 20.14 Direct number expansion

---

```
\int_value:w * \int_value:w <integer>
\int_value:w <integer denotation> <optional space>
```

New: 2018-03-27

Expands the following tokens until an *<integer>* is formed, and leaves a normalized form (no leading sign except for negative numbers, no leading digit 0 except for zero) in the input stream as category code 12 (other) characters. The *<integer>* can consist of any number of signs (with intervening spaces) followed by

- an integer variable (in fact, any TeX register except `\toks`) or
- explicit digits (or by '*octal digits*' or "*hexadecimal digits*" or '*character*').

In this last case expansion stops once a non-digit is found; if that is a space it is removed as in f-expansion, and so `\exp_stop_f:` may be employed as an end marker. Note that protected functions *are* expanded by this process.

This function requires exactly one expansion to produce a value, and so is suitable for use in cases where a number is required "directly". In general, `\int_eval:n` is the preferred approach to generating numbers.

**TeXhackers note:** This is the TeX primitive `\number`.

## 20.15 Primitive conditionals

---

```
\if_int_compare:w * \if_int_compare:w <integer1> <relation> <integer2>
  <true code>
\else:
  <false code>
\fi:
```

Compare two integers using *<relation>*, which must be one of =, < or > with category code 12. The `\else:` branch is optional.

**TeXhackers note:** These are both names for the TeX primitive `\ifnum`.

---

```
\if_case:w * \if_case:w <integer> <case0>
\or:      * \or: <case1>
\or: ...
\else: <default>
\fi:
```

Selects a case to execute based on the value of the *<integer>*. The first case (*<case<sub>0</sub>>*) is executed if *<integer>* is 0, the second (*<case<sub>1</sub>>*) if the *<integer>* is 1, etc. The *<integer>* may be a literal, a constant or an integer expression (e.g. using `\int_eval:n`).

**TeXhackers note:** These are the TeX primitives `\ifcase` and `\or`.

---

```
\if_int_odd:w ∗ \if_int_odd:w <tokens>  <optional space>
    <true code>
\else:
    <true code>
\fi:
```

Expands *<tokens>* until a non-numeric token or a space is found, and tests whether the resulting *<integer>* is odd. If so, *<true code>* is executed. The **\else:** branch is optional.

**TeXhackers note:** This is the TeX primitive `\ifodd`.

# Chapter 21

## The `l3flag` package: Expandable flags

Flags are the only data-type that can be modified in expansion-only contexts. This module is meant mostly for kernel use: in almost all cases, booleans or integers should be preferred to flags because they are very significantly faster.

A flag can hold any non-negative value, which we call its *height*. In expansion-only contexts, a flag can only be “raised”: this increases the *height* by 1. The *height* can also be queried expandably. However, decreasing it, or setting it to zero requires non-expandable assignments.

Flag variables are always local. They are referenced by a *flag name* such as `str-missing`. The *flag name* is used as part of `\use:c` constructions hence is expanded at point of use. It must expand to character tokens only, with no spaces.

A typical use case of flags would be to keep track of whether an exceptional condition has occurred during expandable processing, and produce a meaningful (non-expandable) message after the end of the expandable processing. This is exemplified by `l3str-convert`, which for performance reasons performs conversions of individual characters expandably and for readability reasons produces a single error message describing incorrect inputs that were encountered.

Flags should not be used without carefully considering the fact that raising a flag takes a time and memory proportional to its height. Flags should not be used unless unavoidable.

### 21.1 Setting up flags

---

`\flag_new:n \flag_new:n {<flag name>}`

Creates a new flag with a name given by *flag name*, or raises an error if the name is already taken. The *flag name* may not contain spaces. The declaration is global, but flags are always local variables. The *flag* initially has zero height.

---

`\flag_clear:n \flag_clear:n {<flag name>}`

The *flag*’s height is set to zero. The assignment is local.

---

`\flag_clear_new:n \flag_clear_new:n {<flag name>}`

Ensures that the *<flag>* exists globally by applying `\flag_new:n` if necessary, then applies `\flag_clear:n`, setting the height to zero locally.

---

`\flag_show:n \flag_show:n {<flag name>}`

Displays the *<flag>*'s height in the terminal.

---

`\flag_log:n \flag_log:n {<flag name>}`

Writes the *<flag>*'s height to the log file.

## 21.2 Expandable flag commands

---

`\flag_if_exist_p:n ∗ \flag_if_exist:n {<flag name>}`

`\flag_if_exist:nTF` ∗ This function returns `true` if the *<flag name>* references a flag that has been defined previously, and `false` otherwise.

---

`\flag_if_raised_p:n ∗ \flag_if_raised:n {<flag name>}`

`\flag_if_raised:nTF` ∗ This function returns `true` if the *<flag>* has non-zero height, and `false` if the *<flag>* has zero height.

---

`\flag_height:n ∗ \flag_height:n {<flag name>}`

Expands to the height of the *<flag>* as an integer denotation.

---

`\flag_raise:n ∗ \flag_raise:n {<flag name>}`

The *<flag>*'s height is increased by 1 locally.

## Chapter 22

# The `l3clist` package

## Comma separated lists

Comma lists (in short, `clist`) contain ordered data where items can be added to the left or right end of the list. This data type allows basic list manipulations such as adding/removing items, applying a function to every item, removing duplicate items, extracting a given item, using the comma list with specified separators, and so on. Sequences (defined in `l3seq`) are safer, faster, and provide more features, so they should often be preferred to comma lists. Comma lists are mostly useful when interfacing with L<sup>A</sup>T<sub>E</sub>X 2<sub><</sub> or other code that expects or provides items separated by commas.

Several items can be added at once. To ease input of comma lists from data provided by a user outside an `\ExplSyntaxOn ... \ExplSyntaxOff` block, spaces are removed from both sides of each comma-delimited argument upon input. Blank arguments are ignored, to allow for trailing commas or repeated commas (which may otherwise arise when concatenating comma lists “by hand”). In addition, a set of braces is removed if the result of space-trimming is braced: this allows the storage of any item in a comma list. For instance,

```
\clist_new:N \l_my_clist
\clist_put_left:Nn \l_my_clist { ~a~ , ~{b}~ , c~\d }
\clist_put_right:Nn \l_my_clist { ~{e~} , , {{f}} , }
```

results in `\l_my_clist` containing `a,b,c~\d,{e~},{{f}}` namely the five items `a`, `b`, `c~\d`, `e~` and `{f}`. Comma lists normally do not contain empty or blank items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_set:Nn \l_my_clist { , ~ , , }
\clist_if_empty:NTF \l_my_clist { true } { false }
```

and it leaves `true` in the input stream. To include an “unsafe” item (empty, or one that contains a comma, or starts or ends with a space, or is a single brace group), surround it with braces.

Any `n`-type token list is a valid comma list input for `l3clist` functions, which will split the token list at every comma and process the items as described above. On the other hand, `N`-type functions expect comma list variables, which are particular token list variables in which this processing of items (and removal of blank items) has already

occurred. Because comma list variables are token list variables, expanding them once yields their items separated by commas, and  $\l3tl$  functions such as  $\l3tl_show:N$  can be applied to them. (These functions often have  $\l3clist$  analogues, which should be preferred.)

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using  $\lseq_set_from_clist:Nn$  (see  $\l3seq$ ) may be advisable if speed is important. The exception is that  $\l3clist_if_in:NnTF$  and  $\l3clist_remove_duplicates:N$  may be faster than their sequence analogues for large lists. However, these functions work slowly for “unsafe” items that must be braced, and may produce errors when their argument contains {, } or # (assuming the usual T<sub>E</sub>X category codes apply). The sequence data type should thus certainly be preferred to comma lists to store such items.

## 22.1 Creating and initialising comma lists

---

`\clist_new:N \clist_new:N <comma list>`

`\clist_new:c` Creates a new  $\langle\text{comma list}\rangle$  or raises an error if the name is already taken. The declaration is global. The  $\langle\text{comma list}\rangle$  initially contains no items.

---

`\clist_const:Nn \clist_const:Nn <clist var> <comma list>`

`\clist_const:(Nx|cn|cx)` Creates a new constant  $\langle\text{clist var}\rangle$  or raises an error if the name is already taken. The value of the  $\langle\text{clist var}\rangle$  is set globally to the  $\langle\text{comma list}\rangle$ .

---

`\clist_clear:N \clist_clear:N <comma list>`

`\clist_clear:c` Clears all items from the  $\langle\text{comma list}\rangle$ .  
`\clist_gclear:N`

---

`\clist_gclear:c`

---

`\clist_clear_new:N \clist_clear_new:N <comma list>`

`\clist_clear_new:c` Ensures that the  $\langle\text{comma list}\rangle$  exists globally by applying  $\l3clist_new:N$  if necessary, then applies  $\l3clist_(g)clear:N$  to leave the list empty.  
`\clist_gclear_new:N`

---

`\clist_gclear_new:c`

---

`\clist_set_eq:NN \clist_set_eq:NN <comma list1> <comma list2>`

`\clist_set_eq:(cN|Nc|cc)` Sets the content of  $\langle\text{comma list}_1\rangle$  equal to that of  $\langle\text{comma list}_2\rangle$ . To set a token list variable equal to a comma list variable, use  $\l3tl_set_eq:NN$ . Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.  
`\clist_gset_eq:NN`

---

`\clist_gset_eq:(cN|Nc|cc)`  $\l3clist_set_from_seq:NN$   $\l3clist_set_from_seq:Nn <\text{comma list}> <\text{sequence}>$

---

`\clist_gset_from_seq:NN`

---

`\clist_gset_from_seq:(cN|Nc|cc)`

---

`New: 2014-07-17`

Converts the data in the  $\langle\text{sequence}\rangle$  into a  $\langle\text{comma list}\rangle$ : the original  $\langle\text{sequence}\rangle$  is unchanged. Items which contain either spaces or commas are surrounded by braces.

---

```
\clist_concat:NNN          \clist_concat:NNN <comma list1> <comma list2> <comma list3>
\clist_concat:ccc          Concatenates the content of <comma list2> and <comma list3> together and saves the
\clist_gconcat:NNN          result in <comma list1>. The items in <comma list2> are placed at the left side of the new
\clist_gconcat:ccc          comma list.
```

---

```
\clist_if_exist_p:N * \clist_if_exist_p:N <comma list>
\clist_if_exist_p:c * \clist_if_exist:NTF <comma list> {<true code>} {<false code>}
\clist_if_exist:NTF * Tests whether the <comma list> is currently defined. This does not check that the <comma
\clist_if_exist:cTF * list> really is a comma list.
```

---

New: 2012-03-03

## 22.2 Adding data to comma lists

---

```
\clist_set:Nn               \clist_set:Nn <comma list> {<item1>, ..., <itemn>}
\clist_set:(NV|No|Nx|cn|cV|co|cx)
\clist_gset:Nn
\clist_gset:(NV|No|Nx|cn|cV|co|cx)
```

---

New: 2011-09-06

Sets <comma list> to contain the <items>, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some <tokens> as a single <item> even if the <tokens> contain commas or spaces, add a set of braces: \clist\_set:Nn <comma list> { <tokens> } .

---

```
\clist_put_left:Nn          \clist_put_left:Nn <comma list> {<item1>, ..., <itemn>}
\clist_put_left:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_left:Nn
\clist_gput_left:(NV|No|Nx|cn|cV|co|cx)
```

---

Updated: 2011-09-05

Appends the <items> to the left of the <comma list>. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some <tokens> as a single <item> even if the <tokens> contain commas or spaces, add a set of braces: \clist\_put\_left:Nn <comma list> { <tokens> } .

---

```
\clist_put_right:Nn          \clist_put_right:Nn <comma list> {<item1>, ..., <itemn>}
\clist_put_right:(NV|No|Nx|cn|cV|co|cx)
\clist_gput_right:Nn
\clist_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

---

Updated: 2011-09-05

Appends the <items> to the right of the <comma list>. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some <tokens> as a single <item> even if the <tokens> contain commas or spaces, add a set of braces: \clist\_put\_right:Nn <comma list> { <tokens> } .

## 22.3 Modifying comma lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

---

```
\clist_remove_duplicates:N \clist_remove_duplicates:N <comma list>
\clist_remove_duplicates:c
\clist_gremove_duplicates:N
\clist_gremove_duplicates:c
```

---

Removes duplicate items from the *<comma list>*, leaving the left most copy of each item in the *<comma list>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

**TeXhackers note:** This function iterates through every item in the *<comma list>* and does a comparison with the *<items>* already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the *<comma list>* contains {, }, or # (assuming the usual TeX category codes apply).

---

```
\clist_remove_all:Nn \clist_remove_all:Nn <comma list> {\<item>}
\clist_remove_all:(cn|NV|cV)
\clist_gremove_all:Nn
\clist_gremove_all:(cn|NV|cV)
```

---

Updated: 2011-09-06

Removes every occurrence of *<item>* from the *<comma list>*. The *<item>* comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

**TeXhackers note:** The function may fail if the *<item>* contains {, }, or # (assuming the usual TeX category codes apply).

---

```
\clist_reverse:N \clist_reverse:N <comma list>
\clist_reverse:c
\clist_greverse:N Reverses the order of items stored in the <comma list>.
\clist_greverse:c
```

---

New: 2014-07-18

---

```
\clist_reverse:n \clist_reverse:n <comma list>
```

---

New: 2014-07-18 Leaves the items in the *<comma list>* in the input stream in reverse order. Contrarily to other what is done for other n-type *<comma list>* arguments, braces and spaces are preserved by this process.

**TeXhackers note:** The result is returned within `\unexpanded`, which means that the comma list does not expand further when appearing in an x-type or e-type argument expansion.

---

```
\clist_sort:Nn          \clist_sort:Nn <clist var> {<comparison code>}
\clist_sort:cn          Sorts the items in the <clist var> according to the <comparison code>, and assigns the
\clist_gsort:Nn          result to <clist var>. The details of sorting comparison are described in Section 6.1.
\clist_gsort:cn
```

---

New: 2017-02-06

## 22.4 Comma list conditionals

---

```
\clist_if_empty_p:N *  \clist_if_empty_p:N <comma list>
\clist_if_empty_p:c *  \clist_if_empty:nTF <comma list> {<true code>} {<false code>}
\clist_if_empty:nTF *  Tests if the <comma list> is empty (containing no items).
\clist_if_empty:cTF *
```

---

---

```
\clist_if_empty_p:n *  \clist_if_empty_p:n <comma list>
\clist_if_empty:nTF *  \clist_if_empty:nTF <comma list> {<true code>} {<false code>}
```

---

New: 2014-07-05 Tests if the <comma list> is empty (containing no items). The rules for space trimming are as for other n-type comma-list functions, hence the comma list {~,~,~,~} (without outer braces) is empty, while {~,{},{} (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

---

```
\clist_if_in:NnTF          \clist_if_in:NnTF <comma list> {<item>} {<true code>} {<false code>}
\clist_if_in:(NV|No|cn|cV|co)TF
\clist_if_in:nnTF
\clist_if_in:(nV|no)TF
```

---

Updated: 2011-09-06

Tests if the <item> is present in the <comma list>. In the case of an n-type <comma list>, the usual rules of space trimming and brace stripping apply. Hence,

```
\clist_if_in:nnTF { a , {b}~ , {b} , c } { b } {true} {false}
```

yields true.

**TeXhackers note:** The function may fail if the <item> contains {, }, or # (assuming the usual TeX category codes apply).

## 22.5 Mapping over comma lists

The functions described in this section apply a specified function to each item of a comma list. All mappings are done at the current group level, i.e. any local assignments made by the <function> or <code> discussed below remain in effect after the loop.

When the comma list is given explicitly, as an n-type argument, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is {a, {{b}}, {}, {c}, } then the arguments passed to the mapped function are ‘a’, ‘{b}’, an empty argument, and ‘c’.

When the comma list is given as an N-type argument, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using n-type comma lists.

---

\clist\_map\_function:NN  $\star$  \clist\_map\_function:NN *<comma list>* *<function>*  
\clist\_map\_function:cN  $\star$  Applies *<function>* to every *<item>* stored in the *<comma list>*. The *<function>* receives one argument for each iteration. The *<items>* are returned from left to right. The function \clist\_map\_inline:Nn is in general more efficient than \clist\_map\_function:NN.  


---

Updated: 2012-06-29

---

\clist\_map\_inline:Nn \clist\_map\_inline:Nn *<comma list>* {*<inline function>*}  
\clist\_map\_inline:cn \clist\_map\_inline:nn Applies *<inline function>* to every *<item>* stored within the *<comma list>*. The *<inline function>* should consist of code which receives the *<item>* as #1. The *<items>* are returned from left to right.  


---

Updated: 2012-06-29

---

\clist\_map\_variable>NNn \clist\_map\_variable>NNn *<comma list>* *<variable>* {*<code>*}  
\clist\_map\_variable:cNn \clist\_map\_variable:nNn Stores each *<item>* of the *<comma list>* in turn in the (token list) *<variable>* and applies the *<code>*. The *<code>* will usually make use of the *<variable>*, but this is not enforced. The assignments to the *<variable>* are local. Its value after the loop is the last *<item>* in the *<comma list>*, or its original value if there were no *<item>*. The *<items>* are returned from left to right.  


---

Updated: 2012-06-29

---

\clist\_map\_tokens:Nn \clist\_map\_tokens:Nn *<clist var>* {*<code>*}  
\clist\_map\_tokens:cn \clist\_map\_tokens:nn {*<comma list>*} {*<code>*}  
\clist\_map\_tokens:nn  $\star$  Calls *<code>* {*<item>*} for every *<item>* stored in the *<comma list>*. The *<code>* receives each *<item>* as a trailing brace group. If the *<code>* consists of a single function this is equivalent to \clist\_map\_function:nN.  


---

New: 2021-05-05

---

\clist\_map\_break:  $\star$  \clist\_map\_break:  


---

Updated: 2012-06-29 Used to terminate a \clist\_map\_... function before all entries in the *<comma list>* have been processed. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \clist_map_break: }
  {
    % Do something useful
  }
}
```

Use outside of a \clist\_map\_... scenario leads to low level T<sub>E</sub>X errors.

**T<sub>E</sub>Xhackers note:** When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

---

```
\clist_map_break:n ☆ \clist_map_break:n {<code>}
```

Updated: 2012-06-29

Used to terminate a `\clist_map_...` function before all entries in the *<comma list>* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\clist_map_inline:Nn \l_my_clist
{
  \str_if_eq:nnTF { #1 } { bingo }
    { \clist_map_break:n { <code> } }
  {
    % Do something useful
  }
}
```

Use outside of a `\clist_map_...` scenario leads to low level TeX errors.

**TeXhackers note:** When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

---

```
\clist_count:N ★ \clist_count:N <comma list>
```

**\clist\_count:c ★** Leaves the number of items in the *<comma list>* in the input stream as an *<integer denotation>*. The total number of items in a *<comma list>* includes those which are

New: 2012-07-13 duplicates, *i.e.* every item in a *<comma list>* is counted.

## 22.6 Using the content of comma lists directly

---

```
\clist_use:Nnnn ★ \clist_use:Nnnn <clist var> {<separator between two>}
\clist_use:cnnn ★ {<separator between more than two>} {<separator between final two>}
```

New: 2013-05-26

Places the contents of the *<clist var>* in the input stream, with the appropriate *<separator>* between the items. Namely, if the comma list has more than two items, the *<separator between more than two>* is placed between each pair of items except the last, for which the *<separator between final two>* is used. If the comma list has exactly two items, then they are placed in the input stream separated by the *<separator between two>*. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nnnn \l_tmpa_clist { ~and~ } { ,~ } { ,~and~ }
```

inserts “**a, b, c, de, and f**” in the input stream. The first separator argument is not used in this case because the comma list has more than 2 items.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an **x**-type or **e**-type argument expansion.

---

```
\clist_use:Nn * \clist_use:Nn <clist var> {<separator>}
```

**New: 2013-05-26** Places the contents of the *<clist var>* in the input stream, with the *<separator>* between the items. If the comma list has a single item, it is placed in the input stream, and a comma list with no items produces no output. An error is raised if the variable does not exist or if it is invalid.

For example,

```
\clist_set:Nn \l_tmpa_clist { a , b , , c , {de} , f }
\clist_use:Nn \l_tmpa_clist { ~and~ }
```

inserts “a and b and c and de and f” in the input stream.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

```
\clist_use:nnnn * \clist_use:nnnn <comma list> {<separator between two>}
\clist_use:nn * {<separator between more than two>} {<separator between final two>}
\clist_use:nn <comma list> {<separator>}
```

**New: 2021-05-10**

Places the contents of the *<comma list>* in the input stream, with the appropriate *<separator>* between the items. As for `\clist_set:Nn`, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The *<separators>* are then inserted in the same way as for `\clist_use:Nnnn` and `\clist_use:Nn`, respectively.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<items>* do not expand further when appearing in an `x`-type or `e`-type argument expansion.

## 22.7 Comma lists as stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

---

```
\clist_get:NN \clist_get:NN <comma list> <token list variable>
\clist_get:cN \clist_get:cN <comma list> <token list variable>
\clist_get:NNTF \clist_get:NNTF <comma list> <token list variable>
\clist_get:cNTF \clist_get:cNTF <comma list> <token list variable>
```

**New: 2012-05-14** Stores the left-most item from a *<comma list>* in the *<token list variable>* without removing it from the *<comma list>*. The *<token list variable>* is assigned locally. In the non-branching version, if the *<comma list>* is empty the *<token list variable>* is set to the marker value `\q_no_value`.

**Updated: 2019-02-16**

---

```
\clist_pop:NN \clist_pop:NN <comma list> <token list variable>
\clist_pop:cN \clist_pop:cN <comma list> <token list variable>
```

**Updated: 2011-09-06** Pops the left-most item from a *<comma list>* into the *<token list variable>*, *i.e.* removes the item from the comma list and stores it in the *<token list variable>*. Both of the variables are assigned locally.

---

```
\clist_gpop:NN \clist_gpop:NN <comma list> <token list variable>
```

---

```
\clist_gpop:cN
```

Pops the left-most item from a *<comma list>* into the *<token list variable>*, i.e. removes the item from the comma list and stores it in the *<token list variable>*. The *<comma list>* is modified globally, while the assignment of the *<token list variable>* is local.

---

```
\clist_pop:NNTF \clist_pop:NNTF <comma list> <token list variable> {<true code>} {<false code>}
```

---

```
\clist_pop:cNTF
```

New: 2012-05-14 If the *<comma list>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<comma list>* is non-empty, pops the top item from the *<comma list>* in the *<token list variable>*, i.e. removes the item from the *<comma list>*. Both the *<comma list>* and the *<token list variable>* are assigned locally.

---

```
\clist_gpop:NNTF \clist_gpop:NNTF <comma list> <token list variable> {<true code>} {<false code>}
```

---

```
\clist_gpop:cNTF
```

New: 2012-05-14 If the *<comma list>* is empty, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<comma list>* is non-empty, pops the top item from the *<comma list>* in the *<token list variable>*, i.e. removes the item from the *<comma list>*. The *<comma list>* is modified globally, while the *<token list variable>* is assigned locally.

---

```
\clist_push:Nn \clist_push:Nn <comma list> {<items>}
```

---

```
\clist_push:(NV|No|Nx|cn|cV|co|cx)
```

---

```
\clist_gpush:Nn
```

---

```
\clist_gpush:(NV|No|Nx|cn|cV|co|cx)
```

Adds the *{<items>}* to the top of the *<comma list>*. Spaces are removed from both sides of each item as for any n-type comma list.

## 22.8 Using a single item

---

```
\clist_item:Nn * \clist_item:Nn <comma list> {<integer expression>}
```

---

```
\clist_item:cn *
```

---

```
\clist_item:nn *
```

---

```
New: 2014-07-17
```

Indexing items in the *<comma list>* from 1 at the top (left), this function evaluates the *<integer expression>* and leaves the appropriate item from the comma list in the input stream. If the *<integer expression>* is negative, indexing occurs from the bottom (right) of the comma list. When the *<integer expression>* is larger than the number of items in the *<comma list>* (as calculated by `\clist_count:N`) then the function expands to nothing.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<item>* does not expand further when appearing in an x-type or e-type argument expansion.

---

`\clist_rand_item:N` \* `\clist_rand_item:N` (*clist var*)  
`\clist_rand_item:c` \* `\clist_rand_item:n` {⟨comma list⟩}  
`\clist_rand_item:n` \* Selects a pseudo-random item of the ⟨comma list⟩. If the ⟨comma list⟩ has no item, the result is empty.  
New: 2016-12-06

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the ⟨item⟩ does not expand further when appearing in an `x`-type or `e`-type argument expansion.

## 22.9 Viewing comma lists

---

`\clist_show:N` `\clist_show:N` ⟨comma list⟩  
`\clist_show:c` Displays the entries in the ⟨comma list⟩ in the terminal.  
Updated: 2021-04-29

---

`\clist_show:n` `\clist_show:n` {⟨tokens⟩}  
Updated: 2013-08-03 Displays the entries in the comma list in the terminal.

---

`\clist_log:N` `\clist_log:N` ⟨comma list⟩  
`\clist_log:c` Writes the entries in the ⟨comma list⟩ in the log file. See also `\clist_show:N` which displays the result in the terminal.  
New: 2014-08-22  
Updated: 2021-04-29

---

`\clist_log:n` `\clist_log:n` {⟨tokens⟩}  
New: 2014-08-22 Writes the entries in the comma list in the log file. See also `\clist_show:n` which displays the result in the terminal.

## 22.10 Constant and scratch comma lists

---

`\c_empty_clist` Constant that is always empty.  
New: 2012-07-02

---

`\l_tmpa_clist` Scratch comma lists for local assignment. These are never used by the kernel code, and  
`\l_tmpb_clist` so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten  
New: 2011-09-06 by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_clist` Scratch comma lists for global assignment. These are never used by the kernel code, and  
`\g_tmpb_clist` so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten  
New: 2011-09-06 by other non-kernel code and so should only be used for short-term storage.

## Chapter 23

# The `I3token` package

## Token manipulation

This module deals with tokens. Now this is perhaps not the most precise description so let's try with a better description: When programming in TEX, it is often desirable to know just what a certain token is: is it a control sequence or something else. Similarly one often needs to know if a control sequence is expandable or not, a macro or a primitive, how many arguments it takes etc. Another thing of great importance (especially when it comes to document commands) is looking ahead in the token stream to see if a certain character is present and maybe even remove it or disregard other tokens while scanning. This module provides functions for both and as such has two primary function categories: `\token_` for anything that deals with tokens and `\peek_` for looking ahead in the token stream.

Most functions we describe here can be used on control sequences, as those are tokens as well.

It is important to distinguish two aspects of a token: its “shape” (for lack of a better word), which affects the matching of delimited arguments and the comparison of token lists containing this token, and its “meaning”, which affects whether the token expands or what operation it performs. One can have tokens of different shapes with the same meaning, but not the converse.

For instance, `\if:w`, `\if_charcode:w`, and `\tex_if:D` are three names for the same internal operation of TEX, namely the primitive testing the next two characters for equality of their character code. They have the same meaning hence behave identically in many situations. However, TEX distinguishes them when searching for a delimited argument. Namely, the example function `\show_until_if:w` defined below takes everything until `\if:w` as an argument, despite the presence of other copies of `\if:w` under different names.

```
\cs_new:Npn \show_until_if:w #1 \if:w { \tl_show:n {#1} }
\show_until_if:w \tex_if:D \if_charcode:w \if:w
```

A list of all possible shapes and a list of all possible meanings are given in section 23.7.

## 23.1 Creating character tokens

---

\char\_set\_active\_eq:NN  
\char\_set\_active\_eq:Nc  
\char\_gset\_active\_eq:NN  
\char\_gset\_active\_eq:Nc

---

Updated: 2015-11-12

\char\_set\_active\_eq:NN ⟨char⟩ ⟨function⟩

Sets the behaviour of the ⟨char⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨function⟩. The category code of the ⟨char⟩ is *unchanged* by this process. The ⟨function⟩ may itself be an active character.

---

\char\_set\_active\_eq:nN  
\char\_set\_active\_eq:nc  
\char\_gset\_active\_eq:nN  
\char\_gset\_active\_eq:nc

---

New: 2015-11-12

\char\_set\_active\_eq:nN {⟨integer expression⟩} ⟨function⟩

Sets the behaviour of the ⟨char⟩ which has character code as given by the ⟨integer expression⟩ in situations where it is active (category code 13) to be equivalent to that of the ⟨function⟩. The category code of the ⟨char⟩ is *unchanged* by this process. The ⟨function⟩ may itself be an active character.

---

\char\_generate:nn ∗ \char\_generate:nn {⟨charcode⟩} {⟨catcode⟩}

---

New: 2015-09-09  
Updated: 2019-01-16

Generates a character token of the given ⟨charcode⟩ and ⟨catcode⟩ (both of which may be integer expressions). The ⟨catcode⟩ may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The ⟨charcode⟩ may be any one valid for the engine in use, except that for ⟨catcode⟩ 10, ⟨charcode⟩ 0 is not allowed. Active characters cannot be generated in older versions of X<sub>E</sub>T<sub>E</sub>X. Another way to build token lists with unusual category codes is \regex\_replace:nnN {.\*} {⟨replacement⟩} ⟨tl var⟩.

**TeXhackers note:** Exactly two expansions are needed to produce the character.

---

\c\_catcode\_other\_space\_tl  
New: 2011-09-05

Token list containing one character with category code 12, (“other”), and character code 32 (space).

## 23.2 Manipulating and interrogating character tokens

---

```
\char_set_catcode_escape:N          \char_set_catcode_letter:N <character>
\char_set_catcode_group_begin:N
\char_set_catcode_group_end:N
\char_set_catcode_math_toggle:N
\char_set_catcode_alignment:N
\char_set_catcode_end_line:N
\char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
\char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
\char_set_catcode_space:N
\char_set_catcode_letter:N
\char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
```

---

Updated: 2015-11-11

Sets the category code of the *<character>* to that indicated in the function name. Depending on the current category code of the *<token>* the escape token may also be needed:

`\char_set_catcode_other:N \%`

The assignment is local.

---

```
\char_set_catcode_escape:n          \char_set_catcode_letter:n {<integer expression>}
\char_set_catcode_group_begin:n
\char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
\char_set_catcode_alignment:n
\char_set_catcode_end_line:n
\char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
\char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
\char_set_catcode_space:n
\char_set_catcode_letter:n
\char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
```

---

Updated: 2015-11-11

Sets the category code of the *<character>* which has character code as given by the *<integer expression>*. This version can be used to set up characters which cannot otherwise be given (*cf.* the N-type variants). The assignment is local.

---

`\char_set_catcode:nn \char_set_catcode:nn {integer expression} {integer expression}`

Updated: 2015-11-11

These functions set the category code of the *character* which has character code as given by the *integer expression*. The first *integer expression* is the character code and the second is the category code to apply. The setting applies within the current TeX group. In general, the symbolic functions `\char_set_catcode_<type>` should be preferred, but there are cases where these lower-level functions may be useful.

---

`\char_value_catcode:n ∗ \char_value_catcode:n {integer expression}`

Expands to the current category code of the *character* with character code given by the *integer expression*.

---

`\char_show_value_catcode:n \char_show_value_catcode:n {integer expression}`

Displays the current category code of the *character* with character code given by the *integer expression* on the terminal.

---

`\char_set_lccode:nn \char_set_lccode:nn {integer expression} {integer expression}`

Updated: 2015-08-06

Sets up the behaviour of the *character* when found inside `\text_lowercase:n`, such that *character*<sub>1</sub> will be converted into *character*<sub>2</sub>. The two *characters* may be specified using an *integer expression* for the character code concerned. This may include the TeX ‘*character*’ method for converting a single character into its character code:

```
\char_set_lccode:nn { '\A' } { '\a' } % Standard behaviour
\char_set_lccode:nn { '\A' } { '\A + 32' }
\char_set_lccode:nn { 50 } { 60 }
```

The setting applies within the current TeX group.

---

`\char_value_lccode:n ∗ \char_value_lccode:n {integer expression}`

Expands to the current lower case code of the *character* with character code given by the *integer expression*.

---

`\char_show_value_lccode:n \char_show_value_lccode:n {integer expression}`

Displays the current lower case code of the *character* with character code given by the *integer expression* on the terminal.

---

`\char_set_uccode:nn \char_set_uccode:nn {integer expression} {integer expression}`

Updated: 2015-08-06

Sets up the behaviour of the *character* when found inside `\text_uppercase:n`, such that *character*<sub>1</sub> will be converted into *character*<sub>2</sub>. The two *characters* may be specified using an *integer expression* for the character code concerned. This may include the TeX ‘*character*’ method for converting a single character into its character code:

```
\char_set_uccode:nn { '\a' } { '\A' } % Standard behaviour
\char_set_uccode:nn { '\A' } { '\A - 32' }
\char_set_uccode:nn { 60 } { 50 }
```

The setting applies within the current TeX group.

<code>\char_value_uccode:n</code>	<code>\char_value_uccode:n {&lt;integer expression&gt;}</code>	Expands to the current upper case code of the <i>&lt;character&gt;</i> with character code given by the <i>&lt;integer expression&gt;</i> .
<code>\char_show_value_uccode:n</code>	<code>\char_show_value_uccode:n {&lt;integer expression&gt;}</code>	Displays the current upper case code of the <i>&lt;character&gt;</i> with character code given by the <i>&lt;integer expression&gt;</i> on the terminal.
<code>\char_set_mathcode:nn</code>	<code>\char_set_mathcode:nn {&lt;intexpr1&gt;} {&lt;intexpr2&gt;}</code>	This function sets up the math code of <i>&lt;character&gt;</i> . The <i>&lt;character&gt;</i> is specified as an <i>&lt;integer expression&gt;</i> which will be used as the character code of the relevant character. The setting applies within the current TeX group.
<code>\char_value_mathcode:n</code>	<code>\char_value_mathcode:n {&lt;integer expression&gt;}</code>	Expands to the current math code of the <i>&lt;character&gt;</i> with character code given by the <i>&lt;integer expression&gt;</i> .
<code>\char_show_value_mathcode:n</code>	<code>\char_show_value_mathcode:n {&lt;integer expression&gt;}</code>	Displays the current math code of the <i>&lt;character&gt;</i> with character code given by the <i>&lt;integer expression&gt;</i> on the terminal.
<code>\char_set_sfcode:nn</code>	<code>\char_set_sfcode:nn {&lt;intexpr1&gt;} {&lt;intexpr2&gt;}</code>	This function sets up the space factor for the <i>&lt;character&gt;</i> . The <i>&lt;character&gt;</i> is specified as an <i>&lt;integer expression&gt;</i> which will be used as the character code of the relevant character. The setting applies within the current TeX group.
<code>\char_value_sfcode:n</code>	<code>\char_value_sfcode:n {&lt;integer expression&gt;}</code>	Expands to the current space factor for the <i>&lt;character&gt;</i> with character code given by the <i>&lt;integer expression&gt;</i> .
<code>\char_show_value_sfcode:n</code>	<code>\char_show_value_sfcode:n {&lt;integer expression&gt;}</code>	Displays the current space factor for the <i>&lt;character&gt;</i> with character code given by the <i>&lt;integer expression&gt;</i> on the terminal.
<code>\l_char_active_seq</code>	New: 2012-01-23 Updated: 2015-11-11	Used to track which tokens may require special handling at the document level as they are (or have been at some point) of category <i>&lt;active&gt;</i> (catcode 13). Each entry in the sequence consists of a single escaped token, for example <code>\~{}</code> . Active tokens should be added to the sequence when they are defined for general document use.
<code>\l_char_special_seq</code>	New: 2012-01-23 Updated: 2015-11-11	Used to track which tokens will require special handling when working with verbatim-like material at the document level as they are not of categories <i>&lt;letter&gt;</i> (catcode 11) or <i>&lt;other&gt;</i> (catcode 12). Each entry in the sequence consists of a single escaped token, for example <code>\\"{}</code> for the backslash or <code>\{\}</code> for an opening brace. Escaped tokens should be added to the sequence when they are defined for general document use.

## 23.3 Generic tokens

---

```
\c_group_begin_token
\c_group_end_token
\c_math_toggle_token
\c_alignment_token
\c_parameter_token
\c_math_superscript_token
\c_math_subscript_token
\c_space_token
```

---

These are implicit tokens which have the category code described by their name. They are used internally for test purposes but are also available to the programmer for other uses.

---

```
\c_catcode_letter_token
\c_catcode_other_token
```

---

These are implicit tokens which have the category code described by their name. They are used internally for test purposes and should not be used other than for category code tests.

---

```
\c_catcode_active_tl
```

---

A token list containing an active token. This is used internally for test purposes and should not be used other than in appropriately-constructed category code tests.

## 23.4 Converting tokens

---

```
\token_to_meaning:N * \token_to_meaning:N <token>
\token_to_meaning:c *
```

---

Inserts the current meaning of the *<token>* into the input stream as a series of characters of category code 12 (other). This is the primitive TeX description of the *<token>*, thus for example both functions defined by `\cs_set_nopar:Npn` and token list variables defined using `\tl_new:N` are described as **macros**.

**TeXhackers note:** This is the TeX primitive `\meaning`. The *<token>* can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal TeX category codes apply) even though these are not valid N-type arguments.

---

```
\token_to_str:N * \token_to_str:N <token>
```

---

---

```
\token_to_str:c *
```

---

Converts the given *<token>* into a series of characters with category code 12 (other). If the *<token>* is a control sequence, this will start with the current escape character with category code 12 (the escape character is part of the *<token>*). This function requires only a single expansion.

**TeXhackers note:** `\token_to_str:N` is the TeX primitive `\string` renamed. The *<token>* can thus be an explicit space tokens or an explicit begin-group or end-group character token (`{` or `}` when normal TeX category codes apply) even though these are not valid N-type arguments.

## 23.5 Token conditionals

---

```
\token_if_group_begin_p:N * \token_if_group_begin_p:N <token>
\token_if_group_begin:NTF * \token_if_group_begin:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of a begin group token ({ when normal TeX category codes are in force). Note that an explicit begin group token cannot be tested in this way, as it is not a valid N-type argument.

---

```
\token_if_group_end_p:N * \token_if_group_end_p:N <token>
\token_if_group_end:NTF * \token_if_group_end:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of an end group token (} when normal TeX category codes are in force). Note that an explicit end group token cannot be tested in this way, as it is not a valid N-type argument.

---

```
\token_if_math_toggle_p:N * \token_if_math_toggle_p:N <token>
\token_if_math_toggle:NTF * \token_if_math_toggle:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of a math shift token (\$) when normal TeX category codes are in force).

---

```
\token_if_alignment_p:N * \token_if_alignment_p:N <token>
\token_if_alignment:NTF * \token_if_alignment:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of an alignment token (& when normal TeX category codes are in force).

---

```
\token_if_parameter_p:N * \token_if_parameter_p:N <token>
\token_if_parameter:NTF * \token_if_parameter:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of a macro parameter token (# when normal TeX category codes are in force).

---

```
\token_if_math_superscript_p:N * \token_if_math_superscript_p:N <token>
\token_if_math_superscript:NTF * \token_if_math_superscript:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of a superscript token (^ when normal TeX category codes are in force).

---

```
\token_if_math_subscript_p:N * \token_if_math_subscript_p:N <token>
\token_if_math_subscript:NTF * \token_if_math_subscript:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of a subscript token (\_) when normal TeX category codes are in force).

---

```
\token_if_space_p:N * \token_if_space_p:N <token>
\token_if_space:NTF * \token_if_space:NTF <token> {\{true code\}} {\{false code\}}
```

---

Tests if *<token>* has the category code of a space token. Note that an explicit space token with character code 32 cannot be tested in this way, as it is not a valid N-type argument.

<code>\token_if_letter_p:N * \token_if_letter_p:N &lt;token&gt;</code>	<code>\token_if_letter:NTF * \token_if_letter:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if <i>&lt;token&gt;</i> has the category code of a letter token.	
<code>\token_if_other_p:N * \token_if_other_p:N &lt;token&gt;</code>	<code>\token_if_other:NTF * \token_if_other:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if <i>&lt;token&gt;</i> has the category code of an “other” token.	
<code>\token_if_active_p:N * \token_if_active_p:N &lt;token&gt;</code>	<code>\token_if_active:NTF * \token_if_active:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if <i>&lt;token&gt;</i> has the category code of an active character.	
<code>\token_if_eq_catcode_p&gt;NN * \token_if_eq_catcode_p&gt;NN &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt;</code>	<code>\token_if_eq_catcode:NNTF * \token_if_eq_catcode:NNTF &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {\{true code\}} {\{false code\}}</code>
Tests if the two <i>&lt;tokens&gt;</i> have the same category code.	
<code>\token_if_eq_charcode_p&gt;NN * \token_if_eq_charcode_p&gt;NN &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt;</code>	<code>\token_if_eq_charcode:NNTF * \token_if_eq_charcode:NNTF &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {\{true code\}} {\{false code\}}</code>
Tests if the two <i>&lt;tokens&gt;</i> have the same character code.	
<code>\token_if_eq_meaning_p&gt;NN * \token_if_eq_meaning_p&gt;NN &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt;</code>	<code>\token_if_eq_meaning:NNTF * \token_if_eq_meaning:NNTF &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {\{true code\}} {\{false code\}}</code>
Tests if the two <i>&lt;tokens&gt;</i> have the same meaning when expanded.	
<code>\token_if_macro_p:N * \token_if_macro_p:N &lt;token&gt;</code>	<code>\token_if_macro:NTF * \token_if_macro:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Updated: 2011-05-23 Tests if the <i>&lt;token&gt;</i> is a TeX macro.	
<code>\token_if_cs_p:N * \token_if_cs_p:N &lt;token&gt;</code>	<code>\token_if_cs:NTF * \token_if_cs:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if the <i>&lt;token&gt;</i> is a control sequence.	
<code>\token_if_expandable_p:N * \token_if_expandable_p:N &lt;token&gt;</code>	<code>\token_if_expandable:NTF * \token_if_expandable:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if the <i>&lt;token&gt;</i> is expandable. This test returns <i>false</i> for an undefined token.	
<code>\token_if_long_macro_p:N * \token_if_long_macro_p:N &lt;token&gt;</code>	<code>\token_if_long_macro:NTF * \token_if_long_macro:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Updated: 2012-01-20 Tests if the <i>&lt;token&gt;</i> is a long macro.	
<code>\token_if_protected_macro_p:N * \token_if_protected_macro_p:N &lt;token&gt;</code>	<code>\token_if_protected_macro:NTF * \token_if_protected_macro:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Updated: 2012-01-20 Tests if the <i>&lt;token&gt;</i> is a protected macro: for a macro which is both protected and long this returns <i>false</i> .	

---

```
\token_if_protected_long_macro_p:N * \token_if_protected_long_macro_p:N <token>
\token_if_protected_long_macro:NTF * \token_if_protected_long_macro:NTF <token> {\{true code\}} {\{false
code\}}
```

---

Updated: 2012-01-20

Tests if the *<token>* is a protected long macro.

---

```
\token_if_chardef_p:N * \token_if_chardef_p:N <token>
\token_if_chardef:NTF * \token_if_chardef:NTF <token> {\{true code\}} {\{false code\}}
```

---

Updated: 2012-01-20

Tests if the *<token>* is defined to be a chardef.

**TeXhackers note:** Booleans, boxes and small integer constants are implemented as `\chardefs`.

---

```
\token_if_mathchardef_p:N * \token_if_mathchardef_p:N <token>
\token_if_mathchardef:NTF * \token_if_mathchardef:NTF <token> {\{true code\}} {\{false code\}}
```

---

Updated: 2012-01-20

Tests if the *<token>* is defined to be a mathchardef.

---

```
\token_if_font_selection_p:N * \token_if_font_selection_p:N <token>
\token_if_font_selection:NTF * \token_if_font_selection:NTF <token> {\{true code\}} {\{false code\}}
```

---

New: 2020-10-27

Tests if the *<token>* is defined to be a font selection command.

---

```
\token_if_dim_register_p:N * \token_if_dim_register_p:N <token>
\token_if_dim_register:NTF * \token_if_dim_register:NTF <token> {\{true code\}} {\{false code\}}
```

---

Updated: 2012-01-20

Tests if the *<token>* is defined to be a dimension register.

---

```
\token_if_int_register_p:N * \token_if_int_register_p:N <token>
\token_if_int_register:NTF * \token_if_int_register:NTF <token> {\{true code\}} {\{false code\}}
```

---

Updated: 2012-01-20

Tests if the *<token>* is defined to be a integer register.

**TeXhackers note:** Constant integers may be implemented as integer registers, `\chardefs`, or `\mathchardefs` depending on their value.

---

```
\token_if_muskip_register_p:N * \token_if_muskip_register_p:N <token>
\token_if_muskip_register:NTF * \token_if_muskip_register:NTF <token> {\{true code\}} {\{false code\}}
```

---

New: 2012-02-15

Tests if the *<token>* is defined to be a muskip register.

---

```
\token_if_skip_register_p:N * \token_if_skip_register_p:N <token>
\token_if_skip_register:NTF * \token_if_skip_register:NTF <token> {\<true code>} {\<false code>}
```

---

Updated: 2012-01-20

Tests if the *<token>* is defined to be a skip register.

---

```
\token_if_toks_register_p:N * \token_if_toks_register_p:N <token>
\token_if_toks_register:NTF * \token_if_toks_register:NTF <token> {\<true code>} {\<false code>}
```

---

Updated: 2012-01-20

Tests if the *<token>* is defined to be a toks register (not used by L<sup>A</sup>T<sub>E</sub>X3).

---

```
\token_if_primitive_p:N * \token_if_primitive_p:N <token>
\token_if_primitive:NTF * \token_if_primitive:NTF <token> {\<true code>} {\<false code>}
```

---

Updated: 2020-09-11 Tests if the *<token>* is an engine primitive. In L<sup>A</sup>T<sub>E</sub>X this includes primitive-like commands defined using `{token.set_lua}`.

---

```
\token_case_catcode:Nn    * \token_case_meaning:NnTF <test token>
\token_case_catcode:NnTF * {
\token_case_charcode:Nn   *   <token case1> {\<code case1>}
\token_case_charcode:NnTF *   <token case2> {\<code case2>}
\token_case_meaning:Nn    *   ...
\token_case_meaning:NnTF *   <token casen> {\<code casen>}
}
```

---

New: 2020-12-03

{<true code>}  
{<false code>}

This function compares the *<test token>* in turn with each of the *<token cases>*. If the two are equal (as described for `\token_if_eq_catcode:NNTF`, `\token_if_eq_charcode:NNTF` and `\token_if_eq_meaning:NNTF`, respectively) then the associated *<code>* is left in the input stream and other cases are discarded. If any of the cases are matched, the *<true code>* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *<false code>* is inserted. The functions `\token_case_catcode:Nn`, `\token_case_charcode:Nn`, and `\token_case_meaning:Nn`, which do nothing if there is no match, are also available.

## 23.6 Peeking ahead at the next token

There is often a need to look ahead at the next token in the input stream while leaving it in place. This is handled using the “peek” functions. The generic `\peek_after:Nw` is provided along with a family of predefined tests for common cases. As peeking ahead does *not* skip spaces the predefined tests include both a space-respecting and space-skipping version. In addition, using `\peek_analysis_map_inline:n`, one can map through the following tokens in the input stream and repeatedly perform some tests.

---

```
\peek_after:Nw \peek_after:Nw <function> <token>
```

---

Locally sets the test variable `\l_peek_token` equal to *<token>* (as an implicit token, *not* as a token list), and then expands the *<function>*. The *<token>* remains in the input stream as the next item after the *<function>*. The *<token>* here may be `,`, `{` or `}` (assuming normal T<sub>E</sub>X category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

---

**\peek\_gafter:Nw** **\peek\_gafter:Nw** *<function>* *<token>*

Globally sets the test variable `\g_peek_token` equal to *<token>* (as an implicit token, *not* as a token list), and then expands the *<function>*. The *<token>* remains in the input stream as the next item after the *<function>*. The *<token>* here may be `,`, `{` or `}` (assuming normal TeX category codes), *i.e.* it is not necessarily the next argument which would be grabbed by a normal function.

---

**\l\_peek\_token** Token set by `\peek_after:Nw` and available for testing as described above.

---

**\g\_peek\_token** Token set by `\peek_gafter:Nw` and available for testing as described above.

---

**\peek\_catcode:NTF** **\peek\_catcode:NTF** *<test token>* *{<true code>}* *{<false code>}*

---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

---

**\peek\_catcode\_remove:NTF** **\peek\_catcode\_remove:NTF** *<test token>* *{<true code>}* *{<false code>}*

---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

---

**\peek\_charcode:NTF** **\peek\_charcode:NTF** *<test token>* *{<true code>}* *{<false code>}*

---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

---

**\peek\_charcode\_remove:NTF** **\peek\_charcode\_remove:NTF** *<test token>* *{<true code>}* *{<false code>}*

---

Updated: 2012-12-20

Tests if the next *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test `\token_if_eq_charcode:NNTF`). Spaces are respected by the test and the *<token>* is removed from the input stream if the test is true. The function then places either the *<true code>* or *<false code>* in the input stream (as appropriate to the result of the test).

---

**\peek\_meaning:NTF** **\peek\_meaning:NTF** *<test token>* *{<true code>}* *{<false code>}*

---

Updated: 2011-07-02

Tests if the next *<token>* in the input stream has the same meaning as the *<test token>* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *<token>* is left in the input stream after the *<true code>* or *<false code>* (as appropriate to the result of the test).

---

`\peek_meaning_remove:NTF` `\peek_meaning_remove:NTF {test token} {\(true code\)} {\(false code\)}`

Updated: 2011-07-02

Tests if the next *token* in the input stream has the same meaning as the *test token* (as defined by the test `\token_if_eq_meaning:NNTF`). Spaces are respected by the test and the *token* is removed from the input stream if the test is true. The function then places either the *true code* or *false code* in the input stream (as appropriate to the result of the test).

---

`\peek_remove_spaces:n` `\peek_remove_spaces:n {\code}`

New: 2018-10-01

Peeks ahead and detect if the following token is a space (category code 10 and character code 32). If so, removes the token and checks the next token. Once a non-space token is found, the *code* will be inserted into the input stream. Typically this will contain a `\peek` operation, but this is not required.

---

`\peek_remove_filler:n` `\peek_remove_filler:n {\code}`

New: 2022-01-10

Peeks ahead and detect if the following token is a space (category code 10) or has meaning equal to `\scan_stop:`. If so, removes the token and checks the next token. If neither of these cases apply, expands the next token using f-type expansion, then checks the resulting leading token in the same way. If after expansion the next token is neither of the two test cases, the *code* will be inserted into the input stream. Typically this will contain a `\peek` operation, but this is not required.

**TeXhackers note:** This is essentially a macro-based implementation of how TeX handles the search for a left brace after for example `\everypar`, except that any non-expandable token cleanly ends the *filler* (i.e. it does not lead to a TeX error).

In contrast to TeX's filler removal, a construct `\exp_not:N \foo` will be treated in the same way as `\foo`.

---

`\peek_N_type:TF` `\peek_N_type:TF {\(true code\)} {\(false code\)}`

Updated: 2012-12-20

Tests if the next *token* in the input stream can be safely grabbed as an N-type argument. The test is *false* if the next *token* is either an explicit or implicit begin-group or end-group token (with any character code), or an explicit or implicit space character (with character code 32 and category code 10), or an outer token (never used in L<sup>A</sup>T<sub>E</sub>X3) and *true* in all other cases. Note that a *true* result ensures that the next *token* is a valid N-type argument. However, if the next *token* is for instance `\c_space_token`, the test takes the *false* branch, even though the next *token* is in fact a valid N-type argument. The *token* is left in the input stream after the *true code* or *false code* (as appropriate to the result of the test).

---

`\peek_analysis_map_inline:n \peek_analysis_map_inline:n {⟨inline function⟩}`

New: 2020-12-03

Updated: 2022-10-03

---

Repeatedly removes one  $\langle token \rangle$  from the input stream and applies the  $\langle inline function \rangle$  to it, until `\peek_analysis_map_break:` is called. The  $\langle inline function \rangle$  receives three arguments for each  $\langle token \rangle$  in the input stream:

- $\langle tokens \rangle$ , which both o-expand and x-expand to the  $\langle token \rangle$ . The detailed form of  $\langle tokens \rangle$  may change in later releases.
- $\langle char code \rangle$ , a decimal representation of the character code of the  $\langle token \rangle$ , -1 if it is a control sequence.
- $\langle catcode \rangle$ , a capital hexadecimal digit which denotes the category code of the  $\langle token \rangle$  (0: control sequence, 1: begin-group, 2: end-group, 3: math shift, 4: alignment tab, 6: parameter, 7: superscript, 8: subscript, A: space, B: letter, C: other, D: active). This can be converted to an integer by writing " $\langle catcode \rangle$ ".

These arguments are the same as for `\tl_analysis_map_inline:nn` defined in `\l3tl-analysis`. The  $\langle char code \rangle$  and  $\langle catcode \rangle$  do not take the meaning of a control sequence or active character into account: for instance, upon encountering the token `\c_group_begin_token` in the input stream, `\peek_analysis_map_inline:n` calls the  $\langle inline function \rangle$  with #1 being `\exp_not:n { \c_group_begin_token }` (with the current implementation), #2 being -1, and #3 being 0, as for any other control sequence. In contrast, upon encountering an explicit begin-group token {, the  $\langle inline function \rangle$  is called with arguments `\exp_after:wN { \if_false: } \fi:, 123` and 1.

The mapping is done at the current group level, i.e. any local assignments made by the  $\langle inline function \rangle$  remain in effect after the loop. Within the code, `\l1_peek_token` is set equal (as a token, not a token list) to the token under consideration.

---

`\peek_analysis_map_break: \peek_analysis_map_inline:n  
\peek_analysis_map_break:n { ... \peek_analysis_map_break:n {⟨code⟩} }`

New: 2020-12-03 Stops the `\peek_analysis_map_inline:n` loop from seeking more tokens, and inserts  $\langle code \rangle$  in the input stream (empty for `\peek_analysis_map_break:`).

---

`\peek_regex:nTF \peek_regex:nTF {⟨regex⟩} {⟨true code⟩} {⟨false code⟩}`

`\peek_regex:NTF` Tests if the  $\langle tokens \rangle$  that follow in the input stream match the  $\langle regular expression \rangle$ . Any  $\langle tokens \rangle$  that have been read are left in the input stream after the  $\langle true code \rangle$  or  $\langle false code \rangle$  (as appropriate to the result of the test). See `\l3regex` for documentation of the syntax of regular expressions. The  $\langle regular expression \rangle$  is implicitly anchored at the start, so for instance `\peek_regex:nTF { a }` is essentially equivalent to `\peek_charcode:NTF a`.

**TEXhackers note:** Implicit character tokens are correctly considered by `\peek_regex:nTF` as control sequences, while functions that inspect individual tokens (for instance `\peek_charcode:NTF`) only take into account their meaning.

The `\peek_regex:nTF` function only inspects as few tokens as necessary to determine whether the regular expression matches. For instance `\peek_regex:nTF { abc | [a-z] } { } abc` will only inspect the first token a even though the first branch abc of the alternative is preferred in functions such as `\peek_regex_remove_once:n`. This may have an effect on tokenization if the input stream has not yet been tokenized and category codes are changed.

---

```
\peek_regex_remove_once:nTF \peek_regex_remove_once:nTF {\<regex>} {\<true code>} {\<false code>}  
\peek_regex_remove_once:NTF
```

---

New: 2020-12-03

Tests if the  $\langle tokens \rangle$  that follow in the input stream match the  $\langle regex \rangle$ . If the test is true, the  $\langle tokens \rangle$  are removed from the input stream and the  $\langle true code \rangle$  is inserted, while if the test is false, the  $\langle false code \rangle$  is inserted followed by the  $\langle tokens \rangle$  that were originally in the input stream. See  $\backslash 3\text{regex}$  for documentation of the syntax of regular expressions. The  $\langle regular expression \rangle$  is implicitly anchored at the start, so for instance  $\backslash \text{peek\_regex\_remove\_once:nTF} \{ \text{a} \}$  is essentially equivalent to  $\backslash \text{peek\_charcode\_remove:NTF} \text{ a}$ .

**TeXhackers note:** Implicit character tokens are correctly considered by  $\backslash \text{peek\_regex\_remove\_once:nTF}$  as control sequences, while functions that inspect individual tokens (for instance  $\backslash \text{peek\_charcode:NTF}$ ) only take into account their meaning.

---

```
\peek_regex_replace_once:nn \peek_regex_replace_once:nnTF {\<regex>} {\<replacement>} {\<true code>}  
\peek_regex_replace_once:nnTF {\<false code>}  
\peek_regex_replace_once:Nn  
\peek_regex_replace_once:NnTF
```

---

New: 2020-12-03

If the  $\langle tokens \rangle$  that follow in the input stream match the  $\langle regex \rangle$ , replaces them according to the  $\langle replacement \rangle$  as for  $\backslash \text{regex\_replace\_once:nnN}$ , and leaves the result in the input stream, after the  $\langle true code \rangle$ . Otherwise, leaves  $\langle false code \rangle$  followed by the  $\langle tokens \rangle$  that were originally in the input stream, with no modifications. See  $\backslash 3\text{regex}$  for documentation of the syntax of regular expressions and of the  $\langle replacement \rangle$ : for instance  $\backslash 0$  in the  $\langle replacement \rangle$  is replaced by the tokens that were matched in the input stream. The  $\langle regular expression \rangle$  is implicitly anchored at the start. In contrast to  $\backslash \text{regex\_replace\_once:nnN}$ , no error arises if the  $\langle replacement \rangle$  leads to an unbalanced token list: the tokens are inserted into the input stream without issue.

**TeXhackers note:** Implicit character tokens are correctly considered by  $\backslash \text{peek\_regex\_replace\_once:nnTF}$  as control sequences, while functions that inspect individual tokens (for instance  $\backslash \text{peek\_charcode:NTF}$ ) only take into account their meaning.

## 23.7 Description of all possible tokens

Let us end by reviewing every case that a given token can fall into. This section is quite technical and some details are only meant for completeness. We distinguish the meaning of the token, which controls the expansion of the token and its effect on TeX's state, and its shape, which is used when comparing token lists such as for delimited arguments. Two tokens of the same shape must have the same meaning, but the converse does not hold.

A token has one of the following shapes.

- A control sequence, characterized by the sequence of characters that constitute its name: for instance,  $\backslash \text{use:n}$  is a five-letter control sequence.

- An active character token, characterized by its character code (between 0 and 1114111 for LuaTeX and XeTeX and less for other engines) and category code 13.
- A character token, characterized by its character code and category code (one of 1, 2, 3, 4, 6, 7, 8, 10, 11 or 12 whose meaning is described below).

There are also a few internal tokens. The following list may be incomplete in some engines.

- Expanding `\the\font` results in a token that looks identical to the command that was used to select the current font (such as `\tenrm`) but it differs from it in shape.
- A “frozen” `\relax`, which differs from the primitive in shape (but has the same meaning), is inserted when the closing `\fi` of a conditional is encountered before the conditional is evaluated.
- Expanding `\noexpand <token>` (when the `<token>` is expandable) results in an internal token, displayed (temporarily) as `\notexpanded: <token>`, whose shape coincides with the `<token>` and whose meaning differs from `\relax`.
- An `\outer endtemplate:` can be encountered when peeking ahead at the next token; this expands to another internal token, `end of alignment template`.
- Tricky programming might access a frozen `\endwrite`.
- Some frozen tokens can only be accessed in interactive sessions: `\cr`, `\right`, `\endgroup`, `\fi`, `\inaccessible`.
- In LuaTeX, there is also the strange case of “bytes” `~~~~~1100xy` where  $x, y$  are any two lowercase hexadecimal digits, so that the hexadecimal number ranges from `\text{110000}=1114112$` to `$1100ff = 1114367`. These are used to output individual bytes to files, rather than UTF-8. For the purposes of token comparisons they behave like non-expandable primitive control sequences (*not characters*) whose `\meaning` is `the_character_` followed by the given byte. If this byte is in the range `80–ff` this gives an “invalid utf-8 sequence” error: applying `\token_to_str:N` or `\token_to_meaning:N` to these tokens is unsafe. Unfortunately, they don’t seem to be detectable safely by any means except perhaps Lua code.

The meaning of a (non-active) character token is fixed by its category code (and character code) and cannot be changed. We call these tokens *explicit* character tokens. Category codes that a character token can have are listed below by giving a sample output of the TeX primitive `\meaning`, together with their L<sup>A</sup>T<sub>E</sub>X3 names and most common example:

- 1 begin-group character (`group_begin`, often `{`),
- 2 end-group character (`group_end`, often `}`),
- 3 math shift character (`math_toggle`, often `$`),
- 4 alignment tab character (`alignment`, often `&`),
- 6 macro parameter character (`parameter`, often `#`),
- 7 superscript character (`math_superscript`, often `^`),

- 8 subscript character (`math_subscript`, often `_`),
- 10 blank space (`space`, often character code 32),
- 11 the letter (`letter`, such as `A`),
- 12 the character (`other`, such as `0`).

Category code 13 (`active`) is discussed below. Input characters can also have several other category codes which do not lead to character tokens for later processing: 0 (`escape`), 5 (`end_line`), 9 (`ignore`), 14 (`comment`), and 15 (`invalid`).

The meaning of a control sequence or active character can be identical to that of any character token listed above (with any character code), and we call such tokens *implicit* character tokens. The meaning is otherwise in the following list:

- a macro, used in L<sup>A</sup>T<sub>E</sub>X3 for most functions and some variables (`t1`, `fp`, `seq`, ...),
- a primitive such as `\def` or `\topmark`, used in L<sup>A</sup>T<sub>E</sub>X3 for some functions,
- a register such as `\count123`, used in L<sup>A</sup>T<sub>E</sub>X3 for the implementation of some variables (`int`, `dim`, ...),
- a constant integer such as `\char"56` or `\mathchar"121`,
- a font selection command,
- undefined.

Macros can be `\protected` or not, `\long` or not (the opposite of what L<sup>A</sup>T<sub>E</sub>X3 calls `nopar`), and `\outer` or not (unused in L<sup>A</sup>T<sub>E</sub>X3). Their `\meaning` takes the form

*<prefix> macro:<argument>-><replacement>*

where *<prefix>* is among `\protected\long\outer`, *<argument>* describes parameters that the macro expects, such as `#1#2#3`, and *<replacement>* describes how the parameters are manipulated, such as `\int_eval:n{#2+#1*#3}`.

Now is perhaps a good time to mention some subtleties relating to tokens with category code 10 (space). Any input character with this category code (normally, space and tab characters) becomes a normal space, with character code 32 and category code 10.

When a macro takes an undelimited argument, explicit space characters (with character code 32 and category code 10) are ignored. If the following token is an explicit character token with category code 1 (begin-group) and an arbitrary character code, then T<sub>E</sub>X scans ahead to obtain an equal number of explicit character tokens with category code 1 (begin-group) and 2 (end-group), and the resulting list of tokens (with outer braces removed) becomes the argument. Otherwise, a single token is taken as the argument for the macro: we call such single tokens “N-type”, as they are suitable to be used as an argument for a function with the signature `:N`.

When a macro takes a delimited argument T<sub>E</sub>X scans ahead until finding the delimiter (outside any pairs of begin-group/end-group explicit characters), and the resulting list of tokens (with outer braces removed) becomes the argument. Note that explicit space characters at the start of the argument are *not* ignored in this case (and they prevent brace-stripping).

## 23.8 Deprecated functions

---

```
\char_lowercase:N      * \char_lowercase:N <char>
\char_uppercase:N     * Converts the <char> to the equivalent case-changed character as detailed by the func-
\char_titlecase:N      * tion name (see \str_foldcase:n and \text_titlecase:n for details of these terms).
\char_foldcase:N       *
\char_str_lowercase:N  * The case mapping is carried out with no context-dependence (cf. \text_uppercase:n,
\char_str_uppercase:N  * etc.) The str versions always generate “other” (category code 12) characters, whilst the
\char_str_titlecase:N   * standard versions generate characters with the category code of the <char> (i.e. only the
\char_str_foldcase:N    * character code changes).
```

---

New: 2020-01-09

## Chapter 24

# The l3prop package

## Property lists

expl3 implements a *property list* data type, which contain an unordered list of entries each of which consists of a *key* and an associated *value*. The *key* and *value* may both be any *balanced text*, the *key* is processed using `\t1_to_str:n`, meaning that category codes are ignored. It is possible to map functions to property lists such that the function is applied to every key–value pair within the list.

Each entry in a property list must have a unique *key*: if an entry is added to a property list which already contains the *key* then the new entry overwrites the existing one. The *keys* are compared on a string basis, using the same method as `\str_if_eq:nn`.

Property lists are intended for storing key-based information for use within code. This is in contrast to key–value lists, which are a form of *input* parsed by the l3keys module.

### 24.1 Creating and initialising property lists

---

`\prop_new:N \prop_new:N <property list>`

`\prop_new:c` Creates a new *property list* or raises an error if the name is already taken. The declaration is global. The *property list* initially contains no entries.

---

`\prop_clear:N \prop_clear:N <property list>`

`\prop_clear:c \prop_gclear:N \prop_gclear:c`

---

`\prop_clear_new:N \prop_clear_new:N <property list>`

`\prop_clear_new:c \prop_gclear_new:N \prop_gclear_new:c` Ensures that the *property list* exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

<code>\prop_set_eq:NN</code>	<code>\prop_set_eq:NN &lt;property list<sub>1</sub>&gt; &lt;property list<sub>2</sub>&gt;</code>
<code>\prop_set_eq:(cN Nc cc)</code>	Sets the content of <code>&lt;property list<sub>1</sub>&gt;</code> equal to that of <code>&lt;property list<sub>2</sub>&gt;</code> .
<code>\prop_gset_eq:NN</code>	
<code>\prop_gset_eq:(cN Nc cc)</code>	
<code>\prop_set_from_keyval:Nn</code>	<code>\prop_set_from_keyval:Nn &lt;property list&gt;</code>
<code>\prop_set_from_keyval:cn</code>	{
<code>\prop_gset_from_keyval:Nn</code>	<code>&lt;key1&gt; = &lt;value1&gt; ,</code>
<code>\prop_gset_from_keyval:cn</code>	<code>&lt;key2&gt; = &lt;value2&gt; , ...</code>
	}
New: 2017-11-28	
Updated: 2021-11-07	Sets <code>&lt;property list&gt;</code> to contain key–value pairs given in the second argument. If duplicate keys appear only the last of the values is kept.

Spaces are trimmed around every `<key>` and every `<value>`, and if the result of trimming spaces consists of a single brace group then a set of outer braces is removed. This enables both the `<key>` and the `<value>` to contain spaces, commas or equal signs. The `<key>` is then processed by `\tl_to_str:n`. This function correctly detects the `=` and `,` signs provided they have the standard category code 12 or they are active.

Notice that in contrast to most keyval lists (*e.g.* those in `I3keys`), each key here *must* be followed with an `=` sign.

<code>\prop_const_from_keyval:Nn</code>	<code>\prop_const_from_keyval:Nn &lt;property list&gt;</code>
<code>\prop_const_from_keyval:cn</code>	{
	<code>&lt;key1&gt; = &lt;value1&gt; ,</code>
	<code>&lt;key2&gt; = &lt;value2&gt; , ...</code>

Creates a new constant `<property list>` or raises an error if the name is already taken. The `<property list>` is set globally to contain key–value pairs given in the second argument, processed in the way described for `\prop_set_from_keyval:Nn`. If duplicate keys appear only the last of the values is kept. This function correctly detects the `=` and `,` signs provided they have the standard category code 12 or they are active.

Notice that in contrast to most keyval lists (*e.g.* those in `I3keys`), each key here *must* be followed with an `=` sign.

## 24.2 Adding and updating property list entries

<code>\prop_put:Nnn</code>	<code>\prop_put:Nnn</code>
<code>\prop_put:(NnV Nno Nnx NVn NVV NVx Nvx Non Noo Nxx cnn cnV cno cnx cVn cVV cVx cvx con coo cxx)</code>	<code>&lt;property list&gt;</code>
<code>\prop_gput:Nnn</code>	{ <code>&lt;key&gt;</code> }
<code>\prop_gput:(NnV Nno Nnx NVn NVV NVx Nvx Non Noo Nxx cnn cnV cno cnx cVn cVV cVx cvx con coo cxx)</code>	{ <code>&lt;value&gt;</code> }
	Updated: 2012-07-09

Adds an entry to the `<property list>` which may be accessed using the `<key>` and which has `<value>`. If the `<key>` is already present in the `<property list>`, the existing entry is overwritten by the new `<value>`. Both the `<key>` and `<value>` may contain any `<balanced text>`. The `<key>` is stored after processing with `\tl_to_str:n`, meaning that category codes are ignored.

---

```
\prop_put_if_new:Nnn  
\prop_put_if_new:cnn  
\prop_gput_if_new:Nnn  
\prop_gput_if_new:cnn
```

---

`\prop_put_if_new:Nnn` *property list* {*key*} {*value*}

If the *key* is present in the *property list* then no action is taken. Otherwise, a new entry is added as described for `\prop_put:Nnn`.

---

```
\prop_concat:NNN  
\prop_concat:ccc  
\prop_gconcat:NNN  
\prop_gconcat:ccc
```

---

`\prop_concat:NNN` *property list*<sub>1</sub> *property list*<sub>2</sub> *property list*<sub>3</sub>

Combines the key–value pairs of *property list*<sub>2</sub> and *property list*<sub>3</sub>, and saves the result in *property list*<sub>1</sub>. If a key appears in both *property list*<sub>2</sub> and *property list*<sub>3</sub> then the last value, namely the value in *property list*<sub>3</sub> is kept.

New: 2021-05-16

---

```
\prop_put_from_keyval:Nn  
\prop_put_from_keyval:cn  
\prop_gput_from_keyval:Nn  
\prop_gput_from_keyval:cn
```

---

`\prop_put_from_keyval:Nn` *property list*

```
{  
  <key1> = <value1> ,  
  <key2> = <value2> , ...  
}
```

New: 2021-05-16

Updated: 2021-11-07

Updates the *property list* by adding entries for each key–value pair given in the second argument. The addition is done through `\prop_put:Nnn`, hence if the *property list* already contains some of the keys, the corresponding values are discarded and replaced by those given in the key–value list. If duplicate keys appear in the key–value list then only the last of the values is kept.

The function is equivalent to storing the key–value pairs in a temporary property list using `\prop_set_from_keyval:Nn`, then combining *property list* with the temporary variable using `\prop_concat:NNN`. In particular, the *keys* and *values* are space-trimmed and unbraced as described in `\prop_set_from_keyval:Nn`. This function correctly detects the = and , signs provided they have the standard category code 12 or they are active.

## 24.3 Recovering values from property lists

---

```
\prop_get:NnN  
\prop_get:(NVN|NvN|NoN|cnN|cVN|cvN|coN)
```

---

Updated: 2011-08-28

`\prop_get:NnN` *property list* {*key*} {*tl var*}

Recovering the *value* stored with *key* from the *property list*, and places this in the *token list variable*. If the *key* is not found in the *property list* then the *token list variable* is set to the special marker `\q_no_value`. The *token list variable* is set within the current TeX group. See also `\prop_get:NnNTF`.

---

```
\prop_pop:NnN  
\prop_pop:(NoN|cnN|coN)
```

---

`\prop_pop:NnN` *property list* {*key*} {*tl var*}

Recovering the *value* stored with *key* from the *property list*, and places this in the *token list variable*. If the *key* is not found in the *property list* then the *token list variable* is set to the special marker `\q_no_value`. The *key* and *value* are then deleted from the property list. Both assignments are local. See also `\prop_get:NnNTF`.

---

`\prop_gpop:NnN` \* `\prop_gpop:(NoN|cnN|coN)` \* `\prop_gpop:NnN <property list> {<key>} <tl var>`  
 Updated: 2011-08-18

Recovers the *<value>* stored with *<key>* from the *<property list>*, and places this in the *<token list variable>*. If the *<key>* is not found in the *<property list>* then the *<token list variable>* is set to the special marker `\q_no_value`. The *<key>* and *<value>* are then deleted from the property list. The *<property list>* is modified globally, while the assignment of the *<token list variable>* is local. See also `\prop_gpop:NnNTF`.

---

`\prop_item:Nn` \* `\prop_item:Nn <property list> {<key>}`  
`\prop_item:cn` \* `\prop_item:cn <property list> {<key>}`  
 New: 2014-07-17

Expands to the *<value>* corresponding to the *<key>* in the *<property list>*. If the *<key>* is missing, this has an empty expansion.

**TeXhackers note:** This function is slower than the non-expandable analogue `\prop_get:NnN`. The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the *<value>* does not expand further when appearing in an x-type or e-type argument expansion.

---

`\prop_count:N` \* `\prop_count:N <property list>`  
`\prop_count:c` \* `\prop_count:c <property list>`

Leaves the number of key–value pairs in the *<property list>* in the input stream as an *<integer denotation>*.

---

`\prop_to_keyval:N` \* `\prop_to_keyval:N <property list>`

Expands to the *<property list>* in a key–value notation. Keep in mind that a *<property list>* is *unordered*, while key–value interfaces don’t necessarily are, so this can’t be used for arbitrary interfaces.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the key–value list does not expand further when appearing in an x-type or e-type argument expansion. It also needs exactly two steps of expansion.

## 24.4 Modifying property lists

---

`\prop_remove:Nn`  
`\prop_remove:(NV|cn|cV)`  
`\prop_gremove:Nn`  
`\prop_gremove:(NV|cn|cV)`  
 New: 2012-05-12

`\prop_remove:Nn <property list> {<key>}`

Removes the entry listed under *<key>* from the *<property list>*. If the *<key>* is not found in the *<property list>* no change occurs, *i.e* there is no need to test for the existence of a key before deleting it.

## 24.5 Property list conditionals

---

`\prop_if_exist_p:N` \* `\prop_if_exist_p:N <property list>`  
`\prop_if_exist_p:c` \* `\prop_if_exist:NTF <property list> {<true code>} {<false code>}`  
`\prop_if_exist:NTF` \* `\prop_if_exist:cTF` \* Tests whether the *<property list>* is currently defined. This does not check that the *<property list>* really is a property list variable.  
 New: 2012-03-03

---

```
\prop_if_empty_p:N * \prop_if_empty_p:N <property list>
\prop_if_empty_p:c * \prop_if_empty:NTF <property list> {\<true code>} {\<false code>}
\prop_if_empty:NTF * Tests if the <property list> is empty (containing no entries).
\prop_if_empty:cTF *
```

---

```
\prop_if_in_p:Nn * \prop_if_in:NnTF <property list> {\<key>} {\<true code>} {\<false code>}
\prop_if_in_p:(NV|No|cn|cV|co) *
\prop_if_in:NnTF *
\prop_if_in:(NV|No|cn|cV|co)TF *
```

---

Updated: 2011-09-15

Tests if the *<key>* is present in the *<property list>*, making the comparison using the method described by `\str_if_eq:nnTF`.

**TEXhackers note:** This function iterates through every key–value pair in the *<property list>* and is therefore slower than using the non-expandable `\prop_get:NnNTF`.

## 24.6 Recovering values from property lists with branching

The functions in this section combine tests for the presence of a key in a property list with recovery of the associated valued. This makes them useful for cases where different cases follow dependent on the presence or absence of a key in a property list. They offer increased readability and performance over separate testing and recovery phases.

---

```
\prop_get:NnNTF * \prop_get:NnNTF <property list> {\<key>} <token list variable>
\prop_get:(NVN|NvN|NoN|cnN|cVN|cvN|coN)TF {\<true code>} {\<false code>}
```

---

Updated: 2012-05-19

If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream. The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, stores the corresponding *<value>* in the *<token list variable>* without removing it from the *<property list>*, then leaves the *<true code>* in the input stream. The *<token list variable>* is assigned locally.

---

```
\prop_pop:NnNTF \prop_pop:NnNTF <property list> {\<key>} <token list variable> {\<true code>}
\prop_pop:cNNTF {\<false code>}
```

---

New: 2011-08-18 If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream.  
 Updated: 2012-05-19 The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, i.e. removes the item from the *<property list>*. Both the *<property list>* and the *<token list variable>* are assigned locally.

---

```
\prop_gpop:NnNTF \prop_gpop:NnNTF <property list> {<key>} <token list variable> {{true code}}  
\prop_gpop:cNNTF {{false code}}
```

---

New: 2011-08-18 If the *<key>* is not present in the *<property list>*, leaves the *<false code>* in the input stream.

Updated: 2012-05-19 The value of the *<token list variable>* is not defined in this case and should not be relied upon. If the *<key>* is present in the *<property list>*, pops the corresponding *<value>* in the *<token list variable>*, *i.e.* removes the item from the *<property list>*. The *<property list>* is modified globally, while the *<token list variable>* is assigned locally.

## 24.7 Mapping over property lists

All mappings are done at the current group level, *i.e.* any local assignments made by the *<function>* or *<code>* discussed below remain in effect after the loop.

---

```
\prop_map_function:NN ☆ \prop_map_function:NN <property list> <function>
```

---

Updated: 2013-01-08 Applies *<function>* to every *<entry>* stored in the *<property list>*. The *<function>* receives two arguments for each iteration: the *<key>* and associated *<value>*. The order in which *<entries>* are returned is not defined and should not be relied upon. To pass further arguments to the *<function>*, see `\prop_map_tokens:Nn`.

---

```
\prop_map_inline:Nn \prop_map_inline:Nn <property list> {{inline function}}
```

---

Updated: 2013-01-08 Applies *<inline function>* to every *<entry>* stored within the *<property list>*. The *<inline function>* should consist of code which receives the *<key>* as #1 and the *<value>* as #2. The order in which *<entries>* are returned is not defined and should not be relied upon.

---

```
\prop_map_tokens:Nn ☆ \prop_map_tokens:Nn <property list> {{code}}
```

---

Updated: 2013-01-08 Analogue of `\prop_map_function:NN` which maps several tokens instead of a single function. The *<code>* receives each key–value pair in the *<property list>* as two trailing brace groups. For instance,

```
\prop_map_tokens:Nn \l_my_prop { \str_if_eq:nnT { mykey } }
```

expands to the value corresponding to `mykey`: for each pair in `\l_my_prop` the function `\str_if_eq:nnT` receives `mykey`, the *<key>* and the *<value>* as its three arguments. For that specific task, `\prop_item:Nn` is faster.

---

`\prop_map_break: ☆ \prop_map_break:`

Updated: 2012-06-29

Used to terminate a `\prop_map_...` function before all entries in the *property list* have been processed. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \prop_map_break: }
        {
            % Do something useful
        }
}
```

Use outside of a `\prop_map_...` scenario leads to low level T<sub>E</sub>X errors.

**T<sub>E</sub>Xhackers note:** When the mapping is broken, additional tokens may be inserted before further items are taken from the input stream. This depends on the design of the mapping function.

---

`\prop_map_break:n ☆ \prop_map_break:n {<code>}`

Updated: 2012-06-29

Used to terminate a `\prop_map_...` function before all entries in the *property list* have been processed, inserting the *<code>* after the mapping has ended. This normally takes place within a conditional statement, for example

```
\prop_map_inline:Nn \l_my_prop
{
    \str_if_eq:nnTF { #1 } { bingo }
        { \prop_map_break:n { <code> } }
        {
            % Do something useful
        }
}
```

Use outside of a `\prop_map_...` scenario leads to low level T<sub>E</sub>X errors.

**T<sub>E</sub>Xhackers note:** When the mapping is broken, additional tokens may be inserted before the *<code>* is inserted into the input stream. This depends on the design of the mapping function.

## 24.8 Viewing property lists

---

`\prop_show:N \prop_show:N <property list>`

`\prop_show:c`

Displays the entries in the *property list* in the terminal.

Updated: 2021-04-29

---

`\prop_log:N`    `\prop_log:N <property list>`  
`\prop_log:c`    Writes the entries in the `<property list>` in the log file.

New: 2014-08-12  
Updated: 2021-04-29

---

## 24.9 Scratch property lists

---

`\l_tmpa_prop` Scratch property lists for local assignment. These are never used by the kernel code, and  
`\l_tmpb_prop` so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten  
New: 2012-06-23 by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_prop` Scratch property lists for global assignment. These are never used by the kernel code, and  
`\g_tmpb_prop` so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten  
New: 2012-06-23 by other non-kernel code and so should only be used for short-term storage.

## 24.10 Constants

---

`\c_empty_prop` A permanently-empty property list used for internal comparisons.

# Chapter 25

## The **\3skip** package Dimensions and skips

LATEX3 provides two general length variables: `dim` and `skip`. Lengths stored as `dim` variables have a fixed length, whereas `skip` lengths have a rubber (stretch/shrink) component. In addition, the `muskip` type is available for use in math mode: this is a special form of `skip` where the lengths involved are determined by the current math font (in `mu`). There are common features in the creation and setting of length variables, but for clarity the functions are grouped by variable type.

### 25.1 Creating and initialising `dim` variables

---

`\dim_new:N \dim_new:N <dimension>`

`\dim_new:c` Creates a new `<dimension>` or raises an error if the name is already taken. The declaration is global. The `<dimension>` is initially equal to 0 pt.

---

`\dim_const:Nn \dim_const:Nn <dimension> {<dimension expression>}`

`\dim_const:cn` Creates a new constant `<dimension>` or raises an error if the name is already taken. The value of the `<dimension>` is set globally to the `<dimension expression>`.  
New: 2012-03-05

---

`\dim_zero:N \dim_zero:N <dimension>`

`\dim_zero:c` Sets `<dimension>` to 0 pt.  
`\dim_gzero:N`  
`\dim_gzero:c`

---

`\dim_zero_new:N \dim_zero_new:N <dimension>`

`\dim_zero_new:c` Ensures that the `<dimension>` exists globally by applying `\dim_new:N` if necessary, then applies `\dim_(g)zero:N` to leave the `<dimension>` set to zero.  
`\dim_gzero_new:N`  
`\dim_gzero_new:c`

---

New: 2012-01-07

---

`\dim_if_exist_p:N` \* `\dim_if_exist_p:N` *dimension*  
`\dim_if_exist_p:c` \* `\dim_if_exist:NTF` *dimension* {*true code*} {*false code*}  
`\dim_if_exist:NTF` \* Tests whether the *dimension* is currently defined. This does not check that the `\dim_if_exist:cTF` \* *dimension* really is a dimension variable.

New: 2012-03-03

---

## 25.2 Setting dim variables

---

`\dim_add:Nn` `\dim_add:Nn` *dimension* {*dimension expression*}  
`\dim_add:cn`

Adds the result of the *dimension expression* to the current content of the *dimension*.

`\dim_gadd:Nn`  
`\dim_gadd:cn`

Updated: 2011-10-22

---

---

`\dim_set:Nn` `\dim_set:Nn` *dimension* {*dimension expression*}

`\dim_set:cn`  
`\dim_gset:Nn`  
`\dim_gset:cn`

Sets *dimension* to the value of *dimension expression*, which must evaluate to a length with units.

Updated: 2011-10-22

---

---

`\dim_set_eq:NN` `\dim_set_eq:NN` *dimension<sub>1</sub>* *dimension<sub>2</sub>*

`\dim_set_eq:(cN|Nc|cc)` Sets the content of *dimension<sub>1</sub>* equal to that of *dimension<sub>2</sub>*.

`\dim_gset_eq:NN`

`\dim_gset_eq:(cN|Nc|cc)`

---

---

`\dim_sub:Nn` `\dim_sub:Nn` *dimension* {*dimension expression*}

`\dim_sub:cn`  
`\dim_gsub:Nn`  
`\dim_gsub:cn`

Subtracts the result of the *dimension expression* from the current content of the *dimension*.

Updated: 2011-10-22

---

## 25.3 Utilities for dimension calculations

---

`\dim_abs:n` \* `\dim_abs:n` {*dimexpr*}

Updated: 2012-09-26 Converts the *dimexpr* to its absolute value, leaving the result in the input stream as a *dimension denotation*.

---

`\dim_max:nn` \* `\dim_max:nn` {*dimexpr<sub>1</sub>*} {*dimexpr<sub>2</sub>*}

`\dim_min:nn` \* `\dim_min:nn` {*dimexpr<sub>1</sub>*} {*dimexpr<sub>2</sub>*}

New: 2012-09-09 Evaluates the two *dimension expressions* and leaves either the maximum or minimum  
Updated: 2012-09-26 value in the input stream as appropriate, as a *dimension denotation*.

---

```
\dim_ratio:nn ☆ \dim_ratio:nn {⟨dimexpr1⟩} {⟨dimexpr2⟩}
```

Updated: 2011-10-22 Parses the two ⟨dimension expressions⟩ and converts the ratio of the two to a form suitable for use inside a ⟨dimension expression⟩. This ratio is then left in the input stream, allowing syntax such as

```
\dim_set:Nn \l_my_dim
{ 10 pt * \dim_ratio:nn { 5 pt } { 10 pt } }
```

The output of \dim\_ratio:nn on full expansion is a ratio expression between two integers, with all distances converted to scaled points. Thus

```
\tl_set:Nx \l_my_tl { \dim_ratio:nn { 5 pt } { 10 pt } }
\tl_show:N \l_my_tl
```

displays 327680/655360 on the terminal.

## 25.4 Dimension expression conditionals

---

```
\dim_compare_p:nNn ★ \dim_compare_p:nNn {⟨dimexpr1⟩} <relation> {⟨dimexpr2⟩}
\dim_compare:nNnTF ★ \dim_compare:nNnTF
  {⟨dimexpr1⟩} <relation> {⟨dimexpr2⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the ⟨dimension expressions⟩ as described for \dim\_eval:n. The two results are then compared using the ⟨relation⟩:

Equal	=
Greater than	>
Less than	<

This function is less flexible than \dim\_compare:nTF but around 5 times faster.

---

```
\dim_compare_p:n ∗ \dim_compare_p:n
\dim_compare:nTF ∗   {
  Updated: 2013-01-13
    ⟨dimexpr1⟩ ⟨relation1⟩
    ...
    ⟨dimexprN⟩ ⟨relationN⟩
    ⟨dimexprN+1⟩
}
\dim_compare:nTF
{
  ⟨dimexpr1⟩ ⟨relation1⟩
  ...
  ⟨dimexprN⟩ ⟨relationN⟩
  ⟨dimexprN+1⟩
}
{⟨true code⟩} {⟨false code⟩}
```

---

This function evaluates the  $\langle dimension\ expressions\rangle$  as described for `\dim_eval:n` and compares consecutive result using the corresponding  $\langle relation\rangle$ , namely it compares  $\langle dimexpr_1\rangle$  and  $\langle dimexpr_2\rangle$  using the  $\langle relation_1\rangle$ , then  $\langle dimexpr_2\rangle$  and  $\langle dimexpr_3\rangle$  using the  $\langle relation_2\rangle$ , until finally comparing  $\langle dimexpr_N\rangle$  and  $\langle dimexpr_{N+1}\rangle$  using the  $\langle relation_N\rangle$ . The test yields `true` if all comparisons are `true`. Each  $\langle dimension\ expression\rangle$  is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other  $\langle dimension\ expression\rangle$  is evaluated and no other comparison is performed. The  $\langle relations\rangle$  can be any of the following:

Equal	= or ==
Greater than or equal to	$\geq$
Greater than	$>$
Less than or equal to	$\leq$
Less than	$<$
Not equal	$\neq$

This function is more flexible than `\dim_compare:nNnTF` but around 5 times slower.

---

```
\dim_case:nn   ∗ \dim_case:nnTF {⟨test dimension expression⟩}
\dim_case:nnTF ∗   {
  ⟨dimexpr case1⟩ {⟨code case1⟩}
  ⟨dimexpr case2⟩ {⟨code case2⟩}
  ...
  ⟨dimexpr casen⟩ {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}
```

---

New: 2013-07-24

This function evaluates the *⟨test dimension expression⟩* and compares this in turn to each of the *⟨dimension expression cases⟩*. If the two are equal then the associated *⟨code⟩* is left in the input stream and other cases are discarded. If any of the cases are matched, the *⟨true code⟩* is also inserted into the input stream (after the code for the appropriate case), while if none match then the *⟨false code⟩* is inserted. The function `\dim_case:nn`, which does nothing if there is no match, is also available. For example

```
\dim_set:Nn \l_tmpa_dim { 5 pt }
\dim_case:nnF
{ 2 \l_tmpa_dim }
{
  { 5 pt }      { Small }
  { 4 pt + 6 pt } { Medium }
  { - 10 pt }    { Negative }
}
{ No idea! }
```

leaves “Medium” in the input stream.

## 25.5 Dimension expression loops

---

```
\dim_do_until:nNnn ∗ \dim_do_until:nNnn {⟨dimexpr1⟩} {⟨relation⟩} {⟨dimexpr2⟩} {⟨code⟩}
```

Places the *⟨code⟩* in the input stream for TeX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is `false` then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is `true`.

---

```
\dim_do_while:nNnn ∗ \dim_do_while:nNnn {⟨dimexpr1⟩} {⟨relation⟩} {⟨dimexpr2⟩} {⟨code⟩}
```

Places the *⟨code⟩* in the input stream for TeX to process, and then evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`. If the test is `true` then the *⟨code⟩* is inserted into the input stream again and a loop occurs until the *⟨relation⟩* is `false`.

---

```
\dim_until_do:nNnn ∗ \dim_until_do:nNnn {⟨dimexpr1⟩} {⟨relation⟩} {⟨dimexpr2⟩} {⟨code⟩}
```

Evaluates the relationship between the two *⟨dimension expressions⟩* as described for `\dim_compare:nNnTF`, and then places the *⟨code⟩* in the input stream if the *⟨relation⟩* is `false`. After the *⟨code⟩* has been processed by TeX the test is repeated, and a loop occurs until the test is `true`.

<code>\dim_while_do:nNnn</code> ☆	<code>\dim_while_do:nNnn {\langle dimexpr_1\rangle} {\langle relation\rangle} {\langle dimexpr_2\rangle} {\langle code\rangle}</code>
	Evaluates the relationship between the two <i>dimension expressions</i> as described for <code>\dim_compare:nNnTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is true. After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is false.
<code>\dim_do_until:nn</code> ☆	<code>\dim_do_until:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	Places the <i>code</i> in the input stream for TeX to process, and then evaluates the <i>dimension relation</i> as described for <code>\dim_compare:nTF</code> . If the test is false then the <i>code</i> is inserted into the input stream again and a loop occurs until the <i>relation</i> is true.
<code>\dim_do_while:nn</code> ☆	<code>\dim_do_while:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	Places the <i>code</i> in the input stream for TeX to process, and then evaluates the <i>dimension relation</i> as described for <code>\dim_compare:nTF</code> . If the test is true then the <i>code</i> is inserted into the input stream again and a loop occurs until the <i>relation</i> is false.
<code>\dim_until_do:nn</code> ☆	<code>\dim_until_do:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	Evaluates the <i>dimension relation</i> as described for <code>\dim_compare:nTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is false. After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is true.
<code>\dim_while_do:nn</code> ☆	<code>\dim_while_do:nn {\langle dimension relation\rangle} {\langle code\rangle}</code>
Updated: 2013-01-13	Evaluates the <i>dimension relation</i> as described for <code>\dim_compare:nTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is true. After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is false.

## 25.6 Dimension step functions

<code>\dim_step_function:nnnN</code> ☆	<code>\dim_step_function:nnnN {\langle initial value\rangle} {\langle step\rangle} {\langle final value\rangle} {\langle function\rangle}</code>
New: 2018-02-18	This function first evaluates the <i>initial value</i> , <i>step</i> and <i>final value</i> , all of which should be dimension expressions. The <i>function</i> is then placed in front of each <i>value</i> from the <i>initial value</i> to the <i>final value</i> in turn (using <i>step</i> between each <i>value</i> ). The <i>step</i> must be non-zero. If the <i>step</i> is positive, the loop stops when the <i>value</i> becomes larger than the <i>final value</i> . If the <i>step</i> is negative, the loop stops when the <i>value</i> becomes smaller than the <i>final value</i> . The <i>function</i> should absorb one argument.
<code>\dim_step_inline:nnnn</code>	<code>\dim_step_inline:nnnn {\langle initial value\rangle} {\langle step\rangle} {\langle final value\rangle} {\langle code\rangle}</code>
New: 2018-02-18	This function first evaluates the <i>initial value</i> , <i>step</i> and <i>final value</i> , all of which should be dimension expressions. Then for each <i>value</i> from the <i>initial value</i> to the <i>final value</i> in turn (using <i>step</i> between each <i>value</i> ), the <i>code</i> is inserted into the input stream with #1 replaced by the current <i>value</i> . Thus the <i>code</i> should define a function of one argument (#1).

---

```
\dim_step_variable:nnnNn \dim_step_variable:nnnNn
  {<initial value>} {<step>} {<final value>} {tl var} {code}
```

New: 2018-02-18

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be dimension expressions. Then for each *<value>* from the *<initial value>* to the *<final value>* in turn (using *<step>* between each *<value>*), the *<code>* is inserted into the input stream, with the *<tl var>* defined as the current *<value>*. Thus the *<code>* should make use of the *<tl var>*.

## 25.7 Using `dim` expressions and variables

---

```
\dim_eval:n * \dim_eval:n {dimension expression}
```

Updated: 2011-10-22 Evaluates the *<dimension expression>*, expanding any dimensions and token list variables within the *<expression>* to their content (without requiring `\dim_use:N`/`\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *<dimension denotation>* after two expansions. This is expressed in points (pt), and requires suitable termination if used in a TeX-style assignment as it is *not* an *<internal dimension>*.

---

```
\dim_sign:n * \dim_sign:n {dimexpr}
```

New: 2018-11-03 Evaluates the *<dimexpr>* then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

---

```
\dim_use:N * \dim_use:N {dimension}
```

`\dim_use:c` \* Recovers the content of a *<dimension>* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a *<dimension>* is required (such as in the argument of `\dim_eval:n`).

**TeXhackers note:** `\dim_use:N` is the TeX primitive `\the:` this is one of several L<sup>A</sup>T<sub>E</sub>X3 names for this primitive.

---

```
\dim_to_decimal:n * \dim_to_decimal:n {dimexpr}
```

New: 2014-07-15 Evaluates the *<dimension expression>*, and leaves the result, expressed in points (pt) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal:n { 1bp }
```

leaves 1.00374 in the input stream, *i.e.* the magnitude of one “big point” when converted to (TeX) points.

---

```
\dim_to_decimal_in_bp:n ∗ \dim_to_decimal_in_bp:n {⟨dimexpr⟩}
```

New: 2014-07-15

Evaluates the ⟨dimension expression⟩, and leaves the result, expressed in big points (bp) in the input stream, with *no units*. The result is rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_bp:n { 1pt }
```

leaves 0.99628 in the input stream, *i.e.* the magnitude of one (TeX) point when converted to big points.

---

```
\dim_to_decimal_in_sp:n ∗ \dim_to_decimal_in_sp:n {⟨dimexpr⟩}
```

New: 2015-05-18

Evaluates the ⟨dimension expression⟩, and leaves the result, expressed in scaled points (sp) in the input stream, with *no units*. The result is necessarily an integer.

---

```
\dim_to_decimal_in_unit:nn ∗ \dim_to_decimal_in_unit:nn {⟨dimexpr₁⟩} {⟨dimexpr₂⟩}
```

New: 2014-07-15

Evaluates the ⟨dimension expressions⟩, and leaves the value of ⟨dimexpr₁⟩, expressed in a unit given by ⟨dimexpr₂⟩, in the input stream. The result is a decimal number, rounded by TeX to four or five decimal places. If the decimal part of the result is zero, it is omitted, together with the decimal marker.

For example

```
\dim_to_decimal_in_unit:nn { 1bp } { 1mm }
```

leaves 0.35277 in the input stream, *i.e.* the magnitude of one big point when converted to millimetres.

Note that this function is not optimised for any particular output and as such may give different results to \dim\_to\_decimal\_in\_bp:n or \dim\_to\_decimal\_in\_sp:n. In particular, the latter is able to take a wider range of input values as it is not limited by the ability to calculate a ratio using ε-Tex primitives, which is required internally by \dim\_to\_decimal\_in\_unit:nn.

---

```
\dim_to_fp:n ∗ \dim_to_fp:n {⟨dimexpr⟩}
```

New: 2012-05-08

Expands to an internal floating point number equal to the value of the ⟨dimexpr⟩ in pt. Since dimension expressions are evaluated much faster than their floating point equivalent, \dim\_to\_fp:n can be used to speed up parts of a computation where a low precision and a smaller range are acceptable.

## 25.8 Viewing dim variables

---

```
\dim_show:N \dim_show:N ⟨dimension⟩
```

---

```
\dim_show:c
```

Displays the value of the ⟨dimension⟩ on the terminal.

---

**\dim\_show:n** `\dim_show:n {<dimension expression>}`

New: 2011-11-22 Displays the result of evaluating the *<dimension expression>* on the terminal.

Updated: 2015-08-07

---

---

**\dim\_log:N** `\dim_log:N {<dimension>}`

**\dim\_log:c** Writes the value of the *<dimension>* in the log file.

New: 2014-08-22

Updated: 2015-08-03

---

---

**\dim\_log:n** `\dim_log:n {<dimension expression>}`

New: 2014-08-22 Writes the result of evaluating the *<dimension expression>* in the log file.

Updated: 2015-08-07

---

## 25.9 Constant dimensions

---

**\c\_max\_dim** The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

---

**\c\_zero\_dim** A zero length as a dimension. This can also be used as a component of a skip.

## 25.10 Scratch dimensions

---

**\l\_tmpa\_dim** Scratch dimension for local assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

**\g\_tmpa\_dim** Scratch dimension for global assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 25.11 Creating and initialising skip variables

---

**\skip\_new:N** `\skip_new:N <skip>`

---

**\skip\_new:c** Creates a new *<skip>* or raises an error if the name is already taken. The declaration is global. The *<skip>* is initially equal to 0 pt.

<code>\skip_const:Nn</code>	<code>\skip_const:Nn &lt;skip&gt; {&lt;skip expression&gt;}</code>
<code>\skip_const:cn</code>	Creates a new constant <code>&lt;skip&gt;</code> or raises an error if the name is already taken. The value of the <code>&lt;skip&gt;</code> is set globally to the <code>&lt;skip expression&gt;</code> .
<code>\skip_zero:N</code>	<code>\skip_zero:N &lt;skip&gt;</code>
<code>\skip_zero:c</code>	Sets <code>&lt;skip&gt;</code> to 0 pt.
<code>\skip_gzero:N</code>	<code>\skip_gzero:N &lt;skip&gt;</code>
<code>\skip_gzero:c</code>	
<code>\skip_zero_new:N</code>	<code>\skip_zero_new:N &lt;skip&gt;</code>
<code>\skip_zero_new:c</code>	Ensures that the <code>&lt;skip&gt;</code> exists globally by applying <code>\skip_new:N</code> if necessary, then applies <code>\skip_(g)zero:N</code> to leave the <code>&lt;skip&gt;</code> set to zero.
<code>\skip_gzero_new:N</code>	
<code>\skip_gzero_new:c</code>	
New: 2012-03-05	
<code>\skip_if_exist_p:N</code>	<code>\skip_if_exist_p:N &lt;skip&gt;</code>
<code>\skip_if_exist_p:c</code>	<code>\skip_if_exist_p:c &lt;skip&gt; {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\skip_if_exist:NTF</code>	Tests whether the <code>&lt;skip&gt;</code> is currently defined. This does not check that the <code>&lt;skip&gt;</code> really is a skip variable.
<code>\skip_if_exist:cTF</code>	
New: 2012-01-07	
<code>\skip_if_exist_p:N *</code>	<code>\skip_if_exist_p:N &lt;skip&gt;</code>
<code>\skip_if_exist_p:c *</code>	<code>\skip_if_exist_p:c &lt;skip&gt; {&lt;true code&gt;} {&lt;false code&gt;}</code>
<code>\skip_if_exist:NTF *</code>	
<code>\skip_if_exist:cTF *</code>	
New: 2012-03-03	

## 25.12 Setting skip variables

<code>\skip_add:Nn</code>	<code>\skip_add:Nn &lt;skip&gt; {&lt;skip expression&gt;}</code>
<code>\skip_add:cn</code>	Adds the result of the <code>&lt;skip expression&gt;</code> to the current content of the <code>&lt;skip&gt;</code> .
<code>\skip_gadd:Nn</code>	
<code>\skip_gadd:cn</code>	
Updated: 2011-10-22	
<code>\skip_set:Nn</code>	<code>\skip_set:Nn &lt;skip&gt; {&lt;skip expression&gt;}</code>
<code>\skip_set:cn</code>	Sets <code>&lt;skip&gt;</code> to the value of <code>&lt;skip expression&gt;</code> , which must evaluate to a length with units and may include a rubber component (for example <code>1 cm plus 0.5 cm</code> ).
<code>\skip_gset:Nn</code>	
<code>\skip_gset:cn</code>	
Updated: 2011-10-22	
<code>\skip_set_eq:NN</code>	<code>\skip_set_eq:NN &lt;skip<sub>1</sub>&gt; &lt;skip<sub>2</sub>&gt;</code>
<code>\skip_set_eq:(cN Nc cc)</code>	Sets the content of <code>&lt;skip<sub>1</sub>&gt;</code> equal to that of <code>&lt;skip<sub>2</sub>&gt;</code> .
<code>\skip_gset_eq:NN</code>	
<code>\skip_gset_eq:(cN Nc cc)</code>	
Updated: 2011-10-22	
<code>\skip_sub:Nn</code>	<code>\skip_sub:Nn &lt;skip&gt; {&lt;skip expression&gt;}</code>
<code>\skip_sub:cn</code>	Subtracts the result of the <code>&lt;skip expression&gt;</code> from the current content of the <code>&lt;skip&gt;</code> .
<code>\skip_gsub:Nn</code>	
<code>\skip_gsub:cn</code>	
Updated: 2011-10-22	

## 25.13 Skip expression conditionals

---

```
\skip_if_eq_p:nn * \skip_if_eq_p:nn {⟨skipexpr₁⟩} {⟨skipexpr₂⟩}
\skip_if_eq:nnTF * \skip_if_eq:nnTF
  {⟨skipexpr₁⟩} {⟨skipexpr₂⟩}
  {⟨true code⟩} {⟨false code⟩}
```

This function first evaluates each of the *⟨skip expressions⟩* as described for `\skip_eval:n`. The two results are then compared for exact equality, *i.e.* both the fixed and rubber components must be the same for the test to be true.

---

```
\skip_if_finite_p:n * \skip_if_finite_p:n {⟨skipexpr⟩}
\skip_if_finite:nTF * \skip_if_finite:nTF {⟨skipexpr⟩} {⟨true code⟩} {⟨false code⟩}
```

New: 2012-03-05 Evaluates the *⟨skip expression⟩* as described for `\skip_eval:n`, and then tests if all of its components are finite.

## 25.14 Using skip expressions and variables

---

```
\skip_eval:n * \skip_eval:n {⟨skip expression⟩}
```

Updated: 2011-10-22 Evaluates the *⟨skip expression⟩*, expanding any skips and token list variables within the *⟨expression⟩* to their content (without requiring `\skip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *⟨glue denotation⟩* after two expansions. This is expressed in points (`pt`), and requires suitable termination if used in a TeX-style assignment as it is *not* an *⟨internal glue⟩*.

---

```
\skip_use:N * \skip_use:N {⟨skip⟩}
```

`\skip_use:c *` Recovers the content of a *⟨skip⟩* and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a *⟨dimension⟩* or *⟨skip⟩* is required (such as in the argument of `\skip_eval:n`).

**TeXhackers note:** `\skip_use:N` is the TeX primitive `\the:` this is one of several L<sup>A</sup>T<sub>E</sub>X3 names for this primitive.

## 25.15 Viewing skip variables

---

```
\skip_show:N * \skip_show:N {⟨skip⟩}
```

Updated: 2015-08-03 `\skip_show:c` Displays the value of the *⟨skip⟩* on the terminal.

---

```
\skip_show:n * \skip_show:n {⟨skip expression⟩}
```

New: 2011-11-22 Updated: 2015-08-07 Displays the result of evaluating the *⟨skip expression⟩* on the terminal.

---

`\skip_log:N`    `\skip_log:N <skip>`  
`\skip_log:c`    Writes the value of the `<skip>` in the log file.

New: 2014-08-22  
Updated: 2015-08-03

---

---

`\skip_log:n`    `\skip_log:n {<skip expression>}`  
New: 2014-08-22    Writes the result of evaluating the `<skip expression>` in the log file.  
Updated: 2015-08-07

---

## 25.16 Constant skips

---

`\c_max_skip`    The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.  
Updated: 2012-11-02

---

---

`\c_zero_skip`    A zero length as a skip, with no stretch nor shrink component.  
Updated: 2012-11-01

---

## 25.17 Scratch skips

---

`\l_tmpa_skip`    Scratch skip for local assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_skip`    Scratch skip for global assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 25.18 Inserting skips into the output

---

`\skip_horizontal:N`    `\skip_horizontal:N <skip>`  
`\skip_horizontal:c`    `\skip_horizontal:n {<skipexpr>}`  
`\skip_horizontal:n`    Inserts a horizontal `<skip>` into the current list. The argument can also be a `<dim>`.  
Updated: 2011-10-22

**TeXhackers note:** `\skip_horizontal:N` is the TeX primitive `\hskip` renamed.

---

```
\skip_vertical:N \skip_vertical:N <skip>
\skip_vertical:c \skip_vertical:n {{skipexpr}}
```

**\skip\_vertical:n** Inserts a vertical *<skip>* into the current list. The argument can also be a *(dim)*.

Updated: 2011-10-22

**TeXhackers note:** `\skip_vertical:N` is the TeX primitive `\vskip` renamed.

## 25.19 Creating and initialising **muskip** variables

---

```
\muskip_new:N \muskip_new:N <muskip>
\muskip_new:c
```

Creates a new *<muskip>* or raises an error if the name is already taken. The declaration is global. The *<muskip>* is initially equal to 0 mu.

---

```
\muskip_const:Nn \muskip_const:Nn <muskip> {{muskip expression}}
\muskip_const:cn
```

Creates a new constant *<muskip>* or raises an error if the name is already taken. The value of the *<muskip>* is set globally to the *<muskip expression>*.

New: 2012-03-05

---

```
\muskip_zero:N \skip_zero:N <muskip>
\muskip_zero:c
```

Sets *<muskip>* to 0 mu.

**\muskip\_gzero:N**

**\muskip\_gzero:c**

---

```
\muskip_zero_new:N \muskip_zero_new:N <muskip>
\muskip_zero_new:c
```

Ensures that the *<muskip>* exists globally by applying `\muskip_new:N` if necessary, then applies `\muskip_(g)zero:N` to leave the *<muskip>* set to zero.

New: 2012-01-07

---

```
\muskip_if_exist_p:N * \muskip_if_exist_p:N <muskip>
\muskip_if_exist_p:c * \muskip_if_exist:NTF <muskip> {{true code}} {{false code}}
\muskip_if_exist:NTF *
```

Tests whether the *<muskip>* is currently defined. This does not check that the *<muskip>* really is a muskip variable.

New: 2012-03-03

## 25.20 Setting **muskip** variables

---

```
\muskip_add:Nn \muskip_add:Nn <muskip> {{muskip expression}}
\muskip_add:cn
```

Adds the result of the *<muskip expression>* to the current content of the *<muskip>*.

**\muskip\_gadd:Nn**

**\muskip\_gadd:cn**

Updated: 2011-10-22

---

`\muskip_set:Nn` `\muskip_set:cn` `\muskip_gset:Nn` `\muskip_gset:cn` `\muskip_set:NN` `\muskip_set:(cN|Nc|cc)`

Sets  $\langle \text{muskip} \rangle$  to the value of  $\langle \text{muskip expression} \rangle$ , which must evaluate to a math length with units and may include a rubber component (for example `1 mu plus 0.5 mu`).

Updated: 2011-10-22

---

`\muskip_set_eq:NN` `\muskip_set_eq:(cN|Nc|cc)` `\muskip_gset_eq:NN` `\muskip_gset_eq:(cN|Nc|cc)` `\muskip_set_eq:NN` `\muskip_set_eq:(cN|Nc|cc)`

Sets the content of  $\langle \text{muskip}_1 \rangle$  equal to that of  $\langle \text{muskip}_2 \rangle$ .

---

`\muskip_sub:Nn` `\muskip_sub:cn` `\muskip_gsub:Nn` `\muskip_gsub:cn` `\muskip_sub:Nn` `\muskip_sub:NN` `\muskip_sub:(cN|Nc|cc)`

Subtracts the result of the  $\langle \text{muskip expression} \rangle$  from the current content of the  $\langle \text{muskip} \rangle$ .

Updated: 2011-10-22

## 25.21 Using `muskip` expressions and variables

---

`\muskip_eval:n` \* `\muskip_eval:n`  $\{\langle \text{muskip expression} \rangle\}$

Updated: 2011-10-22 Evaluates the  $\langle \text{muskip expression} \rangle$ , expanding any skips and token list variables within the  $\langle \text{expression} \rangle$  to their content (without requiring `\muskip_use:N`/`\t1_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a  $\langle \text{muglue denotation} \rangle$  after two expansions. This is expressed in `mu`, and requires suitable termination if used in a TeX-style assignment as it is *not* an  $\langle \text{internal muglue} \rangle$ .

---

`\muskip_use:N` \* `\muskip_use:N`  $\langle \text{muskip} \rangle$

`\muskip_use:c` \* Recovers the content of a  $\langle \text{skip} \rangle$  and places it directly in the input stream. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where a  $\langle \text{dimension} \rangle$  is required (such as in the argument of `\muskip_eval:n`).

**TeXhackers note:** `\muskip_use:N` is the TeX primitive `\the:` this is one of several L<sup>A</sup>T<sub>E</sub>X3 names for this primitive.

## 25.22 Viewing `muskip` variables

---

`\muskip_show:N` `\muskip_show:N`  $\langle \text{muskip} \rangle$   
`\muskip_show:c` Displays the value of the  $\langle \text{muskip} \rangle$  on the terminal.  
Updated: 2015-08-03

---

`\muskip_show:n` `\muskip_show:n {<muskip expression>}`

New: 2011-11-22 Displays the result of evaluating the *<muskip expression>* on the terminal.

Updated: 2015-08-07

---

`\muskip_log:N` `\muskip_log:N <muskip>`

`\muskip_log:c` Writes the value of the *<muskip>* in the log file.

New: 2014-08-22

Updated: 2015-08-03

---

`\muskip_log:n` `\muskip_log:n {<muskip expression>}`

Writes the result of evaluating the *<muskip expression>* in the log file.

Updated: 2015-08-07

---

## 25.23 Constant muskips

---

`\c_max_muskip` The maximum value that can be stored as a muskip, with no stretch nor shrink component.

---

`\c_zero_muskip` A zero length as a muskip, with no stretch nor shrink component.

## 25.24 Scratch muskips

---

`\l_tmpa_muskip` Scratch muskip for local assignment. These are never used by the kernel code, and so  
`\l_tmpb_muskip` are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_muskip` Scratch muskip for global assignment. These are never used by the kernel code, and so  
`\g_tmpb_muskip` are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 25.25 Primitive conditional

---

```
\if_dim:w ∗ \if_dim:w <dimen1> <relation> <dimen2>
    <true code>
\else:
    <false>
\fi:
```

Compare two dimensions. The *<relation>* is one of *<*, *=* or *>* with category code 12.

**TEXhackers note:** This is the T<sub>E</sub>X primitive `\ifdim`.

## Chapter 26

# The **I3keys** package Key–value interfaces

The key–value method is a popular system for creating large numbers of settings for controlling function or package behaviour. The system normally results in input of the form

```
\MyModuleSetup{  
    key-one = value one,  
    key-two = value two  
}
```

or

```
\MyModuleMacro[  
    key-one = value one,  
    key-two = value two  
]{argument}
```

for the user.

The high level functions here are intended as a method to create key–value controls. Keys are themselves created using a key–value interface, minimising the number of functions and arguments required. Each key is created by setting one or more *properties* of the key:

```
\keys_define:nn { mymodule }  
{  
    key-one .code:n = code including parameter #1,  
    key-two .tl_set:N = \l_mymodule_store_tl  
}
```

These values can then be set as with other key–value approaches:

```
\keys_set:nn { mymodule }  
{  
    key-one = value one,  
    key-two = value two  
}
```

As illustrated, keys are created inside a *(module)*: a set of related keys, typically those for a single module/L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package. See Section for suggestions on how to divide large numbers of keys for a single module.

At a document level, `\keys_set:nn` is used within a document function, for example

```
\DeclareDocumentCommand \MyModuleSetup { m }
  { \keys_set:nn { mymodule } { #1 } }
\DeclareDocumentCommand \MyModuleMacro { o m }
{
  \group_begin:
  \keys_set:nn { mymodule } { #1 }
  % Main code for \MyModuleMacro
  \group_end:
}
```

Key names may contain any tokens, as they are handled internally using `\tl_to_str:n`. As discussed in section 26.2, it is suggested that the character / is reserved for sub-division of keys into logical groups. Functions and variables are *not* expanded when creating key names, and so

```
\tl_set:Nn \l_mymodule_tmp_tl { key }
\keys_define:nn { mymodule }
{
  \l_mymodule_tmp_tl .code:n = code
}
```

creates a key called `\l_mymodule_tmp_tl`, and not one called `key`.

## 26.1 Creating keys

---

`\keys_define:nn` `\keys_define:nn {<module>} {<keyval list>}`

Updated: 2017-11-14

Parses the *<keyval list>* and defines the keys listed there for *<module>*. The *<module>* name is treated as a string. In practice the *<module>* should be chosen to be unique to the module in question (unless deliberately adding keys to an existing module).

The *<keyval list>* should consist of one or more key names along with an associated key *property*. The properties of a key determine how it acts. The individual properties are described in the following text; a typical use of `\keys_define:nn` might read

```
\keys_define:nn { mymodule }
{
  keyname .code:n = Some~code~using~#1,
  keyname .value_required:n = true
}
```

where the properties of the key begin from the . after the key name.

The various properties available take either no arguments at all, or require one or more arguments. This is indicated in the name of the property using an argument specification. In the following discussion, each property is illustrated attached to an arbitrary *<key>*, which when used may be supplied with a *<value>*. All key *definitions* are local.

Key properties are applied in the reading order and so the ordering is significant. Key properties which define “actions”, such as `.code:n`, `.tl_set:N`, etc., override one another. Some other properties are mutually exclusive, notably `.value_required:n` and `.value_forbidden:n`, and so they replace one another. However, properties covering non-exclusive behaviours may be given in any order. Thus for example the following definitions are equivalent.

```
\keys_define:nn { mymodule }
{
  keyname .code:n          = Some~code~using~#1,
  keyname .value_required:n = true
}
\keys_define:nn { mymodule }
{
  keyname .value_required:n = true,
  keyname .code:n          = Some~code~using~#1
}
```

Note that with the exception of the special `.undefine:` property, all key properties define the key within the current TeX scope.

---

`.bool_set:N`      `<key> .bool_set:N = <boolean variable>`  
`.bool_set:c`      Defines `<key>` to set `<boolean variable>` to `<value>` (which must be either “true” or “false”). If the variable does not exist, it will be created globally at the point that the key is set up.

Updated: 2013-07-08

---

`.bool_set_inverse:N`      `<key> .bool_set_inverse:N = <boolean variable>`  
`.bool_set_inverse:c`      Defines `<key>` to set `<boolean variable>` to the logical inverse of `<value>` (which must be either “true” or “false”). If the `<boolean variable>` does not exist, it will be created globally at the point that the key is set up.

New: 2011-08-28

Updated: 2013-07-08

---

`.choice:`      `<key> .choice:`

Sets `<key>` to act as a choice key. Each valid choice for `<key>` must then be created, as discussed in section 26.3.

---

`.choices:nn`      `<key> .choices:nn = {\<choices>} {\<code>}`  
`.choices:(Vn|on|xn)`      Sets `<key>` to act as a choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section 26.3.

---

`.clist_set:N`      `<key> .clist_set:N = <comma list variable>`  
`.clist_set:c`      Defines `<key>` to set `<comma list variable>` to `<value>`. Spaces around commas and empty items will be stripped. If the variable does not exist, it is created globally at the point that the key is set up.

New: 2011-09-11

---

```
.code:n      <key> .code:n = {<code>}
```

Updated: 2013-07-10 Stores the *<code>* for execution when *<key>* is used. The *<code>* can include one parameter (#1), which will be the *<value>* given for the *<key>*.

---

```
.cs_set:Np    <key> .cs_set:Np = <control sequence> <arg. spec.>
```

Defines *<key>* to set *<control sequence>* to have *<arg. spec.>* and replacement text *<value>*.

```
.cs_set:cp
```

```
.cs_set_protected:Np
```

```
.cs_set_protected:cp
```

```
.cs_gset:Np
```

```
.cs_gset:cp
```

```
.cs_gset_protected:Np
```

```
.cs_gset_protected:cp
```

---

New: 2020-01-11

---

```
.default:n    <key> .default:n = {<default>}
```

Creates a *<default>* value for *<key>*, which is used if no value is given. This will be used if only the key name is given, but not if a blank *<value>* is given:

```
\keys_define:nn { mymodule }
{
    key .code:n      = Hello~#1,
    key .default:n = World
}
\keys_set:nn { mymodule }
{
    key = Fred, % Prints 'Hello Fred'
    key,          % Prints 'Hello World'
    key = ,       % Prints 'Hello '
}
```

The default does not affect keys where values are required or forbidden. Thus a required value cannot be supplied by a default value, and giving a default value for a key which cannot take a value does not trigger an error.

---

```
.dim_set:N    <key> .dim_set:N = <dimension>
```

Defines *<key>* to set *<dimension>* to *<value>* (which must a dimension expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

Updated: 2020-01-17

---

```
.fp_set:N     <key> .fp_set:N = <floating point>
```

Defines *<key>* to set *<floating point>* to *<value>* (which must a floating point expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

Updated: 2020-01-17

---

```
.groups:n     <key> .groups:n = {<groups>}
```

New: 2013-07-14 Defines *<key>* as belonging to the *<groups>* declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 26.7.

---

`.inherit:n <key> .inherit:n = {<parents>}`

New: 2016-11-22 Specifies that the `<key>` path should inherit the keys listed as any of the `<parents>` (a comma list), which can be a module or a subgroup. For example, after setting

```
\keys_define:nn { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:nn { } { bar .inherit:n = foo }
```

setting

```
\keys_set:nn { bar } { test = a }
```

will be equivalent to

```
\keys_set:nn { foo } { test = a }
```

Inheritance applies at point of use, not at definition, thus keys may be added to the `<parent>` after the use of `.inherit:n` and will be active. If more than one `<parent>` is specified, the presence of the `<key>` will be tested for each in turn, with the first successful hit taking priority.

---

`.initial:n <key> .initial:n = {<value>}`

`.initial:(V|o|x)` Initialises the `<key>` with the `<value>`, equivalent to

Updated: 2013-07-09

```
\keys_set:nn {<module>} { <key> = <value> }
```

---

`.int_set:N <key> .int_set:N = <integer>`

Defines `<key>` to set `<integer>` to `<value>` (which must be an integer expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

Updated: 2020-01-17

---

`.legacy_if_set:n <key> .legacy_if_set:n = <switch>`

`.legacy_if_gset:n` Defines `<key>` to set legacy `\if <switch>` to `<value>` (which must be either “`true`” or “`false`”). The `<switch>` is the name of the switch *without the leading \if*.

The `inverse` versions will set the `<switch>` to the logical opposite of the `<value>`.

Updated: 2022-01-15

---

`.meta:n <key> .meta:n = {<keyval list>}`

Updated: 2013-07-10

Makes `<key>` a meta-key, which will set `<keyval list>` in one go. The `<keyval list>` can refer as `#1` to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`’s default value).

---

`.meta:nn <key> .meta:nn = {<path>} {<keyval list>}`

New: 2013-07-10

Makes `<key>` a meta-key, which will set `<keyval list>` in one go using the `<path>` in place of the current one. The `<keyval list>` can refer as `#1` to the value given at the time the `<key>` is used (or, if no value is given, the `<key>`’s default value).

---

**.multichoice:** `<key> .multichoice:`

**New:** 2011-08-21    Sets `<key>` to act as a multiple choice key. Each valid choice for `<key>` must then be created, as discussed in section [26.3](#).

---

**.multichoices:nn**    `<key> .multichoices:nn {<choices>} {<code>}`

**.multichoices:(Vn|on|xn)**    Sets `<key>` to act as a multiple choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_tl` will be the name of the choice made, and `\l_keys_choice_int` will be the position of the choice in the list of `<choices>` (indexed from 1). Choices are discussed in detail in section [26.3](#).

---

**.muskip\_set:N**    `<key> .muskip_set:N = <muskip>`

**.muskip\_set:c**    Defines `<key>` to set `<muskip>` to `<value>` (which must be a muskip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

**New:** 2019-05-05

**Updated:** 2020-01-17

---

**.prop\_put:N**    `<key> .prop_put:N = <property list>`

**.prop\_put:c**    **.prop\_gput:N**    Defines `<key>` to put the `<value>` onto the `<property list>` stored under the `<key>`. If the variable does not exist, it is created globally at the point that the key is set up.

**New:** 2019-01-31

---

**.skip\_set:N**    `<key> .skip_set:N = <skip>`

**.skip\_set:c**    **.skip\_gset:N**    Defines `<key>` to set `<skip>` to `<value>` (which must be a skip expression). If the variable does not exist, it is created globally at the point that the key is set up. The key will require a value at point-of-use unless a default is set.

**Updated:** 2020-01-17

---

**.str\_set:N**    `<key> .str_set:N = <string variable>`

**.str\_set:c**    **.str\_gset:N**    Defines `<key>` to set `<string variable>` to `<value>`. If the variable does not exist, it is created globally at the point that the key is set up.

**New:** 2021-10-30

---

**.str\_set\_x:N**    `<key> .str_set_x:N = <string variable>`

**.str\_set\_x:c**    **.str\_gset\_x:N**    Defines `<key>` to set `<string variable>` to `<value>`, which will be subjected to an x-type expansion (*i.e.* using `\str_set:Nx`). If the variable does not exist, it is created globally at the point that the key is set up.

**New:** 2021-10-30

---

**.tl\_set:N**    `<key> .tl_set:N = <token list variable>`

**.tl\_set:c**    **.tl\_gset:N**    Defines `<key>` to set `<token list variable>` to `<value>`. If the variable does not exist, it is created globally at the point that the key is set up.

---

**.tl\_set\_x:N** *<key>* .tl\_set\_x:N = *(token list variable)*  
.tl\_set\_x:c  
.tl\_gset\_x:N  
.tl\_gset\_x:c

Defines *<key>* to set *(token list variable)* to *<value>*, which will be subjected to an x-type expansion (*i.e.* using \tl\_set:Nx). If the variable does not exist, it is created globally at the point that the key is set up.

---

**.undefine:** *<key>* .undefine:

New: 2015-07-14 Removes the definition of the *<key>* within the current scope.

---

**.value\_forbidden:n** *<key>* .value\_forbidden:n = true|false

New: 2015-07-14 Specifies that *<key>* cannot receive a *<value>* when used. If a *<value>* is given then an error will be issued. Setting the property “**false**” cancels the restriction.

---

**.value\_required:n** *<key>* .value\_required:n = true|false

New: 2015-07-14 Specifies that *<key>* must receive a *<value>* when used. If a *<value>* is not given then an error will be issued. Setting the property “**false**” cancels the restriction.

## 26.2 Sub-dividing keys

When creating large numbers of keys, it may be desirable to divide them into several sub-groups for a given module. This can be achieved either by adding a sub-division to the module name:

```
\keys_define:nn { mymodule / subgroup }
  { key .code:n = code }
```

or to the key name:

```
\keys_define:nn { mymodule }
  { subgroup / key .code:n = code }
```

As illustrated, the best choice of token for sub-dividing keys in this way is */*. This is because of the method that is used to represent keys internally. Both of the above code fragments set the same key, which has full name `mymodule/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

## 26.3 Choice and multiple choice keys

The l3keys system supports two types of choice key, in which a series of pre-defined input values are linked to varying implementations. Choice keys are usually created so that the various values are mutually-exclusive: only one can apply at any one time. “Multiple” choice keys are also supported: these allow a selection of values to be chosen at the same time.

Mutually-exclusive choices are created by setting the `.choice:` property:

```
\keys_define:nn { mymodule }
  { key .choice: }
```

For keys which are set up as choices, the valid choices are generated by creating sub-keys of the choice key. This can be carried out in two ways.

In many cases, choices execute similar code which is dependant only on the name of the choice or the position of the choice in the list of all possibilities. Here, the keys can share the same code, and can be rapidly created using the `.choices:nn` property.

```
\keys_define:nn { mymodule }
{
    key .choices:nn =
        { choice-a, choice-b, choice-c }
    {
        You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
        which~is~in~position~\int_use:N \l_keys_choice_int \c_space_tl
        in~the~list.
    }
}
```

The index `\l_keys_choice_int` in the list of choices starts at 1.

---

`\l_keys_choice_int` Inside the code block for a choice generated using `.choices:nn`, the variables `\l_keys_choice_tl` and `\l_keys_choice_int` are available to indicate the name of the current choice, and its position in the comma list. The position is indexed from 1. Note that, as with standard key code generated using `.code:n`, the value passed to the key (i.e. the choice name) is also available as `#1`.

On the other hand, it is sometimes useful to create choices which use entirely different code from one another. This can be achieved by setting the `.choice:` property of a key, then manually defining sub-keys.

```
\keys_define:nn { mymodule }
{
    key .choice:,,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
}
```

It is possible to mix the two methods, but manually-created choices should *not* use `\l_keys_choice_tl` or `\l_keys_choice_int`. These variables do not have defined behaviour when used outside of code created using `.choices:nn` (*i.e.* anything might happen).

It is possible to allow choice keys to take values which have not previously been defined by adding code for the special `unknown` choice. The general behavior of the `unknown` key is described in Section 26.6. A typical example in the case of a choice would be to issue a custom error message:

```
\keys_define:nn { mymodule }
{
    key .choice:,,
    key / choice-a .code:n = code-a,
    key / choice-b .code:n = code-b,
    key / choice-c .code:n = code-c,
```

```

key / unknown .code:n =
  \msg_error:nnnn { mymodule } { unknown-choice }
    { key } % Name of choice key
    { choice-a , choice-b , choice-c } % Valid choices
    { \exp_not:n {\#1} } % Invalid choice given
%
%
}

```

Multiple choices are created in a very similar manner to mutually-exclusive choices, using the properties `.multichoice:` and `.multichoices:nn`. As with mutually exclusive choices, multiple choices are define as sub-keys. Thus both

```

\keys_define:nn { mymodule }
{
  key .multichoices:nn =
    { choice-a, choice-b, choice-c }
    {
      You~gave~choice~'\tl_use:N \l_keys_choice_tl',~
      which~is~in~position~
      \int_use:N \l_keys_choice_int \c_space_tl
      in~the~list.
    }
}

```

and

```

\keys_define:nn { mymodule }
{
  key .multichoice:,
  key / choice-a .code:n = code-a,
  key / choice-b .code:n = code-b,
  key / choice-c .code:n = code-c,
}

```

are valid.

When a multiple choice key is set

```

\keys_set:nn { mymodule }
{
  key = { a , b , c } % 'key' defined as a multiple choice
}

```

each choice is applied in turn, equivalent to a `clist` mapping or to applying each value individually:

```

\keys_set:nn { mymodule }
{
  key = a ,
  key = b ,
  key = c ,
}

```

Thus each separate choice will have passed to it the `\l_keys_choice_tl` and `\l_keys_choice_int` in exactly the same way as described for `.choices:nn`.

## 26.4 Key usage scope

Some keys will be used as settings which have a strictly limited scope of usage. Some will be only available once, others will only be valid until typesetting begins. To allow formats to support this in a structured way, l3keys allows this information to be specified using the `.usage:n` property.

---

`.usage:n <key> .usage:n = <scope>`

New: 2022-01-10 Defines the `<key>` to have usage within the `<scope>`, which should be one of `general`, `preamble` or `load`.

---

`\l_keys_usage_load_prop`  
`\l_keys_usage_preamble_prop`

New: 2022-01-10

l3keys itself does *not* attempt to redefine keys based on the usage scope. Rather, this information is made available with these two property lists. These hold an entry for each module (prefix); the value of each entry is a comma-separated list of the usage-restricted key(s).

## 26.5 Setting keys

---

`\keys_set:nn \keys_set:nn {<module>} {<keyval list>}`

`\keys_set:(nV|nv|no)`  
Updated: 2017-11-14 Parses the `<keyval list>`, and sets those keys which are defined for `<module>`. The behaviour on finding an unknown key can be set by defining a special `unknown` key: this is illustrated later.

---

`\l_keys_key_str` For each key processed, information of the full *path* of the key, the *name* of the key and  
`\l_keys_path_str` the *value* of the key is available within three token list variables. These may be used  
`\l_keys_value_t1` within the code of the key.

Updated: 2020-02-08 The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_t1`, and is not processed in any way by `\keys_set:nn`.

The *path* of the key is a “full” description of the key, and is unique for each key. It consists of the module and full key name, thus for example

`\keys_set:nn { mymodule } { key-a = some-value }`

has path `mymodule/key-a` while

`\keys_set:nn { mymodule } { subset / key-a = some-value }`

has path `mymodule/subset/key-a`. This information is stored in `\l_keys_path_str`.

The *name* of the key is the part of the path after the last `/`, and thus is not unique. In the preceding examples, both keys have name `key-a` despite having different paths. This information is stored in `\l_keys_key_str`.

## 26.6 Handling of unknown keys

If a key has not previously been defined (is unknown), `\keys_set:nn` looks for a special `unknown` key for the same module, and if this is not defined raises an error indicating that the key name was unknown. This mechanism can be used for example to issue custom error texts.

```
\keys_define:nn { mymodule }
{
    unknown .code:n =
        You~tried~to~set~key~'\l_keys_key_str'~to~'#1'.
}
```

---

```
\keys_set_known:nn          \keys_set_known:nn {\langle module\rangle} {\langle keyval list\rangle}
\keys_set_known:(nV|nv|no)   \keys_set_known:nnN {\langle module\rangle} {\langle keyval list\rangle} {\langle tl\rangle}
\keys_set_known:nnN          \keys_set_known:nnnN {\langle module\rangle} {\langle keyval list\rangle} {\langle root\rangle} {\langle tl\rangle}
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nnnN
\keys_set_known:(nVnN|nvN|noN)
```

---

New: 2011-08-23

Updated: 2019-01-29

These functions set keys which are known for the `\langle module\rangle`, and simply ignore other keys. The `\keys_set_known:nn` function parses the `\langle keyval list\rangle`, and sets those keys which are defined for `\langle module\rangle`. Any keys which are unknown are not processed further by the parser. In addition, `\keys_set_known:nnN` stores the key–value pairs in the `\langle tl\rangle` in comma-separated form (*i.e.* an edited version of the `\langle keyval list\rangle`). When a `\langle root\rangle` is given (`\keys_set_known:nnnN`), the key–value entries are returned relative to this point in the key tree. When it is absent, only the key name and value are provided. The correct list is returned by nested calls.

## 26.7 Selective key setting

In some cases it may be useful to be able to select only some keys for setting, even though these keys have the same path. For example, with a set of keys defined using

```
\keys_define:nn { mymodule }
{
    key-one .code:n = { \my_func:n {#1} } ,
    key-two .tl_set:N = \l_my_a_tl ,
    key-three .tl_set:N = \l_my_b_tl ,
    key-four .fp_set:N = \l_my_a_fp ,
}
```

the use of `\keys_set:nn` attempts to set all four keys. However, in some contexts it may only be sensible to set some keys, or to control the order of setting. To do this, keys may be assigned to *groups*: arbitrary sets which are independent of the key tree. Thus modifying the example to read

```
\keys_define:nn { mymodule }
{
    key-one .code:n = { \my_func:n {#1} } ,
    key-one .groups:n = { first } ,
    key-two .tl_set:N = \l_my_a_tl ,
    key-two .groups:n = { first } ,
    key-three .tl_set:N = \l_my_b_tl ,
    key-three .groups:n = { second } ,
    key-four .fp_set:N = \l_my_a_fp ,
}
```

assigns `key-one` and `key-two` to group `first`, `key-three` to group `second`, while `key-four` is not assigned to a group.

Selective key setting may be achieved either by selecting one or more groups to be made “active”, or by marking one or more groups to be ignored in key setting.

---

<code>\keys_set_filter:nnn</code> <code>\keys_set_filter:(nnV nnv nno)</code> <code>\keys_set_filter:nnnN</code> <code>\keys_set_filter:(nnVN nnvN nnoN)</code> <code>\keys_set_filter:nnnnN</code> <code>\keys_set_filter:(nnVnN nnvnN nnonN)</code>	<code>\keys_set_filter:nnn {&lt;module&gt;} {&lt;groups&gt;} {&lt;keyval list&gt;}</code> <code>\keys_set_filter:nnnN {&lt;module&gt;} {&lt;groups&gt;} {&lt;keyval list&gt;} &lt;tl&gt;</code> <code>\keys_set_filter:nnnnN {&lt;module&gt;} {&lt;groups&gt;} {&lt;keyval list&gt;} &lt;root&gt;</code> <code>&lt;tl&gt;</code>
--	---

---

New: 2013-07-14

Updated: 2019-01-29

Activates key filtering in an “opt-out” sense: keys assigned to any of the `<groups>` specified are ignored. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus always set. The key-value pairs for each key which is filtered out are stored in the `<tl>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_filter:nnn` version skips this stage.

Use of `\keys_set_filter:nnnnN` can be nested, with the correct residual `<keyval list>` returned at each stage. In the version which takes a `<root>` argument, the key list is returned relative to that point in the key tree. In the cases without a `<root>` argument, only the key names and values are returned.

---

<code>\keys_set_groups:nnn</code> <code>\keys_set_groups:(nnV nnv nno)</code>	<code>\keys_set_groups:nnn {&lt;module&gt;} {&lt;groups&gt;} {&lt;keyval list&gt;}</code>
--	---

---

New: 2013-07-14

Updated: 2017-05-27

Activates key filtering in an “opt-in” sense: only keys assigned to one or more of the `<groups>` specified are set. The `<groups>` are given as a comma-separated list. Unknown keys are not assigned to any group and are thus never set.

## 26.8 Digesting keys

---

<code>\keys_compile:nnN</code> <code>\keys_compile:nnN {&lt;module&gt;} {&lt;keyval list&gt;} &lt;tl&gt;</code>
--

---

New: 2022-03-09

Parses the `<keyval list>` as for `\keys_set:nn`, placing the resulting code for those which set variables or functions into the `<tl>`. Thus this function “precompiles” the keyval list into a set of results which can be applied rapidly.

## 26.9 Utility functions for keys

---

\keys\_if\_exist\_p:nn \* \keys\_if\_exist\_p:nnn {\langle module \rangle} {\langle key \rangle}  
\keys\_if\_exist\_p:ne \* \keys\_if\_exist:nnTF {\langle module \rangle} {\langle key \rangle} {\langle true code \rangle} {\langle false code \rangle}  
\keys\_if\_exist:nnTF \* Tests if the *key* exists for *module*, i.e. if any code has been defined for *key*.  
\keys\_if\_exist:neTF \*

---

Updated: 2022-01-10

---

\keys\_if\_choice\_exist\_p:nnn \* \keys\_if\_choice\_exist\_p:nnn {\langle module \rangle} {\langle key \rangle} {\langle choice \rangle}  
\keys\_if\_choice\_exist:nnnTF \* \keys\_if\_choice\_exist:nnnTF {\langle module \rangle} {\langle key \rangle} {\langle choice \rangle} {\langle true code \rangle}  
{\langle false code \rangle}

---

New: 2011-08-21

Updated: 2017-11-14

Tests if the *choice* is defined for the *key* within the *module*, i.e. if any code has been defined for *key*/*choice*. The test is **false** if the *key* itself is not defined.

---

\keys\_show:nn \keys\_show:nn {\langle module \rangle} {\langle key \rangle}

---

Updated: 2015-08-09 Displays in the terminal the information associated to the *key* for a *module*, including the function which is used to actually implement it.

---

\keys\_log:nn \keys\_log:nn {\langle module \rangle} {\langle key \rangle}

---

New: 2014-08-22 Writes in the log file the information associated to the *key* for a *module*. See also Updated: 2015-08-09 \keys\_show:nn which displays the result in the terminal.

## 26.10 Low-level interface for parsing key–val lists

To re-cap from earlier, a key–value list is input of the form

```
KeyOne = ValueOne ,  
KeyTwo = ValueTwo ,  
KeyThree
```

where each key–value pair is separated by a comma from the rest of the list, and each key–value pair does not necessarily contain an equals sign or a value! Processing this type of input correctly requires a number of careful steps, to correctly account for braces, spaces and the category codes of separators.

While the functions described earlier are used as a high-level interface for processing such input, in special circumstances you may wish to use a lower-level approach. The low-level parsing system converts a *key–value list* into *keys* and associated *values*. After the parsing phase is completed, the resulting keys and values (or keys alone) are available for further processing. This processing is not carried out by the low-level parser itself, and so the parser requires the names of two functions along with the key–value list. One function is needed to process key–value pairs (it receives two arguments), and a second function is required for keys given without any value (it is called with a single argument).

The parser does not double # tokens or expand any input. Active tokens = and , appearing at the outer level of braces are converted to category “other” (12) so that the

parser does not “miss” any due to category code changes. Spaces are removed from the ends of the keys and values. Keys and values which are given in braces have exactly one set removed (after space trimming), thus

```
key = {value here},
```

and

```
key = value here,
```

are treated identically.

---

```
\keyval_parse:nnn ☆ \keyval_parse:nnn {\<code1>} {\<code2>} {\<key-value list>}
```

New: 2020-12-19      Parses the *<key-value list>* into a series of *<keys>* and associated *<values>*, or keys alone  
Updated: 2021-05-10      if no *<value>* was given). *<code<sub>1</sub>>* receives each *<key>* (with no *<value>*) as a trailing brace group, whereas *<code<sub>2</sub>>* is appended by two brace groups, the *<key>* and *<value>*. The order of the *<keys>* in the *<key-value list>* is preserved. Thus

---

```
\keyval_parse:nnn
{ \use_none:nn { code 1 } }
{ \use_none:nnn { code 2 } }
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\use_none:nnn { code 2 } { key1 } { value1 }
\use_none:nnn { code 2 } { key2 } { value2 }
\use_none:nnn { code 2 } { key3 } { }
\use_none:nn { code 1 } { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the *<key>* and *<value>*, then one *outer* set of braces is removed from the *<key>* and *<value>* as part of the processing. If you need exactly the output shown above, you’ll need to either *x*-type or *e*-type expand the function.

**TEXhackers note:** The result of each list element is returned within *\exp\_not:n*, which means that the converted input stream does not expand further when appearing in an *x*-type or *e*-type argument expansion.

---

```
\keyval_parse:NNn ☆ \keyval_parse:NNn <function1> <function2> {<key-value list>}
```

---

Updated: 2021-05-10

Parses the  $\langle key-value list \rangle$  into a series of  $\langle keys \rangle$  and associated  $\langle values \rangle$ , or keys alone (if no  $\langle value \rangle$  was given).  $\langle function_1 \rangle$  should take one argument, while  $\langle function_2 \rangle$  should absorb two arguments. After  $\backslash keyval\_parse:NNn$  has parsed the  $\langle key-value list \rangle$ ,  $\langle function_1 \rangle$  is used to process keys given with no value and  $\langle function_2 \rangle$  is used to process keys given with a value. The order of the  $\langle keys \rangle$  in the  $\langle key-value list \rangle$  is preserved. Thus

```
\keyval_parse:NNn \function:n \function:nn  
{ key1 = value1 , key2 = value2, key3 = , key4 }
```

is converted into an input stream

```
\function:nn { key1 } { value1 }  
\function:nn { key2 } { value2 }  
\function:nn { key3 } { }  
\function:n { key4 }
```

Note that there is a difference between an empty value (an equals sign followed by nothing) and a missing value (no equals sign at all). Spaces are trimmed from the ends of the  $\langle key \rangle$  and  $\langle value \rangle$ , then one outer set of braces is removed from the  $\langle key \rangle$  and  $\langle value \rangle$  as part of the processing.

This shares the implementation of  $\backslash keyval\_parse:nnn$ , the difference is only semantically.

**TeXhackers note:** The result is returned within  $\exp_{\text{not}}:n$ , which means that the converted input stream does not expand further when appearing in an **x**-type or **e**-type argument expansion.

# Chapter 27

## The `I3intarray` package: fast global integer arrays

### 27.1 `I3intarray` documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast `I3seq`, where access time is linear). These arrays have several important features

- The size of the array is fixed and must be given at point of initialisation
- The absolute value of each entry has maximum  $2^{30} - 1$  (*i.e.* one power lower than the usual `\c_max_int` ceiling of  $2^{31} - 1$ )

The use of `intarray` data is therefore recommended for cases where the need for fast access is of paramount importance.

---

`\intarray_new:Nn \intarray_new:Nn <intarray var> {<size>}`

`\intarray_new:cn` Evaluates the integer expression `<size>` and allocates an `<integer array variable>` with that number of (zero) entries. The variable name should start with `\g_` because assignments are always global.  
New: 2018-03-29

---

`\intarray_count:N * \intarray_count:N <intarray var>`

`\intarray_count:c *` Expands to the number of entries in the `<integer array variable>`. Contrarily to `\seq_count:N` this is performed in constant time.  
New: 2018-03-29

---

`\intarray_gset:Nnn \intarray_gset:Nnn <intarray var> {<position>} {<value>}`

`\intarray_gset:cnn` Stores the result of evaluating the integer expression `<value>` into the `<integer array variable>` at the (integer expression) `<position>`. If the `<position>` is not between 1 and the `\intarray_count:N`, or the `<value>`'s absolute value is bigger than  $2^{30} - 1$ , an error occurs. Assignments are always global.  
New: 2018-03-29

---

```
\intarray_const_from_clist:Nn \intarray_const_from_clist:Nn <intarray var> <intexpr clist>
\intarray_const_from_clist:cn
```

---

New: 2018-05-04

Creates a new constant *<integer array variable>* or raises an error if the name is already taken. The *<integer array variable>* is set (globally) to contain as its items the results of evaluating each *<integer expression>* in the *<comma list>*.

---

```
\intarray_gzero:N \intarray_gzero:N <intarray var>
```

**\intarray\_gzero:c** Sets all entries of the *<integer array variable>* to zero. Assignments are always global.

---

New: 2018-05-04

---

```
\intarray_item:Nn * \intarray_item:Nn <intarray var> {<position>}
```

**\intarray\_item:cn \*** Expands to the integer entry stored at the (integer expression) *<position>* in the *<integer array variable>*. If the *<position>* is not between 1 and the *\intarray\_count:N*, an error occurs.

---

```
\intarray_rand_item:N * \intarray_rand_item:N <intarray var>
```

**\intarray\_rand\_item:c \*** Selects a pseudo-random item of the *<integer array>*. If the *<integer array>* is empty, produce an error.

---

```
\intarray_show:N \intarray_show:N <intarray var>
```

```
\intarray_show:c \intarray_log:N <intarray var>
```

**\intarray\_log:N** Displays the items in the *<integer array variable>* in the terminal or writes them in the log file.

---

New: 2018-05-04

### 27.1.1 Implementation notes

It is a wrapper around the `\fontdimen` primitive, used to store arrays of integers (with a restricted range: absolute value at most  $2^{30} - 1$ ). In contrast to `\seq` sequences the access to individual entries is done in constant time rather than linear time, but only integers can be stored. More precisely, the primitive `\fontdimen` stores dimensions but the `\intarray` package transparently converts these from/to integers. Assignments are always global.

While LuaTeX's memory is extensible, other engines can "only" deal with a bit less than  $4 \times 10^6$  entries in all `\fontdimen` arrays combined (with default TeX Live settings).

## Chapter 28

# The **l3fp** package: Floating points

A decimal floating point number is one which is stored as a significand and a separate exponent. The module implements expandably a wide set of arithmetic, trigonometric, and other operations on decimal floating point numbers, to be used within floating point expressions. Floating point expressions support the following operations with their usual precedence.

- Basic arithmetic: addition  $x + y$ , subtraction  $x - y$ , multiplication  $x * y$ , division  $x / y$ , square root  $\sqrt{x}$ , and parentheses.
- Comparison operators:  $x < y$ ,  $x \leq y$ ,  $x >? y$ ,  $x != y$  etc.
- Boolean logic: sign  $\text{sign } x$ , negation  $\text{!} x$ , conjunction  $x \&& y$ , disjunction  $x || y$ , ternary operator  $x ? y : z$ .
- Exponentials:  $\exp x$ ,  $\ln x$ ,  $x^y$ ,  $\log_b x$ .
- Integer factorial:  $\text{fact } x$ .
- Trigonometry:  $\sin x$ ,  $\cos x$ ,  $\tan x$ ,  $\cot x$ ,  $\sec x$ ,  $\csc x$  expecting their arguments in radians, and  $\text{sind } x$ ,  $\text{cosd } x$ ,  $\text{tand } x$ ,  $\text{cotd } x$ ,  $\text{secd } x$ ,  $\text{csed } x$  expecting their arguments in degrees.
- Inverse trigonometric functions:  $\text{asin } x$ ,  $\text{acos } x$ ,  $\text{atan } x$ ,  $\text{acot } x$ ,  $\text{asec } x$ ,  $\text{acsc } x$  giving a result in radians, and  $\text{asind } x$ ,  $\text{acosd } x$ ,  $\text{atand } x$ ,  $\text{cotd } x$ ,  $\text{secd } x$ ,  $\text{csed } x$  giving a result in degrees.

(not yet) Hyperbolic functions and their inverse functions:  $\sinh x$ ,  $\cosh x$ ,  $\tanh x$ ,  $\coth x$ ,  $\text{sech } x$ ,  $\text{csch } x$ , and  $\text{asinh } x$ ,  $\text{acosh } x$ ,  $\text{atanh } x$ ,  $\text{acoth } x$ ,  $\text{asech } x$ ,  $\text{acsch } x$ .

- Extrema:  $\max(x_1, x_2, \dots)$ ,  $\min(x_1, x_2, \dots)$ ,  $\text{abs}(x)$ .
- Rounding functions, controlled by two optional values,  $n$  (number of places, 0 by default) and  $t$  (behavior on a tie, `nan` by default):
  - $\text{trunc}(x, n)$  rounds towards zero,
  - $\text{floor}(x, n)$  rounds towards  $-\infty$ ,

- $\text{ceil}(x, n)$  rounds towards  $+\infty$ ,
- $\text{round}(x, n, t)$  rounds to the closest value, with ties rounded to an even value by default, towards zero if  $t = 0$ , towards  $+\infty$  if  $t > 0$  and towards  $-\infty$  if  $t < 0$ .

And (*not yet*) modulo, and “quantize”.

- Random numbers: `rand()`, `randint(m, n)`.
- Constants: `pi`, `deg` (one degree in radians).
- Dimensions, automatically expressed in points, *e.g.*, `pc` is 12.
- Automatic conversion (no need for `\(\langle type \rangle\_use:N\)`) of integer, dimension, and skip variables to floating point numbers, expressing dimensions in points and ignoring the stretch and shrink components of skips.
- Tuples:  $(x_1, \dots, x_n)$  that can be stored in variables, added together, multiplied or divided by a floating point number, and nested.

Floating point numbers can be given either explicitly (in a form such as `1.234e-34`, or `-.0001`), or as a stored floating point variable, which is automatically replaced by its current value. A “floating point” is a floating point number or a tuple thereof. See section 28.9.1 for a description of what a floating point is, section 28.9.2 for details about how an expression is parsed, and section 28.9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 28.7.

An example of use could be the following.

```
\LaTeX{} can now compute: $ \frac{\sin(3.5)}{2} + 2 \cdot 10^{-3}
= \ExplSyntaxOn \fp_to_decimal:n { \sin(3.5)/2 + 2e-3 } $.
```

The operation `round` can be used to limit the result’s precision. Adding `+0` avoids the possibly undesirable output `-0`, replacing it by `+0`. However, the `l3fp` module is mostly meant as an underlying tool for higher-level commands. For example, one could provide a function to typeset nicely the result of floating point computations.

```
\documentclass{article}
\usepackage{xparse, siunitx}
\ExplSyntaxOn
\NewDocumentCommand{\calcnum}{m}
  { \num{ \fp_to_scientific:n {#1} } }
\ExplSyntaxOff
\begin{document}
\calcnum{2 pi * sin(2.3^5)}
\end{document}
```

See the documentation of `siunitx` for various options of `\num`.

## 28.1 Creating and initialising floating point variables

---

\fp\_new:N      \fp\_new:N *fp var*  
\fp\_new:c      Creates a new *fp var* or raises an error if the name is already taken. The declaration is  
Updated: 2012-05-08 global. The *fp var* is initially +0.

---

\fp\_const:Nn    \fp\_const:Nn *fp var* {*floating point expression*}  
\fp\_const:cn    Creates a new constant *fp var* or raises an error if the name is already taken. The  
Updated: 2012-05-08 *fp var* is set globally equal to the result of evaluating the *floating point expression*.

---

\fp\_zero:N      \fp\_zero:N *fp var*  
\fp\_zero:c      Sets the *fp var* to +0.  
\fp\_gzero:N     \fp\_gzero:N  
\fp\_gzero:c  
Updated: 2012-05-08

---

\fp\_zero\_new:N   \fp\_zero\_new:N *fp var*  
\fp\_zero\_new:c   Ensures that the *fp var* exists globally by applying \fp\_new:N if necessary, then applies  
\fp\_gzero\_new:N  
\fp\_gzero\_new:c

Updated: 2012-05-08

## 28.2 Setting floating point variables

---

\fp\_set:Nn      \fp\_set:Nn *fp var* {*floating point expression*}  
\fp\_set:cn      Sets *fp var* equal to the result of computing the *floating point expression*.  
\fp\_gset:Nn  
\fp\_gset:cn  
Updated: 2012-05-08

---

\fp\_set\_eq:NN     \fp\_set\_eq:NN *fp var*<sub>1</sub> *fp var*<sub>2</sub>  
\fp\_set\_eq:(cN|Nc|cc)   Sets the floating point variable *fp var*<sub>1</sub> equal to the current value of *fp var*<sub>2</sub>.  
\fp\_gset\_eq:(cN|Nc|cc)  
Updated: 2012-05-08

---

\fp\_add:Nn      \fp\_add:Nn *fp var* {*floating point expression*}  
\fp\_add:cn  
\fp\_gadd:Nn     Adds the result of computing the *floating point expression* to the *fp var*. This also  
\fp\_gadd:cn applies if *fp var* and *floating point expression* evaluate to tuples of the same size.  
Updated: 2012-05-08

---

\fp_sub:Nn	\fp_sub:Nn <i>fp var</i> {\i{floating point expression}}
\fp_sub:cn	Subtracts the result of computing the <i>floating point expression</i> from the <i>fp var</i> . This also applies if <i>fp var</i> and <i>floating point expression</i> evaluate to tuples of the same size.
\fp_gsub:Nn	
\fp_gsub:cn	

---

Updated: 2012-05-08

## 28.3 Using floating points

---

\fp_eval:n	* \fp_eval:n {\i{floating point expression}}
New: 2012-05-08	Evaluates the <i>floating point expression</i> and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_eval:n</code> and they are combined as $(\langle fp_1 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items. This function is identical to <code>\fp_to_decimal:n</code> .
Updated: 2012-07-08	

---

\fp_sign:n	* \fp_sign:n {\i{fpexpr}}
New: 2018-11-03	Evaluates the <i>fpexpr</i> and leaves its sign in the input stream using <code>\fp_eval:n {sign(\langle result \rangle)}</code> : +1 for positive numbers and for $+\infty$ , -1 for negative numbers and for $-\infty$ , $\pm 0$ for $\pm 0$ . If the operand is a tuple or is <code>nan</code> , then “invalid operation” occurs and the result is 0.

---

\fp_to_decimal:N	* \fp_to_decimal:N <i>fp var</i>
\fp_to_decimal:c	* \fp_to_decimal:n {\i{floating point expression}}
\fp_to_decimal:n	* Evaluates the <i>floating point expression</i> and expresses the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and <code>nan</code> trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as $(\langle fp_1 \rangle, \dots, \langle fp_n \rangle)$ if $n > 1$ and $(\langle fp_1 \rangle,)$ or $()$ for fewer items.
New: 2012-05-08	
Updated: 2012-07-08	

---

\fp_to_dim:N	* \fp_to_dim:N <i>fp var</i>
\fp_to_dim:c	* \fp_to_dim:n {\i{floating point expression}}
\fp_to_dim:n	* Evaluates the <i>floating point expression</i> and expresses the result as a dimension (in pt) suitable for use in dimension expressions. The output is identical to <code>\fp_to_decimal:n</code> , with an additional trailing <code>pt</code> (both letter tokens). In particular, the result may be outside the range $[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]$ of valid TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values $\pm\infty$ and <code>nan</code> , trigger an “invalid operation” exception.

---

\fp_to_int:N	* \fp_to_int:N <i>fp var</i>
\fp_to_int:c	* \fp_to_int:n {\i{floating point expression}}
\fp_to_int:n	* Evaluates the <i>floating point expression</i> , and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range $[-2^{31} + 1, 2^{31} - 1]$ of valid TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values $\pm\infty$ and <code>nan</code> , trigger an “invalid operation” exception.

---

---

\fp\_to\_scientific:N \* \fp\_to\_scientific:N *fp var*  
\fp\_to\_scientific:c \* \fp\_to\_scientific:n {{floating point expression}}  
\fp\_to\_scientific:n \* Evaluates the *floating point expression* and expresses the result in scientific notation:  
New: 2012-05-08  
Updated: 2016-03-22      *(optional -)(digit).(15 digits)e(optional sign)(exponent)*

---

The leading *digit* is non-zero except in the case of  $\pm 0$ . The values  $\pm\infty$  and `nan` trigger an “invalid operation” exception. Normal category codes apply: thus the `e` is category code 11 (a letter). For a tuple, each item is converted using `\fp_to_scientific:n` and they are combined as  $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$  if  $n > 1$  and  $(\langle fp_1 \rangle,)$  or  $()$  for fewer items.

---

\fp\_to\_tl:N \* \fp\_to\_tl:N *fp var*  
\fp\_to\_tl:c \* \fp\_to\_tl:n {{floating point expression}}  
\fp\_to\_tl:n \* Evaluates the *floating point expression* and expresses the result in (almost) the shortest possible form. Numbers in the ranges  $(0, 10^{-3})$  and  $[10^{16}, \infty)$  are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from `\fp_to_scientific:n`). Numbers in the range  $[10^{-3}, 10^{16})$  are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see `\fp_to_decimal:n`). Negative numbers start with `-`. The special values  $\pm 0$ ,  $\pm\infty$  and `nan` are rendered as `0`, `-0`, `inf`, `-inf`, and `nan` respectively. Normal category codes apply and thus `inf` or `nan`, if produced, are made up of letters. For a tuple, each item is converted using `\fp_to_tl:n` and they are combined as  $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$  if  $n > 1$  and  $(\langle fp_1 \rangle,)$  or  $()$  for fewer items.

---



---

\fp\_use:N \* \fp\_use:N *fp var*  
\fp\_use:c \* Inserts the value of the *fp var* into the input stream as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed. Integers are expressed without a decimal separator. The values  $\pm\infty$  and `nan` trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_to_decimal:n` and they are combined as  $(\langle fp_1 \rangle, \langle fp_2 \rangle, \dots, \langle fp_n \rangle)$  if  $n > 1$  and  $(\langle fp_1 \rangle,)$  or  $()$  for fewer items. This function is identical to `\fp_to_decimal:N`.

---

## 28.4 Floating point conditionals

---

\fp\_if\_exist\_p:N \* \fp\_if\_exist\_p:N *fp var*  
\fp\_if\_exist\_p:c \* \fp\_if\_exist:NTF *fp var* {{true code}} {{false code}}  
\fp\_if\_exist:NTF \* Tests whether the *fp var* is currently defined. This does not check that the *fp var* really is a floating point variable.  
Updated: 2012-05-08

---

---

```
\fp_compare_p:nNn * \fp_compare_p:nNn {\⟨fpexpr₁⟩} ⟨relation⟩ {\⟨fpexpr₂⟩}
\fp_compare:nNnTF * \fp_compare:nNnTF {\⟨fpexpr₁⟩} ⟨relation⟩ {\⟨fpexpr₂⟩} {\⟨true code⟩} {\⟨false code⟩}
```

---

Updated: 2012-05-08

Compares the  $\langle fpexpr_1 \rangle$  and the  $\langle fpexpr_2 \rangle$ , and returns `true` if the  $\langle relation \rangle$  is obeyed. Two floating points  $x$  and  $y$  may obey four mutually exclusive relations:  $x < y$ ,  $x = y$ ,  $x > y$ , or  $x?y$  (“not ordered”). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Note that a `nan` is distinct from any value, even another `nan`, hence  $x = x$  is not true for a `nan`. To test if a value is `nan`, compare it to an arbitrary number with the “not ordered” relation.

```
\fp_compare:nNnTF {⟨value⟩} ? { 0 }
{ } % ⟨value⟩ is nan
{ } % ⟨value⟩ is not nan
```

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no `nan`). At present any other comparison with tuples yields `?` (not ordered). This is experimental.

This function is less flexible than `\fp_compare:nTF` but slightly faster. It is provided for consistency with `\int_compare:nNnTF` and `\dim_compare:nNnTF`.

---

```
\fp_compare_p:n ∗ \fp_compare_p:n
\fp_compare:nTF ∗ {
  Updated: 2013-12-14
    ⟨fpexpr1⟩ ⟨relation1⟩
    ...
    ⟨fpexprN⟩ ⟨relationN⟩
    ⟨fpexprN+1⟩
}
\fp_compare:nTF
{
  ⟨fpexpr1⟩ ⟨relation1⟩
  ...
  ⟨fpexprN⟩ ⟨relationN⟩
  ⟨fpexprN+1⟩
}
{⟨true code⟩} {⟨false code⟩}
```

---

Evaluates the *<floating point expressions>* as described for `\fp_eval:n` and compares consecutive result using the corresponding *<relation>*, namely it compares *<fpexpr<sub>1</sub>>* and *<fpexpr<sub>2</sub>>* using the *<relation<sub>1</sub>>*, then *<fpexpr<sub>2</sub>>* and *<fpexpr<sub>3</sub>>* using the *<relation<sub>2</sub>>*, until finally comparing *<fpexpr<sub>N</sub>>* and *<fpexpr<sub>N+1</sub>>* using the *<relation<sub>N</sub>>*. The test yields `true` if all comparisons are `true`. Each *<floating point expression>* is evaluated only once. Contrarily to `\int_compare:nTF`, all *<floating point expressions>* are computed, even if one comparison is `false`. Two floating points *x* and *y* may obey four mutually exclusive relations: *x < y*, *x = y*, *x > y*, or *x?y* (“not ordered”). The last case occurs exactly if one or both operands is `nan` or is a tuple, unless they are equal tuples. Each *<relation>* can be any (non-empty) combination of `<`, `=`, `>`, and `?`, plus an optional leading `!` (which negates the *<relation>*), with the restriction that the *<relation>* may not start with `?`, as this symbol has a different meaning (in combination with `:`) within floating point expressions. The comparison *x < relation y* is then `true` if the *<relation>* does not start with `!` and the actual relation (`<`, `=`, `>`, or `?`) between *x* and *y* appears within the *<relation>*, or on the contrary if the *<relation>* starts with `!` and the relation between *x* and *y* does not appear within the *<relation>*. Common choices of *<relation>* include `>=` (greater or equal), `!=` (not equal), `!?` or `<=>` (comparable).

This function is more flexible than `\fp_compare:nNnTF` and only slightly slower.

## 28.5 Floating point expression loops

---

```
\fp_do_until:nNnn ∗ \fp_do_until:nNnn {⟨fpexpr1⟩} ⟨relation⟩ {⟨fpexpr2⟩} {⟨code⟩}
```

---

New: 2012-08-16 Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<floating point expressions>* as described for `\fp_compare:nNnTF`. If the test is `false` then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is `true`.

---

```
\fp_do_while:nNnn ∗ \fp_do_while:nNnn {⟨fpexpr1⟩} ⟨relation⟩ {⟨fpexpr2⟩} {⟨code⟩}
```

---

New: 2012-08-16 Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<floating point expressions>* as described for `\fp_compare:nNnTF`. If the test is `true` then the *<code>* is inserted into the input stream again and a loop occurs until the *<relation>* is `false`.

<code>\fp_until_do:nNnn</code>	<code>\fp_until_do:nNnn {&lt;fpexpr<sub>1</sub>&gt;} &lt;relation&gt; {&lt;fpexpr<sub>2</sub>&gt;} {&lt;code&gt;}</code>
<small>New: 2012-08-16</small>	Evaluates the relationship between the two <i>&lt;floating point expressions&gt;</i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <b>false</b> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <b>true</b> .
<code>\fp_while_do:nNnn</code>	<code>\fp_while_do:nNnn {&lt;fpexpr<sub>1</sub>&gt;} &lt;relation&gt; {&lt;fpexpr<sub>2</sub>&gt;} {&lt;code&gt;}</code>
<small>New: 2012-08-16</small>	Evaluates the relationship between the two <i>&lt;floating point expressions&gt;</i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <b>true</b> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <b>false</b> .
<code>\fp_do_until:nn</code>	<code>\fp_do_until:nn { &lt;fpexpr<sub>1</sub>&gt; &lt;relation&gt; &lt;fpexpr<sub>2</sub>&gt; } {&lt;code&gt;}</code>
<small>New: 2012-08-16 Updated: 2013-12-14</small>	Places the <i>&lt;code&gt;</i> in the input stream for TeX to process, and then evaluates the relationship between the two <i>&lt;floating point expressions&gt;</i> as described for <code>\fp_compare:nTF</code> . If the test is <b>false</b> then the <i>&lt;code&gt;</i> is inserted into the input stream again and a loop occurs until the <i>&lt;relation&gt;</i> is <b>true</b> .
<code>\fp_do_while:nn</code>	<code>\fp_do_while:nn { &lt;fpexpr<sub>1</sub>&gt; &lt;relation&gt; &lt;fpexpr<sub>2</sub>&gt; } {&lt;code&gt;}</code>
<small>New: 2012-08-16 Updated: 2013-12-14</small>	Places the <i>&lt;code&gt;</i> in the input stream for TeX to process, and then evaluates the relationship between the two <i>&lt;floating point expressions&gt;</i> as described for <code>\fp_compare:nTF</code> . If the test is <b>true</b> then the <i>&lt;code&gt;</i> is inserted into the input stream again and a loop occurs until the <i>&lt;relation&gt;</i> is <b>false</b> .
<code>\fp_until_do:nn</code>	<code>\fp_until_do:nn { &lt;fpexpr<sub>1</sub>&gt; &lt;relation&gt; &lt;fpexpr<sub>2</sub>&gt; } {&lt;code&gt;}</code>
<small>New: 2012-08-16 Updated: 2013-12-14</small>	Evaluates the relationship between the two <i>&lt;floating point expressions&gt;</i> as described for <code>\fp_compare:nTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <b>false</b> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <b>true</b> .
<code>\fp_while_do:nn</code>	<code>\fp_while_do:nn { &lt;fpexpr<sub>1</sub>&gt; &lt;relation&gt; &lt;fpexpr<sub>2</sub>&gt; } {&lt;code&gt;}</code>
<small>New: 2012-08-16 Updated: 2013-12-14</small>	Evaluates the relationship between the two <i>&lt;floating point expressions&gt;</i> as described for <code>\fp_compare:nTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <b>true</b> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <b>false</b> .

---

\fp\_step\_function:nnnN  $\star$  \fp\_step\_function:nnnN {*initial value*} {*step*} {*final value*} {*function*}  
\fp\_step\_function:nnnc  $\star$  This function first evaluates the *initial value*, *step* and *final value*, each of which should be a floating point expression evaluating to a floating point number, not a tuple. The *function* is then placed in front of each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*). The *step* must be non-zero. If the *step* is positive, the loop stops when the *value* becomes larger than the *final value*. If the *step* is negative, the loop stops when the *value* becomes smaller than the *final value*. The *function* should absorb one numerical argument. For example

```
\cs_set:Npn \my_func:n #1 { [I~saw~#1] \quad }  
\fp_step_function:nnnN { 1.0 } { 0.1 } { 1.5 } \my_func:n
```

would print

```
[I saw 1.0] [I saw 1.1] [I saw 1.2] [I saw 1.3] [I saw 1.4] [I saw 1.5]
```

**TEXhackers note:** Due to rounding, it may happen that adding the *step* to the *value* does not change the *value*; such cases give an error, as they would otherwise lead to an infinite loop.

---

\fp\_step\_inline:nnnn \fp\_step\_inline:nnnn {*initial value*} {*step*} {*final value*} {*code*}

---

New: 2016-11-21  
Updated: 2016-12-06 This function first evaluates the *initial value*, *step* and *final value*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream with #1 replaced by the current *value*. Thus the *code* should define a function of one argument (#1).

---

\fp\_step\_variable:nnnN \fp\_step\_variable:nnnN  
{*initial value*} {*step*} {*final value*} {*tl var*} {*code*}

---

New: 2017-04-12 This function first evaluates the *initial value*, *step* and *final value*, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each *value* from the *initial value* to the *final value* in turn (using *step* between each *value*), the *code* is inserted into the input stream, with the *tl var* defined as the current *value*. Thus the *code* should make use of the *tl var*.

## 28.6 Some useful constants, and scratch variables

---

\c\_zero\_fp Zero, with either sign.  
\c\_minus\_zero\_fp

New: 2012-05-08

---

\c\_one\_fp One as an fp: useful for comparisons in some places.

New: 2012-05-08

---

**\c\_inf\_fp** Infinity, with either sign. These can be input directly in a floating point expression as **\c\_minus\_inf\_fp** `inf` and `-inf`.

New: 2012-05-08

---

---

**\c\_e\_fp** The value of the base of the natural logarithm,  $e = \exp(1)$ .

Updated: 2012-05-08

---

---

**\c\_pi\_fp** The value of  $\pi$ . This can be input directly in a floating point expression as `pi`.

Updated: 2013-11-17

---

---

**\c\_one\_degree\_fp** The value of  $1^\circ$  in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

---

**\l\_tmpa\_fp** Scratch floating points for local assignment. These are never used by the kernel code, and **\l\_tmpb\_fp** so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

**\g\_tmpa\_fp** Scratch floating points for global assignment. These are never used by the kernel code, **\g\_tmpb\_fp** and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

## 28.7 Floating point exceptions

The functions defined in this section are experimental, and their functionality may be altered or removed altogether.

“Exceptions” may occur when performing some floating point operations, such as  $0 / 0$ , or  $10 ** 1e9999$ . The relevant IEEE standard defines 5 types of exceptions, of which we implement 4.

- *Overflow* occurs whenever the result of an operation is too large to be represented as a normal floating point number. This results in  $\pm\infty$ .
- *Underflow* occurs whenever the result of an operation is too close to 0 to be represented as a normal floating point number. This results in  $\pm 0$ .
- *Invalid operation* occurs for operations with no defined outcome, for instance  $0/0$  or  $\sin(\infty)$ , and results in a `nan`. It also occurs for conversion functions whose target type does not have the appropriate infinite or `nan` value (e.g., `\fp_to_dim:n`).
- *Division by zero* occurs when dividing a non-zero number by 0, or when evaluating functions at poles, e.g.,  $\ln(0)$  or  $\cot(0)$ . This results in  $\pm\infty$ .

(not yet) *Inexact* occurs whenever the result of a computation is not exact, in other words, almost always. At the moment, this exception is entirely ignored in L<sup>A</sup>T<sub>E</sub>X3.

To each exception we associate a “flag”: `fp_overflow`, `fp_underflow`, `fp_invalid_operation` and `fp_division_by_zero`. The state of these flags can be tested and modified with commands from `\3flag`.

By default, the “invalid operation” exception triggers an (expandable) error, and raises the corresponding flag. Other exceptions raise the corresponding flag but do not trigger an error. The behaviour when an exception occurs can be modified (using `\fp_trap:nn`) to either produce an error and raise the flag, or only raise the flag, or do nothing at all.

---

`\fp_trap:nn`    `\fp_trap:nn {<exception>} {<trap type>}`

New: 2012-07-19 All occurrences of the `<exception>` (`overflow`, `underflow`, `invalid_operation` or `division_by_zero`) within the current group are treated as `<trap type>`, which can be  
Updated: 2017-02-13

- `none`: the `<exception>` will be entirely ignored, and leave no trace;
- `flag`: the `<exception>` will turn the corresponding flag on when it occurs;
- `error`: additionally, the `<exception>` will halt the T<sub>E</sub>X run and display some information about the current operation in the terminal.

*This function is experimental, and may be altered or removed.*

---

`flag_fp_overflow`    Flags denoting the occurrence of various floating-point exceptions.  
`flag_fp_underflow`  
`flag_fp_invalid_operation`  
`flag_fp_division_by_zero`

## 28.8 Viewing floating points

---

`\fp_show:N`    `\fp_show:N {fp var}`  
`\fp_show:c`    `\fp_show:n {floating point expression}`  
`\fp_show:n`    Evaluates the `<floating point expression>` and displays the result in the terminal.

New: 2012-05-08  
Updated: 2021-04-29

---

`\fp_log:N`    `\fp_log:N {fp var}`  
`\fp_log:c`    `\fp_log:n {floating point expression}`  
`\fp_log:n`    Evaluates the `<floating point expression>` and writes the result in the log file.

New: 2014-08-22  
Updated: 2021-04-29

## 28.9 Floating point expressions

### 28.9.1 Input of floating point numbers

We support four types of floating point numbers:

- $\pm m \cdot 10^n$ , a floating point number, with integer  $1 \leq m \leq 10^{16}$ , and  $-10000 \leq n \leq 10000$ ;
- $\pm 0$ , zero, with a given sign;
- $\pm\infty$ , infinity, with a given sign;
- **nan**, is “not a number”, and can be either quiet or signalling (*not yet*: this distinction is currently unsupported);

Normal floating point numbers are stored in base 10, with up to 16 significant figures.

On input, a normal floating point number consists of:

- $\langle sign \rangle$ : a possibly empty string of + and - characters;
- $\langle significand \rangle$ : a non-empty string of digits together with zero or one dot;
- $\langle exponent \rangle$  optionally: the character e or E, followed by a possibly empty string of + and - tokens, and a non-empty string of digits.

The sign of the resulting number is + if  $\langle sign \rangle$  contains an even number of -, and - otherwise, hence, an empty  $\langle sign \rangle$  denotes a non-negative input. The stored significand is obtained from  $\langle significand \rangle$  by omitting the decimal separator and leading zeros, and rounding to 16 significant digits, filling with trailing zeros if necessary. In particular, the value stored is exact if the input  $\langle significand \rangle$  has at most 16 digits. The stored  $\langle exponent \rangle$  is obtained by combining the input  $\langle exponent \rangle$  (0 if absent) with a shift depending on the position of the significand and the number of leading zeros.

A special case arises if the resulting  $\langle exponent \rangle$  is either too large or too small for the floating point number to be represented. This results either in an overflow (the number is then replaced by  $\pm\infty$ ), or an underflow (resulting in  $\pm 0$ ).

The result is thus  $\pm 0$  if and only if  $\langle significand \rangle$  contains no non-zero digit (*i.e.*, consists only in characters 0, and an optional period), or if there is an underflow. Note that a single dot is currently a valid floating point number, equal to +0, but that is not guaranteed to remain true.

The  $\langle significand \rangle$  must be non-empty, so e1 and e-1 are not valid floating point numbers. Note that the latter could be mistaken with the difference of “e” and 1. To avoid confusions, the base of natural logarithms cannot be input as e and should be input as exp(1) or \c\_e\_fp (which is faster).

Special numbers are input as follows:

- **inf** represents  $+\infty$ , and can be preceded by any  $\langle sign \rangle$ , yielding  $\pm\infty$  as appropriate.
- **nan** represents a (quiet) non-number. It can be preceded by any sign, but that sign is ignored.
- Any unrecognizable string triggers an error, and produces a **nan**.
- Note that commands such as \infty, \pi, or \sin do not work in floating point expressions. They may silently be interpreted as completely unexpected numbers, because integer constants (allowed in expressions) are commonly stored as mathematical characters.

### 28.9.2 Precedence of operators

We list here all the operations supported in floating point expressions, in order of decreasing precedence: operations listed earlier bind more tightly than operations listed below them.

- Function calls (`sin`, `ln`, *etc*).
- Binary `**` and `^` (right associative).
- Unary `+`, `-`, `!`.
- Implicit multiplication by juxtaposition (`2pi`) when neither factor is in parentheses.
- Binary `*` and `/`, implicit multiplication by juxtaposition with parentheses (for instance `3(4+5)`).
- Binary `+` and `-`.
- Comparisons `>=`, `!=`, `<?`, *etc*.
- Logical `and`, denoted by `&&`.
- Logical `or`, denoted by `||`.
- Ternary operator `?:` (right associative).
- Comma (to build tuples).

The precedence of operations can be overridden using parentheses. In particular, the precedence of juxtaposition implies that

$$\begin{aligned} 1/2\text{pi} &= 1/(2\pi), \\ 1/2\text{pi}(\text{pi} + \text{pi}) &= (2\pi)^{-1}(\pi + \pi) \simeq 1, \\ \text{sin}2\text{pi} &= \sin(2)\pi \neq 0, \\ 2^{\wedge}2\text{max}(3, 5) &= 2^2 \max(3, 5) = 20, \\ \text{1in}/\text{1cm} &= (\text{1in})/(\text{1cm}) = 2.54. \end{aligned}$$

Functions are called on the value of their argument, contrarily to TeX macros.

### 28.9.3 Operations

We now present the various operations allowed in floating point expressions, from the lowest precedence to the highest. When used as a truth value, a floating point expression is `false` if it is  $\pm 0$ , and `true` otherwise, including when it is `nan` or a tuple such as  $(0, 0)$ . Tuples are only supported to some extent by operations that work with truth values (`?:`, `||`, `&&`, `!`), by comparisons (`!<=>?`), and by `+`, `-`, `*`, `/`. Unless otherwise specified, providing a tuple as an argument of any other operation yields the “invalid operation” exception and a `nan` result.

?: \fp\_eval:n { ⟨operand<sub>1</sub>⟩ ? ⟨operand<sub>2</sub>⟩ : ⟨operand<sub>3</sub>⟩ }

The ternary operator ?: results in ⟨operand<sub>2</sub>⟩ if ⟨operand<sub>1</sub>⟩ is true (not ±0), and ⟨operand<sub>3</sub>⟩ if ⟨operand<sub>1</sub>⟩ is false (±0). All three ⟨operands⟩ are evaluated in all cases; they may be tuples. The operator is right associative, hence

```
\fp_eval:n
{
  1 + 3 > 4 ? 1 :
  2 + 4 > 5 ? 2 :
  3 + 5 > 6 ? 3 : 4
}
```

first tests whether  $1 + 3 > 4$ ; since this isn't true, the branch following : is taken, and  $2 + 4 > 5$  is compared; since this is true, the branch before : is taken, and everything else is (evaluated then) ignored. That allows testing for various cases in a concise manner, with the drawback that all computations are made in all cases.

|| \fp\_eval:n { ⟨operand<sub>1</sub>⟩ || ⟨operand<sub>2</sub>⟩ }

If ⟨operand<sub>1</sub>⟩ is true (not ±0), use that value, otherwise the value of ⟨operand<sub>2</sub>⟩. Both ⟨operands⟩ are evaluated in all cases; they may be tuples. In ⟨operand<sub>1</sub>⟩ || ⟨operand<sub>2</sub>⟩ || ... || ⟨operands<sub>n</sub>⟩, the first true (nonzero) ⟨operand⟩ is used and if all are zero the last one (±0) is used.

&& \fp\_eval:n { ⟨operand<sub>1</sub>⟩ && ⟨operand<sub>2</sub>⟩ }

If ⟨operand<sub>1</sub>⟩ is false (equal to ±0), use that value, otherwise the value of ⟨operand<sub>2</sub>⟩. Both ⟨operands⟩ are evaluated in all cases; they may be tuples. In ⟨operand<sub>1</sub>⟩ && ⟨operand<sub>2</sub>⟩ && ... && ⟨operands<sub>n</sub>⟩, the first false (±0) ⟨operand⟩ is used and if none is zero the last one is used.

---

```
<          \fp_eval:n
=          {
>          ⟨operand1⟩ ⟨relation1⟩
?


---

Updated: 2013-12-14
          ...
          ⟨operandN⟩ ⟨relationN⟩
          ⟨operandN+1⟩
}
```

Each ⟨relation⟩ consists of a non-empty string of <, =, >, and ?, optionally preceded by !, and may not start with ?. This evaluates to +1 if all comparisons ⟨operand<sub>i</sub>⟩ ⟨relation<sub>i</sub>⟩ ⟨operand<sub>i+1</sub>⟩ are true, and +0 otherwise. All ⟨operands⟩ are evaluated (once) in all cases. See \fp\_compare:nTF for details.

+ \fp\_eval:n { ⟨operand<sub>1</sub>⟩ + ⟨operand<sub>2</sub>⟩ }
- \fp\_eval:n { ⟨operand<sub>1</sub>⟩ - ⟨operand<sub>2</sub>⟩ }

Computes the sum or the difference of its two ⟨operands⟩. The “invalid operation” exception occurs for  $\infty - \infty$ . “Underflow” and “overflow” occur when appropriate. These operations supports the itemwise addition or subtraction of two tuples, but if they have a different number of items the “invalid operation” exception occurs and the result is nan.

- `* \fp_eval:n { <operand1> * <operand2> }`  
`/ \fp_eval:n { <operand1> / <operand2> }`
  - Computes the product or the ratio of its two *<operands>*. The “invalid operation” exception occurs for  $\infty/\infty$ ,  $0/0$ , or  $0 * \infty$ . “Division by zero” occurs when dividing a finite non-zero number by  $\pm 0$ . “Underflow” and “overflow” occur when appropriate. When *<operand<sub>1</sub>>* is a tuple and *<operand<sub>2</sub>>* is a floating point number, each item of *<operand<sub>1</sub>>* is multiplied or divided by *<operand<sub>2</sub>>*. Multiplication also supports the case where *<operand<sub>1</sub>>* is a floating point number and *<operand<sub>2</sub>>* a tuple. Other combinations yield an “invalid operation” exception and a `nan` result.
- `+ \fp_eval:n { + <operand> }`  
`- \fp_eval:n { - <operand> }`  
`! \fp_eval:n { ! <operand> }`
  - The unary `+` does nothing, the unary `-` changes the sign of the *<operand>* (for a tuple, of all its components), and `! <operand>` evaluates to 1 if *<operand>* is false (is  $\pm 0$ ) and 0 otherwise (this is the `not` boolean function). Those operations never raise exceptions.
- `** \fp_eval:n { <operand1> ** <operand2> }`  
`^ \fp_eval:n { <operand1> ^ <operand2> }`
  - Raises *<operand<sub>1</sub>>* to the power *<operand<sub>2</sub>>*. This operation is right associative, hence  $2^{** 2 ** 3} = 256$ . If *<operand<sub>1</sub>>* is negative or  $-0$  then: the result’s sign is `+` if the *<operand<sub>2</sub>>* is infinite and  $(-1)^p$  if the *<operand<sub>2</sub>>* is  $p/5^q$  with *p*, *q* integers; the result is `+0` if `abs(<operand1>)^<operand2>` evaluates to zero; in other cases the “invalid operation” exception occurs because the sign cannot be determined. “Division by zero” occurs when raising  $\pm 0$  to a finite strictly negative power. “Underflow” and “overflow” occur when appropriate. If either operand is a tuple, “invalid operation” occurs.
- `abs \fp_eval:n { abs( <fpexpr> ) }`
  - Computes the absolute value of the *<fpexpr>*. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also `\fp_abs:n`.
- `exp \fp_eval:n { exp( <fpexpr> ) }`
  - Computes the exponential of the *<fpexpr>*. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
- `fact \fp_eval:n { fact( <fpexpr> ) }`
  - Computes the factorial of the *<fpexpr>*. If the *<fpexpr>* is an integer between  $-0$  and 3248 included, the result is finite and correctly rounded. Larger positive integers give  $+\infty$  with “overflow”, while `fact(+∞) = +∞` and `fact(nan) = nan` with no exception. All other inputs give `nan` with the “invalid operation” exception.
- `ln \fp_eval:n { ln( <fpexpr> ) }`
  - Computes the natural logarithm of the *<fpexpr>*. Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for  $\ln(-0)$ . “Division by zero” occurs when evaluating  $\ln(+0) = -\infty$ . “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

---

**logb**  $\star \text{\fp\_eval:n} \{ \text{logb}(\langle fexpr \rangle) \}$

New: 2018-11-03 Determines the exponent of the  $\langle fexpr \rangle$ , namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating  $\text{logb}(\pm 0) = -\infty$ . Other special values are  $\text{logb}(\pm\infty) = +\infty$  and  $\text{logb}(\text{nan}) = \text{nan}$ . If the operand is a tuple or is  $\text{nan}$ , then “invalid operation” occurs and the result is  $\text{nan}$ .

---

**max**  $\text{\fp\_eval:n} \{ \text{max}(\langle fexpr_1 \rangle, \langle fexpr_2 \rangle, \dots) \}$   
**min**  $\text{\fp\_eval:n} \{ \text{min}(\langle fexpr_1 \rangle, \langle fexpr_2 \rangle, \dots) \}$

Evaluates each  $\langle fexpr \rangle$  and computes the largest (smallest) of those. If any of the  $\langle fexpr \rangle$  is a  $\text{nan}$  or tuple, the result is  $\text{nan}$ . If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.

---

**round**  $\text{\fp\_eval:n} \{ \text{round}(\langle fexpr \rangle) \}$

**trunc**  $\text{\fp\_eval:n} \{ \text{round}(\langle fexpr_1 \rangle, \langle fexpr_2 \rangle) \}$

**ceil**  $\text{\fp\_eval:n} \{ \text{round}(\langle fexpr_1 \rangle, \langle fexpr_2 \rangle, \langle fexpr_3 \rangle) \}$

**floor** Only **round** accepts a third argument. Evaluates  $\langle fexpr_1 \rangle = x$  and  $\langle fexpr_2 \rangle = n$  and  $\langle fexpr_3 \rangle = t$  then rounds  $x$  to  $n$  places. If  $n$  is an integer, this rounds  $x$  to a multiple of  $10^{-n}$ ; if  $n = +\infty$ , this always yields  $x$ ; if  $n = -\infty$ , this yields one of  $\pm 0$ ,  $\pm\infty$ , or  $\text{nan}$ ; if  $n = \text{nan}$ , this yields  $\text{nan}$ ; if  $n$  is neither  $\pm\infty$  nor an integer, then an “invalid operation” exception is raised. When  $\langle fexpr_2 \rangle$  is omitted,  $n = 0$ , i.e.,  $\langle fexpr_1 \rangle$  is rounded to an integer. The rounding direction depends on the function.

- **round** yields the multiple of  $10^{-n}$  closest to  $x$ , with ties ( $x$  half-way between two such multiples) rounded as follows. If  $t$  is  $\text{nan}$  (or not given) the even multiple is chosen (“ties to even”), if  $t = \pm 0$  the multiple closest to 0 is chosen (“ties to zero”), if  $t$  is positive/negative the multiple closest to  $\infty/-\infty$  is chosen (“ties towards positive/negative infinity”).
- **floor** yields the largest multiple of  $10^{-n}$  smaller or equal to  $x$  (“round towards negative infinity”);
- **ceil** yields the smallest multiple of  $10^{-n}$  greater or equal to  $x$  (“round towards positive infinity”);
- **trunc** yields a multiple of  $10^{-n}$  with the same sign as  $x$  and with the largest absolute value less than that of  $x$  (“round towards zero”).

“Overflow” occurs if  $x$  is finite and the result is infinite (this can only happen if  $\langle fexpr_2 \rangle < -9984$ ). If any operand is a tuple, “invalid operation” occurs.

---

**sign**  $\text{\fp\_eval:n} \{ \text{sign}(\langle fexpr \rangle) \}$

Evaluates the  $\langle fexpr \rangle$  and determines its sign:  $+1$  for positive numbers and for  $+\infty$ ,  $-1$  for negative numbers and for  $-\infty$ ,  $\pm 0$  for  $\pm 0$ , and  $\text{nan}$  for  $\text{nan}$ . If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

---

sin	\fp_eval:n { sin( <fpexpr> ) }
cos	\fp_eval:n { cos( <fpexpr> ) }
tan	\fp_eval:n { tan( <fpexpr> ) }
cot	\fp_eval:n { cot( <fpexpr> ) }
csc	\fp_eval:n { csc( <fpexpr> ) }
sec	\fp_eval:n { sec( <fpexpr> ) }

---

Updated: 2013-11-17

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the  $\langle fpexpr \rangle$  given in radians. For arguments given in degrees, see `sind`, `cosd`, etc. Note that since  $\pi$  is irrational,  $\sin(8\pi)$  is not quite zero, while its analogue  $\text{sind}(8 \times 180)$  is exactly zero. The trigonometric functions are undefined for an argument of  $\pm\infty$ , leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

---

sind	\fp_eval:n { sind( <fpexpr> ) }
cosd	\fp_eval:n { cosd( <fpexpr> ) }
tand	\fp_eval:n { tand( <fpexpr> ) }
cotd	\fp_eval:n { cotd( <fpexpr> ) }
cscd	\fp_eval:n { cscd( <fpexpr> ) }
secd	\fp_eval:n { secd( <fpexpr> ) }

---

New: 2013-11-02

Computes the sine, cosine, tangent, cotangent, cosecant, or secant of the  $\langle fpexpr \rangle$  given in degrees. For arguments given in radians, see `sin`, `cos`, etc. Note that since  $\pi$  is irrational,  $\sin(8\pi)$  is not quite zero, while its analogue  $\text{sind}(8 \times 180)$  is exactly zero. The trigonometric functions are undefined for an argument of  $\pm\infty$ , leading to the “invalid operation” exception. Additionally, evaluating tangent, cotangent, cosecant, or secant at one of their poles leads to a “division by zero” exception. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

---

asin	\fp_eval:n { asin( <fpexpr> ) }
acos	\fp_eval:n { acos( <fpexpr> ) }
acsc	\fp_eval:n { acsc( <fpexpr> ) }
asec	\fp_eval:n { asec( <fpexpr> ) }

---

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the  $\langle fpexpr \rangle$  and returns the result in radians, in the range  $[-\pi/2, \pi/2]$  for `asin` and `acsc` and  $[0, \pi]$  for `acos` and `asec`. For a result in degrees, use `asind`, etc. If the argument of `asin` or `acos` lies outside the range  $[-1, 1]$ , or the argument of `acsc` or `asec` inside the range  $(-1, 1)$ , an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

---

asind	\fp_eval:n { asind( <fpexpr> ) }
acosd	\fp_eval:n { acosd( <fpexpr> ) }
acscd	\fp_eval:n { acscd( <fpexpr> ) }
asecd	\fp_eval:n { aecd( <fpexpr> ) }

---

New: 2013-11-02

Computes the arcsine, arccosine, arccosecant, or arcsecant of the  $\langle fpexpr \rangle$  and returns the result in degrees, in the range  $[-90, 90]$  for `asin` and `acsc` and  $[0, 180]$  for `acos` and `asec`. For a result in radians, use `asin`, etc. If the argument of `asin` or `acos` lies outside the range  $[-1, 1]$ , or the argument of `acsc` or `asec` inside the range  $(-1, 1)$ , an “invalid operation” exception is raised. “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.

---

<code>atan</code>	<code>\fp_eval:n { atan( &lt;fpexpr&gt; ) }</code>
<code>acot</code>	<code>\fp_eval:n { atan( &lt;fpexpr1&gt; , &lt;fpexpr2&gt; ) }</code>

---

New: 2013-11-02

<code>\fp_eval:n { acot( &lt;fpexpr&gt; ) }</code>
<code>\fp_eval:n { acot( &lt;fpexpr1&gt; , &lt;fpexpr2&gt; ) }</code>

---

Those functions yield an angle in radians: `atand` and `acotd` are their analogs in degrees. The one-argument versions compute the arctangent or arccotangent of the  $\langle fpexpr \rangle$ : arctangent takes values in the range  $[-\pi/2, \pi/2]$ , and arccotangent in the range  $[0, \pi]$ . The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates  $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$ : this is the arctangent of  $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ , possibly shifted by  $\pi$  depending on the signs of  $\langle fpexpr_1 \rangle$  and  $\langle fpexpr_2 \rangle$ . The two-argument arccotangent computes the angle in polar coordinates of the point  $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$ , equal to the arccotangent of  $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ , possibly shifted by  $\pi$ . Both two-argument functions take values in the wider range  $[-\pi, \pi]$ . The ratio  $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$  need not be defined for the two-argument arctangent: when both expressions yield  $\pm 0$ , or when both yield  $\pm\infty$ , the resulting angle is one of  $\{\pm\pi/4, \pm 3\pi/4\}$  depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

---

<code>atand</code>	<code>\fp_eval:n { atand( &lt;fpexpr&gt; ) }</code>
<code>acotd</code>	<code>\fp_eval:n { atand( &lt;fpexpr1&gt; , &lt;fpexpr2&gt; ) }</code>

---

New: 2013-11-02

<code>\fp_eval:n { acotd( &lt;fpexpr&gt; ) }</code>
<code>\fp_eval:n { acotd( &lt;fpexpr1&gt; , &lt;fpexpr2&gt; ) }</code>

---

Those functions yield an angle in degrees: `atand` and `acotd` are their analogs in radians. The one-argument versions compute the arctangent or arccotangent of the  $\langle fpexpr \rangle$ : arctangent takes values in the range  $[-90, 90]$ , and arccotangent in the range  $[0, 180]$ . The two-argument arctangent computes the angle in polar coordinates of the point with Cartesian coordinates  $(\langle fpexpr_2 \rangle, \langle fpexpr_1 \rangle)$ : this is the arctangent of  $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ , possibly shifted by 180 depending on the signs of  $\langle fpexpr_1 \rangle$  and  $\langle fpexpr_2 \rangle$ . The two-argument arccotangent computes the angle in polar coordinates of the point  $(\langle fpexpr_1 \rangle, \langle fpexpr_2 \rangle)$ , equal to the arccotangent of  $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$ , possibly shifted by 180. Both two-argument functions take values in the wider range  $[-180, 180]$ . The ratio  $\langle fpexpr_1 \rangle / \langle fpexpr_2 \rangle$  need not be defined for the two-argument arctangent: when both expressions yield  $\pm 0$ , or when both yield  $\pm\infty$ , the resulting angle is one of  $\{\pm 45, \pm 135\}$  depending on signs. The “underflow” exception can occur. If any operand is a tuple, “invalid operation” occurs.

---

<code>sqrt</code>	<code>\fp_eval:n { sqrt( &lt;fpexpr&gt; ) }</code>
-------------------	--

---

New: 2013-12-14

Computes the square root of the  $\langle fpexpr \rangle$ . The “invalid operation” is raised when the  $\langle fpexpr \rangle$  is negative or is a tuple; no other exception can occur. Special values yield  $\sqrt{-0} = -0$ ,  $\sqrt{+0} = +0$ ,  $\sqrt{+\infty} = +\infty$  and  $\sqrt{\text{nan}} = \text{nan}$ .

---

`rand` `\fp_eval:n { rand() }`

New: 2016-12-05 Produces a pseudo-random floating-point number (multiple of  $10^{-16}$ ) between 0 included and 1 excluded. This is not available in older versions of X<sub>E</sub>T<sub>E</sub>X. The random seed can be queried using `\sys_rand_seed:` and set using `\sys_gset_rand_seed:n`.

**TeXhackers note:** This is based on pseudo-random numbers provided by the engine's primitive `\pdfuniformdeviate` in pdfT<sub>E</sub>X, pT<sub>E</sub>X, upT<sub>E</sub>X and `\uniformdeviate` in LuaT<sub>E</sub>X and X<sub>E</sub>T<sub>E</sub>X. The underlying code is based on Metapost, which follows an additive scheme recommended in Section 3.6 of "The Art of Computer Programming, Volume 2".

While we are more careful than `\uniformdeviate` to preserve uniformity of the underlying stream of 28-bit pseudo-random integers, these pseudo-random numbers should of course not be relied upon for serious numerical computations nor cryptography.

---

`randint` `\fp_eval:n { randint( <fpexpr> ) }`  
`\fp_eval:n { randint( <fpexpr1>, <fpexpr2> ) }`

New: 2016-12-05 Produces a pseudo-random integer between 1 and  $\langle fpexpr \rangle$  or between  $\langle fpexpr_1 \rangle$  and  $\langle fpexpr_2 \rangle$  inclusive. The bounds must be integers in the range  $(-10^{16}, 10^{16})$  and the first must be smaller or equal to the second. See `rand` for important comments on how these pseudo-random numbers are generated.

---

`inf` The special values  $+\infty$ ,  $-\infty$ , and `nan` are represented as `inf`, `-inf` and `nan` (see `\c_-inf_fp`, `\c_minus_inf_fp` and `\c_nan_fp`).

---

`pi` The value of  $\pi$  (see `\c_pi_fp`).

---

`deg` The value of  $1^\circ$  in radians (see `\c_one_degree_fp`).

---

**em** Those units of measurement are equal to their values in **pt**, namely  
**ex**

<b>in</b>	$1 \text{ in} = 72.27 \text{ pt}$
<b>pt</b>	$1 \text{ pt} = 1 \text{ pt}$
<b>pc</b>	$1 \text{ pc} = 12 \text{ pt}$
<b>cm</b>	$1 \text{ cm} = \frac{1}{2.54} \text{ in} = 28.45275590551181 \text{ pt}$
<b>mm</b>	$1 \text{ mm} = \frac{1}{25.4} \text{ in} = 2.845275590551181 \text{ pt}$
<b>dd</b>	$1 \text{ dd} = 0.376065 \text{ mm} = 1.07000856496063 \text{ pt}$
<b>cc</b>	$1 \text{ cc} = 12 \text{ dd} = 12.84010277952756 \text{ pt}$
<b>nd</b>	$1 \text{ nd} = 0.375 \text{ mm} = 1.066978346456693 \text{ pt}$
<b>nc</b>	$1 \text{ nc} = 12 \text{ nd} = 12.80374015748031 \text{ pt}$
<b>bp</b>	$1 \text{ bp} = \frac{1}{72} \text{ in} = 1.00375 \text{ pt}$
<b>sp</b>	$1 \text{ sp} = 2^{-16} \text{ pt} = 1.52587890625 \times 10^{-5} \text{ pt}.$

---

The values of the (font-dependent) units **em** and **ex** are gathered from **T<sub>E</sub>X** when the surrounding floating point expression is evaluated.

---

**true** Other names for 1 and +0.  
**false**

---

**\fp\_abs:n** \* **\fp\_abs:n** {*floating point expression*}

New: 2012-05-14    Updated: 2012-07-08 Evaluates the *floating point expression* as described for **\fp\_eval:n** and leaves the absolute value of the result in the input stream. If the argument is  $\pm\infty$ , **nan** or a tuple, “invalid operation” occurs. Within floating point expressions, **abs()** can be used; it accepts  $\pm\infty$  and **nan** as arguments.

**\fp\_max:nn** \* **\fp\_max:nn** {*fp expression 1*} {*fp expression 2*}

New: 2012-09-26 **\fp\_min:nn** \* Evaluates the *floating point expressions* as described for **\fp\_eval:n** and leaves the resulting larger (**max**) or smaller (**min**) value in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, **max()** and **min()** can be used.

## 28.10 Disclaimer and roadmap

The package may break down if the escape character is among 0123456789\_+, or if it receives a **T<sub>E</sub>X** primitive conditional affected by **\exp\_not:N**.

The following need to be done. I’ll try to time-order the items.

- Function to count items in a tuple (and to determine if something is a tuple).
- Decide what exponent range to consider.

- Support signalling `nan`.
- Modulo and remainder, and rounding function `quantize` (and its friends analogous to `trunc`, `ceil`, `floor`).
- `\fp_format:nn {⟨fexpr⟩} {⟨format⟩}`, but what should `⟨format⟩` be? More general pretty printing?
- Add `and`, `or`, `xor`? Perhaps under the names `all`, `any`, and `xor`?
- Add  $\log(x, b)$  for logarithm of  $x$  in base  $b$ .
- `hypot` (Euclidean length). Cartesian-to-polar transform.
- Hyperbolic functions `cosh`, `sinh`, `tanh`.
- Inverse hyperbolics.
- Base conversion, input such as `0xAB.CDEF`.
- Factorial (not with !), gamma function.
- Improve coefficients of the `sin` and `tan` series.
- Treat upper and lower case letters identically in identifiers, and ignore underscores.
- Add an `array(1,2,3)` and `i=complex(0,1)`.
- Provide an experimental `map` function? Perhaps easier to implement if it is a single character, `@sin(1,2)`?
- Provide an `isnan` function analogue of `\fp_if_nan:nTF`?
- Support keyword arguments?

`Pgfmath` also provides box-measurements (depth, height, width), but boxes are not possible expandably.

Bugs, and tests to add.

- Check that functions are monotonic when they should.
- Add exceptions to `?:`, `!<=>?`, `&&`, `||`, and `!`.
- Logarithms of numbers very close to 1 are inaccurate.
- When rounding towards  $-\infty$ , `\dim_to_fp:n {0pt}` should return  $-0$ , not  $+0$ .
- The result of  $(\pm 0) + (\pm 0)$ , of  $x + (-x)$ , and of  $(-x) + x$  should depend on the rounding mode.
- `0e9999999999` gives a `TEX` “number too large” error.
- Subnormals are not implemented.

Possible optimizations/improvements.

- Document that `l3trial/l3fp-types` introduces tools for adding new types.
- In subsection 28.9.1, write a grammar.

- It would be nice if the `parse` auxiliaries for each operation were set up in the corresponding module, rather than centralizing in `l3fp-parse`.
- Some functions should get an `_o` ending to indicate that they expand after their result.
- More care should be given to distinguish expandable/restricted expandable (auxiliary and internal) functions.
- The code for the `ternary` set of functions is ugly.
- There are many `\~` missing in the doc to avoid bad line-breaks.
- The algorithm for computing the logarithm of the significand could be made to use a 5 terms Taylor series instead of 10 terms by taking  $c = 2000/(\lfloor 200x \rfloor + 1) \in [10, 95]$  instead of  $c \in [1, 10]$ . Also, it would then be possible to simplify the computation of  $t$ . However, we would then have to hard-code the logarithms of 44 small integers instead of 9.
- Improve notations in the explanations of the division algorithm (`l3fp-basics`).
- Understand and document `\_fp_basics_pack_weird_low>NNNNw` and `\_fp_basics_pack_weird_high>NNNNNNNNw` better. Move the other `basics_pack` auxiliaries to `l3fp-aux` under a better name.
- Find out if underflow can really occur for trigonometric functions, and redoc as appropriate.
- Add bibliography. Some of Kahan's articles, some previous TeX fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, `@/` or whatever)?

# Chapter 29

## The **l3fparray** package: fast global floating point arrays

### 29.1 **l3fparray** documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast **l3seq**, where access time is linear). The interface is very close to that of **l3intarray**. The size of the array is fixed and must be given at point of initialisation

---

**\fparray\_new:Nn** *\fparray\_new:Nn <fparray var> {<size>}*

New: 2018-05-05

Evaluates the integer expression *<size>* and allocates an *<floating point array variable>* with that number of (zero) entries. The variable name should start with **\g\_** because assignments are always global.

---

**\fparray\_count:N \*** *\fparray\_count:N <fparray var>*

New: 2018-05-05

Expands to the number of entries in the *<floating point array variable>*. This is performed in constant time.

---

**\fparray\_gset:Nnn** *\fparray\_gset:Nnn <fparray var> {<position>} {<value>}*

New: 2018-05-05

Stores the result of evaluating the floating point expression *<value>* into the *<floating point array variable>* at the (integer expression) *<position>*. If the *<position>* is not between 1 and the **\fparray\_count:N**, an error occurs. Assignments are always global.

---

**\fparray\_gzero:N** *\fparray\_gzero:N <fparray var>*

New: 2018-05-05

Sets all entries of the *<floating point array variable>* to +0. Assignments are always global.

---

**\fparray\_item:Nn** \* *\fparray\_item:Nn <fparray var> {<position>}*

**\fparray\_item\_to\_tl:Nn** \*

Applies **\fp\_use:N** or **\fp\_to\_tl:N** (respectively) to the floating point entry stored at the (integer expression) *<position>* in the *<floating point array variable>*. If the *<position>* is not between 1 and the **\fparray\_count:N**, an error occurs.

# Chapter 30

## The `\I3cctab` package Category code tables

A category code table enables rapid switching of all category codes in one operation. For LuaTeX, this is possible over the entire Unicode range. For other engines, only the 8-bit range (0–255) is covered by such tables.

### 30.1 Creating and initialising category code tables

---

`\cctab_new:N` `\cctab_new:N` *(category code table)*

`\cctab_new:c`

Creates a new *(category code table)* variable or raises an error if the name is already taken. The declaration is global. The *(category code table)* is initialised with the codes as used by iniTeX.

---

`\cctab_const:Nn` `\cctab_const:Nn` *(category code table)* *{(category code set up)}*

`\cctab_const:cn`

Creates a new *(category code table)*, applies (in a group) the *(category code set up)* on top of iniTeX settings, then saves them globally as a constant table. The *(category code set up)* can include a call to `\cctab_select:N`.

---

`\cctab_gset:Nn` `\cctab_gset:Nn` *(category code table)* *{(category code set up)}*

`\cctab_gset:cn`

Starting from the iniTeX category codes, applies (in a group) the *(category code set up)*, then saves them globally in the *(category code table)*. The *(category code set up)* can include a call to `\cctab_select:N`.

### 30.2 Using category code tables

---

`\cctab_begin:N` `\cctab_begin:N` *(category code table)*

`\cctab_begin:c`

Switches locally the category codes in force to those stored in the *(category code table)*. The prevailing codes before the function is called are added to a stack, for use with `\cctab_end:`. This function does not start a TeX group.

---

`\cctab_end:` `\cctab_end:`

Updated: 2020-07-02 Ends the scope of a *category code table* started using `\cctab_begin:N`, returning the codes to those in force before the matching `\cctab_begin:N` was used. This must be used within the same TEX group (and at the same TEX group level) as the matching `\cctab_begin:N`.

---

`\cctab_select:N` `\cctab_select:N` *(category code table)*

New: 2020-05-19 Updated: 2020-07-02 Selects the *category code table* for the scope of the current group. This is in particular useful in the *(setup)* arguments of `\tl_set_rescan:Nnn`, `\tl_rescan:nn`, `\cctab_const:Nn`, and `\cctab_gset:Nn`.

---

`\cctab_item:Nn` \* `\cctab_item:Nn` *(category code table)* *{(integer expression)}*

New: 2021-05-10 Determines the *(character)* with character code given by the *(integer expression)* and expands to its category code specified by the *(category code table)*.

### 30.3 Category code table conditionals

---

`\cctab_if_exist_p:N` \* `\cctab_if_exist_p:N` *(category code table)*

`\cctab_if_exist_p:c` \* `\cctab_if_exist:NTF` *(category code table)* *{(true code)}* *{(false code)}*

`\cctab_if_exist:NTF` \* Tests whether the *(category code table)* is currently defined. This does not check that the `\cctab_if_exist:cTF` \* *(category code table)* really is a category code table.

### 30.4 Constant category code tables

---

`\c_code_cctab`

Category code table for the `expl3` code environment; this does *not* include `\o`, which is retained as an “other” character.

---

`\c_document_cctab`

Category code table for a standard LATEX document, as set by the LATEX kernel. In particular, the upper-half of the 8-bit range will be set to “active” with `pdfTEX` *only*. No `babel` shorthands will be activated.

---

`\c_initex_cctab`

Category code table as set up by `initTEX`.

---

Updated: 2020-07-02

---

`\c_other_cctab`

Category code table where all characters have category code 12 (other).

---

Updated: 2020-07-02

---

`\c_str_cctab`

Category code table where all characters have category code 12 (other) with the exception of spaces, which have category code 10 (space).

**Part V**  
**Text manipulation**

## Chapter 31

# The `I3unicode` package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. Most of the code here is internal, but there are a small set of public functions. These work with Unicode *(codepoints)* and are designed to give useable results with both Unicode-aware and 8-bit engines.

---

`\codepoint_generate:nn *` `\codepoint_generate:nn {\<codepoint>} {\<catcode>}`

---

New: 2022-10-09

Updated: 2022-11-09

---

Generates one or more character tokens representing the  $\langle \text{codepoint} \rangle$ . With Unicode engines, exactly one character token will be generated, and this will have the  $\langle \text{catcode} \rangle$  specified as the second argument:

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 10 (space)
- 11 (letter)
- 12 (other)
- 13 (active)

For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the  $\langle \text{codepoint} \rangle$ . For all codepoints outside of the classical ASCII range, the generated character tokens will be active (category code 13); for codepoints in the ASCII range, the given  $\langle \text{catcode} \rangle$  will be used. To allow the result of this function to be used inside a expansion context, the result is protected by `\exp_not:n`.

---

`\codepoint_str_generate:n *` `\codepoint_str_generate:n {\<codepoint>}`

---

New: 2022-10-09

---

Generates one or more character tokens representing the  $\langle \text{codepoint} \rangle$ . With Unicode engines, exactly one character token will be generated. For 8-bit engines, between one and four character tokens will be produced: these will be the bytes of the UTF-8 representation of the  $\langle \text{codepoint} \rangle$ . All of the generated character tokens will be of category code 12, except any spaces (codepoint 32), which will be category code 10.

---

`\codepoint_to_nfd:n *` `\codepoint_to_nfd:n {\<codepoint>}`

---

New: 2022-10-09

Converts the  $\langle \text{codepoint} \rangle$  to the Unicode Normalization Form Canonical Decomposition. The generated character(s) will have the current category code as they would if typed in directly for Unicode engines; for 8-bit engines, active characters are used for all codepoints outside of the ASCII range.

## Chapter 32

# The `I3text` package: text processing

This module deals with manipulation of (formatted) text; such material is comprised of a restricted set of token list content. The functions provided here concern conversion of textual content for example in case changing, generation of bookmarks and extraction to tags. All of the major functions operate by expansion. Begin-group and end-group tokens in the  $\langle text \rangle$  are normalized and become  $\{$  and  $\}$ , respectively.

### 32.1 Expanding text

---

```
\text_expand:n * \text_expand:n {\langle text \rangle}
```

---

New: 2020-01-02

Takes user input  $\langle text \rangle$  and expands the content. Protected commands (typically formatting) are left in place, and no processing takes place of math mode material (as delimited by pairs given in `\l_text_math_delims_t1` or as the argument to commands listed in `\l_text_math_arg_t1`). Commands which are neither engine- nor L<sup>A</sup>T<sub>E</sub>X protected are expanded exhaustively. Any commands listed in `\l_text_expand_exclude_t1`, `\l_text_accents_t1` and `\l_text_letterlike_t1` are excluded from expansion.

---

```
\text_declare_expand_equivalent:Nn \text_declare_expand_equivalent:Nn \langle cmd \rangle {\langle replacement \rangle}  
\text_declare_expand_equivalent:cn
```

---

New: 2020-01-22

Declares that the  $\langle replacement \rangle$  tokens should be used whenever the  $\langle cmd \rangle$  (a single token) is encountered. The  $\langle replacement \rangle$  tokens should be expandable.

## 32.2 Case changing

---

```
\text_lowercase:n      * \text_uppercase:n  {\tokens}
\text_uppercase:n      * \text_uppercase:nn {\language} {\tokens}
\text_titlecase:n      * Takes user input \text{text} first applies \text_expand, then transforms the case of character
\text_titlecase:first:n * tokens as specified by the function name. The category code of letters are not changed
\text_lowercase:nn       * by this process when Unicode engines are used; in 8-bit engines, case changed charters
\text_uppercase:nn       * in the ASCII range will have the current prevailing category code, while those outside of
\text_titlecase:nn       * it will be represented by active characters.
```

---

New: 2019-11-20

Updated: 2022-10-13

---

Upper- and lowercase have the obvious meanings. Titlecasing may be regarded informally as converting the first character of the  $\langle \text{tokens} \rangle$  to uppercase and the rest to lowercase. However, the process is more complex than this as there are some situations where a single lowercase character maps to a special form, for example *ij* in Dutch which becomes *IJ*. The `titlecase_first` variant does not attempt any case changing at all after the first letter has been processed.

Importantly, notice that these functions are intended for working with user *text for typesetting*. For case changing programmatic data see the `!3str` module and discussion there of `\str_lowercase:n`, `\str_uppercase:n` and `\str_casfold:n`.

Case changing does not take place within math mode material so for example

```
\text_uppercase:n { Some~text~$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

The first mandatory argument of commands listed in `\l_text_case_exclude_arg_t1` is excluded from case changing; the latter are entirely non-textual content (such as labels).

The standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. For pTeX, only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Language-sensitive conversions are enabled using the  $\langle \text{language} \rangle$  argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- Armenian (`hy` and `hy-x-yiwn`) The setting `hy` maps the codepoint U+0587, the ligature of letters *ech* and *yiwn*, to the codepoints for capital *ech* and *vew* when uppercasing: this follows the spelling reform which is used in Armenia. The alternative `hy-x-yiwn` maps U+0587 to capital *ech* and *yiwn* on uppercasing (also the output if Armenian is not selected at all).
- Azeri and Turkish (`az` and `tr`). The case pairs I/i-dotless and I-dot/i are activated for these languages. The combining dot mark is removed when lowercasing I-dot and introduced when upper casing i-dotless.
- German (`de-x-eszett`). An alternative mapping for German in which the lower-case *Eszett* maps to a *großes Eszett*. Since there is a T1 slot for the *großes Eszett* in T1, this tailoring is available with pdfTeX as well as in the Unicode T<sub>E</sub>X engines.

- Greek (**e1**). Removes accents from Greek letters when uppercasing; titlecasing leaves accents in place. A variant **e1-x-iota** is available which retains the *yptogrammeni* (subscript muted iota) when uppercasing: the standard version converts these to a following capital iota.
- Lithuanian (**lt**). The lowercase letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lowercasing of the relevant uppercase letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when uppercasing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Medieval Latin (**la-xmedieval**). The characters u and V are interchanged on case changing.
- Dutch (**n1**). Capitalisation of ij at the beginning of titlecased input produces IJ rather than Ij. The output retains two separate letters, thus this transformation is available using pdfTEX.

For titlecasing, note that there are two functions available. The function `\text-titlecase:n` applies (broadly) uppercasing to the first letter of the input, then lowercasing to the remainder. In contrast, `\text_titlecase_first:n` *only* carries out the uppercasing operation, and leaves the balance of the input unchanged. Determining whether non-letter characters at the start of text should switch from upper- to lowercasing is controllable. When `\l_text_titlecase_check_letter_bool` is **true**, characters which are not letters (category code 11) are left unchanged and “skipped”: the first *letter* is uppercased. (With 8-bit engines, this is extended to active characters which form part of a multi-byte letter codepoint.) When `\l_text_titlecase_check_letter_bool` is **false**, the first character is uppercased, and the rest lowercased, irrespective of the nature of the character.

---

```
\text_declare_case_equivalent:Nn \text_declare_case_equivalent:Nn <cmd> {<replacement>}
\text_declare_case_equivalent:cn


---


New: 2022-07-04
```

---

Declares that the `<replacement>` tokens should be used whenever the `<cmd>` (a single token) is encountered during case changing.

---

```
\text_case_switch:nnnn * \text_case_switch:nnnn {<normal>} {<upper>} {<lower>} {<title>}
```

---

New: 2022-07-04 Context-sensitive function which will expand to one of the `<normal>`, `<upper>`, `<lower>` or `<title>` tokens depending on the current case changing operation. Outside of case changing, the `<normal>` tokens are produced. Within case changing, the appropriate mapping tokens are inserted.

## 32.3 Removing formatting from text

---

`\text_purify:n * \text_purify:n {<text>}`

New: 2020-03-05

Updated: 2020-05-14

Takes user input  $\langle text \rangle$  and expands as described for `\text_expand:n`, then removes all functions from the resulting text. Math mode material (as delimited by pairs given in `\l_text_math_delims_tl` or as the argument to commands listed in `\l_text_math_arg_tl`) is left contained in a pair of \$ delimiters. Non-expandable functions present in the  $\langle text \rangle$  must either have a defined equivalent (see `\text_declare_purify_equivalent:Nn`) or will be removed from the result. Implicit tokens are converted to their explicit equivalent.

---

`\text_declare_purify_equivalent:Nn \text_declare_purify_equivalent:Nn <cmd> {<replacement>}`  
`\text_declare_purify_equivalent:Nx`

New: 2020-03-05

Declares that the  $\langle replacement \rangle$  tokens should be used whenever the  $\langle cmd \rangle$  (a single token) is encountered. The  $\langle replacement \rangle$  tokens should be expandable.

## 32.4 Control variables

---

`\l_text_accents_tl` Lists commands which represent accents, and which are left unchanged by expansion. (Defined only for the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package.)

---

`\l_text_letterlike_tl` Lists commands which represent letters; these are left unchanged by expansion. (Defined only for the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package.)

---

`\l_text_math_arg_tl` Lists commands present in the  $\langle text \rangle$  where the argument of the command should be treated as math mode material. The treatment here is similar to `\l_text_math_delims_tl` but for a command rather than paired delimiters.

---

`\l_text_math_delims_tl` Lists pairs of tokens which delimit (in-line) math mode content; such content *may* be excluded from processing.

---

`\l_text_case_exclude_arg_tl`

Lists commands where the first mandatory argument is excluded from case changing.

---

`\l_text_expand_exclude_tl` Lists commands which are excluded from expansion. This protection includes everything up to and including their first braced argument.

---

`\l_text_titlecase_check_letter_bool`

Controls how the start of titlecasing is handled: when `true`, the first *letter* in text is considered. The standard setting is `true`.

## 32.5 Mapping to graphemes

Grapheme splitting is implemented using the algorithm described in Unicode Standard Annex #29. This includes support for extended grapheme clusters. Text starting with a line feed or carriage return character will drop this due to standard TeX processing. At present extended pictograms are not supported: these may be added in a future release.

---

`\text_map_function:nN ☆ \text_map_function:nN <text> {(function)}`

New: 2022-08-04

Takes user input *<text>* and expands as described for `\text_expand:n`, then maps over the *graphemes* within the result, passing each grapheme to the *(function)*. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The *(function)* should accept one argument as *(balanced text)*: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_inline:nn`.

---

`\text_map_inline:nn \text_map_inline:nn <text> {(inline function)}`

New: 2022-08-04

Takes user input *<text>* and expands as described for `\text_expand:n`, then maps over the *graphemes* within the result, passing each grapheme to the *(inline function)*. Broadly a grapheme is a “user perceived character”: the Unicode Consortium describe the decomposition of input to graphemes in depth, and the approach used here implements that algorithm. The *(inline function)* should consist of code which receives the grapheme as *(balanced text)*: this may be comprise codepoints or may be a control sequence. With 8-bit engines, the codepoint(s) themselves may of course be made up of multiple bytes: the mapping will pass the correct codepoints independent of the engine in use. See also `\text_map_function:nN`.

---

`\text_map_break: ☆ \text_map_break:  
\text_map_break:n ☆ \text_map_break:n {(code)}`

New: 2022-08-04

Used to terminate a `\text_map_...` function before all entries in the *<text>* have been processed. This normally takes place within a conditional statement.

## **Part VI**

# **Typesetting**

## Chapter 33

# The `\I3box` package

## Boxes

Box variables contain typeset material that can be inserted on the page or in other boxes. Their contents cannot be converted back to lists of tokens. There are three kinds of box operations: horizontal mode denoted with prefix `\hbox_`, vertical mode with prefix `\vbox_`, and the generic operations working in both modes with prefix `\box_`. For instance, a new box variable containing the words “Hello, world!” (in a horizontal box) can be obtained by the following code.

```
\box_new:N \l_hello_box
\hbox_set:Nn \l_hello_box { Hello, ~ world! }
```

The argument is typeset inside a `\TeX` group so that any variables assigned during the construction of this box restores its value afterwards.

Box variables from `\I3box` are compatible with those of `\LaTeX2\varepsilon` and plain `\TeX` and can be used interchangeably. The `\I3box` commands to construct boxes, such as `\hbox:n` or `\hbox_set:Nn`, are “color-safe”, meaning that

```
\hbox:n { \color_select:n { blue } Hello, } ~ world!
```

will result in “Hello,” taking the color blue, but “world!” remaining with the prevailing color outside the box.

### 33.1 Creating and initialising boxes

---

```
\box_new:N \box_new:N <box>
```

`\box_new:c` Creates a new `<box>` or raises an error if the name is already taken. The declaration is global. The `<box>` is initially void.

---

```
\box_clear:N \box_clear:N <box>
```

`\box_clear:c` Clears the content of the `<box>` by setting the box equal to `\c_empty_box`.

---

```
\box_gclear:N \box_gclear:c
```

---

`\box_clear_new:N` `\box_clear_new:N <box>`  
`\box_clear_new:c`  
`\box_gclear_new:N`  
`\box_gclear_new:c`

Ensures that the `<box>` exists globally by applying `\box_new:N` if necessary, then applies `\box_(g)clear:N` to leave the `<box>` empty.

---

---

`\box_set_eq:NN` `\box_set_eq:NN <box1> <box2>`  
`\box_set_eq:(cN|Nc|cc)`  
`\box_gset_eq:NN`  
`\box_gset_eq:(cN|Nc|cc)`

Sets the content of `<box1>` equal to that of `<box2>`.

---

---

`\box_if_exist_p:N` \* `\box_if_exist_p:N <box>`  
`\box_if_exist_p:c` \* `\box_if_exist:NTF <box> {<true code>} {<false code>}`  
`\box_if_exist:NTF` \* Tests whether the `<box>` is currently defined. This does not check that the `<box>` really is a box.

---

New: 2012-03-03

---

## 33.2 Using boxes

---

`\box_use:N` `\box_use:N <box>`  
`\box_use:c`

Inserts the current content of the `<box>` onto the current list for typesetting. An error is raised if the variable does not exist or if it is invalid.

---

**TeXhackers note:** This is the TeX primitive `\copy`.

---

`\box_move_right:nn` `\box_move_right:nn {<dimexpr>} {<box function>}`  
`\box_move_left:nn`

This function operates in vertical mode, and inserts the material specified by the `<box function>` such that its reference point is displaced horizontally by the given `<dimexpr>` from the reference point for typesetting, to the right or left as appropriate. The `<box function>` should be a box operation such as `\box_use:N \<box>` or a “raw” box specification such as `\vbox:n { xyz }`.

---

---

`\box_move_up:nn` `\box_move_up:nn {<dimexpr>} {<box function>}`  
`\box_move_down:nn`

This function operates in horizontal mode, and inserts the material specified by the `<box function>` such that its reference point is displaced vertically by the given `<dimexpr>` from the reference point for typesetting, up or down as appropriate. The `<box function>` should be a box operation such as `\box_use:N \<box>` or a “raw” box specification such as `\vbox:n { xyz }`.

---

### 33.3 Measuring and setting box dimensions

---

`\box_dp:N \box_dp:N <box>`

`\box_dp:c` Calculates the depth (below the baseline) of the `<box>` in a form suitable for use in a *(dimension expression)*.

**TeXhackers note:** This is the TeX primitive `\dp`.

---

`\box_ht:N \box_ht:N <box>`

`\box_ht:c` Calculates the height (above the baseline) of the `<box>` in a form suitable for use in a *(dimension expression)*.

**TeXhackers note:** This is the TeX primitive `\ht`.

---

`\box_wd:N \box_wd:N <box>`

`\box_wd:c` Calculates the width of the `<box>` in a form suitable for use in a *(dimension expression)*.

**TeXhackers note:** This is the TeX primitive `\wd`.

---

`\box_ht_plus_dp:N \box_ht_plus_dp:N <box>`

`\box_ht_plus_dp:c` Calculates the total vertical size (height plus depth) of the `<box>` in a form suitable for use in a *(dimension expression)*.  
New: 2021-05-05

---

`\box_set_dp:Nn \box_set_dp:Nn <box> {<dimension expression>}`

`\box_set_dp:cn` Set the depth (below the baseline) of the `<box>` to the value of the `{<dimension expression>}`.  
`\box_gset_dp:Nn`  
`\box_gset_dp:cn`

Updated: 2019-01-22

---

`\box_set_ht:Nn \box_set_ht:Nn <box> {<dimension expression>}`

`\box_set_ht:cn` Set the height (above the baseline) of the `<box>` to the value of the `{<dimension expression>}`.  
`\box_gset_ht:Nn`  
`\box_gset_ht:cn`

Updated: 2019-01-22

---

`\box_set_wd:Nn \box_set_wd:Nn <box> {<dimension expression>}`

`\box_set_wd:cn` Set the width of the `<box>` to the value of the `{<dimension expression>}`.  
`\box_gset_wd:Nn`  
`\box_gset_wd:cn`

Updated: 2019-01-22

## 33.4 Box conditionals

---

\box\_if\_empty\_p:N \* \box\_if\_empty\_p:N ⟨box⟩  
\box\_if\_empty\_p:c \* \box\_if\_empty:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}  
\box\_if\_empty:NTF \* Tests if ⟨box⟩ is a empty (equal to \c\_empty\_box).  
\box\_if\_empty:cTF \*

---

\box\_if\_horizontal\_p:N \* \box\_if\_horizontal\_p:N ⟨box⟩  
\box\_if\_horizontal\_p:c \* \box\_if\_horizontal:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}  
\box\_if\_horizontal:NTF \* Tests if ⟨box⟩ is a horizontal box.  
\box\_if\_horizontal:cTF \*

---

\box\_if\_vertical\_p:N \* \box\_if\_vertical\_p:N ⟨box⟩  
\box\_if\_vertical\_p:c \* \box\_if\_vertical:NTF ⟨box⟩ {⟨true code⟩} {⟨false code⟩}  
\box\_if\_vertical:NTF \* Tests if ⟨box⟩ is a vertical box.  
\box\_if\_vertical:cTF \*

---

## 33.5 The last box inserted

---

\box\_set\_to\_last:N \box\_set\_to\_last:N ⟨box⟩  
\box\_set\_to\_last:c  
\box\_gset\_to\_last:N Sets the ⟨box⟩ equal to the last item (box) added to the current partial list, removing the item from the list at the same time. When applied to the main vertical list, the ⟨box⟩ is always void as it is not possible to recover the last added item.  
\box\_gset\_to\_last:c

---

## 33.6 Constant boxes

---

\c\_empty\_box This is a permanently empty box, which is neither set as horizontal nor vertical.

Updated: 2012-11-04

**TeXhackers note:** At the TeX level this is a void box.

## 33.7 Scratch boxes

---

\l\_tmpa\_box Scratch boxes for local assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.  
\l\_tmpb\_box  
Updated: 2012-11-04

---

\g\_tmpa\_box Scratch boxes for global assignment. These are never used by the kernel code, and so are safe for use with any LATEX3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.  
\g\_tmpb\_box

## 33.8 Viewing box contents

---

`\box_show:N` `\box_show:N <box>`

`\box_show:c` Shows full details of the content of the `<box>` in the terminal.

Updated: 2012-05-11

---

`\box_show:Nnn` `\box_show:Nnn <box> {<intexpr1>} {<intexpr2>}`

`\box_show:cnn` Display the contents of `<box>` in the terminal, showing the first `<intexpr1>` items of the box, and descending into `<intexpr2>` group levels.  
New: 2012-05-11

---

`\box_log:N` `\box_log:N <box>`

`\box_log:c` Writes full details of the content of the `<box>` to the log.

New: 2012-05-11

---

`\box_log:Nnn` `\box_log:Nnn <box> {<intexpr1>} {<intexpr2>}`

`\box_log:cnn` Writes the contents of `<box>` to the log, showing the first `<intexpr1>` items of the box, and descending into `<intexpr2>` group levels.  
New: 2012-05-11

---

## 33.9 Boxes and color

All L<sup>A</sup>T<sub>E</sub>X3 boxes are “color safe”: a color set inside the box stops applying after the end of the box has occurred.

## 33.10 Horizontal mode boxes

---

`\hbox:n` `\hbox:n {<contents>}`

Updated: 2017-04-05 Typesets the `<contents>` into a horizontal box of natural width and then includes this box in the current list for typesetting.

---

`\hbox_to_wd:nn` `\hbox_to_wd:nn {<dimexpr>} {<contents>}`

Updated: 2017-04-05 Typesets the `<contents>` into a horizontal box of width `<dimexpr>` and then includes this box in the current list for typesetting.

---

`\hbox_to_zero:n` `\hbox_to_zero:n {<contents>}`

Updated: 2017-04-05 Typesets the `<contents>` into a horizontal box of zero width and then includes this box in the current list for typesetting.

---

`\hbox_set:Nn` `\hbox_set:Nn <box> {<contents>}`

`\hbox_set:cn` Typesets the `<contents>` at natural width and then stores the result inside the `<box>`.

`\hbox_gset:Nn`

`\hbox_gset:cn`

Updated: 2017-04-05

---

---

\hbox\_set\_to\_wd:Nnn \hbox\_set\_to\_wd:cnn  
\hbox\_gset\_to\_wd:Nnn \hbox\_gset\_to\_wd:cnn

Typesets the *contents* to the width given by the *dimexpr* and then stores the result inside the *box*.

Updated: 2017-04-05

---

\hbox\_overlap\_center:n \hbox\_overlap\_center:n {*contents*}

New: 2020-08-25 Typesets the *contents* into a horizontal box of zero width such that material protrudes equally to both sides of the insertion point.

\hbox\_overlap\_right:n \hbox\_overlap\_right:n {*contents*}

Updated: 2017-04-05 Typesets the *contents* into a horizontal box of zero width such that material protrudes to the right of the insertion point.

\hbox\_overlap\_left:n \hbox\_overlap\_left:n {*contents*}

Updated: 2017-04-05 Typesets the *contents* into a horizontal box of zero width such that material protrudes to the left of the insertion point.

---

\hbox\_set:Nw \hbox\_set:cw  
\hbox\_set\_end:  
\hbox\_gset:Nw \hbox\_gset:cw  
\hbox\_gset\_end:

Typesets the *contents* at natural width and then stores the result inside the *box*. In contrast to \hbox\_set:Nn this function does not absorb the argument when finding the *content*, and so can be used in circumstances where the *content* may not be a simple argument.

Updated: 2017-04-05

---

\hbox\_set\_to\_wd:Nnw \hbox\_set\_to\_wd:cnw  
\hbox\_gset\_to\_wd:Nnw \hbox\_gset\_to\_wd:cnw

Typesets the *contents* to the width given by the *dimexpr* and then stores the result inside the *box*. In contrast to \hbox\_set\_to\_wd:Nnn this function does not absorb the argument when finding the *content*, and so can be used in circumstances where the *content* may not be a simple argument

New: 2017-06-08

---

\hbox\_unpack:N \hbox\_unpack:N {*box*}  
\hbox\_unpack:c

Unpacks the content of the horizontal *box*, retaining any stretching or shrinking applied when the *box* was set.

**TeXhackers note:** This is the TeX primitive \unhcopy.

### 33.11 Vertical mode boxes

Vertical boxes inherit their baseline from their contents. The standard case is that the baseline of the box is at the same position as that of the last item added to the box. This means that the box has no depth unless the last item added to it had depth. As a result most vertical boxes have a large height value and small or zero depth. The exception are

`_top` boxes, where the reference point is that of the first item added. These tend to have a large depth and small height, although the latter is typically non-zero.

---

`\vbox:n` `\vbox:n {\langle contents \rangle}`

Updated: 2017-04-05 Typesets the `\langle contents \rangle` into a vertical box of natural height and includes this box in the current list for typesetting.

---

`\vbox_top:n` `\vbox_top:n {\langle contents \rangle}`

Updated: 2017-04-05 Typesets the `\langle contents \rangle` into a vertical box of natural height and includes this box in the current list for typesetting. The baseline of the box is equal to that of the *first* item added to the box.

---

`\vbox_to_ht:nn` `\vbox_to_ht:nn {\langle dimexpr \rangle} {\langle contents \rangle}`

Updated: 2017-04-05 Typesets the `\langle contents \rangle` into a vertical box of height `\langle dimexpr \rangle` and then includes this box in the current list for typesetting.

---

`\vbox_to_zero:n` `\vbox_to_zero:n {\langle contents \rangle}`

Updated: 2017-04-05 Typesets the `\langle contents \rangle` into a vertical box of zero height and then includes this box in the current list for typesetting.

---

`\vbox_set:Nn` `\vbox_set:Nn {\langle box \rangle} {\langle contents \rangle}`

`\vbox_set:cn`  
`\vbox_gset:Nn`  
`\vbox_gset:cn`

Updated: 2017-04-05

Typesets the `\langle contents \rangle` at natural height and then stores the result inside the `\langle box \rangle`.

---

`\vbox_set_top:Nn` `\vbox_set_top:Nn {\langle box \rangle} {\langle contents \rangle}`

`\vbox_set_top:cn`  
`\vbox_gset_top:Nn`  
`\vbox_gset_top:cn`

Updated: 2017-04-05

Typesets the `\langle contents \rangle` at natural height and then stores the result inside the `\langle box \rangle`. The baseline of the box is equal to that of the *first* item added to the box.

---

`\vbox_set_to_ht:Nnn` `\vbox_set_to_ht:Nnn {\langle box \rangle} {\langle dimexpr \rangle} {\langle contents \rangle}`

`\vbox_set_to_ht:cnn`  
`\vbox_gset_to_ht:Nnn`  
`\vbox_gset_to_ht:cnn`

Updated: 2017-04-05

Typesets the `\langle contents \rangle` to the height given by the `\langle dimexpr \rangle` and then stores the result inside the `\langle box \rangle`.

---

`\vbox_set:Nw` `\vbox_set:Nw {\langle box \rangle} {\langle contents \rangle} \vbox_set_end:`

`\vbox_set:cw`  
`\vbox_set_end:`  
`\vbox_gset:Nw`  
`\vbox_gset:cw`  
`\vbox_gset_end:`

Updated: 2017-04-05

Typesets the `\langle contents \rangle` at natural height and then stores the result inside the `\langle box \rangle`. In contrast to `\vbox_set:Nn` this function does not absorb the argument when finding the `\langle content \rangle`, and so can be used in circumstances where the `\langle content \rangle` may not be a simple argument.

---

```
\vbox_set_to_ht:Nnw \vbox_set_to_ht:cnw
\ vbox_gset_to_ht:Nnw \vbox_gset_to_ht:cnw
```

Typesets the *contents* to the height given by the *dimexpr* and then stores the result inside the *box*. In contrast to `\vbox_set_to_ht:Nnn` this function does not absorb the argument when finding the *content*, and so can be used in circumstances where the *content* may not be a simple argument

New: 2017-06-08

---

---

```
\vbox_set_split_to_ht>NNn \vbox_set_split_to_ht>NNn <box1> <box2> {<dimexpr>}
\vbox_set_split_to_ht:(cNn|Ncn|ccn)
\ vbox_gset_split_to_ht:NNn
\ vbox_gset_split_to_ht:(cNn|Ncn|ccn)
```

---

Updated: 2018-12-29

---

Sets *box<sub>1</sub>* to contain material to the height given by the *dimexpr* by removing content from the top of *box<sub>2</sub>* (which must be a vertical box).

---

```
\vbox_unpack:N \vbox_unpack:N <box>
```

---

Unpacks the content of the vertical *box*, retaining any stretching or shrinking applied when the *box* was set.

**TeXhackers note:** This is the TeX primitive `\unvcopy`.

### 33.12 Using boxes efficiently

The functions above for using box contents work in exactly the same way as for any other `expl3` variable. However, for efficiency reasons, it is also useful to have functions which *drop* box contents on use. When a box is dropped, the box becomes empty at the group level *where the box was originally set* rather than necessarily *at the current group level*. For example, with

```
\hbox_set:Nn \l_tmpa_box { A }
\group_begin:
\hbox_set:Nn \l_tmpa_box { B }
\group_begin:
\box_use_drop:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
\group_end:
\box_show:N \l_tmpa_box
```

the first use of `\box_show:N` will show an entirely cleared (void) box, and the second will show the letter A in the box.

These functions should be preferred when the content of the box is no longer required after use. Note that due to the unusual scoping behaviour of `drop` functions they may be applied to both local and global boxes: the latter will naturally be set and thus cleared at a global level.

---

`\box_use_drop:N` `\box_use_drop:N <box>`

`\box_use_drop:c`

Inserts the current content of the `<box>` onto the current list for typesetting then drops the box content. An error is raised if the variable does not exist or if it is invalid. This function may be applied to local or global boxes.

**TeXhackers note:** This is the `\box` primitive.

---

`\box_set_eq_drop:NN` `\box_set_eq_drop:NN <box1> <box2>`

`\box_set_eq_drop:(cN|Nc|cc)`

Sets the content of `<box1>` equal to that of `<box2>`, then drops `<box2>`.

New: 2019-01-17

---

---

`\box_gset_eq_drop:NN` `\box_gset_eq_drop:NN <box1> <box2>`

`\box_gset_eq_drop:(cN|Nc|cc)`

Sets the content of `<box1>` globally equal to that of `<box2>`, then drops `<box2>`.

New: 2019-01-17

---

---

`\hbox_unpack_drop:N` `\hbox_unpack_drop:N <box>`

`\hbox_unpack_drop:c`

Unpacks the content of the horizontal `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The original `<box>` is then dropped.

**TeXhackers note:** This is the TeX primitive `\unhbox`.

---

`\vbox_unpack_drop:N` `\vbox_unpack_drop:N <box>`

`\vbox_unpack_drop:c`

Unpacks the content of the vertical `<box>`, retaining any stretching or shrinking applied when the `<box>` was set. The original `<box>` is then dropped.

**TeXhackers note:** This is the TeX primitive `\unvbox`.

### 33.13 Affine transformations

Affine transformations are changes which (informally) preserve straight lines. Simple translations are affine transformations, but are better handled in TeX by doing the translation first, then inserting an unmodified box. On the other hand, rotation and resizing of boxed material can best be handled by modifying boxes. These transformations are described here.

---

```
\box_autosize_to_wd_and_ht:Nnn \box_autosize_to_wd_and_ht:Nnn <box> {\<x-size>} {\<y-size>}
```

```
\box_autosize_to_wd_and_ht:cnn
```

```
\box_gautosize_to_wd_and_ht:Nnn
```

```
\box_gautosize_to_wd_and_ht:cnn
```

---

New: 2017-04-04

Updated: 2019-01-22

---

Resizes the  $\langle box \rangle$  to fit within the given  $\langle x\text{-size} \rangle$  (horizontally) and  $\langle y\text{-size} \rangle$  (vertically); both of the sizes are dimension expressions. The  $\langle y\text{-size} \rangle$  is the height only: it does not include any depth. The updated  $\langle box \rangle$  is an **hbox**, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. The final size of the  $\langle box \rangle$  is the smaller of  $\{\langle x\text{-size} \rangle\}$  and  $\{\langle y\text{-size} \rangle\}$ , *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y\text{-size} \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_autosize_to_wd_and_ht_plus_dp:Nnn \box_autosize_to_wd_and_ht_plus_dp:Nnn <box> {\<x-size>} {\<y-size>}
```

```
\box_autosize_to_wd_and_ht_plus_dp:cnn {\<y-size>}
```

```
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
```

```
\box_gautosize_to_wd_and_ht_plus_dp:cnn
```

---

New: 2017-04-04

Updated: 2019-01-22

---

Resizes the  $\langle box \rangle$  to fit within the given  $\langle x\text{-size} \rangle$  (horizontally) and  $\langle y\text{-size} \rangle$  (vertically); both of the sizes are dimension expressions. The  $\langle y\text{-size} \rangle$  is the total vertical size (height plus depth). The updated  $\langle box \rangle$  is an **hbox**, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. The final size of the  $\langle box \rangle$  is the smaller of  $\{\langle x\text{-size} \rangle\}$  and  $\{\langle y\text{-size} \rangle\}$ , *i.e.* the result fits within the dimensions specified. Negative sizes cause the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y\text{-size} \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_ht:Nn \box_resize_to_ht:Nn <box> {\<y-size>}
```

```
\box_resize_to_ht:cn
```

```
\box_gresize_to_ht:Nn
```

```
\box_gresize_to_ht:cn
```

---

Updated: 2019-01-22

Resizes the  $\langle box \rangle$  to  $\langle y\text{-size} \rangle$  (vertically), scaling the horizontal size by the same amount;  $\langle y\text{-size} \rangle$  is a dimension expression. The  $\langle y\text{-size} \rangle$  is the height only: it does not include any depth. The updated  $\langle box \rangle$  is an **hbox**, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. A negative  $\langle y\text{-size} \rangle$  causes the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y\text{-size} \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_ht_plus_dp:Nn  \box_resize_to_ht_plus_dp:Nn <box> {<y-size>}
\box_resize_to_ht_plus_dp:cN
\box_gresize_to_ht_plus_dp:Nn
\box_gresize_to_ht_plus_dp:cN
```

---

Updated: 2019-01-22

Resizes the  $\langle box \rangle$  to  $\langle y-size \rangle$  (vertically), scaling the horizontal size by the same amount;  $\langle y-size \rangle$  is a dimension expression. The  $\langle y-size \rangle$  is the total vertical size (height plus depth). The updated  $\langle box \rangle$  is an `hbox`, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. A negative  $\langle y-size \rangle$  causes the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y-size \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_wd:Nn  \box_resize_to_wd:Nn <box> {<x-size>}
```

---

```
\box_resize_to_wd:cN
\box_gresize_to_wd:Nn
\box_gresize_to_wd:cN
```

---

Updated: 2019-01-22

Resizes the  $\langle box \rangle$  to  $\langle x-size \rangle$  (horizontally), scaling the vertical size by the same amount;  $\langle x-size \rangle$  is a dimension expression. The updated  $\langle box \rangle$  is an `hbox`, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. A negative  $\langle x-size \rangle$  causes the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle x-size \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_wd_and_ht:Nnn  \box_resize_to_wd_and_ht:Nnn <box> {<x-size>} {<y-size>}
```

---

```
\box_resize_to_wd_and_ht:cnn
\box_gresize_to_wd_and_ht:Nnn
\box_gresize_to_wd_and_ht:cnn
```

New: 2014-07-03

Updated: 2019-01-22

Resizes the  $\langle box \rangle$  to  $\langle x-size \rangle$  (horizontally) and  $\langle y-size \rangle$  (vertically): both of the sizes are dimension expressions. The  $\langle y-size \rangle$  is the height only and does not include any depth. The updated  $\langle box \rangle$  is an `hbox`, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. Negative sizes cause the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y-size \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_wd_and_ht_plus_dp:Nnn  \box_resize_to_wd_and_ht_plus_dp:Nnn <box> {<x-size>} {<y-size>}
```

---

```
\box_resize_to_wd_and_ht_plus_dp:cnn
\box_gresize_to_wd_and_ht_plus_dp:Nnn
\box_gresize_to_wd_and_ht_plus_dp:cnn
```

New: 2017-04-06

Updated: 2019-01-22

Resizes the  $\langle box \rangle$  to  $\langle x-size \rangle$  (horizontally) and  $\langle y-size \rangle$  (vertically): both of the sizes are dimension expressions. The  $\langle y-size \rangle$  is the total vertical size (height plus depth). The updated  $\langle box \rangle$  is an `hbox`, irrespective of the nature of the  $\langle box \rangle$  before the resizing is applied. Negative sizes cause the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y-size \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_rotate:Nn \box_rotate:Nn <box> {<angle>}
```

**\box\_rotate:cN** Rotates the  $\langle box \rangle$  by  $\langle angle \rangle$  (in degrees) anti-clockwise about its reference point. The reference point of the updated box is moved horizontally such that it is at the left side of the smallest rectangle enclosing the rotated material. The updated  $\langle box \rangle$  is an **hbox**, irrespective of the nature of the  $\langle box \rangle$  before the rotation is applied.

**Updated:** 2019-01-22

---

```
\box_scale:Nnn \box_scale:Nnn <box> {<x-scale>} {<y-scale>}
```

**\box\_scale:cnn** Scales the  $\langle box \rangle$  by factors  $\langle x-scale \rangle$  and  $\langle y-scale \rangle$  in the horizontal and vertical directions, respectively (both scales are integer expressions). The updated  $\langle box \rangle$  is an **hbox**, irrespective of the nature of the  $\langle box \rangle$  before the scaling is applied. Negative scalings cause the material in the  $\langle box \rangle$  to be reversed in direction, but the reference point of the  $\langle box \rangle$  is unchanged. Thus a negative  $\langle y-scale \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

**Updated:** 2019-01-22

### 33.14 Primitive box conditionals

---

```
\if_hbox:N * \if_hbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is  $\langle box \rangle$  is a horizontal box.

**TeXhackers note:** This is the TeX primitive **\ifhbox**.

---

```
\if_vbox:N * \if_vbox:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is  $\langle box \rangle$  is a vertical box.

**TeXhackers note:** This is the TeX primitive **\ifvbox**.

---

```
\if_box_empty:N * \if_box_empty:N <box>
  <true code>
\else:
  <false code>
\fi:
```

Tests is  $\langle box \rangle$  is an empty (void) box.

**TeXhackers note:** This is the TeX primitive **\ifvoid**.

# Chapter 34

## The **l3coffins** package Coffin code layer

The material in this module provides the low-level support system for coffins. For details about the design concept of a coffin, see the `xcoffins` module (in the `l3experimental` bundle).

### 34.1 Creating and initialising coffins

---

`\coffin_new:N` `\coffin_new:N`  $\langle \text{coffin} \rangle$   
`\coffin_new:c` Creates a new  $\langle \text{coffin} \rangle$  or raises an error if the name is already taken. The declaration is global. The  $\langle \text{coffin} \rangle$  is initially empty.  
New: 2011-08-17

---

`\coffin_clear:N` `\coffin_clear:N`  $\langle \text{coffin} \rangle$   
`\coffin_clear:c` Clears the content of the  $\langle \text{coffin} \rangle$ .  
`\coffin_gclear:N`  
`\coffin_gclear:c`  
New: 2011-08-17  
Updated: 2019-01-21

---

`\coffin_set_eq:NN` `\coffin_set_eq:NN`  $\langle \text{coffin}_1 \rangle$   $\langle \text{coffin}_2 \rangle$   
`\coffin_set_eq:(Nc|cN|cc)` Sets both the content and poles of  $\langle \text{coffin}_1 \rangle$  equal to those of  $\langle \text{coffin}_2 \rangle$ .  
`\coffin_gset_eq:NN`  
`\coffin_gset_eq:(Nc|cN|cc)`  
New: 2011-08-17  
Updated: 2019-01-21

---

`\coffin_if_exist_p:N` \* `\coffin_if_exist_p:N`  $\langle \text{coffin} \rangle$   
`\coffin_if_exist_p:c` \* `\coffin_if_exist:NTF`  $\langle \text{coffin} \rangle$  {{*true code*}} {{*false code*}}  
`\coffin_if_exist:NTF` \* Tests whether the  $\langle \text{coffin} \rangle$  is currently defined.  
`\coffin_if_exist:cTF` \*  
New: 2012-06-20

## 34.2 Setting coffin content and poles

---

```
\hcoffin_set:Nn \hcoffin_set:Nn <coffin> {<material>}
```

```
\hcoffin_set:cN
```

```
\hcoffin_gset:Nn
```

```
\hcoffin_gset:cN
```

Typesets the *<material>* in horizontal mode, storing the result in the *<coffin>*. The standard poles for the *<coffin>* are then set up based on the size of the typeset material.

---

New: 2011-08-17

Updated: 2019-01-21

---

---

```
\hcoffin_set:Nw \hcoffin_set:Nw <coffin> <material> \hcoffin_set_end:
```

```
\hcoffin_set:cw
```

```
\hcoffin_set_end:
```

```
\hcoffin_gset:Nw
```

```
\hcoffin_gset:cw
```

```
\hcoffin_gset_end:
```

---

New: 2011-09-10

Updated: 2019-01-21

---

---

```
\vcoffin_set:Nnn \vcoffin_set:Nnn <coffin> {<width>} {<material>}
```

```
\vcoffin_set:cnn
```

```
\vcoffin_gset:Nnn
```

```
\vcoffin_gset:cnn
```

Typesets the *<material>* in vertical mode constrained to the given *<width>* and stores the result in the *<coffin>*. The standard poles for the *<coffin>* are then set up based on the size of the typeset material.

---

New: 2011-08-17

Updated: 2019-01-21

---

---

```
\vcoffin_set:Nnw \vcoffin_set:Nnw <coffin> {<width>} {<material>} \vcoffin_set_end:
```

```
\vcoffin_set:cnw
```

```
\vcoffin_set_end:
```

```
\vcoffin_gset:Nnw
```

```
\vcoffin_gset:cnw
```

```
\vcoffin_gset_end:
```

Typesets the *<material>* in vertical mode constrained to the given *<width>* and stores the result in the *<coffin>*. The standard poles for the *<coffin>* are then set up based on the size of the typeset material. These functions are useful for setting the entire contents of an environment in a coffin.

---

New: 2011-09-10

Updated: 2019-01-21

---

---

```
\coffin_set_horizontal_pole:Nnn \coffin_set_horizontal_pole:Nnn <coffin>
```

```
\coffin_set_horizontal_pole:cnn {<pole>} {<offset>}
```

```
\coffin_gset_horizontal_pole:Nnn
```

```
\coffin_gset_horizontal_pole:cnn
```

---

New: 2012-07-20

Updated: 2019-01-21

---

Sets the *<pole>* to run horizontally through the *<coffin>*. The *<pole>* is placed at the *<offset>* from the baseline of the *<coffin>*. The *<offset>* should be given as a dimension expression.

---

```
\coffin_set_vertical_pole:Nnn \coffin_set_vertical_pole:Nnn <coffin> {\<pole>} {\<offset>}  
\coffin_set_vertical_pole:cnn  
\coffin_gset_vertical_pole:Nnn  
\coffin_gset_vertical_pole:cnn
```

---

New: 2012-07-20

Updated: 2019-01-21

---

Sets the  $\langle pole \rangle$  to run vertically through the  $\langle coffin \rangle$ . The  $\langle pole \rangle$  is placed at the  $\langle offset \rangle$  from the left-hand edge of the bounding box of the  $\langle coffin \rangle$ . The  $\langle offset \rangle$  should be given as a dimension expression.

### 34.3 Coffin affine transformations

---

```
\coffin_resize:Nnn \coffin_resize:Nnn <coffin> {\<width>} {\<total-height>}
```

```
\coffin_resize:cnn  
\coffin_gresize:Nnn  
\coffin_gresize:cnn
```

Resized the  $\langle coffin \rangle$  to  $\langle width \rangle$  and  $\langle total-height \rangle$ , both of which should be given as dimension expressions.

---

Updated: 2019-01-23

---

---

```
\coffin_rotate:Nn \coffin_rotate:Nn <coffin> {\<angle>}
```

```
\coffin_rotate:cn  
\coffin_grotate:Nn  
\coffin_grotate:cn
```

Rotates the  $\langle coffin \rangle$  by the given  $\langle angle \rangle$  (given in degrees counter-clockwise). This process rotates both the coffin content and poles. Multiple rotations do not result in the bounding box of the coffin growing unnecessarily.

---

---

```
\coffin_scale:Nnn \coffin_scale:Nnn <coffin> {\<x-scale>} {\<y-scale>}
```

```
\coffin_scale:cnn  
\coffin_gscale:Nnn  
\coffin_gscale:cnn
```

Scales the  $\langle coffin \rangle$  by a factors  $\langle x-scale \rangle$  and  $\langle y-scale \rangle$  in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

---

Updated: 2019-01-23

---

### 34.4 Joining and using coffins

---

```
\coffin_attach:NnnNnnnn
```

```
\coffin_attach:(cnnNnnnn|Nnnccnnnn|cnnccnnnn)
```

```
\coffin_gattach:NnnNnnnn
```

```
\coffin_gattach:(cnnNnnnn|Nnnccnnnn|cnnccnnnn)
```

---

```
\coffin_attach:NnnNnnnn
```

```
  <coffin1> {\<coffin1-pole1>} {\<coffin1-pole2>}
```

```
  <coffin2> {\<coffin2-pole1>} {\<coffin2-pole2>}
```

```
  {\<x-offset>} {\<y-offset>}
```

---

Updated: 2019-01-22

---

This function attaches  $\langle coffin_2 \rangle$  to  $\langle coffin_1 \rangle$  such that the bounding box of  $\langle coffin_1 \rangle$  is not altered, *i.e.*  $\langle coffin_2 \rangle$  can protrude outside of the bounding box of the coffin. The alignment is carried out by first calculating  $\langle handle_1 \rangle$ , the point of intersection of  $\langle coffin_1\text{-}pole_1 \rangle$  and  $\langle coffin_1\text{-}pole_2 \rangle$ , and  $\langle handle_2 \rangle$ , the point of intersection of  $\langle coffin_2\text{-}pole_1 \rangle$  and  $\langle coffin_2\text{-}pole_2 \rangle$ .  $\langle coffin_2 \rangle$  is then attached to  $\langle coffin_1 \rangle$  such that the relationship between  $\langle handle_1 \rangle$  and  $\langle handle_2 \rangle$  is described by the  $\langle x\text{-}offset \rangle$  and  $\langle y\text{-}offset \rangle$ . The two offsets should be given as dimension expressions.

---

<code>\coffin_join:NnnNnnnn</code> <code>\coffin_join:(cnnNnnnn Nnnccnnnn cnncnnnn)</code> <code>\coffin_gjoin:NnnNnnnn</code> <code>\coffin_gjoin:(cnnNnnnn Nnnccnnnn cnncnnnn)</code>	<code>\coffin_join:NnnNnnnn</code> <code>{\coffin_1} {\{coffin<sub>1</sub>-pole<sub>1</sub>\}} {\{coffin<sub>1</sub>-pole<sub>2</sub>\}}</code> <code>{\coffin_2} {\{coffin<sub>2</sub>-pole<sub>1</sub>\}} {\{coffin<sub>2</sub>-pole<sub>2</sub>\}}</code> <code>{\{x-offset\}} {\{y-offset\}}</code>
--	--

---

Updated: 2019-01-22

This function joins  $\langle \text{coffin}_2 \rangle$  to  $\langle \text{coffin}_1 \rangle$  such that the bounding box of  $\langle \text{coffin}_1 \rangle$  may expand. The new bounding box covers the area containing the bounding boxes of the two original coffins. The alignment is carried out by first calculating  $\langle \text{handle}_1 \rangle$ , the point of intersection of  $\langle \text{coffin}_1\text{-pole}_1 \rangle$  and  $\langle \text{coffin}_1\text{-pole}_2 \rangle$ , and  $\langle \text{handle}_2 \rangle$ , the point of intersection of  $\langle \text{coffin}_2\text{-pole}_1 \rangle$  and  $\langle \text{coffin}_2\text{-pole}_2 \rangle$ .  $\langle \text{coffin}_2 \rangle$  is then attached to  $\langle \text{coffin}_1 \rangle$  such that the relationship between  $\langle \text{handle}_1 \rangle$  and  $\langle \text{handle}_2 \rangle$  is described by the  $\langle x\text{-offset} \rangle$  and  $\langle y\text{-offset} \rangle$ . The two offsets should be given as dimension expressions.

---

<code>\coffin_typeset:Nnnnn</code> <code>\coffin_typeset:cnnnn</code>	<code>\coffin_typeset:Nnnnn</code> $\langle \text{coffin} \rangle$ $\{\langle \text{pole}_1 \rangle\}$ $\{\langle \text{pole}_2 \rangle\}$ $\{\langle x\text{-offset} \rangle\}$ $\{\langle y\text{-offset} \rangle\}$
--	---

---

Updated: 2012-07-20

Typesetting is carried out by first calculating  $\langle \text{handle} \rangle$ , the point of intersection of  $\langle \text{pole}_1 \rangle$  and  $\langle \text{pole}_2 \rangle$ . The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the  $\langle \text{handle} \rangle$  is described by the  $\langle x\text{-offset} \rangle$  and  $\langle y\text{-offset} \rangle$ . The two offsets should be given as dimension expressions. Typesetting a coffin is therefore analogous to carrying out an alignment where the “parent” coffin is the current insertion point.

## 34.5 Measuring coffins

---

<code>\coffin_dp:N</code> <code>\coffin_dp:N</code> $\langle \text{coffin} \rangle$ <code>\coffin_dp:c</code>	
--	--

---

Calculates the depth (below the baseline) of the  $\langle \text{coffin} \rangle$  in a form suitable for use in a  $\langle \text{dimension expression} \rangle$ .

---

<code>\coffin_ht:N</code> <code>\coffin_ht:N</code> $\langle \text{coffin} \rangle$ <code>\coffin_ht:c</code>	
--	--

---

Calculates the height (above the baseline) of the  $\langle \text{coffin} \rangle$  in a form suitable for use in a  $\langle \text{dimension expression} \rangle$ .

---

<code>\coffin_wd:N</code> <code>\coffin_wd:N</code> $\langle \text{coffin} \rangle$ <code>\coffin_wd:c</code>	
--	--

---

Calculates the width of the  $\langle \text{coffin} \rangle$  in a form suitable for use in a  $\langle \text{dimension expression} \rangle$ .

## 34.6 Coffin diagnostics

---

<code>\coffin_display_handles:Nn</code> <code>\coffin_display_handles:Nn</code> $\langle \text{coffin} \rangle$ $\{\langle \text{color} \rangle\}$ <code>\coffin_display_handles:cn</code>	
---	--

---

Updated: 2011-09-02

This function first calculates the intersections between all of the  $\langle \text{poles} \rangle$  of the  $\langle \text{coffin} \rangle$  to give a set of  $\langle \text{handles} \rangle$ . It then prints the  $\langle \text{coffin} \rangle$  at the current location in the source, with the position of the  $\langle \text{handles} \rangle$  marked on the coffin. The  $\langle \text{handles} \rangle$  are labelled as part of this process: the locations of the  $\langle \text{handles} \rangle$  and the labels are both printed in the  $\langle \text{color} \rangle$  specified.

---

```
\coffin_mark_handle:Nnnn \coffin_mark_handle:Nnnn <coffin> {\<pole1>} {\<pole2>} {\<color>}
```

---

```
\coffin_mark_handle:cnnn
```

---

Updated: 2011-09-02

This function first calculates the *<handle>* for the *<coffin>* as defined by the intersection of *<pole1>* and *<pole2>*. It then marks the position of the *<handle>* on the *<coffin>*. The *<handle>* are labelled as part of this process: the location of the *<handle>* and the label are both printed in the *<color>* specified.

---

```
\coffin_show_structure:N \coffin_show_structure:N <coffin>
```

---

```
\coffin_show_structure:c
```

---

Updated: 2015-08-01

This function shows the structural information about the *<coffin>* in the terminal. The width, height and depth of the typeset material are given, along with the location of all of the poles of the coffin.

Notice that the poles of a coffin are defined by four values: the *x* and *y* co-ordinates of a point that the pole passes through and the *x*- and *y*-components of a vector denoting the direction of the pole. It is the ratio between the later, rather than the absolute values, which determines the direction of the pole.

---

```
\coffin_log_structure:N \coffin_log_structure:N <coffin>
```

---

```
\coffin_log_structure:c
```

---

New: 2014-08-22

---

Updated: 2015-08-01

This function writes the structural information about the *<coffin>* in the log file. See also *\coffin\_show\_structure:N* which displays the result in the terminal.

---

```
\coffin_show:N \coffin_show:N <coffin>
```

---

```
\coffin_show:c \coffin_log:N <coffin>
```

---

```
\coffin_log:N
```

---

```
\coffin_log:c
```

Shows full details of poles and contents of the *<coffin>* in the terminal or log file. See *\coffin\_show\_structure:N* and *\box\_show:N* to show separately the pole structure and the contents.

---

```
\coffin_show:Nnn \coffin_show:Nnn <coffin> {\<intexpr1>} {\<intexpr2>}
```

---

```
\coffin_show:cnn \coffin_log:Nnn <coffin> {\<intexpr1>} {\<intexpr2>}
```

---

```
\coffin_log:Nnn
```

---

```
\coffin_log:cnn
```

---

New: 2021-05-11

Shows poles and contents of the *<coffin>* in the terminal or log file, showing the first *<intexpr1>* items in the coffin, and descending into *<intexpr2>* group levels. See *\coffin\_show\_structure:N* and *\box\_show:Nnn* to show separately the pole structure and the contents.

## 34.7 Constants and variables

---

```
\c_empty_coffin
```

A permanently empty coffin.

---

```
\l_tmpa_coffin
```

---

```
\l_tmpb_coffin
```

---

New: 2012-06-19

Scratch coffins for local assignment. These are never used by the kernel code, and so are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by other non-kernel code and so should only be used for short-term storage.

---

`\g_tmpa_coffin` Scratch coffins for global assignment. These are never used by the kernel code, and so  
`\g_tmpb_coffin` are safe for use with any L<sup>A</sup>T<sub>E</sub>X3-defined function. However, they may be overwritten by  
other non-kernel code and so should only be used for short-term storage.

New: 2019-01-24

# Chapter 35

## The `l3color` package Color support

### 35.1 Color in boxes

Controlling the color of text in boxes requires a small number of control functions, so that the boxed material uses the color at the point where it is set, rather than where it is used.

---

```
\color_group_begin: \color_group_begin:  
\color_group_end: ...  
New: 2011-09-03
```

---

Creates a color group: one used to “trap” color settings. This grouping is built in to for example `\hbox_set:Nn`.

---

```
\color_ensure_current: \color_ensure_current:
```

---

New: 2011-09-03 Ensures that material inside a box uses the foreground color at the point where the box is set, rather than that in force when the box is used. This function should usually be used within a `\color_group_begin: ... \color_group_end:` group.

### 35.2 Color models

A color *model* is a way to represent sets of colors. Different models are particularly suitable for different output methods, *e.g.* screen or print. Parameter-based models can describe a very large number of unique colors, and have a varying number of *axes* which define a color space. In contrast, various proprietary models are available which define *spot* colors (more formally separations).

Core models are used to pass color information to output; these are “native” to `l3color`. Core models use real numbers in the range [0, 1] to represent values. The core models supported here are

- `gray` Grayscale color, with a single axis running from 0 (fully black) to 1 (fully white)
- `rgb` Red-green-blue color, with three axes, one for each of the components

- **cmyk** Cyan-magenta-yellow-black color, with four axes, one for each of the components

There are also interface models: these are convenient for users but have to be manipulated before storing/passing to the backend. Interface models are primarily integer-based: see below for more detail. The supported interface models are

- **Gray** Grayscale color, with a single axis running from 0 (fully black) to 15 (fully white)
- **hsb** Hue-saturation-brightness color, with three axes, all real values in the range [0, 1] for hue saturation and brightness
- **Hsb** Hue-saturation-brightness color, with three axes, integer in the range [0, 360] for hue, real values in the range [0, 1] for saturation and brightness
- **HSB** Hue-saturation-brightness color, with three axes, integers in the range [0, 240] for hue, saturation and brightness
- **HTML** HTML format representation of RGB color given as a single six-digit hexadecimal number
- **RGB** Red-green-blue color, with three axes, one for each of the components, values as integers from 0 to 255
- **wave** Light wavelength, a real number in the range 380 to 780 (nanometres)

All interface models are internally stored as **rgb**.

Finally, there are a small number of models which are parsed to allow data transfer from **xcolor** but which should not be used by end-users. These are

- **cmy** Cyan-magenta-yellow color with three axes, one for each of the components; converted to **cmyk**
- **tHsb** “Tuned” hue-saturation-brightness color with three axes, integer in the range [0, 360] for hue, real values in the range [0, 1] for saturation and brightness; converted to **rgb** using the standard tuning map defined by **xcolor**
- **&spot** Spot color tint with one value; treated as a gray tint as spot color data is not available for extraction

To allow parsing of data from **xcolor**, any leading model up the first : will be discarded; the approach of selecting an internal form for data is *not* used in **l3color**.

Additional models may be created to allow mixing of separation colors with each other or with those from other models. See Section 35.9 for more detail of color support for additional models.

When color is selected by model, the *(values)* given are specified as a comma-separated list. The length of the list will therefore be determined by the detail of the model involved.

Color models (and interconversion) are complex, and more details are given in the manual to the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  **xcolor** package and in the *PostScript Language Reference Manual*, published by Addison-Wesley.

### 35.3 Color expressions

In addition to allowing specification of color by model and values, l3color also supports color expressions. These are created by combining one or more color names, with the amount of each specified as a percentage. The latter is given between ! symbols in the expression. Thus for example

```
red!50!green
```

is a mixture of 50 % red and 50 % green. A trailing percentage is interpreted as implicitly followed by `white`, and so

```
red!25
```

specifies 25 % red mixed with 75 % white.

Where the models for the mixed colors are different, the model of the first color is used. Thus

```
red!50!cyan
```

will result in a color specification using the `rgb` model, made up of 50% red and 50% of cyan *expressed in* `rgb`. This may be important as color model interconversion is not exact.

The one exception to the above is where the first model in an expression is `gray`. In this case, the order of mixing is “swapped” internally, so that for example

```
black!50!red
```

has the same result as

```
red!50!black
```

(the predefined colors `black` and `white` use the `gray` model).

Where more than two colors are mixed in an expression, evaluation takes place in a stepwise fashion. Thus in

```
cyan!50!magenta!10!yellow
```

the sub-expression

```
cyan!50!magenta
```

is first evaluated to give an intermediate color specification, before the second step

```
<intermediate>!10!yellow
```

where `<intermediate>` represents this transitory calculated value.

Within a color expression, . may be used to represent the color active for typesetting (the current color). This allows for example

```
.!50
```

to mean a mixture of 50 % of current color with white.

(Color expressions supported here are a subset of those provided by the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\epsilon$</sub>  `xcolor` package. At present, only such features as are clearly useful have been added here.)

## 35.4 Named colors

Color names are stored in a single namespace, which makes them accessible as part of color expressions. Whilst they are not reserved in a technical sense, the names `black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta` and `yellow` have special meaning and should not be redefined. Color names should be made up of letters, numbers and spaces only: other characters are reserved for use in color expressions. In particular, `.` represents the current color at the start of a color expression.

---

`\color_set:nn \color_set:nn {<name>} {<color expression>}`

Evaluates the `<color expression>` and stores the resulting color specification as the `<name>`.

---

`\color_set:nnn \color_set:nnn {<name>} {<model(s)>} {<value(s)>}`

Stores the color specification equivalent to the `<model(s)>` and `<values>` as the `<name>`.

---

`\color_set_eq:nn \color_set_eq:nn {<name1>} {<name2>}`

Copies the color specification in `<name2>` to `<name1>`. The special name `.` may be used to represent the current color, allowing it to be saved to a name.

---

`\color_if_exist_p:n * \color_if_exist:n {<name>}`  
`\color_if_exist:nTF * \color_if_exist:n {<name>} {<true code>} {<false code>}`

New: 2022-08-12 Tests whether `<name>` is currently defined to provide a color specification.

---

`\color_show:n \color_show:n {<name>}`  
`\color_log:n \color_log:n {<name>}`

New: 2021-05-11 Displays the color specification stored in the `<name>` on the terminal or log file.

## 35.5 Selecting colors

General selection of color is safe when split across pages: a stack is used to ensure that the correct color is re-selected on the new page.

These commands set the current color (`.`): other more specialised functions such as fill and stroke selectors do *not* adjust this value.

---

`\color_select:n \color_select:n {<color expression>}`

Parses the `<color expression>` and then activates the resulting color specification for typeset material.

---

`\color_select:nn \color_select:nn {<model(s)>} {<value(s)>}`

Activates the color specification equivalent to the `<model(s)>` and `<value(s)>` for typeset material.

---

`\l_color_fixed_model_t1` When this is set to a non-empty value, colors will be converted to the specified model when they are selected. Note that included images and similar are not influenced by this setting.

## 35.6 Colors for fills and strokes

Colors for drawing operations and so forth are split into strokes and fills (the latter may also be referred to as non-stroke color). The fill color is used for text under normal circumstances. Depending on the backend, stroke color may use a *stack*, in which case it exhibits the same page breaking behavior as general color. However, dvips/dvisvgm do not support this, and so color will need to be contained within a scope, such as `\draw_begin:/\draw_end:`.

---

`\color_fill:n` `\color_fill:n {<color expression>}`

`\color_stroke:n` Parses the `<color expression>` and then activates the resulting color specification for filling or stroking.

---

`\color_fill:nn` `\color_fill:nn {<model(s)>} {<value(s)>}`

`\color_stroke:nn` Activates the color specification equivalent to the `<model(s)>` and `<value(s)>` for filling or stroking.

---

`color.sc` When using dvips, this PostScript variables hold the stroke color.

### 35.6.1 Coloring math mode material

Coloring math mode material using `\color_select:nn(n)` has some restrictions and often leads to spacing issues and/or poor input syntax. Avoiding generating `\mathord` atoms whilst coloring only those parts of the input which are required needs careful handling. The functionality here covers this important use case.

---

`\color_math:nn` `\color_math:nn {<color expression>}{<content>}`

`\color_math:nnn` `\color_math:nnn {<model(s)>} {<value(s)>} {<content>}`

New: 2022-01-26 Works as for `\color_select:n(n)` but applies color only to the math mode `<content>`. The function does not generate a group and the `<content>` therefore retains its math atom states. Sub/superscripts are also properly handled.

---

`\l_color_math_active_t1` This list controls which tokens are considered as math active and should therefore be replaced by their definition during searching for sub/superscripts.  
New: 2022-01-26

## 35.7 Multiple color models

When selecting or setting a color with an explicit model, it is possible to give values for more than one model at one time. This is particularly useful where automated conversion between models does not give the desired outcome. To do this, the list of models and list of values are both subdivided using / characters (as for the similar function in xcolor). For example, to save a color with explicit cmyk and rgb values, one could use

```
\color_set:nnn { foo } { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

The manually-specified conversion will be used in preference to automated calculation whenever the model(s) listed are used: both in expressions and when a fixed model is active.

Similarly, the same syntax can be applied to directly selecting a color.

```
\color_select:nn { cmyk / rgb }
{ 0.1 , 0.2 , 0.3 , 0.4 / 0.1, 0.2 , 0.3 }
```

Again, this list is used when a fixed model is active: the first entry is used unless there is a fixed model matching one of the other entries.

## 35.8 Exporting color specifications

The major use of color expressions is in setting typesetting output, but there are other places in which some form of color information is required. These may need data in a different format or using a different model to the internal representation. Thus a set of functions are available to export colors in different formats.

Valid export targets are

- **backend** Two brace groups: the first containing the model, the second containing space-separated values appropriate for the model; this is the format required by backend functions of `expl3`
- **comma-sep-cmyk** Comma-separated cyan-magenta-yellow-black values
- **comma-sep-rgb** Comma-separated red-green-blue values suitable for use as a PDF annotation color
- **HTML** Uppercase two-digit hexadecimal values, expressing a red-green-blue color; the digits are *not* separated
- **space-sep-cmyk** Space-separated cyan-magenta-yellow-black values
- **space-sep-rgb** Space-separated red-green-blue values suitable for use as a PDF annotation color

---

```
\color_export:nnN \color_export:nnN {<color expression>} {<format>} {<t1>}
```

Parses the *<color expression>* as described earlier, then converts to the *<format>* specified and assigns the data to the *<t1>*.

---

```
\color_export:nnnN \color_export:nnnN {<model>} {<value(s)>} {<format>} {<t1>}
```

Expresses the combination of *<model>* and *<value(s)>* in an internal representation, then converts to the *<format>* specified and assigns the data to the *<t1>*.

## 35.9 Creating new color models

Additional color models are required to support specialist workflows, for example those involving separations (see <https://helpx.adobe.com/inDesign/using/spot-process-colors.html> for details of the use of separations in print). Color models may be split into families; for the standard device-based color models (DeviceCMYK, DeviceRGB, DeviceGray), these are synonymous. This is not generally the case: see the PDF reference for more details. (Note that l3color uses the shorter names cmyk, etc.)

---

```
\color_model_new:nnn \color_model_new:nnn {\model} {\family} {\params}
```

Creates a new *model* which is derived from the color model *family*. The latter should be one of

- **DeviceN**
- **ICCBased**
- **Separation**

(The *family* may be given in mixed case as-in the PDF reference: internally, case of these strings is folded.) Depending on the *family*, one or more *params* are mandatory or optional.

For a **Separation** space, there are three *compulsory* keys.

- **name** The name of the Separation, for example the formal name of a spot color ink. Such a *name* may contain spaces, etc., which are not permitted in the *model*.
- **alternative-model** An alternative device colorspace, one of **cmyk**, **rgb**, **gray** or **CIELAB**. The three parameter-based models work as described above; see below for details of CIELAB colors.
- **alternative-values** A comma-separated list of values appropriate to the **alternative-model**. This information is used by the PDF application if the **Separation** is not available.

CIELAB color separations are created using the **alternative-model = CIELAB** setting. These colors must also have an **illuminant** key, one of **a**, **c**, **e**, **d50**, **d55**, **d65** or **d75**. The **alternative-values** in this case are the three parameters *L\**, *a\** and *b\** of the CIELAB model. Full details of this device-independent color approach are given in the documentation to the **colorspace** package.

CIELAB colors *cannot* be converted into other device-dependent color spaces, and as such, mixing can only occur if colors set up using the CIELAB model are also given with an alternative parameter-based model. If that is not the case, l3color will fallback to using black as the colorant in any mixing.

For a **DeviceN** space, there is one *compulsory* key.

- **names** The names of the components of the **DeviceN** space. Each should be either the *name* of a **Separation** model, a process color name (**cyan**, etc.) or the special name **none**.

For a **ICCBased** space, there is one *compulsory* key.

- **file** The name of the file containing the profile.

### 35.9.1 Color profiles

Color profiles are used to ensure color accuracy by linking to collaboration. Applying a profile can be used to standardise color which is otherwise device-dependence.

---

```
\color_profile_apply:nn \color_profile_apply:nn {<profile>} {<model>}
```

---

New: 2021-02-23 This function applies a *<profile>* to one of the device *<models>*. The profile will then apply to all color of the selected *<model>*. The *<profile>* should specify an ICC profile file. The *<model>* has to be one the standard device models: `cmyk`, `gray` or `rgb`.

# Chapter 36

## The `l3pdf` package Core PDF support

### 36.1 Objects

---

`\pdf_object_new:n` `\pdf_object_new:n {⟨object⟩}`

New: 2022-08-23 Declares `⟨object⟩` as a PDF object. The object may be referenced from this point on, and written later using `\pdf_object_write:nnn`.

---

`\pdf_object_write:nnn` `\pdf_object_write:nn {⟨object⟩} {⟨type⟩} {⟨content⟩}`

`\pdf_object_write:nnx` New: 2022-08-23 Writes the `⟨content⟩` as content of the `⟨object⟩`. Depending on the `⟨type⟩` declared for the object, the format required for the `⟨data⟩` will vary

array A space-separated list of values

dict Key-value pairs in the form `/⟨key⟩ ⟨value⟩`

fstream Two brace groups: `⟨file name⟩` and `⟨file content⟩`

stream Two brace groups: `⟨attributes (dictionary)⟩` and `⟨stream contents⟩`

---

`\pdf_object_ref:n` \* `\pdf_object_ref:n {⟨object⟩}`

New: 2021-02-10 Inserts the appropriate information to reference the `⟨object⟩` in for example page resource allocation

---

```
\pdf_object_unnamed_write:nn \pdf_object_unnamed_write:nn {\<type>} {\<content>}  
\pdf_object_unnamed_write:nx
```

---

New: 2021-02-10

Writes the *<content>* as content of an anonymous object. Depending on the *<type>*, the format required for the *<data>* will vary

**array** A space-separated list of values

**dict** Key-value pairs in the form /*<key>* *<value>*

**fstream** Two brace groups: *<attributes (dictionary)>* and *<file name>*

**stream** Two brace groups: *<attributes (dictionary)>* and *<stream contents>*

---

```
\pdf_object_ref_last: * \pdf_object_ref_last:
```

---

New: 2021-02-10 Inserts the appropriate information to reference the last *<object>* created. This is particularly useful for anonymous objects.

---

```
\pdf_pageobject_ref:n * \pdf_pagobject_ref:n {\<pageobject>}
```

---

New: 2021-02-10 Inserts the appropriate information to reference the *<pageobject>*.

---

```
\pdf_object_if_exist_p:n * \pdf_object_if_exist_p:n {\<object>}  
\pdf_object_if_exist:nTF * \pdf_object_if_exist:nTF {\<object>}
```

---

New: 2020-05-15 Tests whether an object with name *{<object>}* has been defined.

## 36.2 Version

---

```
\pdf_version_compare_p:Nn * \pdf_version_compare:NnTF {\<comparator>} {\<version>} {\<true code>} {\<false code>}
```

---

New: 2021-02-10

Compares the version of the PDF being created with the *<version>* string specified, using the *<comparator>*. Either the *<true code>* or *<false code>* will be left in the output stream.

---

```
\pdf_version_gset:n \pdf_version_gset:n {\<version>}
```

---

**\pdf\_version\_min\_gset:n** Sets the *<version>* of the PDF being created. The **min** version will not alter the output version unless it is currently lower than the *<version>* requested.

This function may only be used up to the point where the PDF file is initialised. With dvips it sets **\pdf\_version\_major:** and **\pdf\_version\_minor:** and allows to compare the values with **\pdf\_version\_compare:Nn**, but the PDF version itself still has to be set with the command line option **-dCompatibilityLevel** of ps2pdf.

---

```
\pdf_version: * \pdf_version:
```

---

**\pdf\_version\_major:** \* Expands to the currently-active PDF version.

**\pdf\_version\_minor:** \*

---

New: 2021-02-10

### 36.3 Page (media) size

---

`\pdf_pagesize_gset:nn \pdf_pagesize_gset:nn {width} {height}`

New: 2023-01-14 Sets the page size (mediabox) of the PDF being created to the *width* and *height*, both of which are *dimexpr*.

### 36.4 Compression

---

`\pdf_uncompress: \pdf_uncompress:`

New: 2021-02-10 Disables any compression of the PDF, where possible.

This function may only be used up to the point where the PDF file is initialised.

### 36.5 Destinations

Destinations are the places a link jumped too. Unlike the name may suggest they don't described an exact location in the PDF. Instead a destination contains a reference to a page along with an instruction how to display this page. The normally used "XYZ *top left zoom*" for example instructs the viewer to show the page with the given *zoom* and the top left corner at the *top left* coordinates—which then gives the impression that there is an anchor at this position.

If an instruction takes a coordinate, it is calculated by the following commands relative to the location the command is issued. So to get a specific coordinate one has to move the command to the right place.

---

`\pdf_destination:nn \pdf_destination:nnn {<name>} {<type or integer>}`

New: 2021-01-03

This creates a destination.  $\{<\text{type or integer}>\}$  can be one of `fit`, `fith`, `fitv`, `fitb`, `fitbh`, `fitbv`, `fitr`, `xyz` or an integer representing a scale factor in percent. `fitr` here gives only a lightweight version of `/FitR`: The backend code defines `fitr` so that it will with `pdfLATEX` and `LuaLATEX` use the coordinates of the surrounding box, with `dvips` and `dvipdfmx` it falls back to `fit`. For full control use `\pdf_destination:nnnn`.

The keywords match to the PDF names as described in the following tabular.

Keyword	PDF	Remarks
<code>fit</code>	<code>/Fit</code>	Fits the page to the window
<code>fith</code>	<code>/FitH top</code>	Fits the width of the page to the window
<code>fitv</code>	<code>/FitV left</code>	Fits the height of the page to the window
<code>fitb</code>	<code>/FitB</code>	Fits the page bounding box to the window
<code>fitbh</code>	<code>/FitBH top</code>	Fits the width of the page bounding box to the window.
<code>fitbv</code>	<code>/FitBV left</code>	Fits the height of the page bounding box to the window.
<code>fitr</code>	<code>/FitR left bottom right top</code>	Fits the rectangle specified by the four coordinates to the window (see above for the restrictions)
<code>xyz</code>	<code>/XYZ left top null</code>	Sets a coordinate but doesn't change the zoom.
$\{<\text{integer}>\}$	<code>/XYZ left top zoom</code>	Sets a coordinate and a zoom meaning $\{<\text{integer}>\}\%$ .

---

`\pdf_destination:nnnn \pdf_destination:nnnn {<name>} {<width>} {<height>} {<depth>}`

New: 2021-01-17

This creates a destination with `/FitR` type with the given dimensions relative to the current location. The destination is in a box of size zero, but it doesn't switch to horizontal mode.

## 36.6 Deprecated functions

---

\pdf\_object\_new:nn \pdf\_object\_new:nn {\object} {\type}

New: 2021-02-10 Declares *⟨object⟩* as a PDF object of *⟨type⟩*, which should be one of

- **array**
- **dict**
- **fstream**
- **stream**

The object may be referenced from this point on, and written later using \pdf\_object\_write:nn.

Deprecated in favor of \pdf\_object\_new:n.

---

\pdf\_object\_write:nn \pdf\_object\_write:nn {\object} {\content}

\pdf\_object\_write:nx New: 2021-02-10 Writes the *⟨content⟩* as content of the *⟨object⟩*. Depending on the *⟨type⟩* declared for the object, the format required for the *⟨data⟩* will vary

**array** A space-separated list of values

**dict** Key–value pairs in the form /⟨key⟩ ⟨value⟩

**fstream** Two brace groups: ⟨file name⟩ and ⟨file content⟩

**stream** Two brace groups: ⟨attributes (dictionary)⟩ and ⟨stream contents⟩

Deprecated in favor of \pdf\_object\_write:nnn.

**Part VII**  
**Additions and removals**

# Chapter 37

## The **I3candidates** package Experimental additions to **I3kernel**

### 37.1 Important notice

This module provides a space in which functions can be added to **I3kernel** (`expl3`) while still being experimental.

**As such, the functions here may not remain in their current form, or indeed at all, in **I3kernel** in the future.**

In contrast to the material in **I3experimental**, the functions here are all *small* additions to the kernel. We encourage programmers to test them out and report back on the **LaTeX-L** mailing list.

Thus, if you intend to use any of these functions from the candidate module in a public package offered to others for productive use (e.g., being placed on CTAN) please consider the following points carefully:

- Be prepared that your public packages might require updating when such functions are being finalized.
- Consider informing us that you use a particular function in your public package, e.g., by discussing this on the **LaTeX-L** mailing list. This way it becomes easier to coordinate any updates necessary without issues for the users of your package.
- Discussing and understanding use cases for a particular addition or concept also helps to ensure that we provide the right interfaces in the final version so please give us feedback if you consider a certain candidate function useful (or not).

We only add functions in this space if we consider them being serious candidates for a final inclusion into the kernel. However, real use sometimes leads to better ideas, so functions from this module are **not necessarily stable** and we may have to adjust them!

## 37.2 Additions to `I3box`

---

```
\box_clip:N          \box_clip:N <box>
\box_clip:c          \box_gclip:N
\box_gclip:c
```

---

Updated: 2019-01-23

These functions require the `LATEX3` native drivers: they do not work with the `LATEX2ε` graphics drivers!

**TeXhackers note:** Clipping is implemented by the driver, and as such the full content of the box is placed in the output file. Thus clipping does not remove any information from the raw output, and hidden material can therefore be viewed by direct examination of the file.

---

```
\box_set_trim:Nnnnn \box_set_trim:Nnnnn <box> {<left>} {<bottom>} {<right>} {<top>}
```

---

Adjusts the bounding box of the `<box>` `<left>` is removed from the left-hand edge of the bounding box, `<right>` from the right-hand edge and so fourth. All adjustments are *dimension expressions*. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated `<box>` is an hbox, irrespective of the nature of the `<box>` before the trim operation is applied. The behavior of the operation where the trims requested is greater than the size of the box is undefined.

New: 2019-01-23

---

```
\box_set_viewport:Nnnnn \box_set_viewport:Nnnnn <box> {<llx>} {<llx>} {<urx>} {<ury>}
```

---

Adjusts the bounding box of the `<box>` such that it has lower-left co-ordinates (`<llx>`, `<llx>`) and upper-right co-ordinates (`<urx>`, `<ury>`). All four co-ordinate positions are *dimension expressions*. Material outside of the bounding box is still displayed in the output unless `\box_clip:N` is subsequently applied. The updated `<box>` is an hbox, irrespective of the nature of the `<box>` before the viewport operation is applied.

New: 2019-01-23

## 37.3 Additions to `I3expan`

---

```
\exp_args_generate:n \exp_args_generate:n {<variant argument specifiers>}
```

---

New: 2018-04-04

Updated: 2019-02-08

Defines `\exp_args:N`(`variant`) functions for each `<variant>` given in the comma list `{<variant argument specifiers>}`. Each `<variant>` should consist of the letters `N`, `c`, `n`, `V`, `v`, `o`, `f`, `e`, `x`, `p` and the resulting function is protected if the letter `x` appears in the `<variant>`. This is only useful for cases where `\cs_generate_variant:Nn` is not applicable.

## 37.4 Additions to `I3fp`

---

```
\fp_if_nan_p:n * \fp_if_nan:n {<fpexpr>}
```

---

`\fp_if_nan:nTF *` Evaluates the `<fpexpr>` and tests whether the result is exactly `nan`. The test returns `false` for any other result, even a tuple containing `nan`.

## 37.5 Additions to **I3file**

---

`\iow_allow_break: \iow_allow_break:`

New: 2018-12-29

In the first argument of `\iow_wrap:nnnN` (for instance in messages), inserts a break-point that allows a line break. In other words this is a zero-width breaking space.

---

`\ior_get_term:nN \ior_get_term:nN <prompt> <token list variable>`

`\ior_str_get_term:nN`

New: 2019-03-23

Function that reads one or more lines (until an equal number of left and right braces are found) from the terminal and stores the result locally in the `<token list>` variable. Tokenization occurs as described for `\ior_get:NN` or `\ior_str_get:NN`, respectively. When the `<prompt>` is empty, TeX will wait for input without any other indication: typically the programmer will have provided a suitable text using e.g. `\iow_term:n`. Where the `<prompt>` is given, it will appear in the terminal followed by an =, e.g.

`prompt=`

---

`\ior_shell_open:Nn \ior_shell_open:Nn <stream> {<shell command>}`

New: 2019-05-08

Opens the *pseudo-file* created by the output of the `<shell command>` for reading using `<stream>` as the control sequence for access. If the `<stream>` was already open it is closed before the new operation begins. The `<stream>` is available for access immediately and will remain allocated to `<shell command>` until a `\ior_close:N` instruction is given or the TeX run ends. If piped system calls are disabled an error is raised.

For details of handling of the `<shell command>`, see `\sys_get_shell:nnNTF`.

## 37.6 Additions to **I3flag**

---

`\flag_raise_if_clear:n * \flag_raise_if_clear:n {<flag name>}`

New: 2018-04-02

Ensures the `<flag>` is raised by making its height at least 1, locally.

## 37.7 Additions to **I3intarray**

---

`\intarray_gset_rand:Nnn \intarray_gset_rand:Nnn <intarray var> {<minimum>} {<maximum>}`

`\intarray_gset_rand:cnn \intarray_gset_rand:Nn <intarray var> {<maximum>}`

`\intarray_gset_rand:Nn`

`\intarray_gset_rand:cn`

New: 2018-05-05

Evaluates the integer expressions `<minimum>` and `<maximum>` then sets each entry (independently) of the `<integer array variable>` to a pseudo-random number between the two (with bounds included). If the absolute value of either bound is bigger than  $2^{30} - 1$ , an error occurs. Entries are generated in the same way as repeated calls to `\int_rand:nn` or `\int_rand:n` respectively, in particular for the second function the `<minimum>` is 1. Assignments are always global. This is not available in older versions of X<sub>Ǝ</sub>T<sub>E</sub>X.

---

`\intarray_to_clist:N ☆ \intarray_to_clist:N <intarray var>`

New: 2018-05-04

Converts the `<intarray>` to integer denotations separated by commas. All tokens have category code other. If the `<intarray>` has no entry the result is empty; otherwise the result has one fewer comma than the number of items.

## 37.8 Additions to **I3msg**

---

\msg\_show\_eval:Nn \msg\_show\_eval:Nn *function* {\iexpression}

---

\msg\_log\_eval:Nn

New: 2017-12-04

Shows or logs the *expression* (turned into a string), an equal sign, and the result of applying the *function* to the {\iexpression} (with f-expansion). For instance, if the *function* is \int\_eval:n and the *expression* is 1+2 then this logs > 1+2=3.

---

\msg\_show\_item:n \* \seq\_map\_function:NN *seq* \msg\_show\_item:n

\msg\_show\_item\_unbraced:n \* \prop\_map\_function:NN *prop* \msg\_show\_item:nn

\msg\_show\_item:nn \*

\msg\_show\_item\_unbraced:nn \*

New: 2017-12-04

---

Used in the text of messages for \msg\_show:nnxxxx to show or log a list of items or key-value pairs. The one-argument functions are used for sequences, clist or token lists and the others for property lists. These functions turn their arguments to strings.

## 37.9 Additions to **I3prg**

---

\bool\_set\_inverse:N \bool\_set\_inverse:N *boolean*

\bool\_set\_inverse:c

\bool\_gset\_inverse:N Toggles the *boolean* from true to false and conversely: sets it to the inverse of its current value.

\bool\_gset\_inverse:c

New: 2018-05-10

---

---

```
\bool_case_true:n    * \bool_case_true:nTF
\bool_case_true:nTF *   {
\bool_case_false:n  *     {⟨boolexpr case1⟩} {⟨code case1⟩}
\bool_case_false:nTF *     {⟨boolexpr case2⟩} {⟨code case2⟩}
                           ...
                           {⟨boolexpr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}
```

New: 2019-02-10

---

Evaluates in turn each of the *⟨boolean expression cases⟩* until the first one that evaluates to *true* or to *false*, for *\bool\_case\_true:n* and *\bool\_case\_false:n*, respectively. The *⟨code⟩* associated to this first case is left in the input stream, followed by the *⟨true code⟩*, and other cases are discarded. If none of the cases match then only the *⟨false code⟩* is inserted. The functions *\bool\_case\_true:n* and *\bool\_case\_false:n*, which do nothing if there is no match, are also available. For example

```
\bool_case_true:nF
{
  { \dim_compare_p:n { \l__mypkg_wd_dim <= 10pt } }
    { Fits }
  { \int_compare_p:n { \l__mypkg_total_int >= 10 } }
    { Many }
  { \l__mypkg_special_bool }
    { Special }
}
{ No idea! }
```

leaves “Fits” or “Many” or “Special” or “No idea!” in the input stream, in a way similar to some other language’s “if ... elseif ... elseif ... else ...”.

## 37.10 Additions to `I3prop`

---

```
\prop_rand_key_value:N * \prop_rand_key_value:N ⟨property list⟩
```

---

```
\prop_rand_key_value:c * Selects a pseudo-random key–value pair from the ⟨property list⟩ and returns {⟨key⟩} and {⟨value⟩}. If the ⟨property list⟩ is empty the result is empty. This is not available in older versions of XHTEX.
```

New: 2016-12-06

---

**TeXhackers note:** The result is returned within the *\unexpanded* primitive (*\exp\_not:n*), which means that the *⟨value⟩* does not expand further when appearing in an *x*-type or *e*-type argument expansion.

## 37.11 Additions to `\3seq`

---

`\seq_mapthread_function:NNN` ☆ `\seq_mapthread_function:NNN <seq1> <seq2> <function>`  
`\seq_mapthread_function:(NcN|cNN|ccN)` ☆

---

Applies `<function>` to every pair of items `<seq1-item>-<seq2-item>` from the two sequences, returning items from both sequences from left to right. The `<function>` receives two n-type arguments for each iteration. The mapping terminates when the end of either sequence is reached (*i.e.* whichever sequence has fewer items determines how many iterations occur).

---

`\seq_set_filter:NNn` `\seq_set_filter:NNn <sequence1> <sequence2> {<inline boolexpr>}`

---

`\seq_gset_filter:NNn` Evaluates the `<inline boolexpr>` for every `<item>` stored within the `<sequence2>`. The `<inline boolexpr>` receives the `<item>` as #1. The sequence of all `<items>` for which the `<inline boolexpr>` evaluated to `true` is assigned to `<sequence1>`.

**TEXhackers note:** Contrarily to other mapping functions, `\seq_map_break:` cannot be used in this function, and would lead to low-level TeX errors.

---

`\seq_set_from_function:NnN` `\seq_set_from_function:NnN <seq var> {<loop code>} <function>`  
`\seq_gset_from_function:NnN`

---

New: 2018-04-06

Sets the `<seq var>` equal to a sequence whose items are obtained by x-expanding `<loop code>` `<function>`. This expansion must result in successive calls to the `<function>` with no nonexpandable tokens in between. More precisely the `<function>` is replaced by a wrapper function that inserts the appropriate separators between items in the sequence. The `<loop code>` must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`.

---

`\seq_set_from_inline_x:Nnn` `\seq_set_from_inline_x:Nnn <seq var> {<loop code>} {<inline code>}`  
`\seq_gset_from_inline_x:Nnn`

---

New: 2018-04-06

Sets the `<seq var>` equal to a sequence whose items are obtained by x-expanding `<loop code>` applied to a `<function>` derived from the `<inline code>`. A `<function>` is defined, that takes one argument, x-expands the `<inline code>` with that argument as #1, then adds appropriate separators to turn the result into an item of the sequence. The x-expansion of `<loop code>` `<function>` must result in successive calls to the `<function>` with no nonexpandable tokens in between. The `<loop code>` must be expandable; it can be for example `\tl_map_function:NN <tl var>` or `\clist_map_function:nN {<clist>}` or `\int_step_function:nnnN {<initial value>} {<step>} {<final value>}`, but not the analogous “inline” mappings.

---

<code>\seq_set_item:Nnn</code> <code>\seq_set_item:cnn</code> <code>\seq_set_item:NnnTF</code> <code>\seq_set_item:cnnTF</code> <code>\seq_gset_item:Nnn</code> <code>\seq_gset_item:cnn</code> <code>\seq_gset_item:NnnTF</code> <code>\seq_gset_item:cnnTF</code>	<code>\seq_set_item:Nnn &lt;seq var&gt; {&lt;integer&gt;} {&lt;item&gt;}</code> <code>\seq_set_item:NnnTF &lt;seq var&gt; {&lt;integer&gt;} {&lt;item&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
--	---

Removes the item of *<sequence>* at the position given by evaluating the *<integer expression>* and replaces it by *<item>*. Items are indexed from 1 on the left/top of the *<sequence>*, or from -1 on the right/bottom. If the *<integer expression>* is zero or is larger (in absolute value) than the number of items in the sequence, the *<sequence>* is not modified. In these cases, `\seq_set_item:Nnn` raises an error while `\seq_set_item:NnnTF` runs the *<false code>*. In cases where the assignment was successful, *<true code>* is run afterwards.

New: 2021-04-29

---



---

<code>\seq_pop_item:NnN</code> <code>\seq_pop_item:cnN</code> <code>\seq_pop_item:NnNTF</code> <code>\seq_pop_item:cnnTF</code> <code>\seq_gpop_item:Nnn</code> <code>\seq_gpop_item:cnn</code> <code>\seq_gpop_item:NnNTF</code> <code>\seq_gpop_item:cnnTF</code>	<code>\seq_pop_item:NnN &lt;seq var&gt; {&lt;integer&gt;} {&lt;t1 var&gt;}</code> <code>\seq_pop_item:NnNTF &lt;seq var&gt; {&lt;integer&gt;} {&lt;t1 var&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
--	---

Removes the *<item>* at position *<integer expression>* in the *<sequence>*, and places it in the *<token list variable>*. Items are indexed from 1 on the left/top of the *<sequence>*, or from -1 on the right/bottom. If the position is zero or is larger (in absolute value) than the number of items in the sequence, the *<seq var>* is not modified, the *<token list>* is set to the special marker `\q_no_value`, and the *<false code>* is left in the input stream; otherwise the *<true code>* is. The *<token list>* assignment is local while the *<sequence>* is assigned locally for pop or globally for gpop functions.

New: 2021-04-28

---

## 37.12 Additions to l3sys

---

<code>\c_sys_engine_version_str</code>	The version string of the current engine, in the same form as given in the banner issued when running a job. For pdfTeX and LuaTeX this is of the form
--	--

New: 2018-05-02

---

*(major).(minor).(revision)*

For XeTeX, the form is

*(major).(minor)*

For pTeX and upTeX, only releases since TeX Live 2018 make the data available, and the form is more complex, as it comprises the pTeX version, the upTeX version and the e-pTeX version.

*p(major).(minor).(revision)-u(major).(minor)-(epTeX)*

where the u part is only present for upTeX.

---

<code>\sys_if_rand_exist_p: *</code> <code>\sys_if_rand_exist:TF *</code>	<code>\sys_if_rand_exist_p:</code>
--	------------------------------------

---

New: 2017-05-27 Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX, LuaTeX, pTeX, upTeX and recent releases of XeTeX.

---

### 37.13 Additions to l3tl

---

```
\tl_range_braced:Nnn * \tl_range_braced:Nnn <tl var> {\langle start index\rangle} {\langle end index\rangle}
\tl_range_braced:cnn * \tl_range_braced:nnn {\langle token list\rangle} {\langle start index\rangle} {\langle end index\rangle}
\tl_range_braced:nnn * \tl_range_unbraced:Nnn <tl var> {\langle start index\rangle} {\langle end index\rangle}
\tl_range_unbraced:Nnn * \tl_range_unbraced:nnn {\langle token list\rangle} {\langle start index\rangle} {\langle end index\rangle}
\tl_range_unbraced:cnn *
\tl_range_unbraced:nnn *
```

---

New: 2017-07-15

Leaves in the input stream the items from the  $\langle start index \rangle$  to the  $\langle end index \rangle$  inclusive, using the same indexing as `\tl_range:nnn`. Spaces are ignored. Regardless of whether items appear with or without braces in the  $\langle token list \rangle$ , the `\tl_range_braced:nnn` function wraps each item in braces, while `\tl_range_unbraced:nnn` does not (overall it removes an outer set of braces). For instance,

```
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range_braced:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `{b}{c}{d}{e{}}`, `{c}{d}{e{}}{f}`, `{e{}}{f}`, and an empty line to the terminal, while

```
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { 2 } { 5 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { -4 } { -1 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { -2 } { -1 } }
\iow_term:x { \tl_range_unbraced:nnn { abcd~{e{}}f } { 0 } { -1 } }
```

prints `bcd{}`, `cde{}`, `e{}`, and an empty line to the terminal. Because braces are removed, the result of `\tl_range_unbraced:nnn` may have a different number of items as for `\tl_range:nnn` or `\tl_range_braced:nnn`. In cases where preserving spaces is important, consider the slower function `\tl_range:nnn`.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the  $\langle item \rangle$  does not expand further when appearing in an `x`-type or `e`-type argument expansion.

---

```
\tl_build_begin:N \tl_build_begin:N <tl var>
\tl_build_gbegin:N
```

---

New: 2018-04-01

Clears the  $\langle tl var \rangle$  and sets it up to support other `\tl_build_...` functions, which allow accumulating large numbers of tokens piece by piece much more efficiently than standard l3tl functions. Until `\tl_build_end:N <tl var>` is called, applying any function from l3tl other than `\tl_build_...` will lead to incorrect results. The `begin` and `gbegin` functions must be used for local and global  $\langle tl var \rangle$  respectively.

---

```
\tl_build_clear:N \tl_build_clear:N <tl var>
\tl_build_gclear:N
```

---

New: 2018-04-01

Clears the  $\langle tl var \rangle$  and sets it up to support other `\tl_build_...` functions. The `clear` and `gclear` functions must be used for local and global  $\langle tl var \rangle$  respectively.

---

```
\tl_build_put_left:Nn          \tl_build_put_left:Nn <tl var> {<tokens>}
\tl_build_put_left:Nx          \tl_build_put_right:Nn <tl var> {<tokens>}
```

Adds *<tokens>* to the left or right side of the current contents of *<tl var>*. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `put` and `gput` functions must be used for local and global *<tl var>* respectively. The `right` functions are about twice faster than the `left` functions.

---

New: 2018-04-01

---

```
\tl_build_get:NN \tl_build_get:NN <tl var1> <tl var2>
```

New: 2018-04-01

Stores the contents of the *<tl var1>* in the *<tl var2>*. The *<tl var1>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The *<tl var2>* is a “normal” token list variable, assigned locally using `\tl_set:Nn`.

---

```
\tl_build_end:N   \tl_build_end:N <tl var>
```

```
\tl_build_gend:N
```

New: 2018-04-01

Gets the contents of *<tl var>* and stores that into the *<tl var>* using `\tl_set:Nn` or `\tl_gset:Nn`. The *<tl var>* must have been set up with `\tl_build_begin:N` or `\tl_build_gbegin:N`. The `end` and `gend` functions must be used for local and global *<tl var>* respectively. These functions completely remove the setup code that enabled *<tl var>* to be used for other `\tl_build_...` functions.

## 37.14 Additions to `I3token`

---

```
\c_catcode_active_space_tl
```

Token list containing one character with category code 13, (“active”), and character code 32 (space).

New: 2017-08-07

---

```
\peek_catcode_collect_inline:Nn \peek_catcode_collect_inline:Nn <test token> {{inline code}}
\peek_charcode_collect_inline:Nn \peek_charcode_collect_inline:Nn <test token> {{inline code}}
\peek_meaning_collect_inline:Nn \peek_meaning_collect_inline:Nn <test token> {{inline code}}
```

---

New: 2018-09-23

Collects and removes tokens from the input stream until finding a token that does not match the *<test token>* (as defined by the test `\token_if_eq_catcode:NNTF` or `\token_if_eq_charcode:NNTF` or `\token_if_eq_meaning:NNTF`). The collected tokens are passed to the *<inline code>* as #1. When begin-group or end-group tokens (usually { or }) are collected they are replaced by implicit `\c_group_begin_token` and `\c_group_end_token`, and when spaces (including `\c_space_token`) are collected they are replaced by explicit spaces.

For example the following code prints “Hello” to the terminal and leave “, world!” in the input stream.

```
\peek_catcode_collect_inline:Nn A { \iow_term:n {#1} } Hello,~world!
```

Another example is that the following code tests if the next token is \*, ignoring intervening spaces, but putting them back using #1 if there is no \*.

```
\peek_meaning_collect_inline:Nn \c_space_token
{ \peek_charcode:NTF * { star } { no~star #1 } }
```

## **Part VIII**

## **Implementation**

# Chapter 38

## I3bootstrap implementation

```
1 〈*package〉  
2 〈@@=kernel〉
```

### 38.1 LuATEX-specific code

Depending on the versions available, the LATEX format may not have the raw \Umath primitive names available. We fix that globally: it should cause no issues. Older LuATEX versions do not have a pre-built table of the primitive names here so sort one out ourselves. These end up globally-defined but at that is true with a newer format anyway and as they all start \U this should be reasonably safe.

```
3  \begingroup  
4    \expandafter\ifx\csname directlua\endcsname\relax  
5    \else  
6      \directlua{  
7        local i  
8        local t = {}  
9        for _,i in pairs(tex.extraprimitives("luatex")) do  
10          if string.match(i,"^U") then  
11            if not string.match(i,"^Uchar$") then %$  
12              table.insert(t,i)  
13            end  
14          end  
15        end  
16        tex.enableprimitives("", t)  
17      }%  
18    \fi  
19  \endgroup
```

### 38.2 The \pdfstrcmp primitive in XETEX

Only pdfTEX has a primitive called \pdfstrcmp. The XETEX version is just \strcmp, so there is some shuffling to do. As this is still a real primitive, using the pdfTEX name is “safe”.

```
20  \begingroup\expandafter\expandafter\expandafter\endgroup  
21    \expandafter\ifx\csname pdfstrcmp\endcsname\relax  
22      \let\pdfstrcmp\strcmp
```

```
23 \fi
```

### 38.3 Loading support Lua code

When  $\text{\LaTeX}$  is used there are various pieces of Lua code which need to be loaded. The code itself is defined in `l3luatex` and is extracted into a separate file. Thus here the task is to load the Lua code both now and (if required) at the start of each job.

```
24 \begingroup\expandafter\expandafter\expandafter\endgroup
25 \expandafter\ifx\csname directlua\endcsname\relax
26 \else
27 \ifnum\luatexversion<110 %
28 \else
```

For  $\text{\LaTeX}$  we make sure the basic support is loaded: this is only necessary in plain.

```
29 \begingroup\expandafter\expandafter\expandafter\endgroup
30 \expandafter\ifx\csname newcatcodetable\endcsname\relax
31 \input{ltluatex}%
32 \fi
33 \begingroup\expandafter\expandafter\expandafter\endgroup
34 \expandafter\ifx\csname newluabytecode\endcsname\relax
35 \else
36 \newluabytecode\@expl@luadata@bytecode
37 \fi
38 \directlua{require("expl3")}%
```

As the user might be making a custom format, no assumption is made about matching package mode with only loading the Lua code once. Instead, a query to Lua reveals what mode is in operation.

```
39 \ifnum 0%
40   \directlua{
41     if status.ini_version then
42       tex.write("1")
43     end
44   }>0 %
45   \everyjob\expandafter{%
46     \the\expandafter\everyjob
47     \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
48   }%
49   \fi
50   \fi
51 \fi
```

### 38.4 Engine requirements

The code currently requires  $\varepsilon\text{-}\text{\TeX}$  and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

For  $\text{\LaTeX}$ , we require at least Lua 5.3 and the `token.set_lua` function. This is available at least since  $\text{\LaTeX}$  1.10.

```
52 \begingroup
53   \def\next{\endgroup}%
54   \def\ShortText{Required primitives not found}%
```

```

55  \def\LongText%
56  {%
57      The L3 programming layer requires the e-TeX primitives and additional
58      \LineBreak functionality as described in the README file.
59      \LineBreak
60      These are available in the engines\LineBreak
61      - pdfTeX v1.40.0\LineBreak
62      - XeTeX v0.99992\LineBreak
63      - LuaTeX v1.10\LineBreak
64      - e-(u)pTeX mid-2012\LineBreak
65      - Prote (2021)\LineBreak
66      or later.\LineBreak
67      \LineBreak
68  }%
69 \ifnum0%
70     \expandafter\ifx\csname expanded\endcsname\relax
71         \expandafter\ifx\csname pdfstrcmp\endcsname\relax\else 1\fi
72     \else
73         \expandafter\ifx\csname luatexversion\endcsname\relax
74             1%
75         \else
76             \ifnum\luatexversion<110 \else 1\fi
77         \fi
78     \fi
79 =0 %
80     \newlinechar`\^^J %
81     \def\LineBreak{\noexpand\MessageBreak}%
82     \expandafter\ifx\csname PackageError\endcsname\relax
83         \def\LineBreak{`^^J}%
84         \begingroup
85             \lccode`\~=`\ \lccode`\}=`\ %
86             \lccode`\T=`\T\lccode`\H=`\H%
87             \catcode`\ =11 %
88 \lowercase{\endgroup\def\PackageError#1#2#3{%
89 \begingroup\errorcontextlines-1\immediate\write0{}{\errhelp{#3}\def%
90 \#1 Error: #2.^^J^^J
91 Type H <return> for immediate help}\def~{\errmessage{%
92 \}}\endgroup}%
93     \fi
94     \edef\next
95     {%
96         \noexpand\PackageError{expl3}{\ShortText}
97             {\LongText Loading of expl3 will abort!}%
98         \endgroup
99         \noexpand\endinput
100    }%
101   \fi
102 \next

```

## 38.5 Extending allocators

The ability to extend TeX's allocation routine to allow for  $\varepsilon$ -TeX has been around since 1997 in the `etex` package. Loading this support is delayed until here as we are now sure

that the  $\varepsilon$ - $\text{\TeX}$  extensions and  $\text{\pdfstrcmp}$  or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For  $\text{\LaTeX}2\varepsilon$  we need to make sure that the extended pool is being used:  $\text{expl3}$  uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the  $\text{etex}$  package needs to be loaded to do the job. In that case, some inserts are reserved also as these have to be from the standard pool. Note that  $\text{\reserveinserts}$  is  $\text{\outer}$  and so is accessed here by  $\text{csname}$ . In earlier versions, loading  $\text{etex}$  was done directly and so  $\text{\reserveinserts}$  appeared in the code: this then required a  $\text{\relax}$  after  $\text{\RequirePackage}$  to prevent an error with “unsafe” definitions as seen for example with  $\text{\capoptions}$ . The optional loading here is done using a group and  $\text{\ifx}$  test as we are not quite in the position to have a single name for  $\text{\pdfstrcmp}$  just yet.

```

103 \begingroup
104   \def\@tempa{LaTeX2e}%
105   \def\next{}%
106   \ifx\fmtname\@tempa
107     \expandafter\ifx\csname extrafloats\endcsname\relax
108       \def\next
109         {%
110           \RequirePackage{etex}%
111           \csname reserveinserts\endcsname{32}%
112         }%
113       \fi
114     \fi
115   \expandafter\endgroup
116   \next

```

## 38.6 The $\text{\LaTeX}3$ code environment

The code environment is now set up.

**\ExplSyntaxOff** Before changing any category codes, in package mode we need to save the situation before loading. Note the set up here means that once applied  $\text{\ExplSyntaxOff}$  becomes a “do nothing” command until  $\text{\ExplSyntaxOn}$  is used.

```

117 \protected\edef\ExplSyntaxOff
118   {%
119     \protected\def\noexpand\ExplSyntaxOff{}%
120     \catcode 9 = \the\catcode 9\relax
121     \catcode 32 = \the\catcode 32\relax
122     \catcode 34 = \the\catcode 34\relax
123     \catcode 38 = \the\catcode 38\relax
124     \catcode 58 = \the\catcode 58\relax
125     \catcode 94 = \the\catcode 94\relax
126     \catcode 95 = \the\catcode 95\relax
127     \catcode 124 = \the\catcode 124\relax
128     \catcode 126 = \the\catcode 126\relax
129     \endlinechar = \the\endlinechar\relax
130     \chardef\csname\detokenize{l_kernel_expl_bool}\endcsname = 0\relax
131   }%

```

(End definition for  $\text{\ExplSyntaxOff}$ . This function is documented on page 9.)

The code environment is now set up.

```

132 \catcode 9    = 9\relax
133 \catcode 32   = 9\relax
134 \catcode 34   = 12\relax
135 \catcode 38   = 4\relax
136 \catcode 58   = 11\relax
137 \catcode 94   = 7\relax
138 \catcode 95   = 11\relax
139 \catcode 124  = 12\relax
140 \catcode 126  = 10\relax
141 \endlinechar = 32\relax

```

\l\_kernel\_expl\_bool The status for code syntax: this is on at present.

```
142 \chardef\l_kernel_expl_bool = 1\relax
```

*(End definition for \l\_kernel\_expl\_bool.)*

\ExplSyntaxOn The idea here is that multiple \ExplSyntaxOn calls are not going to mess up category codes, and that multiple calls to \ExplSyntaxOff are also not wasting time. Applying \ExplSyntaxOn alters the definition of \ExplSyntaxOff and so in package mode this function should not be used until after the end of the loading process!

```

143 \protected \def \ExplSyntaxOn
144   {
145     \bool_if:NF \l_kernel_expl_bool
146     {
147       \cs_set_protected:Npx \ExplSyntaxOff
148       {
149         \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
150         \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
151         \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
152         \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
153         \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
154         \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
155         \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
156         \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
157         \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
158         \tex_endlinechar:D =
159           \tex_the:D \tex_endlinechar:D \scan_stop:
160         \bool_set_false:N \l_kernel_expl_bool
161         \cs_set_protected:Npn \ExplSyntaxOff { }
162       }
163     }
164     \char_set_catcode_ignore:n      { 9 } % tab
165     \char_set_catcode_ignore:n      { 32 } % space
166     \char_set_catcode_other:n      { 34 } % double quote
167     \char_set_catcode_alignment:n  { 38 } % ampersand
168     \char_set_catcode_letter:n    { 58 } % colon
169     \char_set_catcode_math_superscript:n { 94 } % circumflex
170     \char_set_catcode_letter:n    { 95 } % underscore
171     \char_set_catcode_other:n      { 124 } % pipe
172     \char_set_catcode_space:n     { 126 } % tilde
173     \tex_endlinechar:D = 32 \scan_stop:
174     \bool_set_true:N \l_kernel_expl_bool
175   }

```

(End definition for \ExplSyntaxOn. This function is documented on page 9.)

<sup>176</sup> ⟨/package⟩

# Chapter 39

## I3names implementation

177 `(*package & tex)`

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `0000`.

178 `(@@=kernel)`

The code here simply renames all of the primitives to new, internal, names.

The `\let` primitive is renamed by hand first as it is essential for the entire process to follow. This also uses `\global`, as that way we avoid leaving an unneeded csname in the hash table.

179 `\let \tex_global:D \global`  
180 `\let \tex_let:D \let`

Everything is inside a (rather long) group, which keeps `\_kernel_primitive:NN` trapped.

181 `\begingroup`

`\_kernel_primitive:NN` A temporary function to actually do the renaming.

182 `\long \def \_kernel_primitive:NN #1#2`  
183 `{ \tex_global:D \tex_let:D #2 #1 }`

*(End definition for `\_kernel_primitive:NN`.)*

To allow extracting “just the names”, a bit of DocStrip fiddling.

184 `(/package & tex)`  
185 `(*names | tex)`  
186 `(*names | package)`

In the current incarnation of this package, all TeX primitives are given a new name of the form `\tex_oldname:D`. But first three special cases which have symbolic original names. These are given modified new names, so that they may be entered without catcode tricks.

187 `\_kernel_primitive:NN \` `\tex_space:D`  
188 `\_kernel_primitive:NN \/` `\tex_italiccorrection:D`  
189 `\_kernel_primitive:NN \-` `\tex_hyphen:D`

Now all the other primitives.

190 `\_kernel_primitive:NN \above` `\tex_above:D`  
191 `\_kernel_primitive:NN \abovedisplayshortskip` `\tex_abovedisplayshortskip:D`  
192 `\_kernel_primitive:NN \abovedisplayskip` `\tex_abovedisplayskip:D`  
193 `\_kernel_primitive:NN \abovewithdelims` `\tex_abovewithdelims:D`

```

194  \__kernel_primitive:NN \accent          \tex_accent:D
195  \__kernel_primitive:NN \adjdemerits    \tex_adjdemerits:D
196  \__kernel_primitive:NN \advance         \tex_advance:D
197  \__kernel_primitive:NN \afterassignment   \tex_afterassignment:D
198  \__kernel_primitive:NN \aftergroup      \tex_aftergroup:D
199  \__kernel_primitive:NN \atop           \tex_atop:D
200  \__kernel_primitive:NN \atopwithdelims \tex_atopwithdelims:D
201  \__kernel_primitive:NN \badness        \tex_badness:D
202  \__kernel_primitive:NN \baselineskip    \tex_baselineskip:D
203  \__kernel_primitive:NN \batchmode       \tex_batchmode:D
204  \__kernel_primitive:NN \begingroup      \tex_begingroup:D
205  \__kernel_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
206  \__kernel_primitive:NN \belowdisplayskip  \tex_belowdisplayskip:D
207  \__kernel_primitive:NN \binoppenalty    \tex_binoppenalty:D
208  \__kernel_primitive:NN \botmark         \tex_botmark:D
209  \__kernel_primitive:NN \box             \tex_box:D
210  \__kernel_primitive:NN \boxmaxdepth     \tex_boxmaxdepth:D
211  \__kernel_primitive:NN \brokenpenalty   \tex_brokenpenalty:D
212  \__kernel_primitive:NN \catcode         \tex_catcode:D
213  \__kernel_primitive:NN \char            \tex_char:D
214  \__kernel_primitive:NN \chardef         \tex_chardef:D
215  \__kernel_primitive:NN \cleaders        \tex_cleaders:D
216  \__kernel_primitive:NN \closein         \tex_closein:D
217  \__kernel_primitive:NN \closeout        \tex_closeout:D
218  \__kernel_primitive:NN \clubpenalty     \tex_clubpenalty:D
219  \__kernel_primitive:NN \copy            \tex_copy:D
220  \__kernel_primitive:NN \count           \tex_count:D
221  \__kernel_primitive:NN \countdef        \tex_countdef:D
222  \__kernel_primitive:NN \cr              \tex_cr:D
223  \__kernel_primitive:NN \crrc            \tex_crrc:D
224  \__kernel_primitive:NN \csname         \tex_csname:D
225  \__kernel_primitive:NN \day             \tex_day:D
226  \__kernel_primitive:NN \deadcycles     \tex_deadcycles:D
227  \__kernel_primitive:NN \def             \tex_def:D
228  \__kernel_primitive:NN \defaulthyphenchar \tex_defaulthyphenchar:D
229  \__kernel_primitive:NN \defaultskewchar \tex_defaultskewchar:D
230  \__kernel_primitive:NN \delcode         \tex_delcode:D
231  \__kernel_primitive:NN \delimiter       \texDelimiter:D
232  \__kernel_primitive:NN \delimiterfactor \texDelimiterfactor:D
233  \__kernel_primitive:NN \delimitershortfall \texDelimitershortfall:D
234  \__kernel_primitive:NN \dimen           \tex_dimen:D
235  \__kernel_primitive:NN \dimendef        \tex_dimendef:D
236  \__kernel_primitive:NN \discretionary  \tex_discretionary:D
237  \__kernel_primitive:NN \displayindent   \tex_displayindent:D
238  \__kernel_primitive:NN \displaylimits   \tex_displaylimits:D
239  \__kernel_primitive:NN \displaystyle    \tex_displaystyle:D
240  \__kernel_primitive:NN \displaywidowpenalty \tex_displaywidowpenalty:D
241  \__kernel_primitive:NN \displaywidth    \tex_displaywidth:D
242  \__kernel_primitive:NN \divide          \tex_divide:D
243  \__kernel_primitive:NN \doublehyphendemerits \tex_doublehyphendemerits:D
244  \__kernel_primitive:NN \dp              \tex_dp:D
245  \__kernel_primitive:NN \dump            \tex_dump:D
246  \__kernel_primitive:NN \edef            \tex_edef:D
247  \__kernel_primitive:NN \else            \tex_else:D

```

```

248  \__kernel_primitive:NN \emergencystretch           \tex_emergencystretch:D
249  \__kernel_primitive:NN \end                         \tex_end:D
250  \__kernel_primitive:NN \endcsname                  \tex_endcsname:D
251  \__kernel_primitive:NN \endgroup                   \tex_endgroup:D
252  \__kernel_primitive:NN \endinput                   \tex_endinput:D
253  \__kernel_primitive:NN \endlinechar                \tex_endlinechar:D
254  \__kernel_primitive:NN \eqno                        \tex_eqno:D
255  \__kernel_primitive:NN \errhelp                    \tex_errhelp:D
256  \__kernel_primitive:NN \errmessage                 \tex_errmessage:D
257  \__kernel_primitive:NN \errorcontextlines          \tex_errorcontextlines:D
258  \__kernel_primitive:NN \errorstopmode              \tex_errorstopmode:D
259  \__kernel_primitive:NN \escapechar                 \tex_escapechar:D
260  \__kernel_primitive:NN \everycr                    \tex_everycr:D
261  \__kernel_primitive:NN \everydisplay               \tex_everydisplay:D
262  \__kernel_primitive:NN \everyhbox                  \tex_everyhbox:D
263  \__kernel_primitive:NN \everyjob                   \tex_everyjob:D
264  \__kernel_primitive:NN \everymath                  \tex_everymath:D
265  \__kernel_primitive:NN \everypar                   \tex_everypar:D
266  \__kernel_primitive:NN \everyvbox                  \tex_everyvbox:D
267  \__kernel_primitive:NN \exhyphenpenalty            \tex_exhyphenpenalty:D
268  \__kernel_primitive:NN \expandafter                \tex_expandafter:D
269  \__kernel_primitive:NN \fam                         \tex_fam:D
270  \__kernel_primitive:NN \fi                          \tex_ifi:D
271  \__kernel_primitive:NN \finalhyphendemerits        \tex_finalhyphendemerits:D
272  \__kernel_primitive:NN \firstmark                  \tex_firstmark:D
273  \__kernel_primitive:NN \floatingpenalty            \tex_floatingpenalty:D
274  \__kernel_primitive:NN \font                        \tex_font:D
275  \__kernel_primitive:NN \fontdimen                  \tex_fontdimen:D
276  \__kernel_primitive:NN \fontname                   \tex_fontname:D
277  \__kernel_primitive:NN \futurelet                  \tex_futurelet:D
278  \__kernel_primitive:NN \gdef                        \tex_gdef:D
279  \__kernel_primitive:NN \global                      \tex_global:D
280  \__kernel_primitive:NN \globaldefs                 \tex_globaldefs:D
281  \__kernel_primitive:NN \halign                      \tex_halign:D
282  \__kernel_primitive:NN \hangafter                  \tex_hangafter:D
283  \__kernel_primitive:NN \hangindent                 \tex_hangindent:D
284  \__kernel_primitive:NN \hbadness                   \tex_hbadness:D
285  \__kernel_primitive:NN \hbox                        \tex_hbox:D
286  \__kernel_primitive:NN \hfil                         \tex_hfil:D
287  \__kernel_primitive:NN \hfill                      \tex_hfill:D
288  \__kernel_primitive:NN \hfilneg                     \tex_hfilneg:D
289  \__kernel_primitive:NN \hfuzz                      \tex_hfuzz:D
290  \__kernel_primitive:NN \hoffset                    \tex_hoffset:D
291  \__kernel_primitive:NN \holdinginserts             \tex_holdinginserts:D
292  \__kernel_primitive:NN \hrule                      \tex_hrule:D
293  \__kernel_primitive:NN \hsize                      \tex_hsize:D
294  \__kernel_primitive:NN \hskip                      \tex_hskip:D
295  \__kernel_primitive:NN \hss                         \tex_hss:D
296  \__kernel_primitive:NN \ht                          \tex_ht:D
297  \__kernel_primitive:NN \hyphenation                \tex_hyphenation:D
298  \__kernel_primitive:NN \hyphenchar                 \tex_hyphenchar:D
299  \__kernel_primitive:NN \hyphenpenalty              \tex_hyphenpenalty:D
300  \__kernel_primitive:NN \if                         \tex_if:D
301  \__kernel_primitive:NN \ifcase                     \tex_ifcase:D

```

```

302  \__kernel_primitive:NN \ifcat          \tex_ifcat:D
303  \__kernel_primitive:NN \ifdim          \tex_ifdim:D
304  \__kernel_primitive:NN \ifeof          \tex_ifeof:D
305  \__kernel_primitive:NN \iffalse         \tex_iffalse:D
306  \__kernel_primitive:NN \ifhbox          \tex_ifhbox:D
307  \__kernel_primitive:NN \ifhmode         \tex_ifhmode:D
308  \__kernel_primitive:NN \ifinner         \tex_ifinner:D
309  \__kernel_primitive:NN \ifmmode         \tex_ifmmode:D
310  \__kernel_primitive:NN \ifnum           \tex_ifnum:D
311  \__kernel_primitive:NN \ifodd           \tex_ifodd:D
312  \__kernel_primitive:NN \iftrue          \tex_iftrue:D
313  \__kernel_primitive:NN \ifvbox          \tex_ifvbox:D
314  \__kernel_primitive:NN \ifvmode         \tex_ifvmode:D
315  \__kernel_primitive:NN \ifvoid          \tex_ifvoid:D
316  \__kernel_primitive:NN \ifx             \tex_ifx:D
317  \__kernel_primitive:NN \ignorespaces    \tex_ignorespaces:D
318  \__kernel_primitive:NN \immediate        \tex_immediate:D
319  \__kernel_primitive:NN \indent           \tex_indent:D
320  \__kernel_primitive:NN \input            \tex_input:D
321  \__kernel_primitive:NN \inputlineno       \tex_inputlineno:D
322  \__kernel_primitive:NN \insert           \tex_insert:D
323  \__kernel_primitive:NN \insertpenalties   \tex_insertpenalties:D
324  \__kernel_primitive:NN \interlinepenalty   \tex_interlinepenalty:D
325  \__kernel_primitive:NN \jobname          \tex_jobname:D
326  \__kernel_primitive:NN \kern              \tex_kern:D
327  \__kernel_primitive:NN \language          \tex_language:D
328  \__kernel_primitive:NN \lastbox           \tex_lastbox:D
329  \__kernel_primitive:NN \lastkern          \tex_lastkern:D
330  \__kernel_primitive:NN \lastpenalty        \tex_lastpenalty:D
331  \__kernel_primitive:NN \lastskip          \tex_lastskip:D
332  \__kernel_primitive:NN \lccode           \tex_lccode:D
333  \__kernel_primitive:NN \leaders           \tex_leaders:D
334  \__kernel_primitive:NN \left              \tex_left:D
335  \__kernel_primitive:NN \lefthyphenmin     \tex_lefthyphenmin:D
336  \__kernel_primitive:NN \leftskip          \tex_leftskip:D
337  \__kernel_primitive:NN \leqno             \tex_leqno:D
338  \__kernel_primitive:NN \let               \tex_let:D
339  \__kernel_primitive:NN \limits            \tex_limits:D
340  \__kernel_primitive:NN \linepenalty        \tex_linepenalty:D
341  \__kernel_primitive:NN \lineskip          \tex_lineskip:D
342  \__kernel_primitive:NN \lineskiplimit      \tex_lineskiplimit:D
343  \__kernel_primitive:NN \long               \tex_long:D
344  \__kernel_primitive:NN \looseness          \tex_looseness:D
345  \__kernel_primitive:NN \lower              \tex_lower:D
346  \__kernel_primitive:NN \lowercase          \tex_lowercase:D
347  \__kernel_primitive:NN \mag               \tex_mag:D
348  \__kernel_primitive:NN \mark              \tex_mark:D
349  \__kernel_primitive:NN \mathaccent          \tex_mathaccent:D
350  \__kernel_primitive:NN \mathbin            \tex_mathbin:D
351  \__kernel_primitive:NN \mathchar           \tex_mathchar:D
352  \__kernel_primitive:NN \mathchardef        \tex_mathchardef:D
353  \__kernel_primitive:NN \mathchoice          \tex_mathchoice:D
354  \__kernel_primitive:NN \mathclose          \tex_mathclose:D
355  \__kernel_primitive:NN \mathcode           \tex_mathcode:D

```

```

356  \__kernel_primitive:NN \mathinner
357  \__kernel_primitive:NN \mathop
358  \__kernel_primitive:NN \mathopen
359  \__kernel_primitive:NN \mathord
360  \__kernel_primitive:NN \mathpunct
361  \__kernel_primitive:NN \mathrel
362  \__kernel_primitive:NN \mathsurround
363  \__kernel_primitive:NN \maxdeadcycles
364  \__kernel_primitive:NN \maxdepth
365  \__kernel_primitive:NN \meaning
366  \__kernel_primitive:NN \medmuskip
367  \__kernel_primitive:NN \message
368  \__kernel_primitive:NN \mkern
369  \__kernel_primitive:NN \month
370  \__kernel_primitive:NN \moveleft
371  \__kernel_primitive:NN \overright
372  \__kernel_primitive:NN \mskip
373  \__kernel_primitive:NN \multiply
374  \__kernel_primitive:NN \muskip
375  \__kernel_primitive:NN \muskipdef
376  \__kernel_primitive:NN \newlinechar
377  \__kernel_primitive:NN \noalign
378  \__kernel_primitive:NN \noboundary
379  \__kernel_primitive:NN \noexpand
380  \__kernel_primitive:NN \noindent
381  \__kernel_primitive:NN \nolimits
382  \__kernel_primitive:NN \nonscript
383  \__kernel_primitive:NN \nonstopmode
384  \__kernel_primitive:NN \nulldelimiterspace
385  \__kernel_primitive:NN \nullfont
386  \__kernel_primitive:NN \number
387  \__kernel_primitive:NN \omit
388  \__kernel_primitive:NN \openin
389  \__kernel_primitive:NN \openout
390  \__kernel_primitive:NN \or
391  \__kernel_primitive:NN \outer
392  \__kernel_primitive:NN \output
393  \__kernel_primitive:NN \outputpenalty
394  \__kernel_primitive:NN \over
395  \__kernel_primitive:NN \overfullrule
396  \__kernel_primitive:NN \overline
397  \__kernel_primitive:NN \overwithdelims
398  \__kernel_primitive:NN \pagedepth
399  \__kernel_primitive:NN \pagefillstretch
400  \__kernel_primitive:NN \pagefillstretch
401  \__kernel_primitive:NN \pagefilstretch
402  \__kernel_primitive:NN \pagegoal
403  \__kernel_primitive:NN \pageshrink
404  \__kernel_primitive:NN \pagestretch
405  \__kernel_primitive:NN \pagetotal
406  \__kernel_primitive:NN \par
407  \__kernel_primitive:NN \parfillskip
408  \__kernel_primitive:NN \parindent
409  \__kernel_primitive:NN \parshape

```

\tex\_mathinner:D  
\text{tex\\_mathop}:D  
\text{tex\\_mathopen}:D  
\text{tex\\_mathord}:D  
\text{tex\\_mathpunct}:D  
\text{tex\\_mathrel}:D  
\text{tex\\_mathsurround}:D  
\text{tex\\_maxdeadcycles}:D  
\text{tex\\_maxdepth}:D  
\text{tex\\_meaning}:D  
\text{tex\\_medmuskip}:D  
\text{tex\\_message}:D  
\text{tex\\_mkern}:D  
\text{tex\\_month}:D  
\text{tex\\_moveleft}:D  
\text{tex\\_moveright}:D  
\text{tex\\_mskip}:D  
\text{tex\\_multiply}:D  
\text{tex\\_muskip}:D  
\text{tex\\_muskipdef}:D  
\text{tex\\_newlinechar}:D  
\text{tex\\_noalign}:D  
\text{tex\\_noboundary}:D  
\text{tex\\_noexpand}:D  
\text{tex\\_noindent}:D  
\text{tex\\_nolimits}:D  
\text{tex\\_nonscript}:D  
\text{tex\\_nonstopmode}:D  
\text{tex\\_nulldelimiterspace}:D  
\text{tex\\_nullfont}:D  
\text{tex\\_number}:D  
\text{tex\\_omit}:D  
\text{tex\\_openin}:D  
\text{tex\\_openout}:D  
\text{tex\\_or}:D  
\text{tex\\_outer}:D  
\text{tex\\_output}:D  
\text{tex\\_outputpenalty}:D  
\text{tex\\_over}:D  
\text{tex\\_overfullrule}:D  
\text{tex\\_overline}:D  
\text{tex\\_overwithdelims}:D  
\text{tex\\_pagedepth}:D  
\text{tex\\_pagefillstretch}:D  
\text{tex\\_pagefillstretch}:D  
\text{tex\\_pagefilstretch}:D  
\text{tex\\_pagegoal}:D  
\text{tex\\_pageshrink}:D  
\text{tex\\_pagestretch}:D  
\text{tex\\_pagetotal}:D  
\text{tex\\_par}:D  
\text{tex\\_parfillskip}:D  
\text{tex\\_parindent}:D  
\text{tex\\_parshape}:D

```

410  \__kernel_primitive:NN \parskip          \tex_parskip:D
411  \__kernel_primitive:NN \patterns        \tex_patterns:D
412  \__kernel_primitive:NN \pausing         \tex_pausing:D
413  \__kernel_primitive:NN \penalty          \tex_penalty:D
414  \__kernel_primitive:NN \postdisplaypenalty \tex_postdisplaypenalty:D
415  \__kernel_primitive:NN \predisplaypenalty \tex_predisplaypenalty:D
416  \__kernel_primitive:NN \predisplaysize   \tex_predisplaysize:D
417  \__kernel_primitive:NN \pretolerance    \tex_pretolerance:D
418  \__kernel_primitive:NN \prevdepth       \tex_prevdepth:D
419  \__kernel_primitive:NN \prevgraf        \tex_prevgraf:D
420  \__kernel_primitive:NN \radical          \tex_radical:D
421  \__kernel_primitive:NN \raise            \tex_raise:D
422  \__kernel_primitive:NN \read             \tex_read:D
423  \__kernel_primitive:NN \relax            \tex_relax:D
424  \__kernel_primitive:NN \relpenalty       \tex_relpenalty:D
425  \__kernel_primitive:NN \right           \tex_right:D
426  \__kernel_primitive:NN \righthyphenmin  \tex_righthyphenmin:D
427  \__kernel_primitive:NN \rightskip        \tex_rightskip:D
428  \__kernel_primitive:NN \romannumeral    \tex_roman numeral:D
429  \__kernel_primitive:NN \scriptfont      \tex_scriptfont:D
430  \__kernel_primitive:NN \scriptscriptfont \tex_scriptscriptfont:D
431  \__kernel_primitive:NN \scriptscriptstyle \tex_scriptscriptstyle:D
432  \__kernel_primitive:NN \scriptspace     \tex_scriptspace:D
433  \__kernel_primitive:NN \scriptstyle     \tex_scriptstyle:D
434  \__kernel_primitive:NN \scrollmode      \tex_scrollmode:D
435  \__kernel_primitive:NN \setbox           \tex_setbox:D
436  \__kernel_primitive:NN \setlanguage     \tex_setlanguage:D
437  \__kernel_primitive:NN \sfcode          \tex_sfcode:D
438  \__kernel_primitive:NN \shipout         \tex_shipout:D
439  \__kernel_primitive:NN \show            \tex_show:D
440  \__kernel_primitive:NN \showbox         \tex_showbox:D
441  \__kernel_primitive:NN \showboxbreadth \tex_showboxbreadth:D
442  \__kernel_primitive:NN \showboxdepth   \tex_showboxdepth:D
443  \__kernel_primitive:NN \showlists       \tex_showlists:D
444  \__kernel_primitive:NN \showthe         \tex_showthe:D
445  \__kernel_primitive:NN \skewchar        \tex_skewchar:D
446  \__kernel_primitive:NN \skip            \tex_skip:D
447  \__kernel_primitive:NN \skipdef         \tex_skipdef:D
448  \__kernel_primitive:NN \spacefactor    \tex_spacefactor:D
449  \__kernel_primitive:NN \spaceskip      \tex_spaceskip:D
450  \__kernel_primitive:NN \span            \tex_span:D
451  \__kernel_primitive:NN \special         \tex_special:D
452  \__kernel_primitive:NN \splitbotmark   \tex_splitbotmark:D
453  \__kernel_primitive:NN \splitfirstmark \tex_splitfirstmark:D
454  \__kernel_primitive:NN \splitmaxdepth  \tex_splitmaxdepth:D
455  \__kernel_primitive:NN \splittopskip   \tex_splittopskip:D
456  \__kernel_primitive:NN \string          \tex_string:D
457  \__kernel_primitive:NN \tabskip         \tex_tabskip:D
458  \__kernel_primitive:NN \textfont        \tex_textfont:D
459  \__kernel_primitive:NN \textstyle       \tex_textstyle:D
460  \__kernel_primitive:NN \the             \tex_the:D
461  \__kernel_primitive:NN \thickmuskip    \tex_thickmuskip:D
462  \__kernel_primitive:NN \thinmuskip     \tex_thinmuskip:D
463  \__kernel_primitive:NN \time            \tex_time:D

```

```

464  \__kernel_primitive:NN \toks          \tex_toks:D
465  \__kernel_primitive:NN \toksdef       \tex_toksdef:D
466  \__kernel_primitive:NN \tolerance     \tex_tolerance:D
467  \__kernel_primitive:NN \topmark       \tex_topmark:D
468  \__kernel_primitive:NN \topskip        \tex_topskip:D
469  \__kernel_primitive:NN \tracingcommands \tex_tracingcommands:D
470  \__kernel_primitive:NN \tracinglostchars \tex_tracinglostchars:D
471  \__kernel_primitive:NN \tracingmacros   \tex_tracingmacros:D
472  \__kernel_primitive:NN \tracingonline    \tex_tracingonline:D
473  \__kernel_primitive:NN \tracingoutput    \tex_tracingoutput:D
474  \__kernel_primitive:NN \tracingpages     \tex_tracingpages:D
475  \__kernel_primitive:NN \tracingparagraphs \tex_tracingparagraphs:D
476  \__kernel_primitive:NN \tracingrestores  \tex_tracingrestores:D
477  \__kernel_primitive:NN \tracingstats      \tex_tracingstats:D
478  \__kernel_primitive:NN \uccode          \tex_uccode:D
479  \__kernel_primitive:NN \uchyph          \tex_uchyph:D
480  \__kernel_primitive:NN \underline        \tex_underline:D
481  \__kernel_primitive:NN \unhbox          \tex_unhbox:D
482  \__kernel_primitive:NN \unhcyan        \tex_unhcyan:D
483  \__kernel_primitive:NN \unkern          \tex_unkern:D
484  \__kernel_primitive:NN \unpenalty       \tex_unpenalty:D
485  \__kernel_primitive:NN \unskip           \tex_unskip:D
486  \__kernel_primitive:NN \unvbox          \tex_unvbox:D
487  \__kernel_primitive:NN \unvcopy         \tex_unvcopy:D
488  \__kernel_primitive:NN \uppercase        \tex_uppercase:D
489  \__kernel_primitive:NN \vadjust          \tex_vadjust:D
490  \__kernel_primitive:NN \valign          \tex_valign:D
491  \__kernel_primitive:NN \vbadness         \tex_vbadness:D
492  \__kernel_primitive:NN \vbox             \tex_vbox:D
493  \__kernel_primitive:NN \vcenter          \tex_vcenter:D
494  \__kernel_primitive:NN \vfil             \tex_vfil:D
495  \__kernel_primitive:NN \vfill             \tex_vfill:D
496  \__kernel_primitive:NN \vfilneg          \tex_vfilneg:D
497  \__kernel_primitive:NN \vfuzz             \tex_vfuzz:D
498  \__kernel_primitive:NN \voffset          \tex_voffset:D
499  \__kernel_primitive:NN \vrule             \tex_vrule:D
500  \__kernel_primitive:NN \vsized           \tex_vsized:D
501  \__kernel_primitive:NN \vskip             \tex_vskip:D
502  \__kernel_primitive:NN \vsplit            \tex_vsplit:D
503  \__kernel_primitive:NN \vss               \tex_vss:D
504  \__kernel_primitive:NN \vtop              \tex_vtop:D
505  \__kernel_primitive:NN \wd                \tex_wd:D
506  \__kernel_primitive:NN \widowpenalty    \tex_widowpenalty:D
507  \__kernel_primitive:NN \write             \tex_write:D
508  \__kernel_primitive:NN \xdef              \tex_xdef:D
509  \__kernel_primitive:NN \xleaders          \tex_xleaders:D
510  \__kernel_primitive:NN \xspaceskip       \tex_xspaceskip:D
511  \__kernel_primitive:NN \year              \tex_year:D

```

Primitives introduced by  $\varepsilon$ -TEX.

```

512  \__kernel_primitive:NN \beginL          \tex_beginL:D
513  \__kernel_primitive:NN \beginR          \tex_beginR:D
514  \__kernel_primitive:NN \botmarks         \tex_botmarks:D
515  \__kernel_primitive:NN \clubpenalties   \tex_clubpenalties:D
516  \__kernel_primitive:NN \currentgrouplevel \tex_currentgrouplevel:D

```

```

517  \__kernel_primitive:NN \currentgroupotype          \tex_currentgroupotype:D
518  \__kernel_primitive:NN \currentifbranch          \tex_currentifbranch:D
519  \__kernel_primitive:NN \currentiflevel           \tex_currentiflevel:D
520  \__kernel_primitive:NN \currentiftype            \tex_currentiftype:D
521  \__kernel_primitive:NN \detokenize               \tex_detokenize:D
522  \__kernel_primitive:NN \dimexpr                 \tex_dimexpr:D
523  \__kernel_primitive:NN \displaywidowpenalties    \tex_displaywidowpenalties:D
524  \__kernel_primitive:NN \endL                     \tex_endL:D
525  \__kernel_primitive:NN \endR                     \tex_endR:D
526  \__kernel_primitive:NN \eTeXrevision             \tex_eTeXrevision:D
527  \__kernel_primitive:NN \eTeXversion              \tex_eTeXversion:D
528  \__kernel_primitive:NN \everyeof                 \tex_everyeof:D
529  \__kernel_primitive:NN \firstmarks              \tex_firstmarks:D
530  \__kernel_primitive:NN \fontchardp              \tex_fontchardp:D
531  \__kernel_primitive:NN \fontcharht              \tex_fontcharht:D
532  \__kernel_primitive:NN \fontcharic              \tex_fontcharic:D
533  \__kernel_primitive:NN \fontcharwd              \tex_fontcharwd:D
534  \__kernel_primitive:NN \glueexpr                \tex_glueexpr:D
535  \__kernel_primitive:NN \glueshrink              \tex_glueshrink:D
536  \__kernel_primitive:NN \glueshrinkorder         \tex_glueshrinkorder:D
537  \__kernel_primitive:NN \gluestretch              \tex_gluestretch:D
538  \__kernel_primitive:NN \gluestretchorder         \tex_gluestretchorder:D
539  \__kernel_primitive:NN \gluetomu                \tex_gluetomu:D
540  \__kernel_primitive:NN \ifcsname                \tex_ifcsname:D
541  \__kernel_primitive:NN \ifdefined               \tex_ifdefined:D
542  \__kernel_primitive:NN \iffontchar              \tex_iffontchar:D
543  \__kernel_primitive:NN \interactionmode         \tex_interactionmode:D
544  \__kernel_primitive:NN \interlinepenalties       \tex_interlinepenalties:D
545  \__kernel_primitive:NN \lastlinefit             \tex_lastlinefit:D
546  \__kernel_primitive:NN \lastnodetype            \tex_lastnodetype:D
547  \__kernel_primitive:NN \marks                   \tex_marks:D
548  \__kernel_primitive:NN \middle                  \tex_middle:D
549  \__kernel_primitive:NN \muexpr                 \tex_muexpr:D
550  \__kernel_primitive:NN \mutoglue                \tex_mutoglue:D
551  \__kernel_primitive:NN \numexpr                 \tex_numexpr:D
552  \__kernel_primitive:NN \pagediscards            \tex_pagediscards:D
553  \__kernel_primitive:NN \parshapedimen          \tex_parshapedimen:D
554  \__kernel_primitive:NN \parshapeindent          \tex_parshapeindent:D
555  \__kernel_primitive:NN \parshapelen            \tex_parshapelen:D
556  \__kernel_primitive:NN \predisplaydirection     \tex_predisplaydirection:D
557  \__kernel_primitive:NN \protected               \tex_protected:D
558  \__kernel_primitive:NN \readline                \tex_readline:D
559  \__kernel_primitive:NN \savinghyphcodes        \tex_savinghyphcodes:D
560  \__kernel_primitive:NN \savingvdiscards         \tex_savingvdiscards:D
561  \__kernel_primitive:NN \scantokens              \tex_scantokens:D
562  \__kernel_primitive:NN \showgroups              \tex_showgroups:D
563  \__kernel_primitive:NN \showifs                 \tex_showifs:D
564  \__kernel_primitive:NN \showtokens              \tex_showtokens:D
565  \__kernel_primitive:NN \splitbotmarks          \tex_splitbotmarks:D
566  \__kernel_primitive:NN \splitediscards          \tex_splitediscards:D
567  \__kernel_primitive:NN \splitfirstmarks        \tex_splitfirstmarks:D
568  \__kernel_primitive:NN \TeXXeTstate             \tex_TeXXeTstate:D
569  \__kernel_primitive:NN \topmarks                \tex_topmarks:D
570  \__kernel_primitive:NN \tracingassigns          \tex_tracingassigns:D

```

```

571  \__kernel_primitive:NN \tracinggroups          \tex_tracinggroups:D
572  \__kernel_primitive:NN \tracingifs           \tex_tracingifs:D
573  \__kernel_primitive:NN \tracingnesting        \tex_tracingnesting:D
574  \__kernel_primitive:NN \tracingscantokens     \tex_tracingscantokens:D
575  \__kernel_primitive:NN \unexpanded            \tex_unexpanded:D
576  \__kernel_primitive:NN \unless                \tex_unless:D
577  \__kernel_primitive:NN \widowpenalties       \tex_widowpenalties:D

```

Post- $\varepsilon$ - $\text{\TeX}$  primitives do not always end up with the same name in all engines, if indeed they are available cross-engine anyway. We therefore take the approach of preferring the shortest name that makes sense. First, we deal with the primitives introduced by pdf $\text{\TeX}$  which directly relate to PDF output: these are copied with the names unchanged.

```

578  \__kernel_primitive:NN \pdfannot              \tex_pdfannot:D
579  \__kernel_primitive:NN \pdfcatalog            \tex_pdfcatalog:D
580  \__kernel_primitive:NN \pdfcompresslevel      \tex_pdfcompresslevel:D
581  \__kernel_primitive:NN \pdfcolorstack         \tex_pdfcolorstack:D
582  \__kernel_primitive:NN \pdfcolorstackinit     \tex_pdfcolorstackinit:D
583  \__kernel_primitive:NN \pdfcreationdate       \tex_pdfcreationdate:D
584  \__kernel_primitive:NN \pdfdecimaldigits     \tex_pdfdecimaldigits:D
585  \__kernel_primitive:NN \pdfdest                \tex_pdfdest:D
586  \__kernel_primitive:NN \pdfdestmargin         \tex_pdfdestmargin:D
587  \__kernel_primitive:NN \pdfendlink            \tex_pdfendlink:D
588  \__kernel_primitive:NN \pdfendthread          \tex_pdfendthread:D
589  \__kernel_primitive:NN \pdffontattr           \tex_pdffontattr:D
590  \__kernel_primitive:NN \pdffontname           \tex_pdffontname:D
591  \__kernel_primitive:NN \pdffontobjnum         \tex_pdffontobjnum:D
592  \__kernel_primitive:NN \pdfgamma               \tex_pdfgamma:D
593  \__kernel_primitive:NN \pdfimageapplygamma    \tex_pdfimageapplygamma:D
594  \__kernel_primitive:NN \pdfimagegamma          \tex_pdfimagegamma:D
595  \__kernel_primitive:NN \pdfgentounicode       \tex_pdfgentounicode:D
596  \__kernel_primitive:NN \pdfglyphtounicode     \tex_pdfglyphtounicode:D
597  \__kernel_primitive:NN \pdfhorigin             \tex_pdfhorigin:D
598  \__kernel_primitive:NN \pdfimagehicolor       \tex_pdfimagehicolor:D
599  \__kernel_primitive:NN \pdfimageresolution    \tex_pdfimageresolution:D
600  \__kernel_primitive:NN \pdfincludechars       \tex_pdfincludechars:D
601  \__kernel_primitive:NN \pdfinclusioncopyfonts \tex_pdfinclusioncopyfonts:D
602  \__kernel_primitive:NN \pdfinclusionerrorlevel \tex_pdfinclusionerrorlevel:D
603                                \tex_pdfinclusionerrorlevel:D
604  \__kernel_primitive:NN \pdfinfo                \tex_pdfinfo:D
605  \__kernel_primitive:NN \pdflastannot          \tex_pdflastannot:D
606  \__kernel_primitive:NN \pdflastlink           \tex_pdflastlink:D
607  \__kernel_primitive:NN \pdflastobj            \tex_pdflastobj:D
608  \__kernel_primitive:NN \pdflastxform          \tex_pdflastxform:D
609  \__kernel_primitive:NN \pdflastximage         \tex_pdflastximage:D
610  \__kernel_primitive:NN \pdflastximagecolordepth \tex_pdflastximagecolordepth:D
611                                \tex_pdflastximagecolordepth:D
612  \__kernel_primitive:NN \pdflastximagepages    \tex_pdflastximagepages:D
613  \__kernel_primitive:NN \pdflinkmargin         \tex_pdflinkmargin:D
614  \__kernel_primitive:NN \pdfliteral             \tex_pdfliteral:D
615  \__kernel_primitive:NN \pdfmajorversion       \tex_pdfmajorversion:D
616  \__kernel_primitive:NN \pdfminorversion       \tex_pdfminorversion:D
617  \__kernel_primitive:NN \pdfnames              \tex_pdfnames:D
618  \__kernel_primitive:NN \pdfobj                \tex_pdfobj:D
619  \__kernel_primitive:NN \pdfobjcompresslevel   \tex_pdfobjcompresslevel:D

```

```

620  \__kernel_primitive:NN \pdfoutline           \tex_pdfoutline:D
621  \__kernel_primitive:NN \pdfoutput            \tex_pdfoutput:D
622  \__kernel_primitive:NN \pdfpageattr          \tex_pdfpageattr:D
623  \__kernel_primitive:NN \pdfpagesattr         \tex_pdfpagesattr:D
624  \__kernel_primitive:NN \pdfpagebox           \tex_pdfpagebox:D
625  \__kernel_primitive:NN \pdfpageref            \tex_pdfpageref:D
626  \__kernel_primitive:NN \pdfpageresources     \tex_pdfpageresources:D
627  \__kernel_primitive:NN \pdfpagesattr          \tex_pdfpagesattr:D
628  \__kernel_primitive:NN \pdfrefobj             \tex_pdfrefobj:D
629  \__kernel_primitive:NN \pdfrefxform          \tex_pdfrefxform:D
630  \__kernel_primitive:NN \pdfrefximage          \tex_pdfrefximage:D
631  \__kernel_primitive:NN \pdfrestore            \tex_pdfrestore:D
632  \__kernel_primitive:NN \pdfretval             \tex_pdfretval:D
633  \__kernel_primitive:NN \pdfsave               \tex_pdfsave:D
634  \__kernel_primitive:NN \pdfsetmatrix          \tex_pdfsetmatrix:D
635  \__kernel_primitive:NN \pdfstartlink         \tex_pdfstartlink:D
636  \__kernel_primitive:NN \pdfstartthread        \tex_pdfstartthread:D
637  \__kernel_primitive:NN \pdfsuppressptexinfo    \tex_pdfsuppressptexinfo:D
638  \__kernel_primitive:NN \pdfthread             \tex_pdfthread:D
639  \__kernel_primitive:NN \pdfthreadmargin       \tex_pdfthreadmargin:D
640  \__kernel_primitive:NN \pdftrailer            \tex_pdftrailer:D
641  \__kernel_primitive:NN \pdfuniqueresname      \tex_pdfuniqueresname:D
642  \__kernel_primitive:NN \pdfvorigin            \tex_pdfvorigin:D
643  \__kernel_primitive:NN \pdfxform              \tex_pdfxform:D
644  \__kernel_primitive:NN \pdfxformname          \tex_pdfxformname:D
645  \__kernel_primitive:NN \pdfximage              \tex_pdfximage:D
646  \__kernel_primitive:NN \pdfximagebbox         \tex_pdfximagebbox:D

```

These are not related to PDF output and either already appear in other engines without the \pdf prefix, or might reasonably do so at some future stage. We therefore drop the leading pdf here.

```

647  \__kernel_primitive:NN \ifpdfabsdim          \tex_ifabsdim:D
648  \__kernel_primitive:NN \ifpdfabsnum          \tex_ifabsnum:D
649  \__kernel_primitive:NN \ifpdfprimitive        \tex_ifprimitive:D
650  \__kernel_primitive:NN \pdfadjustspacing     \tex_adjustspacing:D
651  \__kernel_primitive:NN \pdfcopyfont          \tex_copyfont:D
652  \__kernel_primitive:NN \pdfdraftmode         \tex_draftmode:D
653  \__kernel_primitive:NN \pdfeachlinedepth     \tex_eachlinedepth:D
654  \__kernel_primitive:NN \pdfeachlineheight    \tex_eachlineheight:D
655  \__kernel_primitive:NN \pdfelapsedtime       \tex_elapsedtime:D
656  \__kernel_primitive:NN \pdffirstlineheight   \tex_firstlineheight:D
657  \__kernel_primitive:NN \pdffontexpand         \tex_fontexpand:D
658  \__kernel_primitive:NN \pdffontsize           \tex_fontsize:D
659  \__kernel_primitive:NN \pdfignoreddimen      \tex_ignoreddimen:D
660  \__kernel_primitive:NN \pdfinsertht          \tex_insertht:D
661  \__kernel_primitive:NN \pdflastlinedepth      \tex_lastlinedepth:D
662  \__kernel_primitive:NN \pdflastxpos           \tex_lastxpos:D
663  \__kernel_primitive:NN \pdflastypos          \tex_lastypos:D
664  \__kernel_primitive:NN \pdfmapfile            \tex_mapfile:D
665  \__kernel_primitive:NN \pdfmapline             \tex_mapline:D
666  \__kernel_primitive:NN \pdfnoligatures        \tex_noligatures:D
667  \__kernel_primitive:NN \pdfnormaldeviate      \tex_normaldeviate:D
668  \__kernel_primitive:NN \pdfpageheight         \tex_pageheight:D
669  \__kernel_primitive:NN \pdfpagewidth          \tex_pagewidth:D

```

```

670  \__kernel_primitive:NN \pdfpkmode          \tex_pkmode:D
671  \__kernel_primitive:NN \pdfpkresolution   \tex_pkresolution:D
672  \__kernel_primitive:NN \pdfprimitive       \tex_primitive:D
673  \__kernel_primitive:NN \pdfprotrudechars  \tex_protrudechars:D
674  \__kernel_primitive:NN \pdfpxdimen        \tex_pxdimen:D
675  \__kernel_primitive:NN \pdfrandomseed     \tex_randomseed:D
676  \__kernel_primitive:NN \pdfresettimer     \tex_resettimer:D
677  \__kernel_primitive:NN \pdfsavepos        \tex_savepos:D
678  \__kernel_primitive:NN \pdfsetrandomseed   \tex_setrandomseed:D
679  \__kernel_primitive:NN \pdfshellescape    \tex_shellescape:D
680  \__kernel_primitive:NN \pdftracingfonts   \tex_tracingfonts:D
681  \__kernel_primitive:NN \pdfuniformdeviate \tex_uniformdeviate:D

```

The version primitives are not related to PDF mode but are pdftEX-specific, so again are carried forward unchanged.

```

682  \__kernel_primitive:NN \pdftexbanner      \tex_pdftexbanner:D
683  \__kernel_primitive:NN \pdftexrevision    \tex_pdftexrevision:D
684  \__kernel_primitive:NN \pdftexversion     \tex_pdftexversion:D

```

These ones appear in pdftEX but don't have pdf in the name at all: no decisions to make.

```

685  \__kernel_primitive:NN \efcode            \tex_efcode:D
686  \__kernel_primitive:NN \ifincsname       \tex_ifincsname:D
687  \__kernel_primitive:NN \leftmarginkern   \tex_leftmarginkern:D
688  \__kernel_primitive:NN \letterspacefont \tex_letterspacefont:D
689  \__kernel_primitive:NN \lpcode           \tex_lpcode:D
690  \__kernel_primitive:NN \quitvmode        \tex_quitvmode:D
691  \__kernel_primitive:NN \rightmarginkern  \tex_rightmarginkern:D
692  \__kernel_primitive:NN \rpcode          \tex_rpcode:D
693  \__kernel_primitive:NN \synctex         \tex_synctex:D
694  \__kernel_primitive:NN \tagcode          \tex_tagcode:D

```

Post pdftEX primitive availability gets more complex. Both XETEX and LuaTeX have varying names for some primitives from pdftEX. Particularly for LuaTeX tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

695  </names | package>
696  <*package>
697    \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
698    \tex_long:D \tex_def:D \use_none:n #1 { }
699    \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
700    {
701      \tex_ifdefined:D #1
702        \tex_expandafter:D \use_ii:nn
703        \tex_fi:D
704        \use_none:n { \tex_global:D \tex_let:D #2 #1 }
705    }
706  </package>
707  <*names | package>

```

Some pdftEX primitives are handled here because they got dropped in LuaTeX but the corresponding internal names are emulated later. The Lua code is already loaded at this point, so we shouldn't overwrite them.

```

708  \__kernel_primitive:NN \pdfstrcmp          \tex_strcmp:D
709  \__kernel_primitive:NN \pdffilesize       \tex_filesize:D

```

```

710  \__kernel_primitive:NN \pdfmdfivesum           \tex_mdfivesum:D
711  \__kernel_primitive:NN \pdffilemoddate        \tex_filemoddate:D
712  \__kernel_primitive:NN \pdffiledump          \tex_filedump:D

```

X<sub>E</sub>T<sub>E</sub>X-specific primitives. Note that X<sub>E</sub>T<sub>E</sub>X's `\strcmp` is handled earlier and is “rolled up” into `\pdfstrcmp`. A few cross-compatibility names which lack the `pdf` of the original are handled later.

```

713  \__kernel_primitive:NN \suppressfontnotfounderror
714    \tex_suppressfontnotfounderror:D
715  \__kernel_primitive:NN \XeTeXcharclass          \tex_XeTeXcharclass:D
716  \__kernel_primitive:NN \XeTeXcharglyph         \tex_XeTeXcharglyph:D
717  \__kernel_primitive:NN \XeTeXcountfeatures     \tex_XeTeXcountfeatures:D
718  \__kernel_primitive:NN \XeTeXcountglyphs       \tex_XeTeXcountglyphs:D
719  \__kernel_primitive:NN \XeTeXcountselectors    \tex_XeTeXcountselectors:D
720  \__kernel_primitive:NN \XeTeXcountvariations   \tex_XeTeXcountvariations:D
721  \__kernel_primitive:NN \XeTeXdefaultencoding   \tex_XeTeXdefaultencoding:D
722  \__kernel_primitive:NN \XeTeXdashbreakstate    \tex_XeTeXdashbreakstate:D
723  \__kernel_primitive:NN \XeTeXfeaturecode       \tex_XeTeXfeaturecode:D
724  \__kernel_primitive:NN \XeTeXfeaturename      \tex_XeTeXfeaturename:D
725  \__kernel_primitive:NN \XeTeXfindfeaturebyname
726    \tex_XeTeXfindfeaturebyname:D
727  \__kernel_primitive:NN \XeTeXfindselectorbyname
728    \tex_XeTeXfindselectorbyname:D
729  \__kernel_primitive:NN \XeTeXfindvariationbyname
730    \tex_XeTeXfindvariationbyname:D
731  \__kernel_primitive:NN \XeTeXfirstfontchar     \tex_XeTeXfirstfontchar:D
732  \__kernel_primitive:NN \XeTeXfonttype          \tex_XeTeXfonttype:D
733  \__kernel_primitive:NN \XeTeXgenerateactualtext
734    \tex_XeTeXgenerateactualtext:D
735  \__kernel_primitive:NN \XeTeXglyph             \tex_XeTeXglyph:D
736  \__kernel_primitive:NN \XeTeXglyphbounds       \tex_XeTeXglyphbounds:D
737  \__kernel_primitive:NN \XeTeXglyphindex        \tex_XeTeXglyphindex:D
738  \__kernel_primitive:NN \XeTeXglyphname         \tex_XeTeXglyphname:D
739  \__kernel_primitive:NN \XeTeXinputencoding     \tex_XeTeXinputencoding:D
740  \__kernel_primitive:NN \XeTeXinputnormalization
741    \tex_XeTeXinputnormalization:D
742  \__kernel_primitive:NN \XeTeXinterchartokenstate
743    \tex_XeTeXinterchartokenstate:D
744  \__kernel_primitive:NN \XeTeXinterchartoks      \tex_XeTeXinterchartoks:D
745  \__kernel_primitive:NN \XeTeXisdefaultselector
746    \tex_XeTeXisdefaultselector:D
747  \__kernel_primitive:NN \XeTeXisexclusivefeature
748    \tex_XeTeXisexclusivefeature:D
749  \__kernel_primitive:NN \XeTeXlastfontchar      \tex_XeTeXlastfontchar:D
750  \__kernel_primitive:NN \XeTeXlinebreakskip     \tex_XeTeXlinebreakskip:D
751  \__kernel_primitive:NN \XeTeXlinebreaklocale   \tex_XeTeXlinebreaklocale:D
752  \__kernel_primitive:NN \XeTeXlinebreakpenalty   \tex_XeTeXlinebreakpenalty:D
753  \__kernel_primitive:NN \XeTeXOTcountfeatures   \tex_XeTeXOTcountfeatures:D
754  \__kernel_primitive:NN \XeTeXOTcountlanguages  \tex_XeTeXOTcountlanguages:D
755  \__kernel_primitive:NN \XeTeXOTcountsheets    \tex_XeTeXOTcountsheets:D
756  \__kernel_primitive:NN \XeTeXOTfeaturetag     \tex_XeTeXOTfeaturetag:D
757  \__kernel_primitive:NN \XeTeXOTlanguagetag    \tex_XeTeXOTlanguagetag:D
758  \__kernel_primitive:NN \XeTeXOTscripttag      \tex_XeTeXOTscripttag:D
759  \__kernel_primitive:NN \XeTeXpdffile          \tex_XeTeXpdffile:D

```

```

760  \__kernel_primitive:NN \XeTeXpdfpagecount      \tex_XeTeXpdfpagecount:D
761  \__kernel_primitive:NN \XeTeXpicfile          \tex_XeTeXpicfile:D
762  \__kernel_primitive:NN \XeTeXrevision         \tex_XeTeXrevision:D
763  \__kernel_primitive:NN \XeTeXselectorname     \tex_XeTeXselectorname:D
764  \__kernel_primitive:NN \XeTeXtracingfonts    \tex_XeTeXtracingfonts:D
765  \__kernel_primitive:NN \XeTeXupwardsmode      \tex_XeTeXupwardsmode:D
766  \__kernel_primitive:NN \XeTeXuseglyphmetrics  \tex_XeTeXuseglyphmetrics:D
767  \__kernel_primitive:NN \XeTeXvariation        \tex_XeTeXvariation:D
768  \__kernel_primitive:NN \XeTeXvariationdefault \tex_XeTeXvariationdefault:D
769  \__kernel_primitive:NN \XeTeXvariationmax     \tex_XeTeXvariationmax:D
770  \__kernel_primitive:NN \XeTeXvariationmin     \tex_XeTeXvariationmin:D
771  \__kernel_primitive:NN \XeTeXvariationname    \tex_XeTeXvariationname:D
772  \__kernel_primitive:NN \XeTeXversion          \tex_XeTeXversion:D

```

Primitives from pdfTeX that XeTeX renames: also helps with LuaTeX.

```

773  \__kernel_primitive:NN \creationdate          \tex_creationdate:D
774  \__kernel_primitive:NN \elapsedtime         \tex_elapsedtime:D
775  \__kernel_primitive:NN \filedump            \tex_filedump:D
776  \__kernel_primitive:NN \filemoddate        \tex_filemoddate:D
777  \__kernel_primitive:NN \filesize             \tex_filesize:D
778  \__kernel_primitive:NN \mdfivesum          \tex_mdfivesum:D
779  \__kernel_primitive:NN \ifprimitive        \tex_ifprimitive:D
780  \__kernel_primitive:NN \primitive           \tex_primitive:D
781  \__kernel_primitive:NN \resettimer          \tex_resettimer:D
782  \__kernel_primitive:NN \shellescape         \tex_shellescape:D

```

Primitives from LuaTeX, some of which have been ported back to XeTeX.

```

783  \__kernel_primitive:NN \alignmark            \tex_alignmark:D
784  \__kernel_primitive:NN \aligntab             \tex_aligntab:D
785  \__kernel_primitive:NN \attribute            \tex_attribute:D
786  \__kernel_primitive:NN \attributedef       \tex_attributedef:D
787  \__kernel_primitive:NN \automaticdiscretionary \tex_automaticdiscretionary:D
788  \__kernel_primitive:NN \automatichyphenmode \tex_automatichyphenmode:D
789  \__kernel_primitive:NN \automatichyphenpenalty \tex_automatichyphenpenalty:D
790  \__kernel_primitive:NN \beginicsname         \tex_beginicsname:D
791  \__kernel_primitive:NN \bodydir              \tex_bodydir:D
792  \__kernel_primitive:NN \bodydirection       \tex_bodydirection:D
793  \__kernel_primitive:NN \boxdir               \tex_boxdir:D
794  \__kernel_primitive:NN \boxdirection        \tex_boxdirection:D
795  \__kernel_primitive:NN \breakafterdirmode   \tex_breakafterdirmode:D
796  \__kernel_primitive:NN \catcodetable       \tex_catcodetable:D
797  \__kernel_primitive:NN \clearmarks          \tex_clearmarks:D
798  \__kernel_primitive:NN \crampeddisplaystyle \tex_crampeddisplaystyle:D
799  \__kernel_primitive:NN \crampedscriptscriptstyle \tex_crampedscriptscriptstyle:D
800  \__kernel_primitive:NN \crampedscriptstyle   \tex_crampedscriptstyle:D
801  \__kernel_primitive:NN \crampedtextstyle     \tex_crampedtextstyle:D
802  \__kernel_primitive:NN \csstring            \tex_csstring:D
803  \__kernel_primitive:NN \discretionaryligaturemode \tex_discretionaryligaturemode:D
804  \__kernel_primitive:NN \directlua           \tex_directlua:D
805  \__kernel_primitive:NN \dvieextension       \tex_dviextension:D
806  \__kernel_primitive:NN \dvifeedback         \tex_dvifeedback:D

```

```

811  \__kernel_primitive:NN \dvivariable           \tex_dvivariable:D
812  \__kernel_primitive:NN \eTeXglueshrinkorder   \tex_eTeXglueshrinkorder:D
813  \__kernel_primitive:NN \eTeXgluestretchorder   \tex_eTeXgluestretchorder:D
814  \__kernel_primitive:NN \etoksapp              \tex_etoksapp:D
815  \__kernel_primitive:NN \etokspre              \tex_etokspre:D
816  \__kernel_primitive:NN \exceptionpenalty     \tex_exceptionpenalty:D
817  \__kernel_primitive:NN \explicithyphenpenalty \tex_explicithyphenpenalty:D
818  \__kernel_primitive:NN \expanded               \tex_expanded:D
819  \__kernel_primitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
820  \__kernel_primitive:NN \firstvalidlanguage    \tex_firstvalidlanguage:D
821  \__kernel_primitive:NN \fontid                \tex_fontid:D
822  \__kernel_primitive:NN \formatname           \tex_formatname:D
823  \__kernel_primitive:NN \hjcode                \tex_hjcode:D
824  \__kernel_primitive:NN \hpack                 \tex_hpack:D
825  \__kernel_primitive:NN \hyphenationbounds    \tex_hyphenationbounds:D
826  \__kernel_primitive:NN \hyphenationmin       \tex_hyphenationmin:D
827  \__kernel_primitive:NN \hyphenpenaltymode    \tex_hyphenpenaltymode:D
828  \__kernel_primitive:NN \gleaders              \tex_gleaders:D
829  \__kernel_primitive:NN \ifcondition          \tex_ifcondition:D
830  \__kernel_primitive:NN \immediateassigned    \tex_immediateassigned:D
831  \__kernel_primitive:NN \immediateassignment  \tex_immediateassignment:D
832  \__kernel_primitive:NN \initcatcodetable    \tex_initcatcodetable:D
833  \__kernel_primitive:NN \lastnamedcs         \tex_lastnamedcs:D
834  \__kernel_primitive:NN \latelua              \tex_latelua:D
835  \__kernel_primitive:NN \lateluafunction      \tex_lateluafunction:D
836  \__kernel_primitive:NN \leftghost            \tex_leftghost:D
837  \__kernel_primitive:NN \letcharcode          \tex_letcharcode:D
838  \__kernel_primitive:NN \linedir               \tex_linedir:D
839  \__kernel_primitive:NN \linedirection        \tex_linedirection:D
840  \__kernel_primitive:NN \localbrokenpenalty   \tex_localbrokenpenalty:D
841  \__kernel_primitive:NN \localinterlinepenalty \tex_localinterlinepenalty:D
842  \__kernel_primitive:NN \luabytecode          \tex_luabytecode:D
843  \__kernel_primitive:NN \luabytecodecall     \tex_luabytecodecall:D
844  \__kernel_primitive:NN \luacopyinputnodes   \tex_luacopyinputnodes:D
845  \__kernel_primitive:NN \luadef               \tex_luadef:D
846  \__kernel_primitive:NN \localleftbox         \tex_localleftbox:D
847  \__kernel_primitive:NN \localrightbox        \tex_localrightbox:D
848  \__kernel_primitive:NN \luaescapestring     \tex_luaescapestring:D
849  \__kernel_primitive:NN \luafunction          \tex_luafunction:D
850  \__kernel_primitive:NN \luafunctioncall    \tex_luafunctioncall:D
851  \__kernel_primitive:NN \luatexbanner        \tex_luatexbanner:D
852  \__kernel_primitive:NN \luatexrevision      \tex_luatexrevision:D
853  \__kernel_primitive:NN \luatexversion       \tex_luatexversion:D
854  \__kernel_primitive:NN \mathdefaultsmode    \tex_mathdefaultsmode:D
855  \__kernel_primitive:NN \mathdelimitersmode   \tex_mathdelimitersmode:D
856  \__kernel_primitive:NN \mathdir              \tex_mathdir:D
857  \__kernel_primitive:NN \mathdirection       \tex_mathdirection:D
858  \__kernel_primitive:NN \mathdisplayskipmode \tex_mathdisplayskipmode:D
859  \__kernel_primitive:NN \matheqdirmode       \tex_matheqdirmode:D
860  \__kernel_primitive:NN \matheqnogapstep     \tex_matheqnogapstep:D
861  \__kernel_primitive:NN \mathnolimitsmode     \tex_mathnolimitsmode:D
862  \__kernel_primitive:NN \mathoption           \tex_mathoption:D
863  \__kernel_primitive:NN \mathpenaltiesmode   \tex_mathpenaltiesmode:D
864  \__kernel_primitive:NN \mathrulesfam        \tex_mathrulesfam:D

```

```

865  \__kernel_primitive:NN \mathscriptsmode          \tex_mathscriptsmode:D
866  \__kernel_primitive:NN \mathscriptboxmode       \tex_mathscriptboxmode:D
867  \__kernel_primitive:NN \mathscriptcharmode     \tex_mathscriptcharmode:D
868  \__kernel_primitive:NN \mathstyle              \tex_mathstyle:D
869  \__kernel_primitive:NN \mathsurroundmode        \tex_mathsurroundmode:D
870  \__kernel_primitive:NN \mathsurroundskip       \tex_mathsurroundskip:D
871  \__kernel_primitive:NN \nohrule               \tex_nohtilde:D
872  \__kernel_primitive:NN \nokerns                \tex_nokerns:D
873  \__kernel_primitive:NN \noligs                 \tex_noligs:D
874  \__kernel_primitive:NN \nospaces              \tex_nospaces:D
875  \__kernel_primitive:NN \novrule               \tex_novrule:D
876  \__kernel_primitive:NN \outputbox             \tex_outputbox:D
877  \__kernel_primitive:NN \pagebottomoffset      \tex_pagebottomoffset:D
878  \__kernel_primitive:NN \pagedir                \tex_pagedir:D
879  \__kernel_primitive:NN \pagedirection         \tex_pagedirection:D
880  \__kernel_primitive:NN \pageleftoffset        \tex_pageleftoffset:D
881  \__kernel_primitive:NN \pagerightoffset       \tex_pagerightoffset:D
882  \__kernel_primitive:NN \pagetopoffset         \tex_pagetopoffset:D
883  \__kernel_primitive:NN \pardir                \tex_pardir:D
884  \__kernel_primitive:NN \pardirection          \tex_pardirection:D
885  \__kernel_primitive:NN \pdfextension          \tex_pdfextension:D
886  \__kernel_primitive:NN \pdffeedback          \tex_pdffeedback:D
887  \__kernel_primitive:NN \pdfvariable           \tex_pdfvariable:D
888  \__kernel_primitive:NN \postexhyphenchar      \tex_postexhyphenchar:D
889  \__kernel_primitive:NN \posthyphenchar         \tex_posthyphenchar:D
890  \__kernel_primitive:NN \prebinopenpenalty      \tex_prebinopenpenalty:D
891  \__kernel_primitive:NN \predisplaygapfactor   \tex_predisplaygapfactor:D
892  \__kernel_primitive:NN \preexhyphenchar        \tex_preexhyphenchar:D
893  \__kernel_primitive:NN \prehyphenchar          \tex_prehyphenchar:D
894  \__kernel_primitive:NN \prerelpenalty          \tex_prerelpenalty:D
895  \__kernel_primitive:NN \rightghost             \tex_rightghost:D
896  \__kernel_primitive:NN \savecatcodetable       \tex_savecatcodetable:D
897  \__kernel_primitive:NN \scantexttokens         \tex_scantexttokens:D
898  \__kernel_primitive:NN \setfontid              \tex_setfontid:D
899  \__kernel_primitive:NN \shapemode              \tex_shapemode:D
900  \__kernel_primitive:NN \suppressifcsnameerror \tex_suppressifcsnameerror:D
901  \__kernel_primitive:NN \suppresslongerror      \tex_suppresslongerror:D
902  \__kernel_primitive:NN \suppressmathparerror    \tex_suppressmathparerror:D
903  \__kernel_primitive:NN \suppressoutererror     \tex_suppressoutererror:D
904  \__kernel_primitive:NN \suppressprimitiveerror \tex_suppressprimitiveerror:D
905  \__kernel_primitive:NN \textdir                \tex_textdir:D
906  \__kernel_primitive:NN \textdirection          \tex_textdirection:D
907  \__kernel_primitive:NN \toksapp                \tex_toksapp:D
908  \__kernel_primitive:NN \tokspre                \tex_tokspre:D
909  \__kernel_primitive:NN \tpack                  \tex_tpck:D
910  \__kernel_primitive:NN \vpack                  \tex_vpack:D
911  \__kernel_primitive:NN \vpack

```

Primitives from pdftEX that LuaTeX renames.

```

912  \__kernel_primitive:NN \adjustspacing          \tex_adjustspacing:D
913  \__kernel_primitive:NN \copyfont              \tex_copyfont:D
914  \__kernel_primitive:NN \draftmode             \tex_draftmode:D
915  \__kernel_primitive:NN \expandglyphsinfont   \tex_fontexpand:D
916  \__kernel_primitive:NN \ifabsdim              \tex_ifabsdim:D
917  \__kernel_primitive:NN \ifabsnum              \tex_ifabsnum:D

```

```

918  \__kernel_primitive:NN \ignoresligaturesinfont \tex_ignoresligaturesinfont:D
919  \__kernel_primitive:NN \insertht \tex_insertht:D
920  \__kernel_primitive:NN \lastsavedboxresourceindex
921    \tex_pdflastxform:D
922  \__kernel_primitive:NN \lastsavedimageresourceindex
923    \tex_pdflastximage:D
924  \__kernel_primitive:NN \lastsavedimageresourcepages
925    \tex_pdflastximagepages:D
926  \__kernel_primitive:NN \lastxpos \tex_lastxpos:D
927  \__kernel_primitive:NN \lastypos \tex_lastypos:D
928  \__kernel_primitive:NN \normaldeviate \tex_normaldeviate:D
929  \__kernel_primitive:NN \outputmode \tex_pdfoutput:D
930  \__kernel_primitive:NN \pageheight \tex_pageheight:D
931  \__kernel_primitive:NN \pagewidth \tex_pagewidth:D
932  \__kernel_primitive:NN \protrudechars \tex_protrudechars:D
933  \__kernel_primitive:NN \pxdimen \tex_pxdimen:D
934  \__kernel_primitive:NN \randomseed \tex_randomseed:D
935  \__kernel_primitive:NN \useboxresource \tex_pdfrefxform:D
936  \__kernel_primitive:NN \useimageresource \tex_pdfrefximage:D
937  \__kernel_primitive:NN \savepos \tex_savepos:D
938  \__kernel_primitive:NN \saveboxresource \tex_pdfxform:D
939  \__kernel_primitive:NN \saveimageresource \tex_pdfximage:D
940  \__kernel_primitive:NN \setrandomseed \tex_setrandomseed:D
941  \__kernel_primitive:NN \tracingfonts \tex_tracingfonts:D
942  \__kernel_primitive:NN \uniformdeviate \tex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by Xe<sub>H</sub>T<sub>E</sub>X and LuaT<sub>E</sub>X in a somewhat complex fashion: a few first as \Xe<sub>T</sub>e<sub>X</sub>... which were then renamed with LuaT<sub>E</sub>X having a lot more. These names now all start \U... and mainly \Umath....

```

943  \__kernel_primitive:NN \Uchar \tex_Uchar:D
944  \__kernel_primitive:NN \Ucharcat \tex_Ucharcat:D
945  \__kernel_primitive:NN \Udelcode \tex_Udelcode:D
946  \__kernel_primitive:NN \Udelcodenum \tex_Udelcodenum:D
947  \__kernel_primitive:NN \Udelimiter \tex_Udelimiter:D
948  \__kernel_primitive:NN \Udelimiterover \tex_Udelimiterover:D
949  \__kernel_primitive:NN \Udelimiterunder \tex_Udelimiterunder:D
950  \__kernel_primitive:NN \Uhextensible \tex_Uhextensible:D
951  \__kernel_primitive:NN \Umathaccent \tex_Umathaccent:D
952  \__kernel_primitive:NN \Umathaxis \tex_Umathaxis:D
953  \__kernel_primitive:NN \Umathbinbinspacing \tex_Umathbinbinspacing:D
954  \__kernel_primitive:NN \Umathbinclosespacing \tex_Umathbinclosespacing:D
955  \__kernel_primitive:NN \Umathbininnerspacing \tex_Umathbininnerspacing:D
956  \__kernel_primitive:NN \Umathbinopenspacing \tex_Umathbinopenspacing:D
957  \__kernel_primitive:NN \Umathbinopspacing \tex_Umathbinopspacing:D
958  \__kernel_primitive:NN \Umathbinordspacing \tex_Umathbinordspacing:D
959  \__kernel_primitive:NN \Umathbinpunctspacing \tex_Umathbinpunctspacing:D
960  \__kernel_primitive:NN \Umathbinrelspacing \tex_Umathbinrelspacing:D
961  \__kernel_primitive:NN \Umathchar \tex_Umathchar:D
962  \__kernel_primitive:NN \Umathcharclass \tex_Umathcharclass:D
963  \__kernel_primitive:NN \Umathchardef \tex_Umathchardef:D
964  \__kernel_primitive:NN \Umathcharfam \tex_Umathcharfam:D
965  \__kernel_primitive:NN \Umathcharnum \tex_Umathcharnum:D
966  \__kernel_primitive:NN \Umathcharnumdef \tex_Umathcharnumdef:D
967  \__kernel_primitive:NN \Umathcharslot \tex_Umathcharslot:D

```

```

968  \__kernel_primitive:NN \Umathclosebinspacing  \tex_Umathclosebinspacing:D
969  \__kernel_primitive:NN \Umathcloseclosespacing
970   \tex_Umathcloseclosespacing:D
971  \__kernel_primitive:NN \Umathcloseinnerspacing
972   \tex_Umathcloseinnerspacing:D
973  \__kernel_primitive:NN \Umathcloseopenspacing \tex_Umathcloseopenspacing:D
974  \__kernel_primitive:NN \Umathcloseopspacing   \tex_Umathcloseopspacing:D
975  \__kernel_primitive:NN \Umathcloseordspacing  \tex_Umathcloseordspacing:D
976  \__kernel_primitive:NN \Umathclosepunctspacing
977   \tex_Umathclosepunctspacing:D
978  \__kernel_primitive:NN \Umathcloserelspacing  \tex_Umathcloserelspacing:D
979  \__kernel_primitive:NN \Umathcode          \tex_Umathcode:D
980  \__kernel_primitive:NN \Umathcodenum       \tex_Umathcodenum:D
981  \__kernel_primitive:NN \Umathconnectoroverlapmin
982   \tex_Umathconnectoroverlapmin:D
983  \__kernel_primitive:NN \Umathfractiondelsize \tex_Umathfractiondelsize:D
984  \__kernel_primitive:NN \Umathfractiondenomdown
985   \tex_Umathfractiondenomdown:D
986  \__kernel_primitive:NN \Umathfractiondenomvgap
987   \tex_Umathfractiondenomvgap:D
988  \__kernel_primitive:NN \Umathfractionnumup   \tex_Umathfractionnumup:D
989  \__kernel_primitive:NN \Umathfractionnumvgap \tex_Umathfractionnumvgap:D
990  \__kernel_primitive:NN \Umathfractionrule    \tex_Umathfractionrule:D
991  \__kernel_primitive:NN \Umathinnerbinspacing \tex_Umathinnerbinspacing:D
992  \__kernel_primitive:NN \Umathinnerclosespacing
993   \tex_Umathinnerclosespacing:D
994  \__kernel_primitive:NN \Umathinnerinnerspacing
995   \tex_Umathinnerinnerspacing:D
996  \__kernel_primitive:NN \Umathinneropenspacing \tex_Umathinneropenspacing:D
997  \__kernel_primitive:NN \Umathinneropspacing  \tex_Umathinneropspacing:D
998  \__kernel_primitive:NN \Umathinnerordspacing \tex_Umathinnerordspacing:D
999  \__kernel_primitive:NN \Umathinnerpunctspacing
1000   \tex_Umathinnerpunctspacing:D
1001 \__kernel_primitive:NN \Umathinnerrelspacing \tex_Umathinnerrelspacing:D
1002 \__kernel_primitive:NN \Umathlimitabovebgap \tex_Umathlimitabovebgap:D
1003 \__kernel_primitive:NN \Umathlimitabovekern \tex_Umathlimitabovekern:D
1004 \__kernel_primitive:NN \Umathlimitabovevgap \tex_Umathlimitabovevgap:D
1005 \__kernel_primitive:NN \Umathlimitbelowbgap \tex_Umathlimitbelowbgap:D
1006 \__kernel_primitive:NN \Umathlimitbelowkern \tex_Umathlimitbelowkern:D
1007 \__kernel_primitive:NN \Umathlimitbelowvgap \tex_Umathlimitbelowvgap:D
1008 \__kernel_primitive:NN \Umathnolimitsubfactor \tex_Umathnolimitsubfactor:D
1009 \__kernel_primitive:NN \Umathnolimitsupfactor \tex_Umathnolimitsupfactor:D
1010 \__kernel_primitive:NN \Umathopbinspacing   \tex_Umathopbinspacing:D
1011 \__kernel_primitive:NN \Umathpclosespacing  \tex_Umathpclosespacing:D
1012 \__kernel_primitive:NN \Umathopenbinspacing \tex_Umathopenbinspacing:D
1013 \__kernel_primitive:NN \Umathopenclosespacing \tex_Umathopenclosespacing:D
1014 \__kernel_primitive:NN \Umathopeninnerspacing \tex_Umathopeninnerspacing:D
1015 \__kernel_primitive:NN \Umathopenopenspacing \tex_Umathopenopenspacing:D
1016 \__kernel_primitive:NN \Umathopenopspacing  \tex_Umathopenopspacing:D
1017 \__kernel_primitive:NN \Umathopenordspacing \tex_Umathopenordspacing:D
1018 \__kernel_primitive:NN \Umathopenpunctspacing \tex_Umathopenpunctspacing:D
1019 \__kernel_primitive:NN \Umathopenrelspacing \tex_Umathopenrelspacing:D
1020 \__kernel_primitive:NN \Umathoperatorsize   \tex_Umathoperatorsize:D
1021 \__kernel_primitive:NN \Umathopinnerspacing \tex_Umathopinnerspacing:D

```

```

1022  \__kernel_primitive:NN \Umathopopenspacing      \tex_Umathopopenspacing:D
1023  \__kernel_primitive:NN \Umathopspacing          \tex_Umathopspacing:D
1024  \__kernel_primitive:NN \Umathopordspacing       \tex_Umathopordspacing:D
1025  \__kernel_primitive:NN \Umathoppunctspacing     \tex_Umathoppunctspacing:D
1026  \__kernel_primitive:NN \Umathoprelspacing       \tex_Umathoprelspacing:D
1027  \__kernel_primitive:NN \Umathordbinspacing       \tex_Umathordbinspacing:D
1028  \__kernel_primitive:NN \Umathordclosespacing     \tex_Umathordclosespacing:D
1029  \__kernel_primitive:NN \Umathordinnerspacing     \tex_Umathordinnerspacing:D
1030  \__kernel_primitive:NN \Umathordopenspacing      \tex_Umathordopenspacing:D
1031  \__kernel_primitive:NN \Umathordd spacing        \tex_Umathordd spacing:D
1032  \__kernel_primitive:NN \Umathordordspacing       \tex_Umathordordspacing:D
1033  \__kernel_primitive:NN \Umathordpunctspacing     \tex_Umathordpunctspacing:D
1034  \__kernel_primitive:NN \Umathordre l spacing     \tex_Umathordre l spacing:D
1035  \__kernel_primitive:NN \Umathoverbarkern        \tex_Umathoverbarkern:D
1036  \__kernel_primitive:NN \Umathoverbarrule        \tex_Umathoverbarrule:D
1037  \__kernel_primitive:NN \Umathoverbarvgap         \tex_Umathoverbarvgap:D
1038  \__kernel_primitive:NN \Umathoverdelimiterbgap   \tex_Umathoverdelimiterbgap:D
1039  \__kernel_primitive:NN \Umathoverdelimtervgap    \tex_Umathoverdelimtervgap:D
1040  \__kernel_primitive:NN \Umathoverdelimitervgap   \tex_Umathoverdelimitervgap:D
1041  \__kernel_primitive:NN \Umathpunctbinspacing     \tex_Umathpunctbinspacing:D
1042  \__kernel_primitive:NN \Umathpunctclosespacing   \tex_Umathpunctclosespacing:D
1043  \__kernel_primitive:NN \Umathpunctclosespacing  \tex_Umathpunctclosespacing:D
1044  \__kernel_primitive:NN \Umathpunctinnerspacing   \tex_Umathpunctinnerspacing:D
1045  \__kernel_primitive:NN \Umathpuncttinerspacing  \tex_Umathpuncttinerspacing:D
1046  \__kernel_primitive:NN \Umathpuncttopenspacing   \tex_Umathpuncttopenspacing:D
1047  \__kernel_primitive:NN \Umathpuncttopspacing     \tex_Umathpuncttopspacing:D
1048  \__kernel_primitive:NN \Umathpunctordspacing    \tex_Umathpunctordspacing:D
1049  \__kernel_primitive:NN \Umathpunctordspacing     \tex_Umathpunctordspacing:D
1050  \__kernel_primitive:NN \Umathpunctpunctspacing  \tex_Umathpunctpunctspacing:D
1051  \__kernel_primitive:NN \Umathpunctrelspacing    \tex_Umathpunctrelspacing:D
1052  \__kernel_primitive:NN \Umathquad                \tex_Umathquad:D
1053  \__kernel_primitive:NN \Umathradicaldegreeafter \tex_Umathradicaldegreeafter:D
1054  \__kernel_primitive:NN \Umathradicaldegreebefore \tex_Umathradicaldegreebefore:D
1055  \__kernel_primitive:NN \Umathradicaldegreeraise  \tex_Umathradicaldegreeraise:D
1056  \__kernel_primitive:NN \Umathradicaldegreeraise \tex_Umathradicaldegreeraise:D
1057  \__kernel_primitive:NN \Umathradicalrule        \tex_Umathradicalrule:D
1058  \__kernel_primitive:NN \Umathradicalrule        \tex_Umathradicalrule:D
1059  \__kernel_primitive:NN \Umathradicalrule        \tex_Umathradicalrule:D
1060  \__kernel_primitive:NN \Umathradicalkern        \tex_Umathradicalkern:D
1061  \__kernel_primitive:NN \Umathradicalrule        \tex_Umathradicalrule:D
1062  \__kernel_primitive:NN \Umathradicalvgap        \tex_Umathradicalvgap:D
1063  \__kernel_primitive:NN \Umathrelbinspacing     \tex_Umathrelbinspacing:D
1064  \__kernel_primitive:NN \Umathrelclosespacing   \tex_Umathrelclosespacing:D
1065  \__kernel_primitive:NN \Umathrelinnerspacing   \tex_Umathrelinnerspacing:D
1066  \__kernel_primitive:NN \Umathrelopspacing      \tex_Umathrelopspacing:D
1067  \__kernel_primitive:NN \Umathrelopspacing      \tex_Umathrelopspacing:D
1068  \__kernel_primitive:NN \Umathrelordspacing    \tex_Umathrelordspacing:D
1069  \__kernel_primitive:NN \Umathrelpunctspacing  \tex_Umathrelpunctspacing:D
1070  \__kernel_primitive:NN \Umathrelrelspacing     \tex_Umathrelrelspacing:D
1071  \__kernel_primitive:NN \Umathskewedfractionhgap \tex_Umathskewedfractionhgap:D
1072  \__kernel_primitive:NN \Umathskewedfractionvgap \tex_Umathskewedfractionvgap:D
1073  \__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
1074  \__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
1075  \__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D

```

```

1076  \__kernel_primitive:NN \Umathstackdenomdown \tex_Umathstackdenomdown:D
1077  \__kernel_primitive:NN \Umathstacknumup \tex_Umathstacknumup:D
1078  \__kernel_primitive:NN \Umathstackvgap \tex_Umathstackvgap:D
1079  \__kernel_primitive:NN \Umathsubshiftdown \tex_Umathsubshiftdown:D
1080  \__kernel_primitive:NN \Umathsubshiftdrop \tex_Umathsubshiftdrop:D
1081  \__kernel_primitive:NN \Umathsubsupshiftdown \tex_Umathsubsupshiftdown:D
1082  \__kernel_primitive:NN \Umathsubsupvgap \tex_Umathsubsupvgap:D
1083  \__kernel_primitive:NN \Umathsubtopmax \tex_Umathsubtopmax:D
1084  \__kernel_primitive:NN \Umathsupbottommin \tex_Umathsupbottommin:D
1085  \__kernel_primitive:NN \Umathsupshiftdrop \tex_Umathsupshiftdrop:D
1086  \__kernel_primitive:NN \Umathsupshiftup \tex_Umathsupshiftup:D
1087  \__kernel_primitive:NN \Umathsupsubbottommax \tex_Umathsupsubbottommax:D
1088  \__kernel_primitive:NN \Umathunderbarkern \tex_Umathunderbarkern:D
1089  \__kernel_primitive:NN \Umathunderbarrule \tex_Umathunderbarrule:D
1090  \__kernel_primitive:NN \Umathunderbarvgap \tex_Umathunderbarvgap:D
1091  \__kernel_primitive:NN \Umathunderdelimitergap
    \tex_Umathunderdelimitergap:D
1092  \__kernel_primitive:NN \Umathunderdelimtervgap
    \tex_Umathunderdelimtervgap:D
1093  \__kernel_primitive:NN \Umathunderdelimtervgap
    \tex_Umathunderdelimtervgap:D
1094  \__kernel_primitive:NN \Unosubscript \tex_Unosubscript:D
1095  \__kernel_primitive:NN \Unosuperscript \tex_Unosuperscript:D
1096  \__kernel_primitive:NN \Uoverdelimiter \tex_Uoverdelimiter:D
1097  \__kernel_primitive:NN \Uradical \tex_Uradical:D
1098  \__kernel_primitive:NN \Uroot \tex_Uroot:D
1099  \__kernel_primitive:NN \Uskewed \tex_Uskewed:D
1100  \__kernel_primitive:NN \Uskewedwithdelims \tex_Uskewedwithdelims:D
1101  \__kernel_primitive:NN \Ustack \tex_Ustack:D
1102  \__kernel_primitive:NN \Ustartdisplaymath \tex_Ustartdisplaymath:D
1103  \__kernel_primitive:NN \Ustartmath \tex_Ustartmath:D
1104  \__kernel_primitive:NN \Ustopdisplaymath \tex_Ustopdisplaymath:D
1105  \__kernel_primitive:NN \Ustopmath \tex_Ustopmath:D
1106  \__kernel_primitive:NN \Usubscript \tex_Usubscript:D
1107  \__kernel_primitive:NN \Usuperscript \tex_Usuperscript:D
1108  \__kernel_primitive:NN \Uunderdelimiter \tex_Uunderdelimiter:D
1109  \__kernel_primitive:NN \Uvextensible \tex_Uvextensible:D

```

Primitives from pTeX.

```

1111  \__kernel_primitive:NN \autospacing \tex_autospacing:D
1112  \__kernel_primitive:NN \autoxspacing \tex_autoxspacing:D
1113  \__kernel_primitive:NN \currentcjktoken \tex_currentcjktoken:D
1114  \__kernel_primitive:NN \currentspacingmode \tex_currentspacingmode:D
1115  \__kernel_primitive:NN \currentxspacingmode \tex_currentxspacingmode:D
1116  \__kernel_primitive:NN \disinhibitglue \tex_disinhibitglue:D
1117  \__kernel_primitive:NN \dtou \tex_dtou:D
1118  \__kernel_primitive:NN \epTeXinputencoding \tex_epTeXinputencoding:D
1119  \__kernel_primitive:NN \epTeXversion \tex_epTeXversion:D
1120  \__kernel_primitive:NN \euc \tex_euc:D
1121  \__kernel_primitive:NN \hfi \tex_hfi:D
1122  \__kernel_primitive:NN \ifdbox \tex_ifdbox:D
1123  \__kernel_primitive:NN \ifddir \tex_ifddir:D
1124  \__kernel_primitive:NN \ifjfont \tex_ifjfont:D
1125  \__kernel_primitive:NN \ifmbox \tex_ifmbox:D
1126  \__kernel_primitive:NN \ifmdir \tex_ifmdir:D
1127  \__kernel_primitive:NN \iftbox \tex_iftbox:D
1128  \__kernel_primitive:NN \iftfont \tex_iftfont:D

```

```

1129  \__kernel_primitive:NN \iftdir           \tex_iftdir:D
1130  \__kernel_primitive:NN \ifybox           \tex_ifybox:D
1131  \__kernel_primitive:NN \ifydir           \tex_ifydir:D
1132  \__kernel_primitive:NN \inhibitglue      \tex_inhibitglue:D
1133  \__kernel_primitive:NN \inhibitxspcode   \tex_inhibitxspcode:D
1134  \__kernel_primitive:NN \jcharwidowpenalty \tex_jcharwidowpenalty:D
1135  \__kernel_primitive:NN \jfam              \tex_jfam:D
1136  \__kernel_primitive:NN \jfont              \tex_jfont:D
1137  \__kernel_primitive:NN \jis                \tex_jis:D
1138  \__kernel_primitive:NN \kanjiskip         \tex_kanjiskip:D
1139  \__kernel_primitive:NN \kansuji           \tex_kansuji:D
1140  \__kernel_primitive:NN \kansujichar       \tex_kansujichar:D
1141  \__kernel_primitive:NN \kcatcode          \tex_kcatcode:D
1142  \__kernel_primitive:NN \kuten              \tex_kuten:D
1143  \__kernel_primitive:NN \lastnodechar      \tex_lastnodechar:D
1144  \__kernel_primitive:NN \lastnodefont      \tex_lastnodefont:D
1145  \__kernel_primitive:NN \lastnodesubtype   \tex_lastnodesubtype:D
1146  \__kernel_primitive:NN \noautospacing     \tex_noautospacing:D
1147  \__kernel_primitive:NN \noautoxspacing    \tex_noautoxspacing:D
1148  \__kernel_primitive:NN \pagefistretch      \tex_pagefistretch:D
1149  \__kernel_primitive:NN \postbreakpenalty   \tex_postbreakpenalty:D
1150  \__kernel_primitive:NN \prebreakpenalty    \tex_prebreakpenalty:D
1151  \__kernel_primitive:NN \ptexlineendmode    \tex_lineendmode:D
1152  \__kernel_primitive:NN \ptxminorversion    \tex_ptxminorversion:D
1153  \__kernel_primitive:NN \ptexrevision       \tex_ptexrevision:D
1154  \__kernel_primitive:NN \ptexversion        \tex_ptexversion:D
1155  \__kernel_primitive:NN \readpapersizespecial \tex_readpapersizespecial:D
1156  \__kernel_primitive:NN \scriptbaselineshiftfactor \tex_scriptbaselineshiftfactor
1157   \tex_scriptbaselineshiftfactor:D
1158  \__kernel_primitive:NN \scriptscriptbaselineshiftfactor \tex_scriptscriptbaselineshiftfactor
1159   \tex_scriptscriptbaselineshiftfactor:D
1160  \__kernel_primitive:NN \showmode           \tex_showmode:D
1161  \__kernel_primitive:NN \sjis               \tex_sjis:D
1162  \__kernel_primitive:NN \tate                \tex_tate:D
1163  \__kernel_primitive:NN \tbaseline shift    \tex_tbaseline shift:D
1164  \__kernel_primitive:NN \textbaselineshiftfactor \tex_textbaselineshiftfactor
1165   \tex_textbaselineshiftfactor:D
1166  \__kernel_primitive:NN \tfont               \tex_tfont:D
1167  \__kernel_primitive:NN \toucs              \tex_toucs:D
1168  \__kernel_primitive:NN \ucs                \tex_ucs:D
1169  \__kernel_primitive:NN \xkanjiskip         \tex_xkanjiskip:D
1170  \__kernel_primitive:NN \xspcode            \tex_xspcode:D
1171  \__kernel_primitive:NN \ybaseline shift    \tex_ybaseline shift:D
1172  \__kernel_primitive:NN \yoko               \tex_yoko:D
1173  \__kernel_primitive:NN \vfi                \tex_vfi:D

```

Primitives from upTeX.

```

1174  \__kernel_primitive:NN \currentcjktoken  \tex_currentcjktoken:D
1175  \__kernel_primitive:NN \disablecjktoken  \tex_disablecjktoken:D
1176  \__kernel_primitive:NN \enablecjktoken   \tex_enablecjktoken:D
1177  \__kernel_primitive:NN \forcecjktoken    \tex_forcecjktoken:D
1178  \__kernel_primitive:NN \kchar              \tex_kchar:D
1179  \__kernel_primitive:NN \kchardef          \tex_kchardef:D
1180  \__kernel_primitive:NN \kuten              \tex_kuten:D
1181  \__kernel_primitive:NN \uptexrevision    \tex_uptexrevision:D

```

```
1182  \__kernel_primitive:NN \uptexversion          \tex_uptexversion:D
```

Omega primitives provided by pTeX (listed separately mainly to allow understanding of their source).

```
1183  \__kernel_primitive:NN \odelcode           \tex_odelcode:D
1184  \__kernel_primitive:NN \odelimiter        \tex_odelimiter:D
1185  \__kernel_primitive:NN \omathaccent       \tex_omathaccent:D
1186  \__kernel_primitive:NN \omathchar         \tex_omathchar:D
1187  \__kernel_primitive:NN \omathchardef      \tex_omathchardef:D
1188  \__kernel_primitive:NN \omathcode         \tex_omathcode:D
1189  \__kernel_primitive:NN \oradical          \tex_oradical:D
```

Newer cross-engine primitives.

```
1190  \__kernel_primitive:NN \partokencontext    \tex_partokencontext:D
1191  \__kernel_primitive:NN \partokenname       \tex_partokenname:D
1192  \__kernel_primitive:NN \showstream         \tex_showstream:D
1193  \__kernel_primitive:NN \tracingstacklevels \tex_tracingstacklevels:D
```

End of the “just the names” part of the source.

```
1194 </names | package>
1195 </names | tex>
1196 <*package>
1197 <*tex>
```

The job is done: close the group (using the primitive renamed!).

```
1198 \tex_endgroup:D
```

L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> moves a few primitives, so these are sorted out. In newer versions of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>, `expl3` is loaded rather early, so only some primitives are already renamed, so we need two tests here. At the beginning of the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> format, the primitives `\end` and `\input` are renamed, and only later on the other ones.

```
1199 \tex_ifdefined:D \@@end
1200 \tex_let:D \tex_end:D                      \@@end
1201 \tex_let:D \tex_input:D                     \@@input
1202 \tex_fi:D
```

If `\@@@hyph` is defined, we are loading `expl3` in a pre-2020/10/01 release of L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>, so a few other primitives have to be tested as well.

```
1203 \tex_ifdefined:D \@@hyph
1204 \tex_let:D \tex_everydisplay:D             \frozen@everydisplay
1205 \tex_let:D \tex_everymath:D                \frozen@everymath
1206 \tex_let:D \tex_hyphen:D                  \@@hyph
1207 \tex_let:D \tex_italiccorrection:D       \@@italiccorr
1208 \tex_let:D \tex_underline:D               \@@underline
```

The `\shipout` primitive is particularly tricky as a number of packages want to hook in here. First, we see if a sufficiently-new kernel has saved a copy: if it has, just use that. Otherwise, we need to check each of the possible packages/classes that might move it: here, we are looking for those which do *not* delay action to the `\AtBeginDocument` hook. (We cannot use `\primitive` as that doesn’t allow us to make a direct copy of the primitive *itself*.) As we know that L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> is in use, we use it’s `\@tfor` loop here.

```
1209 \tex_ifdefined:D \@@shipout
1210   \tex_let:D \tex_shipout:D \@@shipout
1211 \tex_fi:D
1212 \tex_begingroup:D
1213   \tex_edef:D \I_tmpa_t1 { \tex_string:D \shipout }
```

```

1214     \tex_edef:D \l_tmpb_t1 { \tex_meaning:D \shipout }
1215     \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1216     \tex_else:D
1217     \tex_expandafter:D \otfor \tex_expandafter:D \otempa \tex_string:D :=
1218         \CROP@shipout
1219         \dup@shipout
1220         \GPTorg@shipout
1221         \LL@shipout
1222         \mem@oldshipout
1223         \open@shipout
1224         \pgfpages@originalshipout
1225         \pr@shipout
1226         \Shipout
1227         \verso@orig@shipout
1228         \do
1229         {
1230             \tex_edef:D \l_tmpb_t1
1231                 { \tex_expandafter:D \tex_meaning:D \otempa }
1232             \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1233                 \tex_global:D \tex_expandafter:D \tex_let:D
1234                     \tex_expandafter:D \tex_shipout:D \otempa
1235                     \tex_fi:D
1236                 }
1237             \tex_fi:D
1238         \tex_endgroup:D

```

Some tidying up is needed for `\(pdf)tracingfonts`. Newer LuaTeX has this simply as `\tracingfonts`, but that is overwritten by the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  kernel. So any spurious definition has to be removed, then the real version saved either from the pdfTeX name or from LuaTeX. In the latter case, we leave `\@@tracingfonts` available: this might be useful and almost all L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  users will have `expl3` loaded by `fontspec`. (We follow the usual kernel convention that `@@` is used for saved primitives.)

```

1239     \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1240     \tex_ifdefined:D \pdftracingfonts
1241         \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1242     \tex_else:D
1243         \tex_ifdefined:D \tex_directlua:D
1244             \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1245             \tex_let:D \tex_tracingfonts:D \@@tracingfonts
1246         \tex_fi:D
1247     \tex_fi:D
1248 \tex_fi:D

```

That is also true for the LuaTeX primitives under L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  (depending on the format-building date). There are a few primitives that get the right names anyway so are missing here!

```

1249 \tex_ifdefined:D \luatexsuppressfontnotfounderror
1250     \tex_let:D \tex_alignmark:D           \luatexalignmark
1251     \tex_let:D \tex_aligntab:D          \luatexaligntab
1252     \tex_let:D \tex_attribute:D        \luatexattribute
1253     \tex_let:D \tex_attributedef:D      \luatexattributedef
1254     \tex_let:D \tex_catcodetable:D     \luatexcatcodetable
1255     \tex_let:D \tex_clearmarks:D       \luatexclearmarks
1256     \tex_let:D \tex_crampeddisplaystyle:D \luatexcrampeddisplaystyle

```

```

1257 \tex_let:D \tex_crampedscriptscriptstyle:D           \luatexcrampedscriptscriptstyle
1258                                         \luatexcrampedscriptscriptstyle
1259 \tex_let:D \tex_crampedscriptstyle:D    \luatexcrampedscriptstyle
1260 \tex_let:D \tex_crampedtextstyle:D   \luatexcrampedtextstyle
1261 \tex_let:D \tex_fontid:D          \luatexfontid
1262 \tex_let:D \tex_formatname:D     \luatexformatname
1263 \tex_let:D \tex_gleaders:D      \luatexleaders
1264 \tex_let:D \tex_initcatcodetable:D \luatexinitcatcodetable
1265 \tex_let:D \tex_latelua:D       \luatexlatelua
1266 \tex_let:D \tex_luaescapestring:D \luatexluaescapestring
1267 \tex_let:D \tex_luafunction:D    \luatexluafunction
1268 \tex_let:D \tex_mathstyle:D     \luatexmathstyle
1269 \tex_let:D \tex_nokerns:D       \luatexnokerns
1270 \tex_let:D \tex_noligs:D        \luatexnoligs
1271 \tex_let:D \tex_outputbox:D     \luatexoutputbox
1272 \tex_let:D \tex_pageleftoffset:D \luatexpageleftoffset
1273 \tex_let:D \tex_pagetopoffset:D \luatexpagetopoffset
1274 \tex_let:D \tex_postexhyphenchar:D \luatexpostexhyphenchar
1275 \tex_let:D \tex_posthyphenchar:D \luatexposthyphenchar
1276 \tex_let:D \tex_preehyphenchar:D \luatexpreehyphenchar
1277 \tex_let:D \tex_prehyphenchar:D \luatexprehyphenchar
1278 \tex_let:D \tex_savecatcodetable:D \luatexsavecatcodetable
1279 \tex_let:D \tex_scantexttokens:D \luatexscantexttokens
1280 \tex_let:D \tex_suppressifcsnameerror:D \luatexsuppressifcsnameerror
1281                                         \luatexsuppressifcsnameerror
1282 \tex_let:D \tex_suppresslongerror:D   \luatexsuppresslongerror
1283 \tex_let:D \tex_suppressmathparerror:D \luatexsuppressmathparerror
1284                                         \luatexsuppressmathparerror
1285 \tex_let:D \tex_suppressoutererror:D \luatexsuppressoutererror
1286 \tex_let:D \tex_Uchar:D           \luatexUchar
1287 \tex_let:D \tex_suppressfontnotfounderror:D \luatexsuppressfontnotfounderror
1288                                         \luatexsuppressfontnotfounderror

```

Which also covers those slightly odd ones.

```

1289 \tex_let:D \tex_bodydir:D           \luatexbodydir
1290 \tex_let:D \tex_boxdir:D           \luatexboxdir
1291 \tex_let:D \tex_leftghost:D        \luatexleftghost
1292 \tex_let:D \tex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1293 \tex_let:D \tex_localinterlinepenalty:D \luatexlocalinterlinepenalty
1294                                         \luatexlocalinterlinepenalty
1295 \tex_let:D \tex_localeleftbox:D    \luatexlocaleleftbox
1296 \tex_let:D \tex_localrightbox:D   \luatexlocalrightbox
1297 \tex_let:D \tex_mathdir:D         \luatexmathdir
1298 \tex_let:D \tex_pagebottomoffset:D \luatexpagebottomoffset
1299 \tex_let:D \tex_pagedir:D         \luatexpagedir
1300 \tex_let:D \tex_pageheight:D      \luatexpageheight
1301 \tex_let:D \tex_pagerightoffset:D \luatexpagerightoffset
1302 \tex_let:D \tex_pagewidth:D       \luatexpagewidth
1303 \tex_let:D \tex_pardir:D          \luatexpardir
1304 \tex_let:D \tex_rightghost:D     \luatexrightghost
1305 \tex_let:D \tex_textdir:D         \luatextextdir
1306 \tex_fi:D

```

Only pdf $\mathrm{T}\bar{\mathrm{E}}\mathrm{X}$  and Lua $\mathrm{T}\bar{\mathrm{E}}\mathrm{X}$  define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1307 \tex_ifnum:D 0
1308   \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_hi:D
1309   \tex_ifdefined:D \tex_luatexversion:D 1 \tex_hi:D
1310   = 0 %
1311   \tex_let:D \tex_mapfile:D \tex_undefined:D
1312   \tex_let:D \tex_mapline:D \tex_undefined:D
1313 \tex_hi:D

```

A few packages do unfortunate things to date-related primitives.

```

1314 \tex_begingroup:D
1315   \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_time:D }
1316   \tex_edef:D \l_tmpb_t1 { \tex_string:D \time }
1317   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1318   \tex_else:D
1319     \tex_global:D \tex_let:D \tex_time:D \tex_undefined:D
1320   \tex_hi:D
1321   \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_day:D }
1322   \tex_edef:D \l_tmpb_t1 { \tex_string:D \day }
1323   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1324   \tex_else:D
1325     \tex_global:D \tex_let:D \tex_day:D \tex_undefined:D
1326   \tex_hi:D
1327   \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_month:D }
1328   \tex_edef:D \l_tmpb_t1 { \tex_string:D \month }
1329   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1330   \tex_else:D
1331     \tex_global:D \tex_let:D \tex_month:D \tex_undefined:D
1332   \tex_hi:D
1333   \tex_edef:D \l_tmpa_t1 { \tex_meaning:D \tex_year:D }
1334   \tex_edef:D \l_tmpb_t1 { \tex_string:D \year }
1335   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1336   \tex_else:D
1337     \tex_global:D \tex_let:D \tex_year:D \tex_undefined:D
1338   \tex_hi:D
1339 \tex_endgroup:D

```

Up to v0.80, LuaTE<sub>X</sub> defines the pdfTE<sub>X</sub> version data: rather confusing. Removing them means that \tex\_pdftexversion:D is a marker for pdfTE<sub>X</sub> alone: useful in engine-dependent code later.

```

1340 \tex_ifdefined:D \tex_luatexversion:D
1341   \tex_let:D \tex_pdftexbanner:D \tex_undefined:D
1342   \tex_let:D \tex_pdftexrevision:D \tex_undefined:D
1343   \tex_let:D \tex_pdftexversion:D \tex_undefined:D
1344 \tex_hi:D

```

cslatex moves a couple of primitives which we recover here; as there is no other marker, we can only work by looking for the names.

```

1345 \tex_ifdefined:D \orieveryjob
1346   \tex_let:D \tex_everyjob:D \orieveryjob
1347 \tex_hi:D
1348 \tex_ifdefined:D \oripdfoutput
1349   \tex_let:D \tex_pdfoutput:D \oripdfoutput
1350 \tex_hi:D

```

For ConTeXt, two tests are needed. Both Mark II and Mark IV move several primitives: these are all covered by the first test, again using \end as a marker. For Mark IV,

a few more primitives are moved: they are implemented using some Lua code in the current ConTeXt.

```

1351 \tex_ifdefined:D \normalend
1352   \tex_let:D \tex_end:D      \normalend
1353   \tex_let:D \tex_everyjob:D \normaleveryjob
1354   \tex_let:D \tex_input:D   \normalinput
1355   \tex_let:D \tex_language:D \normallanguage
1356   \tex_let:D \tex_mathop:D  \normalmathop
1357   \tex_let:D \tex_month:D   \normalmonth
1358   \tex_let:D \tex_outer:D   \normalouter
1359   \tex_let:D \tex_over:D    \normalover
1360   \tex_let:D \tex_vcenter:D \normalvcenter
1361   \tex_let:D \tex_unexpanded:D \normalunexpanded
1362   \tex_let:D \tex_expanded:D \normalexpanded
1363 \tex_fi:D
1364 \tex_ifdefined:D \normalitaliccorrection
1365   \tex_let:D \tex_hoffset:D   \normalhoffset
1366   \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1367   \tex_let:D \tex_voffset:D   \normalvoffset
1368   \tex_let:D \tex_showtokens:D \normalshowtokens
1369   \tex_let:D \tex_bodydir:D   \spac_directions_normal_body_dir
1370   \tex_let:D \tex_pagedir:D  \spac_directions_normal_page_dir
1371 \tex_fi:D
1372 \tex_ifdefined:D \normalleft
1373   \tex_let:D \tex_left:D     \normalleft
1374   \tex_let:D \tex_middle:D   \normalmiddle
1375   \tex_let:D \tex_right:D   \normalright
1376 \tex_fi:D
1377 </tex>

```

In LuaTeX, we additionally emulate some primitives using Lua code.

```
1378 {*lua}
```

\tex\_strcmp:D Compare two strings, expanding to 0 if they are equal, -1 if the first one is smaller and 1 if the second one is smaller. Here “smaller” refers to codepoint order which does not correspond to the user expected order for most non-ASCII strings.

```

1379 local minus_tok = token_new(string.byte'-', 12)
1380 local zero_tok = token_new(string.byte'0', 12)
1381 local one_tok = token_new(string.byte'1', 12)
1382 luacmd('tex_strcmp:D', function()
1383   local first = scan_string()
1384   local second = scan_string()
1385   if first < second then
1386     put_next(minus_tok, one_tok)
1387   else
1388     put_next(first == second and zero_tok or one_tok)
1389   end
1390 end, 'global')

```

(End definition for \tex\_strcmp:D. This function is documented on page ??.)

\tex\_Ucharcat:D Creating arbitrary chars using `tex.cprint`. The alternative approach using `token.new(...)` is about 10% slower but needed to create arbitrary space tokens.

```

1391 local sprint = tex.sprint
1392 local cprint = tex.cprint
1393 luacmd('tex_Ucharcat:D', function()
1394     local charcode = scan_int()
1395     local catcode = scan_int()
1396     if catcode == 10 then
1397         sprint(token_new(charcode, 10))
1398     else
1399         cprint(catcode, utf8_char(charcode))
1400     end
1401 end, 'global')

```

(End definition for \tex\_Ucharcat:D. This function is documented on page ??.)

\tex\_filesize:D Wrap the function from ltxutils.

```

1402 luacmd('tex_filesize:D', function()
1403     local size = filesize(scan_string())
1404     if size then write(size) end
1405 end, 'global')

```

(End definition for \tex\_filesize:D. This function is documented on page ??.)

\tex\_mdfivesum:D There are two cases: Either hash a file or a string. Both are already implemented in l3luatex or built-in.

```

1406 luacmd('tex_mdfivesum:D', function()
1407     local hash
1408     if scan_keyword"file" then
1409         hash = filemd5sum(scan_string())
1410     else
1411         hash = md5_HEX(scan_string())
1412     end
1413     if hash then write(hash) end
1414 end, 'global')

```

(End definition for \tex\_mdfivesum:D. This function is documented on page ??.)

\tex\_filemoddate:D A primitive for getting the modification date of a file.

```

1415 luacmd('tex_filemoddate:D', function()
1416     local date = filemoddate(scan_string())
1417     if date then write(date) end
1418 end, 'global')

```

(End definition for \tex\_filemoddate:D. This function is documented on page ??.)

\tex\_filedump:D An emulated primitive for getting a hexdump from a (partial) file. The length has a default of 0. This is consistent with pdfTEX, but it effectively makes the primitive useless without an explicit `length`. Therefore we allow the keyword `whole` to be used instead of a `length`, indicating that the whole remaining file should be read.

```

1419 luacmd('tex_filedump:D', function()
1420     local offset = scan_keyword'offset' and scan_int() or nil
1421     local length = scan_keyword'length' and scan_int()
1422             or not scan_keyword'whole' and 0 or nil
1423     local data = filedump(scan_string(), offset, length)
1424     if data then write(data) end
1425 end, 'global')

```

(End definition for \tex\_filedump:D. This function is documented on page ??.)

1426 ⟨/lua⟩

1427 ⟨/package⟩

# Chapter 40

## I3kernel-functions: kernel-reserved functions

### 40.1 Internal kernel functions

---

\\_\\_kernel\\_chk\\_cs\\_exist:N \\_\\_kernel\\_chk\\_cs\\_exist:N *<cs>*

---

\\_\\_kernel\\_chk\\_cs\\_exist:c This function is only created if debugging is enabled. It checks that *<cs>* exists according to the criteria for \cs\_if\_exist:p:N, and if not raises a kernel-level error.

---

\\_\\_kernel\\_chk\\_defined:NT \\_\\_kernel\\_chk\\_defined:NT *<variable>* {\i<true code>}

If *<variable>* is not defined (according to \cs\_if\_exist:NTF), this triggers an error, otherwise the *<true code>* is run.

---

\\_\\_kernel\\_chk\\_expr:nNnN \\_\\_kernel\\_chk\\_expr:nNnN {\i<expr>} *<eval>* {\i<convert>} *<caller>*

This function is only created if debugging is enabled. By default it is equivalent to \use\_i:nnnn. When expression checking is enabled, it leaves in the input stream the result of \tex\_the:D *<eval>* *<expr>* \tex\_relax:D after checking that no token was left over. If any token was not taken as part of the expression, there is an error message displaying the result of the evaluation as well as the *<caller>*. For instance *<eval>* can be \\_\\_int\_eval:w and *<caller>* can be \int\_eval:n or \int\_set:Nn. The argument *<convert>* is empty except for mu expressions where it is \tex\_mutoglu:D, used for internal purposes.

---

\\_\\_kernel\\_chk\\_tl\\_type:NnnT \\_\\_kernel\\_chk\\_tl\\_type:NnnT *<control sequence>* {\i<specific type>} {\i<reconstruction>} {\i<true code>}

Helper to test that the *<control sequence>* is a variable of the given *<specific type>* of token list. Produces suitable error messages if the *<control sequence>* does not exist, or if it is not a token list variable at all, or if the *<control sequence>* differs from the result of x-expanding *<reconstruction>*. If all of these tests succeed then the *<true code>* is run.

---

```
\_\_kernel\_codepoint\_to\_bytes:n * \_\_kernel\_codepoint\_to\_bytes:n {\langle codepoint\rangle}
```

Converts the  $\langle codepoint \rangle$  to UTF-8 bytes. The expansion of this function comprises four brace groups, each of which will contain a hexadecimal value: the appropriate byte. As UTF-8 is a variable-length, one or more of the groups may be empty: the bytes read in the logical order, such that a two-byte codepoint will have groups #1 and #2 filled and #3 and #4 empty.

---

```
\_\_kernel\_cs\_parm\_from\_arg\_count:nnF \_\_kernel\_cs\_parm\_from\_arg\_count:nnF {\langle follow-on \rangle} {\langle args \rangle}  
{\langle false code \rangle}
```

Evaluates the number of  $\langle args \rangle$  and leaves the  $\langle follow-on \rangle$  code followed by a brace group containing the required number of primitive parameter markers (#1, *etc.*). If the number of  $\langle args \rangle$  is outside the range [0, 9], the  $\langle false code \rangle$  is inserted *instead* of the  $\langle follow-on \rangle$ .

---

```
\_\_kernel\_dependency\_version\_check:Nn \_\_kernel\_dependency\_version\_check:Nn {\langle \date \rangle} {\langle file \rangle}  
\_\_kernel\_dependency\_version\_check:nn \_\_kernel\_dependency\_version\_check:nn {\langle date \rangle} {\langle file \rangle}
```

Checks if the loaded version of the expl3 kernel is at least  $\langle date \rangle$ , required by  $\langle file \rangle$ . If the kernel date is older than  $\langle date \rangle$ , the loading of  $\langle file \rangle$  is aborted and an error is raised.

---

```
\_\_kernel\_deprecation\_code:nn \_\_kernel\_deprecation\_code:nn {\langle error code \rangle} {\langle working code \rangle}
```

Stores both an  $\langle error \rangle$  and  $\langle working \rangle$  definition for given material such that they can be exchanged by `\debug_on:` and `\debug_off:`.

---

```
\_\_kernel\_exp\_not:w * \_\_kernel\_exp\_not:w {\langle expandable tokens \rangle} {\langle content \rangle}
```

Carries out expansion on the  $\langle expandable tokens \rangle$  before preventing further expansion of the  $\langle content \rangle$  as for `\exp_not:n`. Typically, the  $\langle expandable tokens \rangle$  will alter the nature of the  $\langle content \rangle$ , *i.e.* allow it to be generated in some way.

`\l_kernel_expl_bool` A boolean which records the current code syntax status: `true` if currently inside a code environment. This variable should only be set by `\ExplSyntaxOn`/`\ExplSyntaxOff`.

(*End definition for \l\_kernel\_expl\_bool.*)

`\c_kernel_expl_date_tl` A token list containing the release date of the l3kernel preloaded in L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> used to check if dependencies match.

(*End definition for \c\_kernel\_expl\_date\_tl.*)

---

```
\_\_kernel\_file\_missing:n \_\_kernel\_file\_missing:n {\langle name \rangle}
```

Expands the  $\langle name \rangle$  as per `\_\_kernel_file_name_sanitize:n` then produces an error message indicating that this file was not found.

---

```
\_\_kernel\_file\_name\_sanitize:n * \_\_kernel\_file\_name\_sanitize:n {\langle name \rangle}
```

---

Updated: 2021-04-17

Expands the file name using a `\csname`-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using `\ifcsname`. This is less conservative than the token-by-token approach used before, but it is much faster.

---

```
\__kernel_file_input_push:n \__kernel_file_input_push:n {\<name>}
\__kernel_file_input_pop: \__kernel_file_input_pop:
```

Used to push and pop data from the internal file stack: needed only in package mode, where interfacing with the L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> kernel is necessary.

---

```
\__kernel_int_add:nnn * \__kernel_int_add:nnn {\<integer1>} {\<integer2>} {\<integer3>}
```

Expands to the result of adding the three *<integers>* (which must be suitable input for `\int_eval:w`), avoiding intermediate overflow. Overflow occurs only if the overall result is outside  $[-2^{31}+1, 2^{31}-1]$ . The *<integers>* may be of the form `\int_eval:w ... \scan_stop:` but may be evaluated more than once.

---

```
\__kernel_intarray_gset:Nnn \__kernel_intarray_gset:Nnn {\<intarray var>} {\<index>} {\<value>}
```

New: 2018-03-31

Faster version of `\intarray_gset:Nnn`. Stores the *<value>* into the *<integer array variable>* at the *<position>*. The *<index>* and *<value>* must be suitable for a direct assignment to a T<sub>E</sub>X count register, for instance expanding to an integer denotation or obtained through the primitive `\numexpr` (which may be un-terminated). No bound checking is performed: the caller is responsible for ensuring that the *<position>* is between 1 and the `\intarray_count:N`, and the *<value>*'s absolute value is at most  $2^{30}-1$ . Assignments are always global.

---

```
\__kernel_intarray_item:Nn * \__kernel_intarray_item:Nn {\<intarray var>} {\<index>}
```

New: 2018-03-31

Faster version of `\intarray_item:Nn`. Expands to the integer entry stored at the *<index>* in the *<integer array variable>*. The *<index>* must be suitable for a direct assignment to a T<sub>E</sub>X count register and must be between 1 and the `\intarray_count:N`, lest a low-level T<sub>E</sub>X error occur.

---

```
\__kernel_intarray_range_to_clist:Nnn \__kernel_intarray_range_to_clist:Nnn {\<intarray var>} {\<start index>} {\<end index>}
```

New: 2020-07-12

Converts to integer denotations separated by commas the entries of the *<intarray>* from positions *<start index>* to *<end index>* included. The *<start index>* and *<end index>* must be suitable for a direct assignment to a T<sub>E</sub>X count register, must be between 1 and the `\intarray_count:N`, and be suitably ordered. All tokens have category code other.

---

```
\__kernel_intarray_gset_range_from_clist:Nnn \__kernel_intarray_gset_range_from_clist:Nnn {\<intarray var>} {\<start index>} {\<integer clist>}
```

New: 2020-07-12

Stores the entries of the *<clist>* as entries of the *<intarray var>* starting from the *<start index>*, upwards. This is done without any bound checking. The *<start index>* and all entries of the *<integer comma list>* (which do not undergo space trimming and brace stripping as in normal clist mappings) must be suitable for a direct assignment to a T<sub>E</sub>X count register. An empty entry may stop the loop.

---

`\_\_kernel\_ior\_open:Nn` `\_\_kernel\_ior\_open:Nn` *⟨stream⟩ {⟨file name⟩}*  
`\_\_kernel\_ior\_open:No`

This function has identical syntax to the public version. However, it does not take precautions against active characters in the *⟨file name⟩*, and it does not attempt to add a *⟨path⟩* to the *⟨file name⟩*: it is therefore intended to be used by higher-level functions which have already fully expanded the *⟨file name⟩* and which need to perform multiple open or close operations. See for example the implementation of `\file_get_full_name:nN`,

---

`\_\_kernel\_iow\_with:Nnn` `\_\_kernel\_iow\_with:Nnn` *⟨integer⟩ {⟨value⟩} {⟨code⟩}*

If the *⟨integer⟩* is equal to the *⟨value⟩* then this function simply runs the *⟨code⟩*. Otherwise it saves the current value of the *⟨integer⟩*, sets it to the *⟨value⟩*, runs the *⟨code⟩*, and restores the *⟨integer⟩* to its former value. This is used to ensure that the `\newlinechar` is 10 when writing to a stream, which lets `\iow_newline:` work, and that `\errorcontextlines` is  $-1$  when displaying a message.

`\_\_kernel_kern:n` `\_\_kernel_kern:n` *{⟨length⟩}*  
Inserts a kern of the specified *⟨length⟩*, a dimension expression.

(End definition for `\_\_kernel_kern:n`.)

`\g_kernel_prg_map_int` This integer is used by non-expandable mapping functions to track the level of nesting in force. The functions `\⟨type⟩_map_1:w`, `\⟨type⟩_map_2:w`, etc., labelled by `\g_kernel_prg_map_int` hold functions to be mapped over various list datatypes in inline and variable mappings.

(End definition for `\g_kernel_prg_map_int`.)

---

`\_\_kernel_quark_new_test:N \_\_kernel_quark_new_test:N \langle name\rangle:\langle arg spec\rangle`

Defines a quark-test function `\langle name\rangle:\langle arg spec\rangle` which tests if its argument is `\q\_\_<namespace>\recursion_tail`, then acts accordingly, as described below for each possible `\langle arg spec\rangle`.

The `\langle namespace\rangle` is determined as the first (nonempty) `_`-delimited word in `\langle name\rangle` and is used internally in the definition of auxiliaries. The function `\_\_kernel_quark_new_test:N` does *not* define the `\q\_\_<namespace>\recursion_tail` and `\q\_\_<namespace>\recursion_stop` quarks. They should be manually defined with `\quark_new:N`.

There are 6 different types of quark-test functions. Which one is defined depends on the `\langle arg spec\rangle`, which *must* be one of the options listed now. Four of them are modeled after `\quark_if_recursion_tail:(N|n)` and `\quark_if_recursion_tail_do:(N|n)n`.

`n` defines `\langle name\rangle:n` such that it checks if #1 contains only `\q\_\_<namespace>\recursion_tail`, and if so consumes all tokens up to `\q\_\_<namespace>\recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:n`).

`nn` defines `\langle name\rangle:nn` such that it checks if #1 contains only `\q\_\_<namespace>\recursion_tail`, and if so consumes all tokens up to `\q\_\_<namespace>\recursion_stop`, then executes the code #2 after that (*c.f.* `\quark_if_recursion_tail_stop_do:nn`).

`N` defines `\langle name\rangle:N` such that it checks if #1 is `\q\_\_<namespace>\recursion_tail`, and if so consumes all tokens up to `\q\_\_<namespace>\recursion_stop` (*c.f.* `\quark_if_recursion_tail_stop:N`).

`Nn` defines `\langle name\rangle:Nn` such that it checks if #1 is `\q\_\_<namespace>\recursion_tail`, and if so consumes all tokens up to `\q\_\_<namespace>\recursion_stop`, then executes the code #2 after that (*c.f.* `\quark_if_recursion_tail_stop_do:Nn`).

The last two are modeled after `\quark_if_recursion_tail_break:(n|N)N`, and in those cases the quark `\q\_\_<namespace>\recursion_stop` is not used (and thus needs not be defined).

`nN` defines `\langle name\rangle:nN` such that it checks if #1 contains only `\q\_\_<namespace>\recursion_tail`, and if so uses the `\langle type\rangle_map_break:` function #2.

`NN` defines `\langle name\rangle:NN` such that it checks if #1 is `\q\_\_<namespace>\recursion_tail`, and if so uses the `\langle type\rangle_map_break:` function #2.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

---

`\_\_kernel_quark_new_conditional:Nn \_\_kernel_quark_new_conditional:Nn  
  \_\_<namespace>\_quark_if_<name>:<arg spec> {<conditions>}`

Defines a collection of quark conditionals that test if their argument is the quark  $\backslash q\_<namespace>\_<name>$  and perform suitable actions. The  $\langle conditions \rangle$  are a comma-separated list of one or more of p, T, F, and TF, and one conditional is defined for each  $\langle condition \rangle$  in the list, as described for `\prg_new_conditional:Npnn`. The conditionals are defined using `\prg_new_conditional:Npnn`, so that their name is obtained by adding p, T, F, or TF to the base name `\_\_<namespace>\_quark_if_<name>:<arg spec>`.

The first argument of `\_\_kernel_quark_new_conditional:Nn` must contain `_quark_if_` and `:`, as these markers are used to determine the  $\langle name \rangle$  of the quark  $\backslash q\_<namespace>\_<name>$  to be tested. This quark should be manually defined with `\quark_new:N`, as `\_\_kernel_quark_new_conditional:Nn` does *not* define it.

The function `\_\_kernel_quark_new_conditional:Nn` can define 2 different types of quark conditionals. Which one is defined depends on the  $\langle arg spec \rangle$ , which *must* be one of the following options, modeled after `\quark_if_nil:(N|n)(TF)`.

`n` defines `\_\_<namespace>\_quark_if_<name>:&n(TF)` such that it checks if #1 contains only  $\backslash q\_<namespace>\_<name>$ , and executes the proper conditional branch.

`N` defines `\_\_<namespace>\_quark_if_<name>:&N(TF)` such that it checks if #1 is  $\backslash q\_<namespace>\_<name>$ , and executes the proper conditional branch.

Any other signature, as well as a function without signature are errors, and in such case the definition is aborted.

`\c_\_kernel_randint_max_int` Maximal allowed argument to `\_\_kernel_randint:n`. Equal to  $2^{17} - 1$ .

(End definition for `\c_\_kernel_randint_max_int`.)

---

`\_\_kernel_randint:n \_\_kernel_randint:n {<max>}`

Used in an integer expression this gives a pseudo-random number between 1 and  $\langle max \rangle$  included. One must have  $\langle max \rangle \leq 2^{17} - 1$ . The  $\langle max \rangle$  must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent).

---

`\_\_kernel_randint:nn \_\_kernel_randint:nn {<min>} {<max>}`

Used in an integer expression this gives a pseudo-random number between  $\langle min \rangle$  and  $\langle max \rangle$  included. The  $\langle min \rangle$  and  $\langle max \rangle$  must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges  $R = \langle max \rangle - \langle min \rangle + 1 \leq 2^{17} - 1$ ,  $\langle min \rangle - 1 + \_\_kernel_randint:n\{R\}$  is faster.

---

`\_\_kernel_register_show:N \_\_kernel_register_show:N <register>`

`\_\_kernel_register_show:c` Used to show the contents of a TeX register at the terminal, formatted such that internal parts of the mechanism are not visible.

---

`\_\_kernel_register_log:N \_\_kernel_register_log:N <register>`

`\_\_kernel_register_log:c` Used to write the contents of a TeX register to the log file in a form similar to `\_\_kernel_register_show:N`.

---

```
\_\_kernel\_str\_to\_other:n ★ \_\_kernel\_str\_to\_other:n {⟨token list⟩}
```

Converts the ⟨*token list*⟩ to a ⟨*other string*⟩, where spaces have category code “other”. This function can be f-expanded without fear of losing a leading space, since spaces do not have category code 10 in its result. It takes a time quadratic in the character count of the string.

---

```
\_\_kernel\_str\_to\_other\_fast:n ★ \_\_kernel\_str\_to\_other\_fast:n {⟨token list⟩}
```

Same behaviour \\_\\_kernel\\_str\\_to\\_other:n but only restricted-expandable. It takes a time linear in the character count of the string.

---

```
\_\_kernel\_tl\_to\_str:w ★ \_\_kernel\_tl\_to\_str:w {⟨expandable tokens⟩} {⟨tokens⟩}
```

Carries out expansion on the ⟨*expandable tokens*⟩ before conversion of the ⟨*tokens*⟩ to a string as describe for \tl\_to\_str:n. Typically, the ⟨*expandable tokens*⟩ will alter the nature of the ⟨*tokens*⟩, *i.e.* allow it to be generated in some way. This function requires only a single expansion.

---

```
\_\_kernel\_tl\_set:Nx \_\_kernel\_tl\_set:Nx {⟨tl var⟩} {⟨tokens⟩}
```

---

```
\_\_kernel\_tl\_gset:Nx
```

Fully expands ⟨*tokens*⟩ and assigns the result to ⟨*tl var*⟩. ⟨*tokens*⟩ must be given in braces and there must be no token between ⟨*tl var*⟩ and ⟨*tokens*⟩.

---

```
\_\_kernel\_codepoint\_data:nn ★ \_\_kernel\_codepoint\_data:nn {⟨type⟩} {⟨codepoint⟩}
```

Expands to the appropriate value for the ⟨*type*⟩ of data requested for a ⟨*codepoint*⟩. The current list of ⟨*types*⟩ and results are

**lowercase** The *single* codepoint specified by `UnicodeData.txt` for lowercase mapping of the codepoint: will be equal to the input ⟨*codepoint*⟩ if there is no mapping specified in `UnicodeData.txt`

**uppercase** The *single* codepoint specified by `UnicodeData.txt` for uppercase mapping of the codepoint: will be equal to the input ⟨*codepoint*⟩ if there is no mapping specified in `UnicodeData.txt`

---

```
\_\_kernel\_codepoint\_case:nn ★ \_\_kernel\_codepoint\_case:nn {⟨mapping⟩} {⟨codepoint⟩}
```

Expands to a list of three balanced text, of which at least the first will contain a codepoint. This list of up to three codepoints specifies the full case mapping for the input ⟨*codepoint*⟩. The ⟨*mapping*⟩ should be one of

- `casifold`
- `lowercase`
- `titlecase`
- `uppercase`

## 40.2 Kernel backend functions

These functions are required to pass information to the backend. The nature of these means that they are defined only when the relevant backend is in use.

---

```
\__kernel_backend_literal:n \__kernel_backend_literal:n {<content>}
\__kernel_backend_literal:(e|x)
```

---

Adds the *<content>* literally to the current vertical list as a whatsit. The nature of the *<content>* will depend on the backend in use.

---

```
\__kernel_backend_literal_postscript:n \__kernel_backend_literal_postscript:n {<PostScript>}
\__kernel_backend_literal_postscript:x
```

---

Adds the *<PostScript>* literally to the current vertical list as a whatsit. No positioning is applied.

---

```
\__kernel_backend_literal_pdf:n \__kernel_backend_literal_pdf:n {<PDF instructions>}
\__kernel_backend_literal_pdf:x
```

---

Adds the *<PDF instructions>* literally to the current vertical list as a whatsit. No positioning is applied.

---

```
\__kernel_backend_literal_svg:n \__kernel_backend_literal_svg:n {<SVG instructions>}
\__kernel_backend_literal_svg:x
```

---

Adds the *<SVG instructions>* literally to the current vertical list as a whatsit. No positioning is applied.

---

```
\__kernel_backend_postscript:n \__kernel_backend_postscript:n {<PostScript>}
\__kernel_backend_postscript:x
```

---

Adds the *<PostScript>* to the current vertical list as a whatsit. The PostScript reference point is adjusted to match the current position. The PostScript is inserted inside a SDict begin/end pair.

---

```
\__kernel_backend_align_begin: \__kernel_backend_align_begin:
\__kernel_backend_align_end:  <PostScript literals>
\__kernel_backend_align_end:
```

---

Arranges to align the PostScript and DVI current positions and scales.

---

```
\__kernel_backend_scope_begin: \__kernel_backend_scope_begin:
\__kernel_backend_scope_end:  <content>
\__kernel_backend_scope_end:
```

---

Creates a scope for instructions at the backend level.

---

```
\__kernel_backend_matrix:n \__kernel_backend_matrix:n {<matrix>}
\__kernel_backend_matrix:x
```

---

Applies the *<matrix>* to the current transformation matrix.

---

```
\g__kernel_backend_header_bool
```

---

Specifies whether to write headers for the backend.

---

`\l_kernel_color_stack_int` The color stack used in pdfTeX and LuaTeX for the main color.

# Chapter 41

## I3basics implementation

1428 ⟨\*package⟩

### 41.1 Renaming some TeX primitives (again)

Having given all the TeX primitives a consistent name, we need to give sensible names to the ones we actually want to use. These will be defined as needed in the appropriate modules, but we do a few now, just to get started.<sup>8</sup>

\if\_true: Then some conditionals.  
\if\_false:  
  \or:  
  \else:  
  \fi:  
\reverse\_if:N  
  \if:w  
\if\_charcode:w  
\if\_catcode:w  
\if\_meaning:w  
1429 \tex\_let:D \if\_true:                   \tex\_iftrue:D  
1430 \tex\_let:D \if\_false:                   \tex\_iffalse:D  
1431 \tex\_let:D \or:                       \tex\_or:D  
1432 \tex\_let:D \else:                      \tex\_else:D  
1433 \tex\_let:D \fi:                       \tex\_fi:D  
1434 \tex\_let:D \reverse\_if:N               \tex\_unless:D  
1435 \tex\_let:D \if:w                       \tex\_if:D  
1436 \tex\_let:D \if\_charcode:w             \tex\_if:D  
1437 \tex\_let:D \if\_catcode:w               \tex\_ifcat:D  
1438 \tex\_let:D \if\_meaning:w             \tex\_ifx:D  
1439 \tex\_let:D \if\_bool:N                   \tex\_ifodd:D

(End definition for \if\_true: and others. These functions are documented on page 27.)

\if\_mode\_math: TeX lets us detect some if its modes.  
\if\_mode\_horizontal:  
  1440 \tex\_let:D \if\_mode\_math:           \tex\_ifemode:D  
\if\_mode\_vertical:  
  1441 \tex\_let:D \if\_mode\_horizontal: \tex\_ifhmode:D  
\if\_mode\_inner:  
  1442 \tex\_let:D \if\_mode\_vertical:   \tex\_ifvmode:D  
  1443 \tex\_let:D \if\_mode\_inner:       \tex\_ifinner:D

(End definition for \if\_mode\_math: and others. These functions are documented on page 28.)

\if\_cs\_exist:N Building csnames and testing if control sequences exist.  
\if\_cs\_exist:w  
  \cs:w  
  \cs\_end:  
1444 \tex\_let:D \if\_cs\_exist:N           \tex\_ifdefined:D  
1445 \tex\_let:D \if\_cs\_exist:w           \tex\_ifcsname:D  
1446 \tex\_let:D \cs:w                   \tex\_csname:D  
1447 \tex\_let:D \cs\_end:               \tex\_endcsname:D

---

<sup>8</sup>This renaming gets expensive in terms of csname usage, an alternative scheme would be to just use the \tex\_...:D name in the cases where no good alternative exists.

(End definition for `\if_cs_exist:N` and others. These functions are documented on page 28.)

`\exp_after:wN` The five `\exp_` functions are used in the `l3expan` module where they are described.

```
1448 \tex_let:D \exp_after:wN      \tex_expandafter:D  
1449 \tex_let:D \exp_not:N       \tex_noexpand:D  
1450 \tex_let:D \exp_not:n      \tex_unexpanded:D  
1451 \tex_let:D \exp:w        \tex_roman numeral:D  
1452 \tex_chardef:D \exp_end: = 0 ~
```

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 38.)

`\token_to_meaning:N` Examining a control sequence or token.

```
1453 \tex_let:D \token_to_meaning:N \tex_meaning:D  
1454 \tex_let:D \cs_meaning:N     \tex_meaning:D
```

(End definition for `\token_to_meaning:N` and `\cs_meaning:N`. These functions are documented on page 187.)

`\tl_to_str:n` Making strings.

```
1455 \tex_let:D \tl_to_str:n      \tex_detokenize:D  
1456 \tex_let:D \token_to_str:N    \tex_string:D  
1457 \tex_let:D \__kernel_tl_to_str:w \tex_detokenize:D
```

(End definition for `\tl_to_str:n`, `\token_to_str:N`, and `\__kernel_tl_to_str:w`. These functions are documented on page 108.)

`\scan_stop:` The next three are basic functions for which there also exist versions that are safe inside alignments. These safe versions are defined in the `l3prg` module.

```
1458 \tex_let:D \scan_stop:      \tex_relax:D  
1459 \tex_let:D \group_begin:    \tex_begingroup:D  
1460 \tex_let:D \group_end:     \tex_endgroup:D
```

(End definition for `\scan_stop:`, `\group_begin:`, and `\group_end:`. These functions are documented on page 13.)

```
1461 <@=int>
```

`\if_int_compare:w` For integers.

```
1462 \tex_let:D \if_int_compare:w   \tex_ifnum:D  
1463 \tex_let:D \__int_to_roman:w  \tex_roman numeral:D
```

(End definition for `\if_int_compare:w` and `\__int_to_roman:w`. This function is documented on page 168.)

`\group_insert_after:N` Adding material after the end of a group.

```
1464 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 14.)

`\exp_args:Nc` Discussed in `l3expan`, but needed much earlier.

```
1465 \tex_long:D \tex_def:D \exp_args:Nc #1#2  
1466 { \exp_after:wN #1 \cs:w #2 \cs_end: }  
1467 \tex_long:D \tex_def:D \exp_args:cc #1#2  
1468 { \cs:w #1 \exp_after:wN \cs_end: \cs:w #2 \cs_end: }
```

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 34.)

`\token_to_meaning:c` A small number of variants defined by hand. Some of the necessary functions (`\use_i:nn`, `\use_ii:nn`, and `\exp_args:Nc`) are not defined at that point yet, but will be defined before those variants are used. The `\cs_meaning:c` command must check for an undefined control sequence to avoid defining it mistakenly.

```

1469 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
1470 \tex_long:D \tex_def:D \cs_meaning:c #1
1471 {
1472   \if_cs_exist:w #1 \cs_end:
1473     \exp_after:wN \use_i:nn
1474   \else:
1475     \exp_after:wN \use_ii:nn
1476   \fi:
1477   { \exp_args:Nc \cs_meaning:N {#1} }
1478   { \tl_to_str:n {undefined} }
1479 }
1480 \tex_let:D \token_to_meaning:c = \cs_meaning:c

```

(End definition for `\token_to_meaning:N`. This function is documented on page 187.)

## 41.2 Defining some constants

`\c_zero_int` We need the constant `\c_zero_int` which is used by some functions in the `I3alloc` module. The rest are defined in the `I3int` module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the `I3int` module is required but it can't be used until the allocation has been set up properly!

```
1481 \tex_chardef:D \c_zero_int = 0 ~
```

(End definition for `\c_zero_int`. This variable is documented on page 167.)

`\c_max_register_int` This is here as this particular integer is needed both in package mode and to bootstrap `I3alloc`, and is documented in `I3int`. `LuaTeX` and those which contain parts of the Omega extensions have more registers available than `ε-TEX`.

```

1482 \tex_ifdefined:D \tex_luatexversion:D
1483   \tex_chardef:D \c_max_register_int = 65 535 ~
1484 \tex_else:D
1485   \tex_ifdefined:D \tex_omathchardef:D
1486     \tex_omathchardef:D \c_max_register_int = 65535 ~
1487   \tex_else:D
1488     \tex_mathchardef:D \c_max_register_int = 32767 ~
1489   \tex_fi:D
1490 \tex_fi:D

```

(End definition for `\c_max_register_int`. This variable is documented on page 167.)

## 41.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

All assignment functions in `LATEX3` should be naturally protected; after all, the `TEX` primitives for assignments are and it can be a cause of problems if others aren't.

```

1491 \tex_let:D \cs_set_nopar:Npn          \tex_def:D
1492 \tex_let:D \cs_set_nopar:Npx          \tex_eodef:D

```

```

\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npx

```

```

1493 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
1494   { \tex_long:D \tex_def:D }
1495 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
1496   { \tex_long:D \tex_edef:D }
1497 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
1498   { \tex_protected:D \tex_def:D }
1499 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
1500   { \tex_protected:D \tex_edef:D }
1501 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
1502   { \tex_protected:D \tex_long:D \tex_def:D }
1503 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
1504   { \tex_protected:D \tex_long:D \tex_edef:D }

```

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 16.)

`\cs_gset_nopar:Npn` Global versions of the above functions.

```

\cs_gset_nopar:Npx
\cs_gset_nopar:Npx
\cs_gset:Npn
\cs_gset:Npx
\cs_gset_protected_nopar:Npn
\cs_gset_protected_nopar:Npx
\cs_gset_protected:Npn
\cs_gset_protected:Npx
1505 \tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
1506 \tex_let:D \cs_gset_nopar:Npx          \tex_xdef:D
1507 \cs_set_protected:Npn \cs_gset:Npn
1508   { \tex_long:D \tex_gdef:D }
1509 \cs_set_protected:Npn \cs_gset:Npx
1510   { \tex_long:D \tex_xdef:D }
1511 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
1512   { \tex_protected:D \tex_gdef:D }
1513 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
1514   { \tex_protected:D \tex_xdef:D }
1515 \cs_set_protected:Npn \cs_gset_protected:Npn
1516   { \tex_protected:D \tex_long:D \tex_gdef:D }
1517 \cs_set_protected:Npn \cs_gset_protected:Npx
1518   { \tex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 16.)

## 41.4 Selecting tokens

1519 `<@=exp>`

`\l__exp_internal_tl` Scratch token list variable for `\b3expan`, used by `\use:x`, used in defining conditionals. We don't use `t1` methods because `\b3asics` is loaded earlier.

```
1520 \cs_set_nopar:Npn \l__exp_internal_tl { }
```

(End definition for `\l__exp_internal_tl`.)

`\use:c` This macro grabs its argument and returns a csname from it.

```
1521 \cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }
```

(End definition for `\use:c`. This function is documented on page 20.)

`\use:x` Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which we've set up above.

```

1522 \cs_set_protected:Npn \use:x #1
1523   {
1524     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
1525     \l__exp_internal_tl
1526   }

```

(End definition for `\use:x`. This function is documented on page 25.)

1527 ⟨`@@=use`⟩

`\use:e` In non-LuaTeXengines older than 2019, `\expanded` is emulated.

```
1528 \cs_set:Npn \use:e #1 { \tex_expanded:D {#1} }
1529 \tex_ifdefined:D \tex_expanded:D \tex_else:D
1530   \cs_set:Npn \use:e #1 { \exp_args:Ne \use:n {#1} }
1531 \tex_if:D
```

(End definition for `\use:e`. This function is documented on page 25.)

1532 ⟨`@@=exp`⟩

`\use:n` These macros grab their arguments and return them back to the input (with outer braces removed).

```
1533 \cs_set:Npn \use:n    #1      {#1}
1534 \cs_set:Npn \use:nn   #1#2     {#1#2}
1535 \cs_set:Npn \use:nnn  #1#2#3   {#1#2#3}
1536 \cs_set:Npn \use:nnnn #1#2#3#4 {#1#2#3#4}
```

(End definition for `\use:n` and others. These functions are documented on page 23.)

`\use_i:nn` The equivalent to L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub>'s `\@firstoftwo` and `\@secondoftwo`.

```
1537 \cs_set:Npn \use_i:nn #1#2 {#1}
1538 \cs_set:Npn \use_i:nn #1#2 {#2}
```

(End definition for `\use_i:nn` and `\use_i:nn`. These functions are documented on page 24.)

`\use_i:nnn` We also need something for picking up arguments from a longer list.

```
1539 \cs_set:Npn \use_i:nnn #1#2#3 {#1}
1540 \cs_set:Npn \use_i:nnn #1#2#3 {#2}
1541 \cs_set:Npn \use_iii:nnn #1#2#3 {#3}
1542 \cs_set:Npn \use_i_ii:nnn #1#2#3 {#1#2}
1543 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}
1544 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#2}
1545 \cs_set:Npn \use_iii:nnnn #1#2#3#4 {#3}
1546 \cs_set:Npn \use_iv:nnnn #1#2#3#4 {#4}
```

(End definition for `\use_i:nnn` and others. These functions are documented on page 24.)

`\use_ii_i:nn`

```
1547 \cs_set:Npn \use_ii_i:nn #1#2 { #2 #1 }
```

(End definition for `\use_ii_i:nn`. This function is documented on page 24.)

`\use_none_delimit_by_q_nil:w` Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
1548 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
1549 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
1550 \cs_set:Npn \use_none_delimit_by_q_recursion_stop:w #1 \q_recursion_stop { }
```

(End definition for `\use_none_delimit_by_q_nil:w`, `\use_none_delimit_by_q_stop:w`, and `\use_none_delimit_by_q_recursion_stop:w`. These functions are documented on page 25.)

```
\use_i_delimit_by_q_nil:nw  
\use_i_delimit_by_q_stop:nw  
\use_i_delimit_by_q_recursion_stop:nw
```

Same as above but execute first argument after gobbling. Very useful when you need to skip the rest of a mapping sequence but want an easy way to control what should be expanded next.

```
1551 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}  
1552 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}  
1553 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw  
1554 #1#2 \q_recursion_stop {#1}
```

(End definition for `\use_i_delimit_by_q_nil:nw`, `\use_i_delimit_by_q_stop:nw`, and `\use_i_delimit_by_q_recursion_stop:nw`. These functions are documented on page 25.)

## 41.5 Gobbling tokens from input

```
\use_none:n  
\use_none:nn  
\use_none:nnn  
\use_none:nnnn  
\use_none:nnnnn  
\use_none:nnnnnn  
\use_none:nnnnnnn  
\use_none:nnnnnnnn
```

To gobble tokens from the input we use a standard naming convention: the number of tokens gobbled is given by the number of n's following the : in the name. Although we could define functions to remove ten arguments or more using separate calls of `\use_none:nnnnn`, this is very non-intuitive to the programmer who will assume that expanding such a function once takes care of gobbling all the tokens in one go.

```
1555 \cs_set:Npn \use_none:n #1 {}  
1556 \cs_set:Npn \use_none:nn #1#2 {}  
1557 \cs_set:Npn \use_none:nnn #1#2#3 {}  
1558 \cs_set:Npn \use_none:nnnn #1#2#3#4 {}  
1559 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 {}  
1560 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 {}  
1561 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 {}  
1562 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 {}  
1563 \cs_set:Npn \use_none:nnnnnnnnn #1#2#3#4#5#6#7#8#9 {}
```

(End definition for `\use_none:n` and others. These functions are documented on page 25.)

## 41.6 Debugging and patching later definitions

```
1564 <@=debug>
```

```
\__kernel_if_debug:TF
```

A more meaningful test of whether debugging is enabled than messing up with guards. We can also more easily change the logic in one place then. This is needed primarily for deprecations.

```
1565 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
```

(End definition for `\__kernel_if_debug:TF`.)

```
\debug_on:n Stubs.
```

```
1566 \cs_set_protected:Npn \debug_on:n #1  
1567 {  
1568   \msg_error:nnx { debug } { enable-debug }  
1569   { \tl_to_str:n { \debug_on:n {#1} } }  
1570 }  
1571 \cs_set_protected:Npn \debug_off:n #1  
1572 {  
1573   \msg_error:nnx { debug } { enable-debug }  
1574   { \tl_to_str:n { \debug_off:n {#1} } }  
1575 }
```

(End definition for `\debug_on:n` and `\debug_off:n`. These functions are documented on page 29.)

```
\debug_suspend:  
\debug_resume:  
1576 \cs_set_protected:Npn \debug_suspend: { }  
1577 \cs_set_protected:Npn \debug_resume: { }
```

(End definition for `\debug_suspend:` and `\debug_resume::`. These functions are documented on page 29.)

`\_kernel_deprecation_code:nn`  
`\g__debug_deprecation_on_tl`  
`\g__debug_deprecation_off_tl`

```
1578 \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }  
1579 \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }  
1580 \cs_set_protected:Npn \_kernel_deprecation_code:nn #1#2  
1581 {  
1582     \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}  
1583     \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}  
1584 }
```

(End definition for `\_kernel_deprecation_code:nn`, `\g__debug_deprecation_on_tl`, and `\g__debug_deprecation_off_tl`.)

## 41.7 Conditional processing and definitions

```
1585 <@=prg>
```

Underneath any predicate function (`_p`) or other conditional forms (TF, etc.) is a built-in logic saying that it after all of the testing and processing must return the `<state>` this leaves `TEX` in. Therefore, a simple user interface could be something like

```
\if_meaning:w #1#2  
    \prg_return_true:  
\else:  
    \if_meaning:w #1#3  
        \prg_return_true:  
\else:  
        \prg_return_false:  
\fi:  
\fi:
```

Usually, a `TEX` programmer would have to insert a number of `\exp_after:wN`s to ensure the state value is returned at exactly the point where the last conditional is finished. However, that obscures the code and forces the `TEX` programmer to prove that he/she knows the  $2^n - 1$  table. We therefore provide the simpler interface.

`\prg_return_true:`  
`\prg_return_false:`

The idea here is that `\exp:w` expands fully any `\else:` and `\fi:` that are waiting to be discarded, before reaching the `\exp_end:` which leaves an empty expansion. The code can then leave either the first or second argument in the input stream. This means that all of the branching code has to contain at least two tokens: see how the logical tests are actually implemented to see this.

```
1586 \cs_set:Npn \prg_return_true:  
1587 { \exp_after:wN \use_i:nn \exp:w }  
1588 \cs_set:Npn \prg_return_false:  
1589 { \exp_after:wN \use_ii:nn \exp:w }
```

An extended state space could be implemented by including a more elaborate function in place of `\use_i:nn`/`\use_ii:nn`. Provided two arguments are absorbed then the code would work.

*(End definition for `\prg_return_true:` and `\prg_return_false:`. These functions are documented on page 64.)*

Private version of `\use_none_delimit_by_q_recursion_stop:w`.

```
1590 \cs_set:Npn \__prg_use_none_delimit_by_q_recursion_stop:w
1591   #1 \q_prg_recursion_stop { }
```

*(End definition for `\__prg_use_none_delimit_by_q_recursion_stop:w`.)*

The user functions for the types using parameter text from the programmer. The various functions only differ by which function is used for the assignment. For those `Npnn` type functions, we must grab the parameter text, reading everything up to a left brace before continuing. Then split the base function into name and signature, and feed `{<name>}{<signature>}{<boolean>}{<set or new>}{<maybe protected>}{<parameters>}{TF,...}{<code>}` to the auxiliary function responsible for defining all conditionals. Note that `e` stands for expandable and `p` for protected.

```
1592 \cs_set_protected:Npn \prg_set_conditional:Npnn
1593   { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }
1594 \cs_set_protected:Npn \prg_gset_conditional:Npnn
1595   { \__prg_generate_conditional_parm:NNNpnn \cs_gset:Npn e }
1596 \cs_set_protected:Npn \prg_new_conditional:Npnn
1597   { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }
1598 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn
1599   { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }
1600 \cs_set_protected:Npn \prg_gset_protected_conditional:Npnn
1601   { \__prg_generate_conditional_parm:NNNpnn \cs_gset_protected:Npn p }
1602 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn
1603   { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }
1604 \cs_set_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#
1605   {
1606     \use:x
1607     {
1608       \__prg_generate_conditional:nnNNNnnn
1609       \cs_split_function:N #3
1610     }
1611     #1 #2 {#4}
1612   }
```

*(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 62.)*

The user functions for the types automatically inserting the correct parameter text based on the signature. The various functions only differ by which function is used for the assignment. Split the base function into name and signature. The second auxiliary generates the parameter text from the number of letters in the signature. Then feed `{<name>}{<signature>}{<boolean>}{<set or new>}{<maybe protected>}{<parameters>}{TF,...}{<code>}` to the auxiliary function responsible for defining all conditionals. If the `<signature>` has more than 9 letters, the definition is aborted since TeX macros have at most 9 arguments. The erroneous case where the function name contains no colon is captured later.

```

1613 \cs_set_protected:Npn \prg_set_conditional:Nnn
1614   { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1615 \cs_set_protected:Npn \prg_gset_conditional:Nnn
1616   { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
1617 \cs_set_protected:Npn \prg_new_conditional:Nnn
1618   { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }
1619 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
1620   { \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
1621 \cs_set_protected:Npn \prg_gset_protected_conditional:Nnn
1622   { \__prg_generate_conditional_count:NNNnn \cs_gset_protected:Npn p }
1623 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
1624   { \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
1625 \cs_set_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
1626   {
1627     \use:x
1628     {
1629       \__prg_generate_conditional_count:nnNNNnn
1630       \cs_split_function:N #3
1631     }
1632     #1 #2
1633   }
1634 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
1635   {
1636     \__kernel_cs_parm_from_arg_count:nnF
1637     { \__prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
1638     { \tl_count:n {#2} }
1639     {
1640       \msg_error:nnxx { kernel } { bad-number-of-arguments }
1641       { \token_to_str:c {#1 : #2} }
1642       { \tl_count:n {#2} }
1643       \use_none:nn
1644     }
1645   }

```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 62.)

```
\__prg_generate_conditional:nnNNNnnn
\__prg_generate_conditional:NNnnnnNw
\__prg_generate_conditional_test:w
\__prg_generate_conditional_fast:nw
```

The workhorse here is going through a list of desired forms, *i.e.*, p, TF, T and F. The first three arguments come from splitting up the base form of the conditional, which gives the name, signature and a boolean to signal whether or not there was a colon in the name. In the absence of a colon, we throw an error and don't define any conditional. The fourth and fifth arguments build up the defining function. The sixth is the parameters to use (possibly empty), the seventh is the list of forms to define, the eighth is the replacement text which we will augment when defining the forms. The use of `\tl_to_str:n` makes the later loop more robust.

A large number of our low-level conditionals look like `<code> \prg_return_true: \else: \prg_return_false: \fi:` so we optimize this special case by calling `\__prg_generate_conditional_fast:nw {<code>}`. This passes `\use_i:nn` instead of `\use_i:ii:nnn` to functions such as `\__prg_generate_p_form:wNNnnnnN`.

```

1646 \cs_set_protected:Npn \__prg_generate_conditional:nnNNNnnn #1#2#3#4#5#6#7#8
1647   {
1648     \if_meaning:w \c_false_bool #3
1649     \msg_error:nnx { kernel } { missing-colon }
```

```

1650      { \token_to_str:c {#1} }
1651      \exp_after:wN \use_none:nn
1652 \fi:
1653 \use:x
1654 {
1655     \exp_not:N \__prg_generate_conditional>NNnnnnNw
1656     \exp_not:n { #4 #5 {#1} {#2} {#6} }
1657     \__prg_generate_conditional_test:w
1658     #8 \s__prg_mark
1659         \__prg_generate_conditional_fast:nw
1660         \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark
1661         \use_none:n
1662         \exp_not:n { {#8} \use_i_ii:nnn }
1663         \tl_to_str:n {#7}
1664         \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }
1665     }
1666 }
1667 \cs_set:Npn \__prg_generate_conditional_test:w
1668     #1 \prg_return_true: \else: \prg_return_false: \fi: \s__prg_mark #2
1669     { #2 {#1} }
1670 \cs_set:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
1671     { \exp_not:n { {#1} \use_i:nn } }

```

Looping through the list of desired forms. First are six arguments and seventh is the form. Use the form to call the correct type. If the form does not exist, the \use:c construction results in \relax, and the error message is displayed (unless the form is empty, to allow for {T, , F}), then \use\_none:nnnnnnnn cleans up. Otherwise, the error message is removed by the variant form.

```

1672 \cs_set_protected:Npn \__prg_generate_conditional>NNnnnnNw #1#2#3#4#5#6#7#8 ,
1673 {
1674     \if_meaning:w \q__prg_recursion_tail #8
1675         \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w
1676     \fi:
1677     \use:c { __prg_generate_ #8 _form:wNNnnnnN }
1678         \tl_if_empty:nF {#8}
1679         {
1680             \msg_error:nnxx
1681                 { kernel } { conditional-form-unknown }
1682                 {#8} { \token_to_str:c { #3 : #4 } }
1683         }
1684         \use_none:nnnnnnn
1685         \s__prg_stop
1686         #1 #2 {#3} {#4} {#5} {#6} #7
1687     \__prg_generate_conditional>NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
1688 }

```

(End definition for \\_\_prg\_generate\_conditional:nnNNnnnn and others.)

\\_\_prg\_generate\_p\_form:wNNnnnn  
\\_\_prg\_generate\_TF\_form:wNNnnnnN  
\\_\_prg\_generate\_T\_form:wNNnnnnN  
\\_\_prg\_generate\_F\_form:wNNnnnnN  
\\_\_prg\_p\_true:w  
\\_\_prg\_T\_true:w  
\\_\_prg\_F\_true:w  
\\_\_prg\_TF\_true:w

How to generate the various forms. Those functions take the following arguments: 1: junk, 2: \cs\_set:Npn or similar, 3: p (for protected conditionals) or e, 4: function name, 5: signature, 6: parameter text, 7: replacement (possibly trimmed by \\_\_prg\_generate\_conditional\_fast:nw), 8: \use\_i\_ii:nnn or \use\_i:nn (for “fast” conditionals). Remember that the logic-returning functions expect two arguments to be present after \exp\_end:: notice the construction of the different variants relies on this, and that

the TF and F variants will be slightly faster than the T version. The p form is only valid for expandable tests, we check for that by making sure that the second argument is empty. For “fast” conditionals, #7 has an extra \if\_. . . To optimize a bit further we don’t use \exp\_after:wN \use\_i:nnn and similar but instead use \\_\\_prg\_TF\_true:w and similar to swap out the macro after \fi:. It would be a tiny bit faster if we directly grabbed the T and F arguments there, but if those are actually missing, the recovery from the runaway argument would not insert \fi: back, messing up nesting of conditionals.

```

1689 \cs_set_protected:Npn \_\_prg_generate_p_form:wNNnnnnN
1690     #1 \s__prg_stop #2#3#4#5#6#7#8
1691 {
1692     \if_meaning:w e #3
1693         \exp_after:wN \use_i:nn
1694     \else:
1695         \exp_after:wN \use_i:nn
1696     \fi:
1697     {
1698         #8
1699         { \exp_args:Nc #2 { #4 _p: #5 } #6 }
1700         { { #7 \exp_end: \c_true_bool \c_false_bool } }
1701         { #7 \_\_prg_p_true:w \fi: \c_false_bool }
1702     }
1703     {
1704         \msg_error:nnx { kernel } { protected-predicate }
1705         { \token_to_str:c { #4 _p: #5 } }
1706     }
1707 }
1708 \cs_set_protected:Npn \_\_prg_generate_T_form:wNNnnnnN
1709     #1 \s__prg_stop #2#3#4#5#6#7#8
1710 {
1711     #8
1712     { \exp_args:Nc #2 { #4 : #5 T } #6 }
1713     { { #7 \exp_end: \use:n \use_none:n } }
1714     { #7 \_\_prg_T_true:w \fi: \use_none:n }
1715 }
1716 \cs_set_protected:Npn \_\_prg_generate_F_form:wNNnnnnN
1717     #1 \s__prg_stop #2#3#4#5#6#7#8
1718 {
1719     #8
1720     { \exp_args:Nc #2 { #4 : #5 F } #6 }
1721     { { #7 \exp_end: { } } }
1722     { #7 \_\_prg_F_true:w \fi: \use:n }
1723 }
1724 \cs_set_protected:Npn \_\_prg_generate_TF_form:wNNnnnnN
1725     #1 \s__prg_stop #2#3#4#5#6#7#8
1726 {
1727     #8
1728     { \exp_args:Nc #2 { #4 : #5 TF } #6 }
1729     { { #7 \exp_end: { } } }
1730     { #7 \_\_prg_TF_true:w \fi: \use_i:nn }
1731 }
1732 \cs_set:Npn \_\_prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }
1733 \cs_set:Npn \_\_prg_T_true:w \fi: \use_none:n { \fi: \use:n }
1734 \cs_set:Npn \_\_prg_F_true:w \fi: \use:n { \fi: \use_none:n }
```

```
1735 \cs_set:Npn \__prg_TF_true:w \fi: \use_i:nn { \fi: \use_i:nn }
```

(End definition for `\__prg_generate_p_form:wNNnnnnN` and others.)

```
\prg_set_eq_conditional:Nn  
\prg_new_eq_conditional:Nn  
\__prg_set_eq_conditional:NNn
```

The setting-equal functions. Split both functions and feed `{(name1)} {(signature1)}` `{(name2)} {(signature2)}` `<boolean2>` `<copying function>` `<conditions>`, `\q__-prg_recursion_tail`, `\q__prg_recursion_stop` to a first auxiliary.

```
1736 \cs_set_protected:Npn \prg_set_eq_conditional:Nn  
  { \__prg_set_eq_conditional:NNn \cs_set_eq:cc }  
1738 \cs_set_protected:Npn \prg_new_eq_conditional:Nn  
  { \__prg_set_eq_conditional:NNn \cs_new_eq:cc }  
1740 \cs_set_protected:Npn \__prg_set_eq_conditional:NNn #1#2#3#4  
  {  
    \use:x  
    {  
      \exp_not:N \__prg_set_eq_conditional:nnNnnNNw  
      \cs_split_function:N #2  
      \cs_split_function:N #3  
      \exp_not:N #1  
      \tl_to_str:n {#4}  
      \exp_not:n { , \q__prg_recursion_tail , \q__prg_recursion_stop }  
    }  
  }  
1751 }
```

(End definition for `\prg_set_eq_conditional:Nn`, `\prg_new_eq_conditional:Nn`, and `\__prg_set_eq_conditional:NNn`. These functions are documented on page 64.)

Split the function to be defined, and setup a manual clist loop over argument #6 of the first auxiliary. The second auxiliary receives twice three arguments coming from splitting the function to be defined and the function to copy. Make sure that both functions contained a colon, otherwise we don't know how to build conditionals, hence abort. Call the looping macro, with arguments `{(name1)} {(signature1)}` `{(name2)} {(signature2)}` `<copying function>` and followed by the comma list. At each step in the loop, make sure that the conditional form we copy is defined, and copy it, otherwise abort.

```
1752 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6  
  {  
    \if_meaning:w \c_false_bool #3  
    \msg_error:nnx { kernel } { missing-colon }  
    { \token_to_str:c {#1} }  
    \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w  
  \fi:  
  \if_meaning:w \c_false_bool #6  
  \msg_error:nnx { kernel } { missing-colon }  
  { \token_to_str:c {#4} }  
  \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w  
  \fi:  
  \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#4} {#5}  
  }  
1766 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,  
  {  
    \if_meaning:w \q__prg_recursion_tail #6  
    \exp_after:wN \__prg_use_none_delimit_by_q_recursion_stop:w  
  \fi:  
  \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
```

```

1772          \tl_if_empty:nF {#6}
1773          {
1774              \msg_error:nxxx
1775                  { kernel } { conditional-form-unknown }
1776                  {#6} { \token_to_str:c { #1 : #2 } }
1777          }
1778          \use_none:nnnnnn
1779          \s_prg_stop
1780          #5 {#1} {#2} {#3} {#4}
1781          \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
1782      }
1783 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn {#1 \s_prg_stop #2#3#4#5#6
1784     { #2 { #3 _p : #4 } { #5 _p : #6 } }
1785 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn {#1 \s_prg_stop #2#3#4#5#6
1786     { #2 { #3 : #4 TF } { #5 : #6 TF } }
1787 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn {#1 \s_prg_stop #2#3#4#5#6
1788     { #2 { #3 : #4 T } { #5 : #6 T } }
1789 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn {#1 \s_prg_stop #2#3#4#5#6
1790     { #2 { #3 : #4 F } { #5 : #6 F } }

```

(End definition for `\__prg_set_eq_conditional:nnNnnNNw` and others.)

All that is left is to define the canonical boolean true and false. I think Michael originated the idea of expandable boolean tests. At first these were supposed to expand into either TT or TF to be tested using `\if:w` but this was later changed to 00 and 01, so they could be used in logical operations. Later again they were changed to being numerical constants with values of 1 for true and 0 for false. We need this from the get-go.

`\c_true_bool` Here are the canonical boolean values.

`\c_false_bool`

```

1791 \tex_chardef:D \c_true_bool = 1 ~
1792 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for `\c_true_bool` and `\c_false_bool`. These variables are documented on page 26.)

## 41.8 Dissecting a control sequence

1793 `<@=cs>`

---

`\__cs_count_signature:N \__cs_count_signature:N <function>`

Splits the `<function>` into the `<name>` (*i.e.* the part before the colon) and the `<signature>` (*i.e.* after the colon). The `<number>` of tokens in the `<signature>` is then left in the input stream. If there was no `<signature>` then the result is the marker value `-1`.

---

`\__cs_get_function_name:N * \__cs_get_function_name:N <function>`

Splits the `<function>` into the `<name>` (*i.e.* the part before the colon) and the `<signature>` (*i.e.* after the colon). The `<name>` is then left in the input stream without the escape character present made up of tokens with category code 12 (other).

---

**\\_cs\_get\_function\_signature:N** \* **\\_cs\_get\_function\_signature:N** *(function)*

Splits the *(function)* into the *(name)* (*i.e.* the part before the colon) and the *(signature)* (*i.e.* after the colon). The *(signature)* is then left in the input stream made up of tokens with category code 12 (other).

---

**\\_cs\_tmp:w** Function used for various short-term usages, for instance defining functions whose definition involves tokens which are hard to insert normally (spaces, characters with category other).

**\cs\_to\_str:N** This converts a control sequence into the character string of its name, removing the leading escape character. This turns out to be a non-trivial matter as there are different cases:

- The usual case of a printable escape character;
- the case of a non-printable escape character, e.g., when the value of the `\escapechar` is negative;
- when the escape character is a space.

One approach to solve this is to test how many tokens result from `\token_to_str:N \a`. If there are two tokens, then the escape character is printable, while if it is non-printable then only one is present.

However, there is an additional complication: the control sequence itself may start with a space. Clearly that should *not* be lost in the process of converting to a string. So the approach adopted is a little more intricate still. When the escape character is printable, `\token_to_str:N \a` yields the escape character itself and a space. The character codes are different, thus the `\if:w` test is false, and TeX reads `\_cs_to_str:N` after turning the following control sequence into a string; this auxiliary removes the escape character, and stops the expansion of the initial `\tex_roman numeral:D`. The second case is that the escape character is not printable. Then the `\if:w` test is unfinished after reading a the space from `\token_to_str:N \a`, and the auxiliary `\_cs_to_str:w` is expanded, feeding – as a second character for the test; the test is false, and TeX skips to `\fi:`, then performs `\token_to_str:N`, and stops the `\tex_roman numeral:D` with `\c_zero_int`. The last case is that the escape character is itself a space. In this case, the `\if:w` test is true, and the auxiliary `\_cs_to_str:w` comes into play, inserting `-\int_value:w`, which expands `\c_zero_int` to the character 0. The initial `\tex_roman numeral:D` then sees 0, which is not a terminated number, followed by the escape character, a space, which is removed, terminating the expansion of `\tex_roman numeral:D`. In all three cases, `\cs_to_str:N` takes two expansion steps to be fully expanded.

```
1794 \cs_set:Npn \cs_to_str:N
1795 {
```

We implement the expansion scheme using `\tex_roman numeral:D` terminating it with `\c_zero_int` rather than using `\exp:w` and `\exp_end:` as we normally do. The reason is that the code heavily depends on terminating the expansion with `\c_zero_int` so we make this dependency explicit.

```
1796 \tex_roman numeral:D
1797 \if:w \token_to_str:N \ \_cs_to_str:w \fi:
1798 \exp_after:wN \_cs_to_str:N \token_to_str:N
1799 }
```

```

1800 \cs_set:Npn \__cs_to_str:N #1 { \c_zero_int }
1801 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
1802   { -\int_value:w \fi: \exp_after:wN \c_zero_int }

```

If speed is a concern we could use `\csstring` in LuaTeX. For the empty csname that primitive gives an empty result while the current `\cs_to_str:N` gives incorrect results in all engines (this is impossible to fix without huge performance hit).

(End definition for `\cs_to_str:N`, `\__cs_to_str:N`, and `\__cs_to_str:w`. This function is documented on page 21.)

### `\cs_split_function:N`

This function takes a function name and splits it into name with the escape char removed and argument specification. In addition to this, a third argument, a boolean `<true>` or `<false>` is returned with `<true>` for when there is a colon in the function and `<false>` if there is not.

We cannot use `:` directly as it has the wrong category code so an `x`-type expansion is used to force the conversion.

First ensure that we actually get a properly evaluated string by expanding `\cs_to_str:N` twice. If the function contained a colon, the auxiliary takes as #1 the function name, delimited by the first colon, then the signature #2, delimited by `\s__cs_mark`, then `\c_true_bool` as #3, and #4 cleans up until `\s__cs_stop`. Otherwise, the #1 contains the function name and `\s__cs_mark \c_true_bool`, #2 is empty, #3 is `\c_false_bool`, and #4 cleans up. The second auxiliary trims the trailing `\s__cs_mark` from the function name if present (that is, if the original function had no colon).

```

1803 \cs_set_protected:Npn \__cs_tmp:w #1
1804   {
1805     \cs_set:Npn \cs_split_function:N ##1
1806       {
1807         \exp_after:wN \exp_after:wN \exp_after:wN
1808         \__cs_split_function_auxi:w
1809         \cs_to_str:N ##1 \s__cs_mark \c_true_bool
1810         ##1 \s__cs_mark \c_false_bool \s__cs_stop
1811       }
1812     \cs_set:Npn \__cs_split_function_auxi:w
1813       ##1 ##1 ##2 \s__cs_mark ##3##4 \s__cs_stop
1814       { \__cs_split_function_auxii:w ##1 \s__cs_mark \s__cs_stop {##2} ##3 }
1815     \cs_set:Npn \__cs_split_function_auxii:w ##1 \s__cs_mark ##2 \s__cs_stop
1816       { {##1} }
1817   }
1818 \exp_after:wN \__cs_tmp:w \token_to_str:N :

```

(End definition for `\cs_split_function:N`, `\__cs_split_function_auxi:w`, and `\__cs_split_function_auxii:w`. This function is documented on page 22.)

## 41.9 Exist or free

A control sequence is said to *exist* (to be used) if has an entry in the hash table and its meaning is different from the primitive `\relax` token. A control sequence is said to be *free* (to be defined) if it does not already exist.

### `\cs_if_exist_p:N`

Two versions for checking existence. For the N form we firstly check for `\scan_stop:` and then if it is in the hash table. There is no problem when inputting something like `\else:` or `\fi:` as TeX will only ever skip input in case the token tested against is `\scan_stop:`.

### `\cs_if_exist_p:c`

### `\cs_if_exist:NTF`

### `\cs_if_exist:cTF`

```

1819 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
1820 {
1821     \if_meaning:w #1 \scan_stop:
1822         \prg_return_false:
1823     \else:
1824         \if_cs_exist:N #1
1825             \prg_return_true:
1826         \else:
1827             \prg_return_false:
1828         \fi:
1829     \fi:
1830 }

```

For the `c` form we firstly check if it is in the hash table and then for `\scan_stop:` so that we do not add it to the hash table unless it was already there. Here we have to be careful as the text to be skipped if the first test is false may contain tokens that disturb the scanner. Therefore, we ensure that the second test is performed after the first one has concluded completely.

```

1831 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
1832 {
1833     \if_cs_exist:w #1 \cs_end:
1834         \exp_after:wN \use_i:nn
1835     \else:
1836         \exp_after:wN \use_ii:nn
1837     \fi:
1838 {
1839     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1840         \prg_return_false:
1841     \else:
1842         \prg_return_true:
1843     \fi:
1844 }
1845 \prg_return_false:
1846 }

```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 27.)

**\cs\_if\_free\_p:N** The logical reversal of the above.

```

\cs_if_free_p:c
1847 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
\cs_if_free:NTF
1848 {
1849     \if_meaning:w #1 \scan_stop:
1850         \prg_return_true:
1851     \else:
1852         \if_cs_exist:N #1
1853             \prg_return_false:
1854         \else:
1855             \prg_return_true:
1856         \fi:
1857     \fi:
1858 }
1859 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
1860 {
1861     \if_cs_exist:w #1 \cs_end:
1862         \exp_after:wN \use_i:nn

```

```

1863     \else:
1864         \exp_after:wN \use_i:nn
1865     \fi:
1866     {
1867         \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
1868         \prg_return_true:
1869     \else:
1870         \prg_return_false:
1871     \fi:
1872 }
1873 { \prg_return_true: }
1874 }
```

(End definition for `\cs_if_free:NTF`. This function is documented on page 27.)

`\cs_if_exist_use:N` The `\cs_if_exist_use:...` functions cannot be implemented as conditionals because the true branch must leave both the control sequence itself and the true code in the input stream. For the `c` variants, we are careful not to put the control sequence in the hash table if it does not exist. In `LuATeX` we could use the `\lastnamedcs` primitive.

```

1875 \cs_set:Npn \cs_if_exist_use:NTF #1#2
1876   { \cs_if_exist:NTF #1 { #1 #2 } }
1877 \cs_set:Npn \cs_if_exist_use:NF #1
1878   { \cs_if_exist:NTF #1 { #1 } }
1879 \cs_set:Npn \cs_if_exist_use:NT #1 #2
1880   { \cs_if_exist:NTF #1 { #1 #2 } { } }
1881 \cs_set:Npn \cs_if_exist_use:N #1
1882   { \cs_if_exist:NTF #1 { #1 } { } }
1883 \cs_set:Npn \cs_if_exist_use:cTF #1#2
1884   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
1885 \cs_set:Npn \cs_if_exist_use:cF #1
1886   { \cs_if_exist:cTF {#1} { \use:c {#1} } }
1887 \cs_set:Npn \cs_if_exist_use:cT #1#2
1888   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
1889 \cs_set:Npn \cs_if_exist_use:c #1
1890   { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }
```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 21.)

## 41.10 Preliminaries for new functions

We provide two kinds of functions that can be used to define control sequences. On the one hand we have functions that check if their argument doesn't already exist, they are called `\..._new`. The second type of defining functions doesn't check if the argument is already defined.

Before we can define them, we need some auxiliary macros that allow us to generate error messages. The next few definitions here are only temporary, they will be redefined later on.

`\msg_error:nxx` If an internal error occurs before `IATEX3` has loaded `I3msg` then the code should issue a usable if terse error message and halt. This can only happen if a coding error is made by the team, so this is a reasonable response. Setting the `\newlinechar` is needed, to turn `^J` into a proper line break in plain `TeX`.

```

1891 \cs_set_protected:Npn \msg_error:nxxx #1#2#3#4
1892 {
1893     \tex_newlinechar:D = '\^J \scan_stop:
1894     \tex_errmessage:D
1895     {
1896         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!~! ^J
1897         Argh,~internal~LaTeX3~error! ^J ^J
1898         Module ~ #1 , ~ message~name~"#2": ^J
1899         Arguments~'#3'~and~'#4' ^J ^J
1900         This~is~one~for~The~LaTeX3~Project:~bailing~out
1901     }
1902     \tex_end:D
1903 }
1904 \cs_set_protected:Npn \msg_error:nnx #1#2#3
1905 { \msg_error:nxxx {#1} {#2} {#3} { } }
1906 \cs_set_protected:Npn \msg_error:nn #1#2
1907 { \msg_error:nxxx {#1} {#2} { } { } }

```

(End definition for `\msg_error:nxxx`, `\msg_error:nnx`, and `\msg_error:nn`. These functions are documented on page ??.)

**\msg\_line\_context:** Another one from l3msg which will be altered later.

```

1908 \cs_set:Npn \msg_line_context:
1909 { on-line~ \tex_the:D \tex_inputlineno:D }

```

(End definition for `\msg_line_context`. This function is documented on page 78.)

**\iow\_log:x** We define a routine to write only to the log file. And a similar one for writing to both the log file and the terminal. These will be redefined later by l3io.

```

1910 \cs_set_protected:Npn \iow_log:x
1911 { \tex_immediate:D \tex_write:D -1 }
1912 \cs_set_protected:Npn \iow_term:x
1913 { \tex_immediate:D \tex_write:D 16 }

```

(End definition for `\iow_log:n`. This function is documented on page 90.)

**\\_kernel\_chk\_if\_free\_cs:N** This command is called by `\cs_new_nopar:Npn` and `\cs_new_eq:NN` etc. to make sure that the argument sequence is not already in use. If it is, an error is signalled. It checks if `\csname` is undefined or `\scan_stop:`. Otherwise an error message is issued. We have to make sure we don't put the argument into the conditional processing since it may be an `\if...` type function!

```

1914 \cs_set_protected:Npn \_kernel_chk_if_free_cs:N #1
1915 {
1916     \cs_if_free:NF #1
1917     {
1918         \msg_error:nxxx { kernel } { command-already-defined }
1919         { \token_to_str:N #1 } { \token_to_meaning:N #1 }
1920     }
1921 }
1922 \cs_set_protected:Npn \_kernel_chk_if_free_cs:c
1923 { \exp_args:Nc \_kernel_chk_if_free_cs:N }

```

(End definition for `\_kernel_chk_if_free_cs:N`.)

## 41.11 Defining new functions

1924 `<@@=cs>`

```
\cs_new_nopar:Npn
\cs_new_nopar:Npx
\cs_new:Npn
\cs_new:Npx
\cs_new_protected_nopar:Npn
\cs_new_protected_nopar:Npx
\cs_new_protected:Npn
\cs_new_protected:Npx
\__cs_tmp:w
```

Function which check that the control sequence is free before defining it.

```
1925 \cs_set:Npn \__cs_tmp:w #1#2
1926 {
1927     \cs_set_protected:Npn #1 ##1
1928     {
1929         \__kernel_chk_if_free_cs:N ##1
1930         #2 ##1
1931     }
1932 }
1933 \__cs_tmp:w \cs_new_nopar:Npn      \cs_gset_nopar:Npn
1934 \__cs_tmp:w \cs_new_nopar:Npx      \cs_gset_nopar:Npx
1935 \__cs_tmp:w \cs_new:Npn          \cs_gset:Npn
1936 \__cs_tmp:w \cs_new:Npx          \cs_gset:Npx
1937 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
1938 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
1939 \__cs_tmp:w \cs_new_protected:Npn      \cs_gset_protected:Npn
1940 \__cs_tmp:w \cs_new_protected:Npx      \cs_gset_protected:Npx
```

(End definition for `\cs_new_nopar:Npn` and others. These functions are documented on page 15.)

<pre>\cs_set_nopar:cpn \cs_set_nopar:cpx \cs_gset_nopar:cpn \cs_gset_nopar:cpx \cs_new_nopar:cpn \cs_new_nopar:cpx</pre>	<p>Like <code>\cs_set_nopar:Npn</code> and <code>\cs_new_nopar:Npn</code>, except that the first argument consists of the sequence of characters that should be used to form the name of the desired control sequence (the c stands for csname argument, see the expansion module). Global versions are also provided.</p> <p><code>\cs_set_nopar:cpn</code> turns <code>&lt;string&gt;</code> into a csname and then assigns <code>&lt;rep-text&gt;</code> to it by using <code>\cs_set_nopar:Npn</code>. This means that there might be a parameter string between the two arguments.</p>
--	---

```
1941 \cs_set:Npn \__cs_tmp:w #1#2
1942     { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
1943 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
1944 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
1945 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
1946 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
1947 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
1948 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx
```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 16.)

<pre>\cs_set:cpn \cs_set:cpx \cs_gset:cpn \cs_gset:cpx \cs_new:cpn \cs_new:cpx</pre>	<p>Variants of the <code>\cs_set:Npn</code> versions which make a csname out of the first arguments. We may also do this globally.</p> <pre>1949 \__cs_tmp:w \cs_set:cpn \cs_set:Npn 1950 \__cs_tmp:w \cs_set:cpx \cs_set:Npx 1951 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn 1952 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx 1953 \__cs_tmp:w \cs_new:cpn \cs_new:Npn 1954 \__cs_tmp:w \cs_new:cpx \cs_new:Npx</pre>
--	---

(End definition for `\cs_set:Npn`. This function is documented on page 16.)

```
\cs_set_protected_nopar:cpx
\cs_set_protected_nopar:cpx
```

Variants of the `\cs_set_protected_nopar:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
1955 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npn
1956 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
1957 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npn
1958 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
1959 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npn
1960 \__cs_tmp:w \cs_new_protected_nopar:cpx \cs_new_protected_nopar:Npx
```

(End definition for `\cs_set_protected_nopar:Npn`. This function is documented on page 16.)

```
\cs_set_protected:cpx
\cs_set_protected:cpx
```

Variants of the `\cs_set_protected:Npn` versions which make a csname out of the first arguments. We may also do this globally.

```
1961 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npn
1962 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
1963 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npn
1964 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
1965 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npn
1966 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx
```

(End definition for `\cs_set_protected:Npn`. This function is documented on page 16.)

## 41.12 Copying definitions

```
\cs_set_eq:NN
\cs_set_eq:cN
\cs_set_eq:Nc
\cs_set_eq:cc
```

These macros allow us to copy the definition of a control sequence to another control sequence.

The = sign allows us to define funny char tokens like = itself or  $\sqcup$  with this function. For the definition of `\c_space_char{~}` to work we need the ~ after the =.

`\cs_set_eq:NN` is long to avoid problems with a literal argument of `\par`. While `\cs_new_eq:NN` will probably never be correct with a first argument of `\par`, define it long in order to throw an “already defined” error rather than “runaway argument”.

```
1967 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
1968 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
1969 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
1970 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
1971 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
1972 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
1973 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
1974 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
1975 \cs_new_protected:Npn \cs_new_eq:NN #1
1976 {
1977     \__kernel_chk_if_free_cs:N #1
1978     \tex_global:D \cs_set_eq:NN #1
1979 }
1980 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
1981 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
1982 \cs_new_protected:Npn \cs_new_eq:cc { \exp_args:Ncc \cs_new_eq:NN }
```

(End definition for `\cs_set_eq:NN`, `\cs_gset_eq:NN`, and `\cs_new_eq:NN`. These functions are documented on page 19.)

## 41.13 Undefining functions

\cs\_undefine:N  
\cs\_undefine:c

The following function is used to free the main memory from the definition of some function that isn't in use any longer. The c variant is careful not to add the control sequence to the hash table if it isn't there yet, and it also avoids nesting TeX conditionals in case #1 is unbalanced in this matter.

```
1983 \cs_new_protected:Npn \cs_undefine:N #1
1984   { \cs_gset_eq:NN #1 \tex_undefined:D }
1985 \cs_new_protected:Npn \cs_undefine:c #1
1986   {
1987     \if_cs_exist:w #1 \cs_end:
1988       \exp_after:wN \use:n
1989     \else:
1990       \exp_after:wN \use_none:n
1991     \fi:
1992   { \cs_gset_eq:cN {#1} \tex_undefined:D }
1993 }
```

(End definition for \cs\_undefine:N. This function is documented on page 20.)

## 41.14 Generating parameter text from argument count

1994 ⟨@=cs⟩

\

\_kernel\_cs\_parm\_from\_arg\_count:nnF  
\\_cs\_parm\_from\_arg\_count\_test:nnF

LaTeX3 provides shorthands to define control sequences and conditionals with a simple parameter text, derived directly from the signature, or more generally from knowing the number of arguments, between 0 and 9. This function expands to its first argument, untouched, followed by a brace group containing the parameter text {#1...#n}, where n is the result of evaluating the second argument (as described in \int\_eval:n). If the second argument gives a result outside the range [0, 9], the third argument is returned instead, normally an error message. Some of the functions use here are not defined yet, but will be defined before this function is called.

```
1995 \cs_set_protected:Npn \_kernel_cs_parm_from_arg_count:nnF #1#2
1996   {
1997     \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
1998     {
1999       \exp_after:wN \exp_not:n
2000       \if_case:w \int_eval:n {#2}
2001         {}
2002         \or: {##1}
2003         \or: {##1##2}
2004         \or: {##1##2##3}
2005         \or: {##1##2##3##4}
2006         \or: {##1##2##3##4##5}
2007         \or: {##1##2##3##4##5##6}
2008         \or: {##1##2##3##4##5##6##7}
2009         \or: {##1##2##3##4##5##6##7##8}
2010         \or: {##1##2##3##4##5##6##7##8##9}
2011         \else: { \c_false_bool }
2012       \fi:
2013     }
2014   {#1}
2015 }
```

```

2016 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
2017  {
2018      \if_meaning:w \c_false_bool #1
2019          \exp_after:wN \use_i:nn
2020      \else:
2021          \exp_after:wN \use_i:nn
2022      \fi:
2023      { #2 {#1} }
2024  }

```

(End definition for `\__kernel_cs_parm_from_arg_count:nnF` and `\__cs_parm_from_arg_count_test:nnF`.)

## 41.15 Defining functions from a given number of arguments

2025 `\__cs_count_signature:N`

Counting the number of tokens in the signature, *i.e.*, the number of arguments the function should take. Since this is not used in any time-critical function, we simply use `\tl_count:n` if there is a signature, otherwise `-1` arguments to signal an error. We need a variant form right away.

```

2026 \cs_new:Npn \__cs_count_signature:N #1
2027  { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2028 \cs_new:Npn \__cs_count_signature:n #1
2029  { \int_eval:n { \__cs_count_signature:nnN #1 } }
2030 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2031  {
2032      \if_meaning:w \c_true_bool #3
2033          \tl_count:n {#2}
2034      \else:
2035          -1
2036      \fi:
2037  }
2038 \cs_new:Npn \__cs_count_signature:c
2039  { \exp_args:Nc \__cs_count_signature:N }

```

(End definition for `\__cs_count_signature:N`, `\__cs_count_signature:n`, and `\__cs_count_signature:nnN`.)

`\cs_generate_from_arg_count>NNnn`

`\cs_generate_from_arg_count:cNnn`

`\cs_generate_from_arg_count:Ncnn`

We provide a constructor function for defining functions with a given number of arguments. For this we need to choose the correct parameter text and then use that when defining. Since TeX supports from zero to nine arguments, we use a simple switch to choose the correct parameter text, ensuring the result is returned after finishing the conditional. If it is not between zero and nine, we throw an error.

1: function to define, 2: with what to define it, 3: the number of args it requires and 4: the replacement text

```

2040 \cs_new_protected:Npn \cs_generate_from_arg_count>NNnn #1#2#3#4
2041  {
2042      \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2043      {
2044          \msg_error:nnxx { kernel } { bad-number-of-arguments }
2045          { \token_to_str:N #1 } { \int_eval:n {#3} }
2046          \use_none:n
2047      }

```

```

2048     {#4}
2049   }

```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

2050 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2051   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
2052 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2053   { \exp_args:NNc \cs_generate_from_arg_count:NNnn }

```

(End definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 19.)

## 41.16 Using the signature to define functions

```

2054 <@@=cs>

```

We can now combine some of the tools we have to provide a simple interface for defining functions, where the number of arguments is read from the signature. For instance, `\cs_set:Nn \foo_bar:nn {#1,#2}`.

We want to define `\cs_set:Nn` as

```

\cs_set_protected:Npn \cs_set:Nn #1#2
{
  \cs_generate_from_arg_count:NNnn #1 \cs_set:Npn
  { \@@_count_signature:N #1 } {#2}
}

```

In short, to define `\cs_set:Nn` we need just use `\cs_set:Npn`, everything else is the same for each variant. Therefore, we can make it simpler by temporarily defining a function to do this for us.

```

2055 \cs_set:Npn \__cs_tmp:w #1#2#3
2056 {
2057   \cs_new_protected:cpx { cs_ #1 : #2 }
2058   {
2059     \exp_not:N \__cs_generate_from_signature:NNn
2060     \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
2061   }
2062 }
2063 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
2064 {
2065   \use:x
2066   {
2067     \__cs_generate_from_signature:nnNNNn
2068     \cs_split_function:N #2
2069   }
2070   #1 #2
2071 }
2072 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
2073 {
2074   \bool_if:NTF #3
2075   {
2076     \cs_set_nopar:Npx \__cs_tmp:w
2077     { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
2078     \tl_if_empty:oF \__cs_tmp:w

```

```

2079         {
2080             \msg_error:nxxxx { kernel } { non-base-function }
2081                 { \token_to_str:N #5 } {#2} { \_cs_tmp:w }
2082         }
2083         \cs_generate_from_arg_count:NNnn
2084             #5 #4 { \tl_count:n {#2} } {#6}
2085     }
2086     {
2087         \msg_error:nnx { kernel } { missing-colon }
2088             { \token_to_str:N #5 }
2089     }
2090 }
2091 \cs_new:Npn \__cs_generate_from_signature:n #1
2092 {
2093     \if:w n #1 \else: \if:w N #1 \else:
2094         \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
2095 }

```

Then we define the 24 variants beginning with N.

```

2096 \__cs_tmp:w { set }                      { Nn } { Npn }
2097 \__cs_tmp:w { set }                      { Nx } { Npx }
2098 \__cs_tmp:w { set_nopar }                { Nn } { Npn }
2099 \__cs_tmp:w { set_nopar }                { Nx } { Npx }
2100 \__cs_tmp:w { set_protected }            { Nn } { Npn }
2101 \__cs_tmp:w { set_protected }            { Nx } { Npx }
2102 \__cs_tmp:w { set_protected_nopar }      { Nn } { Npn }
2103 \__cs_tmp:w { set_protected_nopar }      { Nx } { Npx }
2104 \__cs_tmp:w { gset }                     { Nn } { Npn }
2105 \__cs_tmp:w { gset }                     { Nx } { Npx }
2106 \__cs_tmp:w { gset_nopar }               { Nn } { Npn }
2107 \__cs_tmp:w { gset_nopar }               { Nx } { Npx }
2108 \__cs_tmp:w { gset_protected }           { Nn } { Npn }
2109 \__cs_tmp:w { gset_protected }           { Nx } { Npx }
2110 \__cs_tmp:w { gset_protected_nopar }     { Nn } { Npn }
2111 \__cs_tmp:w { gset_protected_nopar }     { Nx } { Npx }
2112 \__cs_tmp:w { new }                      { Nn } { Npn }
2113 \__cs_tmp:w { new }                      { Nx } { Npx }
2114 \__cs_tmp:w { new_nopar }                { Nn } { Npn }
2115 \__cs_tmp:w { new_nopar }                { Nx } { Npx }
2116 \__cs_tmp:w { new_protected }            { Nn } { Npn }
2117 \__cs_tmp:w { new_protected }            { Nx } { Npx }
2118 \__cs_tmp:w { new_protected_nopar }      { Nn } { Npn }
2119 \__cs_tmp:w { new_protected_nopar }      { Nx } { Npx }

```

(End definition for `\cs_set:Nn` and others. These functions are documented on page 17.)

<pre> \cs_set:cn \cs_set:cx \cs_set_nopar:cn \cs_set_nopar:cx \cs_set_protected:cn \cs_set_protected:cx \cs_set_protected_nopar:cn \cs_set_protected_nopar:cx     \cs_gset:cn     \cs_gset:cx \cs_gset_nopar:cn \cs_gset_nopar:cx \cs_gset_protected:cn \cs_gset_protected:cx \cs_gset_protected_nopar:cn \cs_gset_protected_nopar:cx     \cs_new:cn     \cs_new:cx </pre>	<p>The 24 c variants simply use <code>\exp_args:Nc</code>.</p> <pre> 2120 \cs_set:Npn \__cs_tmp:w #1#2 2121 { 2122     \cs_new_protected:cpx { cs_ #1 : c #2 } 2123     { 2124         \exp_not:N \exp_args:Nc 2125         \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end: 2126     } 2127 } </pre>
--	--

```

2128 \__cs_tmp:w { set }           { n }
2129 \__cs_tmp:w { set }           { x }
2130 \__cs_tmp:w { set_nopar }     { n }
2131 \__cs_tmp:w { set_nopar }     { x }
2132 \__cs_tmp:w { set_protected } { n }
2133 \__cs_tmp:w { set_protected } { x }
2134 \__cs_tmp:w { set_protected_nopar } { n }
2135 \__cs_tmp:w { set_protected_nopar } { x }
2136 \__cs_tmp:w { gset }          { n }
2137 \__cs_tmp:w { gset }          { x }
2138 \__cs_tmp:w { gset_nopar }    { n }
2139 \__cs_tmp:w { gset_nopar }    { x }
2140 \__cs_tmp:w { gset_protected } { n }
2141 \__cs_tmp:w { gset_protected } { x }
2142 \__cs_tmp:w { gset_protected_nopar } { n }
2143 \__cs_tmp:w { gset_protected_nopar } { x }
2144 \__cs_tmp:w { new }           { n }
2145 \__cs_tmp:w { new }           { x }
2146 \__cs_tmp:w { new_nopar }     { n }
2147 \__cs_tmp:w { new_nopar }     { x }
2148 \__cs_tmp:w { new_protected } { n }
2149 \__cs_tmp:w { new_protected } { x }
2150 \__cs_tmp:w { new_protected_nopar } { n }
2151 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for `\cs_set:Nn`. This function is documented on page 17.)

## 41.17 Checking control sequence equality

`\cs_if_eq_p:NN` Check if two control sequences are identical.

```

\cs_if_eq_p:cN 2152 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 2153   {
\cs_if_eq_p:cc 2154     \if_meaning:w #1#2
\cs_if_eq:NNTF 2155       \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 2156   }
\cs_if_eq:NcTF 2157 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:cNTF 2158 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:cctF 2159 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
2160 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
2161 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:NNc \cs_if_eq_p:NN }
2162 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:NNc \cs_if_eq:NNTF }
2163 \cs_new:Npn \cs_if_eq:NcT { \exp_args:NNc \cs_if_eq:NNT }
2164 \cs_new:Npn \cs_if_eq:NcF { \exp_args:NNc \cs_if_eq:NNF }
2165 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
2166 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
2167 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
2168 \cs_new:Npn \cs_if_eq:ccF { \exp_args:Ncc \cs_if_eq:NNF }

```

(End definition for `\cs_if_eq:NNTF`. This function is documented on page 27.)

## 41.18 Diagnostic functions

2169 `<@=kernel>`

\\_\\_kernel\\_chk\\_defined:NT Error if the variable #1 is not defined.

```
2170 \cs_new_protected:Npn \_\_kernel_chk_defined:NT #1#2
2171 {
2172     \cs_if_exist:NTF #1
2173     {#2}
2174     {
2175         \msg_error:nnx { kernel } { variable-not-defined }
2176         { \token_to_str:N #1 }
2177     }
2178 }
```

(End definition for \\_\\_kernel\\_chk\\_defined:NT.)

\\_\\_kernel\_register\_show:N  
\\_\\_kernel\_register\_show:c  
\\_\\_kernel\_register\_log:N  
\\_\\_kernel\_register\_log:c  
  \\_\\_kernel\_register\_show\_aux:NN  
  \\_\\_kernel\_register\_show\_aux:nNN

Simply using the `\show` primitive does not allow for line-wrapping, so instead use `\tl_show:n` and `\tl_log:n` (defined in `l3tl` and that performs line-wrapping). This displays `>~<variable>=<value>`. We expand the value before-hand as otherwise some integers (such as `\currentgrouplevel` or `\currentgrouptype`) altered by the line-wrapping code would show wrong values.

```
2179 \cs_new_protected:Npn \_\_kernel_register_show:N
2180   { \_\_kernel_register_show_aux:NN \tl_show:n }
2181 \cs_new_protected:Npn \_\_kernel_register_show:c
2182   { \exp_args:Nc \_\_kernel_register_show:N }
2183 \cs_new_protected:Npn \_\_kernel_register_log:N
2184   { \_\_kernel_register_show_aux:NN \tl_log:n }
2185 \cs_new_protected:Npn \_\_kernel_register_log:c
2186   { \exp_args:Nc \_\_kernel_register_log:N }
2187 \cs_new_protected:Npn \_\_kernel_register_show_aux:NN #1#2
2188   {
2189     \_\_kernel_chk_defined:NT #2
2190     {
2191         \exp_args:No \_\_kernel_register_show_aux:nNN
2192         { \tex_the:D #2 } #2 #1
2193     }
2194   }
2195 \cs_new_protected:Npn \_\_kernel_register_show_aux:nNN #1#2#3
2196   { \exp_args:No #3 { \token_to_str:N #2 = #1 } }
```

(End definition for \\_\\_kernel\_register\_show:N and others.)

**\cs\_show:N** Some control sequences have a very long name or meaning. Thus, simply using `\show`'s primitive `\show` could lead to overlong lines. The output of this primitive is mimicked to some extent, then the re-built string is given to `\tl_show:n` or `\tl_log:n` for line-wrapping. We must expand the meaning before passing it to the wrapping code as otherwise we would wrongly see the definitions that are in place there. To get correct escape characters, set the `\escapechar` in a group; this also localizes the assignment performed by x-expansion. The `\cs_show:c` and `\cs_log:c` commands convert their argument to a control sequence within a group to avoid showing `\relax` for undefined control sequences.

```
2197 \cs_new_protected:Npn \cs_show:N { \_\_kernel_show:NN \tl_show:n }
2198 \cs_new_protected:Npn \cs_show:c
2199   { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
2200 \cs_new_protected:Npn \cs_log:N { \_\_kernel_show:NN \tl_log:n }
2201 \cs_new_protected:Npn \cs_log:c
```

```

2202 { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
2203 \cs_new_protected:Npn \__kernel_show:NN #1#2
2204 {
2205   \group_begin:
2206     \int_set:Nn \tex_escapechar:D { '\\" }
2207     \exp_args:NNx
2208   \group_end:
2209   #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
2210 }

```

(End definition for `\cs_show:N`, `\cs_log:N`, and `\__kernel_show:NN`. These functions are documented on page 20.)

**\group\_show\_list:** Wrapper around `\showgroups`. Getting TeX to write to the log without interruption the run is done by altering the interaction mode.

```

\__kernel_group_show:NN
2211 \cs_new_protected:Npn \group_show_list:
2212   { \__kernel_group_show:NN \use_none:n 1 }
2213 \cs_new_protected:Npn \group_log_list:
2214   { \__kernel_group_show:NN \int_zero:N 0 }
2215 \cs_new_protected:Npn \__kernel_group_show:NN #1#2
2216 {
2217   \use:x
2218   {
2219     #1 \tex_interactionmode:D
2220     \int_set:Nn \tex_tracingonline:D {#2}
2221     \int_set:Nn \tex_errorcontextlines:D { -1 }
2222     \exp_not:N \exp_after:wN \scan_stop:
2223     \tex_showgroups:D
2224     \int_set:Nn \tex_interactionmode:D
2225       { \int_use:N \tex_interactionmode:D }
2226     \int_set:Nn \tex_tracingonline:D
2227       { \int_use:N \tex_tracingonline:D }
2228     \int_set:Nn \tex_errorcontextlines:D
2229       { \int_use:N \tex_errorcontextlines:D }
2230   }
2231 }

```

(End definition for `\group_show_list:`, `\group_log_list:`, and `\__kernel_group_show:NN`. These functions are documented on page 14.)

## 41.19 Decomposing a macro definition

**\cs\_prefix\_spec:N** **\cs\_parameter\_spec:N** **\cs\_replacement\_spec:N** We sometimes want to test if a control sequence can be expanded to reveal a hidden value. However, we cannot just expand the macro blindly as it may have arguments and none might be present. Therefore we define these functions to pick either the prefix(es), the parameter specification, or the replacement text from a macro. All of this information is returned as characters with catcode 12. If the token in question isn't a macro, the token `\scan_stop:` is returned instead.

```

\__kernel_prefix_arg_replacement:wN
2232 \use:x
2233 {
2234   \exp_not:n { \cs_new:Npn \__kernel_prefix_arg_replacement:wN #1 }
2235   \tl_to_str:n { macro : } \exp_not:n { #2 -> #3 \s__kernel_stop #4 }
2236 }

```

```

2237   { #4 {#1} {#2} {#3} }
2238 \cs_new:Npn \cs_prefix_spec:N #1
2239   {
2240     \token_if_macro:NTF #1
2241     {
2242       \exp_after:wN \__kernel_prefix_arg_replacement:wN
2243         \token_to_meaning:N #1 \s__kernel_stop \use_i:nnn
2244     }
2245     { \scan_stop: }
2246   }
2247 \cs_new:Npn \cs_parameter_spec:N #1
2248   {
2249     \token_if_macro:NTF #1
2250     {
2251       \exp_after:wN \__kernel_prefix_arg_replacement:wN
2252         \token_to_meaning:N #1 \s__kernel_stop \use_ii:nnn
2253     }
2254     { \scan_stop: }
2255   }
2256 \cs_new:Npn \cs_replacement_spec:N #1
2257   {
2258     \token_if_macro:NTF #1
2259     {
2260       \exp_after:wN \__kernel_prefix_arg_replacement:wN
2261         \token_to_meaning:N #1 \s__kernel_stop \use_iii:nnn
2262     }
2263     { \scan_stop: }
2264   }

```

(End definition for `\cs_prefix_spec:N` and others. These functions are documented on page 22.)

## 41.20 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```
2265 \cs_new:Npn \prg_do_nothing: { }
```

(End definition for `\prg_do_nothing:`. This function is documented on page 13.)

## 41.21 Breaking out of mapping functions

```
2266 <@=prg>
```

`\prg_break_point:Nn` In inline mappings, the nesting level must be reset at the end of the mapping, even when the user decides to break out. This is done by putting the code that must be performed as an argument of `\__prg_break_point:Nn`. The breaking functions are then defined to jump to that point and perform the argument of `\__prg_break_point:Nn`, before the user's code (if any). There is a check that we close the correct loop, otherwise we continue breaking.

```

2267 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
2268 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
2269   {
2270     #5

```

```

2271     \if_meaning:w #1 #4
2272         \exp_after:wN \use_iii:nnn
2273     \fi:
2274     \prg_map_break:Nn #1 {#2}
2275 }

```

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 70.)

**\prg\_break\_point:** Very simple analogues of `\prg_break_point:Nn` and `\prg_map_break:Nn`, for use in fast short-term recursions which are not mappings, do not need to support nesting, and in which nothing has to be done at the end of the loop.

```

2276 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
2277 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
2278 \cs_new:Npn \prg_break:n #1#2 \prg_break_point: {#1}

```

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 71.)

## 41.22 Starting a paragraph

**\mode\_leave\_vertical:** The approach here is different to that used by L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> or plain T<sub>E</sub>X, which unbox a void box to force horizontal mode. That inserts the `\everypar` tokens *before* the re-inserted unboxing tokens. The approach here uses either the `\quitvmode` primitive or the equivalent protected macro. In vertical mode, the `\indent` primitive is inserted: this will switch to horizontal mode and insert `\everypar` tokens and nothing else. Unlike the L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> version, the availability of ε-T<sub>E</sub>X means using a mode test can be done at for example the start of an `\halign`.

```

2279 \cs_new_protected:Npn \mode_leave_vertical:
2280 {
2281     \if_mode_vertical:
2282         \exp_after:wN \tex_indent:D
2283     \fi:
2284 }

```

(End definition for `\mode_leave_vertical:`. This function is documented on page 28.)

```
2285 
```

# Chapter 42

## I3expan implementation

2286 `(*package)`

2287 `(@@=exp)`

\l\_\_exp\_internal\_t1 The `\exp_` module has its private variable to temporarily store the result of x-type argument expansion. This is done to avoid interference with other functions using temporary variables.

(End definition for `\l__exp_internal_t1`.)

`\exp_after:wN` These are defined in I3basics, as they are needed “early”. This is just a reminder of that fact!

`\exp_not:N`

`\exp_not:n`

(End definition for `\exp_after:wN`, `\exp_not:N`, and `\exp_not:n`. These functions are documented on page 38.)

### 42.1 General expansion

In this section a general mechanism for defining functions that handle arguments is defined. These general expansion functions are expandable unless x is used. (Any version of x is going to have to use one of the L<sup>A</sup>T<sub>E</sub>X3 names for `\cs_set:Npx` at some point, and so is never going to be expandable.)

The definition of expansion functions with this technique happens in section 42.8. In section 42.2 some common cases are coded by a more direct method for efficiency, typically using calls to `\exp_after:wN`.

\l\_\_exp\_internal\_t1 This scratch token list variable is defined in I3basics.

(End definition for `\l__exp_internal_t1`.)

This code uses internal functions with names that start with `\:::` to perform the expansions. All macros are long since the tokens undergoing expansion may be arbitrary user input.

An argument manipulator `\:::(Z)` always has signature `#1\:::#2#3` where #1 holds the remaining argument manipulations to be performed, `\:::` serves as an end marker for the list of manipulations, #2 is the carried over result of the previous expansion steps and #3 is the argument about to be processed. One exception to this rule is `\:::p`, which has to grab an argument delimited by a left brace.

`\__exp_arg_next:nnn` #1 is the result of an expansion step, #2 is the remaining argument manipulations and #3 is the current result of the expansion chain. This auxiliary function moves #1 back after #3 in the input stream and checks if any expansion is left to be done by calling #2. In by far the most cases we need to add a set of braces to the result of an argument manipulation so it is more effective to do it directly here. Actually, so far only the c of the final argument manipulation variants does not require a set of braces.

```
2288 \cs_new:Npn \__exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
2289 \cs_new:Npn \__exp_arg_next:Nnn #1#2#3 { #2 \::: { #3 #1 } }
```

(*End definition for `\__exp_arg_next:nnn` and `\__exp_arg_next:Nnn`.*)

`\:::` The end marker is just another name for the identity function.

```
2290 \cs_new:Npn \::: #1 {#1}
```

(*End definition for `\:::`. This function is documented on page 42.*)

`\:::n` This function is used to skip an argument that doesn't need to be expanded.

```
2291 \cs_new:Npn \:::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End definition for `\:::n`. This function is documented on page 42.*)

`\:::N` This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
2292 \cs_new:Npn \:::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End definition for `\:::N`. This function is documented on page 42.*)

`\:::p` This function is used to skip an argument that is delimited by a left brace and doesn't need to be expanded. It is not wrapped in braces in the result.

```
2293 \cs_new:Npn \:::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End definition for `\:::p`. This function is documented on page 42.*)

`\:::c` This function is used to skip an argument that is turned into a control sequence without expansion.

```
2294 \cs_new:Npn \:::c #1 \::: #2#3
2295   { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for `\:::c`. This function is documented on page 42.*)

`\:::o` This function is used to expand an argument once.

```
2296 \cs_new:Npn \:::o #1 \::: #2#3
2297   { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for `\:::o`. This function is documented on page 42.*)

`\:::e` With the `\expanded` primitive available, just expand. Otherwise defer to `\exp_args:N` implemented later.

```
2298 \cs_if_exist:NTF \tex_expanded:D
2299   {
2300     \cs_new:Npn \:::e #1 \::: #2#3
2301       { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
2302   }
2303   {
2304     \cs_new:Npn \:::e #1 \::: #2#3
2305       { \exp_args:N \__exp_arg_next:nnn {#3} {#1} {#2} }
2306   }
```

(End definition for `\:::e`. This function is documented on page 42.)

**`\:::f`** This function is used to expand a token list until the first unexpandable token is found. This is achieved through `\exp:w \exp_end_continue_f:w` that expands everything in its way following it. This scanning procedure is terminated once the expansion hits something non-expandable (if that is a space it is removed). We introduce `\exp_stop_f:` to mark such an end-of-expansion marker. For example, f-expanding `\cs_set_eq:Nc \aaa { b \l_tmpa_t1 b }` where `\l_tmpa_t1` contains the characters `lur` gives `\tex_let:D \aaa = \blurb` which then turns out to start with the non-expandable token `\tex_let:D`. Since the expansion of `\exp:w \exp_end_continue_f:w` is empty, we wind up with a fully expanded list, only TeX has not tried to execute any of the non-expandable tokens. This is what differentiates this function from the `x` argument type.

```
2307 \cs_new:Npn \:::f #1 \::: #2#3
2308 {
2309     \exp_after:wN \__exp_arg_next:nnn
2310     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2311     {#1} {#2}
2312 }
2313 \use:nn { \cs_new_eq:NN \exp_stop_f: } { ~ }
```

(End definition for `\:::f` and `\exp_stop_f:`. These functions are documented on page 42.)

**`\:::x`** This function is used to expand an argument fully. We build in the expansion of `\__exp_arg_next:nnn`.

```
2314 \cs_new_protected:Npn \:::x #1 \::: #2#3
2315 {
2316     \cs_set_nopar:Npx \l__exp_internal_tl
2317     { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
2318     \l__exp_internal_tl
2319 }
```

(End definition for `\:::x`. This function is documented on page 42.)

**`\:::v`** These functions return the value of a register, i.e., one of `t1`, `clist`, `int`, `skip`, `dim`, **`\:::V`** `muskip`, or built-in TeX register. The `V` version expects a single token whereas `v` like `c` creates a csname from its argument given in braces and then evaluates it as if it was a `V`. The `\exp:w` sets off an expansion similar to an `f`-type expansion, which we terminate using `\exp_end:`. The argument is returned in braces.

```
2320 \cs_new:Npn \:::V #1 \::: #2#3
2321 {
2322     \exp_after:wN \__exp_arg_next:nnn
2323     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2324     {#1} {#2}
2325 }
2326 \cs_new:Npn \:::v #1 \::: #2#3
2327 {
2328     \exp_after:wN \__exp_arg_next:nnn
2329     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2330     {#1} {#2}
2331 }
```

(End definition for `\:::v` and `\:::V`. These functions are documented on page 42.)

\\_\\_exp\\_eval\\_register:N This function evaluates a register. Now a register might exist as one of two things: A parameter-less macro or a built-in TeX register such as \count. For the TeX registers we have to utilize a \the whereas for the macros we merely have to expand them once. The trick is to find out when to use \the and when not to. What we want here is to find out whether the token expands to something else when hit with \exp\\_after:wN. The technique is to compare the meaning of the token in question when it has been prefixed with \exp\\_not:N and the token itself. If it is a macro, the prefixed \exp\\_not:N temporarily turns it into the primitive \scan\_stop::

```
2332 \cs_new:Npn \_\_exp_eval_register:N #1
2333   {
2334     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
```

If the token was not a macro it may be a malformed variable from a c expansion in which case it is equal to the primitive \scan\_stop:. In that case we throw an error. We could let TeX do it for us but that would result in the rather obscure

```
! You can't use '\relax' after \the.
```

which while quite true doesn't give many hints as to what actually went wrong. We provide something more sensible.

```
2335   \if_meaning:w \scan_stop: #1
2336     \_\_exp_eval_error_msg:w
2337   \fi:
```

The next bit requires some explanation. The function must be initiated by \exp:w and we want to terminate this expansion chain by inserting the \exp\_end: token. However, we have to expand the register #1 before we do that. If it is a TeX register, we need to execute the sequence \exp\_after:wN \exp\_end: \tex\_the:D #1 and if it is a macro we need to execute \exp\_after:wN \exp\_end: #1. We therefore issue the longer of the two sequences and if the register is a macro, we remove the \tex\_the:D.

```
2338   \else:
2339     \exp_after:wN \use_i_ii:nnn
2340   \fi:
2341   \exp_after:wN \exp_end: \tex_the:D #1
2342 }
2343 \cs_new:Npn \_\_exp_eval_register:c #1
2344   { \exp_after:wN \_\_exp_eval_register:N \cs:w #1 \cs_end: }
```

Clean up nicely, then call the undefined control sequence. The result is an error message looking like this:

```
! Undefined control sequence.
<argument> \LaTeX3 error:
                    Erroneous variable used!
1.55 \tl_set:Nv \l_tmpa_tl {undefined_t1}

2345 \cs_new:Npn \_\_exp_eval_error_msg:w #1 \tex_the:D #2
2346   {
2347     \fi:
2348   \fi:
2349   \msg_expandable_error:nnn { kernel } { bad-variable } {#2}
2350   \exp_end:
2351 }
```

(End definition for \\_\\_exp\_eval\_register:N and \\_\\_exp\_eval\_error\_msg:w.)

## 42.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

\exp\_args:Nc  
\exp\_args:cc

(End definition for \exp\_args:Nc and \exp\_args:cc. These functions are documented on page 34.)

\exp\_args:NNc  
\exp\_args:Ncc  
\exp\_args:Nccc

Here are the functions that turn their argument into csnames but are expandable.  
2352 \cs\_new:Npn \exp\_args:NNc #1#2#3  
2353 { \exp\_after:wN #1 \exp\_after:wN #2 \cs:w # 3\cs\_end: }  
2354 \cs\_new:Npn \exp\_args:Ncc #1#2#3  
2355 { \exp\_after:wN #1 \cs:w #2 \exp\_after:wN \cs\_end: \cs:w #3 \cs\_end: }  
2356 \cs\_new:Npn \exp\_args:Nccc #1#2#3#4  
2357 {  
2358 \exp\_after:wN #1  
2359 \cs:w #2 \exp\_after:wN \cs\_end:  
2360 \cs:w #3 \exp\_after:wN \cs\_end:  
2361 \cs:w #4 \cs\_end:  
2362 }

(End definition for \exp\_args:NNc, \exp\_args:Ncc, and \exp\_args:Nccc. These functions are documented on page 36.)

\exp\_args:No  
\exp\_args:NNo  
\exp\_args>NNNo

Those lovely runs of expansion!  
2363 \cs\_new:Npn \exp\_args:No #1#2 { \exp\_after:wN #1 \exp\_after:wN {#2} }  
2364 \cs\_new:Npn \exp\_args:NNo #1#2#3  
2365 { \exp\_after:wN #1 \exp\_after:wN #2 \exp\_after:wN {#3} }  
2366 \cs\_new:Npn \exp\_args>NNNo #1#2#3#4  
2367 { \exp\_after:wN #1 \exp\_after:wN#2 \exp\_after:wN #3 \exp\_after:wN {#4} }

(End definition for \exp\_args:No, \exp\_args:NNo, and \exp\_args>NNNo. These functions are documented on page 35.)

\exp\_args:Ne

When the \expanded primitive is available, use it. Otherwise use \\_\_exp\_e:nn, defined later, to fully expand tokens.

```
2368 \cs_if_exist:NTF \tex_expanded:D
2369 {
2370   \cs_new:Npn \exp_args:Ne #1#2
2371   { \exp_after:wN #1 \tex_expanded:D { {#2} } }
2372 }
2373 {
2374   \cs_new:Npn \exp_args:Ne #1#2
2375   {
2376     \exp_after:wN #1 \exp_after:wN
2377     { \exp:w \__exp_e:nn {#2} { } }
2378   }
2379 }
```

(End definition for \exp\_args:Ne. This function is documented on page 35.)

```

\exp_args:Nf
\exp_args:NV
\exp_args:Nv
2380 \cs_new:Npn \exp_args:Nf #1#2
2381   { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
2382 \cs_new:Npn \exp_args:Nv #1#2
2383   {
2384     \exp_after:wN #1 \exp_after:wN
2385       { \exp:w \__exp_eval_register:c {#2} }
2386   }
2387 \cs_new:Npn \exp_args:NV #1#2
2388   {
2389     \exp_after:wN #1 \exp_after:wN
2390       { \exp:w \__exp_eval_register:N #2 }
2391   }

```

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 35.)

`\exp_args:NNV` Some more hand-tuned function with three arguments. If we forced that an `o` argument always has braces, we could implement `\exp_args:Nco` with less tokens and only two arguments.

```

\exp_args:NNf
\exp_args:Nco
\exp_args:NcV
\exp_args:Ncv
\exp_args:Ncf
\exp_args:NVV
2392 \cs_new:Npn \exp_args:NNV #1#2#3
2393   {
2394     \exp_after:wN #1
2395     \exp_after:wN #2
2396     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
2397   }
2398 \cs_new:Npn \exp_args:NNv #1#2#3
2399   {
2400     \exp_after:wN #1
2401     \exp_after:wN #2
2402     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
2403   }
2404 \cs_if_exist:NTF \tex_expanded:D
2405   {
2406     \cs_new:Npn \exp_args:NNe #1#2#3
2407       {
2408         \exp_after:wN #1
2409         \exp_after:wN #2
2410         \tex_expanded:D { {#3} }
2411       }
2412   }
2413 { \cs_new:Npn \exp_args:NNe { \:::N \:::e \::: } }
2414 \cs_new:Npn \exp_args:NNf #1#2#3
2415   {
2416     \exp_after:wN #1
2417     \exp_after:wN #2
2418     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2419   }
2420 \cs_new:Npn \exp_args:Nco #1#2#3
2421   {
2422     \exp_after:wN #1
2423     \cs:w #2 \exp_after:wN \cs_end:
2424     \exp_after:wN {#3}
2425   }

```

```

2426 \cs_new:Npn \exp_args:NcV #1#2#3
2427 {
2428     \exp_after:wN #1
2429     \cs:w #2 \exp_after:wN \cs_end:
2430     \exp_after:wN { \exp:w \_exp_eval_register:N #3 }
2431 }
2432 \cs_new:Npn \exp_args:Ncv #1#2#3
2433 {
2434     \exp_after:wN #1
2435     \cs:w #2 \exp_after:wN \cs_end:
2436     \exp_after:wN { \exp:w \_exp_eval_register:c {#3} }
2437 }
2438 \cs_new:Npn \exp_args:Ncf #1#2#3
2439 {
2440     \exp_after:wN #1
2441     \cs:w #2 \exp_after:wN \cs_end:
2442     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
2443 }
2444 \cs_new:Npn \exp_args:NNV #1#2#3
2445 {
2446     \exp_after:wN #1
2447     \exp_after:wN { \exp:w \exp_after:wN
2448         \_exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
2449     \exp_after:wN { \exp:w \_exp_eval_register:N #3 }
2450 }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 36.)

<code>\exp_args:NNNV</code> <code>\exp_args:NNNv</code> <code>\exp_args:NcNc</code> <code>\exp_args:NcNo</code> <code>\exp_args:Ncco</code>	A few more that we can hand-tune. <pre> 2451 \cs_new:Npn \exp_args:NNNV #1#2#3#4 2452 { 2453     \exp_after:wN #1 2454     \exp_after:wN #2 2455     \exp_after:wN #3 2456     \exp_after:wN { \exp:w \_exp_eval_register:N #4 } 2457 } 2458 \cs_new:Npn \exp_args:NNNv #1#2#3#4 2459 { 2460     \exp_after:wN #1 2461     \exp_after:wN #2 2462     \exp_after:wN #3 2463     \exp_after:wN { \exp:w \_exp_eval_register:c {#4} } 2464 } 2465 \cs_new:Npn \exp_args:NcNc #1#2#3#4 2466 { 2467     \exp_after:wN #1 2468     \cs:w #2 \exp_after:wN \cs_end: 2469     \exp_after:wN #3 2470     \cs:w #4 \cs_end: 2471 } 2472 \cs_new:Npn \exp_args:NcNo #1#2#3#4 2473 { 2474     \exp_after:wN #1 2475     \cs:w #2 \exp_after:wN \cs_end: </pre>
---	---

```

2476      \exp_after:wN #3
2477      \exp_after:wN {#4}
2478  }
2479 \cs_new:Npn \exp_args:NccO #1#2#3#4
2480  {
2481      \exp_after:wN #1
2482      \cs:w #2 \exp_after:wN \cs_end:
2483      \cs:w #3 \exp_after:wN \cs_end:
2484      \exp_after:wN {#4}
2485  }

```

(End definition for `\exp_args:NNNV` and others. These functions are documented on page 36.)

### `\exp_args:Nx`

```

2486 \cs_new_protected:Npn \exp_args:Nx #1#2
2487  { \use:x { \exp_not:N #1 {#2} } }

```

(End definition for `\exp_args:Nx`. This function is documented on page 35.)

## 42.3 Last-unbraced versions

`\__exp_arg_last_unbraced:nn` There are a few places where the last argument needs to be available unbraced. First some helper macros.

```

2488 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
2489 \cs_new:Npn \:::o_unbraced \::: #1#2
2490  { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
2491 \cs_new:Npn \:::V_unbraced \::: #1#2
2492  {
2493      \exp_after:wN \__exp_arg_last_unbraced:nn
2494      \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
2495  }
2496 \cs_new:Npn \:::v_unbraced \::: #1#2
2497  {
2498      \exp_after:wN \__exp_arg_last_unbraced:nn
2499      \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
2500  }
2501 \cs_if_exist:NTF \tex_expanded:D
2502  {
2503      \cs_new:Npn \:::e_unbraced \::: #1#2
2504      { \tex_expanded:D { \exp_not:n {#1} #2 } }
2505  }
2506  {
2507      \cs_new:Npn \:::f_unbraced \::: #1#2
2508      { \exp:w \__exp_e:nn {#2} {#1} }
2509  }
2510 \cs_new:Npn \:::x_unbraced \::: #1#2
2511  {
2512      \exp_after:wN \__exp_arg_last_unbraced:nn
2513      \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
2514  }
2515 \cs_new_protected:Npn \:::x_unbraced \::: #1#2
2516  {
2517      \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }

```

```

2518     \l__exp_internal_tl
2519 }
```

(End definition for `\_exp_arg_last_unbraced:nn` and others. These functions are documented on page 42.)

`\exp_last_unbraced:Nv`  
`\exp_last_unbraced:NV`  
`\exp_last_unbraced:Ne`  
`\exp_last_unbraced:Nf`  
`\exp_last_unbraced:NNo`  
`\exp_last_unbraced>NNV`  
`\exp_last_unbraced>NNf`  
`\exp_last_unbraced:Nco`  
`\exp_last_unbraced:NcV`  
`\exp_last_unbraced>NNNNo`  
`\exp_last_unbraced>NNNV`  
`\exp_last_unbraced>NNNf`  
`\exp_last_unbraced:Nno`  
`\exp_last_unbraced:Noo`  
`\exp_last_unbraced:Nfo`  
`\exp_last_unbraced:NnNo`  
`\exp_last_unbraced>NNNNo`  
`\exp_last_unbraced>NNNNf`  
`\exp_last_unbraced:Nx`

Now the business end: most of these are hand-tuned for speed, but the general system is in place.

```

2520 \cs_new:Npn \exp_last_unbraced:Nv #1#2 { \exp_after:wN #1 #2 }
2521 \cs_new:Npn \exp_last_unbraced:NV #1#2
2522   { \exp_after:wN #1 \exp:w \_exp_eval_register:N #2 }
2523 \cs_new:Npn \exp_last_unbraced:Ne #1#2
2524   { \exp_after:wN #1 \exp:w \_exp_eval_register:c {#2} }
2525 \cs_if_exist:NTF \tex_expanded:D
2526   {
2527     \cs_new:Npn \exp_last_unbraced:Ne #1#2
2528       { \exp_after:wN #1 \tex_expanded:D {#2} }
2529   }
2530   { \cs_new:Npn \exp_last_unbraced:Ne { \::e_unbraced \::: } }
2531 \cs_new:Npn \exp_last_unbraced:Nf #1#2
2532   { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
2533 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
2534   { \exp_after:wN #1 \exp_after:wN #2 #3 }
2535 \cs_new:Npn \exp_last_unbraced>NNV #1#2#3
2536   {
2537     \exp_after:wN #1
2538     \exp_after:wN #2
2539     \exp:w \_exp_eval_register:N #3
2540   }
2541 \cs_new:Npn \exp_last_unbraced>NNf #1#2#3
2542   {
2543     \exp_after:wN #1
2544     \exp_after:wN #2
2545     \exp:w \exp_end_continue_f:w #3
2546   }
2547 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
2548   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
2549 \cs_new:Npn \exp_last_unbraced:NcV #1#2#3
2550   {
2551     \exp_after:wN #1
2552     \cs:w #2 \exp_after:wN \cs_end:
2553     \exp:w \_exp_eval_register:N #3
2554   }
2555 \cs_new:Npn \exp_last_unbraced>NNNNo #1#2#3#4
2556   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
2557 \cs_new:Npn \exp_last_unbraced>NNNV #1#2#3#4
2558   {
2559     \exp_after:wN #1
2560     \exp_after:wN #2
2561     \exp_after:wN #3
2562     \exp:w \_exp_eval_register:N #4
2563   }
2564 \cs_new:Npn \exp_last_unbraced>NNNf #1#2#3#4
2565   {
```

```

2566     \exp_after:wN #1
2567     \exp_after:wN #2
2568     \exp_after:wN #3
2569     \exp:w \exp_end_continue_f:w #4
2570   }
2571 \cs_new:Npn \exp_last_unbraced:Nno { \::n \::o_unbraced \::: }
2572 \cs_new:Npn \exp_last_unbraced:Noo { \::o \::o_unbraced \::: }
2573 \cs_new:Npn \exp_last_unbraced:Nfo { \::f \::o_unbraced \::: }
2574 \cs_new:Npn \exp_last_unbraced:NnNo { \::n \::N \::o_unbraced \::: }
2575 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
2576   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
2577 \cs_new:Npn \exp_last_unbraced:NNNNf #1#2#3#4#5
2578   {
2579     \exp_after:wN #1
2580     \exp_after:wN #2
2581     \exp_after:wN #3
2582     \exp_after:wN #4
2583     \exp:w \exp_end_continue_f:w #5
2584   }
2585 \cs_new_protected:Npn \exp_last_unbraced:Nx { \::x_unbraced \::: }

```

(End definition for `\exp_last_unbraced:Noo` and others. These functions are documented on page 37.)

If #2 is a single token then this can be implemented as

```

\cs_new:Npn \exp_last_two_unbraced:Noo #1 #2 #3
  { \exp_after:wN \exp_after:wN \exp_after:wN #1 \exp_after:wN #2 #3 }

```

However, for robustness this is not suitable. Instead, a bit of a shuffle is used to ensure that #2 can be multiple tokens.

```

2586 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
2587   { \exp_after:wN \__exp_last_two_unbraced:noN \exp_after:wN {\#3} {\#2} #1 }
2588 \cs_new:Npn \__exp_last_two_unbraced:noN #1#2#3
2589   { \exp_after:wN #3 #2 #1 }

```

(End definition for `\exp_last_two_unbraced:Noo` and `\__exp_last_two_unbraced:noN`. This function is documented on page 38.)

## 42.4 Preventing expansion

`\__kernel_exp_not:w` At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```
2590 \cs_new_eq:NN \__kernel_exp_not:w \tex_unexpanded:D
```

(End definition for `\__kernel_exp_not:w`.)

`\exp_not:c` All these except `\exp_not:c` call the kernel-internal `\__kernel_exp_not:w` namely `\tex_unexpanded:D`.

`\exp_not:o`

`\exp_not:e`

`\exp_not:f`

`\exp_not:V`

`\exp_not:v`

```

2591 \cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }
2592 \cs_new:Npn \exp_not:o #1 { \__kernel_exp_not:w \exp_after:wN {\#1} }
2593 \cs_if_exist:NTF \tex_expanded:D
2594   {
2595     \cs_new:Npn \exp_not:e #1
2596     { \__kernel_exp_not:w \tex_expanded:D { {\#1} } } }

```

```

2597     }
2598     {
2599       \cs_new:Npn \exp_not:e
2600         { \__kernel_exp_not:w \exp_args:N \prg_do_nothing: }
2601     }
2602   \cs_new:Npn \exp_not:f #1
2603     { \__kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }
2604   \cs_new:Npn \exp_not:V #1
2605     {
2606       \__kernel_exp_not:w \exp_after:wN
2607         { \exp:w \__exp_eval_register:N #1 }
2608     }
2609   \cs_new:Npn \exp_not:v #1
2610     {
2611       \__kernel_exp_not:w \exp_after:wN
2612         { \exp:w \__exp_eval_register:c {#1} }
2613     }

```

(End definition for `\exp_not:c` and others. These functions are documented on page 38.)

## 42.5 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrarily” many expansions we need a method to invoke TeX’s expansion mechanism in such a way that (a) we are able to stop it in a controlled manner and (b) the result of what triggered the expansion in the first place is null, i.e., that we do not get any unwanted side effects. There aren’t that many possibilities in TeX; in fact the one explained below might well be the only one (as normally the result of expansion is not null).

The trick here is to make use of the fact that `\tex_roman numeral:D` expands the tokens following it when looking for a number and that its expansion is null if that number turns out to be zero or negative. So we use that to start the expansion sequence: `\exp:w` is set equal to `\tex_roman numeral:D` in `\l3basics`. To stop the expansion sequence in a controlled way all we need to provide is a constant integer zero as part of expanded tokens. As this is an integer constant it immediately stops `\tex_roman numeral:D`’s search for a number. Again, the definition of `\exp_end:` as the integer constant zero is in `\l3basics`. (Note that according to our specification all tokens we expand initiated by `\exp:w` are supposed to be expandable (as well as their replacement text in the expansion) so we will not encounter a “number” that actually result in a roman numeral being generated. Or if we do then the programmer made a mistake.)

If on the other hand we want to stop the initial expansion sequence but continue with an f-type expansion we provide the alphabetic constant ‘`^~@`’ that also represents 0 but this time TeX’s syntax for a `\langle number \rangle` continues searching for an optional space (and it continues expansion doing that) — see TeXbook page 269 for details.

```

2614   \group_begin:
2615     \tex_catcode:D '\^~@ = 13
2616   \cs_new_protected:Npn \exp_end_continue_f:w { '^^@ }

```

If the above definition ever appears outside its proper context the active character `^~@` will be executed so we turn this into an error. The test for existence covers the (unlikely) case that some other code has already defined `^~@`: this is true for example for `xmldtex.tex`.

```

2617   \if_cs_exist:N ^~@
2618   \else:

```

```

2619     \cs_new:Npn ^~@
2620     { \msg_expandable_error:nn { kernel } { bad-exp-end-f } }
2621     \fi:

```

The same but grabbing an argument to remove spaces and braces.

```

2622     \cs_new:Npn \exp_end_continue_f:nw #1 { `^~@ #1 }
2623     \group_end:

```

(*End definition for \exp:w and others. These functions are documented on page 40.*)

## 42.6 Emulating e-type expansion

When the \expanded primitive is available it is used to implement e-type expansion; otherwise we emulate it.

```

2624 \cs_if_exist:NF \tex_expanded:D
2625   {

```

\\_\_exp\_e:nn  
\\_\_exp\_e\_end:nn Repeatedly expand tokens, keeping track of fully-expanded tokens in the second argument to \\_\_exp\_e:nn; this function eventually calls \\_\_exp\_e\_end:nn to leave \exp\_end: in the input stream, followed by the result of the expansion. There are many special cases: spaces, brace groups, \noexpand, \unexpanded, \the, \primitive. While we use brace tricks \if\_false: { \fi:, the expansion of this function is always triggered by \exp:w so brace balance is eventually restored after that is hit with a single step of expansion. Otherwise we could not nest e-type expansions within each other.

```

2626     \cs_new:Npn \__exp_e:nn #1
2627     {
2628       \if_false: { \fi:
2629         \tl_if_head_is_N_type:nTF {#1}
2630           { \__exp_e:N }
2631           {
2632             \tl_if_head_is_group:nTF {#1}
2633               { \__exp_e_group:n }
2634               {
2635                 \tl_if_empty:nTF {#1}
2636                   { \exp_after:wN \__exp_e_end:nn }
2637                   { \exp_after:wN \__exp_e_space:nn }
2638                   \exp_after:wN { \if_false: } \fi:
2639               }
2640           }
2641         #1
2642       }
2643     }
2644     \cs_new:Npn \__exp_e_end:nn #1#2 { \exp_end: #2 }

```

(*End definition for \\_\_exp\_e:nn and \\_\_exp\_e\_end:nn.*)

\\_\_exp\_e\_space:nn For an explicit space character, remove it by f-expansion and put it in the (future) output.

```

2645     \cs_new:Npn \__exp_e_space:nn #1#2
2646       { \exp_args:Nf \__exp_e:nn {#1} { #2 ~ } }

```

(*End definition for \\_\_exp\_e\_space:nn.*)

`\_exp_e_group:n`  
`\_exp_e_put:nn`  
`\_exp_e_put:nnn`

For a group, expand its contents, wrap it in two pairs of braces, and call `\_exp_e_put:nn`. This function places the first item (the double-brace wrapped result) into the output. Importantly, `\tl_head:n` works even if the input contains quarks.

```

2647   \cs_new:Npn \_exp_e_group:n #1
2648   {
2649     \exp_after:wN \_exp_e_put:nn
2650     \exp_after:wN { \exp_after:wN { \exp_after:wN {
2651       \exp:w \if_false: } \fi: \_exp_e:nn {#1} { } } }
2652   }
2653   \cs_new:Npn \_exp_e_put:nn #1
2654   {
2655     \exp_args:NNo \exp_args:No \_exp_e_put:nnn
2656     { \tl_head:n {#1} } {#1}
2657   }
2658   \cs_new:Npn \_exp_e_put:nnn #1#2#3
2659     { \exp_args:No \_exp_e:nn { \use_none:n #2 } { #3 #1 } }
```

(End definition for `\_exp_e_group:n`, `\_exp_e_put:nn`, and `\_exp_e_put:nnn`.)

`\_exp_e:N`  
`\_exp_e:Nnn`  
`\_exp_e_protected:Nnn`  
`\_exp_e_expandable:Nnn`

For an N-type token, call `\_exp_e:Nnn` with arguments the *first token*, the remaining tokens to expand and what's already been expanded. If the *first token* is non-expandable, including `\protected` (`\long` or not) macros, it is put in the result by `\_exp_e_protected:Nnn`. The four special primitives `\unexpanded`, `\noexpand`, `\the`, `\primitive` are detected; otherwise the token is expanded by `\_exp_e_expandable:Nnn`.

```

2660   \cs_new:Npn \_exp_e:N #1
2661   {
2662     \exp_after:wN \_exp_e:Nnn
2663     \exp_after:wN #1
2664     \exp_after:wN { \if_false: } \fi:
2665   }
2666   \cs_new:Npn \_exp_e:Nnn #1
2667   {
2668     \if_case:w
2669       \exp_after:wN \if_meaning:w \exp_not:N #1 #1 1 ~ \fi:
2670       \token_if_protected_macro:NT #1 { 1 ~ }
2671       \token_if_protected_long_macro:NT #1 { 1 ~ }
2672       \if_meaning:w \exp_not:n #1 2 ~ \fi:
2673       \if_meaning:w \exp_not:N #1 3 ~ \fi:
2674       \if_meaning:w \tex_the:D #1 4 ~ \fi:
2675       \if_meaning:w \tex_primitive:D #1 5 ~ \fi:
2676       0 ~
2677       \exp_after:wN \_exp_e_expandable:Nnn
2678       \or: \exp_after:wN \_exp_e_protected:Nnn
2679       \or: \exp_after:wN \_exp_e_unexpanded:Nnn
2680       \or: \exp_after:wN \_exp_e_noexpand:Nnn
2681       \or: \exp_after:wN \_exp_e_the:Nnn
2682       \or: \exp_after:wN \_exp_e_primitive:Nnn
2683       \fi:
2684       #1
2685   }
2686   \cs_new:Npn \_exp_e_protected:Nnn #1#2#3
2687     { \_exp_e:nn {#2} { #3 #1 } }
2688   \cs_new:Npn \_exp_e_expandable:Nnn #1#2
2689     { \exp_args:No \_exp_e:nn { #1 #2 } }
```

(End definition for `\_exp_e:N` and others.)

```
\_exp_e_primitive:Nnn
\_exp_e_primitive_aux:NNw
\_exp_e_primitive_aux:NNnn
  \_exp_e_primitive_other:NNnn
    \_exp_e_primitive_other_aux:nNNnn
```

We don't try hard to make sensible error recovery since the error recovery of `\tex_primitive:D` when followed by something else than a primitive depends on the engine. The only valid case is when what follows is N-type. Then distinguish special primitives `\unexpanded`, `\noexpand`, `\the`, `\primitive` from other primitives. In the “other” case, the only reasonable way to check if the primitive that follows `\tex_primitive:D` is expandable is to expand and compare the before-expansion and after-expansion results. If they coincide then probably the primitive is non-expandable and should be put in the output together with `\tex_primitive:D` (one can cook up contrived counter-examples where the true `\expanded` would have an infinite loop), and otherwise one should continue expanding.

```
2690   \cs_new:Npn \_exp_e_primitive:Nnn #1#2
2691   {
2692     \if_false: { \fi:
2693       \tl_if_head_is_N_type:nTF {#2}
2694         { \_exp_e_primitive_aux:NNw #1 }
2695         {
2696           \msg_expandable_error:nnn { kernel } { e-type }
2697             { Missing-primitive-name }
2698           \_exp_e_primitive_aux:NNw #1 \c_empty_tl
2699         }
2700       #2
2701     }
2702   }
2703   \cs_new:Npn \_exp_e_primitive_aux:NNw #1#2
2704   {
2705     \exp_after:wN \_exp_e_primitive_aux:NNnn
2706     \exp_after:wN #1
2707     \exp_after:wN #2
2708     \exp_after:wN { \if_false: } \fi:
2709   }
2710   \cs_new:Npn \_exp_e_primitive_aux:NNnn #1#2
2711   {
2712     \exp_args:Nf \str_case_e:nnTF { \cs_to_str:N #2 }
2713     {
2714       { unexpanded } { \_exp_e_unexpanded:Nnn \exp_not:n }
2715       { noexpand } { \_exp_e_noexpand:Nnn \exp_not:N }
2716       { the } { \_exp_e_the:Nnn \tex_the:D }
2717       {
2718         \sys_if_engine_xetex:T { pdf }
2719         \sys_if_engine_luatex:T { pdf }
2720         primitive
2721       } { \_exp_e_primitive:Nnn #1 }
2722     }
2723     { \_exp_e_primitive_other:NNnn #1 #2 }
2724   }
2725   \cs_new:Npn \_exp_e_primitive_other:NNnn #1#2#3
2726   {
2727     \exp_args:No \_exp_e_primitive_other_aux:nNNnn
2728       { #1 #2 #3 }
2729       #1 #2 {#3}
2730   }
```

```

2731   \cs_new:Npn \__exp_e_primitive_other_aux:nNNnn #1#2#3#4#5
2732   {
2733     \str_if_eq:nnTF {#1} { #2 #3 #4 }
2734     { \__exp_e:nn {#4} { #5 #2 #3 } }
2735     { \__exp_e:nn {#1} {#5} }
2736   }

```

(End definition for `\__exp_e_primitive:Nnn` and others.)

`\__exp_e_noexpand:Nnn` The `\noexpand` primitive has no effect when followed by a token that is not N-type; otherwise `\__exp_e_put:nn` can grab the next token and put it in the result unchanged.

```

2737   \cs_new:Npn \__exp_e_noexpand:Nnn #1#2
2738   {
2739     \tl_if_head_is_N_type:nTF {#2}
2740     { \__exp_e_put:nn } { \__exp_e:nn } {#2}
2741   }

```

(End definition for `\__exp_e_noexpand:Nnn`.)

`\__exp_e_unexpanded:Nnn` `\__exp_e_unexpanded:nn` `\__exp_e_unexpanded:nN` `\__exp_e_unexpanded:N` The `\unexpanded` primitive expands and ignores any space, `\scan_stop:`, or token affected by `\exp_not:N`, then expects a brace group. Since we only support brace-balanced token lists it is impossible to support the case where the argument of `\unexpanded` starts with an implicit brace. Even though we want to expand and ignore spaces we cannot blindly f-expand because tokens affected by `\exp_not:N` should be discarded without being expanded further.

As usual distinguish four cases: brace group (the normal case, where we just put the item in the result), space (just f-expand to remove the space), empty (an error), or N-type `<token>`. In the last case call `\__exp_e_unexpanded:nN` triggered by an f-expansion. Having a non-expandable `<token>` after `\unexpanded` is an error (we recover by passing {} to `\unexpanded`; this is different from TeX because the error recovery of `\unexpanded` changes the balance of braces), unless that `<token>` is `\scan_stop:` or a space (recall that we don't implement the case of an implicit begin-group token). An expandable `<token>` is instead expanded, unless it is `\noexpand`. The latter primitive can be followed by an expandable N-type token (removed), by a non-expandable one (kept and later causing an error), by a space (removed by f-expansion), or by a brace group or nothing (later causing an error).

```

2742   \cs_new:Npn \__exp_e_unexpanded:Nnn #1 { \__exp_e_unexpanded:nn }
2743   \cs_new:Npn \__exp_e_unexpanded:nn #1
2744   {
2745     \tl_if_head_is_N_type:nTF {#1}
2746     {
2747       \exp_args:Nf \__exp_e_unexpanded:nn
2748       { \__exp_e_unexpanded:nN {#1} #1 }
2749     }
2750   {
2751     \tl_if_head_is_group:nTF {#1}
2752     { \__exp_e_put:nn }
2753     {
2754       \tl_if_empty:nTF {#1}
2755       {
2756         \msg_expandable_error:nnn
2757         { kernel } { e-type }
2758         { \unexpanded missing~brace }

```

```

2759           \__exp_e_end:nn
2760       }
2761   { \exp_args:Nf \__exp_e_unexpanded:nn }
2762 }
2763 {#1}
2764 }
2765 }
2766 \cs_new:Npn \__exp_e_unexpanded:nN #1#2
2767 {
2768     \exp_after:wN \if_meaning:w \exp_not:N #2 #2
2769         \exp_after:wN \use_i:nn
2770     \else:
2771         \exp_after:wN \use_ii:nn
2772     \fi:
2773 {
2774     \token_if_eq_catcode:NNTF #2 \c_space_token
2775     { \exp_stop_f: }
2776     {
2777         \token_if_eq_meaning:NNTF #2 \scan_stop:
2778         { \exp_stop_f: }
2779         {
2780             \msg_expandable_error:nnn
2781                 { kernel } { e-type }
2782                 { \unexpanded missing~brace }
2783             { }
2784         }
2785     }
2786 }
2787 {
2788     \token_if_eq_meaning:NNTF #2 \exp_not:N
2789     {
2790         \exp_args:No \tl_if_head_is_N_type:nT { \use_none:n #1 }
2791             { \__exp_e_unexpanded:N }
2792     }
2793     { \exp_after:wN \exp_stop_f: #2 }
2794 }
2795 }
2796 \cs_new:Npn \__exp_e_unexpanded:N #1
2797 {
2798     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 \else:
2799         \exp_after:wN \use_i:nn
2800     \fi:
2801     \exp_stop_f: #1
2802 }

```

(End definition for `\__exp_e_unexpanded:Nnn` and others.)

`\__exp_e_the:Nnn` Finally implement `\the`. Followed by anything other than an N-type  $\langle token \rangle$  this causes an error (we just let TeX make one), otherwise we test the  $\langle token \rangle$ . If the  $\langle token \rangle$  is expandable, expand it. Otherwise it could be any kind of register, or things like `\numexpr`, so there is no way to deal with all cases. Thankfully, only `\toks` data needs to be protected from expansion since everything else gives a string of characters. If the  $\langle token \rangle$  is `\toks` we find a number and unpack using the `the_toks` functions. If it is a token register we unpack it in a brace group and call `\__exp_e_put:nn` to move it to

the result. Otherwise we unpack and continue expanding (useless but safe) since it is basically impossible to have a handle on where the result of `\the` ends.

```

2803      \cs_new:Npn \__exp_e_the:Nnn #1#2
2804      {
2805          \tl_if_head_is_N_type:nTF {#2}
2806          { \if_false: { \fi: \__exp_e_the:N #2 } }
2807          { \exp_args:No \__exp_e:nn { \tex_the:D #2 } }
2808      }
2809      \cs_new:Npn \__exp_e_the:N #1
2810      {
2811          \exp_after:wN \if_meaning:w \exp_not:N #1 #1
2812          \exp_after:wN \use_i:nn
2813      \else:
2814          \exp_after:wN \use_ii:nn
2815      \fi:
2816      {
2817          \if_meaning:w \tex_toks:D #1
2818          \exp_after:wN \__exp_e_the_toks:wnn \int_value:w
2819          \exp_after:wN \__exp_e_the_toks:n
2820          \exp_after:wN { \int_value:w \if_false: } \fi:
2821      \else:
2822          \__exp_e_if_toks_register:NTF #1
2823          { \exp_after:wN \__exp_e_the_toks_reg:N }
2824          {
2825              \exp_after:wN \__exp_e:nn \exp_after:wN {
2826                  \tex_the:D \if_false: } \fi:
2827              }
2828              \exp_after:wN #1
2829          \fi:
2830      }
2831      {
2832          \exp_after:wN \__exp_e_the:Nnn \exp_after:wN ?
2833          \exp_after:wN { \exp:w \if_false: } \fi:
2834          \exp_after:wN \exp_end: #1
2835      }
2836  }
2837  \cs_new:Npn \__exp_e_the_toks_reg:N #1
2838  {
2839      \exp_after:wN \__exp_e_put:nn \exp_after:wN {
2840          \exp_after:wN {
2841              \tex_the:D \if_false: } \fi: #1 }
2842  }

```

(End definition for `\__exp_e_the:Nnn`, `\__exp_e_the:N`, and `\__exp_e_the_toks_reg:N`.)

`\__exp_e_the_toks:wnn`  
`\__exp_e_the_toks:n`  
`\__exp_e_the_toks:N`

The calling function has applied `\int_value:w` so we collect digits with `\__exp_e_the_toks:n` (which gets the token list as an argument) and `\__exp_e_the_toks:N` (which gets the first token in case it is N-type). The digits are themselves collected into an `\int_value:w` argument to `\__exp_e_the_toks:wnn`. Then that function unpacks the `\toks{number}` into the result. We include ? because `\__exp_e_put:nnn` removes one item from its second argument. Note that our approach is rather crude: in cases like `\the\toks12~34` the first `\int_value:w` removes the space and we will incorrectly unpack the `\the\toks1234`.

```

2843 \cs_new:Npn \__exp_e_the_toks:wnn #1; #
2844 {
2845     \exp_args:No \__exp_e_put:nnn
2846         { \tex_the:D \tex_toks:D #1 } { ? #2 }
2847     }
2848 \cs_new:Npn \__exp_e_the_toks:n #1
2849 {
2850     \tl_if_head_is_N_type:nTF {#1}
2851         { \exp_after:wN \__exp_e_the_toks:N \if_false: { \fi: #1 } }
2852         { ; {#1} }
2853     }
2854 \cs_new:Npn \__exp_e_the_toks:N #1
2855 {
2856     \if_int_compare:w 10 < 9 \token_to_str:N #1 \exp_stop_f:
2857         \exp_after:wN \use_i:nn
2858     \else:
2859         \exp_after:wN \use_ii:nn
2860     \fi:
2861     {
2862         #1
2863         \exp_after:wN \__exp_e_the_toks:n
2864         \exp_after:wN { \if_false: } \fi:
2865     }
2866     {
2867         \exp_after:wN ;
2868         \exp_after:wN { \if_false: } \fi: #1
2869     }
2870 }

```

(End definition for `\__exp_e_the_toks:wnn`, `\__exp_e_the_toks:n`, and `\__exp_e_the_toks:N`.)

```

\__exp_e_if_toks_register:NTF
\__exp_e_the_XeTeXinterchartoks:
\__exp_e_the_errhelp:
\__exp_e_the_everycr:
\__exp_e_the_everydisplay:
\__exp_e_the_everyeof:
\__exp_e_the_everyhbox:
\__exp_e_the_everyjob:
\__exp_e_the_everymath:
\__exp_e_the_everypar:
\__exp_e_the_everyvbox:
    \__exp_e_the_output:
\__exp_e_the_pdfpageattr:
    \__exp_e_the_pdfpageresources:
\__exp_e_the_pdfpagesattr:
\__exp_e_the_pdffpkmode:

```

We need to detect both `\toks` registers like `\toks@` in L<sup>A</sup>T<sub>E</sub>X 2<sub>&</sub> and parameters such as `\everypar`, as the result of unpacking the register should not expand further. Registers are found by `\token_if_toks_register:NTF` by inspecting the meaning. The list of parameters is finite so we just use a `\cs_if_exist:cTF` test to look up in a table. We abuse `\cs_to_str:N`'s ability to remove a leading escape character whatever it is.

```

2871 \prg_new_conditional:Npnn \__exp_e_if_toks_register:N #1 { TF }
2872 {
2873     \token_if_toks_register:NTF #1 { \prg_return_true: }
2874     {
2875         \cs_if_exist:cTF
2876         {
2877             \__exp_e_the_
2878             \exp_after:wN \cs_to_str:N
2879             \token_to_meaning:N #1
2880             :
2881         } { \prg_return_true: } { \prg_return_false: }
2882     }
2883 }
2884 \cs_new_eq:NN \__exp_e_the_XeTeXinterchartoks: ?
2885 \cs_new_eq:NN \__exp_e_the_errhelp: ?
2886 \cs_new_eq:NN \__exp_e_the_everycr: ?
2887 \cs_new_eq:NN \__exp_e_the_everydisplay: ?
2888 \cs_new_eq:NN \__exp_e_the_everyeof: ?

```

```

2889      \cs_new_eq:NN \__exp_e_the_everyhbox: ?
2890      \cs_new_eq:NN \__exp_e_the_everyjob: ?
2891      \cs_new_eq:NN \__exp_e_the_everymath: ?
2892      \cs_new_eq:NN \__exp_e_the_everypar: ?
2893      \cs_new_eq:NN \__exp_e_the_everyvbox: ?
2894      \cs_new_eq:NN \__exp_e_the_output: ?
2895      \cs_new_eq:NN \__exp_e_the_pdfpageattr: ?
2896      \cs_new_eq:NN \__exp_e_the_pdffpageresources: ?
2897      \cs_new_eq:NN \__exp_e_the_pdffpagesattr: ?
2898      \cs_new_eq:NN \__exp_e_the_pdpkmode: ?

```

(End definition for `\__exp_e_if_toks_register:NTF` and others.)

We are done emulating e-type argument expansion when `\expanded` is unavailable.

```
2899 }
```

## 42.7 Defining function variants

```
2900 ⟨@=cs⟩
```

`\s__cs_mark` Internal scan marks. No l3quark yet, so do things by hand.

```
2901 \cs_new_eq:NN \s__cs_mark \scan_stop:
2902 \cs_new_eq:NN \s__cs_stop \scan_stop:
```

(End definition for `\s__cs_mark` and `\s__cs_stop`.)

`\q__cs_recursion_stop` Internal recursion quarks. No l3quark yet, so do things by hand.

```
2903 \cs_new:Npn \q__cs_recursion_stop { \q__cs_recursion_stop }
```

(End definition for `\q__cs_recursion_stop`.)

`\__cs_use_none_delimit_by_s_stop:w` Internal scan marks.

```
2904 \cs_new:Npn \__cs_use_none_delimit_by_s_stop:w #1 \s__cs_stop { }
2905 \cs_new:Npn \__cs_use_i_delimit_by_s_stop:nw #1 #2 \s__cs_stop {#1}
2906 \cs_new:Npn \__cs_use_none_delimit_by_q_recursion_stop:w
2907     #1 \q__cs_recursion_stop { }
```

(End definition for `\__cs_use_none_delimit_by_s_stop:w`, `\__cs_use_i_delimit_by_s_stop:nw`, and `\__cs_use_none_delimit_by_q_recursion_stop:w`.)

`\cs_generate_variant:Nn` #1 : Base form of a function; e.g., `\tl_set:Nn`

`\cs_generate_variant:cn` #2 : One or more variant argument specifiers; e.g., `{Nx, c, cx}`

After making sure that the base form exists, test whether it is protected or not and define `\__cs_tmp:w` as either `\cs_new:Npx` or `\cs_new_protected:Npx`, which is then used to define all the variants (except those involving x-expansion, always protected). Split up the original base function only once, to grab its name and signature. Then we wish to iterate through the comma list of variant argument specifiers, which we first convert to a string: the reason is explained later.

```

2908 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
2909 {
2910     \__cs_generate_variant:N #1
2911     \use:x
2912     {
2913         \__cs_generate_variant:nnNN
2914         \cs_split_function:N #1

```

```

2915     \exp_not:N #1
2916     \tl_to_str:n {#2} ,
2917     \exp_not:N \scan_stop: ,
2918     \exp_not:N \q__cs_recursion_stop
2919   }
2920 }
2921 \cs_new_protected:Npn \cs_generate_variant:cn
2922   { \exp_args:Nc \cs_generate_variant:Nn }

```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 32.)

```

\__cs_generate_variant:N
\__cs_generate_variant:ww
\__cs_generate_variant:wwNw

```

The goal here is to pick up protected parent functions. There are four cases: the parent function can be a primitive or a macro, and can be expandable or not. For non-expandable primitives, all variants should be protected; skipping the `\else:` branch is safe because non-expandable primitives cannot be TeX conditionals.

The other case where variants should be protected is when the parent function is a protected macro: then `protected` appears in the meaning before the first occurrence of `macro`. The `ww` auxiliary removes everything in the meaning string after the first `ma`. We use `ma` rather than the full `macro` because the meaning of the `\firstmark` primitive (and four others) can contain an arbitrary string after a leading `firstmark:`. Then, look for `pr` in the part we extracted: no need to look for anything longer: the only strings we can have are an empty string, `\long`, `\protected`, `\protected\long`, `\first`, `\top`, `\bot`, `\splittop`, or `\splitbot`, with `\` replaced by the appropriate escape character. If `pr` appears in the part before `ma`, the first `\s__cs_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

2923 \cs_new_protected:Npx \__cs_generate_variant:N #1
2924 {
2925   \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
2926   \exp_not:N \exp_not:N #1 #1
2927   \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
2928   \exp_not:N \else:
2929     \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
2930     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
2931     \s__cs_mark
2932     \s__cs_mark \cs_new_protected:Npx
2933     \tl_to_str:n { pr }
2934     \s__cs_mark \cs_new:Npx
2935     \s__cs_stop
2936     \exp_not:N \fi:
2937   }
2938 \exp_last_unbraced:NNNNo
2939   \cs_new_protected:Npn \__cs_generate_variant:ww
2940   #1 { \tl_to_str:n { ma } } #2 \s__cs_mark
2941   { \__cs_generate_variant:wwNw #1 }
2942 \exp_last_unbraced:NNNNo
2943   \cs_new_protected:Npn \__cs_generate_variant:wwNw
2944   #1 { \tl_to_str:n { pr } } #2 \s__cs_mark #3 #4 \s__cs_stop
2945   { \cs_set_eq:NN \__cs_tmp:w #3 }

```

(End definition for `\__cs_generate_variant:N`, `\__cs_generate_variant:ww`, and `\__cs_generate_variant:wwNw`.)

```

\__cs_generate_variant:nnNN #1: Base name.
#2: Base signature.

```

#3 : Boolean.  
#4 : Base function.

If the boolean is `\c_false_bool`, the base function has no colon and we abort with an error; otherwise, set off a loop through the desired variant forms. The original function is retained as #4 for efficiency.

```

2946 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
2947 {
2948   \if_meaning:w \c_false_bool #3
2949     \msg_error:nnx { kernel } { missing-colon }
2950     { \token_to_str:c {#1} }
2951     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2952   \fi:
2953   \__cs_generate_variant:Nnnw #4 {#1}{#2}
2954 }
```

(End definition for `\__cs_generate_variant:nnNN`)

`\__cs_generate_variant:Nnnw` #1 : Base function.  
#2 : Base name.  
#3 : Base signature.  
#4 : Beginning of variant signature.

First check whether to terminate the loop over variant forms. Then, for each variant form, construct a new function name using the original base name, the variant signature consisting of  $l$  letters and the last  $k - l$  letters of the base signature (of length  $k$ ). For example, for a base function `\prop_put:Nnn` which needs a cV variant form, we want the new signature to be `cVn`.

There are further subtleties:

- In `\cs_generate_variant:Nn \foo:nnTF {xxTF}`, we must define `\foo:xxTF` using `\exp_args:Nxx`, rather than a hypothetical `\exp_args:NxxTF`. Thus, we wish to trim a common trailing part from the base signature and the variant signature.
- In `\cs_generate_variant:Nn \foo:on {ox}`, the function `\foo:ox` must be defined using `\exp_args:Nnx`, not `\exp_args:Nox`, to avoid double o expansion.
- Lastly, `\cs_generate_variant:Nn \foo:on {xn}` must trigger an error, because we do not have a means to replace o-expansion by x-expansion. More generally, we can only convert N to c, or convert n to V, v, o, f, x.

All this boils down to a few rules. Only n and N-type arguments can be replaced by `\cs_generate_variant:Nn`. Other argument types are allowed to be passed unchanged from the base form to the variant: in the process they are changed to n except for N and p-type arguments. A common trailing part is ignored.

We compare the base and variant signatures one character at a time within x-expansion. The result is given to `\__cs_generate_variant:wwNN` (defined later) in the form `<processed variant signature> \s__cs_mark <errors> \s__cs_stop <base function> <new function>`. If all went well, `<errors>` is empty; otherwise, it is a kernel error message and some clean-up code.

Note the space after #3 and after the following brace group. Those are ignored by TeX when fetching the last argument for `\__cs_generate_variant_loop:nNwN`, but can be used as a delimiter for `\__cs_generate_variant_loop_end:nwwwNNnn`.

```
2955 \cs_new_protected:Npn \__cs_generate_variant:Nnnw #1#2#3#4 ,
```

```

2956      {
2957      \if_meaning:w \scan_stop: #4
2958          \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
2959      \fi:
2960      \use:x
2961      {
2962          \exp_not:N \__cs_generate_variant:wwNN
2963          \__cs_generate_variant_loop:nNwN { }
2964          #4
2965          \__cs_generate_variant_loop_end:nwwwNNnn
2966          \s__cs_mark
2967          #3 ~
2968          { ~ { } \fi: \__cs_generate_variant_loop_long:wNNnn } ~
2969          { }
2970          \s__cs_stop
2971          \exp_not:N #1 {#2} {#4}
2972      }
2973      \__cs_generate_variant:Nnnw #1 {#2} {#3}
2974  }

```

(End definition for `\__cs_generate_variant:Nnnw`.)

`\__cs_generate_variant_loop:nNwN`  
`\__cs_generate_variant_loop_base:N`  
`\__cs_generate_variant_loop_same:w`  
`\__cs_generate_variant_loop_end:nwwwNNnn`  
`\__cs_generate_variant_loop_long:wNNnn`  
`\__cs_generate_variant_loop_invalid>NNwNNnn`  
`\__cs_generate_variant_loop_special>NNwNNnn`

#1 : Last few consecutive letters common between the base and variant (more precisely, `\__cs_generate_variant_same:N <letter>` for each letter).  
#2 : Next variant letter.  
#3 : Remainder of variant form.  
#4 : Next base letter.

The first argument is populated by `\__cs_generate_variant_loop_same:w` when a variant letter and a base letter match. It is flushed into the input stream whenever the two letters are different: if the loop ends before, the argument is dropped, which means that trailing common letters are ignored.

The case where the two letters are different is only allowed if the base is `N` and the variant is `c`, or when the base is `n` and the variant is `o`, `V`, `v`, `f` or `x`. Otherwise, call `\__cs_generate_variant_loop_invalid>NNwNNnn` to remove the end of the loop, get arguments at the end of the loop, and place an appropriate error message as a second argument of `\__cs_generate_variant:wwNN`. If the letters are distinct and the base letter is indeed `n` or `N`, leave in the input stream whatever argument #1 was collected, and the next variant letter #2, then loop by calling `\__cs_generate_variant_loop:nNwN`.

The loop can stop in three ways.

- If the end of the variant form is encountered first, #2 is `\__cs_generate_variant_loop_end:nwwwNNnn` (expanded by the conditional `\if:w`), which inserts some tokens to end the conditional; grabs the `<base name>` as #7, the `<variant signature>` #8, the `<next base letter>` #1 and the part #3 of the base signature that wasn't read yet; and combines those into the `<new function>` to be defined.
- If the end of the base form is encountered first, #4 is `~{} \fi:` which ends the conditional (with an empty expansion), followed by `\__cs_generate_variant_loop_long:wNNnn`, which places an error as the second argument of `\__cs_generate_variant:wwNN`.
- The loop can be interrupted early if the requested expansion is unavailable, namely when the variant and base letters differ and the base is not the right one (`n` or `N` to

support the variant). In that case too an error is placed as the second argument of `\__cs_generate_variant:wwNN`.

Note that if the variant form has the same length as the base form, #2 is as described in the first point, and #4 as described in the second point above. The `\__cs_generate_variant_loop_end:nwwwNNnn` breaking function takes the empty brace group in #4 as its first argument: this empty brace group produces the correct signature for the full variant.

```

2975 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \s__cs_mark #4
2976 {
2977     \if:w #2 #4
2978         \exp_after:wN \__cs_generate_variant_loop_same:w
2979     \else:
2980         \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
2981             \if:w 0
2982                 \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
2983                 \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
2984                 0
2985                 \__cs_generate_variant_loop_special:NNwNNnn #4#2
2986             \else:
2987                 \__cs_generate_variant_loop_invalid:NNwNNnn #4#2
2988             \fi:
2989             \fi:
2990         \fi:
2991         #1
2992         \prg_do_nothing:
2993         #2
2994         \__cs_generate_variant_loop:nNwN { } #3 \s__cs_mark
2995     }
2996 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
2997 {
2998     \if:w c #1 N \else:
2999         \if:w o #1 n \else:
3000             \if:w V #1 n \else:
3001                 \if:w v #1 n \else:
3002                     \if:w f #1 n \else:
3003                         \if:w e #1 n \else:
3004                             \if:w x #1 n \else:
3005                                 \if:w n #1 n \else:
3006                                     \if:w N #1 N \else:
3007                                         \scan_stop:
3008                                         \fi:
3009                                         \fi:
3010                                         \fi:
3011                                         \fi:
3012                                         \fi:
3013                                         \fi:
3014                                         \fi:
3015                                         \fi:
3016                                         \fi:
3017     }
3018 \cs_new:Npn \__cs_generate_variant_loop_same:w
3019     #1 \prg_do_nothing: #2#3#4
3020     { #3 { #1 \__cs_generate_variant_same:N #2 } }
```

```

3021 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
3022     #1#2 \s__cs_mark #3 ~ #4 \s__cs_stop #5#6#7#8
3023 {
3024     \scan_stop: \scan_stop: \fi:
3025     \s__cs_mark \s__cs_stop
3026     \exp_not:N #6
3027     \exp_not:c { #7 : #8 #1 #3 }
3028 }
3029 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \s__cs_stop #2#3#4#5
3030 {
3031     \exp_not:n
3032     {
3033         \s__cs_mark
3034         \msg_error:nnxx { kernel } { variant-too-long }
3035         {#5} { \token_to_str:N #3 }
3036         \use_none:nnn
3037         \s__cs_stop
3038         #3
3039         #3
3040     }
3041 }
3042 \cs_new:Npn \__cs_generate_variant_loop_invalid>NNwNNnn
3043     #1#2 \fi: \fi: \fi: #3 \s__cs_stop #4#5#6#7
3044 {
3045     \fi: \fi: \fi:
3046     \exp_not:n
3047     {
3048         \s__cs_mark
3049         \msg_error:nnxxxx { kernel } { invalid-variant }
3050         {#7} { \token_to_str:N #5 } {#1} {#2}
3051         \use_none:nnn
3052         \s__cs_stop
3053         #5
3054         #5
3055     }
3056 }
3057 \cs_new:Npn \__cs_generate_variant_loop_special>NNwNNnn
3058     #1#2#3 \s__cs_stop #4#5#6#7
3059 {
3060     #3 \s__cs_stop #4 #5 {#6} {#7}
3061     \exp_not:n
3062     {
3063         \msg_error:nnxxxx
3064         { kernel } { deprecated-variant }
3065         {#7} { \token_to_str:N #5 } {#1} {#2}
3066     }
3067 }

```

(End definition for `\__cs_generate_variant_loop:nNwN` and others.)

`\__cs_generate_variant_same:N`

When the base and variant letters are identical, don't do any expansion. For most argument types, we can use the `n`-type no-expansion, but the `N` and `p` types require a slightly different behaviour with respect to braces. For `V`-type this function could output `N` to avoid adding useless braces but that is not a problem.

```

3068 \cs_new:Npn \__cs_generate_variant_same:N #1
3069 {
3070     \if:w N #1 #1 \else:
3071         \if:w p #1 #1 \else:
3072             \token_to_str:N n
3073             \if:w n #1 \else:
3074                 \__cs_generate_variant_loop_special:NNwNNnn #1#1
3075             \fi:
3076         \fi:
3077     \fi:
3078 }

```

(End definition for `\__cs_generate_variant_same:N`.)

`\__cs_generate_variant:wwNN` If the variant form has already been defined, log its existence (provided `log-functions` is active). Otherwise, make sure that the `\exp_args:N #3` form is defined, and if it contains `x`, change `\__cs_tmp:w` locally to `\cs_new_protected:Npx`. Then define the variant by combining the `\exp_args:N #3` variant and the base function.

```

3079 \cs_new_protected:Npn \__cs_generate_variant:wwNN
3080     #1 \s__cs_mark #2 \s__cs_stop #3#4
3081 {
3082     #2
3083     \cs_if_free:NT #4
3084     {
3085         \group_begin:
3086         \__cs_generate_internal_variant:n {#1}
3087         \__cs_tmp:w #4 { \exp_not:c { \exp_args:N #1 } \exp_not:N #3 }
3088         \group_end:
3089     }
3090 }

```

(End definition for `\__cs_generate_variant:wwNN`.)

`\__cs_generate_internal_variant:n` First test for the presence of `x` (this is where working with strings makes our lives easier), as the result should be protected, and the next variant to be defined using that internal variant should be protected (done by setting `\__cs_tmp:w`). Then call `\__cs_generate_internal_variant>NNn` with arguments `\cs_new_protected:cpx \use:x` (for protected) or `\cs_new:cpx \tex_expanded:D` (expandable) and the signature. If `p` appears in the signature, or if the function to be defined is expandable and the primitive `\expanded` is not available, or if there are more than 8 arguments, call some fall-back code that just puts the appropriate `\::` commands. Otherwise, call `\__cs_generate_internal_one_go:NNn` to construct the `\exp_args:N...` function as a macro taking up to 9 arguments and expanding them using `\use:x` or `\tex_expanded:D`.

```

3091 \cs_new_protected:Npx \__cs_generate_internal_variant:n #1
3092 {
3093     \exp_not:N \__cs_generate_internal_variant:wwNnw
3094     #1 \s__cs_mark
3095     { \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx }
3096     \cs_new_protected:cpx
3097     \use:x
3098     \token_to_str:N x \s__cs_mark
3099     { }
3100     \cs_new:cpx

```

```

3101      \exp_not:N \tex_expanded:D
3102      \s__cs_stop
3103      {#1}
3104  }
3105 \exp_last_unbraced:NNNNo
3106   \cs_new_protected:Npn \__cs_generate_internal_variant:wwNwn #1
3107   { \token_to_str:N x } #2 \s__cs_mark #3#4#5#6 \s__cs_stop #7
3108  {
3109  #3
3110  \cs_if_free:cT { exp_args:N #7 }
3111  { \__cs_generate_internal_variant>NNn #4 #5 {#7} }
3112  }
3113 \cs_set_protected:Npn \__cs_tmp:w #1
3114  {
3115  \cs_new_protected:Npn \__cs_generate_internal_variant>NNn ##1##2##3
3116  {
3117  \if_catcode:w X \use_none:nnnnnnnn ##3
3118    \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3119    \prg_do_nothing: \prg_do_nothing: \prg_do_nothing:
3120    \prg_do_nothing: \prg_do_nothing: X
3121    \exp_after:wN \__cs_generate_internal_test:Nw \exp_after:wN ##2
3122  \else:
3123    \exp_after:wN \__cs_generate_internal_test_aux:w \exp_after:wN #1
3124  \fi:
3125  ##3
3126  \s__cs_mark
3127  {
3128  \use:x
3129  {
3130  ##1 { exp_args:N ##3 }
3131  { \__cs_generate_internal_variant_loop:n ##3 { : \use_i:nn } }
3132  }
3133  }
3134  #1
3135  \s__cs_mark
3136  { \exp_not:n { \__cs_generate_internal_one_go>NNn ##1 ##2 {##3} } }
3137  \s__cs_stop
3138  }
3139 \cs_new_protected:Npn \__cs_generate_internal_test_aux:w
3140  ##1 #1 ##2 \s__cs_mark ##3 ##4 \s__cs_stop {##3}
3141 \cs_if_exist:NTF \tex_expanded:D
3142  {
3143  \cs_new_eq:NN \__cs_generate_internal_test:Nw
3144  \__cs_generate_internal_test_aux:w
3145  }
3146  {
3147  \cs_new_protected:Npn \__cs_generate_internal_test:Nw ##1
3148  {
3149  \if_meaning:w \tex_expanded:D ##1
3150  \exp_after:wN \__cs_generate_internal_test_aux:w
3151  \exp_after:wN #1
3152  \else:
3153  \exp_after:wN \__cs_generate_internal_test_aux:w
3154  \fi:

```

```

3155         }
3156     }
3157   }
3158 \exp_args:No \_cs_tmp:w { \token_to_str:N p }
3159 \cs_new_protected:Npn \_cs_generate_internal_one_go:NNn #1#2#3
3160   {
3161     \_cs_generate_internal_loop:nwnnw
3162     { \exp_not:N ##1 } 1 . { } { }
3163     #3 { ? \_cs_generate_internal_end:w } X ;
3164     23456789 { ? \_cs_generate_internal_long:w } ;
3165     #1 #2 {#3}
3166   }
3167 \cs_new_protected:Npn \_cs_generate_internal_loop:nwnnw #1#2 . #3#4#5#6 ; #7
3168   {
3169     \use_none:n #5
3170     \use_none:n #7
3171     \cs_if_exist_use:cF { \_cs_generate_internal_#5:NN }
3172     { \_cs_generate_internal_other:NN }
3173     #5 #7
3174     #7 .
3175     { #3 #1 } { #4 ## #2 }
3176     #6 ;
3177   }
3178 \cs_new_protected:Npn \_cs_generate_internal_N:NN #1#2
3179   { \_cs_generate_internal_loop:nwnnw { \exp_not:N ##2 } }
3180 \cs_new_protected:Npn \_cs_generate_internal_c:NN #1#2
3181   { \exp_args:No \_cs_generate_internal_loop:nwnnw { \exp_not:c {##2} } }
3182 \cs_new_protected:Npn \_cs_generate_internal_n:NN #1#2
3183   { \_cs_generate_internal_loop:nwnnw { { \exp_not:n {##2} } } }
3184 \cs_new_protected:Npn \_cs_generate_internal_x:NN #1#2
3185   { \_cs_generate_internal_loop:nwnnw { {##2} } }
3186 \cs_new_protected:Npn \_cs_generate_internal_other:NN #1#2
3187   {
3188     \exp_args:No \_cs_generate_internal_loop:nwnnw
3189     {
3190       \exp_after:wN
3191       {
3192         \exp:w \exp_args:NNc \exp_after:wN \exp_end:
3193         { \exp_not:#1 } {##2}
3194       }
3195     }
3196   }
3197 \cs_new_protected:Npn \_cs_generate_internal_end:w #1 . #2#3#4 ; #5 ; #6#7#8
3198   { #6 { \exp_args:N #8 } #3 { #7 {#2} } }
3199 \cs_new_protected:Npn \_cs_generate_internal_long:w #1 N #2#3 . #4#5#6#
3200   {
3201     \exp_args:Nx \_cs_generate_internal_long:nnnNNn
3202     { \_cs_generate_internal_variant_loop:n #2 #6 { : \use_i:nn } }
3203     {#4} {#5}
3204   }
3205 \cs_new:Npn \_cs_generate_internal_long:nnnNNn #1#2#3#4 ; ; #5#6#7
3206   { #5 { \exp_args:N #7 } #3 { #6 { \exp_not:n {#1} {#2} } } }

```

This command grabs char by char outputting `\::#1` (not expanded further). We avoid tests by putting a trailing `:` `\use_i:nn`, which leaves `\cs_end:` and removes the looping

macro. The colon is in fact also turned into `\:::` so that the required structure for `\exp_args:N...` commands is correctly terminated.

```

3207 \cs_new:Npn \__cs_generate_internal_variant_loop:n #1
3208   {
3209     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
3210     \__cs_generate_internal_variant_loop:n
3211   }

```

*(End definition for `\__cs_generate_internal_variant:n` and `\__cs_generate_internal_variant_loop:n`.)*

#### `\prg_generate_conditional_variant:Nnn`

```

\__cs_generate_variant:nnNnn
\__cs_generate_variant:w
\__cs_generate_variant:n
\__cs_generate_variant_p_form:nnn
\__cs_generate_variant_T_form:nnn
\__cs_generate_variant_F_form:nnn
\__cs_generate_variant_TF_form:nnn

3212 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
3213   {
3214     \use:x
3215     {
3216       \__cs_generate_variant:nnNnn
3217         \cs_split_function:N #1
3218     }
3219   }
3220 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3221   {
3222     \if_meaning:w \c_false_bool #3
3223     \msg_error:nnx { kernel } { missing-colon }
3224     { \token_to_str:c {#1} }
3225     \__cs_use_i_delimit_by_s_stop:nw
3226   \fi:
3227   \exp_after:wN \__cs_generate_variant:w
3228   \tl_to_str:n {#5} , \scan_stop: , \q__cs_recursion_stop
3229   \__cs_use_none_delimit_by_s_stop:w \s__cs_mark {#1} {#2} {#4} \s__cs_stop
3230 }
3231 \cs_new_protected:Npn \__cs_generate_variant:w
3232   #1 , #2 \s__cs_mark #3#4#5
3233   {
3234     \if_meaning:w \scan_stop: #1 \scan_stop:
3235     \if_meaning:w \q__cs_nil #1 \q__cs_nil
3236       \use_i:nnn
3237     \fi:
3238     \exp_after:wN \__cs_use_none_delimit_by_q_recursion_stop:w
3239   \else:
3240     \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
3241     { {#3} {#4} {#5} }
3242     {
3243       \msg_error:nnxx
3244         { kernel } { conditional-form-unknown }
3245         {#1} { \token_to_str:c { #3 : #4 } }
3246     }
3247   \fi:
3248   \__cs_generate_variant:w #2 \s__cs_mark {#3} {#4} {#5}
3249 }
3250 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
3251   { \cs_generate_variant:cn { #1 _p : #2 } }
3252 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
3253   { \cs_generate_variant:cn { #1 : #2 T } }

```

```

3254 \cs_new_protected:Npn \__cs_generate_variant_F_form:n #1#2
3255   { \cs_generate_variant:cn { #1 : #2 F } }
3256 \cs_new_protected:Npn \__cs_generate_variant_TF_form:n #1#2
3257   { \cs_generate_variant:cn { #1 : #2 TF } }

```

(End definition for `\prg_generate_conditional_variant:Nnn` and others. This function is documented on page 64.)

### \exp\_args\_generate:n

This function is not used in the kernel hence we can use functions that are defined in later modules. It also does not need to be fast so use inline mappings. For each requested variant we check that there are no characters besides `NnpcofVvx`, in particular that there are no spaces. Then we just call the internal function.

```

3258 \cs_new_protected:Npn \exp_args_generate:n #1
3259   {
3260     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {##1} }
3261     {
3262       \str_map_inline:nn {##1}
3263       {
3264         \str_if_in:nnF { NnpcofVvx } {####1}
3265         {
3266           \msg_error:nnnn { kernel } { invalid-exp-args }
3267           {####1} {##1}
3268           \str_map_break:n { \use_none:nn }
3269         }
3270       }
3271       \__cs_generate_internal_variant:n {##1}
3272     }
3273   }

```

(End definition for `\exp_args_generate:n`. This function is documented on page 306.)

## 42.8 Definitions with the automated technique

Some of these could be done more efficiently, but the complexity of coding then becomes an issue. Notice that the auto-generated functions actually take no arguments themselves.

```

\exp_args:Nnc
\exp_args:Nno
\exp_args:Nnv
\exp_args:Nnv
\exp_args:Nne
\exp_args:Nnf
\exp_args:Noc
\exp_args:Noo
\exp_args:Nof
\exp_args:NVo
\exp_args:Nfo
\exp_args:Nfo
\exp_args:Nff
\exp_args:Nff
\exp_args:Nee
\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx

```

Here are the actual function definitions, using the helper functions above. The group is used because `\__cs_generate_internal_variant:n` redefines `\__cs_tmp:w` locally.

```

3274 \cs_set_protected:Npn \__cs_tmp:w #1
3275   {
3276     \group_begin:
3277     \exp_args:No \__cs_generate_internal_variant:n
3278       { \tl_to_str:n {##1} }
3279     \group_end:
3280   }
3281 \__cs_tmp:w { nc }
3282 \__cs_tmp:w { no }
3283 \__cs_tmp:w { nv }
3284 \__cs_tmp:w { nv }
3285 \__cs_tmp:w { ne }
3286 \__cs_tmp:w { nf }
3287 \__cs_tmp:w { oc }
3288 \__cs_tmp:w { oo }

```

```

3289 \__cs_tmp:w { of }
3290 \__cs_tmp:w { Vo }
3291 \__cs_tmp:w { fo }
3292 \__cs_tmp:w { ff }
3293 \__cs_tmp:w { ee }
3294 \__cs_tmp:w { Nx }
3295 \__cs_tmp:w { cx }
3296 \__cs_tmp:w { nx }
3297 \__cs_tmp:w { ox }
3298 \__cs_tmp:w { xo }
3299 \__cs_tmp:w { xx }

```

(End definition for `\exp_args:Nnc` and others. These functions are documented on page 36.)

```

\exp_args:NNcf
\exp_args:NNno
\exp_args:NNnV
\exp_args:NNoo
\exp_args:NNVV
\exp_args:Ncno
\exp_args:NcnV
\exp_args:Ncoo
\exp_args:NcVV
\exp_args:Nnnc
\exp_args:Nnno
\exp_args:Nnnf
\exp_args:Nfff
\exp_args:Nooo
\exp_args:Noof
\exp_args:Nffo
\exp_args:Neee
\exp_args:NNNx
\exp_args:NNnx
\exp_args:NNox
\exp_args:Nccx
\exp_args:Ncnx
\exp_args:Nnnx
\exp_args:Nnox
\exp_args:Noox

```

(End definition for `\exp_args:NNcf` and others. These functions are documented on page 37.)

```
3325 </package>
```

# Chapter 43

## I3sort implementation

```
3326  <*package>
3327  <@@=sort>
```

### 43.1 Variables

\g\_\_sort\_internal\_seq  
\g\_\_sort\_internal\_tl

Sorting happens in a group; the result is stored in those global variables before being copied outside the group to the proper places. For seq and tl this is more efficient than using \use:x (or some \exp\_args:NNNx) to smuggle the definition outside the group since TeX does not need to re-read tokens. For clist we don't gain anything since the result is converted from seq to clist anyways.

```
3328  \seq_new:N \g__sort_internal_seq
3329  \tl_new:N \g__sort_internal_tl
```

(End definition for \g\_\_sort\_internal\_seq and \g\_\_sort\_internal\_tl.)

\l\_\_sort\_length\_int  
\l\_\_sort\_min\_int  
\l\_\_sort\_top\_int  
\l\_\_sort\_max\_int  
\l\_\_sort\_true\_max\_int

The sequence has \l\_\_sort\_length\_int items and is stored from \l\_\_sort\_min\_int to \l\_\_sort\_top\_int - 1. While reading the sequence in memory, we check that \l\_\_sort\_top\_int remains at most \l\_\_sort\_max\_int, precomputed by \\_\_sort\_compute\_range:. That bound is such that the merge sort only uses \toks registers less than \l\_\_sort\_true\_max\_int, namely those that have not been allocated for use in other code: the user's comparison code could alter these.

```
3330  \int_new:N \l__sort_length_int
3331  \int_new:N \l__sort_min_int
3332  \int_new:N \l__sort_top_int
3333  \int_new:N \l__sort_max_int
3334  \int_new:N \l__sort_true_max_int
```

(End definition for \l\_\_sort\_length\_int and others.)

\l\_\_sort\_block\_int

Merge sort is done in several passes. In each pass, blocks of size \l\_\_sort\_block\_int are merged in pairs. The block size starts at 1, and, for a length in the range  $[2^k + 1, 2^{k+1}]$ , reaches  $2^k$  in the last pass.

```
3335  \int_new:N \l__sort_block_int
```

(End definition for \l\_\_sort\_block\_int.)

\l\_\_sort\_begin\_int When merging two blocks, \l\_\_sort\_begin\_int marks the lowest index in the two blocks, and \l\_\_sort\_end\_int marks the highest index, plus 1.

```
3336 \int_new:N \l__sort_begin_int
3337 \int_new:N \l__sort_end_int
```

(End definition for \l\_\_sort\_begin\_int and \l\_\_sort\_end\_int.)

\l\_\_sort\_A\_int \l\_\_sort\_B\_int \l\_\_sort\_C\_int When merging two blocks (whose end-points are beg and end), A starts from the high end of the low block, and decreases until reaching beg. The index B starts from the top of the range and marks the register in which a sorted item should be put. Finally, C points to the copy of the high block in the interval of registers starting at \l\_\_sort\_length\_int, upwards. C starts from the upper limit of that range.

```
3338 \int_new:N \l__sort_A_int
3339 \int_new:N \l__sort_B_int
3340 \int_new:N \l__sort_C_int
```

(End definition for \l\_\_sort\_A\_int, \l\_\_sort\_B\_int, and \l\_\_sort\_C\_int.)

\s\_\_sort\_mark Internal scan marks.

```
3341 \scan_new:N \s__sort_mark
3342 \scan_new:N \s__sort_stop
```

(End definition for \s\_\_sort\_mark and \s\_\_sort\_stop.)

## 43.2 Finding available \toks registers

\\_\\_sort\_shrink\_range: \\_\\_sort\_shrink\_range\_loop: After \\_\\_sort\_compute\_range: (defined below) determines that \toks registers between \l\_\_sort\_min\_int (included) and \l\_\_sort\_true\_max\_int (excluded) have not yet been assigned, \\_\\_sort\_shrink\_range: computes \l\_\_sort\_max\_int to reflect the need for a buffer when merging blocks in the merge sort. Given  $2^n \leq A \leq 2^n + 2^{n-1}$  registers we can sort  $\lfloor A/2 \rfloor + 2^{n-2}$  items while if we have  $2^n + 2^{n-1} \leq A \leq 2^{n+1}$  registers we can sort  $A - 2^{n-1}$  items. We first find out a power  $2^n$  such that  $2^n \leq A \leq 2^{n+1}$  by repeatedly halving \l\_\_sort\_block\_int, starting at  $2^{15}$  or  $2^{14}$  namely half the total number of registers, then we use the formulas and set \l\_\_sort\_max\_int.

```
3343 \cs_new_protected:Npn \_\_sort_shrink_range:
3344 {
3345     \int_set:Nn \l__sort_A_int
3346         { \l__sort_true_max_int - \l__sort_min_int + 1 }
3347     \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
3348     \_\_sort_shrink_range_loop:
3349     \int_set:Nn \l__sort_max_int
3350     {
3351         \int_compare:nNnTF
3352             { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
3353             {
3354                 \l__sort_min_int
3355                 + ( \l__sort_A_int - 1 ) / 2
3356                 + \l__sort_block_int / 4
3357                 - 1
3358             }
3359             { \l__sort_true_max_int - \l__sort_block_int / 2 }
3360     }
```

```

3361     }
3362 \cs_new_protected:Npn \__sort_shrink_range_loop:
3363 {
3364     \if_int_compare:w \l__sort_A_int < \l__sort_block_int
3365         \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
3366         \exp_after:wN \__sort_shrink_range_loop:
3367     \fi:
3368 }

```

(End definition for `\__sort_shrink_range:` and `\__sort_shrink_range_loop::`)

`\__sort_compute_range:`  
`\__sort_redefine_compute_range:`  
`\c__sort_max_length_int`

First find out what `\toks` have not yet been assigned. There are many cases. In L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> with no package, available `\toks` range from `\count15+1` to `\c_max_register_int` included (this was not altered despite the 2015 changes). When `\loctoks` is defined, namely in plain (e)T<sub>E</sub>X, or when the package etex is loaded in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>, redefine `\__sort_compute_range:` to use the range `\count265` to `\count275 - 1`. The elocalloc package also defines `\loctoks` but uses yet another number for the upper bound, namely `\e@alloc@top` (minus one). We must check for `\loctoks` every time a sorting function is called, as etex or elocalloc could be loaded.

In ConT<sub>E</sub>Xt MkIV the range is from `\c_syst_last_allocated_toks+1` to `\c_max_register_int`, and in MkII it is from `\lastallocatedtoks+1` to `\c_max_register_int`. In all these cases, call `\__sort_shrink_range::`.

```

3369 \cs_new_protected:Npn \__sort_compute_range:
3370 {
3371     \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
3372     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3373     \__sort_shrink_range:
3374     \if_meaning:w \loctoks \tex_undefined:D \else:
3375         \if_meaning:w \loctoks \scan_stop: \else:
3376             \__sort_redefine_compute_range:
3377             \__sort_compute_range:
3378         \fi:
3379     \fi:
3380 }
3381 \cs_new_protected:Npn \__sort_redefine_compute_range:
3382 {
3383     \cs_if_exist:cTF { ver@elocalloc.sty }
3384     {
3385         \cs_gset_protected:Npn \__sort_compute_range:
3386         {
3387             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3388             \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
3389             \__sort_shrink_range:
3390         }
3391     }
3392     {
3393         \cs_gset_protected:Npn \__sort_compute_range:
3394         {
3395             \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
3396             \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
3397             \__sort_shrink_range:
3398         }
3399 }

```

```

3400      }
3401  \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
3402  \tl_map_inline:nn { \lastalloctoks \c_syst_last_allocated_toks }
3403  {
3404      \cs_if_exist:NT #1
3405      {
3406          \cs_gset_protected:Npn \__sort_compute_range:
3407          {
3408              \int_set:Nn \l__sort_min_int { #1 + 1 }
3409              \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
3410              \__sort_shrink_range:
3411          }
3412      }
3413  }

```

(End definition for `\__sort_compute_range:`, `\__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

### 43.3 Protected user commands

`\__sort_main:NNNn`

Sorting happens in three steps. First store items in `\toks` registers ranging from `\l__sort_min_int` to `\l__sort_top_int - 1`, while checking that the list is not too long. If we reach the maximum length, that's an error; exit the group. Secondly, sort the array of `\toks` registers, using the user-defined sorting function: `\__sort_level:` calls `\__sort_compare:nn` as needed. Finally, unpack the `\toks` registers (now sorted) into the target `tl`, or into `\g__sort_internal_seq` for seq and `clist`. This is done by `\__sort_seq:NNNNn` and `\__sort_tl:NNn`.

```

3414  \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
3415  {
3416      \__sort_disable_toksdef:
3417      \__sort_compute_range:
3418      \int_set_eq:NN \l__sort_top_int \l__sort_min_int
3419      #1 #3
3420      {
3421          \if_int_compare:w \l__sort_top_int = \l__sort_max_int
3422              \__sort_too_long_error>NNw #2 #3
3423          \fi:
3424          \tex_toks:D \l__sort_top_int {##1}
3425          \int_incr:N \l__sort_top_int
3426      }
3427      \int_set:Nn \l__sort_length_int
3428      { \l__sort_top_int - \l__sort_min_int }
3429      \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
3430      \int_set:Nn \l__sort_block_int { 1 }
3431      \__sort_level:
3432  }

```

(End definition for `\__sort_main:NNNn`.)

`\tl_sort:Nn`  
`\tl_sort:cn`  
`\tl_gsort:Nn`  
`\tl_gsort:cn`  
`\__sort_tl:NNn`  
`\__sort_tl_toks:w`

Call the main sorting function then unpack `\toks` registers outside the group into the target token list. The unpacking is done by `\__sort_tl_toks:w`; registers are numbered from `\l__sort_min_int` to `\l__sort_top_int - 1`. For expansion behaviour we need

a couple of primitives. The `\tl_gclear:N` reduces memory usage. The `\prg_break_point:` is used by `\_sort_main:NNNn` when the list is too long.

```

3433 \cs_new_protected:Npn \tl_sort:Nn { \_sort_tl:NNn \tl_set_eq:NN }
3434 \cs_generate_variant:Nn \tl_sort:Nn { c }
3435 \cs_new_protected:Npn \tl_gsort:Nn { \_sort_tl:NNn \tl_gset_eq:NN }
3436 \cs_generate_variant:Nn \tl_gsort:Nn { c }
3437 \cs_new_protected:Npn \_sort_tl:NNn #1#2#3
3438 {
3439     \group_begin:
3440         \_sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
3441         \kernel_tl_gset:Nx \g__sort_internal_tl
3442             { \_sort_tl_toks:w \l__sort_min_int ; }
3443     \group_end:
3444     #1 #2 \g__sort_internal_tl
3445     \tl_gclear:N \g__sort_internal_tl
3446     \prg_break_point:
3447 }
3448 \cs_new:Npn \_sort_tl_toks:w #1 ;
3449 {
3450     \if_int_compare:w #1 < \l__sort_top_int
3451         { \tex_the:D \tex_toks:D #1 }
3452         \exp_after:wN \_sort_tl_toks:w
3453         \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
3454     \fi:
3455 }
```

(End definition for `\tl_sort:Nn` and others. These functions are documented on page 117.)

`\seq_sort:Nn`  
`\seq_sort:cn`  
`\seq_gsort:Nn`  
`\seq_gsort:cn`  
`\clist_sort:Nn`  
`\clist_sort:cn`  
`\clist_gsort:Nn`  
`\clist_gsort:cn`  
`\_sort_seq:NNNN`

Use the same general framework for seq and clist. Apply the general sorting code, then unpack `\toks` into `\g__sort_internal_seq`. Outside the group copy or convert (for clist) the data to the target variable. The `\seq_gclear:N` reduces memory usage. The `\prg_break_point:` is used by `\_sort_main:NNNn` when the list is too long.

```

3456 \cs_new_protected:Npn \seq_sort:Nn
3457     { \_sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN }
3458 \cs_generate_variant:Nn \seq_sort:Nn { c }
3459 \cs_new_protected:Npn \seq_gsort:Nn
3460     { \_sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }
3461 \cs_generate_variant:Nn \seq_gsort:Nn { c }
3462 \cs_new_protected:Npn \clist_sort:Nn
3463 {
3464     \_sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
3465         \clist_set_from_seq:NN
3466 }
3467 \cs_generate_variant:Nn \clist_sort:Nn { c }
3468 \cs_new_protected:Npn \clist_gsort:Nn
3469 {
3470     \_sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n
3471         \clist_gset_from_seq:NN
3472 }
3473 \cs_generate_variant:Nn \clist_gsort:Nn { c }
3474 \cs_new_protected:Npn \_sort_seq:NNNNn #1#2#3#4#5
3475 {
3476     \group_begin:
3477         \_sort_main:NNNn #1 #2 #4 {#5}
```

```

3478   \seq_gset_from_inline_x:Nnn \g__sort_internal_seq
3479   {
3480     \int_step_function:nnN
3481       { \l__sort_min_int } { \l__sort_top_int - 1 }
3482     }
3483     { \tex_the:D \tex_toks:D ##1 }
3484   \group_end:
3485   #3 #4 \g__sort_internal_seq
3486   \seq_gclear:N \g__sort_internal_seq
3487   \prg_break_point:
3488 }

```

(End definition for `\seq_sort:Nn` and others. These functions are documented on page 148.)

## 43.4 Merge sort

- `\__sort_level:` This function is called once blocks of size `\l__sort_block_int` (initially 1) are each sorted. If the whole list fits in one block, then we are done (this also takes care of the case of an empty list or a list with one item). Otherwise, go through pairs of blocks starting from 0, then double the block size, and repeat.

```

3489 \cs_new_protected:Npn \__sort_level:
3490 {
3491   \if_int_compare:w \l__sort_block_int < \l__sort_length_int
3492     \l__sort_end_int \l__sort_min_int
3493     \__sort_merge_blocks:
3494     \tex_advance:D \l__sort_block_int \l__sort_block_int
3495     \exp_after:wN \__sort_level:
3496   \fi:
3497 }

```

(End definition for `\__sort_level:..`)

- `\__sort_merge_blocks:` This function is called to merge a pair of blocks, starting at the last value of `\l__sort_end_int` (end-point of the previous pair of blocks). If shifting by one block to the right we reach the end of the list, then this pass has ended: the end of the list is sorted already. Otherwise, store the result of that shift in `A`, which indexes the first block starting from the top end. Then locate the end-point (maximum) of the second block: shift `end` upwards by one more block, but keeping it  $\leq \text{top}$ . Copy this upper block of `\toks` registers in registers above `length`, indexed by `C`: this is covered by `\__sort_copy_block:..`. Once this is done we are ready to do the actual merger using `\__sort_merge_blocks_aux:..`, after shifting `A`, `B` and `C` so that they point to the largest index in their respective ranges rather than pointing just beyond those ranges. Of course, once that pair of blocks is merged, move on to the next pair.

```

3498 \cs_new_protected:Npn \__sort_merge_blocks:
3499 {
3500   \l__sort_begin_int \l__sort_end_int
3501   \tex_advance:D \l__sort_end_int \l__sort_block_int
3502   \if_int_compare:w \l__sort_end_int < \l__sort_top_int
3503     \l__sort_A_int \l__sort_end_int
3504     \tex_advance:D \l__sort_end_int \l__sort_block_int
3505     \if_int_compare:w \l__sort_end_int > \l__sort_top_int
3506       \l__sort_end_int \l__sort_top_int

```

```

3507     \fi:
3508     \l__sort_B_int \l__sort_A_int
3509     \l__sort_C_int \l__sort_top_int
3510     \__sort_copy_block:
3511     \int_decr:N \l__sort_A_int
3512     \int_decr:N \l__sort_B_int
3513     \int_decr:N \l__sort_C_int
3514     \exp_after:wN \__sort_merge_blocks_aux:
3515     \exp_after:wN \__sort_merge_blocks:
3516   \fi:
3517 }

```

(End definition for `\__sort_merge_blocks::`)

`\__sort_copy_block:` We wish to store a copy of the “upper” block of `\toks` registers, ranging between the initial value of `\l__sort_B_int` (included) and `\l__sort_end_int` (excluded) into a new range starting at the initial value of `\l__sort_C_int`, namely `\l__sort_top_int`.

```

3518 \cs_new_protected:Npn \__sort_copy_block:
3519   {
3520     \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
3521     \int_incr:N \l__sort_C_int
3522     \int_incr:N \l__sort_B_int
3523     \if_int_compare:w \l__sort_B_int = \l__sort_end_int
3524       \use_i:nn
3525     \fi:
3526     \__sort_copy_block:
3527   }

```

(End definition for `\__sort_copy_block::`)

`\__sort_merge_blocks_aux:` At this stage, the first block starts at `\l__sort_begin_int`, and ends at `\l__sort_A_int`, and the second block starts at `\l__sort_top_int` and ends at `\l__sort_C_int`. The result of the merger is stored at positions indexed by `\l__sort_B_int`, which starts at `\l__sort_end_int - 1` and decreases down to `\l__sort_begin_int`, covering the full range of the two blocks. In other words, we are building the merger starting with the largest values. The comparison function is defined to return either `swapped` or `same`. Of course, this means the arguments need to be given in the order they appear originally in the list.

```

3528 \cs_new_protected:Npn \__sort_merge_blocks_aux:
3529   {
3530     \exp_after:wN \__sort_compare:nn \exp_after:wN
3531     { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
3532     \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
3533     \prg_do_nothing:
3534     \__sort_return_mark:w
3535     \__sort_return_mark:w
3536     \s__sort_mark
3537     \__sort_return_none_error:
3538   }

```

(End definition for `\__sort_merge_blocks_aux::`)

```

\sort_return_same:
\sort_return_swapped:
\__sort_return_mark:w
\__sort_return_none_error:
\__sort_return_two_error:

Each comparison should call \sort_return_same: or \sort_return_swapped: exactly once. If neither is called, \__sort_return_none_error: is called, since the return_mark removes tokens until \s__sort_mark. If one is called, the return_mark auxiliary removes everything except \__sort_return_same:w (or its swapped analogue) followed by \__sort_return_none_error:. Finally if two or more are called, \__sort_return_two_error: ends up before any \__sort_return_mark:w, so that it produces an error.

3539 \cs_new_protected:Npn \sort_return_same:
3540     #1 \__sort_return_mark:w #2 \s__sort_mark
3541 {
3542     #1
3543     #2
3544     \__sort_return_two_error:
3545     \__sort_return_mark:w
3546     \s__sort_mark
3547     \__sort_return_same:w
3548 }
3549 \cs_new_protected:Npn \sort_return_swapped:
3550     #1 \__sort_return_mark:w #2 \s__sort_mark
3551 {
3552     #1
3553     #2
3554     \__sort_return_two_error:
3555     \__sort_return_mark:w
3556     \s__sort_mark
3557     \__sort_return_swapped:w
3558 }
3559 \cs_new_protected:Npn \__sort_return_mark:w #1 \s__sort_mark { }
3560 \cs_new_protected:Npn \__sort_return_none_error:
3561 {
3562     \msg_error:nnnx { sort } { return-none }
3563     { \tex_the:D \tex_toks:D \l__sort_A_int }
3564     { \tex_the:D \tex_toks:D \l__sort_C_int }
3565     \__sort_return_same:w \__sort_return_none_error:
3566 }
3567 \cs_new_protected:Npn \__sort_return_two_error:
3568 {
3569     \msg_error:nnnx { sort } { return-two }
3570     { \tex_the:D \tex_toks:D \l__sort_A_int }
3571     { \tex_the:D \tex_toks:D \l__sort_C_int }
3572 }

```

(End definition for \sort\_return\_same: and others. These functions are documented on page 44.)

\\_\_sort\_return\_same:w If the comparison function returns same, then the second argument fed to \\_\_sort\_compare:nn should remain to the right of the other one. Since we build the merger starting from the right, we copy that \toks register into the allotted range, then shift the pointers *B* and *C*, and go on to do one more step in the merger, unless the second block has been exhausted: then the remainder of the first block is already in the correct registers and we are done with merging those two blocks.

```

3573 \cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
3574 {
3575     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3576     \int_decr:N \l__sort_B_int

```

```

3577     \int_decr:N \l__sort_C_int
3578     \if_int_compare:w \l__sort_C_int < \l__sort_top_int
3579         \use_i:nn
3580     \fi:
3581     \__sort_merge_blocks_aux:
3582 }

```

(End definition for `\__sort_return_same:w`)

`\__sort_return_swapped:w`

If the comparison function returns `swapped`, then the next item to add to the merger is the first argument, contents of the `\toks` register  $A$ . Then shift the pointers  $A$  and  $B$  to the left, and go for one more step for the merger, unless the left block was exhausted ( $A$  goes below the threshold). In that case, all remaining `\toks` registers in the second block, indexed by  $C$ , are copied to the merger by `\__sort_merge_blocks_end`:

```

3583 \cs_new_protected:Npn \__sort_return_swapped:w #1 \__sort_return_none_error:
3584 {
3585     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_A_int
3586     \int_decr:N \l__sort_B_int
3587     \int_decr:N \l__sort_A_int
3588     \if_int_compare:w \l__sort_A_int < \l__sort_begin_int
3589         \__sort_merge_blocks_end: \use_i:nn
3590     \fi:
3591     \__sort_merge_blocks_aux:
3592 }

```

(End definition for `\__sort_return_swapped:w`)

`\__sort_merge_blocks_end:`

This function's task is to copy the `\toks` registers in the block indexed by  $C$  to the merger indexed by  $B$ . The end can equally be detected by checking when  $B$  reaches the threshold `begin`, or when  $C$  reaches `top`.

```

3593 \cs_new_protected:Npn \__sort_merge_blocks_end:
3594 {
3595     \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
3596     \int_decr:N \l__sort_B_int
3597     \int_decr:N \l__sort_C_int
3598     \if_int_compare:w \l__sort_B_int < \l__sort_begin_int
3599         \use_i:nn
3600     \fi:
3601     \__sort_merge_blocks_end:
3602 }

```

(End definition for `\__sort_merge_blocks_end`)

## 43.5 Expandable sorting

Sorting expandably is very different from sorting and assigning to a variable. Since tokens cannot be stored, they must remain in the input stream, and be read through at every step. It is thus necessarily much slower (at best  $O(n^2 \ln n)$ ) than non-expandable sorting functions ( $O(n \ln n)$ ).

A prototypical version of expandable quicksort is as follows. If the argument has no item, return nothing, otherwise partition, using the first item as a pivot (argument #4 of `\__sort:nnNnn`). The arguments of `\__sort:nnNnn` are 1. items less than #4, 2. items greater or equal to #4, 3. comparison, 4. pivot, 5. next item to test. If #5 is the tail of

the list, call `\tl_sort:nN` on #1 and on #2, placing #4 in between; `\use:ff` expands the parts to make `\tl_sort:nN` f-expandable. Otherwise, compare #4 and #5 using #3. If they are ordered, place #5 amongst the “greater” items, otherwise amongst the “lesser” items, and continue partitioning.

```
\cs_new:Npn \tl_sort:nN #1#2
{
    \tl_if_blank:nF {#1}
    {
        \__sort:nnNnn { } { } #2
        #1 \q__sort_recursion_tail \q__sort_recursion_stop
    }
}
\cs_new:Npn \__sort:nnNnn #1#2#3#4#5
{
    \quark_if_recursion_tail_stop_do:nn {#5}
    { \use:ff { \tl_sort:nN {#1} #3 {#4} } { \tl_sort:nN {#2} #3 } }
    #3 {#4} {#5}
    { \__sort:nnNnn {#1} { #2 {#5} } #3 {#4} }
    { \__sort:nnNnn { #1 {#5} } {#2} #3 {#4} }
}
\cs_generate_variant:Nn \use:nn { ff }
```

There are quite a few optimizations available here: the code below is less legible, but more than twice as fast.

In the simple version of the code, `\__sort:nnNnn` is called  $O(n \ln n)$  times on average (the number of comparisons required by the quicksort algorithm). Hence most of our focus is on optimizing that function.

The first speed up is to avoid testing for the end of the list at every call to `\__sort:nnNnn`. For this, the list is prepared by changing each `\langle item \rangle` of the original token list into `\langle command \rangle \{ \langle item \rangle \}`, just like sequences are stored. We arrange things such that the `\langle command \rangle` is the `\langle conditional \rangle` provided by the user: the loop over the `\langle prepared tokens \rangle` then looks like

```
\cs_new:Npn \__sort_loop:wNn ... #6#7
{
    #6 {\langle pivot \rangle} {#7} \langle loop big \rangle \langle loop small \rangle
    \langle extra arguments \rangle
}
\__sort_loop:wNn ... \langle prepared tokens \rangle
\end-loop \} \s__sort_stop
```

In this example, which matches the structure of `\__sort_quick_split_i:NnnnnNn` and a few other functions below, the `\__sort_loop:wNn` auxiliary normally receives the user’s `\langle conditional \rangle` as #6 and an `\langle item \rangle` as #7. This is compared to the `\langle pivot \rangle` (the argument #5, not shown here), and the `\langle conditional \rangle` leaves the `\langle loop big \rangle` or `\langle loop small \rangle` auxiliary, which both have the same form as `\__sort_loop:wNn`, receiving the next pair `\langle conditional \rangle \{ \langle item \rangle \}` as #6 and #7. At the end, #6 is the `\langle end-loop \rangle` function, which terminates the loop.

The second speed up is to minimize the duplicated tokens between the `true` and `false` branches of the conditional. For this, we introduce two versions of `\__sort:nnNnn`,

which receive the new item as #1 and place it either into the list #2 of items less than the pivot #4 or into the list #3 of items greater or equal to the pivot.

```
\cs_new:Npn \__sort_i:n:nnnnNn #1#2#3#4#5#6
{
    #5 {#4} {#6} \__sort_ii:n:nnnnNn \__sort_i:n:nnnnNn
        {#6} {#2 {#1}} {#3} {#4}
}
\cs_new:Npn \__sort_ii:n:nnnnNn #1#2#3#4#5#6
{
    #5 {#4} {#6} \__sort_ii:n:nnnnNn \__sort_i:n:nnnnNn
        {#6} {#2} {#3 {#1}} {#4}
}
```

Note that the two functions have the form of `\__sort_loop:wNn` above, receiving as #5 the conditional or a function to end the loop. In fact, the lists #2 and #3 must be made of pairs  $\langle\text{conditional}\rangle\{\langle\text{item}\rangle\}$ , so we have to replace {#6} above by {#5 {#6}}, and {#1} by #1. The actual functions have one more argument, so all argument numbers are shifted compared to this code.

The third speed up is to avoid `\use:ff` using a continuation-passing style: `\__sort_quick_split:NnNn` expects a list followed by `\s__sort_mark {\langle\text{code}\rangle}`, and expands to  $\langle\text{code}\rangle\langle\text{sorted list}\rangle$ . Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \s__sort_mark
{
\__sort_quick_split:NnNn #1 ... \s__sort_mark {\langle\text{code}\rangle}
{\langle\text{pivot}\rangle}
}
```

Items which are larger than the  $\langle\text{pivot}\rangle$  are sorted, then placed after code that sorts the smaller items, and after the (braced)  $\langle\text{pivot}\rangle$ .

The fourth speed up is avoid the recursive call to `\tl_sort:nN` with an empty first argument. For this, we introduce functions similar to the `\__sort_i:n:nnnnNn` of the last example, but aware of whether the list of  $\langle\text{conditional}\rangle\{\langle\text{item}\rangle\}$  read so far that are less than the pivot, and the list of those greater or equal, are empty or not: see `\__sort_quick_split:NnNn` and functions defined below. Knowing whether the lists are empty or not is useless if we do not use distinct ending codes as appropriate. The splitting auxiliaries communicate to the  $\langle\text{end-loop}\rangle$  function (that is initially placed after the “prepared” list) by placing a specific ending function, ignored when looping, but useful at the end. In fact, the  $\langle\text{end-loop}\rangle$  function does nothing but place the appropriate ending function in front of all its arguments. The ending functions take care of sorting non-empty sublists, placing the pivot in between, and the continuation before.

The final change in fact slows down the code a little, but is required to avoid memory issues: schematically, when TeX encounters

```
\use:n { \use:n { \use:n { ... } ... } ... }
```

the argument of the first `\use:n` is not completely read by the second `\use:n`, hence must remain in memory; then the argument of the second `\use:n` is not completely read when grabbing the argument of the third `\use:n`, hence must remain in memory, and so on. The memory consumption grows quadratically with the number of nested `\use:n`. In

practice, this means that we must read everything until a trailing `\s__sort_stop` once in a while, otherwise sorting lists of more than a few thousand items would exhaust a typical TeX's memory.

#### `\tl_sort:nN`

```
\__sort_quick_prepare:Nnnn
  \__sort_quick_prepare_end:NNNnw
\__sort_quick_cleanup:w
```

The code within the `\exp_not:f` sorts the list, leaving in most cases a leading `\exp_not:f`, which stops the expansion, letting the result be return within `\exp_not:n`. We filter out the case of a list with no item, which would otherwise cause problems. Then prepare the token list #1 by inserting the conditional #2 before each item. The `prepare` auxiliary receives the conditional as #1, the prepared token list so far as #2, the next prepared item as #3, and the item after that as #4. The loop ends when #4 contains `\prg_break_point:`, then the `prepare_end` auxiliary finds the prepared token list as #4. The scene is then set up for `\__sort_quick_split:NnNn`, which sorts the prepared list and perform the post action placed after `\s__sort_mark`, namely removing the trailing `\s__sort_stop` and `\s__sort_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```
3603 \cs_new:Npn \tl_sort:nN #1#2
3604 {
3605   \exp_not:f
3606   {
3607     \tl_if_blank:nF {#1}
3608     {
3609       \__sort_quick_prepare:Nnnn #2 { } { }
3610       #1
3611       { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
3612       \s__sort_stop
3613     }
3614   }
3615 }
3616 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
3617 {
3618   \prg_break: #4 \prg_break_point:
3619   \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
3620 }
3621 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \s__sort_stop
3622 {
3623   \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
3624   \s__sort_mark { \__sort_quick_cleanup:w \exp_stop_f: }
3625   \s__sort_mark \s__sort_stop
3626 }
3627 \cs_new:Npn \__sort_quick_cleanup:w #1 \s__sort_mark \s__sort_stop {#1}
```

(End definition for `\tl_sort:nN` and others. This function is documented on page 117.)

```
\__sort_quick_split:NnNn
\__sort_quick_only_i:NnnnnNn
  \__sort_quick_only_ii:NnnnnNn
  \__sort_quick_split_i:NnnnnNn
\__sort_quick_split_ii:NnnnnNn
```

The `only_i`, `only_ii`, `split_i` and `split_ii` auxiliaries receive a useless first argument, the new item #2 (that they append to either one of the next two arguments), the list #3 of items less than the pivot, bigger items #4, the pivot #5, a `\langle function \rangle` #6, and an item #7. The `\langle function \rangle` is the user's `\langle conditional \rangle` except at the end of the list where it is `\__sort_quick_end:nnTFNn`. The comparison is applied to the `\langle pivot \rangle` and the `\langle item \rangle`, and calls the `only_i` or `split_i` auxiliaries if the `\langle item \rangle` is smaller, and the `only_ii` or `split_ii` auxiliaries otherwise. In both cases, the next auxiliary goes to work right away, with no intermediate expansion that would slow down operations. Note that the argument #2 left for the next call has the form `\langle conditional \rangle \{ \langle item \rangle \}`, so that the lists #3 and #4 keep the right form to be fed to the next sorting function. The `split` auxiliary

differs from these in that it is missing three of the arguments, which would be empty, and its first argument is always the user's *(conditional)* rather than an ending function.

```

3628 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
3629 {
3630     #3 {#2} {#4} \__sort_quick_only_i:NnnnnNn
3631         \__sort_quick_only_i:NnnnnNn
3632         \__sort_quick_single_end:nnnw
3633         { #3 {#4} } { } { } {#2}
3634     }
3635 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
3636 {
3637     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3638         \__sort_quick_only_i:NnnnnNn
3639         \__sort_quick_only_i_end:nnnw
3640         { #6 {#7} } { #3 #2 } { } {#5}
3641     }
3642 \cs_new:Npn \__sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
3643 {
3644     #6 {#5} {#7} \__sort_quick_only_ii:NnnnnNn
3645         \__sort_quick_split_i:NnnnnNn
3646         \__sort_quick_only_ii_end:nnnw
3647         { #6 {#7} } { } { #4 #2 } {#5}
3648     }
3649 \cs_new:Npn \__sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
3650 {
3651     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3652         \__sort_quick_split_i:NnnnnNn
3653         \__sort_quick_split_end:nnnw
3654         { #6 {#7} } { #3 #2 } {#4} {#5}
3655     }
3656 \cs_new:Npn \__sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
3657 {
3658     #6 {#5} {#7} \__sort_quick_split_ii:NnnnnNn
3659         \__sort_quick_split_i:NnnnnNn
3660         \__sort_quick_split_end:nnnw
3661         { #6 {#7} } {#3} { #4 #2 } {#5}
3662     }

```

*(End definition for \\_\_sort\_quick\_split:NnNn and others.)*

```
\__sort_quick_end:nnTFNn
\__sort_quick_single_end:nnnw
\__sort_quick_only_i_end:nnnw
\__sort_quick_only_ii_end:nnnw
\__sort_quick_split_end:nnnw
```

The `\__sort_quick_end:nnTFNn` appears instead of the user's conditional, and receives as its arguments the pivot #1, a fake item #2, a `true` and a `false` branches #3 and #4, followed by an ending function #5 (one of the four auxiliaries here) and another copy #6 of the fake item. All those are discarded except the function #5. This function receives lists #1 and #2 of items less than or greater than the pivot #3, then a continuation code #5 just after `\s__sort_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\s__sort_stop` and place #6 back into the input stream. When the lists #1 and #2 are empty, the `single` auxiliary simply places the continuation #5 before the pivot {#3}. When #2 is empty, #1 is sorted and placed before the pivot {#3}, taking care to feed the continuation #5 as a continuation for the function sorting #1. When #1 is empty, #2 is sorted, and the continuation argument is used to place the continuation #5 and the pivot {#3} before the sorted result. Finally, when both

lists are non-empty, items larger than the pivot are sorted, then items less than the pivot, and the continuations are done in such a way to place the pivot in between.

```

3663 \cs_new:Npn \__sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
3664 \cs_new:Npn \__sort_quick_single_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3665 { #5 {#3} #6 \s__sort_stop }
3666 \cs_new:Npn \__sort_quick_only_i_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3667 {
3668     \__sort_quick_split:NnNn #1
3669         \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3670     {#3}
3671     #6 \s__sort_stop
3672 }
3673 \cs_new:Npn \__sort_quick_only_ii_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3674 {
3675     \__sort_quick_split:NnNn #2
3676         \__sort_quick_end:nnTFNn { } \s__sort_mark { #5 {#3} }
3677     #6 \s__sort_stop
3678 }
3679 \cs_new:Npn \__sort_quick_split_end:nnwnw #1#2#3#4 \s__sort_mark #5#6 \s__sort_stop
3680 {
3681     \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \s__sort_mark
3682     {
3683         \__sort_quick_split:NnNn #1
3684             \__sort_quick_end:nnTFNn { } \s__sort_mark {#5}
3685             {#3}
3686     }
3687     #6 \s__sort_stop
3688 }
```

(End definition for `\__sort_quick_end:nnTFNn` and others.)

## 43.6 Messages

`\__sort_error:` Bailing out of the sorting code is a bit tricky. It may not be safe to use a delimited argument, so instead we redefine many `\l3sort` commands to be trivial, with `\__sort_level:` jumping to the break point. This error recovery won't work in a group.

```

3689 \cs_new_protected:Npn \__sort_error:
3690 {
3691     \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
3692     \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
3693     \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
3694 }
```

(End definition for `\__sort_error::`)

`\__sort_disable_toksdef:` While sorting, `\toksdef` is locally disabled to prevent users from using `\newtoks` or similar commands in their comparison code: the `\toks` registers that would be assigned are in use by `\l3sort`. In format mode, none of this is needed since there is no `\toks` allocator.

```

3695 \cs_new_protected:Npn \__sort_disable_toksdef:
3696     { \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n }
3697 \cs_new_protected:Npn \__sort_disabled_toksdef:n #1
3698     {
```

```

3699     \msg_error:nnx { sort } { toksdef }
3700         { \token_to_str:N #1 }
3701     \__sort_error:
3702     \tex_toksdef:D #1
3703     }
3704 \msg_new:nnnn { sort } { toksdef }
3705     { Allocation~of~\iow_char:N\\toks-registers-impossible-while-sorting. }
3706     {
3707         The~comparison~code~used~for~sorting~a~list~has~attempted~to~
3708         define~#1~as~a~new~\iow_char:N\\toks~register~using~
3709         \iow_char:N\\newtoks~
3710         or~a~similar~command.~The~list~will~not~be~sorted.
3711     }

```

(End definition for `\__sort_disable_toksdef:` and `\__sort_disabled_toksdef:n.`)

`\__sort_too_long_error>NNw` When there are too many items in a sequence, this is an error, and we clean up properly the mapping over items in the list: break using the type-specific breaking function #1.

```

3712 \cs_new_protected:Npn \__sort_too_long_error:NNw #1#2 \fi:
3713     {
3714         \fi:
3715         \msg_error:nnxxx { sort } { too-large }
3716             { \token_to_str:N #2 }
3717             { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
3718             { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
3719             #1 \__sort_error:
3720     }
3721 \msg_new:nnnn { sort } { too-large }
3722     { The-list~#1-is-too-long-to-be-sorted-by-TeX. }
3723     {
3724         TeX-has~#2~toks~registers~still~available:~
3725         this~only~allows~to~sort~with~up~to~#3~
3726         items.~The~list~will~not~be~sorted.
3727     }

```

(End definition for `\__sort_too_long_error:NNw.`)

```

3728 \msg_new:nnnn { sort } { return-none }
3729     { The~comparison~code~did~not~return. }
3730     {
3731         When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
3732         did~not~call~
3733         \iow_char:N\\sort_return_same:~nor~
3734         \iow_char:N\\sort_return_swapped:~.
3735         Exactly~one~of~these~should~be~called.
3736     }
3737 \msg_new:nnnn { sort } { return-two }
3738     { The~comparison~code~returned~multiple~times. }
3739     {
3740         When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
3741         \iow_char:N\\sort_return_same:~or~
3742         \iow_char:N\\sort_return_swapped:~multiple~times.~
3743         Exactly~one~of~these~should~be~called.
3744     }
3745 \prop_gput:Nnn \g_msg_module_name_prop { sort } { LaTeX }
3746 \prop_gput:Nnn \g_msg_module_type_prop { sort } { }

```

3747 </package>

# Chapter 44

## I3tl-analysis implementation

3748  $\langle @@=t1 \rangle$

### 44.1 Internal functions

- `\s__t1` The format used to store token lists internally uses the scan mark `\s__t1` as a delimiter.  
*(End definition for `\s__t1`.)*

### 44.2 Internal format

The task of the I3tl-analysis module is to convert token lists to an internal format which allows us to extract all the relevant information about individual tokens (category code, character code), as well as reconstruct the token list quickly. This internal format is used in I3regex where we need to support arbitrary tokens, and it is used in conversion functions in I3str-convert, where we wish to support clusters of characters instead of single tokens.

We thus need a way to encode any  $\langle token \rangle$  (even begin-group and end-group character tokens) in a way amenable to manipulating tokens individually. The best we can do is to find  $\langle tokens \rangle$  which both o-expand and x-expand to the given  $\langle token \rangle$ . Collecting more information about the category code and character code is also useful for regular expressions, since most regexes are catcode-agnostic. The internal format thus takes the form of a succession of items of the form

$\langle tokens \rangle \backslash s__t1 \langle catcode \rangle \langle char code \rangle \backslash s__t1$

The  $\langle tokens \rangle$  o- and x-expand to the original token in the token list or to the cluster of tokens corresponding to one Unicode character in the given encoding (for I3str-convert). The  $\langle catcode \rangle$  is given as a single hexadecimal digit, 0 for control sequences. The  $\langle char code \rangle$  is given as a decimal number, -1 for control sequences.

Using delimited arguments lets us build the  $\langle tokens \rangle$  progressively when doing an encoding conversion in I3str-convert. On the other hand, the delimiter `\s__t1` may not appear unbraced in  $\langle tokens \rangle$ . This is not a problem because we are careful to wrap control sequences in braces (as an argument to `\exp_not:n`) when converting from a general token list to the internal format.

The current rule for converting a  $\langle token \rangle$  to a balanced set of  $\langle tokens \rangle$  which both o-expands and x-expands to it is the following.

- A control sequence `\cs` becomes `\exp_not:n { \cs } \s__tl 0 -1 \s__tl`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s__tl 1 <char code> \s__tl`.
- An end-group character `}` becomes `\if_false: { \fi: } \s__tl 2 <char code> \s__tl`.
- A character with any other category code becomes `\exp_not:n {<character>} \s__tl <hex catcode> <char code> \s__tl`.

In contrast, for `\peek_analysis_map_inline:n` we must allow for an input stream containing `\outer` macros, so that wrapping all control sequences in `\exp_not:n` is unsafe. Instead, we write the more elaborate `\_kernel_exp_not:w \exp_after:wN { \exp_not:N \cs }`. (On the other hand we make a better effort by avoiding `\exp_not:n` for characters other than active and macro parameters.)

3749 `(*package)`

### 44.3 Variables and helper functions

<code>\s__tl</code>	The scan mark <code>\s__tl</code> is used as a delimiter in the internal format. This is more practical than using a quark, because we would then need to control expansion much more carefully: compare <code>\int_value:w '#1 \s__tl</code> with <code>\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark</code> to extract a character code followed by the delimiter in an x-expansion.
<small>3750</small>	<code>\scan_new:N \s__tl</code>
	<i>(End definition for <code>\s__tl</code>.)</i>
<code>\l__tl_analysis_token</code> <code>\l__tl_analysis_char_token</code> <code>\l__tl_analysis_next_token</code>	The tokens in the token list are probed with the TeX primitive <code>\futurelet</code> . We use <code>\l__tl_analysis_token</code> in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is <code>\l__tl_analysis_char_token</code> . When getting tokens from the input stream we may need to look two tokens ahead, for which we use <code>\l__tl_analysis_next_token</code> .
	<small>3751</small> <code>\cs_new_eq:NN \l__tl_analysis_token ?</code> <small>3752</small> <code>\cs_new_eq:NN \l__tl_analysis_char_token ?</code> <small>3753</small> <code>\cs_new_eq:NN \l__tl_analysis_next_token ?</code>
	<i>(End definition for <code>\l__tl_analysis_token</code>, <code>\l__tl_analysis_char_token</code>, and <code>\l__tl_analysis_next_token</code>.)</i>
<code>\l__tl_peek_code_t1</code>	Holds some code to be run once the next token has been fully analysed in <code>\peek_analysis_map_inline:n</code> .
	<small>3754</small> <code>\tl_new:N \l__tl_peek_code_t1</code>
	<i>(End definition for <code>\l__tl_peek_code_t1</code>.)</i>
<code>\c__tl_peek_catcodes_t1</code>	A token list containing the character number 32 (space) with all possible category codes except 1 and 2 (begin-group and end-group). Why 32? Because some LuaTeX versions only allow creation of catcode 10 (space) tokens with this character code, so that we decided to make <code>\char_generate:nn</code> refuse to create such weird spaces as well. We do not include the macro parameter case (catcode 6) because it cannot be used as a macro delimiter.

```

3755 \group_begin:
3756 \char_set_active_eq:NN \ \scan_stop:
3757 \tl_const:Nx \c__tl_peek_catcodes_tl
3758 {
3759     \char_generate:nn { 32 } { 3 } 3
3760     \char_generate:nn { 32 } { 4 } 4
3761     \char_generate:nn { 32 } { 7 } 7
3762     \char_generate:nn { 32 } { 8 } 8
3763     \c_space_tl \token_to_str:N A
3764     \char_generate:nn { 32 } { 11 } \token_to_str:N B
3765     \char_generate:nn { 32 } { 12 } \token_to_str:N C
3766     \char_generate:nn { 32 } { 13 } \token_to_str:N D
3767 }
3768 \group_end:

```

(End definition for `\c__tl_peek_catcodes_tl`.)

`\l__tl_analysis_normal_int` The number of normal (N-type argument) tokens since the last special token.

```
3769 \int_new:N \l__tl_analysis_normal_int
```

(End definition for `\l__tl_analysis_normal_int`.)

`\l__tl_analysis_index_int` During the first pass, this is the index in the array being built. During the second pass, it is equal to the maximum index in the array from the first pass.

```
3770 \int_new:N \l__tl_analysis_index_int
```

(End definition for `\l__tl_analysis_index_int`.)

`\l__tl_analysis_nesting_int` Nesting depth of explicit begin-group and end-group characters during the first pass. This lets us detect the end of the token list without a reserved end-marker.

```
3771 \int_new:N \l__tl_analysis_nesting_int
```

(End definition for `\l__tl_analysis_nesting_int`.)

`\l__tl_analysis_type_int` When encountering special characters, we record their “type” in this integer.

```
3772 \int_new:N \l__tl_analysis_type_int
```

(End definition for `\l__tl_analysis_type_int`.)

`\g__tl_analysis_result_tl` The result of the conversion is stored in this token list, with a succession of items of the form

```

<tokens> \s__tl <catcode> <char code> \s__tl
3773 \tl_new:N \g__tl_analysis_result_tl

```

(End definition for `\g__tl_analysis_result_tl`.)

`\_tl_analysis extract charcode:` Extracting the character code from the meaning of `\l__tl_analysis_token`. This has no error checking, and should only be assumed to work for begin-group and end-group character tokens. It produces a number in the form ‘`<char>`’.

```

3774 \cs_new:Npn \__tl_analysis_extract_charcode:
3775 {
3776     \exp_after:wN \__tl_analysis_extract_charcode_aux:w
3777     \token_to_meaning:N \l__tl_analysis_token
3778 }
3779 \cs_new:Npn \__tl_analysis_extract_charcode_aux:w #1 ~ #2 ~ { ' }

```

(End definition for `\_tl_analysis_extract_charcode:` and `\_tl_analysis_extract_charcode_aux:w.`)

`\_tl_analysis_cs_space_count:NN`  
`\_tl_analysis_cs_space_count:w`  
`\_tl_analysis_cs_space_count_end:w`

Counts the number of spaces in the string representation of its second argument, as well as the number of characters following the last space in that representation, and feeds the two numbers as semicolon-delimited arguments to the first argument. When this function is used, the escape character is printable and non-space.

```

3780 \cs_new:Npn \_tl_analysis_cs_space_count:NN #1 #2
3781 {
3782     \exp_after:wN #1
3783     \int_value:w \int_eval:w 0
3784     \exp_after:wN \_tl_analysis_cs_space_count:w
3785     \token_to_str:N #2
3786     \fi: \_tl_analysis_cs_space_count_end:w ; ~ !
3787 }
3788 \cs_new:Npn \_tl_analysis_cs_space_count:w #1 ~
3789 {
3790     \if_false: #1 #1 \fi:
3791     + 1
3792     \_tl_analysis_cs_space_count:w
3793 }
3794 \cs_new:Npn \_tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
3795 { \exp_after:wN ; \int_value:w \str_count_ignore_spaces:n {#1} ; }
```

(End definition for `\_tl_analysis_cs_space_count:NN`, `\_tl_analysis_cs_space_count:w`, and `\_tl_analysis_cs_space_count_end:w.`)

## 44.4 Plan of attack

Our goal is to produce a token list of the form roughly

```

<token 1> \s__tl <catcode 1> <char code 1> \s__tl
<token 2> \s__tl <catcode 2> <char code 2> \s__tl
... <token N> \s__tl <catcode N> <char code N> \s__tl
```

Most but not all tokens can be grabbed as an undelimited (N-type) argument by TeX. The plan is to have a two pass system. In the first pass, locate special tokens, and store them in various `\toks` registers. In the second pass, which is done within an x-expanding assignment, normal tokens are taken in as N-type arguments, and special tokens are retrieved from the `\toks` registers, and removed from the input stream by some means. The whole process takes linear time, because we avoid building the result one item at a time.

We make the escape character printable (backslash, but this later oscillates between slash and backslash): this allows us to distinguish characters from control sequences.

A token has two characteristics: its `\meaning`, and what it looks like for TeX when it is in scanning mode (*e.g.*, when capturing parameters for a macro). For our purposes, we distinguish the following meanings:

- begin-group token (category code 1), either space (character code 32), or non-space;
- end-group token (category code 2), either space (character code 32), or non-space;
- space token (category code 10, character code 32);

- anything else (then the token is always an N-type argument).

The token itself can “look like” one of the following

- a non-active character, in which case its meaning is automatically that associated to its character code and category code, we call it “true” character;
- an active character;
- a control sequence.

The only tokens which are not valid N-type arguments are true begin-group characters, true end-group characters, and true spaces. We detect those characters by scanning ahead with `\futurelet`, then distinguishing true characters from control sequences set equal to them using the `\string` representation.

The second pass is a simple exercise in expandable loops.

`\__tl_analysis:n` Everything is done within a group, and all definitions are local. We use `\group_align_safe_begin/end`: to avoid problems in case `\__tl_analysis:n` is used within an alignment and its argument contains alignment tab tokens.

```

3796 \cs_new_protected:Npn \__tl_analysis:n #1
3797   {
3798     \group_begin:
3799     \group_align_safe_begin:
3800       \__tl_analysis_a:n {#1}
3801       \__tl_analysis_b:n {#1}
3802     \group_align_safe_end:
3803   \group_end:
3804 }
```

(End definition for `\__tl_analysis:n`.)

## 44.5 Disabling active characters

`\__tl_analysis_disable:n` Active characters can cause problems later on in the processing, so we provide a way to disable them, by setting them to `undefined`. Since Unicode contains too many characters to loop over all of them, we instead do this whenever we encounter a character. For pTeX and upTeX we skip characters beyond [0, 255] because `\lccode` only allows those values.

```

3805 \group_begin:
3806   \char_set_catcode_active:N \^^@ 
3807   \cs_new_protected:Npn \__tl_analysis_disable:n #1
3808   {
3809     \tex_lccode:D 0 = #1 \exp_stop_f:
3810     \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
3811   }
3812   \bool_lazy_or:nnT
3813   { \sys_if_engine_ptex_p: }
3814   { \sys_if_engine_uptex_p: }
3815   {
3816     \cs_gset_protected:Npn \__tl_analysis_disable:n #1
3817     {
3818       \if_int_compare:w 256 > #1 \exp_stop_f:
3819         \tex_lccode:D 0 = #1 \exp_stop_f:
```

```

3820           \tex_lowercase:D { \tex_let:D ^^@ } \tex_undefined:D
3821       \fi:
3822   }
3823 }
3824 \group_end:

(End definition for \__tl_analysis_disable:n.)

```

## 44.6 First pass

The goal of this pass is to detect special (non-N-type) tokens, and count how many N-type tokens lie between special tokens. Also, we wish to store some representation of each special token in a `\toks` register.

We have 11 types of tokens:

1. a true non-space begin-group character;
2. a true space begin-group character;
3. a true non-space end-group character;
4. a true space end-group character;
5. a true space blank space character;
6. an active character;
7. any other true character;
8. a control sequence equal to a begin-group token (category code 1);
9. a control sequence equal to an end-group token (category code 2);
10. a control sequence equal to a space token (character code 32, category code 10);
11. any other control sequence.

Our first tool is `\futurelet`. This cannot distinguish case 8 from 1 or 2, nor case 9 from 3 or 4, nor case 10 from case 5. Those cases are later distinguished by applying the `\string` primitive to the following token, after possibly changing the escape character to ensure that a control sequence's string representation cannot be mistaken for the true character.

In cases 6, 7, and 11, the following token is a valid N-type argument, so we grab it and distinguish the case of a character from a control sequence: in the latter case, `\str_tail:n {<token>}` is non-empty, because the escape character is printable.

`\__tl_analysis_a:n` We read tokens one by one using `\futurelet`. While performing the loop, we keep track of the number of true begin-group characters minus the number of true end-group characters in `\l__tl_analysis_nesting_int`. This reaches  $-1$  when we read the closing brace.

```

3825 \cs_new_protected:Npn \__tl_analysis_a:n #1
3826 {
3827     \__tl_analysis_disable:n { 32 }
3828     \int_set:Nn \tex_escapechar:D { 92 }
3829     \int_zero:N \l__tl_analysis_normal_int

```

```

3830     \int_zero:N \l__tl_analysis_index_int
3831     \int_zero:N \l__tl_analysis_nesting_int
3832     \if_false: { \fi: \l__tl_analysis_a_loop:w #1 }
3833     \int_decr:N \l__tl_analysis_index_int
3834 }

```

(End definition for `\l__tl_analysis_a:n`.)

`\l__tl_analysis_a_loop:w` Read one character and check its type.

```

3835 \cs_new_protected:Npn \l__tl_analysis_a_loop:w
3836   { \tex_futurelet:D \l__tl_analysis_token \l__tl_analysis_a_type:w }

```

(End definition for `\l__tl_analysis_a_loop:w`.)

`\l__tl_analysis_a_type:w` At this point, `\l__tl_analysis_token` holds the meaning of the following token. We store in `\l__tl_analysis_type_int` information about the meaning of the token ahead:

- 0 space token;
- 1 begin-group token;
- -1 end-group token;
- 2 other.

The values 0, 1, -1 correspond to how much a true such character changes the nesting level (2 is used only here, and is irrelevant later). Then call the auxiliary for each case. Note that nesting conditionals here is safe because we only skip over `\l__tl_analysis_token` if it matches with one of the character tokens (hence is not a primitive conditional).

```

3837 \cs_new_protected:Npn \l__tl_analysis_a_type:w
3838   {
3839     \l__tl_analysis_type_int =
3840       \if_meaning:w \l__tl_analysis_token \c_space_token
3841         0
3842       \else:
3843         \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
3844           1
3845         \else:
3846           \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
3847             - 1
3848           \else:
3849             2
3850           \fi:
3851         \fi:
3852       \exp_stop_f:
3853       \if_case:w \l__tl_analysis_type_int
3854         \exp_after:wN \l__tl_analysis_a_space:w
3855       \or: \exp_after:wN \l__tl_analysis_a_bgroup:w
3856       \or: \exp_after:wN \l__tl_analysis_a_safe:N
3857       \else: \exp_after:wN \l__tl_analysis_a_egroup:w
3858       \fi:
3859   }

```

(End definition for `\l__tl_analysis_a_type:w`.)

```
\__tl_analysis_a_space:w
\__tl_analysis_a_space_test:w
```

In this branch, the following token's meaning is a blank space. Apply `\string` to that token: a true blank space gives a space, a control sequence gives a result starting with the escape character, an active character gives something else than a space since we disabled the space. We grab as `\l__tl_analysis_char_token` the first character of the string representation then test it in `\__tl_analysis_a_space_test:w`. Also, since `\__tl_analysis_a_store:` expects the special token to be stored in the relevant `\toks` register, we do that. The extra `\exp_not:n` is unnecessary of course, but it makes the treatment of all tokens more homogeneous. If we discover that the next token was actually a control sequence or an active character instead of a true space, then we step the counter of normal tokens. We now have in front of us the whole string representation of the control sequence, including potential spaces; those will appear to be true spaces later in this pass. Hence, all other branches of the code in this first pass need to consider the string representation, so that the second pass does not need to test the meaning of tokens, only strings.

```
3861 \cs_new_protected:Npn \__tl_analysis_a_space:w
3862 {
3863     \tex_afterassignment:D \__tl_analysis_a_space_test:w
3864     \exp_after:wN \cs_set_eq:NN
3865     \exp_after:wN \l__tl_analysis_char_token
3866     \token_to_str:N
3867 }
3868 \cs_new_protected:Npn \__tl_analysis_a_space_test:w
3869 {
3870     \if_meaning:w \l__tl_analysis_char_token \c_space_token
3871     \tex_toks:D \l__tl_analysis_index_int { \exp_not:n { ~ } }
3872     \__tl_analysis_a_store:
3873 \else:
3874     \int_incr:N \l__tl_analysis_normal_int
3875 \fi:
3876 \__tl_analysis_a_loop:w
3877 }
```

(End definition for `\__tl_analysis_a_space:w` and `\__tl_analysis_a_space_test:w`.)

```
\__tl_analysis_a_bgroup:w
\__tl_analysis_a_egroup:w
\__tl_analysis_a_group:nw
\__tl_analysis_a_group_aux:w
\__tl_analysis_a_group_auxi:w
\__tl_analysis_a_group_test:w
```

The token is most likely a true character token with catcode 1 or 2, but it might be a control sequence, or an active character. Optimizing for the first case, we store in a `\toks` register some code that expands to that token. Since we will turn what follows into a string, we make sure the escape character is different from the current character code (by switching between solidus and backslash). To detect the special case of an active character let to the catcode 1 or 2 character with the same character code, we disable the active character with that character code and re-test: if the following token has become undefined we can in fact safely grab it. We are finally ready to turn what follows to a string and test it. This is one place where we need `\l__tl_analysis_char_token` to be a separate control sequence from `\l__tl_analysis_token`, to compare them.

```
3878 \group_begin:
3879     \char_set_catcode_group_begin:N \^@ % {
3880     \cs_new_protected:Npn \__tl_analysis_a_bgroup:w
3881     { \__tl_analysis_a_group:nw { \exp_after:wN \^@ \if_false: } \fi: } }
3882     \char_set_catcode_group_end:N \^@
3883     \cs_new_protected:Npn \__tl_analysis_a_egroup:w
3884     { \__tl_analysis_a_group:nw { \if_false: { \fi: \^@ } } % }
3885 \group_end:
```

```

3886 \cs_new_protected:Npn \__tl_analysis_a_group:nw #1
3887 {
3888     \tex_lccode:D 0 = \__tl_analysis_extract_charcode: \scan_stop:
3889     \tex_lowercase:D { \tex_toks:D \l__tl_analysis_index_int {#1} }
3890     \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
3891         \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }
3892     \fi:
3893     \__tl_analysis_disable:n { \tex_lccode:D 0 }
3894     \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
3895 }
3896 \cs_new_protected:Npn \__tl_analysis_a_group_aux:w
3897 {
3898     \if_meaning:w \l__tl_analysis_token \tex_undefined:D
3899         \exp_after:wN \__tl_analysis_a_safe:N
3900     \else:
3901         \exp_after:wN \__tl_analysis_a_group_auxii:w
3902     \fi:
3903 }
3904 \cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
3905 {
3906     \tex_afterassignment:D \__tl_analysis_a_group_test:w
3907     \exp_after:wN \cs_set_eq:NN
3908     \exp_after:wN \l__tl_analysis_char_token
3909     \token_to_str:N
3910 }
3911 \cs_new_protected:Npn \__tl_analysis_a_group_test:w
3912 {
3913     \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
3914         \__tl_analysis_a_store:
3915     \else:
3916         \int_incr:N \l__tl_analysis_normal_int
3917     \fi:
3918     \__tl_analysis_a_loop:w
3919 }

```

(End definition for `\__tl_analysis_a_bgroup:w` and others.)

`\__tl_analysis_a_store:` This function is called each time we meet a special token; at this point, the `\toks` register `\l__tl_analysis_index_int` holds a token list which expands to the given special token. Also, the value of `\l__tl_analysis_type_int` indicates which case we are in:

- -1 end-group character;
- 0 space character;
- 1 begin-group character.

We need to distinguish further the case of a space character (code 32) from other character codes, because those behave differently in the second pass. Namely, after testing the `\lccode` of 0 (which holds the present character code) we change the cases above to

- -2 space end-group character;
- -1 non-space end-group character;
- 0 space blank space character;

- 1 non-space begin-group character;
- 2 space begin-group character.

This has the property that non-space characters correspond to odd values of `\l__tl_analysis_type_int`. The number of normal tokens until here and the type of special token are packed into a `\skip` register. Finally, we check whether we reached the last closing brace, in which case we stop by disabling the looping function (locally).

```

3920 \cs_new_protected:Npn \__tl_analysis_a_store:
3921 {
3922     \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
3923     \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
3924         \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
3925     \fi:
3926     \tex_skip:D \l__tl_analysis_index_int
3927         = \l__tl_analysis_normal_int sp
3928             plus \l__tl_analysis_type_int sp \scan_stop:
3929     \int_incr:N \l__tl_analysis_index_int
3930     \int_zero:N \l__tl_analysis_normal_int
3931     \if_int_compare:w \l__tl_analysis_nesting_int = - \c_one_int
3932         \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
3933     \fi:
3934 }

```

(End definition for `\__tl_analysis_a_store`.)

`\__tl_analysis_a_safe:N` This should be the simplest case: since the upcoming token is safe, we can simply grab it in a second pass. If the token is a single character (including space), the `\if_charcode:w` test yields true; we disable a potentially active character (that could otherwise masquerade as the true character in the next pass) and we count one “normal” token. On the other hand, if the token is a control sequence, we should replace it by its string representation for compatibility with other code branches. Instead of slowly looping through the characters with the main code, we use the knowledge of how the second pass works: if the control sequence name contains no space, count that token as a number of normal tokens equal to its string length. If the control sequence contains spaces, they should be registered as special characters by increasing `\l__tl_analysis_index_int` (no need to carefully count character between each space), and all characters after the last space should be counted in the following sequence of “normal” tokens.

```

3935 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
3936 {
3937     \if_charcode:w
3938         \scan_stop:
3939         \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
3940         \scan_stop:
3941         \exp_after:wN \use_i:nn
3942     \else:
3943         \exp_after:wN \use_ii:nn
3944     \fi:
3945     {
3946         \__tl_analysis_disable:n { '#1' }
3947         \int_incr:N \l__tl_analysis_normal_int
3948     }
3949 \__tl_analysis_cs_space_count>NN \__tl_analysis_a_cs:ww #1 }

```

```

3950     \__tl_analysis_a_loop:w
3951   }
3952 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
3953 {
3954   \if_int_compare:w #1 > \c_zero_int
3955     \tex_skip:D \l__tl_analysis_index_int
3956     = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
3957     \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
3958   \else:
3959     \tex_advance:D
3960   \fi:
3961   \l__tl_analysis_normal_int #2 \exp_stop_f:
3962 }

```

(End definition for `\__tl_analysis_a_safe:N` and `\__tl_analysis_a_cs:ww`.)

## 44.7 Second pass

The second pass is an exercise in expandable loops. All the necessary information is stored in `\skip` and `\toks` registers.

`\__tl_analysis_b:n` Start the loop with the index 0. No need for an end-marker: the loop stops by itself when the last index is read. We repeatedly oscillate between reading long stretches of normal tokens, and reading special tokens.

```

3963 \cs_new_protected:Npn \__tl_analysis_b:n #1
3964 {
3965   \__kernel_tl_gset:Nx \g__tl_analysis_result_tl
3966   {
3967     \__tl_analysis_b_loop:w 0; #1
3968     \prg_break_point:
3969   }
3970 }
3971 \cs_new:Npn \__tl_analysis_b_loop:w #1;
3972 {
3973   \exp_after:wN \__tl_analysis_b_normals:ww
3974   \int_value:w \tex_skip:D #1 ; #1 ;
3975 }

```

(End definition for `\__tl_analysis_b:n` and `\__tl_analysis_b_loop:w`.)

`\__tl_analysis_b_normals:ww` The first argument is the number of normal tokens which remain to be read, and the second argument is the index in the array produced in the first step. A character's string representation is always one character long, while a control sequence is always longer (we have set the escape character to a printable value). In both cases, we leave `\exp_not:n` `{(token)}` `\s__tl` in the input stream (after x-expansion). Here, `\exp_not:n` is used rather than `\exp_not:N` because #3 could be a macro parameter character or could be `\s__tl` (which must be hidden behind braces in the result).

```

3976 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
3977 {
3978   \if_int_compare:w #1 = \c_zero_int
3979     \__tl_analysis_b_special:w
3980   \fi:
3981   \__tl_analysis_b_normal:wwN #1;

```

```

3982     }
3983 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
3984 {
3985     \exp_not:n { \exp_not:n { #3 } } \s__tl
3986     \if_charcode:w
3987         \scan_stop:
3988         \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
3989         \scan_stop:
3990         \exp_after:wN \__tl_analysis_b_char:Nn
3991         \exp_after:wN \__tl_analysis_b_char_aux:nww
3992     \else:
3993         \exp_after:wN \__tl_analysis_b_cs:Nww
3994     \fi:
3995     #3 #1; #2;
3996 }

```

(End definition for `\__tl_analysis_b_normals:ww` and `\__tl_analysis_b_normal:wwN`.)

`\__tl_analysis_b_char:Nn`  
`\__tl_analysis_b_char_aux:nww`

This function is called here with arguments `\__tl_analysis_b_char_aux:nww` and a normal character, while in the peek analysis code it is called with `\use_none:n` and possibly a space character, which is why the function has signature `Nn`. If the normal token we grab is a character, leave `\catcode` `\charcode` followed by `\s__tl` in the input stream, and call `\__tl_analysis_b_normals:ww` with its first argument decremented.

```

3997 \cs_new:Npx \__tl_analysis_b_char:Nn #1#2
3998 {
3999     \exp_not:N \if_meaning:w #2 \exp_not:N \tex_undefined:D
4000     \token_to_str:N D \exp_not:N \else:
4001     \exp_not:N \if_catcode:w #2 \c_catcode_other_token
4002     \token_to_str:N C \exp_not:N \else:
4003     \exp_not:N \if_catcode:w #2 \c_catcode_letter_token
4004     \token_to_str:N B \exp_not:N \else:
4005     \exp_not:N \if_catcode:w #2 \c_math_toggle_token      3
4006     \exp_not:N \else:
4007     \exp_not:N \if_catcode:w #2 \c_alignment_token        4
4008     \exp_not:N \else:
4009     \exp_not:N \if_catcode:w #2 \c_math_superscript_token 7
4010     \exp_not:N \else:
4011     \exp_not:N \if_catcode:w #2 \c_math_subscript_token   8
4012     \exp_not:N \else:
4013     \exp_not:N \if_catcode:w #2 \c_space_token
4014     \token_to_str:N A \exp_not:N \else:
4015     6
4016     \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: \fi: }
4017     #1 {#2}
4018 }
4019 \cs_new:Npn \__tl_analysis_b_char_aux:nww #1
4020 {
4021     \int_value:w '#1 \s__tl
4022     \exp_after:wN \__tl_analysis_b_normals:ww
4023     \int_value:w \int_eval:w - 1 +
4024 }

```

(End definition for `\__tl_analysis_b_char:Nn` and `\__tl_analysis_b_char_aux:nww`.)

```
\__tl_analysis_b_cs:Nww
\__tl_analysis_b_cs_test:ww
```

If the token we grab is a control sequence, leave 0 -1 (as category code and character code) in the input stream, followed by `\s__tl`, and call `\__tl_analysis_b_normals:ww` with updated arguments.

```
4025 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
4026 {
4027   0 -1 \s__tl
4028   \__tl_analysis_cs_space_count>NN \__tl_analysis_b_cs_test:ww #1
4029 }
4030 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
4031 {
4032   \exp_after:wN \__tl_analysis_b_normals:ww
4033   \int_value:w \int_eval:w
4034   \if_int_compare:w #1 = \c_zero_int
4035     #3
4036   \else:
4037     \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
4038   \fi:
4039   - #2
4040   \exp_after:wN ;
4041   \int_value:w \int_eval:n { #4 + #1 } ;
4042 }
```

(End definition for `\__tl_analysis_b_cs:Nww` and `\__tl_analysis_b_cs_test:ww`.)

```
\__tl_analysis_b_special:w
\__tl_analysis_b_special_char:wn
\__tl_analysis_b_special_space:wn
```

Here, `#1` is the current index in the array built in the first pass. Check now whether we reached the end (we shouldn't keep the trailing end-group character that marked the end of the token list in the first pass). Unpack the `\toks` register: when x-expanding again, we will get the special token. Then leave the category code in the input stream, followed by the character code, and call `\__tl_analysis_b_loop:w` with the next index.

```
4043 \group_begin:
4044   \char_set_catcode_other:N A
4045   \cs_new:Npn \__tl_analysis_b_special:w
4046     \fi: \__tl_analysis_b_normal:wwN 0 ; #1 ;
4047   {
4048     \fi:
4049     \if_int_compare:w #1 = \l__tl_analysis_index_int
4050       \exp_after:wN \prg_break:
4051     \fi:
4052     \tex_the:D \tex_toks:D #1 \s__tl
4053     \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
4054       \token_to_str:N A
4055     \or: 1
4056     \or: 1
4057     \else: 2
4058     \fi:
4059     \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
4060       \exp_after:wN \__tl_analysis_b_special_char:wn \int_value:w
4061     \else:
4062       \exp_after:wN \__tl_analysis_b_special_space:w \int_value:w
4063     \fi:
4064     \int_eval:n { 1 + #1 } \exp_after:wN ;
4065     \token_to_str:N
4066   }
4067 \group_end:
```

```

4068 \cs_new:Npn \__tl_analysis_b_special_char:wN #1 ; #2
4069   {
4070     \int_value:w '#2 \s__tl
4071     \__tl_analysis_b_loop:w #1 ;
4072   }
4073 \cs_new:Npn \__tl_analysis_b_special_space:w #1 ; ~
4074   {
4075     32 \s__tl
4076     \__tl_analysis_b_loop:w #1 ;
4077   }

```

(End definition for `\__tl_analysis_b_special:w`, `\__tl_analysis_b_special_char:wN`, and `\__tl_analysis_b_special_space:w`.)

## 44.8 Mapping through the analysis

```
\tl_analysis_map_inline:Nn
\tl_analysis_map_inline:nn
  \__tl_analysis_map:Nn
\__tl_analysis_map:NwNw
```

First obtain the analysis of the token list into `\g__tl_analysis_result_tl`. To allow nested mappings, increase the nesting depth `\g__kernel_prg_map_int` (shared between all modules), then define the payload macro, which runs the user code and has a name specific to that nesting depth. The looping macro grabs the `tokens`, `catcode` and `char code`; it checks for the end of the loop with `\use_none:n ##2`, normally empty, but which becomes `\tl_map_break:` at the end; it then calls the payload macro with the arguments in the correct order (this is the reason why we cannot directly use the same macro for looping and payload), and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```

4078 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1
4079   { \exp_args:No \tl_analysis_map_inline:nn #1 }
4080 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1
4081   {
4082     \__tl_analysis:n {#1}
4083     \int_gincr:N \g__kernel_prg_map_int
4084     \exp_args:Nc \__tl_analysis_map:Nn
4085       { \__tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }
4086   }
4087 \cs_new_protected:Npn \__tl_analysis_map:Nn #1#2
4088   {
4089     \cs_gset_protected:Npn #1 ##1##2##3 {#2}
4090     \exp_after:wN \__tl_analysis_map:NwNw \exp_after:wN #1
4091       \g__tl_analysis_result_tl
4092       \s__tl { ? \tl_map_break: } \s__tl
4093       \prg_break_point:Nn \tl_map_break:
4094         { \int_gdecr:N \g__kernel_prg_map_int }
4095   }
4096 \cs_new_protected:Npn \__tl_analysis_map:NwNw #1 #2 \s__tl #3 #4 \s__tl
4097   {
4098     \use_none:n #3
4099     #1 {#2} {#4} {#3}
4100     \__tl_analysis_map:NwNw #1
4101   }

```

(End definition for `\tl_analysis_map_inline:Nn` and others. These functions are documented on page 45.)

## 44.9 Showing the results

`\tl_analysis_show:N`

`\tl_analysis_log:N`

`\_tl_analysis_show:NNN`

Add to `\_tl_analysis:n` a third pass to display tokens to the terminal. If the token list variable is not defined, throw the same error as `\tl_show:N` by simply calling that function.

```

4102 \cs_new_protected:Npn \tl_analysis_show:N
4103   { \_tl_analysis_show:NNN \msg_show:nnxxxx \tl_show:N }
4104 \cs_new_protected:Npn \tl_analysis_log:N
4105   { \_tl_analysis_show:NNN \msg_log:nnxxxx \tl_log:N }
4106 \cs_new_protected:Npn \_tl_analysis_show:NNN #1#2#3
4107   {
4108     \tl_if_exist:NTF #3
4109     {
4110       \exp_args:No \_tl_analysis:n {#3}
4111       #1 { tl } { show-analysis }
4112       { \token_to_str:N #3 } { \_tl_analysis_show: } { } { }
4113     }
4114     { #2 #3 }
4115   }

```

(End definition for `\tl_analysis_show:N`, `\tl_analysis_log:N`, and `\_tl_analysis_show:NNN`. These functions are documented on page 45.)

`\tl_analysis_show:n`

`\tl_analysis_log:n`

`\_tl_analysis_show:Nn`

No existence test needed here.

```

4116 \cs_new_protected:Npn \tl_analysis_show:n
4117   { \_tl_analysis_show:Nn \msg_show:nnxxxx }
4118 \cs_new_protected:Npn \tl_analysis_log:n
4119   { \_tl_analysis_show:Nn \msg_log:nnxxxx }
4120 \cs_new_protected:Npn \_tl_analysis_show:Nn #1#2
4121   {
4122     \_tl_analysis:n {#2}
4123     #1 { tl } { show-analysis } { } { \_tl_analysis_show: } { } { }
4124   }

```

(End definition for `\tl_analysis_show:n`, `\tl_analysis_log:n`, and `\_tl_analysis_show:Nn`. These functions are documented on page 45.)

`\_tl_analysis_show:`

`\_tl_analysis_show_loop:wNw`

Here, #1 o- and x-expands to the token; #2 is the category code (one uppercase hexadecimal digit), 0 for control sequences; #3 is the character code, which we ignore. In the cases of control sequences and active characters, the meaning may overflow one line, and we want to truncate it. Those cases are thus separated out.

```

4125 \cs_new:Npn \_tl_analysis_show:
4126   {
4127     \exp_after:wN \_tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
4128     \s__tl { ? \prg_break: } \s__tl
4129     \prg_break_point:
4130   }
4131 \cs_new:Npn \_tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
4132   {
4133     \use_none:n #2
4134     \iow_newline: > \use:nn { ~ } { ~ }
4135     \if_int_compare:w "#2 = \c_zero_int
4136       \exp_after:wN \_tl_analysis_show_cs:n
4137     \else:

```

```

4138 \if_int_compare:w "#2 = 13 \exp_stop_f:
4139   \exp_after:wN \exp_after:wN
4140   \exp_after:wN \__tl_analysis_show_active:n
4141 \else:
4142   \exp_after:wN \exp_after:wN
4143   \exp_after:wN \__tl_analysis_show_normal:n
4144 \fi:
4145 \fi:
4146 {#1}
4147 \__tl_analysis_show_loop:wNw
4148 }

```

(End definition for `\__tl_analysis_show:` and `\__tl_analysis_show_loop:wNw`.)

`\__tl_analysis_show_normal:n` Non-active characters are a simple matter of printing the character, and its meaning. Our test suite checks that begin-group and end-group characters do not mess up TeX's alignment status.

```

4149 \cs_new:Npn \__tl_analysis_show_normal:n #1
4150 {
4151   \exp_after:wN \token_to_str:N #1 ~
4152   ( \exp_after:wN \token_to_meaning:N #1 )
4153 }

```

(End definition for `\__tl_analysis_show_normal:n`.)

`\__tl_analysis_show_value:N` This expands to the value of #1 if it has any.

```

4154 \cs_new:Npn \__tl_analysis_show_value:N #1
4155 {
4156   \token_if_expandable:NF #1
4157   {
4158     \token_if_chardef:NTF      #1 \prg_break: { }
4159     \token_if_mathchardef:NTF  #1 \prg_break: { }
4160     \token_if_dim_register:NTF #1 \prg_break: { }
4161     \token_if_int_register:NTF #1 \prg_break: { }
4162     \token_if_skip_register:NTF #1 \prg_break: { }
4163     \token_if_toks_register:NTF #1 \prg_break: { }
4164     \use_none:nnn
4165     \prg_break_point:
4166     \use:n { \exp_after:wN = \tex_the:D #1 }
4167   }
4168 }

```

(End definition for `\__tl_analysis_show_value:N`.)

`\__tl_analysis_show_cs:n` Control sequences and active characters are printed in the same way, making sure not to go beyond the `\l_iow_line_count_int`. In case of an overflow, we replace the last characters by `\c__tl_analysis_show_etc_str`.

```

4169 \cs_new:Npn \__tl_analysis_show_cs:n #1
4170   { \exp_args:No \__tl_analysis_show_long:nn {#1} { control~sequence= } }
4171 \cs_new:Npn \__tl_analysis_show_active:n #1
4172   { \exp_args:No \__tl_analysis_show_long:nn {#1} { active~character= } }
4173 \cs_new:Npn \__tl_analysis_show_long:nn #1
4174   {
4175     \__tl_analysis_show_long_aux:oofn

```

```

4176      { \token_to_str:N #1 }
4177      { \token_to_meaning:N #1 }
4178      { \__tl_analysis_show_value:N #1 }
4179  }
4180 \cs_new:Npn \__tl_analysis_show_long_aux:nnnn #1#2#3#4
4181  {
4182      \int_compare:nNnTF
4183      { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
4184      > { \l_iow_line_count_int - 3 }
4185      {
4186          \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
4187          {
4188              \l_iow_line_count_int - 3
4189              - \str_count:N \c__tl_analysis_show_etc_str
4190          }
4191          \c__tl_analysis_show_etc_str
4192      }
4193      { #1 ~ ( #4 #2 #3 ) }
4194  }
4195 \cs_generate_variant:Nn \__tl_analysis_show_long_aux:nnnn { oof }

```

(End definition for `\__tl_analysis_show_cs:n` and others.)

## 44.10 Peeking ahead

`\peek_analysis_map_break:` The break statements use the general `\prg_map_break:Nn`.

```

4196 \cs_new:Npn \peek_analysis_map_break:
4197  { \prg_map_break:Nn \peek_analysis_map_break: { } }
4198 \cs_new:Npn \peek_analysis_map_break:n
4199  { \prg_map_break:Nn \peek_analysis_map_break: }

```

(End definition for `\peek_analysis_map_break:` and `\peek_analysis_map_break:n`. These functions are documented on page 194.)

```

\l__tl_peek_chancode_int
4200 \int_new:N \l__tl_peek_chancode_int

```

(End definition for `\l__tl_peek_chancode_int`.)

`\__tl_analysis_char_arg:Nw` After a call to `\futurelet \l__tl_analysis_token` followed by a stringified character token (either explicit space or catcode other character), grab the argument and pass it to `#1`. We only need to do anything in the case of a space.

```

4201 \cs_new:Npn \__tl_analysis_char_arg:Nw
4202  {
4203      \if_meaning:w \l__tl_analysis_token \c_space_token
4204      \exp_after:wN \__tl_analysis_char_arg_aux:Nw
4205      \fi:
4206  }
4207 \cs_new:Npn \__tl_analysis_char_arg_aux:Nw #1 ~ { #1 { ~ } }

```

(End definition for `\__tl_analysis_char_arg:Nw` and `\__tl_analysis_char_arg_aux:Nw`.)

```

\peek_analysis_map_inline:n
\__tl_peek_analysis_loop:NNN
  \__tl_peek_analysis_test:
  \__tl_peek_analysis_exp:N
    \__tl_peek_analysis_exp_active:N
  \__tl_peek_analysis_nonexp:N
    \__tl_peek_analysis_cs:N
  \__tl_peek_analysis_char:N
  \__tl_peek_analysis_char:w
\__tl_peek_analysis_special:
\__tl_peek_analysis_retest:
  \__tl_peek_analysis_next:
\__tl_peek_analysis_nextii:
  \__tl_peek_analysis_str:
\__tl_peek_analysis_str:w
\__tl_peek_analysis_str:n
  \__tl_peek_analysis_active_str:n
    \__tl_peek_analysis_explicit:n
\__tl_peek_analysis_escape:
  \__tl_peek_analysis_collect:w
  \__tl_peek_analysis_collect:n
  \__tl_peek_analysis_collect_loop:
  \__tl_peek_analysis_collect_test:
\__tl_peek_analysis_collect_end:NNN

```

Save the user's code in a control sequence that is suitable for nested maps. We may wish to pass to this function an `\outer` control sequence or active character; for this we will undefine potentially-`\outer` tokens within a group, closed after the function reads its arguments (for an `\outer` active character there is no good alternative). This user's code function also calls the loop auxiliary, and includes the trailing `\prg_break_point:Nn` for when the user wants to stop the loop. The loop auxiliary must remove that break point because it must look at the input stream.

```

4208 \cs_new_protected:Npn \peek_analysis_map_inline:n #1
4209   {
4210     \group_align_safe_begin:
4211     \int_gincr:N \g__kernel_prg_map_int
4212     \cs_set_protected:cpn
4213       { \__tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4214       ##1##2##3
4215     {
4216       \group_end:
4217       #1
4218       \__tl_peek_analysis_loop:NNn
4219         \prg_break_point:Nn \peek_analysis_map_break:
4220           { \group_align_safe_end: }
4221     }
4222     \__tl_peek_analysis_loop:NNn ? ? ?
4223   }

```

The loop starts a group (closed by the user-code function defined above) with a normalized escape character, and checks if the next token is special or N-type (distinguishing expandable from non-expandable tokens).

```

4224 \cs_new_protected:Npn \__tl_peek_analysis_loop:NNn #1#2#3
4225   {
4226     \group_begin:
4227     \tl_set:Nx \l__tl_peek_code_tl
4228     {
4229       \exp_not:c
4230         { \__tl_analysis_map_ \int_use:N \g__kernel_prg_map_int :nnN }
4231     }
4232     \int_set:Nn \tex_escapechar:D { '\\" }
4233     \peek_after:Nw \__tl_peek_analysis_test:
4234   }
4235 \cs_new_protected:Npn \__tl_peek_analysis_test:
4236   {
4237     \if_case:w
4238       \if_catcode:w \exp_not:N \l_peek_token { \c_max_int \fi:
4239       \if_catcode:w \exp_not:N \l_peek_token } \c_max_int \fi:
4240       \if_meaning:w \l_peek_token \c_space_token \c_max_int \fi:
4241       \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
4242         \c_one_int
4243       \fi:
4244       \c_zero_int
4245       \exp_after:wN \exp_after:wN
4246       \exp_after:wN \__tl_peek_analysis_exp:N
4247       \exp_after:wN \exp_not:N
4248     \or:
4249       \exp_after:wN \__tl_peek_analysis_nonexp:N
4250     \else:

```

```

4251     \exp_after:wN \__tl_peek_analysis_special:
4252     \fi:
4253 }

```

Expandable tokens (which are automatically N-type) can be \outer macros, hence the need for \exp\_after:wN and \exp\_not:N in the code above, which allows the next function to safely grab the token as an argument. We run some code that is expanded using the primitive \cs\_set\_nopar:Npx rather than \tl\_set:Nx to avoid grabbing it as an argument as #1 may be \outer. To allow #1 as an argument of the user's function (stored in \l\_\_tl\_peek\_code\_t1), we set it equal to \scan\_stop:, but we do it at the last minute because #1 may be some pretty important function such as \exp\_after:wN. Then we put the user's function and the elaborate first argument \\_\_kernel\_exp\_not:w \exp\_after:wN { \exp\_not:N #1 }: indeed we cannot use \exp\_not:n {#1} as this breaks for an \outer macro and we cannot use \exp\_not:N #1, as o-expanding this yields a “notexpanded” token equal to (a weird) \relax, which would have the wrong value for primitive TeX conditionals such as \if\_meaning:w.

Then we must add {-1}0 if the token is a control sequence and {\charcode}D otherwise. Distinguishing the two cases is easy: since we have made the escape character printable, \token\_to\_str:N gives at least two characters for a control sequence versus a single one for an active character (possibly being a space). Producing the right outcome is trickier, as #1 cannot appear in either branch of the conditional (it could be \outer, or simply a TeX conditional), and can only be safely discarded by \use\_none:n if it is first hit with \exp\_not:N.

```

4254 \cs_new_protected:Npn \__tl_peek_analysis_exp:N #1
4255   {
4256     \cs_set_nopar:Npx \l__tl_peek_code_t1
4257     {
4258       \tex_let:D \exp_not:N #1 \scan_stop:
4259       \exp_not:o \l__tl_peek_code_t1
4260       {
4261         \exp_not:n { \__kernel_exp_not:w \exp_after:wN }
4262         { \exp_not:N \exp_not:N \exp_not:N #1 }
4263       }
4264       \if:w \scan_stop:
4265         \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
4266         \scan_stop:
4267         \exp_after:wN \exp_after:wN
4268         \exp_after:wN \__tl_peek_analysis_exp_active:N
4269       \else:
4270         { -1 } 0
4271         \exp_after:wN \exp_after:wN
4272         \exp_after:wN \use_none:n
4273       \fi:
4274       \exp_not:N #1
4275     }
4276     \l__tl_peek_code_t1
4277   }
4278 \cs_new:Npx \__tl_peek_analysis_exp_active:N #1
4279   { { \exp_not:N \int_value:w '#1 } \token_to_str:N D }

```

For normal non-expandable tokens we must distinguish characters (including active ones and macro parameter characters) from control sequences (whose string representation is more than one character because we made the escape character printable). For a control

sequence call the user code with suitable arguments, wrapping #1 within \exp\_not:n just in case it happens to be equal to a macro parameter character. We do not skip \exp\_not:n when unnecessary, because there might be situations where the argument could be used by the user after further redefinitions of the token, and it seems more convenient to know that \exp\_not:n is always used.

```

4280 \cs_new_protected:Npn \__tl_peek_analysis_nonexp:N #1
4281 {
4282     \if_charcode:w
4283         \scan_stop:
4284         \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
4285         \scan_stop:
4286         \exp_after:wN \__tl_peek_analysis_char:N
4287     \else:
4288         \exp_after:wN \__tl_peek_analysis_cs:N
4289     \fi:
4290     #1
4291 }
4292 \cs_new_protected:Npn \__tl_peek_analysis_cs:N #1
4293 { \l__tl_peek_code_tl { \exp_not:n {#1} } { -1 } 0 }

```

For normal characters we must determine their catcode. The main difficulty is that the character may be an active character masquerading as (i.e., set equal to) itself with a different catcode. Two approaches based on \lowercase can detect this. One could make an active character with the same catcode as #1 and change its definition before testing the catcode of #1, but in some Unicode engine this fills up the hash table uselessly. Instead, we lowercase #1 itself, changing its character code to 32, namely space (because LuaTeX cannot turn catcode 10 characters to anything else than character code 32), then we apply \\_\_tl\_analysis\_b\_char:Nn, which detects active characters by comparing them to \tex\_undefined:D, and we must have undefined the active space for this test to work—we use an x-expanding assignment to get the active space in the right place. Finally \\_\_tl\_peek\_analysis\_char:w puts the arguments in the correct order, including \exp\_not:n for macro parameter characters and active characters (the latter could be macro parameter characters, and it seems more uniform to always put \exp\_not:n).

```

4294 \group_begin:
4295 \char_set_active_eq:NN \ \scan_stop:
4296 \cs_new_protected:Npx \__tl_peek_analysis_char:N #1
4297 {
4298     \cs_set_eq:NN
4299         \char_generate:nn { 32 } { 13 }
4300         \exp_not:N \tex_undefined:D
4301         \tex_lccode:D '#1 = 32 \exp_stop_f:
4302         \tex_lowercase:D
4303         {
4304             \tl_put_right:Nx \exp_not:N \l__tl_peek_code_tl
4305             { \exp_not:n { \__tl_analysis_b_char:Nn \use_none:n } {#1} }
4306         }
4307     \exp_not:n
4308     {
4309         \exp_after:wN \__tl_peek_analysis_char:w
4310         \int_value:w
4311     }
4312     '#1
4313     \exp_not:n { \exp_after:wN \s__tl \l__tl_peek_code_tl }

```

```

4314      #1
4315    }
4316 \group_end:
4317 \cs_new_protected:Npn \__tl_peek_analysis_char:w #1 \s__tl #2#3#4
4318  {
4319    \if_charcode:w 6 #3
4320  \else:
4321    \if_charcode:w D #3
4322  \else:
4323    \exp_args:NNNo
4324  \fi:
4325  \fi:
4326  #2 { \exp_not:n {#4} } {#1} #3
4327 }

```

For special characters the idea is to eventually act with `\token_to_str:N`, then pick up one by one the characters of this string representation until hitting the token that follows. First determine the character code of (the meaning of) the  $\langle token \rangle$  (which we know is a special token), make sure the escape character is different from it, normalize the meanings of two active characters and the empty control sequence, and filter out these cases in `\__tl_peek_analysis_retest::`

```

4328 \cs_new_protected:Npn \__tl_peek_analysis_special:
4329  {
4330    \tex_let:D \l__tl_analysis_token = ~ \l_peek_token
4331    \int_set:Nn \l__tl_peekCharCode_int
4332      { \__tl_analysis_extractCharCode: }
4333    \if_int_compare:w \l__tl_peekCharCode_int = \tex_escapechar:D
4334      \int_set:Nn \tex_escapechar:D { '\v' }
4335    \fi:
4336    \char_set_active_eq:nN { \l__tl_peekCharCode_int } \scan_stop:
4337    \char_set_active_eq:nN { \tex_escapechar:D } \scan_stop:
4338    \cs_set_eq:cN { } \scan_stop:
4339    \tex_futurelet:D \l__tl_analysis_token
4340    \__tl_peek_analysis_retest:
4341  }
4342 \cs_new_protected:Npn \__tl_peek_analysis_retest:
4343  {
4344    \if_meaning:w \l__tl_analysis_token \scan_stop:
4345      \exp_after:wN \__tl_peek_analysis_normal:N
4346    \else:
4347      \exp_after:wN \__tl_peek_analysis_next:
4348    \fi:
4349 }

```

At this point we know the meaning of the  $\langle token \rangle$  in the input stream is `\l_peek_token`, either a space (32, 10) or a begin-group or end-group token (catcode 1 or 2), and we excluded a few cases that would be difficult later (empty control sequence, active character with the same character code as its meaning or as the escape character). Now look at the  $\langle next\ token \rangle$  following it using a combination of `\afterassignment` and `\futurelet`. (In fact look twice to reset an internal TeX flag in case the  $\langle next\ token \rangle$  had been hit with `\exp_not:N`.) The syntax of this primitive is `\futurelet \langle peek\ token \rangle \langle first\ token \rangle \langle next\ token \rangle`, and it sets  $\langle peek\ token \rangle$  equal to  $\langle next\ token \rangle$ . Traditionally, one takes  $\langle first\ token \rangle$  to be some macro that regains control of the code and, e.g., analyses  $\langle peek\ token \rangle$ . Here, both  $\langle first\ token \rangle$  and  $\langle next\ token \rangle$  are mostly unknown tokens in the

input stream (but we know the  $\langle first\ token \rangle$  has catcode 1, 2 or 10), where  $\langle first\ token \rangle$  was already stored as  $\backslash l\_peek\_token$ , and we regain control using  $\backslash afterassignment$ , which inserts its argument after the assignment, hence after  $\langle peek\ token \rangle$  but before  $\langle first\ token \rangle$ .

```

4350 \cs_new_protected:Npn \__tl_peek_analysis_next:
4351 {
4352     \tl_if_empty:oT { \tex_the:D \tex_everyeof:D }
4353     { \tex_everyeof:D { \scan_stop: } }
4354     \tex_afterassignment:D \__tl_peek_analysis_nextii:
4355     \tex_futurelet:D \l__tl_analysis_next_token
4356 }
4357 \cs_new_protected:Npn \__tl_peek_analysis_nextii:
4358 {
4359     \tex_afterassignment:D \__tl_peek_analysis_str:
4360     \tex_futurelet:D \l__tl_analysis_next_token
4361 }
```

We then hit the  $\langle first\ token \rangle$  with  $\backslash token\_to\_str:N$  and grab characters until finding  $\backslash l\_tl\_analysis\_next\_token$ . More precisely, by looking at the first character in the string representation of the  $\langle first\ token \rangle$  we distinguish three cases: a stringified control sequence starts with the escape character; for an explicit character we find that same character; for an explicit character we find anything else (we made sure to exclude the case of an active character whose string representation coincides with the other two cases).

```

4362 \cs_new_protected:Npn \__tl_peek_analysis_str:
4363 {
4364     \exp_after:wN \tex_futurelet:D
4365     \exp_after:wN \l__tl_analysis_token
4366     \exp_after:wN \__tl_peek_analysis_str:w
4367     \token_to_str:N
4368 }
4369 \cs_new_protected:Npn \__tl_peek_analysis_str:w
4370 {
4371     \__tl_analysis_char_arg:Nw \__tl_peek_analysis_str:n
4372     \cs_new_protected:Npn \__tl_peek_analysis_str:n #1
4373     {
4374         \int_case:nnF { '#1' }
4375         {
4376             { \l__tl_peekCharCode_int }
4377             { \__tl_peek_analysis_explicit:n {#1} }
4378             { \tex_escapechar:D } { \__tl_peek_analysis_escape: }
4379             { \__tl_peek_analysis_active_str:n {#1} }
4380         }
4381 }
```

When  $\#1$  is a stringified active character we pass appropriate arguments to the user's code; thankfully  $\backslash char\_generate:nn$  can make active characters.

```

4381 \cs_new_protected:Npn \__tl_peek_analysis_active_str:n #1
4382 {
4383     \tl_put_right:Nx \l__tl_peek_code_tl
4384     {
4385         { \char_generate:nn { '#1' } { 13 } }
4386         { \int_value:w '#1 }
4387         \token_to_str:N D
4388     }
4389     \l__tl_peek_code_tl
```

4390 }

When #1 matches the character we had extracted from the meaning of `\l_peek_token`, the token was an explicit character, which can be a standard space, or a begin-group or end-group character with some character code. In the latter two cases we call `\char_generate:nn` with suitable arguments and put suitable `\if_false: \fi:` constructions to make the result balanced and such that o-expanding or x-expanding gives back a single (unbalanced) begin-group or end-group character.

```

4391 \cs_new_protected:Npn \__tl_peek_analysis_explicit:n #1
4392 {
4393     \tl_put_right:Nx \l__tl_peek_code_tl
4394     {
4395         \if_meaning:w \l_peek_token \c_space_token
4396             { ~ } { 32 } \token_to_str:N A
4397         \else:
4398             \if_catcode:w \l_peek_token \c_group_begin_token
4399             {
4400                 \exp_not:N \exp_after:wN
4401                 \char_generate:nn { '#1 } { 1 }
4402                 \exp_not:N \if_false:
4403                     \if_false: { \fi: }
4404                     \exp_not:N \fi:
4405                 }
4406                 { \int_value:w '#1 }
4407                 1
4408             \else:
4409                 {
4410                     \exp_not:N \if_false:
4411                         { \if_false: } \fi:
4412                         \exp_not:N \fi:
4413                         \char_generate:nn { '#1 } { 2 }
4414                     }
4415                     { \int_value:w '#1 }
4416                     2
4417                 \fi:
4418             \fi:
4419         }
4420     \l__tl_peek_code_tl
4421 }
```

Finally there is the case of a special token whose string representation starts with an escape character, namely the token was a control sequence. In that case we could have grabbed the token directly as an N-type argument, but of course we couldn't know that until we had run all the various tests including stringifying the token. We are thus left with the hard work of picking up one by one the characters in the csname (being careful about spaces), until finding a token that matches the *(next token)* picked up earlier (which was not stringified), such that the control sequence that we found so far indeed has the expected meaning `\l_peek_token`. This comparison with `\l_peek_token` catches a reasonably common case like `\c_group_begin_token` in which the trailing `_` has category code other: without comparison of the constructed csname with `\l_peek_token` collection would stop at `\c`, which is wrong.

```

4422 \cs_new_protected:Npn \__tl_peek_analysis_escape:
4423 {
4424     \tl_clear:N \l__tl_internal_a_tl
```

```

4425     \tex_futurelet:D \l__tl_analysis_token
4426         \_tl_peek_analysis_collect:w
4427     }
4428 \cs_new_protected:Npn \_tl_peek_analysis_collect:w
4429     { \_tl_analysis_char_arg:Nw \_tl_peek_analysis_collect:n }
4430 \cs_new_protected:Npn \_tl_peek_analysis_collect:n #1
4431     {
4432         \tl_put_right:Nn \l__tl_internal_a_tl {#1}
4433         \_tl_peek_analysis_collect_loop:
4434     }
4435 \cs_new_protected:Npn \_tl_peek_analysis_collect_loop:
4436     {
4437         \tex_futurelet:D \l__tl_analysis_token
4438             \_tl_peek_analysis_collect_test:
4439     }
4440 \cs_new_protected:Npn \_tl_peek_analysis_collect_test:
4441     {
4442         \if_meaning:w \l__tl_analysis_token \l__tl_analysis_next_token
4443             \exp_after:WN \if_meaning:w \cs:w \l__tl_internal_a_tl \cs_end: \l_peek_token
4444                 \_tl_peek_analysis_collect_end:NNN
4445             \fi:
4446             \fi:
4447             \_tl_peek_analysis_collect:w
4448     }

```

End by calling the user code with suitable arguments (here **#1**, **#2** are `\fi:`), which closes the group begun early on.

```

4449 \cs_new_protected:Npn \_tl_peek_analysis_collect_end:NNN #1#2#3
4450     {
4451         #1 #2
4452         \tl_put_right:Nx \l__tl_peek_code_tl
4453         {
4454             { \exp_not:N \exp_not:n { \exp_not:c { \l__tl_internal_a_tl } } }
4455             { -1 }
4456             0
4457         }
4458         \l__tl_peek_code_tl
4459     }

```

(End definition for `\peek_analysis_map_inline:n` and others. This function is documented on page 194.)

## 44.11 Messages

`\c__tl_analysis_show_etc_str` When a control sequence (or active character) and its meaning are too long to fit in one line of the terminal, the end is replaced by this token list.

```

4460 \tl_const:Nx \c__tl_analysis_show_etc_str % (
4461     { \token_to_str:N \ETC. } )

```

(End definition for `\c__tl_analysis_show_etc_str`.)

```

4462 \msg_new:nnn { tl } { show-analysis }
4463     {
4464         The~token-list~ \tl_if_empty:nF {#1} { #1 ~ }

```

```
4465     \tl_if_empty:nTF {#2}
4466     { is~empty }
4467     { contains~the~tokens: #2 }
4468   }
4469 </package>
```

# Chapter 45

## I3regex implementation

```
4470  <*package>
```

```
4471  <@@=regex>
```

### 45.1 Plan of attack

Most regex engines use backtracking. This allows to provide very powerful features (back-references come to mind first), but it is costly, and raises the problem of catastrophic backtracking. Since TEX is not first and foremost a programming language, complicated code tends to run slowly, and we must use faster, albeit slightly more restrictive, techniques, coming from automata theory.

Given a regular expression of  $n$  characters, we do the following:

- (Compiling.) Analyse the regex, finding invalid input, and convert it to an internal representation.
- (Building.) Convert the compiled regex to a non-deterministic finite automaton (NFA) with  $O(n)$  states which accepts precisely token lists matching that regex.
- (Matching.) Loop through the query token list one token (one “position”) at a time, exploring in parallel every possible path (“active thread”) through the NFA, considering active threads in an order determined by the quantifiers’ greediness.

We use the following vocabulary in the code comments (and in variable names).

- *Group*: index of the capturing group,  $-1$  for non-capturing groups.
- *Position*: each token in the query is labelled by an integer  $\langle position \rangle$ , with  $\min\_pos - 1 \leq \langle position \rangle \leq \max\_pos$ . The lowest and highest positions  $\min\_pos - 1$  and  $\max\_pos$  correspond to imaginary begin and end markers (with non-existent category code and character code).  $\max\_pos$  is only set quite late in the processing.
- *Query*: the token list to which we apply the regular expression.
- *State*: each state of the NFA is labelled by an integer  $\langle state \rangle$  with  $\min\_state \leq \langle state \rangle < \max\_state$ .
- *Active thread*: state of the NFA that is reached when reading the query token list for the matching. Those threads are ordered according to the greediness of quantifiers.

- *Step*: used when matching, starts at 0, incremented every time a character is read, and is not reset when searching for repeated matches. The integer `\l_regex-step_int` is a unique id for all the steps of the matching algorithm.

We use `I3intarray` to manipulate arrays of integers. We also abuse TeX's `\toks` registers, by accessing them directly by number rather than tying them to control sequence using the `\newtoks` allocation functions. Specifically, these arrays and `\toks` are used as follows. When building, `\toks<state>` holds the tests and actions to perform in the `<state>` of the NFA. When matching,

- `\g_regex_state_active_intarray` holds the last `<step>` in which each `<state>` was active.
- `\g_regex_thread_info_intarray` consists of blocks for each `<thread>` (with  $\min\_thread \leq <thread> < \max\_thread$ ). Each block has  $1+2\l_regex_capturing_group_int$  entries: the `<state>` in which the `<thread>` currently is, followed by the beginnings of all submatches, and then the ends of all submatches. The `<threads>` are ordered starting from the best to the least preferred.
- `\g_regex_submatch_prev_intarray`, `\g_regex_submatch_begin_intarray` and `\g_regex_submatch_end_intarray` hold, for each submatch (as would be extracted by `\regex_extract_all:nnN`), the place where the submatch started to be looked for and its two end-points. For historical reasons, the minimum index is twice `max_state`, and the used registers go up to `\l_regex_submatch_int`. They are organized in blocks of `\l_regex_capturing_group_int` entries, each block corresponding to one match with all its submatches stored in consecutive entries.

When actually building the result,

- `\toks<position>` holds `<tokens>` which o- and x-expand to the `<position>`-th token in the query.
- `\g_regex_balance_intarray` holds the balance of begin-group and end-group character tokens which appear before that point in the token list.

The code is structured as follows. Variables are introduced in the relevant section. First we present some generic helper functions. Then comes the code for compiling a regular expression, and for showing the result of the compilation. The building phase converts a compiled regex to NFA states, and the automaton is run by the code in the following section. The only remaining brick is parsing the replacement text and performing the replacement. We are then ready for all the user functions. Finally, messages, and a little bit of tracing code.

## 45.2 Helpers

- `\_regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route via `\int_eval:n`.
- ```
4472 \cs_new_eq:NN \_regex_int_eval:w \tex_numexpr:D
(End definition for \_regex_int_eval:w.)
```
- `\_regex_standard_escapechar:` Make the `\escapechar` into the standard backslash.
- ```
4473 \cs_new_protected:Npn \_regex_standard_escapechar:
4474     { \int_set:Nn \tex_escapechar:D { '\\" } }
```

(End definition for `\_regex_standard_escapechar:..`)

`\_regex_toks_use:w` Unpack a `\toks` given its number.

```
4475 \cs_new:Npn \_regex_toks_use:w { \tex_the:D \tex_toks:D }
```

(End definition for `\_regex_toks_use:w.`)

`\_regex_toks_clear:N` Empty a `\toks` or set it to a value, given its number.

```
4476 \cs_new_protected:Npn \_regex_toks_clear:N #1
  { \_regex_toks_set:Nn #1 { } }
4477 \cs_new_eq:NN \_regex_toks_set:Nn \tex_toks:D
4478 \cs_new_protected:Npn \_regex_toks_set:No #1
  { \tex_toks:D #1 \exp_after:wN }
```

(End definition for `\_regex_toks_clear:N` and `\_regex_toks_set:Nn.`)

`\_regex_toks_memcpy>NNn` Copy #3 `\toks` registers from #2 onwards to #1 onwards, like C's `memcpy`.

```
4481 \cs_new_protected:Npn \_regex_toks_memcpy>NNn #1#2#3
  {
    \prg_replicate:nn {#3}
    {
      \tex_toks:D #1 = \tex_toks:D #2
      \int_incr:N #1
      \int_incr:N #2
    }
  }
```

(End definition for `\_regex_toks_memcpy>NNn.`)

`\_regex_toks_put_left:Nx` `\_regex_toks_put_right:Nx` `\_regex_toks_put_right:Nn` During the building phase we wish to add x-expanded material to `\toks`, either to the left or to the right. The expansion is done “by hand” for optimization (these operations are used quite a lot). The `Nn` version of `\_regex_toks_put_right:Nx` is provided because it is more efficient than x-expanding with `\exp_not:n`.

```
4490 \cs_new_protected:Npn \_regex_toks_put_left:Nx #1#2
  {
    \cs_set_nopar:Npx \_regex_tmp:w { #2 }
    \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
    { \exp_after:wN \_regex_tmp:w \tex_the:D \tex_toks:D #1 }
  }
4496 \cs_new_protected:Npn \_regex_toks_put_right:Nx #1#2
  {
    \cs_set_nopar:Npx \_regex_tmp:w {#2}
    \tex_toks:D #1 \exp_after:wN
    { \tex_the:D \tex_toks:D \exp_after:wN #1 \_regex_tmp:w }
  }
4502 \cs_new_protected:Npn \_regex_toks_put_right:Nn #1#2
  { \tex_toks:D #1 \exp_after:wN { \tex_the:D \tex_toks:D #1 #2 } }
```

(End definition for `\_regex_toks_put_left:Nx` and `\_regex_toks_put_right:Nx.`)

`\_regex_curr_cs_to_str:` Expands to the string representation of the token (known to be a control sequence) at the current position `\l_regex_curr_pos_int`. It should only be used in x-expansion to avoid losing a leading space.

```
4504 \cs_new:Npn \_regex_curr_cs_to_str:
```

```

4505   {
4506     \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
4507     \l__regex_curr_token_tl
4508   }

```

(End definition for `\_regex_curr_cs_to_str:.`)

`\_regex_intarray_item:NnF` Item of intarray, with a default value.

```

4509 \cs_new:Npn \_regex_intarray_item:NnF #1#2
4510   { \exp_args:Nf \_regex_intarray_item_aux:nNF { \int_eval:n {#2} } #1 }
4511 \cs_new:Npn \_regex_intarray_item_aux:nNF #1#2
4512   {
4513     \if_int_compare:w #1 > \c_zero_int
4514       \exp_after:wN \use_i:nn
4515     \else:
4516       \exp_after:wN \use_ii:nn
4517     \fi:
4518   { \_kernel_intarray_item:Nn #2 {#1} }
4519 }

```

(End definition for `\_regex_intarray_item:NnF` and `\_regex_intarray_item_aux:nNF`.)

`\_regex_maplike_break:` Analogous to `\tl_map_break:`, this correctly exits `\tl_map_inline:nn` and similar constructions and jumps to the matching `\prg_break_point:Nn \_regex_maplike_break: { }.`

```

4520 \cs_new:Npn \_regex_maplike_break:
4521   { \prg_map_break:Nn \_regex_maplike_break: { } }

```

(End definition for `\_regex_maplike_break:.`)

`\_regex_tl_odd_items:n` Map through a token list one pair at a time, leaving the odd-numbered or even-numbered items (the first item is numbered 1).

```

4522 \cs_new:Npn \_regex_tl_odd_items:n #1 { \_regex_tl_even_items:n { ? #1 } }
4523 \cs_new:Npn \_regex_tl_even_items:n #1
4524   {
4525     \_regex_tl_even_items_loop:nn #1 \q__regex_nil \q__regex_nil
4526     \prg_break_point:
4527   }
4528 \cs_new:Npn \_regex_tl_even_items_loop:nn #1#2
4529   {
4530     \_regex_use_none_delimit_by_q_nil:w #2 \prg_break: \q__regex_nil
4531     { \exp_not:n {#2} }
4532     \_regex_tl_even_items_loop:nn
4533   }

```

(End definition for `\_regex_tl_odd_items:n`, `\_regex_tl_even_items:n`, and `\_regex_tl_even_items_loop:nn`.)

### 45.2.1 Constants and variables

`\_regex_tmp:w` Temporary function used for various short-term purposes.

```

4534 \cs_new:Npn \_regex_tmp:w { }

```

(End definition for `\_regex_tmp:w`.)

```

\l__regex_internal_a_t1 Temporary variables used for various purposes.
\l__regex_internal_b_t1
\l__regex_internal_a_int
\l__regex_internal_b_int
\l__regex_internal_c_int
\l__regex_internal_bool
\l__regex_internal_seq
\g__regex_internal_tl

4535 \tl_new:N \l__regex_internal_a_t1
4536 \tl_new:N \l__regex_internal_b_t1
4537 \int_new:N \l__regex_internal_a_int
4538 \int_new:N \l__regex_internal_b_int
4539 \int_new:N \l__regex_internal_c_int
4540 \bool_new:N \l__regex_internal_bool
4541 \seq_new:N \l__regex_internal_seq
4542 \tl_new:N \g__regex_internal_tl

(End definition for \l__regex_internal_a_t1 and others.)

\l__regex_build_t1 This temporary variable is specifically for use with the tl_build machinery.
4543 \tl_new:N \l__regex_build_t1

(End definition for \l__regex_build_t1.)

\c__regex_no_match_regex This regular expression matches nothing, but is still a valid regular expression. We could
use a failing assertion, but I went for an empty class. It is used as the initial value for
regular expressions declared using \regex_new:N.
4544 \tl_const:Nn \c__regex_no_match_regex
4545 {
4546   \__regex_branch:n
4547   { \__regex_class:NnnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
4548 }

(End definition for \c__regex_no_match_regex.)

\l__regex_balance_int During this phase, \l__regex_balance_int counts the balance of begin-group and end-
group character tokens which appear before a given point in the token list. This variable
is also used to keep track of the balance in the replacement text.
4549 \int_new:N \l__regex_balance_int

(End definition for \l__regex_balance_int.)

45.2.2 Testing characters

\c__regex_ascii_min_int
\c__regex_ascii_max_control_int
\c__regex_ascii_max_int

4550 \int_const:Nn \c__regex_ascii_min_int { 0 }
4551 \int_const:Nn \c__regex_ascii_max_control_int { 31 }
4552 \int_const:Nn \c__regex_ascii_max_int { 127 }

(End definition for \c__regex_ascii_min_int, \c__regex_ascii_max_control_int, and \c__regex-
ascii_max_int.)

\c__regex_ascii_lower_int

4553 \int_const:Nn \c__regex_ascii_lower_int { 'a' - 'A' }

(End definition for \c__regex_ascii_lower_int.)

```

### 45.2.3 Internal auxiliaries

\q\_regex_recursion_stop	Internal recursion quarks.
	4554 \quark_new:N \q\_regex_recursion_stop
	(End definition for \q\_regex_recursion_stop.)
\q\_regex_nil	Internal quarks.
	4555 \quark_new:N \q\_regex_nil
	(End definition for \q\_regex_nil.)
egex_use_none_delimit_by_q_recursion_stop:w _regex_use_i_delimit_by_q_recursion_stop:nw \_regex_use_none_delimit_by_q_nil:w	Functions to gobble up to a quark.
	4556 \cs_new:Npn \_regex_use_none_delimit_by_q_recursion_stop:w 4557 #1 \q\_regex_recursion_stop { } 4558 \cs_new:Npn \_regex_use_i_delimit_by_q_recursion_stop:nw 4559 #1 #2 \q\_regex_recursion_stop {#1} 4560 \cs_new:Npn \_regex_use_none_delimit_by_q_nil:w #1 \q\_regex_nil { }
	(End definition for \_regex_use_none_delimit_by_q_recursion_stop:w, \_regex_use_i_delimit_by_q_recursion_stop:nw, and \_regex_use_none_delimit_by_q_nil:w.)
\_regex_quark_if_nil_p:n \_regex_quark_if_nil:nTF	Branching quark conditional.
	4561 \_kernel_quark_new_conditional:Nn \_regex_quark_if_nil:N { F }
	(End definition for \_regex_quark_if_nil:nTF.)
\_regex_break_point:TF \_regex_break_true:w	When testing whether a character of the query token list matches a given character class in the regular expression, we often have to test it against several ranges of characters, checking if any one of those matches. This is done with a structure like
	(test <sub>1</sub> ) ... (test <sub>n</sub> ) \_regex_break_point:TF {<true code>} {<false code>}
	If any of the tests succeeds, it calls \_regex_break_true:w, which cleans up and leaves <true code> in the input stream. Otherwise, \_regex_break_point:TF leaves the <false code> in the input stream.
	4562 \cs_new_protected:Npn \_regex_break_true:w 4563 #1 \_regex_break_point:TF #2 #3 {#2} 4564 \cs_new_protected:Npn \_regex_break_point:TF #1 #2 { #2 }
	(End definition for \_regex_break_point:TF and \_regex_break_true:w.)
\_regex_item_reverse:n	This function makes showing regular expressions easier, and lets us define \D in terms of \d for instance. There is a subtlety: the end of the query is marked by -2, and thus matches \D and other negated properties; this case is caught by another part of the code.
	4565 \cs_new_protected:Npn \_regex_item_reverse:n #1 4566 { 4567 #1 4568 \_regex_break_point:TF { } \_regex_break_true:w 4569 }
	(End definition for \_regex_item_reverse:n.)

```

\__regex_item_caseful_equal:n Simple comparisons triggering \__regex_break_true:w when true.
\__regex_item_caseful_range:nn
4570 \cs_new_protected:Npn \__regex_item_caseful_equal:n #1
4571 {
4572     \if_int_compare:w #1 = \l__regex_curr_char_int
4573         \exp_after:wN \__regex_break_true:w
4574     \fi:
4575 }
4576 \cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
4577 {
4578     \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4579         \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4580             \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
4581         \fi:
4582     \fi:
4583 }

```

*(End definition for \\_\_regex\_item\_caseful\_equal:n and \\_\_regex\_item\_caseful\_range:nn.)*

\\_\_regex\_item\_caseless\_equal:n For caseless matching, we perform the test both on the `curr_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```

4584 \cs_new_protected:Npn \__regex_item_caseless_equal:n #1
4585 {
4586     \if_int_compare:w #1 = \l__regex_curr_char_int
4587         \exp_after:wN \__regex_break_true:w
4588     \fi:
4589     \__regex_maybe_compute_ccc:
4590     \if_int_compare:w #1 = \l__regex_case_changed_char_int
4591         \exp_after:wN \__regex_break_true:w
4592     \fi:
4593 }
4594 \cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
4595 {
4596     \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
4597         \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
4598             \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
4599         \fi:
4600     \fi:
4601     \__regex_maybe_compute_ccc:
4602     \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
4603         \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
4604             \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
4605         \fi:
4606     \fi:
4607 }

```

*(End definition for \\_\_regex\_item\_caseless\_equal:n and \\_\_regex\_item\_caseless\_range:nn.)*

\\_\_regex\_compute\_case\_changed\_char: This function is called when `\l__regex_case_changed_char_int` has not yet been computed. If the current character code is in the range [65,90] (upper-case), then add 32, making it lowercase. If it is in the lower-case letter range [97,122], subtract 32.

```

4608 \cs_new_protected:Npn \__regex_compute_case_changed_char:
4609 {
4610     \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int

```

```

4611   \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
4612   \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
4613     \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
4614       \int_sub:Nn \l__regex_case_changed_char_int
4615         { \c__regex_ascii_lower_int }
4616     \fi:
4617   \fi:
4618 \else:
4619   \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
4620     \int_add:Nn \l__regex_case_changed_char_int
4621       { \c__regex_ascii_lower_int }
4622     \fi:
4623   \fi:
4624   \cs_set_eq:NN \__regex_maybe_compute_ccc: \prg_do_nothing:
4625 }
4626 \cs_new_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:

```

(End definition for `\__regex_compute_case_changed_char:`)

`\__regex_item_equal:n`  
`\__regex_item_range:nn`

Those must always be defined to expand to a `caseful` (default) or `caseless` version, and not be protected: they must expand when compiling, to hard-code which tests are caseless or caseful.

```

4627 \cs_new_eq:NN \__regex_item_equal:n ?
4628 \cs_new_eq:NN \__regex_item_range:nn ?

```

(End definition for `\__regex_item_equal:n` and `\__regex_item_range:nn`)

`\__regex_item_catcode:nT`  
`\__regex_item_catcode_reverse:nT`  
`\__regex_item_catcode:`

The argument is a sum of powers of 4 with exponents given by the allowed category codes (between 0 and 13). Dividing by a given power of 4 gives an odd result if and only if that category code is allowed. If the catcode does not match, then skip the character code tests which follow.

```

4629 \cs_new_protected:Npn \__regex_item_catcode:
4630   {
4631     "
4632     \if_case:w \l__regex_curr_catcode_int
4633       1      \or: 4      \or: 10      \or: 40
4634       \or: 100     \or:        \or: 1000     \or: 4000
4635       \or: 10000    \or:        \or: 100000   \or: 400000
4636       \or: 1000000 \or: 4000000 \else: 1*0
4637     \fi:
4638   }
4639 \cs_new_protected:Npn \__regex_item_catcode:nT #1
4640   {
4641     \if_int_odd:w \int_eval:n { #1 / \__regex_item_catcode: } \exp_stop_f:
4642       \exp_after:wN \use:n
4643     \else:
4644       \exp_after:wN \use_none:n
4645     \fi:
4646   }
4647 \cs_new_protected:Npn \__regex_item_catcode_reverse:nT #1#2
4648   { \__regex_item_catcode:nT {#1} { \__regex_item_reverse:n {#2} } }

```

(End definition for `\__regex_item_catcode:nT`, `\__regex_item_catcode_reverse:nT`, and `\__regex_item_catcode:`)

\\_\\_regex\\_item\\_exact:nn  
\\_\\_regex\\_item\\_exact\\_cs:n This matches an exact *(category)-character code* pair, or an exact control sequence, more precisely one of several possible control sequences, separated by \scan\_stop::

```

4649 \cs_new_protected:Npn \_\_regex_item_exact:nn #1#2
4650 {
4651     \if_int_compare:w #1 = \l__regex_curr_catcode_int
4652         \if_int_compare:w #2 = \l__regex_curr_char_int
4653             \exp_after:wN \exp_after:wN \exp_after:wN \_\_regex_break_true:w
4654             \fi:
4655         \fi:
4656     }
4657 \cs_new_protected:Npn \_\_regex_item_exact_cs:n #1
4658 {
4659     \int_compare:nNnTF \l__regex_curr_catcode_int = 0
4660     {
4661         \__kernel_tl_set:Nx \l__regex_internal_a_tl
4662             { \scan_stop: \_\_regex_curr_cs_to_str: \scan_stop: }
4663             \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
4664                 \l__regex_internal_a_tl
4665                 { \_\_regex_break_true:w } { }
4666             }
4667         { }
4668     }

```

(End definition for \\_\\_regex\\_item\\_exact:nn and \\_\\_regex\\_item\\_exact\\_cs:n.)

\\_\\_regex\\_item\\_cs:n Match a control sequence (the argument is a compiled regex). First test the catcode of the current token to be zero. Then perform the matching test, and break if the csname indeed matches.

```

4669 \cs_new_protected:Npn \_\_regex_item_cs:n #1
4670 {
4671     \int_compare:nNnT \l__regex_curr_catcode_int = 0
4672     {
4673         \group_begin:
4674             \_\_regex_single_match:
4675             \_\_regex_disable_submatches:
4676             \_\_regex_build_for_cs:n {#1}
4677             \bool_set_eq:NN \l__regex_saved_success_bool
4678                 \g__regex_success_bool
4679             \exp_args:Nx \_\_regex_match_cs:n { \_\_regex_curr_cs_to_str: }
4680             \if_meaning:w \c_true_bool \g__regex_success_bool
4681                 \group_insert_after:N \_\_regex_break_true:w
4682             \fi:
4683             \bool_gset_eq:NN \g__regex_success_bool
4684                 \l__regex_saved_success_bool
4685             \group_end:
4686         }
4687     }

```

(End definition for \\_\\_regex\\_item\\_cs:n.)

#### 45.2.4 Character property tests

\\_\\_regex\\_prop\_d: Character property tests for \d, \W, etc. These character properties are not affected by the (?i) option. The characters recognized by each one are as follows: \d=[0-9],  
\\_\\_regex\\_prop\_h:  
\\_\\_regex\\_prop\_s:  
\\_\\_regex\\_prop\_v:  
\\_\\_regex\\_prop\_w:  
\\_\\_regex\\_prop\_N:

`\w=[0-9A-Z_a-z]`, `\s=[\u\^\^I\^\^J\^\^L\^\^M]`, `\h=[\u\^\^I]`, `\v=[\^\^J-\^\^M]`, and the upper case counterparts match anything that the lower case does not match. The order in which the various tests appear is optimized for usual mostly lower case letter text.

```

4688 \cs_new_protected:Npn \__regex_prop_d:
4689   { \__regex_item_caseful_range:nn { '0 } { '9 } }
4690 \cs_new_protected:Npn \__regex_prop_h:
4691   {
4692     \__regex_item_caseful_equal:n { '\ }
4693     \__regex_item_caseful_equal:n { '\^\^I }
4694   }
4695 \cs_new_protected:Npn \__regex_prop_s:
4696   {
4697     \__regex_item_caseful_equal:n { '\ }
4698     \__regex_item_caseful_equal:n { '\^\^I }
4699     \__regex_item_caseful_equal:n { '\^\^J }
4700     \__regex_item_caseful_equal:n { '\^\^L }
4701     \__regex_item_caseful_equal:n { '\^\^M }
4702   }
4703 \cs_new_protected:Npn \__regex_prop_v:
4704   { \__regex_item_caseful_range:nn { '\^\^J } { '\^\^M } } % lf, vtab, ff, cr
4705 \cs_new_protected:Npn \__regex_prop_w:
4706   {
4707     \__regex_item_caseful_range:nn { 'a } { 'z }
4708     \__regex_item_caseful_range:nn { 'A } { 'Z }
4709     \__regex_item_caseful_range:nn { '0 } { '9 }
4710     \__regex_item_caseful_equal:n { '_' }
4711   }
4712 \cs_new_protected:Npn \__regex_prop_N:
4713   {
4714     \__regex_item_reverse:n
4715     { \__regex_item_caseful_equal:n { '\^\^J } }
4716   }

```

(End definition for `\__regex_prop_d:` and others.)

```

\__regex_posix_alnum: POSIX properties. No surprise.
\__regex_posix_alpha:
4717 \cs_new_protected:Npn \__regex_posix_alpha:
4718   { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_ascii:
4719 \cs_new_protected:Npn \__regex_posix_alpha:
4720   { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_cntrl:
4721 \cs_new_protected:Npn \__regex_posix_ascii:
4722   {
\__regex_posix_graph:
4723   \__regex_item_caseful_range:nn
\__regex_posix_lower:
4724     \c__regex_ascii_min_int
4725     \c__regex_ascii_max_int
\__regex_posix_print:
4726   }
\__regex_posix_punct:
4727 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_upper:
4728 \cs_new_protected:Npn \__regex_posix_cntrl:
4729   {
\__regex_posix_space:
4730   \__regex_item_caseful_range:nn
4731     \c__regex_ascii_min_int
4732     \c__regex_ascii_max_control_int
4733     \__regex_item_caseful_equal:n \c__regex_ascii_max_int
4734   }

```

```

4735 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
4736 \cs_new_protected:Npn \__regex_posix_graph:
4737   { \__regex_item_caseful_range:nn { '!' } { '\~{} } }
4738 \cs_new_protected:Npn \__regex_posix_lower:
4739   { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
4740 \cs_new_protected:Npn \__regex_posix_print:
4741   { \__regex_item_caseful_range:nn { '\` } { '\~{} } }
4742 \cs_new_protected:Npn \__regex_posix_punct:
4743   {
4744     \__regex_item_caseful_range:nn { '!' } { '/' }
4745     \__regex_item_caseful_range:nn { ':' } { '@' }
4746     \__regex_item_caseful_range:nn { '[' } { '[' }
4747     \__regex_item_caseful_range:nn { '\{ } { '\~{} }
4748   }
4749 \cs_new_protected:Npn \__regex_posix_space:
4750   {
4751     \__regex_item_caseful_equal:n { '\_{} }
4752     \__regex_item_caseful_range:nn { '\^{}I } { '\^{}M } }
4753   }
4754 \cs_new_protected:Npn \__regex_posix_upper:
4755   { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
4756 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:
4757 \cs_new_protected:Npn \__regex_posix_xdigit:
4758   {
4759     \__regex_posix_digit:
4760     \__regex_item_caseful_range:nn { 'A' } { 'F' }
4761     \__regex_item_caseful_range:nn { 'a' } { 'f' }
4762   }

```

(End definition for `\__regex_posix_alnum:` and others.)

### 45.2.5 Simple character escape

Before actually parsing the regular expression or the replacement text, we go through them once, converting `\n` to the character 10, *etc.* In this pass, we also convert any special character (\*, ?, {, etc.) or escaped alphanumeric character into a marker indicating that this was a special sequence, and replace escaped special characters and non-escaped alphanumeric characters by markers indicating that those were “raw” characters. The rest of the code can then avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

Usage: `\__regex_escape_use:nnnn` *inline 1* *inline 2* *inline 3* {*token list*}

The *token list* is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function *inline 1*, and escaped characters are fed to the function *inline 2* within an `x`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized, and those are replaced by the corresponding character, then fed to *inline 3*. The result is then left in the input stream. Spaces are ignored unless escaped.

The conversion is done within an `x`-expanding assignment.

`\__regex_escape_use:nnnn` The result is built in `\l__regex_internal_a_t1`, which is then left in the input stream. Tracing code is added as appropriate inside this token list. Go through #4 once, applying #1, #2, or #3 as relevant to each character (after de-escaping it).

```

4763 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
4764 {
4765     \group_begin:
4766         \tl_clear:N \l__regex_internal_a_tl
4767         \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
4768         \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
4769         \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
4770         \__regex_standard_escapechar:
4771         \__kernel_tl_gset:Nx \g__regex_internal_tl
4772             { \__kernel_str_to_other_fast:n {#4} }
4773         \tl_put_right:Nx \l__regex_internal_a_tl
4774             {
4775                 \exp_after:wN \__regex_escape_loop:N \g__regex_internal_tl
4776                 \scan_stop: \prg_break_point:
4777             }
4778         \exp_after:wN
4779     \group_end:
4780     \l__regex_internal_a_tl
4781 }

```

(End definition for `\__regex_escape_use:nnnn`.)

`\__regex_escape_loop:N` `\__regex_escape_loop:N` reads one character: if it is special (space, backslash, or end-marker), perform the associated action, otherwise it is simply an unescaped character. After a backslash, the same is done, but unknown characters are “escaped”.

```

4782 \cs_new:Npn \__regex_escape_loop:N #1
4783 {
4784     \cs_if_exist_use:cF { __regex_escape_\token_to_str:N #1:w }
4785         { \__regex_escape_unescaped:N #1 }
4786         \__regex_escape_loop:N
4787     }
4788 \cs_new:cpn { __regex_escape_ \c_backslash_str :w }
4789     \__regex_escape_loop:N #1
4790     {
4791         \cs_if_exist_use:cF { __regex_escape_/\token_to_str:N #1:w }
4792             { \__regex_escape_escaped:N #1 }
4793             \__regex_escape_loop:N
4794     }

```

(End definition for `\__regex_escape_loop:N` and `\__regex_escape_\:w`.)

`\__regex_escape_unescaped:N` Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

4795 \cs_new_eq:NN \__regex_escape_unescaped:N ?
4796 \cs_new_eq:NN \__regex_escape_escaped:N ?
4797 \cs_new_eq:NN \__regex_escape_raw:N ?

```

(End definition for `\__regex_escape_unescaped:N`, `\__regex_escape_escaped:N`, and `\__regex_escape_raw:N`.)

`\__regex_escape_\scan_stop:w` The loop is ended upon seeing the end-marker “`break`”, with an error if the string ended in a backslash. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

```

4798 \cs_new_eq:cN { __regex_escape_ \iow_char:N \scan_stop: :w } \prg_break:
4799 \cs_new:cpn { __regex_escape_ / \iow_char:N \scan_stop: :w }

```

```

4800  {
4801    \msg_expandable_error:nn { regex } { trailing-backslash }
4802    \prg_break:
4803  }
4804 \cs_new:cpn { __regex_escape_~:w } { }
4805 \cs_new:cpx { __regex_escape_/a:w }
4806   { \exp_not:N \__regex_escape_raw:N \iow_char:N \^G }
4807 \cs_new:cpx { __regex_escape/_t:w }
4808   { \exp_not:N \__regex_escape_raw:N \iow_char:N \^I }
4809 \cs_new:cpx { __regex_escape/_n:w }
4810   { \exp_not:N \__regex_escape_raw:N \iow_char:N \^J }
4811 \cs_new:cpx { __regex_escape/_f:w }
4812   { \exp_not:N \__regex_escape_raw:N \iow_char:N \^L }
4813 \cs_new:cpx { __regex_escape/_r:w }
4814   { \exp_not:N \__regex_escape_raw:N \iow_char:N \^M }
4815 \cs_new:cpx { __regex_escape/_e:w }
4816   { \exp_not:N \__regex_escape_raw:N \iow_char:N \^ [ }

```

(End definition for `\__regex_escape_\scan_stop::w` and others.)

`\__regex_escape/_x:w` When `\x` is encountered, `\__regex_escape_x_test:N` is responsible for grabbing some hexadecimal digits, and feeding the result to `\__regex_escape_x_end:w`. If the number is too big interrupt the assignment and produce an error, otherwise call `\__regex_escape_raw:N` on the corresponding character token.

```

4817 \cs_new:cpn { __regex_escape/_x:w } \__regex_escape_loop:N
4818  {
4819    \exp_after:wN \__regex_escape_x_end:w
4820    \int_value:w "0 \__regex_escape_x_test:N
4821  }
4822 \cs_new:Npn \__regex_escape_x_end:w #1 ;
4823  {
4824    \int_compare:nNnTF {#1} > \c_max_char_int
4825    {
4826      \msg_expandable_error:nnn { regex } { x-overflow }
4827      {#1} { \int_to_Hex:n {#1} }
4828    }
4829    {
4830      \exp_last_unbraced:Nf \__regex_escape_raw:N
4831      { \char_generate:nn {#1} { 12 } }
4832    }
4833  }

```

(End definition for `\__regex_escape/_x:w`, `\__regex_escape_x_end:w`, and `\__regex_escape_x_large:n`.)

`\__regex_escape_x_test:N` Find out whether the first character is a left brace (allowing any number of hexadecimal digits), or not (allowing up to two hexadecimal digits). We need to check for the end-of-string marker. Eventually, call either `\__regex_escape_x_loop:N` or `\__regex_escape_x:N`.

```

4834 \cs_new:Npn \__regex_escape_x_test:N #1
4835  {
4836    \if_meaning:w \scan_stop: #1
4837      \exp_after:wN \use_i:nnn \exp_after:wN ;
4838    \fi:
4839    \use:n

```

```

4840 {
4841   \if_charcode:w \c_space_token #1
4842     \exp_after:wN \__regex_escape_x_test:N
4843   \else:
4844     \exp_after:wN \__regex_escape_x_testii:N
4845     \exp_after:wN #1
4846   \fi:
4847 }
4848 }
4849 \cs_new:Npn \__regex_escape_x_testii:N #1
4850 {
4851   \if_charcode:w \c_left_brace_str #1
4852     \exp_after:wN \__regex_escape_x_loop:N
4853   \else:
4854     \__regex_hexadecimal_use:NTF #1
4855     { \exp_after:wN \__regex_escape_x:N }
4856     { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
4857   \fi:
4858 }

```

(End definition for `\__regex_escape_x_test:N` and `\__regex_escape_x_testii:N`.)

`\__regex_escape_x:N` This looks for the second digit in the unbraced case.

```

4859 \cs_new:Npn \__regex_escape_x:N #1
4860 {
4861   \if_meaning:w \scan_stop: #1
4862     \exp_after:wN \use_i:n \exp_after:wN ;
4863   \fi:
4864   \use:n
4865   {
4866     \__regex_hexadecimal_use:NTF #1
4867     { ; \__regex_escape_loop:N }
4868     { ; \__regex_escape_loop:N #1 }
4869   }
4870 }

```

(End definition for `\__regex_escape_x:N`.)

`\__regex_escape_x_loop:N` Grab hexadecimal digits, skip spaces, and at the end, check that there is a right brace, otherwise raise an error outside the assignment.

```

4871 \cs_new:Npn \__regex_escape_x_loop:N #1
4872 {
4873   \if_meaning:w \scan_stop: #1
4874     \exp_after:wN \use_ii:n
4875   \fi:
4876   \use_ii:n
4877   { ; \__regex_escape_x_loop_error:n { } {#1} }
4878   {
4879     \__regex_hexadecimal_use:NTF #1
4880     { \__regex_escape_x_loop:N }
4881     {
4882       \token_if_eq_charcode:NNTF \c_space_token #1
4883       { \__regex_escape_x_loop:N }
4884     }

```

```

4885      ;
4886      \exp_after:wN
4887      \token_if_eq_charcode:NNTF \c_right_brace_str #1
4888      { \__regex_escape_loop:N }
4889      { \__regex_escape_x_loop_error:n {#1} }
4890    }
4891  }
4892 }
4893 }
4894 \cs_new:Npn \__regex_escape_x_loop_error:n #1
4895 {
4896   \msg_expandable_error:nnn { regex } { x-missing-rbrace } {#1}
4897   \__regex_escape_loop:N #1
4898 }

```

(End definition for `\__regex_escape_x_loop:N` and `\__regex_escape_x_loop_error:.`)

`\__regex_hexadecimal_use:NTF` TeX detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

4899 \prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
4900 {
4901   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
4902     #1 \prg_return_true:
4903   \else:
4904     \if_case:w
4905       \int_eval:n { \exp_after:wN ' \token_to_str:N #1 - 'a }
4906       A
4907       \or: B
4908       \or: C
4909       \or: D
4910       \or: E
4911       \or: F
4912     \else:
4913       \prg_return_false:
4914       \exp_after:wN \use_none:n
4915     \fi:
4916     \prg_return_true:
4917   \fi:
4918 }

```

(End definition for `\__regex_hexadecimal_use:NTF`.)

`\__regex_char_if_alphanumeric:NTF` These two tests are used in the first pass when parsing a regular expression. That pass is responsible for finding escaped and non-escaped characters, and recognizing which ones have special meanings and which should be interpreted as “raw” characters. Namely,

- alphanumerics are “raw” if they are not escaped, and may have a special meaning when escaped;
- non-alphanumeric printable ascii characters are “raw” if they are escaped, and may have a special meaning when not escaped;
- characters other than printable ascii are always “raw”.

The code is ugly, and highly based on magic numbers and the ascii codes of characters. This is mostly unavoidable for performance reasons. Maybe the tests can be optimized a little bit more. Here, “alphanumeric” means 0–9, A–Z, a–z; “special” character means non-alphanumeric but printable ascii, from space (hex 20) to del (hex 7E).

```

4919 \prg_new_conditional:Npnn \__regex_char_if_special:N #1 { TF }
4920 {
4921     \if_int_compare:w '#1 > 'Z \exp_stop_f:
4922     \if_int_compare:w '#1 > 'z \exp_stop_f:
4923         \if_int_compare:w '#1 < \c_regex_ascii_max_int
4924             \prg_return_true: \else: \prg_return_false: \fi:
4925         \else:
4926             \if_int_compare:w '#1 < 'a \exp_stop_f:
4927                 \prg_return_true: \else: \prg_return_false: \fi:
4928             \fi:
4929         \else:
4930             \if_int_compare:w '#1 > '9 \exp_stop_f:
4931                 \if_int_compare:w '#1 < 'A \exp_stop_f:
4932                     \prg_return_true: \else: \prg_return_false: \fi:
4933                 \else:
4934                     \if_int_compare:w '#1 < '0 \exp_stop_f:
4935                         \if_int_compare:w '#1 < '\ \exp_stop_f:
4936                             \prg_return_false: \else: \prg_return_true: \fi:
4937                             \else: \prg_return_false: \fi:
4938                         \fi:
4939                     \fi:
4940                 }
4941 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
4942 {
4943     \if_int_compare:w '#1 > 'Z \exp_stop_f:
4944     \if_int_compare:w '#1 > 'z \exp_stop_f:
4945         \prg_return_false:
4946     \else:
4947         \if_int_compare:w '#1 < 'a \exp_stop_f:
4948             \prg_return_false: \else: \prg_return_true: \fi:
4949             \fi:
4950     \else:
4951         \if_int_compare:w '#1 > '9 \exp_stop_f:
4952             \if_int_compare:w '#1 < 'A \exp_stop_f:
4953                 \prg_return_false: \else: \prg_return_true: \fi:
4954             \else:
4955                 \if_int_compare:w '#1 < '0 \exp_stop_f:
4956                     \prg_return_false: \else: \prg_return_true: \fi:
4957                     \fi:
4958                 \fi:
4959             }

```

(End definition for `\__regex_char_if_alphanumeric:NTF` and `\__regex_char_if_special:NTF`.)

### 45.3 Compiling

A regular expression starts its life as a string of characters. In this section, we convert it to internal instructions, resulting in a “compiled” regular expression. This compiled

expression is then turned into states of an automaton in the building phase. Compiled regular expressions consist of the following:

- `\_regex_class:NnnN <boolean> {<tests>} {<min>} {<more>} {lazyness}`
- `\_regex_group:nnnN {<branches>} {<min>} {<more>} {lazyness}`, also `\_regex_group_no_capture:nnnN` and `\_regex_group_resetting:nnnN` with the same syntax.
- `\_regex_branch:n {<contents>}`
- `\_regex_command_K:`
- `\_regex_assertion:Nn <boolean> {<assertion test>}`, where the `<assertion test>` is `\_regex_b_test:` or `\_regex_Z_test:` or `\_regex_A_test:` or `\_regex_G_test:`

Tests can be the following:

- `\_regex_item_caseful_equal:n {<char code>}`
- `\_regex_item_caseless_equal:n {<char code>}`
- `\_regex_item_caseful_range:nn {<min>} {<max>}`
- `\_regex_item_caseless_range:nn {<min>} {<max>}`
- `\_regex_item_catcode:nT {<catcode bitmap>} {<tests>}`
- `\_regex_item_catcode_reverse:nT {<catcode bitmap>} {<tests>}`
- `\_regex_item_reverse:n {<tests>}`
- `\_regex_item_exact:nn {<catcode>} {<char code>}`
- `\_regex_item_exact_cs:n {<csnames>}`, more precisely given as `<csname> \scan_stop: <csname> \scan_stop: <csname>` and so on in a brace group.
- `\_regex_item_cs:n {<compiled regex>}`

#### 45.3.1 Variables used when compiling

`\l_regex_group_level_int` We make sure to open the same number of groups as we close.

4960 `\int_new:N \l_regex_group_level_int`

(End definition for `\l_regex_group_level_int`.)

`\l_regex_mode_int` While compiling, ten modes are recognized, labelled  $-63, -23, -6, -2, 0, 2, 3, 6, 23, 63$ . See section 45.3.3. We only define some of these as constants.

```
4961 \int_new:N \l_regex_mode_int
4962 \int_const:Nn \c_regex_cs_in_class_mode_int { -6 }
4963 \int_const:Nn \c_regex_cs_mode_int { -2 }
4964 \int_const:Nn \c_regex_outer_mode_int { 0 }
4965 \int_const:Nn \c_regex_catcode_mode_int { 2 }
4966 \int_const:Nn \c_regex_class_mode_int { 3 }
4967 \int_const:Nn \c_regex_catcode_in_class_mode_int { 6 }
```

(End definition for `\l_regex_mode_int` and others.)

```
\l_regex_catcodes_int  
  \l_regex_default_catcodes_int  
\l_regex_catcodes_bool
```

We wish to allow constructions such as `\c[^BE](..\cL[a-z]..)`, where the outer catcode test applies to the whole group, but is superseded by the inner catcode test. For this to work, we need to keep track of lists of allowed category codes: `\l_regex_catcodes_int` and `\l_regex_default_catcodes_int` are bitmaps, sums of  $4^c$ , for all allowed catcodes  $c$ . The latter is local to each capturing group, and we reset `\l_regex_catcodes_int` to that value after each character or class, changing it only when encountering a `\c` escape. The boolean records whether the list of categories of a catcode test has to be inverted: compare `\c[^BE]` and `\c[BE]`.

```
4968 \int_new:N \l_regex_catcodes_int  
4969 \int_new:N \l_regex_default_catcodes_int  
4970 \bool_new:N \l_regex_catcodes_bool
```

(End definition for `\l_regex_catcodes_int`, `\l_regex_default_catcodes_int`, and `\l_regex_catcodes_bool`.)

```
\c_regex_catcode_C_int  
\c_regex_catcode_B_int  
\c_regex_catcode_E_int  
\c_regex_catcode_M_int  
\c_regex_catcode_T_int  
\c_regex_catcode_P_int  
\c_regex_catcode_U_int  
\c_regex_catcode_D_int  
\c_regex_catcode_S_int  
\c_regex_catcode_L_int  
\c_regex_catcode_O_int  
\c_regex_catcode_A_int  
\c_regex_all_catcodes_int
```

Constants:  $4^c$  for each category, and the sum of all powers of 4.

```
4971 \int_const:Nn \c_regex_catcode_C_int { "1" }  
4972 \int_const:Nn \c_regex_catcode_B_int { "4" }  
4973 \int_const:Nn \c_regex_catcode_E_int { "10" }  
4974 \int_const:Nn \c_regex_catcode_M_int { "40" }  
4975 \int_const:Nn \c_regex_catcode_T_int { "100" }  
4976 \int_const:Nn \c_regex_catcode_P_int { "1000" }  
4977 \int_const:Nn \c_regex_catcode_U_int { "4000" }  
4978 \int_const:Nn \c_regex_catcode_D_int { "10000" }  
4979 \int_const:Nn \c_regex_catcode_S_int { "100000" }  
4980 \int_const:Nn \c_regex_catcode_L_int { "400000" }  
4981 \int_const:Nn \c_regex_catcode_O_int { "1000000" }  
4982 \int_const:Nn \c_regex_catcode_A_int { "4000000" }  
4983 \int_const:Nn \c_regex_all_catcodes_int { "5515155" }
```

(End definition for `\c_regex_catcode_C_int` and others.)

```
\l_regex_internal_regex
```

The compilation step stores its result in this variable.

```
4984 \cs_new_eq:NN \l_regex_internal_regex \c_regex_no_match_regex
```

(End definition for `\l_regex_internal_regex`.)

```
\l_regex_show_prefix_seq
```

This sequence holds the prefix that makes up the line displayed to the user. The various items must be removed from the right, which is tricky with a token list, hence we use a sequence.

```
4985 \seq_new:N \l_regex_show_prefix_seq
```

(End definition for `\l_regex_show_prefix_seq`.)

```
\l_regex_show_lines_int
```

A hack. To know whether a given class has a single item in it or not, we count the number of lines when showing the class.

```
4986 \int_new:N \l_regex_show_lines_int
```

(End definition for `\l_regex_show_lines_int`.)

### 45.3.2 Generic helpers used when compiling

`\_regex_two_if_eq:NNNTF`  
 Used to compare pairs of things like `\_regex_compile_special:N ?` together. It's often inconvenient to get the catcodes of the character to match so we just compare the character code. Besides, the expanding behaviour of `\if:w` is very useful as that means we can use `\c_left_brace_str` and the like.

```
4987 \prg_new_conditional:Npnn \_regex_two_if_eq:NNNN #1#2#3#4 { TF }
4988 {
4989   \if_meaning:w #1 #3
4990   \if:w #2 #4
4991   \prg_return_true:
4992   \else:
4993   \prg_return_false:
4994   \fi:
4995   \else:
4996   \prg_return_false:
4997   \fi:
4998 }
```

(End definition for `\_regex_two_if_eq:NNNTF`.)

`\_regex_get_digits:NTFw`  
`\_regex_get_digits_loop:w`  
 If followed by some raw digits, collect them one by one in the integer variable `#1`, and take the `true` branch. Otherwise, take the `false` branch.

```
4999 \cs_new_protected:Npn \_regex_get_digits:NTFw #1#2#3#4#5
5000 {
5001   \_regex_if_raw_digit:NNTF #4 #5
5002   { #1 = #5 \_regex_get_digits_loop:nw {#2} }
5003   { #3 #4 #5 }
5004 }
5005 \cs_new:Npn \_regex_get_digits_loop:nw #1#2#3
5006 {
5007   \_regex_if_raw_digit:NNTF #2 #3
5008   { #3 \_regex_get_digits_loop:nw {#1} }
5009   { \scan_stop: #1 #2 #3 }
5010 }
```

(End definition for `\_regex_get_digits:NTFw` and `\_regex_get_digits_loop:w`.)

`\_regex_if_raw_digit:NNTF`  
 Test used when grabbing digits for the `{m,n}` quantifier. It only accepts non-escaped digits.

```
5011 \prg_new_conditional:Npnn \_regex_if_raw_digit:NN #1#2 { TF }
5012 {
5013   \if_meaning:w \_regex_compile_raw:N #1
5014   \if_int_compare:w 1 < 1 #2 \exp_stop_f:
5015   \prg_return_true:
5016   \else:
5017   \prg_return_false:
5018   \fi:
5019   \else:
5020   \prg_return_false:
5021   \fi:
5022 }
```

(End definition for `\_regex_if_raw_digit:NNTF`.)

### 45.3.3 Mode

When compiling the NFA corresponding to a given regex string, we can be in ten distinct modes, which we label by some magic numbers:

- 6  $[\backslash c\{ \dots \}]$  control sequence in a class,
- 2  $\backslash c\{ \dots \}$  control sequence,
- 0 ... outer,
- 2  $\backslash c\dots$  catcode test,
- 6  $[\backslash c\dots]$  catcode test in a class,
- 63  $[\backslash c\{[\dots]\}]$  class inside mode -6,
- 23  $\backslash c\{[\dots]\}$  class inside mode -2,
- 3 [...] class inside mode 0,
- 23  $\backslash c[ \dots ]$  class inside mode 2,
- 63  $[\backslash c[ \dots ]]$  class inside mode 6.

This list is exhaustive, because  $\backslash c$  escape sequences cannot be nested, and character classes cannot be nested directly. The choice of numbers is such as to optimize the most useful tests, and make transitions from one mode to another as simple as possible.

- Even modes mean that we are not directly in a character class. In this case, a left bracket appends 3 to the mode. In a character class, a right bracket changes the mode as  $m \rightarrow (m - 15)/13$ , truncated.
- Grouping, assertion, and anchors are allowed in non-positive even modes (0, -2, -6), and do not change the mode. Otherwise, they trigger an error.
- A left bracket is special in even modes, appending 3 to the mode; in those modes, quantifiers and the dot are recognized, and the right bracket is normal. In odd modes (within classes), the left bracket is normal, but the right bracket ends the class, changing the mode from  $m$  to  $(m - 15)/13$ , truncated; also, ranges are recognized.
- In non-negative modes, left and right braces are normal. In negative modes, however, left braces trigger a warning; right braces end the control sequence, going from -2 to 0 or -6 to 3, with error recovery for odd modes.
- Properties (such as the  $\backslash d$  character class) can appear in any mode.

`\_\_regex_if_in_class:TF` Test whether we are directly in a character class (at the innermost level of nesting). There, many escape sequences are not recognized, and special characters are normal. Also, for every raw character, we must look ahead for a possible raw dash.

```

5023 \cs_new:Npn \_\_regex_if_in_class:TF
5024 {
5025     \if_int_odd:w \l__regex_mode_int
5026         \exp_after:wN \use_i:nn
5027     \else:
5028         \exp_after:wN \use_ii:nn
5029     \fi:
5030 }
```

(End definition for `\_regex_if_in_class:TF`.)

`\_regex_if_in_cs:TF` Right braces are special only directly inside control sequences (at the inner-most level of nesting, not counting groups).

```
5031 \cs_new:Npn \_regex_if_in_cs:TF
5032 {
5033     \if_int_odd:w \l__regex_mode_int
5034         \exp_after:wN \use_i:nn
5035     \else:
5036         \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
5037             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
5038         \else:
5039             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
5040         \fi:
5041     \fi:
5042 }
```

(End definition for `\_regex_if_in_cs:TF`.)

`\_regex_if_in_class_or_catcode:TF` Assertions are only allowed in modes 0, -2, and -6, *i.e.*, even, non-positive modes.

```
5043 \cs_new:Npn \_regex_if_in_class_or_catcode:TF
5044 {
5045     \if_int_odd:w \l__regex_mode_int
5046         \exp_after:wN \use_i:nn
5047     \else:
5048         \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
5049             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
5050         \else:
5051             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
5052         \fi:
5053     \fi:
5054 }
```

(End definition for `\_regex_if_in_class_or_catcode:TF`.)

`\_regex_if_within_catcode:TF` This test takes the true branch if we are in a catcode test, either immediately following it (modes 2 and 6) or in a class on which it applies (modes 23 and 63). This is used to tweak how left brackets behave in modes 2 and 6.

```
5055 \cs_new:Npn \_regex_if_within_catcode:TF
5056 {
5057     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
5058         \exp_after:wN \use_i:nn
5059     \else:
5060         \exp_after:wN \use_i:nn
5061     \fi:
5062 }
```

(End definition for `\_regex_if_within_catcode:TF`.)

`\_regex_chk_c_allowed:T` The `\c` escape sequence is only allowed in modes 0 and 3, *i.e.*, not within any other `\c` escape sequence.

```
5063 \cs_new_protected:Npn \_regex_chk_c_allowed:T
5064 {
5065     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
```

```

5066     \exp_after:wN \use:n
5067 \else:
5068     \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
5069         \exp_after:wN \exp_after:wN \exp_after:wN \use:n
5070     \else:
5071         \msg_error:nn { regex } { c-bad-mode }
5072         \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
5073     \fi:
5074     \fi:
5075 }

```

(End definition for `\_regex_chk_c_allowed:T.`)

`\_regex_mode_quit_c:` This function changes the mode as it is needed just after a catcode test.

```

5076 \cs_new_protected:Npn \_regex_mode_quit_c:
5077 {
5078     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
5079         \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
5080     \else:
5081         \if_int_compare:w \l__regex_mode_int =
5082             \c__regex_catcode_in_class_mode_int
5083             \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
5084         \fi:
5085     \fi:
5086 }

```

(End definition for `\_regex_mode_quit_c:.`)

#### 45.3.4 Framework

`\_regex_compile:w` Used when compiling a user regex or a regex for the `\c{...}` escape sequence within another regex. Start building a token list within a group (with x-expansion at the outset), and set a few variables (group level, catcodes), then start the first branch. At the end, make sure there are no dangling classes nor groups, close the last branch: we are done building `\l__regex_internal_regex`.

```

5087 \cs_new_protected:Npn \_regex_compile:w
5088 {
5089     \group_begin:
5090     \tl_build_begin:N \l__regex_build_tl
5091     \int_zero:N \l__regex_group_level_int
5092     \int_set_eq:NN \l__regex_default_catcodes_int
5093         \c__regex_all_catcodes_int
5094     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5095     \cs_set:Npn \_regex_item_equal:n { \_regex_item_caseful_equal:n }
5096     \cs_set:Npn \_regex_item_range:nn { \_regex_item_caseful_range:nn }
5097     \tl_build_put_right:Nn \l__regex_build_tl
5098         { \_regex_branch:n { \if_false: } \fi: }
5099 }
5100 \cs_new_protected:Npn \_regex_compile_end:
5101 {
5102     \_regex_if_in_class:TF
5103     {
5104         \msg_error:nn { regex } { missing-rbrack }
5105         \use:c { __regex_compile_!}: }

```

```

5106         \prg_do_nothing: \prg_do_nothing:
5107     }
5108     { }
5109     \if_int_compare:w \l__regex_group_level_int > \c_zero_int
5110         \msg_error:nnx { regex } { missing-rparen }
5111         { \int_use:N \l__regex_group_level_int }
5112     \prg_replicate:nn
5113         { \l__regex_group_level_int }
5114     {
5115         \tl_build_put_right:Nn \l__regex_build_tl
5116         {
5117             \if_false: { \fi: }
5118             \if_false: { \fi: } { 1 } { 0 } \c_true_bool
5119         }
5120         \tl_build_end:N \l__regex_build_tl
5121         \exp_args:NNNo
5122     \group_end:
5123     \tl_build_put_right:Nn \l__regex_build_tl
5124         { \l__regex_build_tl }
5125     }
5126     \fi:
5127     \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }
5128     \tl_build_end:N \l__regex_build_tl
5129     \exp_args:NNNx
5130     \group_end:
5131     \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
5132 }

```

(End definition for `\_regex_compile:w` and `\_regex_compile_end:..`)

`\_regex_compile:n` The compilation is done between `\_regex_compile:w` and `\_regex_compile_end:..`, starting in mode 0. Then `\_regex_escape_use:nnnn` distinguishes special characters, escaped alphanumerics, and raw characters, interpreting `\a`, `\x` and other sequences. The 4 trailing `\prg_do_nothing:` are needed because some functions defined later look up to 4 tokens ahead. Before ending, make sure that any `\c{...}` is properly closed. No need to check that brackets are closed properly since `\_regex_compile_end:` does that. However, catch the case of a trailing `\cL` construction.

```

5133 \cs_new_protected:Npn \_regex_compile:n #1
5134 {
5135     \_regex_compile:w
5136     \_regex_standard_escapechar:
5137     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
5138     \_regex_escape_use:nnnn
5139     {
5140         \_regex_char_if_special:NTF ##1
5141             \_regex_compile_special:N \_regex_compile_raw:N ##1
5142     }
5143     {
5144         \_regex_char_if_alphanumeric:NTF ##1
5145             \_regex_compile_escaped:N \_regex_compile_raw:N ##1
5146     }
5147     { \_regex_compile_raw:N ##1 }
5148     { #1 }
5149     \prg_do_nothing: \prg_do_nothing:

```

```

5150   \prg_do_nothing: \prg_do_nothing:
5151   \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
5152     { \msg_error:nn { regex } { c-trailing } }
5153   \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
5154     {
5155       \msg_error:nn { regex } { c-missing-rbrace }
5156       \__regex_compile_end_CS:
5157       \prg_do_nothing: \prg_do_nothing:
5158       \prg_do_nothing: \prg_do_nothing:
5159     }
5160   \__regex_compile_end:
5161 }
```

(End definition for `\__regex_compile:n`.)

- `\__regex_compile_use:n` Use a regex, regardless of whether it is given as a string (in which case we need to compile) or as a regex variable. This is used for `\regex_match_case:nn` and related functions to allow a mixture of explicit regex and regex variables.

```

5162 \cs_new_protected:Npn \__regex_compile_use:n #1
5163   {
5164     \tl_if_single_token:nT {#1}
5165     {
5166       \exp_after:wN \__regex_compile_use_aux:w
5167       \token_to_meaning:N #1 ~ \q__regex_nil
5168     }
5169     \__regex_compile:n {#1} \l__regex_internal_regex
5170   }
5171 \cs_new_protected:Npn \__regex_compile_use_aux:w #1 ~ #2 \q__regex_nil
5172   {
5173     \str_if_eq:nnT { #1 ~ } { macro:->\__regex_branch:n }
5174     { \use_i:nnn }
5175   }
```

(End definition for `\__regex_compile_use:n`.)

- `\__regex_compile_escaped:N`  
`\__regex_compile_special:N` If the special character or escaped alphanumeric has a particular meaning in regexes, the corresponding function is used. Otherwise, it is interpreted as a raw character. We distinguish special characters from escaped alphanumeric characters because they behave differently when appearing as an end-point of a range.

```

5176 \cs_new_protected:Npn \__regex_compile_special:N #1
5177   {
5178     \cs_if_exist_use:cF { __regex_compile_#1: }
5179     { \__regex_compile_raw:N #1 }
5180   }
5181 \cs_new_protected:Npn \__regex_compile_escaped:N #1
5182   {
5183     \cs_if_exist_use:cF { __regex_compile_/#1: }
5184     { \__regex_compile_raw:N #1 }
5185   }
```

(End definition for `\__regex_compile_escaped:N` and `\__regex_compile_special:N`.)

- `\__regex_compile_one:n` This is used after finding one “test”, such as `\d`, or a raw character. If that followed a catcode test (*e.g.*, `\cL`), then restore the mode. If we are not in a class, then the test is

“standalone”, and we need to add `\__regex_class:NnnnN` and search for quantifiers. In any case, insert the test, possibly together with a catcode test if appropriate.

```

5186 \cs_new_protected:Npn \__regex_compile_one:n #1
5187 {
5188     \__regex_mode_quit_c:
5189     \__regex_if_in_class:TF { }
5190     {
5191         \tl_build_put_right:Nn \l__regex_build_tl
5192         { \__regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
5193     }
5194     \tl_build_put_right:Nx \l__regex_build_tl
5195     {
5196         \if_int_compare:w \l__regex_catcodes_int <
5197             \c__regex_all_catcodes_int
5198             \__regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
5199             { \exp_not:N \exp_not:n {#1} }
5200         \else:
5201             \exp_not:N \exp_not:n {#1}
5202         \fi:
5203     }
5204     \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
5205     \__regex_if_in_class:TF { } { \__regex_compile_quantifier:w }
5206 }
```

(End definition for `\__regex_compile_one:n`.)

`\__regex_compile_abort_tokens:n`  
`\__regex_compile_abort_tokens:x`

This function places the collected tokens back in the input stream, each as a raw character. Spaces are not preserved.

```

5207 \cs_new_protected:Npn \__regex_compile_abort_tokens:n #1
5208 {
5209     \use:x
5210     {
5211         \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
5212         \__regex_compile_raw:N
5213     }
5214 }
5215 \cs_generate_variant:Nn \__regex_compile_abort_tokens:n { x }
```

(End definition for `\__regex_compile_abort_tokens:n`.)

### 45.3.5 Quantifiers

`\__regex_compile_if_quantifier:TFw`

This looks ahead and checks whether there are any quantifier (special character equal to either of `?+*{}`). This is useful for the `\u` and `\ur` escape sequences.

```

5216 \cs_new_protected:Npn \__regex_compile_if_quantifier:TFw #1#2#3#4
5217 {
5218     \token_if_eq_meaning:NNTF #3 \__regex_compile_special:N
5219     { \cs_if_exist:cTF { \__regex_compile_quantifier_#4:w } }
5220     { \use_i:nn }
5221     {#1} {#2} #3 #4
5222 }
```

(End definition for `\__regex_compile_if_quantifier:TFw`.)

\\_regex\_compile\_quantifier:w

This looks ahead and finds any quantifier (special character equal to either of `?+*{}`).

```
5223 \cs_new_protected:Npn \_regex_compile_quantifier:w #1#2
5224 {
5225     \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
5226     {
5227         \cs_if_exist_use:cF { _regex_compile_quantifier_#2:w }
5228         { \_regex_compile_quantifier_none: #1 #2 }
5229     }
5230     { \_regex_compile_quantifier_none: #1 #2 }
5231 }
```

(End definition for `\_regex_compile_quantifier:w`.)

Those functions are called whenever there is no quantifier, or a braced construction is invalid (equivalent to no quantifier, and whatever characters were grabbed are left raw).

```
5232 \cs_new_protected:Npn \_regex_compile_quantifier_none:
5233 {
5234     \tl_build_put_right:Nn \l__regex_build_tl
5235     { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
5236 }
5237 \cs_new_protected:Npn \_regex_compile_quantifier_abort:xNN #1#2#3
5238 {
5239     \_regex_compile_quantifier_none:
5240     \msg_warning:nnxx { regex } { invalid-quantifier } {#1} {#3}
5241     \_regex_compile_abort_tokens:x {#1}
5242     #2 #3
5243 }
```

(End definition for `\_regex_compile_quantifier_none:` and `\_regex_compile_quantifier_abort:xNN`.)

Once the “main” quantifier (`?, *`, `+` or a braced construction) is found, we check whether it is lazy (followed by a question mark). We then add to the compiled regex a closing brace (ending `\_regex_class:NnnnN` and friends), the start-point of the range, its end-point, and a boolean, `true` for lazy and `false` for greedy operators.

```
5244 \cs_new_protected:Npn \_regex_compile_quantifier_lazyness:nnNN #1#2#3#4
5245 {
5246     \_regex_two_if_eq>NNNNTF #3 #4 \_regex_compile_special:N ?
5247     {
5248         \tl_build_put_right:Nn \l__regex_build_tl
5249         { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
5250     }
5251     {
5252         \tl_build_put_right:Nn \l__regex_build_tl
5253         { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
5254         #3 #4
5255     }
5256 }
```

(End definition for `\_regex_compile_quantifier_lazyness:nnNN`.)

For each “basic” quantifier, `?`, `*`, `+`, feed the correct arguments to `\_regex_compile_quantifier_lazyness:nnNN`, `-1` means that there is no upper bound on the number of repetitions.

```
5257 \cs_new_protected:cpn { _regex_compile_quantifier_?:w }
```

```

5258 { \_\_regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
5259 \cs_new_protected:cpn { \_\_regex_compile_quantifier_*:w }
5260 { \_\_regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
5261 \cs_new_protected:cpn { \_\_regex_compile_quantifier_+:w }
5262 { \_\_regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }

```

(End definition for `\_\_regex_compile_quantifier_?:w`, `\_\_regex_compile_quantifier_*:w`, and `\_\_regex_compile_quantifier_+:w`.)

```

\_\_regex_compile_quantifier_{:w
\_\_regex_compile_quantifier_braced_auxi:w
\_\_regex_compile_quantifier_braced_auxii:w
\_\_regex_compile_quantifier_braced_auxiii:w

```

Three possible syntaxes: `{(int)}`, `{(int),}`, or `{(int),(int)}`. Any other syntax causes us to abort and put whatever we collected back in the input stream, as `raw` characters, including the opening brace. Grab a number into `\l_\_regex_internal_a_int`. If the number is followed by a right brace, the range is  $[a, a]$ . If followed by a comma, grab one more number, and call the `_ii` or `_iii` auxiliary. Those auxiliaries check for a closing brace, leading to the range  $[a, \infty]$  or  $[a, b]$ , encoded as  $\{a\}\{-1\}$  and  $\{a\}\{b - a\}$ .

```

5263 \cs_new_protected:cpn { \_\_regex_compile_quantifier_ \c_left_brace_str :w }
5264 {
5265     \_\_regex_get_digits:NTFw \l_\_regex_internal_a_int
5266     { \_\_regex_compile_quantifier_braced_auxi:w }
5267     { \_\_regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
5268 }
5269 \cs_new_protected:Npn \_\_regex_compile_quantifier_braced_auxi:w #1#2
5270 {
5271     \str_case_e:nnF { #1 #2 }
5272     {
5273         { \_\_regex_compile_special:N \c_right_brace_str }
5274         {
5275             \exp_args:No \_\_regex_compile_quantifier_lazyness:nnNN
5276             { \int_use:N \l_\_regex_internal_a_int } { 0 }
5277         }
5278         { \_\_regex_compile_special:N , }
5279         {
5280             \_\_regex_get_digits:NTFw \l_\_regex_internal_b_int
5281             { \_\_regex_compile_quantifier_braced_auxiii:w }
5282             { \_\_regex_compile_quantifier_braced_auxii:w }
5283         }
5284     }
5285     {
5286         \_\_regex_compile_quantifier_abort:xNN
5287         { \c_left_brace_str \int_use:N \l_\_regex_internal_a_int }
5288         #1 #2
5289     }
5290 }
5291 \cs_new_protected:Npn \_\_regex_compile_quantifier_braced_auxii:w #1#2
5292 {
5293     \_\_regex_two_if_eq:NNNNTF #1 #2 \_\_regex_compile_special:N \c_right_brace_str
5294     {
5295         \exp_args:No \_\_regex_compile_quantifier_lazyness:nnNN
5296         { \int_use:N \l_\_regex_internal_a_int } { -1 }
5297     }
5298     {
5299         \_\_regex_compile_quantifier_abort:xNN
5300         { \c_left_brace_str \int_use:N \l_\_regex_internal_a_int , }
5301         #1 #2

```

```

5302     }
5303   }
5304 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2
5305   {
5306     \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
5307     {
5308       \if_int_compare:w \l__regex_internal_a_int >
5309         \l__regex_internal_b_int
5310       \msg_error:nnnx { regex } { backwards-quantifier }
5311         { \int_use:N \l__regex_internal_a_int }
5312         { \int_use:N \l__regex_internal_b_int }
5313       \int_zero:N \l__regex_internal_b_int
5314     \else:
5315       \int_sub:Nn \l__regex_internal_b_int \l__regex_internal_a_int
5316     \fi:
5317     \exp_args:Nno \__regex_compile_quantifier_lazyness:nnNN
5318       { \int_use:N \l__regex_internal_a_int }
5319       { \int_use:N \l__regex_internal_b_int }
5320   }
5321   {
5322     \__regex_compile_quantifier_abort:xNN
5323     {
5324       \c_left_brace_str
5325       \int_use:N \l__regex_internal_a_int ,
5326       \int_use:N \l__regex_internal_b_int
5327     }
5328   #1 #2
5329 }
5330 }
```

(End definition for `\__regex_compile_quantifier_{:w}` and others.)

#### 45.3.6 Raw characters

`\__regex_compile_raw_error:N` Within character classes, and following catcode tests, some escaped alphanumeric sequences such as `\b` do not have any meaning. They are replaced by a raw character, after spitting out an error.

```

5331 \cs_new_protected:Npn \__regex_compile_raw_error:N #1
5332   {
5333     \msg_error:nnnx { regex } { bad-escape } {#1}
5334     \__regex_compile_raw:N #1
5335 }
```

(End definition for `\__regex_compile_raw_error:N`.)

`\__regex_compile_raw:N` If we are in a character class and the next character is an unescaped dash, this denotes a range. Otherwise, the current character `#1` matches itself.

```

5336 \cs_new_protected:Npn \__regex_compile_raw:N #1#2#3
5337   {
5338     \__regex_if_in_class:TF
5339     {
5340       \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N -
5341         { \__regex_compile_range:Nw #1 }
5342         {
```

```

5343     \_\_regex_compile_one:n
5344         { \_\_regex_item_equal:n { \int_value:w '#1 } }
5345         #2 #3
5346     }
5347 }
5348 {
5349     \_\_regex_compile_one:n
5350         { \_\_regex_item_equal:n { \int_value:w '#1 } }
5351         #2 #3
5352     }
5353 }

```

(End definition for `\_\_regex_compile_raw:N.`)

We have just read a raw character followed by a dash; this should be followed by an end-point for the range. Valid end-points are: any raw character; any special character, except a right bracket. In particular, escaped characters are forbidden.

```

5354 \prg_new_protected_conditional:Npnn \_\_regex_if_end_range:NN #1#2 { TF }
5355 {
5356     \if_meaning:w \_\_regex_compile_raw:N #1
5357         \prg_return_true:
5358     \else:
5359         \if_meaning:w \_\_regex_compile_special:N #1
5360             \if_charcode:w ] #2
5361                 \prg_return_false:
5362             \else:
5363                 \prg_return_true:
5364             \fi:
5365         \else:
5366             \prg_return_false:
5367         \fi:
5368     \fi:
5369 }
5370 \cs_new_protected:Npn \_\_regex_compile_range:Nw #1#2#3
5371 {
5372     \_\_regex_if_end_range:NNTF #2 #3
5373 {
5374     \if_int_compare:w '#1 > '#3 \exp_stop_f:
5375         \msg_error:nnnx { regex } { range-backwards } {#1} {#3}
5376     \else:
5377         \tl_build_put_right:Nx \l_\_regex_build_tl
5378             {
5379                 \if_int_compare:w '#1 = '#3 \exp_stop_f:
5380                     \_\_regex_item_equal:n
5381                 \else:
5382                     \_\_regex_item_range:nn { \int_value:w '#1 }
5383                 \fi:
5384                 { \int_value:w '#3 }
5385             }
5386         \fi:
5387     }
5388 {
5389     \msg_warning:nnnx { regex } { range-missing-end }
5390         {#1} { \c_backslash_str #3 }

```

```

5391          \tl_build_put_right:Nx \l__regex_build_tl
5392          {
5393              \__regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
5394              \__regex_item_equal:n { \int_value:w '- \exp_stop_f: }
5395          }
5396          #2#3
5397      }
5398  }

```

(End definition for `\__regex_compile_range:Nw` and `\__regex_if_end_range:NNTF`.)

#### 45.3.7 Character properties

`\__regex_compile_..:` In a class, the dot has no special meaning. Outside, insert `\__regex_prop_..:`, which matches any character or control sequence, and refuses `-2` (end-marker).

```

5399  \cs_new_protected:cp { __regex_compile_..: }
5400  {
5401      \exp_not:N \__regex_if_in_class:TF
5402      { \__regex_compile_raw:N . }
5403      { \__regex_compile_one:n \exp_not:c { __regex_prop_..: } }
5404  }
5405  \cs_new_protected:cpn { __regex_prop_..: }
5406  {
5407      \if_int_compare:w \l__regex_curr_char_int > - 2 \exp_stop_f:
5408          \exp_after:wN \__regex_break_true:w
5409      \fi:
5410  }

```

(End definition for `\__regex_compile_..:` and `\__regex_prop_..:`)

`\__regex_compile/_d:` The constants `\__regex_prop_d_:`, etc. hold a list of tests which match the corresponding character class, and jump to the `\__regex_break_point:TF` marker. As for a normal character, we check for quantifiers.

```

5411  \cs_set_protected:Npn \__regex_tmp:w #1#2
5412  {
5413      \cs_new_protected:cp { __regex_compile_/#1: }
5414      { \__regex_compile_one:n \exp_not:c { __regex_prop_#/1: } }
5415      \cs_new_protected:cp { __regex_compile_/#2: }
5416      {
5417          \__regex_compile_one:n
5418          { \__regex_item_reverse:n { \exp_not:c { __regex_prop_#/1: } } }
5419      }
5420  }
5421  \__regex_tmp:w d D
5422  \__regex_tmp:w h H
5423  \__regex_tmp:w s S
5424  \__regex_tmp:w v V
5425  \__regex_tmp:w w W
5426  \cs_new_protected:cpn { __regex_compile/_N: }
5427      { \__regex_compile_one:n \__regex_prop_N: }

```

(End definition for `\__regex_compile/_d:` and others.)

### 45.3.8 Anchoring and simple assertions

`\_regex_compile_anchor_letter>NNN`  
`\_regex_compile/_A:`  
`\_regex_compile/_G:`  
`\_regex_compile/_Z:`  
`\_regex_compile/_z:`  
`\_regex_compile/_b:`  
`\_regex_compile/_B:`  
`\_regex_compile/_^:`  
`\_regex_compile/_$:`

In modes where assertions are forbidden, anchors such as `\A` produce an error (`\A` is invalid in classes); otherwise they add an `\_regex_assertion:Nn` test as appropriate (the only negative assertion is `\B`). The test functions are defined later. The implementation for `$` and `^` is only different from `\A` etc because these are valid in a class.

```

5428 \cs_new_protected:Npn \_regex_compile_anchor_letter:NNN #1#2#3
5429   {
5430     \_regex_if_in_class_or_catcode:TF { \_regex_compile_raw_error:N #1 }
5431     {
5432       \tl_build_put_right:Nn \l__regex_build_tl
5433       { \_regex_assertion:Nn #2 {#3} }
5434     }
5435   }
5436 \cs_new_protected:cpn { __regex_compile/_A: }
5437   { \_regex_compile_anchor_letter:NNN A \c_true_bool \_regex_A_test: }
5438 \cs_new_protected:cpn { __regex_compile/_G: }
5439   { \_regex_compile_anchor_letter:NNN G \c_true_bool \_regex_G_test: }
5440 \cs_new_protected:cpn { __regex_compile/_Z: }
5441   { \_regex_compile_anchor_letter:NNN Z \c_true_bool \_regex_Z_test: }
5442 \cs_new_protected:cpn { __regex_compile/_z: }
5443   { \_regex_compile_anchor_letter:NNN z \c_true_bool \_regex_Z_test: }
5444 \cs_new_protected:cpn { __regex_compile/_b: }
5445   { \_regex_compile_anchor_letter:NNN b \c_true_bool \_regex_b_test: }
5446 \cs_new_protected:cpn { __regex_compile/_B: }
5447   { \_regex_compile_anchor_letter:NNN B \c_false_bool \_regex_b_test: }
5448 \cs_set_protected:Npn \_regex_tmp:w #1#2
5449   {
5450     \cs_new_protected:cpn { __regex_compile:#1: }
5451     {
5452       \_regex_if_in_class_or_catcode:TF { \_regex_compile_raw:N #1 }
5453       {
5454         \tl_build_put_right:Nn \l__regex_build_tl
5455         { \_regex_assertion:Nn \c_true_bool {#2} }
5456       }
5457     }
5458   }
5459 \exp_args:Nx \_regex_tmp:w { \iow_char:N \^ } { \_regex_A_test: }
5460 \exp_args:Nx \_regex_tmp:w { \iow_char:N \$ } { \_regex_Z_test: }

(End definition for \_regex_compile_anchor_letter:NNN and others.)

```

### 45.3.9 Character classes

`\_regex_compile[_]:` Outside a class, right brackets have no meaning. In a class, change the mode ( $m \rightarrow (m - 15)/13$ , truncated) to reflect the fact that we are leaving the class. Look for quantifiers, unless we are still in a class after leaving one (the case of `[...]\cL[...]`). quantifiers.

```

5461 \cs_new_protected:cpn { __regex_compile[_]: }
5462   {
5463     \_regex_if_in_class:TF
5464     {
5465       \if_int_compare:w \l__regex_mode_int >
5466         \c__regex_catcode_in_class_mode_int
5467       \tl_build_put_right:Nn \l__regex_build_tl { \if_false: { \fi: } }

```

```

5468     \fi:
5469     \tex_advance:D \l__regex_mode_int - 15 \exp_stop_f:
5470     \tex_divide:D \l__regex_mode_int 13 \exp_stop_f:
5471     \if_int_odd:w \l__regex_mode_int \else:
5472         \exp_after:wN \__regex_compile_quantifier:w
5473     \fi:
5474 }
5475 { \__regex_compile_raw:N }
5476 }

```

(End definition for `\__regex_compile_J::`)

- `\__regex_compile_[:]`: In a class, left brackets might introduce a POSIX character class, or mean nothing. Immediately following `\c<category>`, we must insert the appropriate catcode test, then parse the class; we pre-expand the catcode as an optimization. Otherwise (modes 0, -2 and -6) just parse the class. The mode is updated later.

```

5477 \cs_new_protected:cpn { \__regex_compile_[:}
5478 {
5479     \__regex_if_in_class:TF
5480     { \__regex_compile_class_posix_test:w }
5481     {
5482         \__regex_if_within_catcode:TF
5483         {
5484             \exp_after:wN \__regex_compile_class_catcode:w
5485             \int_use:N \l__regex_catcodes_int ;
5486         }
5487         { \__regex_compile_class_normal:w }
5488     }
5489 }

```

(End definition for `\__regex_compile_[:::`)

- `\__regex_compile_class_normal:w`: In the “normal” case, we insert `\__regex_class:NnnnN <boolean>` in the compiled code. The `<boolean>` is true for positive classes, and false for negative classes, characterized by a leading `^`. The auxiliary `\__regex_compile_class:TFNN` also checks for a leading `]` which has a special meaning.

```

5490 \cs_new_protected:Npn \__regex_compile_class_normal:w
5491 {
5492     \__regex_compile_class:TFNN
5493     { \__regex_class:NnnnN \c_true_bool }
5494     { \__regex_class:NnnnN \c_false_bool }
5495 }

```

(End definition for `\__regex_compile_class_normal:w`)

- `\__regex_compile_class_catcode:w`: This function is called for a left bracket in modes 2 or 6 (catcode test, and catcode test within a class). In mode 2 the whole construction needs to be put in a class (like single character). Then determine if the class is positive or negative, inserting `\__regex_item_catcode:nT` or the `reverse` variant as appropriate, each with the current catcodes bitmap #1 as an argument, and reset the catcodes.

```

5496 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
5497 {
5498     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
5499         \tl_build_put_right:Nn \l__regex_build_tl

```

```

5500     { \_\_regex\_class:NnnnN \c\_true\_bool { \if\_false: } \fi: }
5501     \fi:
5502     \int\_set\_eq:NN \l\_\_regex\_catcodes\_int \l\_\_regex\_default\_catcodes\_int
5503     \_\_regex\_compile\_class:TFNN
5504     { \_\_regex\_item\_catcode:nT {\#1} }
5505     { \_\_regex\_item\_catcode\_reverse:nT {\#1} }
5506   }

```

(End definition for \\_\\_regex\\_compile\\_class\\_catcode:w.)

\\_\\_regex\\_compile\\_class:TFNN  
\\_\\_regex\\_compile\\_class>NN If the first character is ^, then the class is negative (use #2), otherwise it is positive (use #1). If the next character is a right bracket, then it should be changed to a raw one.

```

5507 \cs\_new\_protected:Npn \_\_regex\_compile\_class:TFNN #1#2#3#4
5508   {
5509     \l\_\_regex\_mode\_int = \int\_value:w \l\_\_regex\_mode\_int 3 \exp\_stop\_f:
5510     \_\_regex\_two\_if\_eq:NNNNTF #3 #4 \_\_regex\_compile\_special:N "
5511     {
5512       \tl\_build\_put\_right:Nn \l\_\_regex\_build\_tl { #2 { \if\_false: } \fi: }
5513       \_\_regex\_compile\_class>NN
5514     }
5515     {
5516       \tl\_build\_put\_right:Nn \l\_\_regex\_build\_tl { #1 { \if\_false: } \fi: }
5517       \_\_regex\_compile\_class>NN #3 #4
5518     }
5519   }
5520 \cs\_new\_protected:Npn \_\_regex\_compile\_class>NN #1#2
5521   {
5522     \token\_if\_eq\_charcode:NNTF #2 ]
5523     { \_\_regex\_compile\_raw:N #2 }
5524     { #1 #2 }
5525   }

```

(End definition for \\_\\_regex\\_compile\\_class:TFNN and \\_\\_regex\\_compile\\_class>NN.)

\\_\\_regex\\_compile\\_class\\_posix\\_test:w  
\\_\\_regex\\_compile\\_class\\_posix:NNNNw  
\\_\\_regex\\_compile\\_class\\_posix\\_loop:w  
\\_\\_regex\\_compile\\_class\\_posix\\_end:w Here we check for a syntax such as [:alpha:]. We also detect [= and [. which have a meaning in POSIX regular expressions, but are not implemented in l3regex. In case we see [:, grab raw characters until hopefully reaching :]. If that's missing, or the POSIX class is unknown, abort. If all is right, add the test to the current class, with an extra \\_\\_regex\\_item\\_reverse:n for negative classes (we make sure to wrap its argument in braces otherwise \regex\\_show:N would not recognize the regex as valid).

```

5526 \cs\_new\_protected:Npn \_\_regex\_compile\_class\_posix\_test:w #1#2
5527   {
5528     \token\_if\_eq\_meaning:NNT \_\_regex\_compile\_special:N #1
5529     {
5530       \str\_case:nn { #2 }
5531       {
5532         : { \_\_regex\_compile\_class\_posix:NNNNw }
5533         = {
5534           \msg\_warning:nnx { regex }
5535             { posix-unsupported } { = }
5536         }
5537         . {
5538           \msg\_warning:nnx { regex }
5539             { posix-unsupported } { . }

```

```

5540         }
5541     }
5542   }
5543   \__regex_compile_raw:N [ #1 #2
5544 ]
5545 \cs_new_protected:Npn \__regex_compile_class_posix:NNNw #1#2#3#4#5#6
5546 {
5547   \__regex_two_if_eq>NNNNTF #5 #6 \__regex_compile_special:N ^
5548   {
5549     \bool_set_false:N \l__regex_internal_bool
5550     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5551     \__regex_compile_class_posix_loop:w
5552   }
5553   {
5554     \bool_set_true:N \l__regex_internal_bool
5555     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5556     \__regex_compile_class_posix_loop:w #5 #6
5557   }
5558 }
5559 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
5560 {
5561   \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
5562   { #2 \__regex_compile_class_posix_loop:w }
5563   { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
5564 }
5565 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
5566 {
5567   \__regex_two_if_eq>NNNNTF #1 #2 \__regex_compile_special:N :
5568   { \__regex_two_if_eq>NNNNTF #3 #4 \__regex_compile_special:N ] }
5569   { \use_i:nn }
5570   {
5571     \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
5572     {
5573       \__regex_compile_one:n
5574       {
5575         \bool_if:NTF \l__regex_internal_bool \use:n \__regex_item_reverse:n
5576         { \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : } }
5577       }
5578     }
5579   {
5580     \msg_warning:nnx { regex } { posix-unknown }
5581     { \l__regex_internal_a_tl }
5582     \__regex_compile_abort_tokens:x
5583     {
5584       [: \bool_if:NF \l__regex_internal_bool { ^ } ]
5585       \l__regex_internal_a_tl :]
5586     }
5587   }
5588 }
5589 {
5590   \msg_error:nnxx { regex } { posix-missing-close }
5591   { [: \l__regex_internal_a_tl ] { #2 #4 }
5592   \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl ]
5593   #1 #2 #3 #4

```

```

5594     }
5595 }
```

(End definition for `\_\_regex\_compile\_class\_posix\_test:w` and others.)

### 45.3.10 Groups and alternations

The contents of a regex group are turned into compiled code in `\l_\_regex_build_t1`, which ends up with items of the form `\_\_regex_branch:n {<concatenation>}`. This construction is done using `\tl_build_...` functions within a T<sub>E</sub>X group, which automatically makes sure that options (case-sensitivity and default catcode) are reset at the end of the group. The argument #1 is `\_\_regex_group:nnN` or a variant thereof. A small subtlety to support `\cL(abc)` as a shorthand for `(\cLa\cLb\cLc)`: exit any pending catcode test, save the category code at the start of the group as the default catcode for that group, and make sure that the catcode is restored to the default outside the group.

```

5596 \cs_new_protected:Npn \_\_regex_compile_group_begin:N #1
5597 {
5598     \tl_build_put_right:Nn \l_\_regex_build_t1 { #1 { \if_false: } \fi: }
5599     \_\_regex_mode_quit_c:
5600     \group_begin:
5601         \tl_build_begin:N \l_\_regex_build_t1
5602         \int_set_eq:NN \l_\_regex_default_catcodes_int \l_\_regex_catcodes_int
5603         \int_incr:N \l_\_regex_group_level_int
5604         \tl_build_put_right:Nn \l_\_regex_build_t1
5605             { \_\_regex_branch:n { \if_false: } \fi: }
5606     }
5607 \cs_new_protected:Npn \_\_regex_compile_group_end:
5608 {
5609     \if_int_compare:w \l_\_regex_group_level_int > \c_zero_int
5610         \tl_build_put_right:Nn \l_\_regex_build_t1 { \if_false: { \fi: } }
5611         \tl_build_end:N \l_\_regex_build_t1
5612         \exp_args:NNNx
5613         \group_end:
5614         \tl_build_put_right:Nn \l_\_regex_build_t1 { \l_\_regex_build_t1 }
5615         \int_set_eq:NN \l_\_regex_catcodes_int \l_\_regex_default_catcodes_int
5616         \exp_after:wN \_\_regex_compile_quantifier:w
5617     \else:
5618         \msg_warning:nn { regex } { extra-rparen }
5619         \exp_after:wN \_\_regex_compile_raw:N \exp_after:wN )
5620     \fi:
5621 }
```

(End definition for `\_\_regex_compile_group_begin:N` and `\_\_regex_compile_group_end:..`)

`\_\_regex_compile_()`: In a class, parentheses are not special. In a catcode test inside a class, a left parenthesis gives an error, to catch `[a\cL(bcd)e]`. Otherwise check for a ?, denoting special groups, and run the code for the corresponding special group.

```

5622 \cs_new_protected:cpn { \_\_regex_compile_(: }
5623 {
5624     \_\_regex_if_in_class:TF { \_\_regex_compile_raw:N ( }
5625     {
5626         \if_int_compare:w \l_\_regex_mode_int =
```

```

5627     \c__regex_catcode_in_class_mode_int
5628     \msg_error:nn { regex } { c-lparen-in-class }
5629     \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
5630     \else:
5631         \exp_after:wN \__regex_compile_lparen:w
5632     \fi:
5633 }
5634 }
5635 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
5636 {
5637     \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N ?
5638     {
5639         \cs_if_exist_use:cF
5640         {
5641             \__regex_compile_special_group_\token_to_str:N #4 :w
5642             {
5643                 \msg_warning:nnx { regex } { special-group-unknown }
5644                 {
5645                     \__regex_compile_group_begin:N \__regex_group:nnnN
5646                     \__regex_compile_raw:N ? #3 #4
5647                 }
5648             }
5649             \__regex_compile_group_begin:N \__regex_group:nnnN
5650             #1 #2 #3 #4
5651         }
5652     }

```

(End definition for `\__regex_compile_()`)

`\__regex_compile_|:` In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

5653 \cs_new_protected:cpn { __regex_compile_|: }
5654 {
5655     \__regex_if_in_class:TF { \__regex_compile_raw:N | }
5656     {
5657         \tl_build_put_right:Nn \l__regex_build_tl
5658         {
5659             \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
5660     }

```

(End definition for `\__regex_compile_|:()`)

`\__regex_compile_():` Within a class, parentheses are not special. Outside, close a group.

```

5661 \cs_new_protected:cpn { __regex_compile_()): }
5662 {
5663     \__regex_if_in_class:TF { \__regex_compile_raw:N ) }
5664     {
5665         \__regex_compile_group_end:
5666     }

```

(End definition for `\__regex_compile_():()`)

`\__regex_compile_special_group_::w`  
`\__regex_compile_special_group_|:w` Non-capturing, and resetting groups are easy to take care of during compilation; for those groups, the harder parts come when building.

```

5666 \cs_new_protected:cpn { __regex_compile_special_group_::w }
5667     {
5668         \__regex_compile_group_begin:N \__regex_group_no_capture:nnnN }

```

```

5668 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
5669   { \__regex_compile_group_begin:N \__regex_group_resetting:nnN }

(End definition for \__regex_compile_special_group_::w and \__regex_compile_special_group_|:w.)
```

\\_\_regex\_compile\_special\_group\_i:w  
\\_\_regex\_compile\_special\_group\_-:w

The match can be made case-insensitive by setting the option with (?i); the original behaviour is restored by (?-i). This is the only supported option.

```

5670 \cs_new_protected:Npn \__regex_compile_special_group_i:w #1#2
5671   {
5672     \__regex_two_if_eq>NNNNTF #1 #2 \__regex_compile_special:N )
5673     {
5674       \cs_set:Npn \__regex_item_equal:n
5675         { \__regex_item_caseless_equal:n }
5676       \cs_set:Npn \__regex_item_range:nn
5677         { \__regex_item_caseless_range:nn }
5678     }
5679   {
5680     \msg_warning:nnx { regex } { unknown-option } { (?i #2) }
5681     \__regex_compile_raw:N (
5682     \__regex_compile_raw:N ?
5683     \__regex_compile_raw:N i
5684     #1 #2
5685   }
5686   }
5687 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
5688   {
5689     \__regex_two_if_eq>NNNNTF #1 #2 \__regex_compile_raw:N i
5690     { \__regex_two_if_eq>NNNNTF #3 #4 \__regex_compile_special:N ) }
5691     { \use_i:nn }
5692   {
5693     \cs_set:Npn \__regex_item_equal:n
5694       { \__regex_item_caseful_equal:n }
5695     \cs_set:Npn \__regex_item_range:nn
5696       { \__regex_item_caseful_range:nn }
5697   }
5698   {
5699     \msg_warning:nnx { regex } { unknown-option } { (?-#2#4) }
5700     \__regex_compile_raw:N (
5701     \__regex_compile_raw:N ?
5702     \__regex_compile_raw:N -
5703     #1 #2 #3 #4
5704   }
5705 }
```

(End definition for \\_\_regex\_compile\_special\_group\_i:w and \\_\_regex\_compile\_special\_group\_-:w.)

### 45.3.11 Catcodes and csnames

\\_\_regex\_compile/\_c:  
\\_\_regex\_compile\_c\_test:NN

The \c escape sequence can be followed by a capital letter representing a character category, by a left bracket which starts a list of categories, or by a brace group holding a regular expression for a control sequence name. Otherwise, raise an error.

```

5706 \cs_new_protected:cpn { __regex_compile/_c: }
5707   { \__regex_chk_c_allowed:T { \__regex_compile_c_test:NN } }
5708 \cs_new_protected:Npn \__regex_compile_c_test:NN #1#2
```

```

5709   {
5710     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5711     {
5712       \int_if_exist:cTF { c__regex_catcode_#2_int }
5713       {
5714         \int_set_eq:Nc \l__regex_catcodes_int
5715           { c__regex_catcode_#2_int }
5716         \l__regex_mode_int
5717           = \if_case:w \l__regex_mode_int
5718             \c__regex_catcode_mode_int
5719           \else:
5720             \c__regex_catcode_in_class_mode_int
5721           \fi:
5722         \token_if_eq_charcode:NNT C #2 { \__regex_compile_c_C>NN }
5723       }
5724     }
5725     \cs_if_exist_use:cF { __regex_compile_c_#2:w } }
5726     {
5727       \msg_error:nnx { regex } { c-missing-category } {#2}
5728       #1 #2
5729     }
5730   }

```

(End definition for `\__regex_compile_c:` and `\__regex_compile_c_test>NN`.)

`\__regex_compile_c_C>NN` If `\cC` is not followed by `.` or `(...)` then complain because that construction cannot match anything, except in cases like `\cC[\c{...}]`, where it has no effect.

```

5731 \cs_new_protected:Npn \__regex_compile_c_C>NN #1#2
5732   {
5733     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
5734     {
5735       \token_if_eq_charcode:NNTF #2 .
5736         { \use_none:n }
5737         { \token_if_eq_charcode>NNF #2 ( ) % }
5738     }
5739     { \use:n }
5740     { \msg_error:nnn { regex } { c-C-invalid } {#2} }
5741     #1 #2
5742   }

```

(End definition for `\__regex_compile_c_C>NN`.)

`\__regex_compile_c_[:w]` When encountering `\c[`, the task is to collect uppercase letters representing character categories. First check for `^` which negates the list of category codes.

```

5743 \cs_new_protected:cpx { __regex_compile_c_[:w] } #1#2
5744   {
5745     \l__regex_mode_int
5746       = \if_case:w \l__regex_mode_int
5747         \c__regex_catcode_mode_int
5748       \else:
5749         \c__regex_catcode_in_class_mode_int
5750       \fi:
5751     \int_zero:N \l__regex_catcodes_int
5752     \__regex_two_if_eq>NNNNTF #1 #2 \__regex_compile_special:N ^

```

```

5753 {
5754   \bool_set_false:N \l__regex_catcodes_bool
5755   \__regex_compile_c_lbrack_loop>NN
5756 }
5757 {
5758   \bool_set_true:N \l__regex_catcodes_bool
5759   \__regex_compile_c_lbrack_loop>NN
5760   #1 #2
5761 }
5762 }
5763 \cs_new_protected:Npn \__regex_compile_c_lbrack_loop>NN #1#2
5764 {
5765   \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5766   {
5767     \int_if_exist:cTF { c__regex_catcode_#2_int }
5768     {
5769       \exp_args:Nc \__regex_compile_c_lbrack_add:N
5770       { c__regex_catcode_#2_int }
5771       \__regex_compile_c_lbrack_loop>NN
5772     }
5773   }
5774   {
5775     \token_if_eq_charcode:NNTF #2 ]
5776     { \__regex_compile_c_lbrack_end: }
5777   }
5778   {
5779     \msg_error:nnx { regex } { c-missing-rbrack } {#2}
5780     \__regex_compile_c_lbrack_end:
5781     #1 #2
5782   }
5783 }
5784 \cs_new_protected:Npn \__regex_compile_c_lbrack_add:N #1
5785 {
5786   \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
5787   \else:
5788     \int_add:Nn \l__regex_catcodes_int {#1}
5789   \fi:
5790 }
5791 \cs_new_protected:Npn \__regex_compile_c_lbrack_end:
5792 {
5793   \if_meaning:w \c_false_bool \l__regex_catcodes_bool
5794   \int_set:Nn \l__regex_catcodes_int
5795   { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
5796   \fi:
5797 }

```

(End definition for `\__regex_compile_c_[:w` and others.)

`\__regex_compile_c_{}`: The case of a left brace is easy, based on what we have done so far: in a group, compile the regular expression, after changing the mode to forbid nesting `\c{...}`. Additionally, disable submatch tracking since groups don't escape the scope of `\c{...}`.

```

5798 \cs_new_protected:cpx { __regex_compile_c_ \c_left_brace_str :w }
5799 {
5800   \__regex_compile:w

```

```

5801     \_\_regex\_disable\_submatches:
5802     \l\_regex\_mode\_int
5803         = \if_case:w \l\_regex\_mode\_int
5804             \c\_regex\_cs\_mode\_int
5805         \else:
5806             \c\_regex\_cs\_in\_class\_mode\_int
5807         \fi:
5808     }

```

(End definition for `\_\_regex\_compile_c_{..}`)

`\_\_regex\_compile_{..}`: We forbid unescaped left braces inside a `\c{...}` escape because they otherwise lead to the confusing question of whether the first right brace in `\c{{}x}` should end `\c` or whether one should match braces.

```

5809 \cs_new_protected:cpn { \_\_regex_compile_ \c_left_brace_str : }
5810 {
5811     \_\_regex_if_in_cs:TF
5812     { \msg_error:n { regex } { cu-lbrace } { c } }
5813     { \exp_after:wN \_\_regex_compile_raw:N \c_left_brace_str }
5814 }

```

(End definition for `\_\_regex_compile_{..}`)

`\_\_regex_compile_{..}`: Non-escaped right braces are only special if they appear when compiling the regular expression for a csname, but not within a class: `\c{{}}` matches the control sequences `\{` and `\}`. So, end compiling the inner regex (this closes any dangling class or group). Then insert the corresponding test in the outer regex. As an optimization, if the control sequence test simply consists of several explicit possibilities (branches) then use `\_\_regex_item_exact_cs:n` with an argument consisting of all possibilities separated by `\scan_stop:`.

```

5815 \flag_new:n { \_\_regex_cs }
5816 \cs_new_protected:cpn { \_\_regex_compile_ \c_right_brace_str : }
5817 {
5818     \_\_regex_if_in_cs:TF
5819     { \_\_regex_compile_end_cs: }
5820     { \exp_after:wN \_\_regex_compile_raw:N \c_right_brace_str }
5821 }
5822 \cs_new_protected:Npn \_\_regex_compile_end_cs:
5823 {
5824     \_\_regex_compile_end:
5825     \flag_clear:n { \_\_regex_cs }
5826     \_\_kernel_tl_set:Nx \l\_regex_internal_a_tl
5827     {
5828         \exp_after:wN \_\_regex_compile_cs_aux:Nn \l\_regex_internal_regex
5829         \q\_regex_nil \q\_regex_nil \q\_regex_recursion_stop
5830     }
5831     \exp_args:Nx \_\_regex_compile_one:n
5832     {
5833         \flag_if_raised:nTF { \_\_regex_cs }
5834         { \_\_regex_item_cs:n { \exp_not:o \l\_regex_internal_regex } }
5835         {
5836             \_\_regex_item_exact_cs:n
5837             { \tl_tail:N \l\_regex_internal_a_tl }
5838         }

```

```

5839     }
5840   }
5841 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
5842   {
5843     \cs_if_eq:NNTF #1 \__regex_branch:n
5844     {
5845       \scan_stop:
5846       \__regex_compile_cs_aux>NNnnnN #2
5847       \q__regex_nil \q__regex_nil \q__regex_nil
5848       \q__regex_nil \q__regex_nil \q__regex_nil \q__regex_recursion_stop
5849       \__regex_compile_cs_aux:Nn
5850     }
5851   {
5852     \__regex_quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
5853     \__regex_use_none_delimit_by_q_recursion_stop:w
5854   }
5855 }
5856 \cs_new:Npn \__regex_compile_cs_aux>NNnnnN #1#2#3#4#5#6
5857   {
5858     \bool_lazy_all:nTF
5859     {
5860       { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
5861       {#2}
5862       { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
5863       { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
5864       { \int_compare_p:nNn {#5} = { 0 } }
5865     }
5866   {
5867     \prg_replicate:nn {#4}
5868       { \char_generate:nn { \use_i:nn #3 } {12} }
5869       \__regex_compile_cs_aux>NNnnnN
5870   }
5871   {
5872     \__regex_quark_if_nil:NF #
5873     {
5874       \flag_raise_if_clear:n { __regex_cs }
5875       \__regex_use_i_delimit_by_q_recursion_stop:nw
5876     }
5877     \__regex_use_none_delimit_by_q_recursion_stop:w
5878   }
5879 }

```

(End definition for `\__regex_compile_`: and others.)

#### 45.3.12 Raw token lists with \u

`\__regex_compile/_u:` The `\u` escape is invalid in classes and directly following a catcode test. Otherwise test for a following `r` (for `\ur`), and call an auxiliary responsible for finding the variable name.

```

5880 \cs_new_protected:cpn { __regex_compile/_u: } #1#2
5881   {
5882     \__regex_if_in_class_or_catcode:TF
5883     { \__regex_compile_raw_error:N u #1 #2 }
5884     {
5885       \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_raw:N r

```

```

5886     { \__regex_compile_u_brace:NNN \__regex_compile_ur_end: }
5887     { \__regex_compile_u_brace:NNN \__regex_compile_u_end: #1 #2 }
5888   }
5889 }

```

(End definition for `\__regex_compile_u::.`)

`\__regex_compile_u_brace:NNN` This enforces the presence of a left brace, then starts a loop to find the variable name.

```

5890 \cs_new:Npn \__regex_compile_u_brace:NNN #1#2#3
5891 {
5892   \__regex_two_if_eq:NNNNTF #2 #3 \__regex_compile_special:N \c_left_brace_str
5893   {
5894     \tl_set:Nn \l__regex_internal_b_tl {#1}
5895     \__kernel_tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
5896     \__regex_compile_u_loop:NN
5897   }
5898   {
5899     \msg_error:nn { regex } { u-missing-lbrace }
5900     \token_if_eq_meaning:NNTF #1 \__regex_compile_ur_end:
5901     { \__regex_compile_raw:N u \__regex_compile_raw:N r }
5902     { \__regex_compile_raw:N u }
5903     #2 #3
5904   }
5905 }

```

(End definition for `\__regex_compile_u_brace:NNN.`)

`\__regex_compile_u_loop:NN` We collect the characters for the argument of \u within an x-expanding assignment. In principle we could just wait to encounter a right brace, but this is unsafe: if the right brace was missing, then we would reach the end-markers of the regex, and continue, leading to obscure fatal errors. Instead, we only allow raw and special characters, and stop when encountering a special right brace, any escaped character, or the end-marker.

```

5906 \cs_new:Npn \__regex_compile_u_loop:NN #1#2
5907 {
5908   \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
5909   { #2 \__regex_compile_u_loop:NN }
5910   {
5911     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
5912     {
5913       \exp_after:wN \token_if_eq_charcode:NNTF \c_right_brace_str #2
5914       { \if_false: { \fi: } \l__regex_internal_b_tl }
5915       {
5916         \if_charcode:w \c_left_brace_str #2
5917         \msg_expandable_error:nnn { regex } { cu-lbrace } { u }
5918       \else:
5919         #2
5920       \fi:
5921       \__regex_compile_u_loop:NN
5922     }
5923   }
5924   {
5925     \if_false: { \fi: }
5926     \msg_error:nnx { regex } { u-missing-rbrace } {#2}
5927     \l__regex_internal_b_tl

```

```

5928         #1 #2
5929     }
5930   }
5931 }
```

(End definition for `\_regex_compile_u_loop:NN`.)

`\_regex_compile_ur_end:` For the `\ur{...}` construction, once we have extracted the variable's name, we replace all groups by non-capturing groups in the compiled regex (passed as the argument of `\_regex_compile_ur:n`). If that has a single branch (namely `\tl_if_empty:oTF` is false) and there is no quantifier, then simply insert the contents of this branch (obtained by `\use_i:nn`, which is expanded later). In all other cases, insert a non-capturing group and look for quantifiers to determine the number of repetition etc.

```

5932 \cs_new_protected:Npn \_regex_compile_ur_end:
5933 {
5934   \group_begin:
5935     \cs_set:Npn \_regex_group:nnN { \_regex_group_no_capture:nnN }
5936     \cs_set:Npn \_regex_group_resetting:nnN { \_regex_group_no_capture:nnN }
5937     \exp_args:NNx
5938   \group_end:
5939     \_regex_compile_ur:n { \use:c { \l__regex_internal_a_tl } }
5940 }
5941 \cs_new_protected:Npn \_regex_compile_ur:n #
5942 {
5943   \tl_if_empty:oTF { \_regex_compile_ur_aux:w #1 {} ? ? \q__regex_nil }
5944   { \_regex_compile_if_quantifier:TFw }
5945   { \use_i:nn }
5946   {
5947     \tl_build_put_right:Nn \l__regex_build_tl
5948       { \_regex_group_no_capture:nnN { \if_false: } \fi: #1 }
5949     \_regex_compile_quantifier:w
5950   }
5951   { \tl_build_put_right:Nn \l__regex_build_tl { \use_i:nn #1 } }
5952 }
5953 \cs_new:Npn \_regex_compile_ur_aux:w \_regex_branch:n #1#2#3 \q__regex_nil {#2}
```

(End definition for `\_regex_compile_ur_end:`, `\_regex_compile_ur:n`, and `\_regex_compile_ur_aux:w`.)

`\_regex_compile_u_end:` Once we have extracted the variable's name, we check for quantifiers, in which case we set up a non-capturing group with a single branch. Inside this branch (we omit it and the group if there is no quantifier), `\_regex_compile_u_payload:` puts the right tests corresponding to the contents of the variable, which we store in `\l__regex_internal_a_tl`. The behaviour of `\u` then depends on whether we are within a `\c{...}` escape (in this case, the variable is turned to a string), or not.

```

5954 \cs_new_protected:Npn \_regex_compile_u_end:
5955 {
5956   \_regex_compile_if_quantifier:TFw
5957   {
5958     \tl_build_put_right:Nn \l__regex_build_tl
5959       {
5960         \_regex_group_no_capture:nnN { \if_false: } \fi:
5961         \_regex_branch:n { \if_false: } \fi:
5962       }
5963 }
```

```

5963     \_\_regex\_compile\_u\_payload:
5964     \tl_build_put_right:Nn \l\_\_regex\_build\_tl { \if_false: { \fi: } }
5965     \_\_regex\_compile\_quantifier:w
5966   }
5967   { \_\_regex\_compile\_u\_payload: }
5968 }
5969 \cs_new_protected:Npn \_\_regex\_compile\_u\_payload:
5970 {
5971   \tl_set:Nv \l\_\_regex\_internal_a_tl { \l\_\_regex\_internal_a_tl }
5972   \if_int_compare:w \l\_\_regex\_mode_int = \c\_\_regex\_outer\_mode\_int
5973     \_\_regex\_compile\_u\_not\_cs:
5974   \else:
5975     \_\_regex\_compile\_u\_in\_cs:
5976   \fi:
5977 }

```

(End definition for `\_\_regex\_compile\_u\_end:` and `\_\_regex\_compile\_u\_payload::`)

`\_\_regex\_compile\_u\_in\_cs:` When `\u` appears within a control sequence, we convert the variable to a string with escaped spaces. Then for each character insert a class matching exactly that character, once.

```

5978 \cs_new_protected:Npn \_\_regex_compile_u_in_cs:
5979 {
5980   \_\_kernel_tl_gset:Nx \g\_\_regex_internal_tl
5981   {
5982     \exp_args:No \_\_kernel_str_to_other_fast:n
5983     { \l\_\_regex_internal_a_tl }
5984   }
5985   \tl_build_put_right:Nx \l\_\_regex_build_tl
5986   {
5987     \tl_map_function:NN \g\_\_regex_internal_tl
5988     \_\_regex_compile_u_in_cs_aux:n
5989   }
5990 }
5991 \cs_new:Npn \_\_regex_compile_u_in_cs_aux:n #1
5992 {
5993   \_\_regex_class:NnnnN \c_true_bool
5994   { \_\_regex_item_caseful_equal:n { \int_value:w '#1 } }
5995   { 1 } { 0 } \c_false_bool
5996 }

```

(End definition for `\_\_regex\_compile\_u\_in\_cs::`)

`\_\_regex\_compile\_u\_not\_cs:` In mode 0, the `\u` escape adds one state to the NFA for each token in `\l\_\_regex\_internal_a_tl`. If a given `<token>` is a control sequence, then insert a string comparison test, otherwise, `\_\_regex\_item\_exact:nn` which compares catcode and character code.

```

5997 \cs_new_protected:Npn \_\_regex_compile_u_not_cs:
5998 {
5999   \tl_analysis_map_inline:Nn \l\_\_regex_internal_a_tl
6000   {
6001     \tl_build_put_right:Nx \l\_\_regex_build_tl
6002     {
6003       \_\_regex_class:NnnnN \c_true_bool
6004     }

```

```

6005     \if_int_compare:w ##3 = \c_zero_int
6006         \_regex_item_exact_cs:n
6007             { \exp_after:wN \cs_to_str:N ##1 }
6008     \else:
6009         \_regex_item_exact:nn { \int_value:w ##3 } { ##2 }
6010     \fi:
6011     }
6012     { 1 } { 0 } \c_false_bool
6013 }
6014 }
6015 }
```

(End definition for `\_regex_compile_u_not_cs:..`)

#### 45.3.13 Other

`\_regex_compile_K:`

The `\K` control sequence is currently the only “command”, which performs some action, rather than matching something. It is allowed in the same contexts as `\b`. At the compilation stage, we leave it as a single control sequence, defined later.

```

6016 \cs_new_protected:cpn { \_regex_compile_K: }
6017 {
6018     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
6019         { \tl_build_put_right:Nn \l__regex_build_tl { \_regex_command_K: } }
6020         { \_regex_compile_raw_error:N K }
6021 }
```

(End definition for `\_regex_compile_K:..`)

#### 45.3.14 Showing regexes

`\_regex_clean_bool:n`

`\_regex_clean_int:n`

`\_regex_clean_int_aux:N`

`\_regex_clean_regex:n`

`\_regex_clean_regex_loop:w`

`\_regex_clean_branch:n`

`\_regex_clean_branch_loop:n`

`\_regex_clean_assertion:Nn`

`\_regex_clean_class:NnnnN`

`\_regex_clean_group:nnnN`

`\_regex_clean_class:n`

`\_regex_clean_class_loop:nnn`

`\_regex_clean_exact_cs:n`

`\_regex_clean_exact_cs:w`

Before showing a regex we check that it is “clean” in the sense that it has the correct internal structure. We do this (in the implementation of `\regex_show:N` and `\regex_log:N`) by comparing it with a cleaned-up version of the same regex. Along the way we also need similar functions for other types: all `\_regex_clean_<type>:n` functions produce valid `<type>` tokens (bool, explicit integer, etc.) from arbitrary input, and the output coincides with the input if that was valid.

```

6022 \cs_new:Npn \_regex_clean_bool:n #1
6023 {
6024     \tl_if_single:nTF {#1}
6025         { \bool_if:NTF #1 \c_true_bool \c_false_bool }
6026         { \c_true_bool }
6027 }
6028 \cs_new:Npn \_regex_clean_int:n #1
6029 {
6030     \tl_if_head_eq_meaning:nNTF {#1} -
6031         { - \exp_args:No \_regex_clean_int:n { \use_none:n #1 } }
6032         { \int_eval:n { 0 \str_map_function:nN {#1} \_regex_clean_int_aux:N } }
6033 }
6034 \cs_new:Npn \_regex_clean_int_aux:N #1
6035 {
6036     \if_int_compare:w 1 < 1 #1 ~
6037         #1
6038     \else:
```

```

6039      \exp_after:wN \str_map_break:
6040      \fi:
6041    }
6042  \cs_new:Npn \__regex_clean_regex:n #1
6043  {
6044    \__regex_clean_regex_loop:w #1
6045    \__regex_branch:n { \q_recursion_tail } \q_recursion_stop
6046  }
6047  \cs_new:Npn \__regex_clean_regex_loop:w #1 \__regex_branch:n #2
6048  {
6049    \quark_if_recursion_tail_stop:n {#2}
6050    \__regex_branch:n { \__regex_clean_branch:n {#2} }
6051    \__regex_clean_regex_loop:w
6052  }
6053  \cs_new:Npn \__regex_clean_branch:n #1
6054  {
6055    \__regex_clean_branch_loop:n #1
6056    ? ? ? ? ? \prg_break_point:
6057  }
6058  \cs_new:Npn \__regex_clean_branch_loop:n #1
6059  {
6060    \tl_if_single:nF {#1} { \prg_break: }
6061    \token_case_meaning:NnF #1
6062    {
6063      \__regex_command_K: { #1 \__regex_clean_branch_loop:n }
6064      \__regex_assertion:Nn { #1 \__regex_clean_assertion:Nn }
6065      \__regex_class:NnnnN { #1 \__regex_clean_class:NnnnN }
6066      \__regex_group:nnnN { #1 \__regex_clean_group:nnnN }
6067      \__regex_group_no_capture:nnnN { #1 \__regex_clean_group:nnnN }
6068      \__regex_group_resetting:nnnN { #1 \__regex_clean_group:nnnN }
6069    }
6070    { \prg_break: }
6071  }
6072  \cs_new:Npn \__regex_clean_assertion:Nn #1#2
6073  {
6074    \__regex_clean_bool:n {#1}
6075    \tl_if_single:nF {#2} { { \__regex_A_test: } \prg_break: }
6076    \token_case_meaning:NnTF #2
6077    {
6078      \__regex_A_test: { }
6079      \__regex_G_test: { }
6080      \__regex_Z_test: { }
6081      \__regex_b_test: { }
6082    }
6083    { {#2} }
6084    { { \__regex_A_test: } \prg_break: }
6085    \__regex_clean_branch_loop:n
6086  }
6087  \cs_new:Npn \__regex_clean_class:NnnnN #1#2#3#4#5
6088  {
6089    \__regex_clean_bool:n {#1}
6090    { \__regex_clean_class:n {#2} }
6091    { \int_max:nn { 0 } { \__regex_clean_int:n {#3} } }
6092    { \int_max:nn { -1 } { \__regex_clean_int:n {#4} } }

```

```

6093     \_\_regex_clean_bool:n {#5}
6094     \_\_regex_clean_branch_loop:n
6095   }
6096 \cs_new:Npn \_\_regex_clean_group:nnnN #1#2#3#4
6097   {
6098     { \_\_regex_clean_regex:n {#1} }
6099     { \int_max:nn { 0 } { \_\_regex_clean_int:n {#2} } }
6100     { \int_max:nn { -1 } { \_\_regex_clean_int:n {#3} } }
6101     \_\_regex_clean_bool:n {#4}
6102     \_\_regex_clean_branch_loop:n
6103   }
6104 \cs_new:Npn \_\_regex_clean_class:n #1
6105   { \_\_regex_clean_class_loop:nnn #1 ????? \prg_break_point: }

```

When cleaning a class there are many cases, including a dozen or so like `\_\_regex_prop_d:` or `\_\_regex_posix_alpha:`. To avoid listing all of them we allow any command that starts with the 13 characters `\_\_regex_prop_` or `\_\_regex_posix` (handily these have the same length, except for the trailing underscore).

```

6106 \cs_new:Npn \_\_regex_clean_class_loop:nnn #1#2#3
6107   {
6108     \tl_if_single:nF {#1} { \prg_break: }
6109     \token_case_meaning:NnTF #1
6110     {
6111       \_\_regex_item_cs:n { #1 { \_\_regex_clean_regex:n {#2} } }
6112       \_\_regex_item_exact_cs:n { #1 { \_\_regex_clean_exact_cs:n {#2} } }
6113       \_\_regex_item_caseful_equal:n { #1 { \_\_regex_clean_int:n {#2} } }
6114       \_\_regex_item_caseless_equal:n { #1 { \_\_regex_clean_int:n {#2} } }
6115       \_\_regex_item_reverse:n { #1 { \_\_regex_clean_class:n {#2} } }
6116     }
6117   { \_\_regex_clean_class_loop:nnn {#3} }
6118   {
6119     \token_case_meaning:NnTF #1
6120     {
6121       \_\_regex_item_caseful_range:nn { }
6122       \_\_regex_item_caseless_range:nn { }
6123       \_\_regex_item_exact:nn { }
6124     }
6125     {
6126       #1 { \_\_regex_clean_int:n {#2} } { \_\_regex_clean_int:n {#3} }
6127       \_\_regex_clean_class_loop:nnn
6128     }
6129     {
6130       \token_case_meaning:NnTF #1
6131       {
6132         \_\_regex_item_catcode:nT { }
6133         \_\_regex_item_catcode_reverse:nT { }
6134       }
6135       {
6136         #1 { \_\_regex_clean_int:n {#2} } { \_\_regex_clean_class:n {#3} }
6137         \_\_regex_clean_class_loop:nnn
6138       }
6139       {
6140         \exp_args:Nf \str_case:nnTF
6141         {

```

```

6142           \exp_args:Nf \str_range:nnn
6143             { \cs_to_str:N #1 } { 1 } { 13 }
6144         }
6145       {
6146         { __regex_prop_ } { }
6147         { __regex_posix } { }
6148       }
6149     {
6150       #1
6151       \__regex_clean_class_loop:nnn {#2} {#3}
6152     }
6153     { \prg_break: }
6154   }
6155 }
6156 }
6157 }
6158 \cs_new:Npn \__regex_clean_exact_cs:n #1
6159 {
6160   \exp_last_unbraced:Nf \use_none:n
6161   {
6162     \__regex_clean_exact_cs:w #1
6163     \scan_stop: \q_recursion_tail \scan_stop:
6164     \q_recursion_stop
6165   }
6166 }
6167 \cs_new:Npn \__regex_clean_exact_cs:w #1 \scan_stop:
6168 {
6169   \quark_if_recursion_tail_stop:n {#1}
6170   \scan_stop: \tl_to_str:n {#1}
6171   \__regex_clean_exact_cs:w
6172 }

```

(End definition for `\__regex_clean_bool:n` and others.)

`\__regex_show:N` Within a group and within `\tl_build_begin:N ... \tl_build_end:N` we redefine all the function that can appear in a compiled regex, then run the regex. The result stored in `\l__regex_internal_a_tl` is then meant to be shown.

```

6173 \cs_new_protected:Npn \__regex_show:N #1
6174 {
6175   \group_begin:
6176   \tl_build_begin:N \l__regex_build_tl
6177   \cs_set_protected:Npn \__regex_branch:n
6178   {
6179     \seq_pop_right:NN \l__regex_show_prefix_seq
6180     \l__regex_internal_a_tl
6181     \__regex_show_one:n { +-branch }
6182     \seq_put_right:No \l__regex_show_prefix_seq
6183     \l__regex_internal_a_tl
6184   \use:n
6185   }
6186   \cs_set_protected:Npn \__regex_group:nnnN
6187   {
6188     \__regex_show_group_aux:nnnnN { } }
6189   \cs_set_protected:Npn \__regex_group_no_capture:nnnN
6190   {
6191     \__regex_show_group_aux:nnnnN { ~(no-capture) } }

```

```

6190   \cs_set_protected:Npn \__regex_group_resetting:n{nnN}
6191     { \__regex_show_group_aux:n{nnnN} { ~{resetting} } }
6192   \cs_set_eq:NN \__regex_class:NnnN \__regex_show_class:NnnN
6193   \cs_set_protected:Npn \__regex_command_K:
6194     { \__regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
6195   \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
6196     {
6197       \__regex_show_one:n
6198         { \bool_if:NF ##1 { negative~ } assertion:~##2 }
6199     }
6200   \cs_set:Npn \__regex_b_test: { word~boundary }
6201   \cs_set:Npn \__regex_Z_test: { anchor~at~end~(\iow_char:N\\Z) }
6202   \cs_set:Npn \__regex_A_test: { anchor~at~start~(\iow_char:N\\A) }
6203   \cs_set:Npn \__regex_G_test: { anchor~at~start~of~match~(\iow_char:N\\G) }
6204   \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
6205     { \__regex_show_one:n { char~code~\__regex_show_char:n{##1} } }
6206   \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
6207     {
6208       \__regex_show_one:n
6209         { range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}] }
6210     }
6211   \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
6212     { \__regex_show_one:n { char~code~\__regex_show_char:n{##1}~(caseless) } }
6213   \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
6214     {
6215       \__regex_show_one:n
6216         { Range~[\__regex_show_char:n{##1}, \__regex_show_char:n{##2}]~(caseless) }
6217     }
6218   \cs_set_protected:Npn \__regex_item_catcode:nT
6219     { \__regex_show_item_catcode:NnT \c_true_bool }
6220   \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
6221     { \__regex_show_item_catcode:NnT \c_false_bool }
6222   \cs_set_protected:Npn \__regex_item_reverse:n
6223     { \__regex_show_scope:nn { Reversed~match } }
6224   \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
6225     { \__regex_show_one:n { char~\__regex_show_char:n{##2},~catcode~##1 } }
6226   \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
6227   \cs_set_protected:Npn \__regex_item_cs:n
6228     { \__regex_show_scope:nn { control~sequence } }
6229   \cs_set:cpx { __regex_prop_:: } { \__regex_show_one:n { any~token } }
6230   \seq_clear:N \l__regex_show_prefix_seq
6231   \__regex_show_push:n { ~ }
6232   \cs_if_exist_use:N #
6233   \tl_build_end:N \l__regex_build_tl
6234   \exp_args:NNNo
6235   \group_end:
6236     \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
6237   }

```

(End definition for `\__regex_show:N`.)

`\__regex_show_char:n` Show a single character, together with its ascii representation if available. This could be extended to beyond ascii. It is not ideal for parentheses themselves.

```
6238 \cs_new:Npn \__regex_show_char:n #1
```

```

6239  {
6240    \int_eval:n {#1}
6241    \int_compare:nT { 32 <= #1 <= 126 }
6242      { ~ ( \char_generate:nn {#1} {12} ) }
6243  }

```

(End definition for `\_regex_show_char:n`.)

`\_regex_show_one:n` Every part of the final message go through this function, which adds one line to the output, with the appropriate prefix.

```

6244 \cs_new_protected:Npn \_regex_show_one:n #1
6245  {
6246    \int_incr:N \l__regex_show_lines_int
6247    \tl_build_put_right:Nx \l__regex_build_tl
6248      {
6249        \exp_not:N \iow_newline:
6250        \seq_map_function:NN \l__regex_show_prefix_seq \use:n
6251          #1
6252      }
6253  }

```

(End definition for `\_regex_show_one:n`.)

`\_regex_show_push:n` Enter and exit levels of nesting. The `scope` function prints its first argument as an “introduction”, then performs its second argument in a deeper level of nesting.

```

\cs_new_protected:Npn \_regex_show_scope:nn
  {
    \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
  \cs_new_protected:Npn \_regex_show_pop:
  {
    \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
  \cs_new_protected:Npn \_regex_show_scope:nn #1#2
  {
    \_regex_show_one:n {#1}
    \_regex_show_push:n { ~ }
    #2
    \_regex_show_pop:
  }

```

(End definition for `\_regex_show_push:n`, `\_regex_show_pop:`, and `\_regex_show_scope:nn`.)

`\_regex_show_group_aux:nnnn` We display all groups in the same way, simply adding a message, (no capture) or (resetting), to special groups. The odd `\use_ii:nn` avoids printing a spurious +-branch for the first branch.

```

\cs_new_protected:Npn \_regex_show_group_aux:nnnnN #1#2#3#4#5
  {
    \_regex_show_one:n { ,-group~begin #1 }
    \_regex_show_push:n { | }
    \use_ii:nn #2
    \_regex_show_pop:
    \_regex_show_one:n
      { '-group~end \_regex_msg_repeated:nnN {#3} {#4} #5 }
  }

```

(End definition for `\_regex_show_group_aux:nnnn`.)

\\_\\_regex\\_show\\_class:NnnN I'm entirely unhappy about this function: I couldn't find a way to test if a class is a single test. Instead, collect the representation of the tests in the class. If that had more than one line, write Match or Don't match on its own line, with the repeating information if any. Then the various tests on lines of their own, and finally a line. Otherwise, we need to evaluate the representation of the tests again (since the prefix is incorrect). That's clunky, but not too expensive, since it's only one test.

```

6274 \cs_set:Npn \_\_regex_show_class:NnnN #1#2#3#4#5
6275 {
6276     \group_begin:
6277         \tl_build_begin:N \l_\_regex_build_tl
6278         \int_zero:N \l_\_regex_show_lines_int
6279         \_\_regex_show_push:n {~}
6280         #2
6281         \int_compare:nTF { \l_\_regex_show_lines_int = 0 }
6282             {
6283                 \group_end:
6284                 \_\_regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
6285             }
6286             {
6287                 \bool_if:nTF
6288                     { #1 && \int_compare_p:n { \l_\_regex_show_lines_int = 1 } }
6289                     {
6290                         \group_end:
6291                         #2
6292                         \tl_build_put_right:Nn \l_\_regex_build_tl
6293                         { \_\_regex_msg_repeated:nnN {#3} {#4} #5 }
6294                     }
6295                     {
6296                         \tl_build_end:N \l_\_regex_build_tl
6297                         \exp_args:NNNo
6298                         \group_end:
6299                         \tl_set:Nn \l_\_regex_internal_a_tl \l_\_regex_build_tl
6300                         \_\_regex_show_one:n
6301                             {
6302                                 \bool_if:NTF #1 { Match } { Don't~match }
6303                                 \_\_regex_msg_repeated:nnN {#3} {#4} #5
6304                             }
6305                             \tl_build_put_right:Nx \l_\_regex_build_tl
6306                             { \exp_not:o \l_\_regex_internal_a_tl }
6307                         }
6308                     }
6309                 }

```

(End definition for \\_\\_regex\\_show\\_class:NnnN.)

\\_\\_regex\\_show\\_item catcode:NnT Produce a sequence of categories which the catcode bitmap #2 contains, and show it, indenting the tests on which this catcode constraint applies.

```

6310 \cs_new_protected:Npn \_\_regex_show_item_catcode:NnT #1#2
6311 {
6312     \seq_set_split:Nnn \l_\_regex_internal_seq { } { CBEMTPUDSLOA }
6313     \seq_set_filter:NNn \l_\_regex_internal_seq \l_\_regex_internal_seq
6314         { \int_if_odd_p:n {#2} / \int_use:c { c_\_regex_catcode_##1_int } } }
6315     \_\_regex_show_scope:nn
6316         {

```

```

6317      categories~
6318      \seq_map_function:NN \l__regex_internal_seq \use:n
6319      ,
6320      \bool_if:NF #1 { negative~ } class
6321    }
6322  }

(End definition for \__regex_show_item_catcode:NnT.)
```

```

\__regex_show_item_exact_cs:n
6323 \cs_new_protected:Npn \__regex_show_item_exact_cs:n #1
6324 {
6325   \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
6326   \seq_set_map_x>NNn \l__regex_internal_seq
6327     \l__regex_internal_seq { \iow_char:N\##1 }
6328   \__regex_show_one:n
6329   { control-sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
6330 }
```

(End definition for \\_\_regex\_show\_item\_exact\_cs:n.)

## 45.4 Building

### 45.4.1 Variables used while building

\l\_\_regex\_min\_state\_int  
\l\_\_regex\_max\_state\_int

The last state that was allocated is \l\_\_regex\_max\_state\_int - 1, so that \l\_\_regex\_max\_state\_int always points to a free state. The min\_state variable is 1 to begin with, but gets shifted in nested calls to the matching code, namely in \c{...} constructions.

```

6331 \int_new:N \l__regex_min_state_int
6332 \int_set:Nn \l__regex_min_state_int { 1 }
6333 \int_new:N \l__regex_max_state_int
```

(End definition for \l\_\_regex\_min\_state\_int and \l\_\_regex\_max\_state\_int.)

\l\_\_regex\_left\_state\_int  
\l\_\_regex\_right\_state\_int  
\l\_\_regex\_left\_state\_seq  
\l\_\_regex\_right\_state\_seq

Alternatives are implemented by branching from a left state into the various choices, then merging those into a right state. We store information about those states in two sequences. Those states are also used to implement group quantifiers. Most often, the left and right pointers only differ by 1.

```

6334 \int_new:N \l__regex_left_state_int
6335 \int_new:N \l__regex_right_state_int
6336 \seq_new:N \l__regex_left_state_seq
6337 \seq_new:N \l__regex_right_state_seq
```

(End definition for \l\_\_regex\_left\_state\_int and others.)

\l\_\_regex\_capturing\_group\_int

\l\_\_regex\_capturing\_group\_int is the next ID number to be assigned to a capturing group. This starts at 0 for the group enclosing the full regular expression, and groups are counted in the order of their left parenthesis, except when encountering resetting groups.

```

6338 \int_new:N \l__regex_capturing_group_int
```

(End definition for \l\_\_regex\_capturing\_group\_int.)

#### 45.4.2 Framework

This phase is about going from a compiled regex to an NFA. Each state of the NFA is stored in a `\toks`. The operations which can appear in the `\toks` are

- `\__regex_action_start_wildcard:N <boolean>` inserted at the start of the regular expression, where a `true <boolean>` makes it unanchored.
- `\__regex_action_success`: marks the exit state of the NFA.
- `\__regex_action_cost:n {<shift>}` is a transition from the current `<state>` to `<state> + <shift>`, which consumes the current character: the target state is saved and will be considered again when matching at the next position.
- `\__regex_action_free:n {<shift>}`, and `\__regex_action_free_group:n {<shift>}` are free transitions, which immediately perform the actions for the state `<state> + <shift>` of the NFA. They differ in how they detect and avoid infinite loops. For now, we just need to know that the `group` variant must be used for transitions back to the start of a group.
- `\__regex_action_submatch:nN {<group>} <key>` where the `<key>` is `<` or `>` for the beginning or end of group numbered `<group>`. This causes the current position in the query to be stored as the `<key>` submatch boundary.
- One of these actions, within a conditional.

We strive to preserve the following properties while building.

- The current capturing group is `capturing_group - 1`, and if a group opened now it would be labelled `capturing_group`.
- The last allocated state is `max_state - 1`, so `max_state` is a free state.
- The `left_state` points to a state to the left of the current group or of the last class.
- The `right_state` points to a newly created, empty state, with some transitions leading to it.
- The `left/right` sequences hold a list of the corresponding end-points of nested groups.

`\__regex_build:n` The `n`-type function first compiles its argument. Reset some variables. Allocate two states, and put a wildcard in state 0 (transitions to state 1 and 0 state). Then build the regex within a (capturing) group numbered 0 (current value of `capturing_group`). Finally, if the match reaches the last state, it is successful. A `false` boolean for argument `#1` for the auxiliaries will suppress the wildcard and make the match anchored: used for `\peek_regex:nTF` and similar.

```

6339 \cs_new_protected:Npn \__regex_build:n
6340   { \__regex_build_aux:Nn \c_true_bool }
6341 \cs_new_protected:Npn \__regex_build:N
6342   { \__regex_build_aux:NN \c_true_bool }
6343 \cs_new_protected:Npn \__regex_build_aux:Nn #1#2
6344   {
6345     \__regex_compile:n {#2}

```

```

6346     \_\_regex\_build\_aux:NN #1 \l\_\_regex\_internal\_regex
6347 }
6348 \cs_new_protected:Npn \_\_regex_build_aux:NN #1#2
6349 {
6350     \_\_regex_standard_escapechar:
6351     \int_zero:N \l\_\_regex_capturing_group_int
6352     \int_set_eq:NN \l\_\_regex_max_state_int \l\_\_regex_min_state_int
6353     \_\_regex_build_new_state:
6354     \_\_regex_build_new_state:
6355     \_\_regex_toks_put_right:Nn \l\_\_regex_left_state_int
6356         { \_\_regex_action_start_wildcard:N #1 }
6357     \_\_regex_group:nnN {#2} { 1 } { 0 } \c_false_bool
6358     \_\_regex_toks_put_right:Nn \l\_\_regex_right_state_int
6359         { \_\_regex_action_success: }
6360 }

```

(End definition for `\_\_regex_build:n` and others.)

`\g\_\_regex_case_int` Case number that was successfully matched in `\regex_match_case:nn` and related functions.

```
6361 \int_new:N \g\_\_regex_case_int
```

(End definition for `\g\_\_regex_case_int`.)

`\l\_\_regex_case_max_group_int` The largest group number appearing in any of the *<regex>* in the argument of `\regex_match_case:nn` and related functions.

```
6362 \int_new:N \l\_\_regex_case_max_group_int
```

(End definition for `\l\_\_regex_case_max_group_int`.)

`\_\_regex_case_build:n` See `\_\_regex_build:n`, but with a loop.

```

6363 \cs_new_protected:Npn \_\_regex_case_build:n #1
6364 {
6365     \_\_regex_case_build_aux:Nn \c_true_bool {#1}
6366     \int_gzero:N \g\_\_regex_case_int
6367 }
6368 \cs_generate_variant:Nn \_\_regex_case_build:n { x }
6369 \cs_new_protected:Npn \_\_regex_case_build_aux:Nn #1#2
6370 {
6371     \_\_regex_standard_escapechar:
6372     \int_set_eq:NN \l\_\_regex_max_state_int \l\_\_regex_min_state_int
6373     \_\_regex_build_new_state:
6374     \_\_regex_build_new_state:
6375     \_\_regex_toks_put_right:Nn \l\_\_regex_left_state_int
6376         { \_\_regex_action_start_wildcard:N #1 }
6377 %
6378     \_\_regex_build_new_state:
6379     \_\_regex_toks_put_left:Nx \l\_\_regex_left_state_int
6380         { \_\_regex_action_submatch:nN { 0 } < }
6381     \_\_regex_push_lr_states:
6382     \int_zero:N \l\_\_regex_case_max_group_int
6383     \int_gzero:N \g\_\_regex_case_int
6384     \tl_map_inline:nn {#2}
6385     {
6386         \int_gincr:N \g\_\_regex_case_int

```

```

6387         \_\_regex\_case\_build\_loop:n {##1}
6388     }
6389     \int_set_eq:NN \l\_\_regex\_capturing\_group\_int \l\_\_regex\_case\_max\_group\_int
6390     \_\_regex\_pop\_lr\_states:
6391   }
6392 \cs_new_protected:Npn \_\_regex\_case\_build\_loop:n #1
6393   {
6394     \int_set:Nn \l\_\_regex\_capturing\_group\_int { 1 }
6395     \_\_regex\_compile\_use:n {#1}
6396     \int_set:Nn \l\_\_regex\_case\_max\_group\_int
6397     {
6398       \int_max:nn { \l\_\_regex\_case\_max\_group\_int }
6399       { \l\_\_regex\_capturing\_group\_int }
6400     }
6401     \seq_pop:NN \l\_\_regex\_right\_state\_seq \l\_\_regex\_internal\_a\_t1
6402     \int_set:Nn \l\_\_regex\_right\_state\_int \l\_\_regex\_internal\_a\_t1
6403     \_\_regex\_toks\_put\_left:Nx \l\_\_regex\_right\_state\_int
6404     {
6405       \_\_regex\_action\_submatch:nN { 0 } >
6406       \int_gset:Nn \g\_\_regex\_case\_int
6407       { \int_use:N \g\_\_regex\_case\_int }
6408       \_\_regex\_action\_success:
6409     }
6410     \_\_regex\_toks\_clear:N \l\_\_regex\_max\_state\_int
6411     \seq_push:No \l\_\_regex\_right\_state\_seq
6412     { \int_use:N \l\_\_regex\_max\_state\_int }
6413     \int_incr:N \l\_\_regex\_max\_state\_int
6414   }

```

(End definition for `\_\_regex\_case\_build:n`, `\_\_regex\_case\_build\_aux:Nn`, and `\_\_regex\_case\_build\_loop:n`.)

`\_\_regex\_build\_for\_cs:n` The matching code relies on some global intarray variables, but only uses a range of their entries. Specifically,

- `\g\_\_regex\_state\_active\_intarray` from `\l\_\_regex\_min\_state\_int` to `\l\_\_regex\_max\_state\_int`;

Here, in this nested call to the matching code, we need the new versions of this range to involve completely new entries of the intarray variables, so we begin by setting (the new) `\l\_\_regex\_min\_state\_int` to (the old) `\l\_\_regex\_max\_state\_int` to use higher entries.

When using a regex to match a cs, we don't insert a wildcard, we anchor at the end, and since we ignore submatches, there is no need to surround the expression with a group. However, for branches to work properly at the outer level, we need to put the appropriate `left` and `right` states in their sequence.

```

6415 \cs_new_protected:Npn \_\_regex\_build\_for\_cs:n #1
6416   {
6417     \int_set_eq:NN \l\_\_regex\_min\_state\_int \l\_\_regex\_max\_state\_int
6418     \_\_regex\_build\_new\_state:
6419     \_\_regex\_build\_new\_state:
6420     \_\_regex\_push\_lr\_states:
6421     #1
6422     \_\_regex\_pop\_lr\_states:

```

```

6423     \_\_regex_toks\_put\_right:Nn \l\_\_regex\_right\_state\_int
6424     {
6425         \if\_int\_compare:w -2 = \l\_\_regex\_curr\_char\_int
6426             \exp\_after:wn \_\_regex\_action\_success:
6427         \fi:
6428     }
6429 }
```

(End definition for `\_\_regex\_build\_for\_cs:n`.)

#### 45.4.3 Helpers for building an nfa

`\_\_regex\_push\_lr\_states:` When building the regular expression, we keep track of pointers to the left-end and right-end of each group without help from TeX's grouping.

```

6430 \cs\_new\_protected:Npn \_\_regex\_push\_lr\_states:
6431 {
6432     \seq\_push:No \l\_\_regex\_left\_state\_seq
6433     { \int\_use:N \l\_\_regex\_left\_state\_int }
6434     \seq\_push:No \l\_\_regex\_right\_state\_seq
6435     { \int\_use:N \l\_\_regex\_right\_state\_int }
6436 }
6437 \cs\_new\_protected:Npn \_\_regex\_pop\_lr\_states:
6438 {
6439     \seq\_pop:NN \l\_\_regex\_left\_state\_seq \l\_\_regex\_internal\_a\_tl
6440     \int\_set:Nn \l\_\_regex\_left\_state\_int \l\_\_regex\_internal\_a\_tl
6441     \seq\_pop:NN \l\_\_regex\_right\_state\_seq \l\_\_regex\_internal\_a\_tl
6442     \int\_set:Nn \l\_\_regex\_right\_state\_int \l\_\_regex\_internal\_a\_tl
6443 }
```

(End definition for `\_\_regex\_push\_lr\_states:` and `\_\_regex\_pop\_lr\_states::`)

`\_\_regex\_build\_transition\_left:NNN` Add a transition from #2 to #3 using the function #1. The `left` function is used for higher priority transitions, and the `right` function for lower priority transitions (which should be performed later). The signatures differ to reflect the differing usage later on. Both functions could be optimized.

```

6444 \cs\_new\_protected:Npn \_\_regex\_build\_transition\_left:NNN #1#2#3
6445 { \_\_regex\_toks\_put\_left:Nx #2 { #1 { \int\_eval:n { #3 - #2 } } } }
6446 \cs\_new\_protected:Npn \_\_regex\_build\_transition\_right:nNn #1#2#3
6447 { \_\_regex\_toks\_put\_right:Nx #2 { #1 { \int\_eval:n { #3 - #2 } } } }
```

(End definition for `\_\_regex\_build\_transition\_left:NNN` and `\_\_regex\_build\_transition\_right:nNn`.)

`\_\_regex\_build\_new\_state:` Add a new empty state to the NFA. Then update the `left`, `right`, and `max` states, so that the `right` state is the new empty state, and the `left` state points to the previously “current” state.

```

6448 \cs\_new\_protected:Npn \_\_regex\_build\_new\_state:
6449 {
6450     \_\_regex\_toks\_clear:N \l\_\_regex\_max\_state\_int
6451     \int\_set\_eq:NN \l\_\_regex\_left\_state\_int \l\_\_regex\_right\_state\_int
6452     \int\_set\_eq:NN \l\_\_regex\_right\_state\_int \l\_\_regex\_max\_state\_int
6453     \int\_incr:N \l\_\_regex\_max\_state\_int
6454 }
```

(End definition for `\_\_regex\_build\_new\_state::`)

```
\_\_regex\_build\_transitions_laziness:NNNNN
```

This function creates a new state, and puts two transitions starting from the old current state. The order of the transitions is controlled by #1, true for lazy quantifiers, and false for greedy quantifiers.

```
6455 \cs_new_protected:Npn \_\_regex_build_transitions_laziness:NNNNN #1#2#3#4#5
6456 {
6457     \_\_regex_build_new_state:
6458     \_\_regex_toks_put_right:Nx \l\_\_regex_left_state_int
6459     {
6460         \if_meaning:w \c_true_bool #1
6461             #2 { \int_eval:n { #3 - \l\_\_regex_left_state_int } }
6462             #4 { \int_eval:n { #5 - \l\_\_regex_left_state_int } }
6463         \else:
6464             #4 { \int_eval:n { #5 - \l\_\_regex_left_state_int } }
6465             #2 { \int_eval:n { #3 - \l\_\_regex_left_state_int } }
6466         \fi:
6467     }
6468 }
```

(End definition for `\_\_regex_build_transitions_laziness:NNNNN`.)

#### 45.4.4 Building classes

```
\_\_regex_class:NnnnN
```

The arguments are: `<boolean> {<tests>} {<min>} {<more>} {<laziness>}`. First store the tests with a trailing `\_\_regex_action_cost:n`, in the true branch of `\_\_regex_break_point:TF` for positive classes, or the false branch for negative classes. The integer `<more>` is 0 for fixed repetitions, -1 for unbounded repetitions, and `<max> - <min>` for a range of repetitions.

```
6469 \cs_new_protected:Npn \_\_regex_class:NnnnN #1#2#3#4#5
6470 {
6471     \cs_set:Npx \_\_regex_tests_action_cost:n ##1
6472     {
6473         \exp_not:n { \exp_not:n {#2} }
6474         \bool_if:NTF #1
6475             { \_\_regex_break_point:TF { \_\_regex_action_cost:n {##1} } { } }
6476             { \_\_regex_break_point:TF { } { \_\_regex_action_cost:n {##1} } }
6477     }
6478     \if_case:w - #4 \exp_stop_f:
6479         \_\_regex_class_repeat:n {#3}
6480     \or: \_\_regex_class_repeat:nN {#3} #5
6481     \else: \_\_regex_class_repeat:nnN {#3} {#4} #5
6482     \fi:
6483 }
6484 \cs_new:Npn \_\_regex_tests_action_cost:n { \_\_regex_action_cost:n }
```

(End definition for `\_\_regex_class:NnnnN` and `\_\_regex_tests_action_cost:n`.)

```
\_\_regex_class_repeat:n
```

This is used for a fixed number of repetitions. Build one state for each repetition, with a transition controlled by the tests that we have collected. That works just fine for `#1 = 0` repetitions: nothing is built.

```
6485 \cs_new_protected:Npn \_\_regex_class_repeat:n #1
6486 {
6487     \prg_replicate:nn {#1}
6488 }
```

```

6489     \_\_regex\_build\_new\_state:
6490     \_\_regex\_build\_transition\_right:nNn \_\_regex\_tests\_action\_cost:n
6491         \l\_\_regex\_left\_state\_int \l\_\_regex\_right\_state\_int
6492     }
6493 }
```

(End definition for `\_\_regex\_class\_repeat:n`.)

`\_\_regex\_class\_repeat:nN`

This implements unbounded repetitions of a single class (*e.g.* the `*` and `+` quantifiers). If the minimum number #1 of repetitions is 0, then build a transition from the current state to itself governed by the tests, and a free transition to a new state (hence skipping the tests). Otherwise, call `\_\_regex\_class\_repeat:n` for the code to match #1 repetitions, and add free transitions from the last state to the previous one, and to a new one. In both cases, the order of transitions is controlled by the lazyness boolean #2.

```

6494 \cs_new_protected:Npn \_\_regex\_class\_repeat:nN #1#2
6495 {
6496     \if_int_compare:w #1 = \c_zero_int
6497         \_\_regex\_build\_transitions\_lazyness:NNNNN #2
6498             \_\_regex\_action\_free:n \l\_\_regex\_right\_state\_int
6499             \_\_regex\_tests\_action\_cost:n \l\_\_regex\_left\_state\_int
6500     \else:
6501         \_\_regex\_class\_repeat:n {#1}
6502             \int_set_eq:NN \l\_\_regex\_internal_a_int \l\_\_regex\_left\_state\_int
6503             \_\_regex\_build\_transitions\_lazyness:NNNNN #2
6504                 \_\_regex\_action\_free:n \l\_\_regex\_right\_state\_int
6505                 \_\_regex\_action\_free:n \l\_\_regex\_internal_a_int
6506     \fi:
6507 }
```

(End definition for `\_\_regex\_class\_repeat:nN`.)

`\_\_regex\_class\_repeat:nnN`

We want to build the code to match from #1 to #1 + #2 repetitions. Match #1 repetitions (can be 0). Compute the final state of the next construction as `a`. Build #2 > 0 states, each with a transition to the next state governed by the tests, and a transition to the final state `a`. The computation of `a` is safe because states are allocated in order, starting from `max_state`.

```

6508 \cs_new_protected:Npn \_\_regex\_class\_repeat:nnN #1#2#3
6509 {
6510     \_\_regex\_class\_repeat:n {#1}
6511     \int_set:Nn \l\_\_regex\_internal_a_int
6512         { \l\_\_regex\_max\_state\_int + #2 - 1 }
6513     \prg_replicate:nn { #2 }
6514     {
6515         \_\_regex\_build\_transitions\_lazyness:NNNNN #3
6516             \_\_regex\_action\_free:n \l\_\_regex\_internal_a_int
6517             \_\_regex\_tests\_action\_cost:n \l\_\_regex\_right\_state\_int
6518     }
6519 }
```

(End definition for `\_\_regex\_class\_repeat:nnN`.)

#### 45.4.5 Building groups

\\_\\_regex\\_group\\_aux:nnnnN

Arguments: {*label*} {*contents*} {*min*} {*more*} {*lazyness*}. If *min* is 0, we need to add a state before building the group, so that the thread which skips the group does not also set the start-point of the submatch. After adding one more state, the **left\_state** is the left end of the group, from which all branches stem, and the **right\_state** is the right end of the group, and all branches end their course in that state. We store those two integers to be queried for each branch, we build the NFA states for the contents #2 of the group, and we forget about the two integers. Once this is done, perform the repetition: either exactly #3 times, or #3 or more times, or between #3 and #3 + #4 times, with lazyness #5. The *label* #1 is used for submatch tracking. Each of the three auxiliaries expects **left\_state** and **right\_state** to be set properly.

```

6520 \cs_new_protected:Npn \_\_regex_group_aux:nnnnN #1#2#3#4#5
6521   {
6522     \if_int_compare:w #3 = \c_zero_int
6523       \_\_regex_build_new_state:
6524       \_\_regex_build_transition_right:nNn \_\_regex_action_free_group:n
6525         \l_\_regex_left_state_int \l_\_regex_right_state_int
6526     \fi:
6527     \_\_regex_build_new_state:
6528     \_\_regex_push_lr_states:
6529       #2
6530     \_\_regex_pop_lr_states:
6531     \if_case:w - #4 \exp_stop_f:
6532       \_\_regex_group_repeat:nn {#1} {#3}
6533     \or: \_\_regex_group_repeat:nnN {#1} {#3}          #5
6534     \else: \_\_regex_group_repeat:nnnN {#1} {#3} {#4} {#5}
6535   \fi:
6536 }

```

(End definition for \\_\\_regex\_group\_aux:nnnnN.)

\\_\\_regex\_group:nnnN  
\\_\\_regex\_group\_no\_capture:nnnN

Hand to \\_\\_regex\_group\_aux:nnnnN the label of that group (expanded), and the group itself, with some extra commands to perform.

```

6537 \cs_new_protected:Npn \_\_regex_group:nnnN #
6538   {
6539     \exp_args:No \_\_regex_group_aux:nnnnN
6540       { \int_use:N \l_\_regex_capturing_group_int }
6541       {
6542         \int_incr:N \l_\_regex_capturing_group_int
6543           #1
6544       }
6545   }
6546 \cs_new_protected:Npn \_\_regex_group_no_capture:nnnN
6547   { \_\_regex_group_aux:nnnnN { -1 } }

```

(End definition for \\_\\_regex\_group:nnnN and \\_\\_regex\_group\_no\_capture:nnnN.)

\\_\\_regex\_group\_resetting:nnnN  
\\_\\_regex\_group\_resetting\_loop:nnnN

Again, hand the label -1 to \\_\\_regex\_group\_aux:nnnnN, but this time we work a little bit harder to keep track of the maximum group label at the end of any branch, and to reset the group number at each branch. This relies on the fact that a compiled regex always is a sequence of items of the form \\_\\_regex\_branch:n {*branch*}.

```
6548 \cs_new_protected:Npn \_\_regex_group_resetting:nnnN #1
```

```

6549  {
6550    \_\_regex\_group\_aux:nnnnN { -1 }
6551    {
6552      \exp\_args:Noo \_\_regex\_group\_resetting\_loop:nnNn
6553      { \int\_use:N \l\_\_regex\_capturing\_group\_int }
6554      { \int\_use:N \l\_\_regex\_capturing\_group\_int }
6555      #1
6556      { ?? \prg\_break:n } { }
6557      \prg\_break\_point:
6558    }
6559  }
6560 \cs\_new\_protected:Npn \_\_regex\_group\_resetting\_loop:nnNn #1#2#3#4
6561  {
6562    \use\_none:nn #3 { \int\_set:Nn \l\_\_regex\_capturing\_group\_int {#1} }
6563    \int\_set:Nn \l\_\_regex\_capturing\_group\_int {#2}
6564    #3 {#4}
6565    \exp\_args:Nf \_\_regex\_group\_resetting\_loop:nnNn
6566    { \int\_max:nn {#1} { \l\_\_regex\_capturing\_group\_int } }
6567    {#2}
6568  }

```

(End definition for `\_\_regex\_group\_resetting:nnN` and `\_\_regex\_group\_resetting\_loop:nnNn`.)

`\_\_regex\_branch:n` Add a free transition from the left state of the current group to a brand new state, starting point of this branch. Once the branch is built, add a transition from its last state to the right state of the group. The left and right states of the group are extracted from the relevant sequences.

```

6569 \cs\_new\_protected:Npn \_\_regex\_branch:n #1
6570  {
6571    \_\_regex\_build\_new\_state:
6572    \seq\_get:NN \l\_\_regex\_left\_state\_seq \l\_\_regex\_internal\_a\_tl
6573    \int\_set:Nn \l\_\_regex\_left\_state\_int \l\_\_regex\_internal\_a\_tl
6574    \_\_regex\_build\_transition\_right:nNn \_\_regex\_action\_free:
6575    \l\_\_regex\_left\_state\_int \l\_\_regex\_right\_state\_int
6576    #1
6577    \seq\_get:NN \l\_\_regex\_right\_state\_seq \l\_\_regex\_internal\_a\_tl
6578    \_\_regex\_build\_transition\_right:nNn \_\_regex\_action\_free:
6579    \l\_\_regex\_right\_state\_int \l\_\_regex\_internal\_a\_tl
6580  }

```

(End definition for `\_\_regex\_branch:n`.)

`\_\_regex\_group\_repeat:nn` This function is called to repeat a group a fixed number of times `#2`; if this is 0 we remove the group altogether (but don't reset the `capturing_group` label). Otherwise, the auxiliary `\_\_regex\_group\_repeat\_aux:n` copies `#2` times the `\toks` for the group, and leaves `internal_a` pointing to the left end of the last repetition. We only record the submatch information at the last repetition. Finally, add a state at the end (the transition to it has been taken care of by the replicating auxiliary).

```

6581 \cs\_new\_protected:Npn \_\_regex\_group\_repeat:nn #1#2
6582  {
6583    \if\_int\_compare:w #2 = \c\_zero\_int
6584    \int\_set:Nn \l\_\_regex\_max\_state\_int
6585    { \l\_\_regex\_left\_state\_int - 1 }
6586    \_\_regex\_build\_new\_state:

```

```

6587     \else:
6588         \_\_regex\_group\_repeat\_aux:n {#2}
6589         \_\_regex\_group\_submatches:nNN {#1}
6590         \l\_\_regex\_internal\_a\_int \l\_\_regex\_right\_state\_int
6591         \_\_regex\_build\_new\_state:
6592     \fi:
6593 }

```

(End definition for `\_\_regex\_group\_repeat:nn`.)

`\_\_regex\_group\_submatches:nNN`

This inserts in states #2 and #3 the code for tracking submatches of the group #1, unless inhibited by a label of -1.

```

6594 \cs_new_protected:Npn \_\_regex_group_submatches:nNN #1#2#3
6595 {
6596     \if_int_compare:w #1 > - \c_one_int
6597         \_\_regex_toks_put_left:Nx #2 { \_\_regex_action_submatch:nN {#1} < }
6598         \_\_regex_toks_put_left:Nx #3 { \_\_regex_action_submatch:nN {#1} > }
6599     \fi:
6600 }

```

(End definition for `\_\_regex_group_submatches:nNN`.)

`\_\_regex_group_repeat_aux:n`

Here we repeat `\toks` ranging from `left_state` to `max_state`,  $#1 > 0$  times. First add a transition so that the copies “chain” properly. Compute the shift `c` between the original copy and the last copy we want. Shift the `right_state` and `max_state` to their final values. We then want to perform `c` copy operations. At the end, `b` is equal to the `max_state`, and `a` points to the left of the last copy of the group.

```

6601 \cs_new_protected:Npn \_\_regex_group_repeat_aux:n #1
6602 {
6603     \_\_regex_build_transition_right:nNn \_\_regex_action_free:n
6604         \l\_\_regex_right_state_int \l\_\_regex_max_state_int
6605         \int_set_eq:NN \l\_\_regex_internal_a_int \l\_\_regex_left_state_int
6606         \int_set_eq:NN \l\_\_regex_internal_b_int \l\_\_regex_max_state_int
6607         \if_int_compare:w \int_eval:n {#1} > \c_one_int
6608             \int_set:Nn \l\_\_regex_internal_c_int
6609             {
6610                 ( #1 - 1 )
6611                 * ( \l\_\_regex_internal_b_int - \l\_\_regex_internal_a_int )
6612             }
6613             \int_add:Nn \l\_\_regex_right_state_int { \l\_\_regex_internal_c_int }
6614             \int_add:Nn \l\_\_regex_max_state_int { \l\_\_regex_internal_c_int }
6615             \_\_regex_toks_memcpy:NNn
6616                 \l\_\_regex_internal_b_int
6617                 \l\_\_regex_internal_a_int
6618                 \l\_\_regex_internal_c_int
6619         \fi:
6620 }

```

(End definition for `\_\_regex_group_repeat_aux:n`.)

`\_\_regex_group_repeat:nnN`

This function is called to repeat a group at least  $n$  times; the case  $n = 0$  is very different from  $n > 0$ . Assume first that  $n = 0$ . Insert submatch tracking information at the start and end of the group, add a free transition from the right end to the “true” left state `a`.

(remember: in this case we had added an extra state before the left state). This forms the loop, which we break away from by adding a free transition from `a` to a new state.

Now consider the case  $n > 0$ . Repeat the group  $n$  times, chaining various copies with a free transition. Add submatch tracking only to the last copy, then add a free transition from the right end back to the left end of the last copy, either before or after the transition to move on towards the rest of the NFA. This transition can end up before submatch tracking, but that is irrelevant since it only does so when going again through the group, recording new matches. Finally, add a state; we already have a transition pointing to it from `\_regex_group_repeat_aux:n`.

```

6621 \cs_new_protected:Npn \_regex_group_repeat:nnN #1#2#3
6622 {
6623   \if_int_compare:w #2 = \c_zero_int
6624     \_regex_group_submatches:nNN {#1}
6625       \l__regex_left_state_int \l__regex_right_state_int
6626       \int_set:Nn \l__regex_internal_a_int
6627         { \l__regex_left_state_int - 1 }
6628       \_regex_build_transition_right:nNn \_regex_action_free:n
6629         \l__regex_right_state_int \l__regex_internal_a_int
6630       \_regex_build_new_state:
6631     \if_meaning:w \c_true_bool #3
6632       \_regex_build_transition_left:NNN \_regex_action_free:n
6633         \l__regex_internal_a_int \l__regex_right_state_int
6634     \else:
6635       \_regex_build_transition_right:nNn \_regex_action_free:n
6636         \l__regex_internal_a_int \l__regex_right_state_int
6637     \fi:
6638   \else:
6639     \_regex_group_repeat_aux:n {#2}
6640     \_regex_group_submatches:nNN {#1}
6641       \l__regex_internal_a_int \l__regex_right_state_int
6642     \if_meaning:w \c_true_bool #3
6643       \_regex_build_transition_right:nNn \_regex_action_free_group:n
6644         \l__regex_right_state_int \l__regex_internal_a_int
6645     \else:
6646       \_regex_build_transition_left:NNN \_regex_action_free_group:n
6647         \l__regex_right_state_int \l__regex_internal_a_int
6648     \fi:
6649     \_regex_build_new_state:
6650   \fi:
6651 }
```

(End definition for `\_regex_group_repeat:nnN`.)

`\_regex_group_repeat:nnnN`

We wish to repeat the group between `#2` and  $\#2 + \#3$  times, with a laziness controlled by `#4`. We insert submatch tracking up front: in principle, we could avoid recording submatches for the first `#2` copies of the group, but that forces us to treat specially the case  $\#2 = 0$ . Repeat that group with submatch tracking  $\#2 + \#3$  times (the maximum number of repetitions). Then our goal is to add `#3` transitions from the end of the `#2`-th group, and each subsequent groups, to the end. For a lazy quantifier, we add those transitions to the left states, before submatch tracking. For the greedy case, we add the transitions to the right states, after submatch tracking and the transitions which go on with more repetitions. In the greedy case with  $\#2 = 0$ , the transition which skips over all

copies of the group must be added separately, because its starting state does not follow the normal pattern: we had to add it “by hand” earlier.

```

6652 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
6653 {
6654     \__regex_group_submatches:nNN {#1}
6655     \l__regex_left_state_int \l__regex_right_state_int
6656     \__regex_group_repeat_aux:n { #2 + #3 }
6657     \if_meaning:w \c_true_bool #4
6658         \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
6659         \prg_replicate:nn { #3 }
6660         {
6661             \int_sub:Nn \l__regex_left_state_int
6662                 { \l__regex_internal_b_int - \l__regex_internal_a_int }
6663             \__regex_build_transition_left:NNN \__regex_action_free:n
6664                 \l__regex_left_state_int \l__regex_max_state_int
6665         }
6666     \else:
6667         \prg_replicate:nn { #3 - 1 }
6668         {
6669             \int_sub:Nn \l__regex_right_state_int
6670                 { \l__regex_internal_b_int - \l__regex_internal_a_int }
6671             \__regex_build_transition_right:nNn \__regex_action_free:n
6672                 \l__regex_right_state_int \l__regex_max_state_int
6673         }
6674     \if_int_compare:w #2 = \c_zero_int
6675         \int_set:Nn \l__regex_right_state_int
6676             { \l__regex_left_state_int - 1 }
6677     \else:
6678         \int_sub:Nn \l__regex_right_state_int
6679             { \l__regex_internal_b_int - \l__regex_internal_a_int }
6680     \fi:
6681     \__regex_build_transition_right:nNn \__regex_action_free:n
6682         \l__regex_right_state_int \l__regex_max_state_int
6683     \fi:
6684     \__regex_build_new_state:
6685 }
```

(End definition for `\__regex_group_repeat:nnnN`.)

#### 45.4.6 Others

`\__regex_assertion:Nn` Usage: `\__regex_assertion:Nn ⟨boolean⟩ {⟨test⟩}`, where the `⟨test⟩` is either of the two other functions. Add a free transition to a new state, conditionally to the assertion test. The `\__regex_b_test:` test is used by the `\b` and `\B` escape: check if the last character was a word character or not, and do the same to the current character. The boundary-markers of the string are non-word characters for this purpose.

```

6686 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
6687 {
6688     \__regex_build_new_state:
6689     \__regex_toks_put_right:Nx \l__regex_left_state_int
6690     {
6691         \exp_not:n {#2}
6692         \__regex_break_point:TF
```

```

6693     \bool_if:NF #1 { { } }
6694     {
6695         \__regex_action_free:n
6696         {
6697             \int_eval:n
6698             { \l__regex_right_state_int - \l__regex_left_state_int }
6699         }
6700     }
6701     \bool_if:NT #1 { { } }
6702 }
6703 }
6704 \cs_new_protected:Npn \__regex_b_test:
6705 {
6706     \group_begin:
6707     \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
6708     \__regex_prop_w:
6709     \__regex_break_point:TF
6710     { \group_end: \__regex_item_reverse:n { \__regex_prop_w: } }
6711     { \group_end: \__regex_prop_w: }
6712 }
6713 \cs_new_protected:Npn \__regex_Z_test:
6714 {
6715     \if_int_compare:w -2 = \l__regex_curr_char_int
6716     \exp_after:wN \__regex_break_true:w
6717     \fi:
6718 }
6719 \cs_new_protected:Npn \__regex_A_test:
6720 {
6721     \if_int_compare:w -2 = \l__regex_last_char_int
6722     \exp_after:wN \__regex_break_true:w
6723     \fi:
6724 }
6725 \cs_new_protected:Npn \__regex_G_test:
6726 {
6727     \if_int_compare:w \l__regex_curr_pos_int = \l__regex_start_pos_int
6728     \exp_after:wN \__regex_break_true:w
6729     \fi:
6730 }

```

(End definition for `\__regex_assertion:Nn` and others.)

`\__regex_command_K:` Change the starting point of the 0-th submatch (full match), and transition to a new state, pretending that this is a fresh thread.

```

6731 \cs_new_protected:Npn \__regex_command_K:
6732 {
6733     \__regex_build_new_state:
6734     \__regex_toks_put_right:Nx \l__regex_left_state_int
6735     {
6736         \__regex_action_submatch:nN { 0 } <
6737         \bool_set_true:N \l__regex_fresh_thread_bool
6738         \__regex_action_free:n
6739         {
6740             \int_eval:n
6741             { \l__regex_right_state_int - \l__regex_left_state_int }

```

```

6742         }
6743     \bool_set_false:N \l__regex_fresh_thread_bool
6744 }
6745 }

(End definition for \__regex_command_K::)

```

## 45.5 Matching

We search for matches by running all the execution threads through the NFA in parallel, reading one token of the query at each step. The NFA contains “free” transitions to other states, and transitions which “consume” the current token. For free transitions, the instruction at the new state of the NFA is performed immediately. When a transition consumes a character, the new state is appended to a list of “active states”, stored in `\g__regex_thread_info_intarray` (together with submatch information): this thread is made active again when the next token is read from the query. At every step (for each token in the query), we unpack that list of active states and the corresponding submatch props, and empty those.

If two paths through the NFA “collide” in the sense that they reach the same state after reading a given token, then they only differ in how they previously matched, and any future execution would be identical for both. (Note that this would be wrong in the presence of back-references.) Hence, we only need to keep one of the two threads: the thread with the highest priority. Our NFA is built in such a way that higher priority actions always come before lower priority actions, which makes things work.

The explanation in the previous paragraph may make us think that we simply need to keep track of which states were visited at a given step: after all, the loop generated when matching `(a?)*` against `a` is broken, isn’t it? No. The group first matches `a`, as it should, then repeats; it attempts to match `a` again but fails; it skips `a`, and finds out that this state has already been seen at this position in the query: the match stops. The capturing group is (wrongly) `a`. What went wrong is that a thread collided with itself, and the later version, which has gone through the group one more times with an empty match, should have a higher priority than not going through the group.

We solve this by distinguishing “normal” free transitions `\__regex_action_free:n` from transitions `\__regex_action_free_group:n` which go back to the start of the group. The former keeps threads unless they have been visited by a “completed” thread, while the latter kind of transition also prevents going back to a state visited by the current thread.

### 45.5.1 Variables used when matching

`\l__regex_min_pos_int`  
`\l__regex_max_pos_int`  
`\l__regex_curr_pos_int`  
`\l__regex_start_pos_int`  
`\l__regex_success_pos_int`

The tokens in the query are indexed from `min_pos` for the first to `max_pos - 1` for the last, and their information is stored in several arrays and `\toks` registers with those numbers. We match without backtracking, keeping all threads in lockstep at the `curr_pos` in the query. The starting point of the current match attempt is `start_pos`, and `success_pos`, updated whenever a thread succeeds, is used as the next starting position.

```

6746 \int_new:N \l__regex_min_pos_int
6747 \int_new:N \l__regex_max_pos_int
6748 \int_new:N \l__regex_curr_pos_int
6749 \int_new:N \l__regex_start_pos_int
6750 \int_new:N \l__regex_success_pos_int

```

(End definition for `\l__regex_min_pos_int` and others.)

```
\l__regex_curr_char_int
\l__regex_curr_catcode_int
\l__regex_curr_token_tl
\l__regex_last_char_int
\l__regex_last_char_success_int
\l__regex_case_changed_char_int
```

The character and category codes of the token at the current position and a token list expanding to that token; the character code of the token at the previous position; the character code of the token just before a successful match; and the character code of the result of changing the case of the current token ( $A-Z \leftrightarrow a-z$ ). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `curr_char` variable is also used in various other phases to hold a character code.

```
6751 \int_new:N \l__regex_curr_char_int
6752 \int_new:N \l__regex_curr_catcode_int
6753 \tl_new:N \l__regex_curr_token_tl
6754 \int_new:N \l__regex_last_char_int
6755 \int_new:N \l__regex_last_char_success_int
6756 \int_new:N \l__regex_case_changed_char_int
```

(End definition for `\l__regex_curr_char_int` and others.)

```
\l__regex_curr_state_int
```

For every character in the token list, each of the active states is considered in turn. The variable `\l__regex_curr_state_int` holds the state of the NFA which is currently considered: transitions are then given as shifts relative to the current state.

```
6757 \int_new:N \l__regex_curr_state_int
```

(End definition for `\l__regex_curr_state_int`.)

```
\l__regex_curr_submatches_tl
\l__regex_success_submatches_tl
```

The submatches for the thread which is currently active are stored in the `curr_submatches` list, which is almost a comma list, but ends with a comma. This list is stored by `\__regex_store_state:n` into an intarray variable, to be retrieved when matching at the next position. When a thread succeeds, this list is copied to `\l__regex_success_submatches_tl`: only the last successful thread remains there.

```
6758 \tl_new:N \l__regex_curr_submatches_tl
6759 \tl_new:N \l__regex_success_submatches_tl
```

(End definition for `\l__regex_curr_submatches_tl` and `\l__regex_success_submatches_tl`.)

```
\l__regex_step_int
```

This integer, always even, is increased every time a character in the query is read, and not reset when doing multiple matches. We store in `\g__regex_state_active_intarray` the last step in which each `\langle state \rangle` in the NFA was encountered. This lets us break infinite loops by not visiting the same state twice in the same step. In fact, the step we store is equal to `step` when we have started performing the operations of `\toks(state)`, but not finished yet. However, once we finish, we store `step + 1` in `\g__regex_state_active_intarray`. This is needed to track submatches properly (see building phase). The `step` is also used to attach each set of submatch information to a given iteration (and automatically discard it when it corresponds to a past step).

```
6760 \int_new:N \l__regex_step_int
```

(End definition for `\l__regex_step_int`.)

```
\l__regex_min_thread_int
\l__regex_max_thread_int
```

All the currently active threads are kept in order of precedence in `\g__regex_thread_info_intarray` together with the corresponding submatch information. Data in this intarray is organized as blocks from `min_thread` (included) to `max_thread` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_thread` is reset to `min_thread`, effectively clearing the array.

```
6761 \int_new:N \l__regex_min_thread_int
6762 \int_new:N \l__regex_max_thread_int
```

(End definition for `\l_regex_min_thread_int` and `\l_regex_max_thread_int`.)

`\g_regex_state_active_intarray`  
`\g_regex_thread_info_intarray`

`\g_regex_state_active_intarray` stores the last `<step>` in which each `<state>` was active. `\g_regex_thread_info_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
6763 \intarray_new:Nn \g_regex_state_active_intarray { 65536 }
6764 \intarray_new:Nn \g_regex_thread_info_intarray { 65536 }
```

(End definition for `\g_regex_state_active_intarray` and `\g_regex_thread_info_intarray`.)

`\l_regex_matched_analysis_tl`  
`\l_regex_curr_analysis_tl`

The list `\l_regex_curr_analysis_tl` consists of a brace group containing three brace groups corresponding to the current token, with the same syntax as `\tl_analysis_map_inline:nn`. The list `\l_regex_matched_analysis_tl` (constructed under the `tl-build` machinery) has one item for each token that has already been treated so far in a given match attempt: each item consists of three brace groups with the same syntax as `\tl_analysis_map_inline:nn`.

```
6765 \tl_new:N \l_regex_matched_analysis_tl
6766 \tl_new:N \l_regex_curr_analysis_tl
```

(End definition for `\l_regex_matched_analysis_tl` and `\l_regex_curr_analysis_tl`.)

`\l_regex_every_match_tl`

Every time a match is found, this token list is used. For single matching, the token list is empty. For multiple matching, the token list is set to repeat the matching, after performing some operation which depends on the user function. See `\_regex_single_match:` and `\_regex_multi_match:n`.

```
6767 \tl_new:N \l_regex_every_match_tl
```

(End definition for `\l_regex_every_match_tl`.)

`\l_regex_fresh_thread_bool`  
`\l_regex_empty_success_bool`  
`\_regex_if_two_empty_matches:F`

When doing multiple matches, we need to avoid infinite loops where each iteration matches the same empty token list. When an empty token list is matched, the next successful match of the same empty token list is suppressed. We detect empty matches by setting `\l_regex_fresh_thread_bool` to `true` for threads which directly come from the start of the regex or from the `\K` command, and testing that boolean whenever a thread succeeds. The function `\_regex_if_two_empty_matches:F` is redefined at every match attempt, depending on whether the previous match was empty or not: if it was, then the function must cancel a purported success if it is empty and at the same spot as the previous match; otherwise, we definitely don't have two identical empty matches, so the function is `\use:n`.

```
6768 \bool_new:N \l_regex_fresh_thread_bool
6769 \bool_new:N \l_regex_empty_success_bool
6770 \cs_new_eq:NN \_regex_if_two_empty_matches:F \use:n
```

(End definition for `\l_regex_fresh_thread_bool`, `\l_regex_empty_success_bool`, and `\_regex_if_two_empty_matches:F`.)

`\g_regex_success_bool`  
`\l_regex_saved_success_bool`  
`\l_regex_match_success_bool`

The boolean `\l_regex_match_success_bool` is true if the current match attempt was successful, and `\g_regex_success_bool` is true if there was at least one successful match. This is the only global variable in this whole module, but we would need it to be local when matching a control sequence with `\c{...}`. This is done by saving the global variable into `\l_regex_saved_success_bool`, which is local, hence not affected by the changes due to inner regex functions.

```
6771 \bool_new:N \g_regex_success_bool
6772 \bool_new:N \l_regex_saved_success_bool
6773 \bool_new:N \l_regex_match_success_bool
```

(End definition for `\g__regex_success_bool`, `\l__regex_saved_success_bool`, and `\l__regex_match-success_bool`.)

#### 45.5.2 Matching: framework

`\__regex_match:n` Initialize the variables that should be set once for each user function (even for multiple matches). Namely, the overall matching is not yet successful; none of the states should be marked as visited (`\g__regex_state_active_intarray`), and we start at step 0; we pretend that there was a previous match ending at the start of the query, which was not empty (to avoid smothering an empty match at the start). Once all this is set up, we are ready for the ride. Find the first match.

```

6774 \cs_new_protected:Npn \__regex_match:n #1
6775   {
6776     \__regex_match_init:
6777     \__regex_match_once_init:
6778     \tl_analysis_map_inline:nn {#1}
6779     { \__regex_match_one_token:nnN {##1} {##2} ##3 }
6780     \__regex_match_one_token:nnN { } { -2 } F
6781     \prg_break_point:Nn \__regex_maplike_break: { }
6782   }
6783 \cs_new_protected:Npn \__regex_match_cs:n #1
6784   {
6785     \int_set_eq:NN \l__regex_min_thread_int \l__regex_max_thread_int
6786     \__regex_match_init:
6787     \__regex_match_once_init:
6788     \str_map_inline:nn {#1}
6789     {
6790       \tl_if_blank:nTF {##1}
6791       { \__regex_match_one_token:nnN {##1} {'##1} A }
6792       { \__regex_match_one_token:nnN {##1} {'##1} C }
6793     }
6794     \__regex_match_one_token:nnN { } { -2 } F
6795     \prg_break_point:Nn \__regex_maplike_break: { }
6796   }
6797 \cs_new_protected:Npn \__regex_match_init:
6798   {
6799     \bool_gset_false:N \g__regex_success_bool
6800     \int_step_inline:nnn
6801       \l__regex_min_state_int { \l__regex_max_state_int - 1 }
6802     {
6803       \__kernel_intarray_gset:Nnn
6804         \g__regex_state_active_intarray {##1} { 1 }
6805     }
6806     \int_zero:N \l__regex_step_int
6807     \int_set:Nn \l__regex_min_pos_int { 2 }
6808     \int_set_eq:NN \l__regex_success_pos_int \l__regex_min_pos_int
6809     \int_set:Nn \l__regex_last_char_success_int { -2 }
6810     \tl_build_begin:N \l__regex_matched_analysis_tl
6811     \tl_clear:N \l__regex_curr_analysis_tl
6812     \int_set:Nn \l__regex_min_submatch_int { 1 }
6813     \int_set_eq:NN \l__regex_submatch_int \l__regex_min_submatch_int
6814     \bool_set_false:N \l__regex_empty_success_bool
6815   }

```

(End definition for `\_regex_match:n`, `\_regex_match_cs:n`, and `\_regex_match_init:..`)

`\_regex_match_once_init:` This function resets various variables used when finding one match. It is called before the loop through characters, and every time we find a match, before searching for another match (this is controlled by the `every_match` token list).

First initialize some variables: set the conditional which detects identical empty matches; this match attempt starts at the previous `success_pos`, is not yet successful, and has no submatches yet; clear the array of active threads, and put the starting state 0 in it. We are then almost ready to read our first token in the query, but we actually start one position earlier than the start because `\_regex_match_one_token:nnN` increments `\l_regex_curr_pos_int` and saves `\l_regex_curr_char_int` as the `last_char` so that word boundaries can be correctly identified.

```
6816 \cs_new_protected:Npn \_regex_match_once_init:
6817 {
6818     \if_meaning:w \c_true_bool \l_regex_empty_success_bool
6819         \cs_set:Npn \_regex_if_two_empty_matches:F
6820             {
6821                 \int_compare:nNnF
6822                     \l_regex_start_pos_int = \l_regex_curr_pos_int
6823             }
6824     \else:
6825         \cs_set_eq:NN \_regex_if_two_empty_matches:F \use:n
6826     \fi:
6827     \int_set_eq:NN \l_regex_start_pos_int \l_regex_success_pos_int
6828     \bool_set_false:N \l_regex_match_success_bool
6829     \tl_set:Nx \l_regex_curr_submatches_tl
6830         { \prg_replicate:nn { 2 * \l_regex_capturing_group_int } { 0 , } }
6831     \int_set_eq:NN \l_regex_max_thread_int \l_regex_min_thread_int
6832     \_regex_store_state:n { \l_regex_min_state_int }
6833     \int_set:Nn \l_regex_curr_pos_int
6834         { \l_regex_start_pos_int - 1 }
6835     \int_set_eq:NN \l_regex_curr_char_int \l_regex_last_char_success_int
6836     \tl_build_get:NN \l_regex_matched_analysis_tl \l_regex_internal_a_tl
6837     \exp_args:NNf \_regex_match_once_init_aux:
6838     \tl_map_inline:nn
6839         { \exp_after:wN \l_regex_internal_a_tl \l_regex_curr_analysis_tl }
6840             { \_regex_match_one_token:nnN ##1 }
6841     \prg_break_point:Nn \_regex_maplike_break: { }
6842 }
6843 \cs_new_protected:Npn \_regex_match_once_init_aux:
6844 {
6845     \tl_build_clear:N \l_regex_matched_analysis_tl
6846     \tl_clear:N \l_regex_curr_analysis_tl
6847 }
```

(End definition for `\_regex_match_once_init:..`)

`\_regex_single_match:` For a single match, the overall success is determined by whether the only match attempt is a success. When doing multiple matches, the overall matching is successful as soon as any match succeeds. Perform the action #1, then find the next match.

```
6848 \cs_new_protected:Npn \_regex_single_match:
6849 {
6850     \tl_set:Nn \l_regex_every_match_tl
```

```

6851   {
6852     \bool_gset_eq:NN
6853       \g__regex_success_bool
6854       \l__regex_match_success_bool
6855       \__regex_maplike_break:
6856   }
6857 }
6858 \cs_new_protected:Npn \__regex_multi_match:n #1
6859 {
6860   \tl_set:Nn \l__regex_every_match_tl
6861   {
6862     \if_meaning:w \c_false_bool \l__regex_match_success_bool
6863       \exp_after:wN \__regex_maplike_break:
6864     \fi:
6865     \bool_gset_true:N \g__regex_success_bool
6866     #1
6867     \__regex_match_once_init:
6868   }
6869 }

```

(End definition for `\__regex_single_match:` and `\__regex_multi_match:n`.)

`\__regex_match_one_token:nnN`  
`\__regex_match_one_active:n`

At each new position, set some variables and get the new character and category from the query. Then unpack the array of active threads, and clear it by resetting its length (`max_thread`). This results in a sequence of `\__regex_use_state_and_submatches:w`  $\langle state \rangle, \langle submatch-clist \rangle$ ; and we consider those states one by one in order. As soon as a thread succeeds, exit the step, and, if there are threads to consider at the next position, and we have not reached the end of the string, repeat the loop. Otherwise, the last thread that succeeded is the match. We explain the `fresh_thread` business when describing `\__regex_action_wildcard::`.

```

6870 \cs_new_protected:Npn \__regex_match_one_token:nnN #1#2#3
6871 {
6872   \int_add:Nn \l__regex_step_int { 2 }
6873   \int_incr:N \l__regex_curr_pos_int
6874   \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
6875   \cs_set_eq:NN \__regex_maybe_compute_ccc: \__regex_compute_case_changed_char:
6876   \tl_set:Nn \l__regex_curr_token_tl {#1}
6877   \int_set:Nn \l__regex_curr_char_int {#2}
6878   \int_set:Nn \l__regex_curr_catcode_int { "#3 }
6879   \tl_build_put_right:Nx \l__regex_matched_analysis_tl
6880     { \exp_not:o \l__regex_curr_analysis_tl }
6881   \tl_set:Nn \l__regex_curr_analysis_tl { { {#1} {#2} #3 } }
6882   \use:x
6883   {
6884     \int_set_eq:NN \l__regex_max_thread_int \l__regex_min_thread_int
6885     \int_step_function:nnN
6886       { \l__regex_min_thread_int }
6887       { \l__regex_max_thread_int - 1 }
6888       \__regex_match_one_active:n
6889   }
6890   \prg_break_point:
6891   \bool_set_false:N \l__regex_fresh_thread_bool
6892   \if_int_compare:w \l__regex_max_thread_int > \l__regex_min_thread_int
6893     \if_int_compare:w -2 < \l__regex_curr_char_int

```

```

6894           \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
6895           \fi:
6896           \fi:
6897           \l__regex_every_match_tl
6898       }
6899   \cs_new:Npn \__regex_match_one_active:n #1
6900   {
6901     \__regex_use_state_and_submatches:w
6902     \__kernel_intarray_range_to_clist:Nnn
6903     \g__regex_thread_info_intarray
6904     { 1 + #1 * (\l__regex_capturing_group_int * 2 + 1) }
6905     { (1 + #1) * (\l__regex_capturing_group_int * 2 + 1) }
6906     ;
6907   }

```

(End definition for `\__regex_match_one_token:nnN` and `\__regex_match_one_active:n`.)

#### 45.5.3 Using states of the nfa

`\__regex_use_state:`

Use the current NFA instruction. The state is initially marked as belonging to the current `step`: this allows normal free transition to repeat, but group-repeating transitions won't. Once we are done exploring all the branches it spawned, the state is marked as `step + 1`: any thread hitting it at that point will be terminated.

```

6908   \cs_new_protected:Npn \__regex_use_state:
6909   {
6910     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6911     { \l__regex_curr_state_int } { \l__regex_step_int }
6912     \__regex_toks_use:w \l__regex_curr_state_int
6913     \__kernel_intarray_gset:Nnn \g__regex_state_active_intarray
6914     { \l__regex_curr_state_int }
6915     { \int_eval:n { \l__regex_step_int + 1 } }
6916   }

```

(End definition for `\__regex_use_state:..`)

`\__regex_use_state_and_submatches:w`

This function is called as one item in the array of active threads after that array has been unpacked for a new step. Update the `curr_state` and `curr_submatches` and use the state if it has not yet been encountered at this step.

```

6917   \cs_new_protected:Npn \__regex_use_state_and_submatches:w #1 , #2 ;
6918   {
6919     \int_set:Nn \l__regex_curr_state_int {#1}
6920     \if_int_compare:w
6921       \__kernel_intarray_item:Nn \g__regex_state_active_intarray
6922       { \l__regex_curr_state_int }
6923       < \l__regex_step_int
6924     \tl_set:Nn \l__regex_curr_submatches_tl { #2 , }
6925     \exp_after:wN \__regex_use_state:
6926   \fi:
6927   \scan_stop:
6928 }

```

(End definition for `\__regex_use_state_and_submatches:w`.)

#### 45.5.4 Actions when matching

\\_regex\_action\_start\_wildcard:N

For an unanchored match, state 0 has a free transition to the next and a costly one to itself, to repeat at the next position. To catch repeated identical empty matches, we need to know if a successful thread corresponds to an empty match. The instruction resetting \l\_\_regex\_fresh\_thread\_bool may be skipped by a successful thread, hence we had to add it to \\_regex\_match\_one\_token:nnN too.

```

6929 \cs_new_protected:Npn \_regex_action_start_wildcard:N #1
6930 {
6931     \bool_set_true:N \l__regex_fresh_thread_bool
6932     \_regex_action_free:n {1}
6933     \bool_set_false:N \l__regex_fresh_thread_bool
6934     \bool_if:NT #1 { \_regex_action_cost:n {0} }
6935 }
```

(End definition for \\_regex\_action\_start\_wildcard:N.)

\\_regex\_action\_free:n

\\_regex\_action\_free\_group:n

\\_regex\_action\_free\_aux:nn

These functions copy a thread after checking that the NFA state has not already been used at this position. If not, store submatches in the new state, and insert the instructions for that state in the input stream. Then restore the old value of \l\_\_regex\_curr\_state\_int and of the current submatches. The two types of free transitions differ by how they test that the state has not been encountered yet: the group version is stricter, and will not use a state if it was used earlier in the current thread, hence forcefully breaking the loop, while the “normal” version will revisit a state even within the thread itself.

```

6936 \cs_new_protected:Npn \_regex_action_free:
6937 {
6938     \_regex_action_free_aux:nn { > \l__regex_step_int \else: } }
6939 \cs_new_protected:Npn \_regex_action_free_group:n
6940 {
6941     \_regex_action_free_aux:nn { < \l__regex_step_int } }
6942 \cs_new_protected:Npn \_regex_action_free_aux:nn #1#2
6943 {
6944     \use:x
6945     {
6946         \int_add:Nn \l__regex_curr_state_int {#2}
6947         \exp_not:n
6948         {
6949             \if_int_compare:w
6950                 \l__kernel_intarray_item:Nn \g__regex_state_active_intarray
6951                 { \l__regex_curr_state_int }
6952                 #1
6953                 \exp_after:wN \_regex_use_state:
6954                 \fi:
6955             }
6956             \int_set:Nn \l__regex_curr_state_int
6957             { \int_use:N \l__regex_curr_state_int }
6958             \tl_set:Nn \exp_not:N \l__regex_curr_submatches_tl
6959             { \exp_not:o \l__regex_curr_submatches_tl }
6960 }
```

(End definition for \\_regex\_action\_free:n, \\_regex\_action\_free\_group:n, and \\_regex\_action\_free\_aux:nn.)

\\_regex\_action\_cost:n

A transition which consumes the current character and shifts the state by #1. The resulting state is stored in the appropriate array for use at the next position, and we also store the current submatches.

```

6960 \cs_new_protected:Npn \__regex_action_cost:n #1
6961 {
6962     \exp_args:Nx \__regex_store_state:n
6963     { \int_eval:n { \l__regex_curr_state_int + #1 } }
6964 }

```

(End definition for `\__regex_action_cost:n`.)

`\__regex_store_state:n`  
`\__regex_store_submatches:` Put the given state and current submatch information in `\g__regex_thread_info_intarray`, and increment the length of the array.

```

6965 \cs_new_protected:Npn \__regex_store_state:n #1
6966 {
6967     \exp_args:No \__regex_store_submatches:nn
6968     \l__regex_curr_submatches_tl {#1}
6969     \int_incr:N \l__regex_max_thread_int
6970 }
6971 \cs_new_protected:Npn \__regex_store_submatches:nn #1#2
6972 {
6973     \__kernel_intarray_gset_range_from_clist:Nnn
6974     \g__regex_thread_info_intarray
6975     {
6976         \__regex_int_eval:w
6977         1 + \l__regex_max_thread_int *
6978         (\l__regex_capturing_group_int * 2 + 1)
6979     }
6980     { #2 , #1 }
6981 }

```

(End definition for `\__regex_store_state:n` and `\__regex_store_submatches:..`)

`\__regex_disable_submatches:` Some user functions don't require tracking submatches. We get a performance improvement by simply defining the relevant functions to remove their argument and do nothing with it.

```

6982 \cs_new_protected:Npn \__regex_disable_submatches:
6983 {
6984     \cs_set_protected:Npn \__regex_store_submatches:n ##1 { }
6985     \cs_set_protected:Npn \__regex_action_submatch:nN ##1##2 { }
6986 }

```

(End definition for `\__regex_disable_submatches:..`)

`\__regex_action_submatch:nN`  
`\__regex_action_submatch_aux:w` Update the current submatches with the information from the current position. Maybe a bottleneck.

```

6987 \cs_new_protected:Npn \__regex_action_submatch:nN #1#2
6988 {
6989     \exp_after:wN \__regex_action_submatch_aux:w
6990     \l__regex_curr_submatches_tl ; {#1} #2
6991 }
6992 \cs_new_protected:Npn \__regex_action_submatch_aux:w #1 ; #2#3
6993 {
6994     \tl_set:Nx \l__regex_curr_submatches_tl
6995     {
6996         \prg_replicate:nn
6997             { #2 \if_meaning:w > #3 + \l__regex_capturing_group_int \fi: }

```

```

6998      { \__regex_action_submatch_auxii:w }
6999      \__regex_action_submatch_auxiii:w
7000      #1
7001      }
7002  }
7003 \cs_new:Npn \__regex_action_submatch_auxii:w
7004     #1 \__regex_action_submatch_auxiii:w #2 ,
7005     { #2 , #1 \__regex_action_submatch_auxiii:w }
7006 \cs_new:Npn \__regex_action_submatch_auxiii:w #1 ,
7007     { \int_use:N \l__regex_curr_pos_int , }

```

(End definition for `\__regex_action_submatch:nN` and others.)

`\__regex_action_success:`: There is a successful match when an execution path reaches the last state in the NFA, unless this marks a second identical empty match. Then mark that there was a successful match; it is empty if it is “fresh”; and we store the current position and submatches. The current step is then interrupted with `\prg_break:`, and only paths with higher precedence are pursued further. The values stored here may be overwritten by a later success of a path with higher precedence.

```

7008 \cs_new_protected:Npn \__regex_action_success:
7009  {
7010      \__regex_if_two_empty_matches:F
7011      {
7012          \bool_set_true:N \l__regex_match_success_bool
7013          \bool_set_eq:NN \l__regex_empty_success_bool
7014              \l__regex_fresh_thread_bool
7015          \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
7016          \int_set_eq:NN \l__regex_last_char_success_int \l__regex_last_char_int
7017          \tl_build_clear:N \l__regex_matched_analysis_tl
7018          \tl_set_eq:NN \l__regex_success_submatches_tl
7019              \l__regex_curr_submatches_tl
7020          \prg_break:
7021      }
7022  }

```

(End definition for `\__regex_action_success:..`)

## 45.6 Replacement

### 45.6.1 Variables and helpers used in replacement

`\l__regex_replacement_csnames_int`: The behaviour of closing braces inside a replacement text depends on whether a sequences `\c{` or `\u{`} has been encountered. The number of “open” such sequences that should be closed by `}` is stored in `\l__regex_replacement_csnames_int`, and decreased by 1 by each `..`.

```
7023 \int_new:N \l__regex_replacement_csnames_int
```

(End definition for `\l__regex_replacement_csnames_int`.)

`\l__regex_replacement_category_tl` `\l__regex_replacement_category_seq`: This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(_))d`.

```

7024 \tl_new:N \l__regex_replacement_category_tl
7025 \seq_new:N \l__regex_replacement_category_seq

```

(End definition for `\l_regex_replacement_category_tl` and `\l_regex_replacement_category_seq`.)

`\g__regex_balance_tl` This token list holds the replacement text for `\__regex_replacement_balance_one_match:n` while it is being built incrementally.

```
7026 \tl_new:N \g__regex_balance_tl
```

(End definition for `\g__regex_balance_tl`.)

`\__regex_replacement_balance_one_match:n` This expects as an argument the first index of a set of entries in `\g__regex_submatch_begin_intarray` (and related arrays) which hold the submatch information for a given match. It can be used within an integer expression to obtain the brace balance incurred by performing the replacement on that match. This combines the braces lost by removing the match, braces added by all the submatches appearing in the replacement, and braces appearing explicitly in the replacement. Even though it is always redefined before use, we initialize it as for an empty replacement. An important property is that concatenating several calls to that function must result in a valid integer expression (hence a leading + in the actual definition).

```
7027 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
  { - \__regex_submatch_balance:n {#1} }
```

(End definition for `\__regex_replacement_balance_one_match:n`.)

`\__regex_replacement_do_one_match:n` The input is the same as `\__regex_replacement_balance_one_match:n`. This function is redefined to expand to the part of the token list from the end of the previous match to a given match, followed by the replacement text. Hence concatenating the result of this function with all possible arguments (one call for each match), as well as the range from the end of the last match to the end of the string, produces the fully replaced token list. The initialization does not matter, but (as an example) we set it as for an empty replacement.

```
7029 \cs_new:Npn \__regex_replacement_do_one_match:n #1
  {
    \__regex_query_range:nn
      { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
      { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
  }
```

(End definition for `\__regex_replacement_do_one_match:n`.)

`\__regex_replacement_exp_not:N` This function lets us navigate around the fact that the primitive `\exp_not:n` requires a braced argument. As far as I can tell, it is only needed if the user tries to include in the replacement text a control sequence set equal to a macro parameter character, such as `\c_parameter_token`. Indeed, within an x-expanding assignment, `\exp_not:N #` behaves as a single #, whereas `\exp_not:n {#}` behaves as a doubled ##.

```
7035 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End definition for `\__regex_replacement_exp_not:N`.)

`\__regex_replacement_exp_not:V` This is used for the implementation of \u, and it gets redefined for `\peek_regex_replace_once:nnTF`.

```
7036 \cs_new_eq:NN \__regex_replacement_exp_not:V \exp_not:V
```

(End definition for `\__regex_replacement_exp_not:V`.)

### 45.6.2 Query and brace balance

\\_\\_regex\\_query\\_range:nn  
\\_\\_regex\\_query\\_range\\_loop:ww

When it is time to extract submatches from the token list, the various tokens are stored in \toks registers numbered from \l\\_\\_regex\\_min\\_pos\\_int inclusive to \l\\_\\_regex\\_max\\_pos\\_int exclusive. The function \\_\\_regex\\_query\\_range:nn {\langle min \rangle} {\langle max \rangle} unpacks registers from the position {\langle min \rangle} to the position {\langle max \rangle} - 1 included. Once this is expanded, a second x-expansion results in the actual tokens from the query. That second expansion is only done by user functions at the very end of their operation, after checking (and correcting) the brace balance first.

```

7037 \cs_new:Npn \_\_regex_query_range:nn #1#2
7038 {
7039     \exp_after:wN \_\_regex_query_range_loop:ww
7040     \int_value:w \_\_regex_int_eval:w #1 \exp_after:wN ;
7041     \int_value:w \_\_regex_int_eval:w #2 ;
7042     \prg_break_point:
7043 }
7044 \cs_new:Npn \_\_regex_query_range_loop:ww #1 ; #2 ;
7045 {
7046     \if_int_compare:w #1 < #2 \exp_stop_f:
7047     \else:
7048         \exp_after:wN \prg_break:
7049     \fi:
7050     \_\_regex_toks_use:w #1 \exp_stop_f:
7051     \exp_after:wN \_\_regex_query_range_loop:ww
7052         \int_value:w \_\_regex_int_eval:w #1 + 1 ; #2 ;
7053 }
```

(End definition for \\_\\_regex\\_query\\_range:nn and \\_\\_regex\\_query\\_range\\_loop:ww.)

\\_\\_regex\\_query\\_submatch:n

Find the start and end positions for a given submatch (of a given match).

```

7054 \cs_new:Npn \_\_regex_query_submatch:n #1
7055 {
7056     \_\_regex_query_range:nn
7057         { \_\_kernel_intarray_item:Nn \g\_\_regex_submatch_begin_intarray {#1} }
7058         { \_\_kernel_intarray_item:Nn \g\_\_regex_submatch_end_intarray {#1} }
7059 }
```

(End definition for \\_\\_regex\\_query\\_submatch:n.)

\\_\\_regex\\_submatch\\_balance:n

Every user function must result in a balanced token list (unbalanced token lists cannot be stored by TeX). When we unpacked the query, we kept track of the brace balance, hence the contribution from a given range is the difference between the brace balances at the {\langle max pos \rangle} and {\langle min pos \rangle}. These two positions are found in the corresponding “submatch” arrays.

```

7060 \cs_new_protected:Npn \_\_regex_submatch_balance:n #1
7061 {
7062     \int_eval:n
7063     {
7064         \_\_regex_intarray_item:NnF \g\_\_regex_balance_intarray
7065             {
7066                 \_\_kernel_intarray_item:Nn
7067                     \g\_\_regex_submatch_end_intarray {#1}
7068             }
7069             { 0 }
```

```

7070      -
7071      \_\_regex\_intarray\_item:NnF \g\_\_regex\_balance\_intarray
7072      {
7073          \_\_kernel\_intarray\_item:Nn
7074          \g\_\_regex\_submatch\_begin\_intarray {\#1}
7075      }
7076      { 0 }
7077  }
7078 }
```

(End definition for `\_\_regex\_submatch\_balance:n`.)

### 45.6.3 Framework

```
\_\_regex\_replacement:n
\_\_regex\_replacement:x
    \_\_regex\_replacement\_apply:Nn
\_\_regex\_replacement\_set:n
```

The replacement text is built incrementally. We keep track in `\l\_\_regex\_balance\_int` of the balance of explicit begin- and end-group tokens and we store in `\g\_\_regex\_balance\_tl` some code to compute the brace balance from submatches (see its description). Detect unescaped right braces, and escaped characters, with trailing `\prg\_do\_nothing`: because some of the later function look-ahead. Once the whole replacement text has been parsed, make sure that there is no open csname. Finally, define the `balance\_one\_match` and `do\_one\_match` functions.

```

7079 \cs_new_protected:Npn \_\_regex\_replacement:n
7080     { \_\_regex\_replacement\_apply:Nn \_\_regex\_replacement\_set:n }
7081 \cs_new_protected:Npn \_\_regex\_replacement\_apply:Nn #1#2
7082     {
7083         \group_begin:
7084         \tl_build_begin:N \l\_\_regex\_build_tl
7085         \int_zero:N \l\_\_regex\_balance\_int
7086         \tl_gclear:N \g\_\_regex\_balance_tl
7087         \_\_regex\_escape\_use:nnnn
7088         {
7089             \if_charcode:w \c_right_brace_str ##1
7090                 \_\_regex\_replacement_rbrace:N
7091             \else:
7092                 \if_charcode:w \c_left_brace_str ##1
7093                     \_\_regex\_replacement_lbrace:N
7094                 \else:
7095                     \_\_regex\_replacement_normal:n
7096                     \fi:
7097                 \fi:
7098                 ##1
7099             }
7100             { \_\_regex\_replacement\_escaped:N ##1 }
7101             { \_\_regex\_replacement\_normal:n ##1 }
7102             {##2}
7103             \prg_do_nothing: \prg_do_nothing:
7104             \if_int_compare:w \l\_\_regex\_replacement\_csnames\_int > \c_zero_int
7105                 \msg_error:nnx { regex } { replacement-missing-rbrace }
7106                 { \int_use:N \l\_\_regex\_replacement\_csnames\_int }
7107                 \tl_build_put_right:Nx \l\_\_regex\_build_tl
7108                 { \prg_replicate:nn \l\_\_regex\_replacement\_csnames\_int \cs_end: }
7109             \fi:
7110             \seq_if_empty:NF \l\_\_regex\_replacement\_category\_seq
```

```

7111     {
7112         \msg_error:nnx { regex } { replacement-missing-rparen }
7113             { \seq_count:N \l__regex_replacement_category_seq }
7114             \seq_clear:N \l__regex_replacement_category_seq
7115         }
7116         \tl_gput_right:Nx \g__regex_balance_tl
7117             { + \int_use:N \l__regex_balance_int }
7118         \tl_build_end:N \l__regex_build_tl
7119         \exp_args:NNo
7120         \group_end:
7121         #1 \l__regex_build_tl
7122     }
7123 \cs_generate_variant:Nn \__regex_replacement:n { x }
7124 \cs_new_protected:Npn \__regex_replacement_set:n #1
7125     {
7126         \cs_set:Npn \__regex_replacement_do_one_match:n ##1
7127             {
7128                 \__regex_query_range:nn
7129                     {
7130                         \__kernel_intarray_item:Nn
7131                             \g__regex_submatch_prev_intarray {##1}
7132                     }
7133                     {
7134                         \__kernel_intarray_item:Nn
7135                             \g__regex_submatch_begin_intarray {##1}
7136                     }
7137                 #1
7138             }
7139         \exp_args:Nno \use:n
7140             { \cs_gset:Npn \__regex_replacement_balance_one_match:n ##1 }
7141             {
7142                 \g__regex_balance_tl
7143                 - \__regex_submatch_balance:n {##1}
7144             }
7145     }

```

(End definition for `\__regex_replacement:n`, `\__regex_replacement_apply:Nn`, and `\__regex_replacement_set:n`.)

```

\__regex_case_replacement:n
\__regex_case_replacement:x
7146 \tl_new:N \g__regex_case_replacement_tl
7147 \tl_new:N \g__regex_case_balance_tl
7148 \cs_new_protected:Npn \__regex_case_replacement:n #1
7149     {
7150         \tl_gset:Nn \g__regex_case_balance_tl
7151             {
7152                 \if_case:w
7153                     \__kernel_intarray_item:Nn
7154                         \g__regex_submatch_case_intarray {##1}
7155                 }
7156             \tl_gset_eq:NN \g__regex_case_replacement_tl \g__regex_case_balance_tl
7157             \tl_map_tokens:nn {#1}
7158                 { \__regex_replacement_apply:Nn \__regex_case_replacement_aux:n }
7159             \tl_gset:Nn \g__regex_balance_tl

```

```

7160      { \g__regex_case_balance_tl \fi: }
7161      \exp_args:No \_regex_replacement_set:n
7162      { \g__regex_case_replacement_tl \fi: }
7163  }
7164 \cs_generate_variant:Nn \_regex_case_replacement:n { x }
7165 \cs_new_protected:Npn \_regex_case_replacement_aux:n #1
7166  {
7167      \tl_gput_right:Nn \g__regex_case_replacement_tl { \or: #1 }
7168      \tl_gput_right:No \g__regex_case_balance_tl
7169      { \exp_after:wN \or: \g__regex_balance_tl }
7170  }

```

(End definition for `\_regex_case_replacement:n`.)

`\_regex_replacement_put:n` This gets redefined for `\peek_regex_replace_once:nnTF`.

```

7171 \cs_new_protected:Npn \_regex_replacement_put:n
7172  { \tl_build_put_right:Nn \l__regex_build_tl }

```

(End definition for `\_regex_replacement_put:n`.)

`\_regex_replacement_normal:n`  
`\_regex_replacement_normal_aux:N`

Most characters are simply sent to the output by `\tl_build_put_right:Nn`, unless a particular category code has been requested: then `\_regex_replacement_c_A:w` or a similar auxiliary is called. One exception is right parentheses, which restore the category code in place before the group started. Note that the sequence is non-empty there: it contains an empty entry corresponding to the initial value of `\l__regex_replacement_category_tl`. The argument `#1` is a single character (including the case of a catcode-other space). In case no specific catcode is requested, we take into account the current catcode regime (at the time the replacement is performed) as much as reasonable, with all impossible catcodes (escape, newline, etc.) being mapped to “other”.

```

7173 \cs_new_protected:Npn \_regex_replacement_normal:n #1
7174  {
7175      \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
7176      { \exp_args:No \_regex_replacement_put:n { \token_to_str:N #1 } }
7177      {
7178          \tl_if_empty:NTF \l__regex_replacement_category_tl
7179          { \_regex_replacement_normal_aux:N #1 }
7180          { %
7181              \token_if_eq_charcode:NNTF #1 )
7182              {
7183                  \seq_pop:NN \l__regex_replacement_category_seq
7184                  \l__regex_replacement_category_tl
7185              }
7186              {
7187                  \use:c { __regex_replacement_c_ \l__regex_replacement_category_tl :w }
7188                  ? #1
7189              }
7190          }
7191      }
7192  }
7193 \cs_new_protected:Npn \_regex_replacement_normal_aux:N #1
7194  {
7195      \token_if_eq_charcode:NNTF #1 \c_space_token
7196      { \_regex_replacement_c_S:w }
7197      {

```

```

7198   \exp_after:wN \exp_after:wN
7199   \if_case:w \tex_catcode:D '#1 \exp_stop_f:
7200     \_\_regex_replacement_c_0:w
7201   \or: \_\_regex_replacement_c_B:w
7202   \or: \_\_regex_replacement_c_E:w
7203   \or: \_\_regex_replacement_c_M:w
7204   \or: \_\_regex_replacement_c_T:w
7205   \or: \_\_regex_replacement_c_0:w
7206   \or: \_\_regex_replacement_c_P:w
7207   \or: \_\_regex_replacement_c_U:w
7208   \or: \_\_regex_replacement_c_D:w
7209   \or: \_\_regex_replacement_c_0:w
7210   \or: \_\_regex_replacement_c_S:w
7211   \or: \_\_regex_replacement_c_L:w
7212   \or: \_\_regex_replacement_c_0:w
7213   \or: \_\_regex_replacement_c_A:w
7214   \else: \_\_regex_replacement_c_0:w
7215   \fi:
7216 }
7217 ? #1
7218 }

```

(End definition for `\_\_regex_replacement_normal:n` and `\_\_regex_replacement_normal_aux:N`.)

`\_\_regex_replacement_escaped:N`

As in parsing a regular expression, we use an auxiliary built from #1 if defined. Otherwise, check for escaped digits (standing from submatches from 0 to 9): anything else is a raw character.

```

7219 \cs_new_protected:Npn \_\_regex_replacement_escaped:N #1
7220 {
7221   \cs_if_exist_use:cF { \_\_regex_replacement_#1:w }
7222   {
7223     \if_int_compare:w 1 < 1#1 \exp_stop_f:
7224       \_\_regex_replacement_put_submatch:n {#1}
7225     \else:
7226       \_\_regex_replacement_normal:n {#1}
7227     \fi:
7228   }
7229 }

```

(End definition for `\_\_regex_replacement_escaped:N`.)

#### 45.6.4 Submatches

`\_\_regex_replacement_put_submatch:n`

Insert a submatch in the replacement text. This is dropped if the submatch number is larger than the number of capturing groups. Unless the submatch appears inside a `\c{...}` or `\u{...}` construction, it must be taken into account in the brace balance. Later on, `##1` will be replaced by a pointer to the 0-th submatch for a given match. There is an `\exp_not:N` here as at the point-of-use of `\g_\_regex_balance_t1` there is an `x`-type expansion which is needed to get `##1` in correctly.

```

7230 \cs_new_protected:Npn \_\_regex_replacement_put_submatch:n #1
7231 {
7232   \if_int_compare:w #1 < \l_\_regex_capturing_group_int
7233     \_\_regex_replacement_put_submatch_aux:n {#1}
7234   \else:

```

```

7235   \msg_expandable_error:nnff { regex } { submatch-too-big }
7236   {#1} { \int_eval:n { \l__regex_capturing_group_int - 1 } }
7237   \fi:
7238 }
7239 \cs_new_protected:Npn \__regex_replacement_put_submatch_aux:n #1
7240 {
7241   \tl_build_put_right:Nn \l__regex_build_tl
7242   { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
7243   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7244     \tl_gput_right:Nn \g__regex_balance_tl
7245     { + \__regex_submatch_balance:n { \int_eval:n { #1 + ##1 } } }
7246   \fi:
7247 }

```

(End definition for `\__regex_replacement_put_submatch:n` and `\__regex_replacement_put_submatch_aux:n`.)

`\__regex_replacement_g:w`  
`\__regex_replacement_g_digits>NN`

Grab digits for the `\g` escape sequence in a primitive assignment to the integer `\l__regex_internal_a_int`. At the end of the run of digits, check that it ends with a right brace.

```

7248 \cs_new_protected:Npn \__regex_replacement_g:w #1#2
7249 {
7250   \token_if_eq_meaning:NNTF #1 \__regex_replacement_lbrace:N
7251   { \l__regex_internal_a_int = \__regex_replacement_g_digits>NN }
7252   { \__regex_replacement_error:NNN g #1 #2 }
7253 }
7254 \cs_new:Npn \__regex_replacement_g_digits>NN #1#2
7255 {
7256   \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
7257   {
7258     \if_int_compare:w 1 < 1#2 \exp_stop_f:
7259     #2
7260     \exp_after:wN \use_i:nnn
7261     \exp_after:wN \__regex_replacement_g_digits>NN
7262   \else:
7263     \exp_stop_f:
7264     \exp_after:wN \__regex_replacement_error:NNN
7265     \exp_after:wN g
7266   \fi:
7267 }
7268 {
7269   \exp_stop_f:
7270   \if_meaning:w \__regex_replacement_rbrace:N #1
7271     \exp_args:No \__regex_replacement_put_submatch:n
7272     { \int_use:N \l__regex_internal_a_int }
7273     \exp_after:wN \use_none:nn
7274   \else:
7275     \exp_after:wN \__regex_replacement_error:NNN
7276     \exp_after:wN g
7277   \fi:
7278 }
7279 #1 #2
7280 }

```

(End definition for `\__regex_replacement_g:w` and `\__regex_replacement_g_digits>NN`.)

#### 45.6.5 Csnames in replacement

`\_\_regex_replacement_c:w`  $\backslash c$  may only be followed by an unescaped character. If followed by a left brace, start a control sequence by calling an auxiliary common with  $\backslash u$ . Otherwise test whether the category is known; if it is not, complain.

```

7281 \cs_new_protected:Npn \_\_regex_replacement_c:w #1#2
7282 {
7283   \token_if_eq_meaning:NNTF #1 \_\_regex_replacement_normal:n
7284   {
7285     \cs_if_exist:cTF { \_\_regex_replacement_c:#2:w }
7286     { \_\_regex_replacement_cat:NNN #2 }
7287     { \_\_regex_replacement_error:NNN c #1#2 }
7288   }
7289   {
7290     \token_if_eq_meaning:NNTF #1 \_\_regex_replacement_lbrace:N
7291     { \_\_regex_replacement_cu_aux:Nw \_\_regex_replacement_exp_not:N }
7292     { \_\_regex_replacement_error:NNN c #1#2 }
7293   }
7294 }
```

(End definition for `\_\_regex_replacement_c:w`.)

`\_\_regex_replacement_cu_aux:Nw` Start a control sequence with  $\backslash cs:w$ , protected from expansion by #1 (either `\_\_regex_replacement_exp_not:N` or `\exp_not:V`), or turned to a string by `\tl_to_str:V` if inside another csname construction  $\backslash c$  or  $\backslash u$ . We use `\tl_to_str:V` rather than `\tl_to_str:N` to deal with integers and other registers.

```

7295 \cs_new_protected:Npn \_\_regex_replacement_cu_aux:Nw #1
7296 {
7297   \if_case:w \l_\_regex_replacement_csnames_int
7298     \tl_build_put_right:Nn \l_\_regex_build_tl
7299     { \exp_not:n { \exp_after:wN #1 \cs:w } }
7300   \else:
7301     \tl_build_put_right:Nn \l_\_regex_build_tl
7302     { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
7303   \fi:
7304   \int_incr:N \l_\_regex_replacement_csnames_int
7305 }
```

(End definition for `\_\_regex_replacement_cu_aux:Nw`.)

`\_\_regex_replacement_u:w` Check that  $\backslash u$  is followed by a left brace. If so, start a control sequence with  $\backslash cs:w$ , which is then unpacked either with `\exp_not:V` or `\tl_to_str:V` depending on the current context.

```

7306 \cs_new_protected:Npn \_\_regex_replacement_u:w #1#2
7307 {
7308   \token_if_eq_meaning:NNTF #1 \_\_regex_replacement_lbrace:N
7309   { \_\_regex_replacement_cu_aux:Nw \_\_regex_replacement_exp_not:V }
7310   { \_\_regex_replacement_error:NNN u #1#2 }
7311 }
```

(End definition for `\_\_regex_replacement_u:w`.)

\\_regex\\_replacement\\_rbrace:N Within a \c{...} or \u{...} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

7312 \cs_new_protected:Npn \_regex_replacement_rbrace:N #1
7313 {
7314     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7315         \tl_build_put_right:Nn \l__regex_build_tl { \cs_end: }
7316         \int_decr:N \l__regex_replacement_csnames_int
7317     \else:
7318         \_regex_replacement_normal:n {#1}
7319     \fi:
7320 }
```

(End definition for \\_regex\_replacement\_rbrace:N.)

\\_regex\\_replacement\\_brace:N Within a \c{...} or \u{...} construction, this is forbidden. Otherwise, this is a raw left brace.

```

7321 \cs_new_protected:Npn \_regex_replacement_lbrace:N #1
7322 {
7323     \if_int_compare:w \l__regex_replacement_csnames_int > \c_zero_int
7324         \msg_error:nnn { regex } { cu-lbrace } { u }
7325     \else:
7326         \_regex_replacement_normal:n {#1}
7327     \fi:
7328 }
```

(End definition for \\_regex\_replacement\_lbrace:N.)

#### 45.6.6 Characters in replacement

\\_regex\\_replacement\\_cat:NNN Here, #1 is a letter among BEMTPUDSLOA and #2#3 denote the next character. Complain if we reach the end of the replacement or if the construction appears inside \c{...} or \u{...}, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

7329 \cs_new_protected:Npn \_regex_replacement_cat:NNN #1#2#3
7330 {
7331     \token_if_eq_meaning:NNTF \prg_do_nothing: #3
7332     { \msg_error:nn { regex } { replacement-catcode-end } }
7333     {
7334         \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
7335         {
7336             \msg_error:nnnn
7337             { regex } { replacement-catcode-in-cs } {#1} {#3}
7338             #2 #3
7339         }
7340     {
7341         \_regex_two_if_eq:NNNNTF #2 #3 \_regex_replacement_normal:n (
7342         {
7343             \seq_push:NV \l__regex_replacement_category_seq
7344                 \l__regex_replacement_category_tl
7345                 \tl_set:Nn \l__regex_replacement_category_tl {#1}
7346             }
7347             {
7348                 \token_if_eq_meaning:NNT #2 \_regex_replacement_escaped:N
```

```

7349      {
7350          \__regex_char_if_alphanumeric:NTF #3
7351          {
7352              \msg_error:nnnn
7353                  { regex } { replacement-catcode-escaped }
7354                  {#1} {#3}
7355              }
7356          { }
7357      }
7358      \use:c { __regex_replacement_c_#1:w } #2 #3
7359  }
7360 }
7361 }
7362 }
```

(End definition for `\__regex_replacement_cat:NNN`.)

We now need to change the category code of the null character many times, hence work in a group. The catcode-specific macros below are defined in alphabetical order; if you are trying to understand the code, start from the end of the alphabet as those categories are simpler than active or begin-group.

```
7363 \group_begin:
```

`\__regex_replacement_char:nNN`

The only way to produce an arbitrary character–catcode pair is to use the `\lowercase` or `\uppercase` primitives. This is a wrapper for our purposes. The first argument is the null character with various catcodes. The second and third arguments are grabbed from the input stream: #3 is the character whose character code to reproduce. We could use `\char_generate:nn` but only for some catcodes (active characters and spaces are not supported).

```

7364     \cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
7365     {
7366         \tex_lccode:D 0 = '#3 \scan_stop:
7367         \tex_lowercase:D { \__regex_replacement_put:n {#1} }
7368     }
```

(End definition for `\__regex_replacement_char:nNN`.)

`\__regex_replacement_c_A:w`

For an active character, expansion must be avoided, twice because we later do two `x`-expansions, to unpack `\toks` for the query, and to expand their contents to tokens of the query.

```

7369     \char_set_catcode_active:N \^^@ 
7370     \cs_new_protected:Npn \__regex_replacement_c_A:w
7371         { \__regex_replacement_char:nNN { \exp_not:n { \exp_not:N \^^@ } } }
```

(End definition for `\__regex_replacement_c_A:w`.)

`\__regex_replacement_c_B:w`

An explicit begin-group token increases the balance, unless within a `\c{...}` or `\u{...}` construction. Add the desired begin-group character, using the standard `\if_false:` trick. We eventually `x`-expand twice. The first time must yield a balanced token list, and the second one gives the bare begin-group token. The `\exp_after:wN` is not strictly needed, but is more consistent with l3tl-analysis.

```

7372     \char_set_catcode_group_begin:N \^^@ 
7373     \cs_new_protected:Npn \__regex_replacement_c_B:w
7374         {
```

```

7375   \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7376     \int_incr:N \l__regex_balance_int
7377   \fi:
7378   \__regex_replacement_char:nNN
7379   { \exp_not:n { \exp_after:wN ^^@ \if_false: } \fi: } }
7380 }
```

(End definition for `\__regex_replacement_c_B:w`.)

#### `\__regex_replacement_c_C:w`

This is not quite catcode-related: when the user requests a character with category “control sequence”, the one-character control symbol is returned. As for the active character, we prepare for two x-expansions.

```

7381   \cs_new_protected:Npn \__regex_replacement_c_C:w #1#2
7382   {
7383     \tl_build_put_right:Nn \l__regex_build_tl
7384     { \exp_not:N \__regex_replacement_exp_not:N \exp_not:c {#2} }
7385 }
```

(End definition for `\__regex_replacement_c_C:w`.)

#### `\__regex_replacement_c_D:w`

Subscripts fit the mould: `\lowercase` the null byte with the correct category.

```

7386   \char_set_catcode_math_subscript:N ^^@
7387   \cs_new_protected:Npn \__regex_replacement_c_D:w
7388   { \__regex_replacement_char:nNN { ^^@ } }
```

(End definition for `\__regex_replacement_c_D:w`.)

#### `\__regex_replacement_c_E:w`

Similar to the begin-group case, the second x-expansion produces the bare end-group token.

```

7389   \char_set_catcode_group_end:N ^^@
7390   \cs_new_protected:Npn \__regex_replacement_c_E:w
7391   {
7392     \if_int_compare:w \l__regex_replacement_csnames_int = \c_zero_int
7393       \int_decr:N \l__regex_balance_int
7394     \fi:
7395     \__regex_replacement_char:nNN
7396     { \exp_not:n { \if_false: { \fi: ^^@ } } }
7397 }
```

(End definition for `\__regex_replacement_c_E:w`.)

#### `\__regex_replacement_c_L:w`

Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```

7398   \char_set_catcode_letter:N ^^@
7399   \cs_new_protected:Npn \__regex_replacement_c_L:w
7400   { \__regex_replacement_char:nNN { ^^@ } }
```

(End definition for `\__regex_replacement_c_L:w`.)

#### `\__regex_replacement_c_M:w`

No surprise here, we lowercase the null math toggle.

```

7401   \char_set_catcode_math_toggle:N ^^@
7402   \cs_new_protected:Npn \__regex_replacement_c_M:w
7403   { \__regex_replacement_char:nNN { ^^@ } }
```

(End definition for `\__regex_replacement_c_M:w`.)

\\_\\_regex\\_replacement\\_c\\_0:w Lowercase an other null byte.

```

7404     \char_set_catcode_other:N \^{\@}
7405     \cs_new_protected:Npn \_\_regex_replacement_c_0:w
7406     { \_\_regex_replacement_char:nNN { ^{\@} } }

```

(End definition for \\_\\_regex\\_replacement\\_c\\_0:w.)

\\_\\_regex\\_replacement\\_c\\_P:w For macro parameters, expansion is a tricky issue. We need to prepare for two x-expansions and passing through various macro definitions. Note that we cannot replace one \exp\_not:n by doubling the macro parameter characters because this would misbehave if a mischievous user asks for \c{\cP\#}, since that macro parameter character would be doubled.

```

7407     \char_set_catcode_parameter:N \^{\@}
7408     \cs_new_protected:Npn \_\_regex_replacement_c_P:w
7409     {
7410         \_\_regex_replacement_char:nNN
7411         { \exp_not:n { \exp_not:n { ^{\@} } } }
7412     }

```

(End definition for \\_\\_regex\\_replacement\\_c\\_P:w.)

\\_\\_regex\\_replacement\\_c\\_S:w Spaces are normalized on input by TeX to have character code 32. It is in fact impossible to get a token with character code 0 and category code 10. Hence we use 32 instead of 0 as our base character.

```

7413     \cs_new_protected:Npn \_\_regex_replacement_c_S:w #1#2
7414     {
7415         \if_int_compare:w '#2 = \c_zero_int
7416             \msg_error:nn { regex } { replacement-null-space }
7417         \fi:
7418         \tex_lccode:D '\ = '#2 \scan_stop:
7419         \tex_lowercase:D { \_\_regex_replacement_put:n {~} }
7420     }

```

(End definition for \\_\\_regex\\_replacement\\_c\\_S:w.)

\\_\\_regex\\_replacement\\_c\\_T:w No surprise for alignment tabs here. Those are surrounded by the appropriate braces whenever necessary, hence they don't cause trouble in alignment settings.

```

7421     \char_set_catcode_alignment:N \^{\@}
7422     \cs_new_protected:Npn \_\_regex_replacement_c_T:w
7423     { \_\_regex_replacement_char:nNN { ^{\@} } }

```

(End definition for \\_\\_regex\\_replacement\\_c\\_T:w.)

\\_\\_regex\\_replacement\\_c\\_U:w Simple call to \\_\\_regex\\_replacement\\_char:nNN which lowers the math superscript ^{\@}.

```

7424     \char_set_catcode_math_superscript:N \^{\@}
7425     \cs_new_protected:Npn \_\_regex_replacement_c_U:w
7426     { \_\_regex_replacement_char:nNN { ^{\@} } }

```

(End definition for \\_\\_regex\\_replacement\\_c\\_U:w.)

Restore the catcode of the null byte.

```

7427     \group_end:

```

### 45.6.7 An error

`\_regex_replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
7428 \cs_new_protected:Npn \_regex_replacement_error:NNN #1#2#3
7429 {
7430     \msg_error:nnx { regex } { replacement-#1 } {#3}
7431     #2 #3
7432 }
```

(End definition for `\_regex_replacement_error:NNN`.)

## 45.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
7433 \cs_new_protected:Npn \regex_new:N #1
7434 { \cs_new_eq:NN #1 \c_regex_no_match_regex }
```

(End definition for `\regex_new:N`. This function is documented on page 54.)

`\l_tmpa_regex` The usual scratch space.

```
7435 \regex_new:N \l_tmpa_regex
7436 \regex_new:N \l_tmpb_regex
7437 \regex_new:N \g_tmpa_regex
7438 \regex_new:N \g_tmpb_regex
```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 59.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.

```
7439 \cs_new_protected:Npn \regex_set:Nn #1#2
7440 {
7441     \__regex_compile:n {#2}
7442     \tl_set_eq:NN #1 \l__regex_internal_regex
7443 }
7444 \cs_new_protected:Npn \regex_gset:Nn #1#2
7445 {
7446     \__regex_compile:n {#2}
7447     \tl_gset_eq:NN #1 \l__regex_internal_regex
7448 }
7449 \cs_new_protected:Npn \regex_const:Nn #1#2
7450 {
7451     \__regex_compile:n {#2}
7452     \tl_const:Nx #1 { \exp_not:o \l__regex_internal_regex }
7453 }
```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 54.)

`\regex_show:n` User functions: the `n` variant requires compilation first. Then show the variable with some appropriate text. The auxiliary `\_regex_show:N` is defined in a different section.

```
7454 \cs_new_protected:Npn \regex_show:n { \_regex_show:Nn \msg_show:nnxxxx }
7455 \cs_new_protected:Npn \regex_log:n { \_regex_show:Nn \msg_log:nnxxxx }
7456 \cs_new_protected:Npn \_regex_show:Nn #1#2
```

`\_regex_show:NN`

```

7457  {
7458    \_\_regex\_compile:n {#2}
7459    \_\_regex\_show:N \l\_\_regex\_internal\_regex
7460    #1 { regex } { show }
7461    { \tl\_to\_str:n {#2} } { }
7462    { \l\_\_regex\_internal\_a\_tl } { }
7463  }
7464 \cs\_new\_protected:Npn \regex\_show:N { \_\_regex\_show:NN \msg\_show:nxxxxx }
7465 \cs\_new\_protected:Npn \regex\_log:N { \_\_regex\_show:NN \msg\_log:nxxxxx }
7466 \cs\_new\_protected:Npn \_\_regex\_show:NN #1#2
7467  {
7468    \_\_kernel\_chk\_tl\_type:NnnT #2 { regex }
7469    { \exp\_args:No \_\_regex\_clean\_regex:n {#2} }
7470    {
7471      \_\_regex\_show:N #2
7472      #1 { regex } { show }
7473      { } { \token\_to\_str:N #2 }
7474      { \l\_\_regex\_internal\_a\_tl } { }
7475    }
7476  }

```

(End definition for `\regex_show:n` and others. These functions are documented on page 54.)

**\regex\_match:nnTF** Those conditionals are based on a common auxiliary defined later. Its first argument builds the NFA corresponding to the regex, and the second argument is the query token list. Once we have performed the match, convert the resulting boolean to `\prg_return_true:` or `false`.

```

7477 \prg\_new\_protected\_conditional:Npnn \regex\_match:nn #1#2 { T , F , TF }
7478  {
7479    \_\_regex\_if\_match:nn { \_\_regex\_build:n {#1} } {#2}
7480    \_\_regex\_return:
7481  }
7482 \prg\_new\_protected\_conditional:Npnn \regex\_match:Nn #1#2 { T , F , TF }
7483  {
7484    \_\_regex\_if\_match:nn { \_\_regex\_build:N #1 } {#2}
7485    \_\_regex\_return:
7486  }

```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page 55.)

**\regex\_count:nnN** Again, use an auxiliary whose first argument builds the NFA.

**\regex\_count:NnN**

```

7487 \cs\_new\_protected:Npn \regex\_count:nnN #1
7488  { \_\_regex\_count:nnN { \_\_regex\_build:n {#1} } }
7489 \cs\_new\_protected:Npn \regex\_count:NnN #1
7490  { \_\_regex\_count:nnN { \_\_regex\_build:N #1 } }

```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 55.)

**\regex\_match\_case:nn** The auxiliary errors if #1 has an odd number of items, and otherwise it sets `\g_regex_case_int` according to which case was found (zero if not found). The `true` branch leaves the corresponding code in the input stream.

```

7491 \cs\_new\_protected:Npn \regex\_match\_case:nnTF #1#2#3
7492  {

```

```

7493     \_\_regex\_match\_case:nnTF {#1} {#2}
7494     {
7495         \tl\_item:nn {#1} { 2 * \g\_\_regex\_case\_int }
7496         #3
7497     }
7498 }
7499 \cs\_new\_protected:Npn \regex\_match\_case:nn #1#2
7500   { \regex\_match\_case:nnTF {#1} {#2} { } { } }
7501 \cs\_new\_protected:Npn \regex\_match\_case:nnT #1#2#3
7502   { \regex\_match\_case:nnTF {#1} {#2} {#3} { } }
7503 \cs\_new\_protected:Npn \regex\_match\_case:nnF #1#2
7504   { \regex\_match\_case:nnTF {#1} {#2} { } }

```

(End definition for `\regex_match_case:nnTF`. This function is documented on page 55.)

We define here 40 user functions, following a common pattern in terms of `:nnN` auxiliaries, defined in the coming subsections. The auxiliary is handed `\_\_regex_build:n` or `\_\_regex_build:N` with the appropriate regex argument, then all other necessary arguments (replacement text, token list, *etc.*) The conditionals call `\_\_regex_return:` to return either true or false once matching has been performed.

```

7505 \cs\_set\_protected:Npn \_\_regex\_tmp:w #1#2#3
7506   {
7507       \cs\_new\_protected:Npn #2 ##1 { #1 { \_\_regex\_build:n {##1} } }
7508       \cs\_new\_protected:Npn #3 ##1 { #1 { \_\_regex\_build:N ##1 } }
7509       \prg\_new\_protected\_conditional:Npnn #2 ##1##2##3 { T , F , TF }
7510         { #1 { \_\_regex\_build:n {##1} } {##2} ##3 \_\_regex\_return: }
7511       \prg\_new\_protected\_conditional:Npnn #3 ##1##2##3 { T , F , TF }
7512         { #1 { \_\_regex\_build:N ##1 } {##2} ##3 \_\_regex\_return: }
7513   }
7514 \_\_regex\_tmp:w \_\_regex\_extract\_once:nnN
7515   \regex\_extract\_once:nnN \regex\_extract\_once:NnN
7516 \_\_regex\_tmp:w \_\_regex\_extract\_all:nnN
7517   \regex\_extract\_all:nnN \regex\_extract\_all:NnN
7518 \_\_regex\_tmp:w \_\_regex\_replace\_once:nnN
7519   \regex\_replace\_once:nnN \regex\_replace\_once:NnN
7520 \_\_regex\_tmp:w \_\_regex\_replace\_all:nnN
7521   \regex\_replace\_all:nnN \regex\_replace\_all:NnN
7522 \_\_regex\_tmp:w \_\_regex\_split:nnN \regex\_split:nnN \regex\_split:NnN

```

(End definition for `\regex_extract_once:nnNTF` and others. These functions are documented on page 56.)

If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_replace_once:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use. The `\tl\_item:nn` is only expanded once we know the value of `\g\_\_regex\_case\_int`, namely which case matched.

```

7523 \cs\_new\_protected:Npn \regex\_replace\_case\_once:nNTF #1#2
7524   {
7525       \int\_if\_odd:nTF { \tl\_count:n {#1} }
7526       {
7527           \msg\_error:nnxxxx { regex } { case-odd }
7528             { \token\_to\_str:N \regex\_replace\_case\_once:nN(TF) } { code }
7529             { \tl\_count:n {#1} } { \tl\_to\_str:n {#1} }
7530           \use\_ii:nn

```

```

7531 }
7532 {
7533     \_\_regex\_replace\_once\_aux:nnN
7534     { \_\_regex\_case\_build:x { \_\_regex\_tl\_odd\_items:n {#1} } }
7535     { \_\_regex\_replacement:x { \tl\_item:nn {#1} { 2 * \g\_\_regex\_case\_int } } }
7536     #2
7537     \bool_if:NTF \g\_\_regex\_success\_bool
7538 }
7539 }
7540 \cs_new_protected:Npn \regex_replace_case_once:nN #1#2
7541     { \regex_replace_case_once:nNTF {#1} {#2} { } { } }
7542 \cs_new_protected:Npn \regex_replace_case_once:nNT #1#2#3
7543     { \regex_replace_case_once:nNTF {#1} {#2} {#3} { } }
7544 \cs_new_protected:Npn \regex_replace_case_once:nNF #1#2
7545     { \regex_replace_case_once:nNTF {#1} {#2} { } }

```

(End definition for `\regex_replace_case_once:nNTF`. This function is documented on page 58.)

`\regex_case_replace_all:nN` If the input is bad (odd number of items) then take the false branch. Otherwise, use the same auxiliary as `\regex_case_replace_all:nnN`, but with more complicated code to build the automaton, and to find what replacement text to use.

```

7546 \cs_new_protected:Npn \regex_case_replace_all:nNTF #1#2
7547 {
7548     \int_if_odd:nTF { \tl_count:n {#1} }
7549     {
7550         \msg_error:nnxxxx { regex } { case-odd }
7551         { \token_to_str:N \regex_case_replace_all:nN(TF) } { code }
7552         { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7553         \use_i:nn
7554     }
7555     {
7556         \_\_regex_replace_all_aux:nnN
7557         { \_\_regex_case_build:x { \_\_regex_tl_odd_items:n {#1} } }
7558         { \_\_regex_case_replacement:x { \_\_regex_tl_even_items:n {#1} } }
7559         #2
7560         \bool_if:NTF \g\_\_regex\_success\_bool
7561     }
7562 }
7563 \cs_new_protected:Npn \regex_case_replace_all:nN #1#2
7564     { \regex_case_replace_all:nNTF {#1} {#2} { } { } }
7565 \cs_new_protected:Npn \regex_case_replace_all:nNT #1#2#3
7566     { \regex_case_replace_all:nNTF {#1} {#2} {#3} { } }
7567 \cs_new_protected:Npn \regex_case_replace_all:nNF #1#2
7568     { \regex_case_replace_all:nNTF {#1} {#2} { } }

```

(End definition for `\regex_case_replace_all:nNTF`. This function is documented on page ??.)

#### 45.7.1 Variables and helpers for user functions

`\l\_\_regex\_match\_count\_int` The number of matches found so far is stored in `\l\_\_regex\_match\_count\_int`. This is only used in the `\regex_count:nnN` functions.

```
7569 \int_new:N \l\_\_regex\_match\_count\_int
```

(End definition for `\l\_\_regex\_match\_count\_int`.)

`--regex_begin` Those flags are raised to indicate begin-group or end-group tokens that had to be added  
`--regex_end` when extracting submatches.

```
7570 \flag_new:n { __regex_begin }
7571 \flag_new:n { __regex_end }
```

(*End definition for `--regex_begin` and `--regex_end`.*)

`\l__regex_min_submatch_int`
`\l__regex_submatch_int`
`\l__regex_zeroth_submatch_int`

The end-points of each submatch are stored in two arrays whose index `<submatch>` ranges from `\l__regex_min_submatch_int` (inclusive) to `\l__regex_submatch_int` (exclusive). Each successful match comes with a 0-th submatch (the full match), and one match for each capturing group: submatches corresponding to the last successful match are labelled starting at `zeroth_submatch`. The entry `\l__regex_zeroth_submatch_int` in `\g__regex_submatch_prev_intarray` holds the position at which that match attempt started: this is used for splitting and replacements.

```
7572 \int_new:N \l__regex_min_submatch_int
7573 \int_new:N \l__regex_submatch_int
7574 \int_new:N \l__regex_zeroth_submatch_int
```

(*End definition for `\l__regex_min_submatch_int`, `\l__regex_submatch_int`, and `\l__regex_zeroth_submatch_int`.*)

`\g__regex_submatch_prev_intarray`
`\g__regex_submatch_begin_intarray`
`\g__regex_submatch_end_intarray`
`\g__regex_submatch_case_intarray`

Hold the place where the match attempt begun, the end-points of each submatch, and which regex case the match corresponds to, respectively.

```
7575 \intarray_new:Nn \g__regex_submatch_prev_intarray { 65536 }
7576 \intarray_new:Nn \g__regex_submatch_begin_intarray { 65536 }
7577 \intarray_new:Nn \g__regex_submatch_end_intarray { 65536 }
7578 \intarray_new:Nn \g__regex_submatch_case_intarray { 65536 }
```

(*End definition for `\g__regex_submatch_prev_intarray` and others.*)

`\g__regex_balance_intarray`

The first thing we do when matching is to store the balance of begin-group/end-group characters into `\g__regex_balance_intarray`.

```
7579 \intarray_new:Nn \g__regex_balance_intarray { 65536 }
```

(*End definition for `\g__regex_balance_intarray`.*)

`\l__regex_added_begin_int`
`\l__regex_added_end_int`

Keep track of the number of left/right braces to add when performing a regex operation such as a replacement.

```
7580 \int_new:N \l__regex_added_begin_int
7581 \int_new:N \l__regex_added_end_int
```

(*End definition for `\l__regex_added_begin_int` and `\l__regex_added_end_int`.*)

`\__regex_return:`

This function triggers either `\prg_return_false:` or `\prg_return_true:` as appropriate to whether a match was found or not. It is used by all user conditionals.

```
7582 \cs_new_protected:Npn \__regex_return:
7583 {
7584     \if_meaning:w \c_true_bool \g__regex_success_bool
7585         \prg_return_true:
7586     \else:
7587         \prg_return_false:
7588     \fi:
7589 }
```

(End definition for `\_regex_return::`)

`\_regex_query_set:n` To easily extract subsets of the input once we found the positions at which to cut, store the input tokens one by one into successive `\toks` registers. Also store the brace balance (used to check for overall brace balance) in an array.

```
7590 \cs_new_protected:Npn \_regex_query_set:n #1
7591 {
7592     \int_zero:N \l__regex_balance_int
7593     \int_zero:N \l__regex_curr_pos_int
7594     \_regex_query_set_aux:nN { } F
7595     \tl_analysis_map_inline:nn {#1}
7596     { \_regex_query_set_aux:nN {##1} ##3 }
7597     \_regex_query_set_aux:nN { } F
7598     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
7599 }
7600 \cs_new_protected:Npn \_regex_query_set_aux:nN #1#2
7601 {
7602     \int_incr:N \l__regex_curr_pos_int
7603     \_regex_toks_set:Nn \l__regex_curr_pos_int {#1}
7604     \_kernel_intarray_gset:Nnn \g__regex_balance_intarray
7605     { \l__regex_curr_pos_int } { \l__regex_balance_int }
7606     \if_case:w "#2 \exp_stop_f:
7607     \or: \int_incr:N \l__regex_balance_int
7608     \or: \int_decr:N \l__regex_balance_int
7609     \fi:
7610 }
```

(End definition for `\_regex_query_set:n` and `\_regex_query_set_aux:nN`.)

### 45.7.2 Matching

`\_regex_if_match:nn` We don't track submatches, and stop after a single match. Build the NFA with #1, and perform the match on the query #2.

```
7611 \cs_new_protected:Npn \_regex_if_match:nn #1#2
7612 {
7613     \group_begin:
7614         \_regex_disable_submatches:
7615         \_regex_single_match:
7616         #1
7617         \_regex_match:n {#2}
7618     \group_end:
7619 }
```

(End definition for `\_regex_if_match:nn`.)

`\_regex_match_case:nnTF` The code would get badly messed up if the number of items in #1 were not even, so we catch this case, then follow the same code as `\regex_match:nnTF` but using `\_regex_case_build:n` and without returning a result.

```
7620 \cs_new_protected:Npn \_regex_match_case:nnTF #1#2
7621 {
7622     \int_if_odd:nTF { \tl_count:n {#1} }
7623     {
7624         \msg_error:nnxxxx { regex } { case-odd }
7625         { \token_to_str:N \regex_match_case:nn(TF) } { code }
```

```

7626     { \tl_count:n {#1} } { \tl_to_str:n {#1} }
7627     \use_i:nn
7628   }
7629   {
7630     \__regex_if_match:nn
7631     { \__regex_case_build:x { \__regex_tl_odd_items:n {#1} } }
7632     {#2}
7633     \bool_if:NTF \g__regex_success_bool
7634   }
7635 }
7636 \cs_new:Npn \__regex_match_case_aux:nn #1#2 { \exp_not:n { {#1} } }

```

(End definition for `\__regex_match_case:nnTF` and `\__regex_match_case:nn`.)

`\__regex_count:nnN` Again, we don't care about submatches. Instead of aborting after the first "longest match" is found, we search for multiple matches, incrementing `\l__regex_match_count_int` every time to record the number of matches. Build the NFA and match. At the end, store the result in the user's variable.

```

7637 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
7638   {
7639     \group_begin:
7640     \__regex_disable_submatches:
7641     \int_zero:N \l__regex_match_count_int
7642     \__regex_multi_match:n { \int_incr:N \l__regex_match_count_int }
7643     #1
7644     \__regex_match:n {#2}
7645     \exp_args:NNo
7646     \group_end:
7647     \int_set:Nn #3 { \int_use:N \l__regex_match_count_int }
7648   }

```

(End definition for `\__regex_count:nnN`.)

### 45.7.3 Extracting submatches

`\__regex_extract_once:nnN` Match once or multiple times. After each match (or after the only match), extract the submatches using `\__regex_extract:..`. At the end, store the sequence containing all the submatches into the user variable `#3` after closing the group.

```

7649 \cs_new_protected:Npn \__regex_extract_once:nnN #1#2#3
7650   {
7651     \group_begin:
7652     \__regex_single_match:
7653     #1
7654     \__regex_match:n {#2}
7655     \__regex_extract:
7656     \__regex_query_set:n {#2}
7657     \__regex_group_end_extract_seq:N #3
7658   }
7659 \cs_new_protected:Npn \__regex_extract_all:nnN #1#2#3
7660   {
7661     \group_begin:
7662     \__regex_multi_match:n { \__regex_extract: }
7663     #1
7664     \__regex_match:n {#2}

```

```

7665      \_regex_query_set:n {#2}
7666      \_regex_group_end_extract_seq:N #3
7667  }

```

(End definition for `\_regex_extract_once:nnN` and `\_regex_extract_all:nnN`.)

`\_regex_split:nnN` Splitting at submatches is a bit more tricky. For each match, extract all submatches, and replace the zeroth submatch by the part of the query between the start of the match attempt and the start of the zeroth submatch. This is inhibited if the delimiter matched an empty token list at the start of this match attempt. After the last match, store the last part of the token list, which ranges from the start of the match attempt to the end of the query. This step is inhibited if the last match was empty and at the very end: decrement `\l_regex_submatch_int`, which controls which matches will be used.

```

7668 \cs_new_protected:Npn \_regex_split:nnN #1#2#3
7669  {
7670    \group_begin:
7671    \_regex_multi_match:n
7672    {
7673      \if_int_compare:w
7674        \l_regex_start_pos_int < \l_regex_success_pos_int
7675        \_regex_extract:
7676        \_kernel_intarray_gset:Nnn \g_regex_submatch_prev_intarray
7677        { \l_regex_zeroth_submatch_int } { 0 }
7678        \_kernel_intarray_gset:Nnn \g_regex_submatch_end_intarray
7679        { \l_regex_zeroth_submatch_int }
7680        {
7681          \_kernel_intarray_item:Nn \g_regex_submatch_begin_intarray
7682          { \l_regex_zeroth_submatch_int }
7683        }
7684        \_kernel_intarray_gset:Nnn \g_regex_submatch_begin_intarray
7685        { \l_regex_zeroth_submatch_int }
7686        { \l_regex_start_pos_int }
7687      \fi:
7688    }
7689    #1
7690    \_regex_match:n {#2}
7691    \_regex_query_set:n {#2}
7692    \_kernel_intarray_gset:Nnn \g_regex_submatch_prev_intarray
7693    { \l_regex_submatch_int } { 0 }
7694    \_kernel_intarray_gset:Nnn \g_regex_submatch_end_intarray
7695    { \l_regex_submatch_int }
7696    { \l_regex_max_pos_int }
7697    \_kernel_intarray_gset:Nnn \g_regex_submatch_begin_intarray
7698    { \l_regex_submatch_int }
7699    { \l_regex_start_pos_int }
7700    \int_incr:N \l_regex_submatch_int
7701    \if_meaning:w \c_true_bool \l_regex_empty_success_bool
7702      \if_int_compare:w \l_regex_start_pos_int = \l_regex_max_pos_int
7703        \int_decr:N \l_regex_submatch_int
7704      \fi:
7705    \fi:
7706    \_regex_group_end_extract_seq:N #3
7707  }

```

(End definition for `\_regex_group_end_extract_seq:N`.)

`\_regex_group_end_extract_seq:N`

The end-points of submatches are stored as entries of two arrays from `\l_regex_min_submatch_int` to `\l_regex_submatch_int` (exclusive). Extract the relevant ranges into `\g_regex_internal_t1`, separated by `\_regex_tmp:w`. We keep track in the two flags `_regex_begin` and `_regex_end` of the number of begin-group or end-group tokens added to make each of these items overall balanced. At this step, }{ is counted as being balanced (same number of begin-group and end-group tokens). This problem is caught by `\_regex_extract_check:w`, explained later. After complaining about any begin-group or end-group tokens we had to add, we are ready to construct the user's sequence outside the group.

```

7708 \cs_new_protected:Npn \_regex_group_end_extract_seq:N #1
7709  {
7710      \flag_clear:n { _regex_begin }
7711      \flag_clear:n { _regex_end }
7712      \cs_set_eq:NN \_regex_tmp:w \scan_stop:
7713      \kernel_tl_gset:Nx \g_regex_internal_t1
7714      {
7715          \int_step_function:nnN { \l_regex_min_submatch_int }
7716          { \l_regex_submatch_int - 1 } \_regex_extract_seq_aux:n
7717          \_regex_tmp:w
7718      }
7719      \int_set:Nn \l_regex_added_begin_int
7720      { \flag_height:n { _regex_begin } }
7721      \int_set:Nn \l_regex_added_end_int
7722      { \flag_height:n { _regex_end } }
7723      \tex_afterassignment:D \_regex_extract_check:w
7724      \kernel_tl_gset:Nx \g_regex_internal_t1
7725      { \g_regex_internal_t1 \if_false: { \fi: } }
7726      \int_compare:nNnT
7727      { \l_regex_added_begin_int + \l_regex_added_end_int } > 0
7728      {
7729          \msg_error:nnxxx { regex } { result-unbalanced }
7730          { splitting-or-extracting-submatches }
7731          { \int_use:N \l_regex_added_begin_int }
7732          { \int_use:N \l_regex_added_end_int }
7733      }
7734      \group_end:
7735      \cs_set_eq:NN \_regex_tmp:w \_regex_extract_map_loop:w
7736      \seq_set_from_function:NnN #1
7737      { \_regex_extract_map:N } \exp_not:n
7738  }

```

(End definition for `\_regex_group_end_extract_seq:N`.)

`\_regex_extract_seq_aux:n`  
`\_regex_extract_seq_aux:ww`

The `:n` auxiliary builds one item of the sequence of submatches. First compute the brace balance of the submatch, then extract the submatch from the query, adding the appropriate braces and raising a flag if the submatch is not balanced.

```

7739 \cs_new:Npn \_regex_extract_seq_aux:n #1
7740  {
7741      \_regex_tmp:w { }
7742      \exp_after:wN \_regex_extract_seq_aux:ww
7743      \int_value:w \_regex_submatch_balance:n {#1} ; #1;

```

```

7744     }
7745 \cs_new:Npn \__regex_extract_seq_aux:ww #1; #2;
7746 {
7747     \if_int_compare:w #1 < \c_zero_int
7748         \prg_replicate:nn {-#1}
7749         {
7750             \flag_raise:n { __regex_begin }
7751             \exp_not:n { { \if_false: } \fi: }
7752         }
7753     \fi:
7754     \__regex_query_submatch:n {#2}
7755     \if_int_compare:w #1 > \c_zero_int
7756         \prg_replicate:nn {#1}
7757         {
7758             \flag_raise:n { __regex_end }
7759             \exp_not:n { \if_false: { \fi: } }
7760         }
7761     \fi:
7762 }

```

(End definition for `\__regex_extract_seq_aux:n` and `\__regex_extract_seq_aux:ww`.)

```

\__regex_extract_check:w
\__regex_extract_check:n
    \__regex_extract_check_loop:w
\__regex_extract_check_end:w

```

In `\__regex_group_end_extract_seq:N` we had to expand `\g__regex_internal_tl` to turn `\if_false:` constructions into actual begin-group and end-group tokens. This is done with a `\__kernel_tl_gset:Nx` assignment, and `\__regex_extract_check:w` is run immediately after this assignment ends, thanks to the `\afterassignment` primitive. If all of the items were properly balanced (enough begin-group tokens before end-group tokens, so `{` is not) then `\__regex_extract_check:w` is called just before the closing brace of the `\__kernel_tl_gset:Nx` (thanks to our sneaky `\if_false: { \fi: }` construction), and finds that there is nothing left to expand. If any of the items is unbalanced, the assignment gets ended early by an extra end-group token, and our check finds more tokens needing to be expanded in a new `\__kernel_tl_gset:Nx` assignment. We need to add a begin-group and an end-group tokens to the unbalanced item, namely to the last item found so far, which we reach through a loop.

```

7763 \cs_new_protected:Npn \__regex_extract_check:w
7764 {
7765     \exp_after:wN \__regex_extract_check:n
7766     \exp_after:wN { \if_false: } \fi:
7767 }
7768 \cs_new_protected:Npn \__regex_extract_check:n #1
7769 {
7770     \tl_if_empty:nF {#1}
7771     {
7772         \int_incr:N \l__regex_added_begin_int
7773         \int_incr:N \l__regex_added_end_int
7774         \tex_afterassignment:D \__regex_extract_check:w
7775         \__kernel_tl_gset:Nx \g__regex_internal_tl
7776         {
7777             \exp_after:wN \__regex_extract_check_loop:w
7778             \g__regex_internal_tl
7779             \__regex_tmp:w \__regex_extract_check_end:w
7780             #1
7781         }

```

```

7782     }
7783   }
7784 \cs_new:Npn \__regex_extract_check_loop:w #1 \__regex_tmp:w #2
7785   {
7786     #2
7787     \exp_not:o {#1}
7788     \__regex_tmp:w { }
7789     \__regex_extract_check_loop:w \prg_do_nothing:
7790   }

```

Arguments of `\__regex_extract_check_end:w` are: #1 is the part of the item before the extra end-group token; #2 is junk; #3 is `\prg_do_nothing:` followed by the not-yet-expanded part of the item after the extra end-group token. In the replacement text, the first brace and the `\if_false: { \fi: }` construction are the added begin-group and end-group tokens (the latter being not-yet expanded, just like #3), while the closing brace after `\exp_not:o {#1}` replaces the extra end-group token that had ended the assignment early. In particular this means that the character code of that end-group token is lost.

```

7791 \cs_new:Npn \__regex_extract_check_end:w
7792   \exp_not:o #1#2 \__regex_extract_check_loop:w #3 \__regex_tmp:w
7793   {
7794     { \exp_not:o {#1} }
7795     #3
7796     \if_false: { \fi: }
7797     \__regex_tmp:w
7798   }

```

(End definition for `\__regex_extract_check:w` and others.)

```

\__regex_extract_map:N
\__regex_extract_map_aux>NNn
\__regex_extract_map_loop:w

```

This receives a `seq` internal function and maps it over all items in `\g__regex_internal_tl`. This token list takes the form `\__regex_tmp:w {} <item1> \__regex_tmp:w {} <item2> ... \__regex_tmp:w`, and the calling code has set `\__regex_tmp:w` equal to `\__regex_extract_map_loop:w`. The loop is otherwise pretty standard, with `\prg_do_nothing:` to avoid losing braces.

```

7799 \cs_new:Npn \__regex_extract_map:N #1
7800   {
7801     \exp_after:wN \__regex_extract_map_aux>NNn
7802     \exp_after:wN #1
7803     \g__regex_internal_tl \use_none:nnn
7804   }
7805 \cs_new:Npn \__regex_extract_map_aux>NNn #1#2#3
7806   { #3 #2 #1 \prg_do_nothing: }
7807 \cs_new:Npn \__regex_extract_map_loop:w #1#2 \__regex_tmp:w #3
7808   {
7809     \exp_after:wN #1 \exp_after:wN {#2}
7810     #3 \__regex_extract_map_loop:w #1 \prg_do_nothing:
7811   }

```

(End definition for `\__regex_extract_map:N`, `\__regex_extract_map_aux>NNn`, and `\__regex_extract_map_loop:w`.)

```

\__regex_extract:
\__regex_extract_aux:

```

Our task here is to store the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. First, we store in `\g__regex_submatch_prev_intarray` the position at which the match attempt started. We extract the rest from the comma list `\l__regex_success_submatches_tl`, which

starts with entries to be stored in `\g__regex_submatch_begin_intarray` and continues with entries for `\g__regex_submatch_end_intarray`.

```

7812 \cs_new_protected:Npn __regex_extract:
7813   {
7814     \if_meaning:w \c_true_bool \g__regex_success_bool
7815       \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
7816       \prg_replicate:nn \l__regex_capturing_group_int
7817         {
7818           \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7819             { \l__regex_submatch_int } { 0 }
7820           \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7821             { \l__regex_submatch_int } { 0 }
7822             \int_incr:N \l__regex_submatch_int
7823         }
7824       \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
7825         { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
7826       \__kernel_intarray_gset:Nnn \g__regex_submatch_case_intarray
7827         { \l__regex_zeroth_submatch_int } { \g__regex_case_int }
7828         \int_zero:N \l__regex_internal_a_int
7829         \exp_after:wN \__regex_extract_aux:w \l__regex_success_submatches_tl
7830           \prg_break_point: \__regex_use_none_delimit_by_q_recursion_stop:w ,
7831             \q__regex_recursion_stop
7832         \fi:
7833       }
7834     \cs_new_protected:Npn __regex_extract_aux:w #1 ,
7835       {
7836         \prg_break: #1 \prg_break_point:
7837         \if_int_compare:w \l__regex_internal_a_int < \l__regex_capturing_group_int
7838           \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
7839             { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int } {#}
7840         \else:
7841           \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
7842             { \__regex_int_eval:w \l__regex_zeroth_submatch_int + \l__regex_internal_a_int - \l__}
7843         \fi:
7844         \int_incr:N \l__regex_internal_a_int
7845         \__regex_extract_aux:w
7846       }

```

(End definition for `\__regex_extract:` and `\__regex_extract_aux:w`.)

#### 45.7.4 Replacement

`\__regex_replace_once:nnN`  
`\__regex_replace_once_aux:nnN`

Build the NFA and the replacement functions, then find a single match. If the match failed, simply exit the group. Otherwise, we do the replacement. Extract submatches. Compute the brace balance corresponding to replacing this match by the replacement (this depends on submatches). Prepare the replaced token list: the replacement function produces the tokens from the start of the query to the start of the match and the replacement text for this match; we need to add the tokens from the end of the match to the end of the query. Finally, store the result in the user's variable after closing the group: this step involves an additional x-expansion, and checks that braces are balanced in the final result.

```

7847 \cs_new_protected:Npn __regex_replace_once:nnN #1#2
7848   { \__regex_replace_once_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7849 \cs_new_protected:Npn __regex_replace_once_aux:nnN #1#2#3

```

```

7850    {
7851      \group_begin:
7852        \__regex_single_match:
7853        #1
7854        \exp_args:No \__regex_match:n {#3}
7855      \bool_if:NTF \g__regex_success_bool
7856      {
7857        \__regex_extract:
7858        \exp_args:No \__regex_query_set:n {#3}
7859        #2
7860        \int_set:Nn \l__regex_balance_int
7861        {
7862          \__regex_replacement_balance_one_match:n
7863          { \l__regex_zeroth_submatch_int }
7864        }
7865        \__kernel_tl_set:Nx \l__regex_internal_a_tl
7866        {
7867          \__regex_replacement_do_one_match:n
7868          { \l__regex_zeroth_submatch_int }
7869          \__regex_query_range:nn
7870          {
7871            \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
7872            { \l__regex_zeroth_submatch_int }
7873          }
7874          { \l__regex_max_pos_int }
7875        }
7876        \__regex_group_end_replace:N #3
7877      }
7878      { \group_end: }
7879    }

```

(End definition for `\__regex_replace_once:nnN` and `\__regex_replace_once_aux:nnN`.)

`\__regex_replace_all:nnN` Match multiple times, and for every match, extract submatches and additionally store the position at which the match attempt started. The entries from `\l__regex_min_submatch_int` to `\l__regex_submatch_int` hold information about submatches of every match in order; each match corresponds to `\l__regex_capturing_group_int` consecutive entries. Compute the brace balance corresponding to doing all the replacements: this is the sum of brace balances for replacing each match. Join together the replacement texts for each match (including the part of the query before the match), and the end of the query.

```

7880  \cs_new_protected:Npn \__regex_replace_all:nnN #1#2
7881  { \__regex_replace_all_aux:nnN {#1} { \__regex_replacement:n {#2} } }
7882  \cs_new_protected:Npn \__regex_replace_all_aux:nnN #1#2#3
7883  {
7884    \group_begin:
7885      \__regex_multi_match:n { \__regex_extract: }
7886      #1
7887      \exp_args:No \__regex_match:n {#3}
7888      \exp_args:No \__regex_query_set:n {#3}
7889      #2
7890      \int_set:Nn \l__regex_balance_int
7891      {
7892        0

```

```

7893   \int_step_function:nnN
7894   { \l__regex_min_submatch_int }
7895   \l__regex_capturing_group_int
7896   { \l__regex_submatch_int - 1 }
7897   \__regex_replacement_balance_one_match:n
7898   }
7899   \_kernel_tl_set:Nx \l__regex_internal_a_tl
7900   {
7901     \int_step_function:nnN
7902     { \l__regex_min_submatch_int }
7903     \l__regex_capturing_group_int
7904     { \l__regex_submatch_int - 1 }
7905     \__regex_replacement_do_one_match:n
7906     \__regex_query_range:nn
7907     \l__regex_start_pos_int \l__regex_max_pos_int
7908   }
7909   \__regex_group_end_replace:N #3
7910 }

```

(End definition for `\__regex_replace_all:nnN`.)

`\__regex_group_end_replace:N`  
`\__regex_group_end_replace_try:`  
`\__regex_group_end_replace_check:w`  
`\__regex_group_end_replace_check:n`

At this stage `\l__regex_internal_a_tl` (x-expands to the desired result). Guess from `\l__regex_balance_int` the number of braces to add before or after the result then try expanding. The simplest case is when `\l__regex_internal_a_tl` together with the braces we insert via `\prg_replicate:nn` give a balanced result, and the assignment ends at the `\if_false: { \fi: }` construction: then `\__regex_group_end_replace_check:w` sees that there is no material left and we successfully found the result. The harder case is that expanding `\l__regex_internal_a_tl` may produce extra closing braces and end the assignment early. Then we grab the remaining code using; importantly, what follows has not yet been expanded so that `\__regex_group_end_replace_check:n` grabs everything until the last brace in `\__regex_group_end_replace_try:`, letting us try again with an extra surrounding pair of braces.

```

7911 \cs_new_protected:Npn \__regex_group_end_replace:N #1
7912   {
7913     \int_set:Nn \l__regex_added_begin_int
7914     { \int_max:nn { - \l__regex_balance_int } { 0 } }
7915     \int_set:Nn \l__regex_added_end_int
7916     { \int_max:nn { \l__regex_balance_int } { 0 } }
7917     \__regex_group_end_replace_try:
7918     \int_compare:nNnT { \l__regex_added_begin_int + \l__regex_added_end_int } > 0
7919     {
7920       \msg_error:nnxxx { regex } { result-unbalanced }
7921       { replacing } { \int_use:N \l__regex_added_begin_int }
7922       { \int_use:N \l__regex_added_end_int }
7923     }
7924     \group_end:
7925     \tl_set_eq:NN #1 \g__regex_internal_tl
7926   }
7927 \cs_new_protected:Npn \__regex_group_end_replace_try:
7928   {
7929     \tex_afterassignment:D \__regex_group_end_replace_check:w
7930     \_kernel_tl_gset:Nx \g__regex_internal_tl
7931   }

```

```

7932      \prg_replicate:nn { \l__regex_added_begin_int } { { \if_false: } \fi: }
7933      \l__regex_internal_a_tl
7934      \prg_replicate:nn { \l__regex_added_end_int } { \if_false: { \fi: } }
7935      \if_false: { \fi: }
7936    }
7937  }
7938 \cs_new_protected:Npn \__regex_group_end_replace_check:w
7939  {
7940    \exp_after:wN \__regex_group_end_replace_check:n
7941    \exp_after:wN { \if_false: } \fi:
7942  }
7943 \cs_new_protected:Npn \__regex_group_end_replace_check:n #1
7944  {
7945    \tl_if_empty:nF {#1}
7946    {
7947      \int_incr:N \l__regex_added_begin_int
7948      \int_incr:N \l__regex_added_end_int
7949      \__regex_group_end_replace_try:
7950    }
7951  }

```

(End definition for `\__regex_group_end_replace:N` and others.)

#### 45.7.5 Peeking ahead

`\l__regex_peek_true_tl` True/false code arguments of `\peek_regex:nTF` or similar.

`\l__regex_peek_false_tl`

```

7952 \tl_new:N \l__regex_peek_true_tl
7953 \tl_new:N \l__regex_peek_false_tl

```

(End definition for `\l__regex_peek_true_tl` and `\l__regex_peek_false_tl`.)

`\l__regex_replacement_tl` When peeking in `\peek_regex_replace_once:nnTF` we need to store the replacement text.

```
7954 \tl_new:N \l__regex_replacement_tl
```

(End definition for `\l__regex_replacement_tl`.)

`\l__regex_input_tl` Stores each token found as `\__regex_input_item:n {⟨tokens⟩}`, where the `⟨tokens⟩` expands to the token found, as for `\tl_analysis_map_inline:nn`.

```

7955 \tl_new:N \l__regex_input_tl
7956 \cs_new_eq:NN \__regex_input_item:n ?

```

(End definition for `\l__regex_input_tl` and `\__regex_input_item:n`.)

`\peek_regex:nTF` The T and F functions just call the corresponding TF function. The four TF functions differ along two axes: whether to remove the token or not, distinguished by using `\__regex_peek_end:` or `\__regex_peek_remove_end:n` (the latter case needs an argument, as we will see), and whether the regex has to be compiled or is already in an N-type variable,

`\peek_regex_remove_once:nTF` distinguished by calling `\__regex_build_aux:Nn` or `\__regex_build_aux>NN`. The first argument of these functions is `\c_false_bool` to indicate that there should be no implicit insertion of a wildcard at the start of the pattern: otherwise the code would keep looking further into the input stream until matching the regex.

```

7957 \cs_new_protected:Npn \peek_regex:nTF #1
7958  {

```

```

7959     \_\_regex\_peek:nnTF
7960     { \_\_regex\_build\_aux:Nn \c\_false\_bool {\#1} }
7961     { \_\_regex\_peek\_end: }
7962   }
7963 \cs\_new\_protected:Npn \peek\_regex:nT #1#2
7964   { \peek\_regex:nTF {\#1} {\#2} { } }
7965 \cs\_new\_protected:Npn \peek\_regex:nF #1 { \peek\_regex:nTF {\#1} { } }
7966 \cs\_new\_protected:Npn \peek\_regex:NTF #1
7967   {
7968     \_\_regex\_peek:nnTF
7969     { \_\_regex\_build\_aux:NN \c\_false\_bool #1 }
7970     { \_\_regex\_peek\_end: }
7971   }
7972 \cs\_new\_protected:Npn \peek\_regex:NT #1#2
7973   { \peek\_regex:NTF #1 {\#2} { } }
7974 \cs\_new\_protected:Npn \peek\_regex:NF #1 { \peek\_regex:NTF {\#1} { } }
7975 \cs\_new\_protected:Npn \peek\_regex\_remove\_once:nTF #1
7976   {
7977     \_\_regex\_peek:nnTF
7978     { \_\_regex\_build\_aux:Nn \c\_false\_bool {\#1} }
7979     { \_\_regex\_peek\_remove\_end:n {\##1} }
7980   }
7981 \cs\_new\_protected:Npn \peek\_regex\_remove\_once:nT #1#2
7982   { \peek\_regex\_remove\_once:nTF {\#1} {\#2} { } }
7983 \cs\_new\_protected:Npn \peek\_regex\_remove\_once:nF #1
7984   { \peek\_regex\_remove\_once:nTF {\#1} { } }
7985 \cs\_new\_protected:Npn \peek\_regex\_remove\_once:NTF #1
7986   {
7987     \_\_regex\_peek:nnTF
7988     { \_\_regex\_build\_aux:NN \c\_false\_bool #1 }
7989     { \_\_regex\_peek\_remove\_end:n {\##1} }
7990   }
7991 \cs\_new\_protected:Npn \peek\_regex\_remove\_once:NT #1#2
7992   { \peek\_regex\_remove\_once:NTF #1 {\#2} { } }
7993 \cs\_new\_protected:Npn \peek\_regex\_remove\_once:NF #1
7994   { \peek\_regex\_remove\_once:NTF #1 { } }

```

(End definition for \peek\\_regex:nTF and others. These functions are documented on page 194.)

```
\_\_regex\_peek:nnTF
\_\_regex\_peek\_aux:nnTF
```

Store the user's true/false codes (plus \group\\_end:) into two token lists. Then build the automaton with #1, without submatch tracking, and aiming for a single match. Then start matching by setting up a few variables like for any regex matching like \regex\\_match:nnTF, with the addition of \l\\_regex\\_input\\_tl that keeps track of the tokens seen, to reinsert them at the end. Instead of \tl\\_analysis\\_map\\_inline:nn on the input, we call \peek\\_analysis\\_map\\_inline:n to go through tokens in the input stream. Since \\_\\_regex\\_match\\_one\\_token:nnN calls \\_\\_regex\\_maplike\\_break: we need to catch that and break the \peek\\_analysis\\_map\\_inline:n loop instead.

```

7995 \cs\_new\_protected:Npn \_\_regex\_peek:nnTF #1
7996   {
7997     \_\_regex\_peek\_aux:nnTF
7998   {
7999     \_\_regex\_disable\_submatches:
8000     #1
8001   }
```

```

8002    }
8003 \cs_new_protected:Npn \__regex_peek_aux:nnTF #1#2#3#4
8004 {
8005     \group_begin:
8006     \tl_set:Nn \l__regex_peek_true_tl { \group_end: #3 }
8007     \tl_set:Nn \l__regex_peek_false_tl { \group_end: #4 }
8008     \__regex_single_match:
8009     #1
8010     \__regex_match_init:
8011     \tl_build_clear:N \l__regex_input_tl
8012     \__regex_match_once_init:
8013     \peek_analysis_map_inline:n
8014     {
8015         \tl_build_put_right:Nn \l__regex_input_tl
8016         { \__regex_input_item:n {##1} }
8017         \__regex_match_one_token:nnN {##1} {##2} ##3
8018         \use_none:nnn
8019         \prg_break_point:Nn \__regex_maplike_break:
8020         { \peek_analysis_map_break:n {#2} }
8021     }
8022 }
```

(End definition for `\__regex_peek:nnTF` and `\__regex_peek_aux:nnTF`.)

`\__regex_peek_end:`  
`\__regex_peek_remove_end:n`

Once the regex matches (or permanently fails to match) we call `\__regex_peek_end:`, or `\__regex_peek_remove_end:n` with argument the last token seen. For `\peek_regex:nTF` we reinsert tokens seen by calling `\__regex_peek_reinsert:N` regardless of the result of the match. For `\peek_regex_remove_once:nTF` we reinsert the tokens seen only if the match failed; otherwise we just reinsert the tokens `#1`, with one expansion. To be more precise, `#1` consists of tokens that o-expand and x-expand to the last token seen, for example it is `\exp_not:N (cs)` for a control sequence. This means that just doing `\exp_after:wN \l__regex_peek_true_tl #1` would be unsafe because the expansion of `(cs)` would be suppressed.

```

8023 \cs_new_protected:Npn \__regex_peek_end:
8024 {
8025     \bool_if:NTF \g__regex_success_bool
8026     { \__regex_peek_reinsert:N \l__regex_peek_true_tl }
8027     { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
8028 }
8029 \cs_new_protected:Npn \__regex_peek_remove_end:n #1
8030 {
8031     \bool_if:NTF \g__regex_success_bool
8032     { \exp_args:NNo \use:nn \l__regex_peek_true_tl {#1} }
8033     { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
8034 }
```

(End definition for `\__regex_peek_end:` and `\__regex_peek_remove_end:n`.)

`\__regex_peek_reinsert:N`  
`\__regex_reinsert_item:n`

Insert the true/false code `#1`, followed by the tokens found, which were stored in `\l__regex_input_tl`. For this, loop through that token list using `\__regex_reinsert_item:n`, which expands `#1` once to get a single token, and jumps over it to expand what follows, with suitable `\exp:w` and `\exp_end:..`. We cannot just use `\use:e` on the whole token list because the result may be unbalanced, which would stop the primitive prematurely, or let it continue beyond where we would like.

```

8035 \cs_new_protected:Npn \__regex_peek_reinsert:N #1
8036 {
8037     \tl_build_end:N \l__regex_input_tl
8038     \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
8039     \exp_after:wN #1 \exp:w \l__regex_input_tl \exp_end:
8040 }
8041 \cs_new_protected:Npn \__regex_reinsert_item:n #1
8042 {
8043     \exp_after:wN \exp_after:wN
8044     \exp_after:wN \exp_end:
8045     \exp_after:wN \exp_after:wN
8046     #1
8047     \exp:w
8048 }

```

(End definition for `\__regex_peek_reinsert:N` and `\__regex_reinsert_item:n`.)

`\peek_regex_replace_once:nn`  
`\peek_regex_replace_once:nnTF`  
`\peek_regex_replace_once:Nn`  
`\peek_regex_replace_once:NnTF`

Similar to `\peek_regex:nTF` above.

```

8049 \cs_new_protected:Npn \peek_regex_replace_once:nnTF #1
8050 { \__regex_peek_replace:nnTF { \__regex_build_aux:Nn \c_false_bool {#1} } }
8051 \cs_new_protected:Npn \peek_regex_replace_once:nnT #1#2#3
8052 { \peek_regex_replace_once:nnTF {#1} {#2} {#3} { } }
8053 \cs_new_protected:Npn \peek_regex_replace_once:nnF #1#2
8054 { \peek_regex_replace_once:nnTF {#1} {#2} { } }
8055 \cs_new_protected:Npn \peek_regex_replace_once:nn #1#2
8056 { \peek_regex_replace_once:nnTF {#1} {#2} { } { } }
8057 \cs_new_protected:Npn \peek_regex_replace_once:NnTF #1
8058 { \__regex_peek_replace:nnTF { \__regex_build_aux:NN \c_false_bool #1 } }
8059 \cs_new_protected:Npn \peek_regex_replace_once:NnT #1#2#3
8060 { \peek_regex_replace_once:NnTF {#1} {#2} {#3} { } }
8061 \cs_new_protected:Npn \peek_regex_replace_once:NnF #1#2
8062 { \peek_regex_replace_once:NnTF {#1} {#2} { } }
8063 \cs_new_protected:Npn \peek_regex_replace_once:Nn #1#2
8064 { \peek_regex_replace_once:NnTF {#1} {#2} { } { } }

```

(End definition for `\peek_regex_replace_once:nnTF` and `\peek_regex_replace_once:NnTF`. These functions are documented on page 195.)

`\__regex_peek_replace:nnTF`

Same as `\__regex_peek:nnTF` (used for `\peek_regex:nTF` above), but without disabling submatches, and with a different end. The replacement text #2 is stored, to be analyzed later.

```

8065 \cs_new_protected:Npn \__regex_peek_replace:nnTF #1#2
8066 {
8067     \tl_set:Nn \l__regex_replacement_tl {#2}
8068     \__regex_peek_aux:nnTF {#1} { \__regex_peek_replace_end: }
8069 }

```

(End definition for `\__regex_peek_replace:nnTF`.)

`\__regex_peek_replace_end:`

If the match failed `\__regex_peek_reinsert:N` reinserts the tokens found. Otherwise, finish storing the submatch information using `\__regex_extract:`, and store the input into `\toks`. Redefine a few auxiliaries to change slightly their expansion behaviour as explained below. Analyse the replacement text with `\__regex_replacement:n`, which as usual defines `\__regex_replacement_do_one_match:n` to insert the tokens from the

start of the match attempt to the beginning of the match, followed by the replacement text. The `\use:x` expands for instance the trailing `\__regex_query_range:nn` down to a sequence of `\__regex_reinsert_item:n {<tokens>}` where `<tokens>` o-expand to a single token that we want to insert. After x-expansion, `\use:x` does `\use:n`, so we have `\exp_after:wN \l__regex_peek_true_tl \exp:w ... \exp_end:`. This is set up such as to obtain `\l__regex_peek_true_tl` followed by the replaced tokens (possibly unbalanced) in the input stream.

```

8070 \cs_new_protected:Npn \__regex_peek_replace_end:
8071 {
8072     \bool_if:NTF \g__regex_success_bool
8073     {
8074         \__regex_extract:
8075         \__regex_query_set_from_input_tl:
8076         \cs_set_eq:NN \__regex_replacement_put:n \__regex_peek_replacement_put:n
8077         \cs_set_eq:NN \__regex_replacement_put_submatch_aux:n
8078             \__regex_peek_replacement_put_submatch_aux:n
8079         \cs_set_eq:NN \__regex_input_item:n \__regex_reinsert_item:n
8080         \cs_set_eq:NN \__regex_replacement_exp_not:N \__regex_peek_replacement_token:n
8081         \cs_set_eq:NN \__regex_replacement_exp_not:V \__regex_peek_replacement_var:N
8082         \exp_args:No \__regex_replacement:n { \l__regex_replacement_tl }
8083         \use:x
8084         {
8085             \exp_not:n { \exp_after:wN \l__regex_peek_true_tl \exp:w }
8086             \__regex_replacement_do_one_match:n
8087                 { \l__regex_zeroth_submatch_int }
8088             \__regex_query_range:nn
8089             {
8090                 \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
8091                     { \l__regex_zeroth_submatch_int }
8092             }
8093             { \l__regex_max_pos_int }
8094             \exp_end:
8095         }
8096     }
8097     { \__regex_peek_reinsert:N \l__regex_peek_false_tl }
8098 }
```

(End definition for `\__regex_peek_replace_end:`)

`\__regex_query_set_from_input_tl:`  
`\__regex_query_set_item:n`: The input was stored into `\l__regex_input_tl` as successive items `\__regex_input_item:n {<tokens>}`. Store that in successive `\toks`. It's not clear whether the empty entries before and after are both useful.

```

8099 \cs_new_protected:Npn \__regex_query_set_from_input_tl:
8100 {
8101     \tl_build_end:N \l__regex_input_tl
8102     \int_zero:N \l__regex_curr_pos_int
8103     \cs_set_eq:NN \__regex_input_item:n \__regex_query_set_item:n
8104     \__regex_query_set_item:n { }
8105     \l__regex_input_tl
8106     \__regex_query_set_item:n { }
8107     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
8108 }
8109 \cs_new_protected:Npn \__regex_query_set_item:n #1
```

```

8110    {
8111      \int_incr:N \l__regex_curr_pos_int
8112      \__regex_toks_set:Nn \l__regex_curr_pos_int { \__regex_input_item:n {#1} }
8113    }

```

(End definition for `\__regex_query_set_from_input_tl: and \__regex_query_set_item:n`)

`\__regex_peek_replacement_put:n`

While building the replacement function `\__regex_replacement_do_one_match:n`, we often want to put simple material, given as #1, whose x-expansion o-expands to a single token. Normally we can just add the token to `\l__regex_build_tl`, but for `\peek_regex_replace_once:nnTF` we eventually want to do some strange expansion that is basically using `\exp_after:wN` to jump through numerous tokens (we cannot use x-expansion like for `\regex_replace_once:nnNTF` because it is ok for the result to be unbalanced since we insert it in the input stream rather than storing it. When within a csname we don't do any such shenanigan because `\cs:w ... \cs_end:` does all the expansion we need.

```

8114 \cs_new_protected:Npn \__regex_peek_replacement_put:n #1
8115  {
8116    \if_case:w \l__regex_replacement_csnames_int
8117      \tl_build_put_right:Nn \l__regex_build_tl
8118      { \exp_not:N \__regex_reinsert_item:n {#1} }
8119    \else:
8120      \tl_build_put_right:Nn \l__regex_build_tl {#1}
8121    \fi:
8122  }

```

(End definition for `\__regex_peek_replacement_put:n`)

`\__regex_peek_replacement_token:n`

When hit with `\exp:w, \__regex_peek_replacement_token:n {\langle token \rangle}` stops `\exp_end:` and does `\exp_after:wN \langle token \rangle \exp:w` to continue expansion after it.

```

8123 \cs_new_protected:Npn \__regex_peek_replacement_token:n #1
8124  { \exp_after:wN \exp_end: \exp_after:wN #1 \exp:w }

```

(End definition for `\__regex_peek_replacement_token:n`)

`\__regex_peek_replacement_put_submatch_aux:n`

While analyzing the replacement we also have to insert submatches found in the query. Since query items `\__regex_input_item:n {\langle tokens \rangle}` expand correctly only when surrounded by `\exp:w ... \exp_end:`, and since these expansion controls are not there within csnames (because `\cs:w ... \cs_end:` make them unnecessary in most cases), we have to put `\exp:w` and `\exp_end:` by hand here.

```

8125 \cs_new_protected:Npn \__regex_peek_replacement_put_submatch_aux:n #1
8126  {
8127    \if_case:w \l__regex_replacement_csnames_int
8128      \tl_build_put_right:Nn \l__regex_build_tl
8129      { \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
8130    \else:
8131      \tl_build_put_right:Nn \l__regex_build_tl
8132      { \exp:w \__regex_query_submatch:n { \int_eval:n { #1 + ##1 } } \exp_end: }
8133    \fi:
8134  }

```

(End definition for `\__regex_peek_replacement_put_submatch_aux:n`)

\\_regex\\_peek\\_replacement\\_var:N This is used for \u outside csnames. It makes sure to continue expansion with \exp:w before expanding the variable #1 and stopping the \exp:w that precedes.

```

8135 \cs_new_protected:Npn \_regex_peek_replacement_var:N #1
8136   {
8137     \exp_after:wN \exp_last_unbraced:NV
8138     \exp_after:wN \exp_end:
8139     \exp_after:wN #1
8140     \exp:w
8141   }

```

(End definition for \\_regex\_peek\_replacement\_var:N.)

## 45.8 Messages

Messages for the preparsing phase.

```

8142 \use:x
8143   {
8144     \msg_new:nnn { regex } { trailing-backslash }
8145     {
8146       Trailing~'\iow_char:N\\'~in~regex~or~replacement. }
8147     \msg_new:nnn { regex } { x-missing-rbrace }
8148     {
8149       Missing-brace~'\iow_char:N\\'~in~regex~
8150       '...~\iow_char:N\\x\iow_char:N\\{...##1'.
8151     }
8152     \msg_new:nnn { regex } { x-overflow }
8153     {
8154       Character~code~##1~too~large~in~
8155       '\iow_char:N\\x\iow_char:N\\{##2\iow_char:N\\}~regex.
8156     }

```

Invalid quantifier.

```

8157 \msg_new:nnnn { regex } { invalid-quantifier }
8158   {
8159     Braced-quantifier~'#1'~may~not~be~followed~by~'#2'. }
8160     The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
8161     The~only~valid~quantifiers~are~'*',~'?',~'+'~,~'{<int>}',~
8162     '{<min>},~and~'{<min>,<max>}',~optionally~followed~by~'?'.
8163   }

```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

8164 \msg_new:nnnn { regex } { missing-rbrack }
8165   {
8166     Missing-right~bracket~inserted~in~regular~expression. }
8167     LaTeX~was~given~a~regular~expression~where~a~character~class~
8168     was~started~with~'[',~but~the~matching~']'~is~missing.
8169   }
8170 \msg_new:nnnn { regex } { missing-rparen }
8171   {
8172     Missing-right~
8173     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
8174     inserted~in~regular~expression.

```

```

8175     }
8176     {
8177         LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
8178         more-left-parentheses-than-right-parentheses.
8179     }
8180 \msg_new:nnnn { regex } { extra-rparen }
8181     { Extra-right-parenthesis-ignored-in-regular-expression. }
8182     {
8183         LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
8184         was-open.-The-parenthesis-will-be-ignored.
8185     }

```

Some escaped alphanumerics are not allowed everywhere.

```

8186 \msg_new:nnnn { regex } { bad-escape }
8187     {
8188         Invalid-escape-'\\iow_char:N\\#1'-
8189         \\_regex_if_in_cs:TF { within-a-control-sequence. }
8190         {
8191             \\_regex_if_in_class:TF
8192                 { in-a-character-class. }
8193                 { following-a-category-test. }
8194         }
8195     }
8196     {
8197         The-escape-sequence-'\\iow_char:N\\#1'-may-not-appear-
8198         \\_regex_if_in_cs:TF
8199         {
8200             within-a-control-sequence-test-introduced-by-
8201             '\\iow_char:N\\c\\iow_char:N\\{'.
8202         }
8203         {
8204             \\_regex_if_in_class:TF
8205                 { within-a-character-class- }
8206                 { following-a-category-test-such-as-'\\iow_char:N\\cL'-' }
8207             because-it-does-not-match-exactly-one-character.
8208         }
8209     }

```

Range errors.

```

8210 \msg_new:nnnn { regex } { range-missing-end }
8211     { Invalid-end-point-for-range-'#1-#2'-in-character-class. }
8212     {
8213         The-end-point-'#2'-of-the-range-'#1-#2'-may-not-serve-as-an-
8214         end-point-for-a-range:-alphanumeric-characters-should-not-be-
8215         escaped,-and-non-alphanumeric-characters-should-be-escaped.
8216     }
8217 \msg_new:nnnn { regex } { range-backwards }
8218     { Range-'[#1-#2]'-out-of-order-in-character-class. }
8219     {
8220         In-ranges-of-characters-'[x-y]'-appearing-in-character-classes,-
8221         the-first-character-code-must-not-be-larger-than-the-second.-~
8222         Here,-'#1'-has-character-code-\int_eval:n {'#1},-while-
8223         '#2'-has-character-code-\int_eval:n {'#2}.
8224     }

```

Errors related to \c and \u.

```
8225 \msg_new:nnnn { regex } { c-bad-mode }
8226   { Invalid-nested-'\\iow_char:N\\c'~escape-in-regular-expression. }
8227   {
8228     The-'\\iow_char:N\\c'~escape~cannot~be~used~within~
8229     a~control~sequence~test~'\\iow_char:N\\c{...}'~
8230     nor~another~category~test.~
8231     To~combine~several~category~tests,~use~'\\iow_char:N\\c[...]'.
8232   }
8233 \msg_new:nnnn { regex } { c-C-invalid }
8234   { '\\iow_char:N\\cC'~should~be~followed~by~'.~or~'(~,~not~'#1'. }
8235   {
8236     The-'\\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
8237     control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
8238     It~only~makes~sense~to~follow~it~by~'.~or~by~a~group.
8239   }
8240 \msg_new:nnnn { regex } { cu-lbrace }
8241   { Left~braces~must~be~escaped~in~'\\iow_char:N\\#1{...}'. }
8242   {
8243     Constructions~such~as~'\\iow_char:N\\#1{...}\\iow_char:N\\{...}'~are~
8244     not~allowed~and~should~be~replaced~by~
8245     '\\iow_char:N\\#1{...}\\token_to_str:N\\{...}'.
8246   }
8247 \msg_new:nnnn { regex } { c-lparen-in-class }
8248   { Catcode~test~cannot~apply~to~group~in~character~class }
8249   {
8250     Construction~such~as~'\\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
8251     class~' [...] '~because~classes~do~not~match~multiple~characters~at~once.
8252   }
8253 \msg_new:nnnn { regex } { c-missing-rbrace }
8254   { Missing~right~brace~inserted~for~'\\iow_char:N\\c'~escape. }
8255   {
8256     LaTeX~was~given~a~regular~expression~where~a~
8257     '\\iow_char:N\\c\\iow_char:N\\{...}'~construction~was~not~ended~
8258     with~a~closing~brace~'\\iow_char:N\\}'.
8259   }
8260 \msg_new:nnnn { regex } { c-missing-rbrack }
8261   { Missing~right~bracket~inserted~for~'\\iow_char:N\\c'~escape. }
8262   {
8263     A~construction~'\\iow_char:N\\c[...]'~appears~in~a~
8264     regular~expression,~but~the~closing~']'~is~not~present.
8265   }
8266 \msg_new:nnnn { regex } { c-missing-category }
8267   { Invalid~character~'#1'~following~'\\iow_char:N\\c'~escape. }
8268   {
8269     In~regular~expressions,~the~'\\iow_char:N\\c'~escape~sequence~
8270     may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
8271     capital~letter~representing~a~character~category,~namely~
8272     one~of~'ABCDELMPSTU'.
8273   }
8274 \msg_new:nnnn { regex } { c-trailing }
8275   { Trailing~category~code~escape~'\\iow_char:N\\c'... }
8276   {
8277     A~regular~expression~ends~with~'\\iow_char:N\\c'~followed~
```

```

8278     by~a~letter.~It~will~be~ignored.
8279   }
8280 \msg_new:nnnn { regex } { u-missing-lbrace }
8281   { Missing-left-brace-following~'\iow_char:N\\u'~escape. }
8282   {
8283     The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
8284     a-brace~group~with~the~name~of~the~variable~to~use.
8285   }
8286 \msg_new:nnnn { regex } { u-missing-rbrace }
8287   { Missing-right-brace~inserted~for~'\iow_char:N\\u'~escape. }
8288   {
8289     LaTeX
8290     \str_if_eq:eeTF { } {#2}
8291       { reached~the~end~of~the~string~}
8292       { encountered~an~escaped~alphanumeric~character~'\iow_char:N\\#2'~}
8293     when~parsing~the~argument~of~an~
8294     '\iow_char:N\\u\iow_char:N\\{...\\}'~escape.
8295   }

```

Errors when encountering the POSIX syntax [:...:].

```

8296 \msg_new:nnnn { regex } { posix-unsupported }
8297   { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
8298   {
8299     The~'[.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
8300     in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
8301     Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
8302   }
8303 \msg_new:nnnn { regex } { posix-unknown }
8304   { POSIX~class~'[:#1:]'~unknown. }
8305   {
8306     '[:#1:]'~is~not~among~the~known~POSIX~classes~
8307     '[:alnum:]',~'[[:alpha:]]',~'[[:ascii:]]',~'[[:blank:]]',~
8308     '[[:cntrl:]]',~'[[:digit:]]',~'[[:graph:]]',~'[[:lower:]]',~
8309     '[[:print:]]',~'[[:punct:]]',~'[[:space:]]',~'[[:upper:]]',~
8310     '[[:word:]]',~and~'[[:xdigit:]]'.
8311   }
8312 \msg_new:nnnn { regex } { posix-missing-close }
8313   { Missing~closing~':]'~for~POSIX~class. }
8314   { The~POSIX~syntax~'#1'~must~be~followed~by~':]',~not~'#2'. }

```

In various cases, the result of a `\l3regex` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

8315 \msg_new:nnnn { regex } { result-unbalanced }
8316   { Missing~brace~inserted~when~#1. }
8317   {
8318     LaTeX~was~asked~to~do~some~regular~expression~operation,~
8319     and~the~resulting~token~list~would~not~have~the~same~number~
8320     of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
8321     #2~left,~#3~right.
8322   }

```

Error message for unknown options.

```

8323 \msg_new:nnnn { regex } { unknown-option }
8324   { Unknown~option~'#1'~for~regular~expressions. }

```

```

8325  {
8326      The~only~available~option~is~'case-insensitive',~toggled~by~
8327      '(?i)'~and~'(?-i)'.
8328  }
8329 \msg_new:nnnn { regex } { special-group-unknown }
8330   { Unknown~special~group-'#1~...~in~a~regular~expression. }
8331   {
8332       The~only~valid~constructions~starting~with~'(?'~are~
8333       '(?:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
8334   }

Errors in the replacement text.

8335 \msg_new:nnnn { regex } { replacement-c }
8336   { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
8337   {
8338       In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
8339       can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~
8340       or~a~brace~group,~not~by~'#1'.
8341   }
8342 \msg_new:nnnn { regex } { replacement-u }
8343   { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
8344   {
8345       In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
8346       must~be~followed~by~a~brace~group~holding~the~name~of~the~
8347       variable~to~use.
8348   }
8349 \msg_new:nnnn { regex } { replacement-g }
8350   {
8351       Missing~brace~for~the~'\iow_char:N\\g'~construction~
8352       in~a~replacement~text.
8353   }
8354   {
8355       In~the~replacement~text~for~a~regular~expression~search,~
8356       submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
8357       or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
8358   }
8359 \msg_new:nnnn { regex } { replacement-catcode-end }
8360   {
8361       Missing~character~for~the~'\iow_char:N\\c<category><character>'~
8362       construction~in~a~replacement~text.
8363   }
8364   {
8365       In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
8366       can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
8367       the~character~category.~Then,~a~character~must~follow.~LaTeX~
8368       reached~the~end~of~the~replacement~when~looking~for~that.
8369   }
8370 \msg_new:nnnn { regex } { replacement-catcode-escaped }
8371   {
8372       Escaped~letter~or~digit~after~category~code~in~replacement~text.
8373   }
8374   {
8375       In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
8376       can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
8377       the~character~category.~Then,~a~character~must~follow,~not~

```

```

8378      '\iow_char:N\\#2'.
8379    }
8380 \msg_new:nnnn { regex } { replacement-catcode-in-cs }
8381  {
8382    Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
8383    '\iow_char:N\\c\\{...}'~in~a~replacement~text.
8384  }
8385  {
8386    In~a~replacement~text,~the~category~codes~of~the~argument~of~
8387    '\iow_char:N\\c\\{...}'~are~ignored~when~building~the~control~
8388    sequence~name.
8389  }
8390 \msg_new:nnnn { regex } { replacement-null-space }
8391  {
8392    TeX~cannot~build~a~space~token~with~character~code~0. }
8393  {
8394    You~asked~for~a~character~token~with~category~space,~
8395    and~character~code~0,~for~instance~through~
8396    '\iow_char:N\\cS\iow_char:N\\x00'.~
8397    This~specific~case~is~impossible~and~will~be~replaced~
8398    by~a~normal~space.
8399  }
8400 \msg_new:nnnn { regex } { replacement-missing-rbrace }
8401  {
8402    Missing~right~brace~inserted~in~replacement~text. }
8403  {
8404    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8405    missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
8406  }
8407 \msg_new:nnnn { regex } { replacement-missing-rparen }
8408  {
8409    Missing~right~parenthesis~inserted~in~replacement~text. }
8410  {
8411    There~ \int_compare:nTF { #1 = 1 } { was } { were } ~ #1~
8412    missing~right~
8413    \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
8414 \msg_new:nnn { regex } { submatch-too-big }
8415  {
8416    Submatch~#1~used~but~regex~only~has~#2~group(s) }

Some escaped alphanumerics are not allowed everywhere.

8417 \msg_new:nnnn { regex } { backwards-quantifier }
8418  {
8419    Quantifier~"#{1,#2}"~is~backwards. }
8420  {
8421    The~values~given~in~a~quantifier~must~be~in~order. }

Used in user commands, and when showing a regex.

8422 \msg_new:nnnn { regex } { case-odd }
8423  {
8424    #1~with~odd~number~of~items }
8425  {
8426    There~must~be~a~#2~part~for~each~regex:~
8427    found~odd~number~of~items~(#3)~in\\
8428    \iow_indent:n {#4}
8429  }
8430 \msg_new:nnn { regex } { show }
8431  {
8432    >~Compiled~regex~
8433    \tl_if_empty:nTF {#1} { variable~ #2 } { {#1} } :
8434    #3

```

```

8429     }
8430 \prop_gput:Nnn \g_msg_module_name_prop { regex } { LaTeX }
8431 \prop_gput:Nnn \g_msg_module_type_prop { regex } { }

```

\\_\\_regex\\_msg\\_repeated:nnN

This is not technically a message, but seems related enough to go there. The arguments are: #1 is the minimum number of repetitions; #2 is the number of allowed extra repetitions (-1 for infinite number), and #3 tells us about lazyness.

```

8432 \cs_new:Npn \_\_regex_msg_repeated:nnN #1#2#3
8433 {
8434     \str_if_eq:eeF { #1 #2 } { 1 0 }
8435     {
8436         , ~ repeated ~
8437         \int_case:nnF {#2}
8438         {
8439             { -1 } { #1-or-more-times,\bool_if:NTF #3 { lazy } { greedy } }
8440             { 0 } { #1-times }
8441         }
8442         {
8443             between~#1-and~\int_eval:n {#1+#2}-times,~
8444             \bool_if:NTF #3 { lazy } { greedy }
8445         }
8446     }
8447 }

```

(End definition for \\_\\_regex\\_msg\\_repeated:nnN.)

## 45.9 Code for tracing

There is a more extensive implementation of tracing in the l3trial package l3trace. Function names are a bit different but could be merged.

\\_\\_regex\\_trace\\_push:nnN  
\\_\\_regex\\_trace\\_pop:nnN  
\\_\\_regex\\_trace:nnx

Here #1 is the module name (regex) and #2 is typically 1. If the module's current tracing level is less than #2 show nothing, otherwise write #3 to the terminal.

```

8448 \cs_new_protected:Npn \_\_regex_trace_push:nnN #1#2#3
8449     { \_\_regex_trace:nnx {#1} {#2} { entering~ \token_to_str:N #3 } }
8450 \cs_new_protected:Npn \_\_regex_trace_pop:nnN #1#2#3
8451     { \_\_regex_trace:nnx {#1} {#2} { leaving~ \token_to_str:N #3 } }
8452 \cs_new_protected:Npn \_\_regex_trace:nnx #1#2#3
8453 {
8454     \int_compare:nNnF
8455         { \int_use:c { g\_regex_trace_#1_int } } < {#2}
8456         { \iow_term:x { Trace:~#3 } }
8457 }

```

(End definition for \\_\\_regex\\_trace\\_push:nnN, \\_\\_regex\\_trace\\_pop:nnN, and \\_\\_regex\\_trace:nnx.)

\g\\_regex\\_trace\\_regex\\_int

No tracing when that is zero.

```
8458 \int_new:N \g\_regex_trace_regex_int
```

(End definition for \g\\_regex\\_trace\\_regex\\_int.)

\\_\\_regex\\_trace\\_states:n This function lists the contents of all states of the NFA, stored in \toks from 0 to \l\\_\\_regex\\_max\\_state\\_int (excluded).

```
8459 \cs_new_protected:Npn \_\_regex_trace_states:n #1
8460   {
8461     \int_step_inline:nnn
8462       \l\_\_regex_min_state_int
8463       { \l\_\_regex_max_state_int - 1 }
8464       {
8465         \_\_regex_trace:nnx { regex } {#1}
8466         { \iow_char:N \toks ##1 = { \_\_regex_toks_use:w ##1 } }
8467       }
8468   }
```

(End definition for \\_\\_regex\\_trace\\_states:n.)

```
8469 </package>
```

# Chapter 46

## I3prg implementation

The following test files are used for this code: *m3prg001.lvt, m3prg002.lvt, m3prg003.lvt.*

8470 `(*package)`

### 46.1 Primitive conditionals

`\if_bool:N` Those two primitive TeX conditionals are synonyms. `\if_bool:N` is defined in I3basics, as it's needed earlier to define quark test functions.  
`\if_predicate:w`

8471 `\cs_new_eq:NN \if_predicate:w \tex_ifodd:D`

(End definition for `\if_bool:N` and `\if_predicate:w`. These functions are documented on page 70.)

### 46.2 Defining a set of conditional functions

`\prg_set_conditional:Npnn` These are all defined in I3basics, as they are needed “early”. This is just a reminder!

`\prg_new_conditional:Npnn`  
`\prg_set_protected_conditional:Npnn`  
`\prg_new_protected_conditional:Npnn`  
`\prg_set_conditional:Nnn`  
`\prg_new_conditional:Nnn`  
`\prg_set_protected_conditional:Nnn`  
`\prg_new_protected_conditional:Nnn`

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 62.)

### 46.3 The boolean data type

8472 `(@@=bool)`

Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

8473 `\cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }`  
8474 `\cs_generate_variant:Nn \bool_new:N { c }`

(End definition for `\bool_new:N`. This function is documented on page 65.)

`\bool_const:Nn` A merger between `\tl_const:Nn` and `\bool_set:Nn`.

8475 `\cs_new_protected:Npn \bool_const:Nn #1#2`  
8476  `{`  
8477  `\__kernel_chk_if_free_cs:N #1`  
8478  `\tex_global:D \tex_chardef:D #1 = \bool_if_p:n {#2}`  
8479  `}`  
8480 `\cs_generate_variant:Nn \bool_const:Nn { c }`

(End definition for `\bool_const:Nn`. This function is documented on page 65.)

`\bool_set_true:N` Setting is already pretty easy. When `check-declarations` is active, the definitions are patched to make sure the boolean exists. This is needed because booleans are not based on token lists nor on TeX registers.

```
8481 \cs_new_protected:Npn \bool_set_true:N #1
8482   { \cs_set_eq:NN #1 \c_true_bool }
8483 \cs_new_protected:Npn \bool_set_false:N #1
8484   { \cs_set_eq:NN #1 \c_false_bool }
8485 \cs_new_protected:Npn \bool_gset_true:N #1
8486   { \cs_gset_eq:NN #1 \c_true_bool }
8487 \cs_new_protected:Npn \bool_gset_false:N #1
8488   { \cs_gset_eq:NN #1 \c_false_bool }
8489 \cs_generate_variant:Nn \bool_set_true:N { c }
8490 \cs_generate_variant:Nn \bool_set_false:N { c }
8491 \cs_generate_variant:Nn \bool_gset_true:N { c }
8492 \cs_generate_variant:Nn \bool_gset_false:N { c }
```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 65.)

`\bool_set_eq:NN` The usual copy code. While it would be cleaner semantically to copy the `\cs_set_eq:NN` family of functions, we copy `\tl_set_eq:NN` because that has the correct checking code.

```
8493 \cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN
8494 \cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
8495 \cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
8496 \cs_generate_variant:Nn \bool_gset_eq:NN { Nc, cN, cc }
```

(End definition for `\bool_set_eq:NN` and `\bool_gset_eq:NN`. These functions are documented on page 65.)

`\bool_set:Nn` This function evaluates a boolean expression and assigns the first argument the meaning `\c_true_bool` or `\c_false_bool`. Again, we include some checking code. It is important to evaluate the expression before applying the `\chardef` primitive, because that primitive sets the left-hand side to `\scan_stop:` before looking for the right-hand side.

```
8497 \cs_new_protected:Npn \bool_set:Nn #1#
8498   {
8499     \exp_last_unbraced:NNNF
8500     \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8501   }
8502 \cs_new_protected:Npn \bool_gset:Nn #1#
8503   {
8504     \exp_last_unbraced:NNNNF
8505     \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
8506   }
8507 \cs_generate_variant:Nn \bool_set:Nn { c }
8508 \cs_generate_variant:Nn \bool_gset:Nn { c }
```

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 65.)

## 46.4 Internal auxiliaries

\q\_\_bool\_recursion\_tail Internal recursion quarks.

```

8509 \quark_new:N \q__bool_recursion_tail
8510 \quark_new:N \q__bool_recursion_stop

```

(End definition for \q\_\_bool\_recursion\_tail and \q\_\_bool\_recursion\_stop.)

\_\_bool\_use\_i\_delimit\_by\_q\_recursion\_stop:nw Functions to gobble up to a quark.

```

8511 \cs_new:Npn \__bool_use_i_delimit_by_q_recursion_stop:nw
8512 #1 #2 \q__bool_recursion_stop {#1}

```

(End definition for \\_\_bool\_use\_i\_delimit\_by\_q\_recursion\_stop:nw.)

\\_\_bool\_if\_recursion\_tail\_stop\_do:nn Functions to query recursion quarks.

```

8513 \__kernel_quark_new_test:N \__bool_if_recursion_tail_stop_do:nn

```

(End definition for \\_\_bool\_if\_recursion\_tail\_stop\_do:nn.)

\bool\_if\_p:N Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

\bool\_if\_p:c

```

8514 \prg_new_conditional:Npnm \bool_if:N #1 { p , T , F , TF }
8515 {
8516     \if_bool:N #1
8517         \prg_return_true:
8518     \else:
8519         \prg_return_false:
8520     \fi:
8521 }
8522 \prg_generate_conditional_variant:Nnn \bool_if:N { c } { p , T , F , TF }

```

(End definition for \bool\_if:NTF. This function is documented on page 65.)

\bool\_to\_str:N Expands to true or false with category code letter.

\bool\_to\_str:c

```

8523 \cs_new:Npn \bool_to_str:N #1 { \bool_if:NTF #1 { true } { false } }
8524 \cs_generate_variant:Nn \bool_to_str:N { c }
8525 \cs_new:Npn \bool_to_str:n #1 { \bool_if:nTF {#1} { true } { false } }

```

(End definition for \bool\_to\_str:N and \bool\_to\_str:n. These functions are documented on page 65.)

\bool\_show:n Show the truth value of the boolean.

\bool\_log:n

```

8526 \cs_new_protected:Npn \bool_show:n
8527 { \msg_show_eval:Nn \bool_to_str:n }
8528 \cs_new_protected:Npn \bool_log:n
8529 { \msg_log_eval:Nn \bool_to_str:n }

```

(End definition for \bool\_show:n and \bool\_log:n. These functions are documented on page 66.)

\bool\_show:N Show the truth value of the boolean, as true or false.

\bool\_show:c

```

8530 \cs_new_protected:Npn \bool_show:N { \__bool_show>NN \tl_show:n }
8531 \cs_generate_variant:Nn \bool_show:N { c }
8532 \cs_new_protected:Npn \bool_log:N { \__bool_show>NN \tl_log:n }
8533 \cs_generate_variant:Nn \bool_log:N { c }
8534 \cs_new_protected:Npn \__bool_show>NN #1#2
8535 {

```

```

8536     \__kernel_chk_defined:NT #2
8537     {
8538         \token_case_meaning:NnF #2
8539         {
8540             \c_true_bool { \exp_args:Nx #1 { \token_to_str:N #2 = true } }
8541             \c_false_bool { \exp_args:Nx #1 { \token_to_str:N #2 = false } }
8542         }
8543         {
8544             \msg_error:nnxxx { kernel } { bad-type }
8545             { \token_to_str:N #2 } { \token_to_meaning:N #2 } { bool }
8546         }
8547     }
8548 }
```

(End definition for `\bool_show:N`, `\bool_log:N`, and `\__bool_show>NN`. These functions are documented on page 66.)

`\l_tmpa_bool` A few booleans just if you need them.

```

8549 \bool_new:N \l_tmpa_bool
8550 \bool_new:N \l_tmpb_bool
8551 \bool_new:N \g_tmpa_bool
8552 \bool_new:N \g_tmpb_bool
```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 66.)

`\bool_if_exist_p:N` Copies of the `cs` functions defined in `!3basics`.

```

\bool_if_exist_p:c 8553 \prg_new_eq_conditional:NNn \bool_if_exist:N \cs_if_exist:N
\bool_if_exist:NTF 8554 { TF , T , F , p }
\bool_if_exist:cTF 8555 \prg_new_eq_conditional:NNn \bool_if_exist:c \cs_if_exist:c
8556 { TF , T , F , p }
```

(End definition for `\bool_if_exist:NTF`. This function is documented on page 66.)

## 46.5 Boolean expressions

`\bool_if_p:n` Evaluating the truth value of a list of predicates is done using an input syntax somewhat similar to the one found in other programming languages with ( and ) for grouping, ! for logical “Not”, `&&` for logical “And” and `||` for logical “Or”. However, they perform eager evaluation. We shall use the terms Not, And, Or, Open and Close for these operations.

Any expression is terminated by a Close operation. Evaluation happens from left to right in the following manner using a GetNext function:

- If an Open is seen, start evaluating a new expression using the Eval function and call GetNext again.
- If a Not is seen, remove the ! and call a GetNext function with the logic reversed.
- If none of the above, reinsert the token found (this is supposed to be a predicate function) in front of an Eval function, which evaluates it to the boolean value `<true>` or `<false>`.

The Eval function then contains a post-processing operation which grabs the instruction following the predicate. This is either And, Or or Close. In each case the truth value is used to determine where to go next. The following situations can arise:

***<true>And*** Current truth value is true, logical And seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

***<false>And*** Current truth value is false, logical And seen, stop using the values of predicates within this sub-expression until the next Close. Then return *<false>*.

***<true>Or*** Current truth value is true, logical Or seen, stop using the values of predicates within this sub-expression until the nearest Close. Then return *<true>*.

***<false>Or*** Current truth value is false, logical Or seen, continue with GetNext to examine truth value of next boolean (sub-)expression.

***<true>Close*** Current truth value is true, Close seen, return *<true>*.

***<false>Close*** Current truth value is false, Close seen, return *<false>*.

```

8557 \prg_new_conditional:Npn \bool_if:n #1 { T , F , TF }
8558 {
8559     \if_predicate:w \bool_if_p:n {#1}
8560         \prg_return_true:
8561     \else:
8562         \prg_return_false:
8563     \fi:
8564 }
```

(End definition for *\bool\_if:nTF*. This function is documented on page 67.)

***\bool\_if\_p:n*** To speed up the case of a single predicate, f-expand and check whether the result is one token (possibly surrounded by spaces), which must be *\c\_true\_bool* or *\c\_false\_bool*. We use a version of *\tl\_if\_single:nTF* optimized for speed since we know that an empty #1 is an error. The auxiliary *\\_bool\_if\_p\_aux:w* removes the trailing parenthesis and gets rid of any space, then returns *\c\_true\_bool* or *\c\_false\_bool* as appropriate. This extra work around is because in a *\bool\_set:Nn*, the underlying *\chardef* turns the bool being set temporarily equal to *\relax*, thus assigning a boolean to itself would fail (gh/1055). For the general case, first issue a *\group\_align\_safe\_begin:* as we are using *&&* as syntax shorthand for the And operation and we need to hide it for TeX. This group is closed after *\\_bool\_get\_next>NN* returns *\c\_true\_bool* or *\c\_false\_bool*. That function requires the trailing parenthesis to know where the expression ends.

```

8565 \cs_new:Npn \bool_if_p:n { \exp_args:Nf \_bool_if_p:n }
8566 \cs_new:Npn \_bool_if_p:n #1
8567 {
8568     \tl_if_empty:oT { \use_none:nn #1 . } { \_bool_if_p_aux:w }
8569     \group_align_safe_begin:
8570     \exp_after:wN
8571     \group_align_safe_end:
8572     \exp:w \exp_end_continue_f:w % (
8573         \_bool_get_next>NN \use_i:nnnn #1 )
8574     }
8575 \cs_new:Npn \_bool_if_p_aux:w #1 \use_i:nnnn #2#3
8576 { \bool_if:NTF #2 \c_true_bool \c_false_bool }
```

(End definition for *\bool\_if\_p:n*, *\\_bool\_if\_p:n*, and *\\_bool\_if\_p\_aux:w*. This function is documented on page 67.)

\\_\\_bool\\_get\\_next:NN The GetNext operation. Its first argument is \use\_i:nnnn, \use\_i:nnnn, \use\_iii:nnnn, or \use\_iv:nnnn (we call these “states”). In the first state, this function eventually expand to the truth value \c\_true\_bool or \c\_false\_bool of the expression which follows until the next unmatched closing parenthesis. For instance “\\_\\_bool\\_get\\_next:NN \use\_i:nnnn \c\_true\_bool && \c\_true\_bool )” (including the closing parenthesis) expands to \c\_true\_bool. In the second state (after a !) the logic is reversed. We call these two states “normal” and the next two “skipping”. In the third state (after \c\_true\_bool||) it always returns \c\_true\_bool. In the fourth state (after \c\_false\_bool&&) it always returns \c\_false\_bool and also stops when encountering ||, not only parentheses. This code itself is a switch: if what follows is neither ! nor (, we assume it is a predicate.

```

8577 \cs_new:Npn \_\_bool_get_next:NN #1#2
8578 {
8579   \use:c
8580   {
8581     __bool_
8582     \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
8583     :Nw
8584   }
8585   #1 #2
8586 }
```

(End definition for \\_\\_bool\\_get\\_next:NN.)

\\_\\_bool\_! :Nw The Not operation reverses the logic: it discards the ! token and calls the GetNext operation with the appropriate first argument. Namely the first and second states are interchanged, but after \c\_true\_bool|| or \c\_false\_bool&& the ! is ignored.

```

8587 \cs_new:cpn { __bool_!:Nw } #1#2
8588 {
8589   \exp_after:wN \_\_bool_get_next:NN
8590   #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
8591 }
```

(End definition for \\_\\_bool\_! :Nw.)

\\_\\_bool\_(:Nw The Open operation starts a sub-expression after discarding the open parenthesis. This is done by calling GetNext (which eventually discards the corresponding closing parenthesis), with a post-processing step which looks for And, Or or Close after the group.

```

8592 \cs_new:cpn { __bool_(:Nw } #1#2
8593 {
8594   \exp_after:wN \_\_bool_choose:NNN \exp_after:wN #1
8595   \int_value:w \_\_bool_get_next:NN \use_i:nnnn
8596 }
```

(End definition for \\_\\_bool\_(:Nw.)

\\_\\_bool\_p:Nw If what follows GetNext is neither ! nor (, evaluate the predicate using the primitive \int\_value:w. The canonical true and false values have numerical values 1 and 0 respectively. Look for And, Or or Close afterwards.

```

8597 \cs_new:cpn { __bool_p:Nw } #1
8598 { \exp_after:wN \_\_bool_choose:NNN \exp_after:wN #1 \int_value:w }
```

(End definition for \\_\\_bool\_p:Nw.)

```
\__bool_choose:NNN
\__bool_)_0:
\__bool_)_1:
\__bool_)_2:
\__bool_&_0:
\__bool_&_1:
\__bool_&_2:
\__bool_|_0:
\__bool_|_1:
\__bool_|_2:
```

The arguments are #1: a function such as \use\_i:nnnn, #2: 0 or 1 encoding the current truth value, #3: the next operation, And, Or or Close. We distinguish three cases according to a combination of #1 and #2. Case 2 is when #1 is \use\_iii:nnnn (state 3), namely after \c\_true\_bool ||. Case 1 is when #1 is \use\_i:nnnn and #2 is true or when #1 is \use\_ii:nnnn and #2 is false, for instance for !\c\_false\_bool. Case 0 includes the same with true/false interchanged and the case where #1 is \use\_iv:nnnn namely after \c\_false\_bool &&.

When seeing ) the current subexpression is done, leave the appropriate boolean. When seeing & in case 0 go into state 4, equivalent to having seen \c\_false\_bool &&. In case 1, namely when the argument is true and we are in a normal state continue in the normal state 1. In case 2, namely when skipping alternatives in an Or, continue in the same state. When seeing | in case 0, continue in a normal state; in particular stop skipping for \c\_false\_bool && because that binds more tightly than ||. In the other two cases start skipping for \c\_true\_bool ||.

```
8599 \cs_new:Npn \__bool_choose:NNN #1#2#3
8600 {
8601     \use:c
8602     {
8603         __bool_ \token_to_str:N #3 _ 
8604         #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
8605     }
8606 }
8607 \cs_new:cpn { __bool_)_0: } { \c_false_bool }
8608 \cs_new:cpn { __bool_)_1: } { \c_true_bool }
8609 \cs_new:cpn { __bool_)_2: } { \c_true_bool }
8610 \cs_new:cpn { __bool_&_0: } & { \__bool_get_next>NN \use_iv:nnnn }
8611 \cs_new:cpn { __bool_&_1: } & { \__bool_get_next>NN \use_i:nnnn }
8612 \cs_new:cpn { __bool_&_2: } & { \__bool_get_next>NN \use_iii:nnnn }
8613 \cs_new:cpn { __bool_|_0: } | { \__bool_get_next>NN \use_i:nnnn }
8614 \cs_new:cpn { __bool_|_1: } | { \__bool_get_next>NN \use_iii:nnnn }
8615 \cs_new:cpn { __bool_|_2: } | { \__bool_get_next>NN \use_iii:nnnn }
```

(End definition for \\_\_bool\_choose:NNN and others.)

**\bool\_lazy\_all\_p:n**  
**\bool\_lazy\_all:nTF**

Go through the list of expressions, stopping whenever an expression is false. If the end is reached without finding any false expression, then the result is true.

```
8616 \cs_new:Npn \bool_lazy_all_p:n #1
8617 { \__bool_lazy_all:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
8618 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
8619 {
8620     \if_predicate:w \bool_lazy_all_p:n {#1}
8621     \prg_return_true:
8622     \else:
8623     \prg_return_false:
8624     \fi:
8625 }
8626 \cs_new:Npn \__bool_lazy_all:n #1
8627 {
8628     \__bool_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
8629     \bool_if:nF {#1}
8630     { \__bool_use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
8631     \__bool_lazy_all:n
8632 }
```

(End definition for `\bool_lazy_all:nTF` and `\_bool_lazy_all:n`. This function is documented on page 68.)

- `\bool_lazy_and:p:nn` Only evaluate the second expression if the first is `true`. Note that #2 must be removed as an argument, not just by skipping to the `\else:` branch of the conditional since #2 may contain unbalanced T<sub>E</sub>X conditionals.

```
8633 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
8634 {
8635     \if_predicate:w
8636         \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
8637         \prg_return_true:
8638     \else:
8639         \prg_return_false:
8640     \fi:
8641 }
```

(End definition for `\bool_lazy_and:nnTF`. This function is documented on page 68.)

- `\bool_lazy_any_p:n` Go through the list of expressions, stopping whenever an expression is `true`. If the end is reached without finding any `true` expression, then the result is `false`.

`\_bool_lazy_any:n`

```
8642 \cs_new:Npn \bool_lazy_any_p:n #1
8643     { \_bool_lazy_any:n #1 \q_bool_recursion_tail \q_bool_recursion_stop }
8644 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
8645 {
8646     \if_predicate:w \bool_lazy_any_p:n {#1}
8647         \prg_return_true:
8648     \else:
8649         \prg_return_false:
8650     \fi:
8651 }
8652 \cs_new:Npn \_bool_lazy_any:n #1
8653 {
8654     \_bool_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
8655     \bool_if:nT {#1}
8656         { \_bool_use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
8657     \_bool_lazy_any:n
8658 }
```

(End definition for `\bool_lazy_any:nTF` and `\_bool_lazy_any:n`. This function is documented on page 68.)

- `\bool_lazy_or_p:nn` Only evaluate the second expression if the first is `false`.

`\bool_lazy_or:nnTF`

```
8659 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
8660 {
8661     \if_predicate:w
8662         \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
8663         \prg_return_true:
8664     \else:
8665         \prg_return_false:
8666     \fi:
8667 }
```

(End definition for `\bool_lazy_or:nnTF`. This function is documented on page 68.)

`\bool_not_p:n` The Not variant just reverses the outcome of `\bool_if_p:n`. Can be optimized but this is nice and simple and according to the implementation plan. Not even particularly useful to have it when the infix notation is easier to use.

```
8668 \cs_new:Npn \bool_not_p:n #1 { \bool_if_p:n { ! ( #1 ) } }
```

(End definition for `\bool_not_p:n`. This function is documented on page 68.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise return `true`.

```
8669 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
8670   {
8671     \bool_if:nT {#1} \reverse_if:N
8672     \if_predicate:w \bool_if_p:n {#2}
8673       \prg_return_true:
8674     \else:
8675       \prg_return_false:
8676     \fi:
8677   }
```

(End definition for `\bool_xor:nnTF`. This function is documented on page 68.)

## 46.6 Logical loops

`\bool_while_do:Nn` A `while` loop where the boolean is tested before executing the statement. The “while” version executes the code as long as the boolean is true; the “until” version executes the code as long as the boolean is false.

`\bool_until_do:cn`

```
8678 \cs_new:Npn \bool_while_do:Nn #1#2
8679   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
8680 \cs_new:Npn \bool_until_do:Nn #1#2
8681   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
8682 \cs_generate_variant:Nn \bool_while_do:Nn { c }
8683 \cs_generate_variant:Nn \bool_until_do:Nn { c }
```

(End definition for `\bool_while_do:Nn` and `\bool_until_do:Nn`. These functions are documented on page 69.)

`\bool_do_while:Nn` A do-while loop where the body is performed at least once and the boolean is tested after executing the body. Otherwise identical to the above functions.

`\bool_do_until:Nn`

```
8684 \cs_new:Npn \bool_do_while:Nn #1#2
8685   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
8686 \cs_new:Npn \bool_do_until:Nn #1#2
8687   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
8688 \cs_generate_variant:Nn \bool_do_while:Nn { c }
8689 \cs_generate_variant:Nn \bool_do_until:Nn { c }
```

(End definition for `\bool_do_while:Nn` and `\bool_do_until:Nn`. These functions are documented on page 69.)

`\bool_while_do:nn` Loop functions with the test either before or after the first body expansion.

`\bool_do_while:nn`

```
8690 \cs_new:Npn \bool_while_do:nn #1#2
8691   {
8692     \bool_if:nT {#1}
8693     {
```

```

8694      #2
8695      \bool_while_do:nn {#1} {#2}
8696    }
8697  }
8698 \cs_new:Npn \bool_do_while:nn #1#2
8699  {
8700    #2
8701    \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
8702  }
8703 \cs_new:Npn \bool_until_do:nn #1#2
8704  {
8705    \bool_if:nF {#1}
8706    {
8707      #2
8708      \bool_until_do:nn {#1} {#2}
8709    }
8710  }
8711 \cs_new:Npn \bool_do_until:nn #1#2
8712  {
8713    #2
8714    \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
8715  }

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 69.)

## 46.7 Producing multiple copies

8716 `<@=prg>`

This function uses a cascading csname technique by David Kastrup (who else :-)

The idea is to make the input 25 result in first adding five, and then 20 copies of the code to be replicated. The technique uses cascading csnames which means that we start building several csnames so we end up with a list of functions to be called in reverse order. This is important here (and other places) because it means that we can for instance make the function that inserts five copies of something to also hand down ten to the next function in line. This is exactly what happens here: in the example with 25 then the next function is the one that inserts two copies but it sees the ten copies handed down by the previous function. In order to avoid the last function to insert say, 100 copies of the original argument just to gobble them again we define separate functions to be inserted first. These functions also close the expansion of `\exp:w`, which ensures that `\prg_replicate:nn` only requires two steps of expansion.

This function has one flaw though: Since it constantly passes down ten copies of its previous argument it severely affects the main memory once you start demanding hundreds of thousands of copies. Now I don't think this is a real limitation for any ordinary use, and if necessary, it is possible to write `\prg_replicate:nn {1000} { \prg_replicate:nn {1000} {{<code>}} }`. An alternative approach is to create a string of m's with `\exp:w` which can be done with just four macros but that method has its own problems since it can exhaust the string pool. Also, it is considerably slower than what we use here so the few extra csnames are well spent I would say.

```

8717 \cs_new:Npn \prg_replicate:nn #1
8718  {
8719    \exp:w

```

```

8720      \exp_after:wN \__prg_replicate_first:N
8721      \int_value:w \int_eval:n {#1}
8722      \cs_end:
8723    }
8724  \cs_new:Npn \__prg_replicate:N #1
8725    { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
8726  \cs_new:Npn \__prg_replicate_first:N #1
8727    { \cs:w __prg_replicate_first_ #1 :n \__prg_replicate:N }

```

Then comes all the functions that do the hard work of inserting all the copies. The first function takes :n as a parameter.

```

8728  \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
8729  \cs_new:cpn { __prg_replicate_0:n } #1
8730    { \cs_end: {#1#1#1#1#1#1#1#1} }
8731  \cs_new:cpn { __prg_replicate_1:n } #1
8732    { \cs_end: {#1#1#1#1#1#1#1#1} #1 }
8733  \cs_new:cpn { __prg_replicate_2:n } #1
8734    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1 }
8735  \cs_new:cpn { __prg_replicate_3:n } #1
8736    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1 }
8737  \cs_new:cpn { __prg_replicate_4:n } #1
8738    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1 }
8739  \cs_new:cpn { __prg_replicate_5:n } #1
8740    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
8741  \cs_new:cpn { __prg_replicate_6:n } #1
8742    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
8743  \cs_new:cpn { __prg_replicate_7:n } #1
8744    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
8745  \cs_new:cpn { __prg_replicate_8:n } #1
8746    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
8747  \cs_new:cpn { __prg_replicate_9:n } #1
8748    { \cs_end: {#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }

```

Users shouldn't ask for something to be replicated once or even not at all but...

```

8749  \cs_new:cpn { __prg_replicate_first_-:n } #1
8750    {
8751      \exp_end:
8752      \msg_expandable_error:nn { prg } { negative-replication }
8753    }
8754  \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
8755  \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
8756  \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
8757  \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
8758  \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
8759  \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
8760  \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
8761  \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
8762  \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
8763  \cs_new:cpn { __prg_replicate_first_9:n } #1
8764    { \exp_end: #1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn` and others. This function is documented on page 69.)

## 46.8 Detecting T<sub>E</sub>X's mode

`\mode_if_vertical:p:` For testing vertical mode. Strikes me here on the bus with David, that as long as we are just talking about returning true and false states, we can just use the primitive conditionals for this and gobbling the `\exp_end:` in the input stream. However this requires knowledge of the implementation so we keep things nice and clean and use the return statements.

```
8765 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
8766   { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 70.)

`\mode_if_horizontal:p:` For testing horizontal mode.

```
8767 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
8768   { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 70.)

`\mode_if_inner:p:` For testing inner mode.

```
8769 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
8770   { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_inner:TF`. This function is documented on page 70.)

`\mode_if_math:p:` For testing math mode. At the beginning of an alignment cell, this should be used only inside a non-expandable function.

```
8771 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
8772   { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }
```

(End definition for `\mode_if_math:TF`. This function is documented on page 70.)

## 46.9 Internal programming functions

`\group_align_safe_begin:` T<sub>E</sub>X's alignment structures present many problems. As Knuth says himself in *T<sub>E</sub>X: The Program*: “It's sort of a miracle whenever `\halign` or `\valign` work, [...]” One problem relates to commands that internally issue a `\cr` but also peek ahead for the next character for use in, say, an optional argument. If the next token happens to be a & with category code 4 we get some sort of weird error message because the underlying `\futurelet` stores the token at the end of the alignment template. This could be a &<sub>4</sub> giving a message like ! Misplaced `\cr`. or even worse: it could be the `\endtemplate` token causing even more trouble! To solve this we have to open a special group so that T<sub>E</sub>X still thinks it's on safe ground but at the same time we don't want to introduce any brace group that may find its way to the output. The following functions help with this by using behaviour documented only in Appendix D of *The T<sub>E</sub>Xbook*... In short evaluating '{ and '}' as numbers will not change the counter T<sub>E</sub>X uses to keep track of its state in an alignment, whereas gobbling a brace using `\if_false:` will affect T<sub>E</sub>X's state without producing any real group. We place the `\if_false: { \fi:` part at that place so that the successive expansions of `\group_align_safe_begin/end:` are always brace balanced.

```
8773 \group_begin:
8774 \tex_catcode:D '^@ = 2 \exp_stop_f:
8775 \cs_new:Npn \group_align_safe_begin:
8776   { \exp:w \if_false: { \fi: '^@ \exp_stop_f: }
```

```
8777 \group_end:  
8778   \cs_new:Npn \group_align_safe_end:  
8779     { \if_int_compare:w ‘{ = \c_zero_int } \fi: }
```

(End definition for `\group_align_begin:` and `\group_align_end:`. These functions are documented on page 71.)

`\g_kernel_prg_map_int` A nesting counter for mapping.

```
8780 \int_new:N \g_kernel_prg_map_int
```

(End definition for `\g_kernel_prg_map_int`.)

`\prg_break_point:Nn` These are defined in `I3basics`, as they are needed “early”. This is just a reminder that is the case!

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 70.)

`\prg_break_point:` Also done in `I3basics`.

```
\prg_break:
```

```
\prg_break:n
```

(End definition for `\prg_break_point:`, `\prg_break:`, and `\prg_break:n`. These functions are documented on page 71.)

```
8781 ⟨/package⟩
```

# Chapter 47

## I3sys implementation

8782 ⟨@@=sys⟩

### 47.1 Kernel code

8783 ⟨\*package⟩  
8784 ⟨\*tex⟩

#### 47.1.1 Detecting the engine

⟨\_\_sys\_const:nn⟩ Set the T, F, TF, p forms of #1 to be constants equal to the result of evaluating the boolean expression #2.

```
8785 \cs_new_protected:Npn __sys_const:nn #1#2
8786 {
8787     \bool_if:nTF {#2}
8788     {
8789         \cs_new_eq:cN {#1 :T} \use:n
8790         \cs_new_eq:cN {#1 :F} \use:none:n
8791         \cs_new_eq:cN {#1 :TF} \use_i:nn
8792         \cs_new_eq:cN {#1 _p:} \c_true_bool
8793     }
8794     {
8795         \cs_new_eq:cN {#1 :T} \use_none:n
8796         \cs_new_eq:cN {#1 :F} \use:n
8797         \cs_new_eq:cN {#1 :TF} \use_ii:nn
8798         \cs_new_eq:cN {#1 _p:} \c_false_bool
8799     }
8800 }
```

(End definition for \_\_sys\_const:nn.)

⟨sys\_if\_engine\_luatex\_p:⟩ Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive.

```
8801 \str_const:Nx \c_sys_engine_str
8802 {
8803     \cs_if_exist:NT \tex_luatexversion:D { luatex }
8804     \cs_if_exist:NT \tex_pdftexversion:D { pdftex }
8805     \cs_if_exist:NT \tex_kanjiskip:D
8806 }
```

```

8807     \cs_if_exist:NTF \tex_enablecjktoken:D
8808         { uptex }
8809         { ptex }
8810     }
8811     \cs_if_exist:NT \tex_XeTeXversion:D { xetex }
8812 }
8813 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
8814 {
8815     \__sys_const:nn { sys_if_engine_ #1 }
8816     { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
8817 }

```

(End definition for `\sys_if_engine_luatex:TF` and others. These functions are documented on page 73.)

### `\c_sys_engine_exec_str` `\c_sys_engine_format_str`

Take the functions defined above, and set up the engine and format names. `\c_sys_engine_exec_str` differs from `\c_sys_engine_str` as it is the *actual* engine name, not a “filtered” version. It differs for `ptex` and `uptex`, which have a leading `e`, and for `luatex`, because L<sup>A</sup>T<sub>E</sub>X uses the LuaHBT<sub>E</sub>X engine.

`\c_sys_engine_format_str` is quite similar to `\c_sys_engine_str`, except that it differentiates `pdfflatex` from `latex` (which is pdfT<sub>E</sub>X in DVI mode). This differentiation, however, is reliable only if the user doesn’t change `\tex_pdfoutput:D` before loading this code.

```

8818 \group_begin:
8819   \cs_set_eq:NN \lua_now:e      \tex_directlua:D
8820   \str_const:Nx \c_sys_engine_exec_str
8821   {
8822     \sys_if_engine_pdftex:T { pdf }
8823     \sys_if_engine_xetex:T { xe }
8824     \sys_if_engine_ptex:T { ep }
8825     \sys_if_engine_uptex:T { eup }
8826     \sys_if_engine_luatex:T
8827     {
8828       lua \lua_now:e
8829       {
8830         if (pcall(require, 'luaharfbuzz')) then ~
8831           tex.print("hb")
8832         end
8833       }
8834     }
8835   tex
8836 }
8837 \group_end:
8838 \str_const:Nx \c_sys_engine_format_str
8839 {
8840   \cs_if_exist:NTF \fmtname
8841   {
8842     \bool_lazy_or:nnTF
8843     { \str_if_eq_p:Vn \fmtname { plain } }
8844     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
8845   {
8846     \sys_if_engine_pdftex:T
8847     { \int_compare:nNnT { \tex_pdfoutput:D } = { 1 } { pdf } }
8848     \sys_if_engine_xetex:T { xe }

```

```

8849   \sys_if_engine_ptex:T { p }
8850   \sys_if_engine_uptex:T { up }
8851   \sys_if_engine_luatex:T
8852   {
8853     \int_compare:nNnT { \tex_pdfoutput:D } = { 0 } { dvi }
8854     lua
8855   }
8856   \str_if_eq:VnTF \fmtname { LaTeX2e }
8857   { latex }
8858   {
8859     \bool_lazy_and:nnT
8860     { \sys_if_engine_pdftex_p: }
8861     { \int_compare_p:nNn { \tex_pdfoutput:D } = { 0 } }
8862     { e }
8863     tex
8864   }
8865 }
8866 { \fmtname }
8867 }
8868 { unknown }
8869 }

```

(End definition for `\c_sys_engine_exec_str` and `\c_sys_engine_format_str`. These variables are documented on page 73.)

### 47.1.2 Randomness

This candidate function is placed there because `\sys_if_rand_exist:TF` is used in `\l3fp-rand`.

`\sys_if_rand_exist:p:` Currently, randomness exists under pdftEX, LuaTeX, pTeX and upTeX.

`\sys_if_rand_exist:TF`

```

8870 \__sys_const:nn { sys_if_rand_exist }
8871 { \cs_if_exist_p:N \tex_uniformdeviate:D }

```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page 311.)

### 47.1.3 Platform

Setting these up requires the file module (file lookup), so is actually implemented there.

(End definition for `\sys_if_platform_unix:TF`, `\sys_if_platform_windows:TF`, and `\c_sys_platform_str`. These functions are documented on page 74.)

### 47.1.4 Configurations

Loading the backend code is pretty simply: check that the backend is valid, then load it up.

```

\sys_load_backend:n
\__sys_load_backend_check:N
  \c_sys_backend_str

  \cs_new_protected:Npn \sys_load_backend:n #
  {
    \sys_finalise:
    \str_if_exist:NTF \c_sys_backend_str
    {
      \str_if_eq:VnF \c_sys_backend_str {#1}
      { \msg_error:nn { sys } { backend-set } }
    }
  }

```

```

8879 }
8880 {
8881     \tl_if_blank:nF {#1}
8882         { \tl_gset:Nn \g__sys_backend_tl {#1} }
8883     \__sys_load_backend_check:N \g__sys_backend_tl
8884     \str_const:Nx \c__sys_backend_str { \g__sys_backend_tl }
8885     \__kernel_sys_configuration_load:n
8886         { l3backend- \c__sys_backend_str }
8887     }
8888 }
8889 \cs_new_protected:Npn \__sys_load_backend_check:N #1
8890 {
8891     \sys_if_engine_xetex:TF
8892     {
8893         \str_case:VnF #1
8894         {
8895             { dvisvgm } { }
8896             { xdvipdfmx } { \tl_gset:Nn #1 { xetex } }
8897             { xetex } { }
8898         }
8899     {
8900         \msg_error:nnxx { sys } { wrong-backend }
8901             #1 { xetex }
8902         \tl_gset:Nn #1 { xetex }
8903     }
8904 }
8905 {
8906     \sys_if_output_pdf:TF
8907     {
8908         \str_if_eq:VnTF #1 { pdfmode }
8909         {
8910             \sys_if_engine_luatex:TF
8911                 { \tl_gset:Nn #1 { luatex } }
8912                 { \tl_gset:Nn #1 { pdftex } }
8913         }
8914     {
8915         \bool_lazy_or:nnF
8916             { \str_if_eq_p:Vn #1 { luatex } }
8917             { \str_if_eq_p:Vn #1 { pdftex } }
8918         {
8919             \msg_error:nnxx { sys } { wrong-backend }
8920                 #1 { \sys_if_engine_luatex:TF { luatex } { pdftex } }
8921             \sys_if_engine_luatex:TF
8922                 { \tl_gset:Nn #1 { luatex } }
8923                 { \tl_gset:Nn #1 { pdftex } }
8924         }
8925     }
8926 }
8927 {
8928     \str_case:VnF #1
8929     {
8930         { dvipdfmx } { }
8931         { dvips } { }
8932         { dvisvgm } { }

```

```

8933     }
8934     {
8935         \msg_error:n { sys } { wrong-backend }
8936             #1 { dvips }
8937             \tl_gset:Nn #1 { dvips }
8938         }
8939     }
8940 }
8941 }
```

(End definition for `\sys_load_backend:n`, `\_sys_load_backend_check:N`, and `\c_sys_backend_str`. These functions are documented on page 75.)

**\sys\_ensure\_backend:** A simple wrapper.

```

8942 \cs_new_protected:Npn \sys_ensure_backend:
8943 {
8944     \str_if_exist:N \c_sys_backend_str
8945         { \sys_load_backend:n { } }
8946 }
```

(End definition for `\sys_ensure_backend:`. This function is documented on page 75.)

```
\g__sys_debug_bool
8947 \bool_new:N \g__sys_debug_bool
```

(End definition for `\g__sys_debug_bool`.)

**\sys\_load\_debug:** Simple.

```

8948 \cs_new_protected:Npn \sys_load_debug:
8949 {
8950     \bool_if:NF \g__sys_debug_bool
8951         { \_kernel_sys_configuration_load:n { 13debug } }
8952     \bool_gset_true:N \g__sys_debug_bool
8953 }
```

(End definition for `\sys_load_debug:`. This function is documented on page 76.)

#### 47.1.5 Access to the shell

```
\l__sys_internal_tl
8954 \tl_new:N \l__sys_internal_tl
```

(End definition for `\l__sys_internal_tl`.)

**\c\_\_sys\_marker\_tl** The same idea as the marker for rescanning token lists.

```
8955 \tl_const:Nx \c__sys_marker_tl { : \token_to_str:N : }
```

(End definition for `\c__sys_marker_tl`.)

```

\sys_get_shell:nnTF Setting using a shell is at this level just a slightly specialised file operation, with an
\sys_get_shell:nnN additional check for quotes, as these are not supported.

  \__sys_get:nnN
  \__sys_get_do:Nw
    8956  \cs_new_protected:Npn \sys_get_shell:nnN #1#2#3
    8957  {
      8958    \sys_get_shell:nnNF {#1} {#2} #3
      8959    { \tl_set:Nn #3 { \q_no_value } }
    8960  }
    8961  \prg_new_protected_conditional:Npnn \sys_get_shell:nnN #1#2#3 { T , F , TF }
    8962  {
      8963    \sys_if_shell:TF
      8964    { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
      8965    { \prg_return_false: }
    8966  }
    8967  \cs_new_protected:Npn \__sys_get:nnN #1#2#3
    8968  {
      8969    \tl_if_in:nnTF {#1} { " }
      8970    {
        8971      \msg_error:nnx
        8972      { kernel } { quote-in-shell } {#1}
        8973      \prg_return_false:
      8974    }
      8975    {
        8976      \group_begin:
        8977        \if_false: { \fi:
        8978          \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
        8979          \exp_args:No \tex_everyeof:D { \c__sys_marker_tl }
        8980          #2 \scan_stop:
        8981          \exp_after:wN \__sys_get_do:Nw
        8982          \exp_after:wN #3
        8983          \exp_after:wN \prg_do_nothing:
        8984            \tex_input:D | "#1" \scan_stop:
        8985          \if_false: } \fi:
        8986          \prg_return_true:
      8987        }
      8988      }
    8989  \exp_args:Nno \use:nn
    8990  { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
    8991  { \c__sys_marker_tl }
    8992  {
      8993    \group_end:
      8994    \tl_set:No #1 {#2}
    8995  }

```

(End definition for `\sys_get_shell:nnNTF` and others. These functions are documented on page 74.)

`\c__sys_shell_stream_int` This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```

  8996  \sys_if_engine_luatex:F
  8997  { \int_const:Nn \c__sys_shell_stream_int { 18 } }

```

(End definition for `\c__sys_shell_stream_int`.)

`\sys_shell_now:n` Execute commands through shell escape immediately.

`\__sys_shell_now:e` For LuaTeX, we use a pseudo-primitive to do the actual work.

```

  8998  </tex>

```

```

8999  <*lua>
9000  do
9001      local os_exec = os.execute
9002
9003      local function shellescape(cmd)
9004          local status,msg = os_exec(cmd)
9005          if status == nil then
9006              write_nl("log","runsystem(.. cmd ..)..." .. msg .. "\n")
9007          elseif status == 0 then
9008              write_nl("log","runsystem(.. cmd ..)..." .. "executed\n")
9009          else
9010              write_nl("log","runsystem(.. cmd ..)..." .. "failed " .. (msg or "") .. "\n")
9011          end
9012      end
9013      luacmd("__sys_shell_now:e", function()
9014          shellescape(scan_string())
9015      end, "global", "protected")
9016  </lua>
9017  <*tex>
9018 \sys_if_engine_luatex:TF
9019  {
9020      \cs_new_protected:Npn \sys_shell_now:n #1
9021          { \__sys_shell_now:e { \exp_not:n {#1} } }
9022      }
9023  {
9024      \cs_new_protected:Npn \sys_shell_now:n #1
9025          { \iow_now:Nn \c__sys_shell_stream_int {#1} }
9026      }
9027 \cs_generate_variant:Nn \sys_shell_now:n { x }
9028 </tex>

```

(End definition for `\sys_shell_now:n` and `\__sys_shell_now:e`. This function is documented on page 75.)

`\sys_shell_shipout:n` Execute commands through shell escape at shipout.

`\__sys_shell_shipout:e` For LuaTeX, we use the same helper as above but delayed using a `late_lua` whatsit. Creating a `late_lua` whatsit works a bit different if we are running under ConTeXt.

```

9029  <*lua>
9030      local new_latelua = nodes and nodes.nuts and nodes.nuts.pool and nodes.nuts.pool.latelua
9031      local whatsit_id = node.id'whatsit'
9032      local latelua_sub = node.subtype'late_lua'
9033      local node_new = node.direct.new
9034      local setfield = node.direct.setwhatsitfield or node.direct.setfield
9035      return function(f)
9036          local n = node_new(whatsit_id, latelua_sub)
9037          setfield(n, 'data', f)
9038          return n
9039      end
9040  end)()
9041  local node_write = node.direct.write
9042
9043  luacmd("__sys_shell_shipout:e", function()
9044      local cmd = scan_string()
9045      node_write(new_latelua(function() shellescape(cmd) end))

```

```

9046     end, "global", "protected")
9047 end
9048 </lua>
9049 <*tex>
9050 \sys_if_engine_luatex:TF
9051 {
9052     \cs_new_protected:Npn \sys_shell_shipout:n #1
9053     { \_sys_shell_shipout:e { \exp_not:n {#1} } }
9054 }
9055 {
9056     \cs_new_protected:Npn \sys_shell_shipout:n #1
9057     { \iow_shipout:Nn \c_sys_shell_stream_int {#1} }
9058 }
9059 \cs_generate_variant:Nn \sys_shell_shipout:n { x }

(End definition for \sys_shell_shipout:n and \_sys_shell_shipout:e. This function is documented
on page 75.)

```

## 47.2 Dynamic (every job) code

```

\sys_everyjob:
\__sys_everyjob:n
\g__sys_everyjob_tl
9060 \cs_new_protected:Npn \sys_everyjob:
9061 {
9062     \tl_use:N \g__sys_everyjob_tl
9063     \tl_gclear:N \g__sys_everyjob_tl
9064 }
9065 \cs_new_protected:Npn \__sys_everyjob:n #1
9066 { \tl_gput_right:Nn \g__sys_everyjob_tl {#1} }
9067 \tl_new:N \g__sys_everyjob_tl

(End definition for \sys_everyjob:, \__sys_everyjob:n, and \g__sys_everyjob_tl. This function is
documented on page ??.)

```

### 47.2.1 The name of the job

#### \c\_sys\_jobname\_str

Inherited from the L<sup>A</sup>T<sub>E</sub>X3 name for the primitive. This *has* to be the primitive as it's set in \everyjob. If the user does

```
pdflatex \input some-file-name
```

then \everyjob is inserted *before* \jobname is changed form **texput**, and thus we would have the wrong result.

```

9068 \__sys_everyjob:n
9069 { \cs_new_eq:NN \c_sys_jobname_str \tex_jobname:D }
```

(End definition for \c\_sys\_jobname\_str. This variable is documented on page 72.)

### 47.2.2 Time and date

```
\c_sys_minute_int
\c_sys_hour_int
\c_sys_day_int
\c_sys_month_int
\c_sys_year_int
```

```
9070 \__sys_everyjob:n
9071   {
9072     \group_begin:
9073       \cs_set:Npn \__sys_tmp:w #1
9074         {
9075           \str_if_eq:eeTF { \cs_meaning:N #1 } { \token_to_str:N #1 }
9076             { #1 }
9077             {
9078               \cs_if_exist:NTF \tex_primitive:D
9079                 {
9080                   \bool_lazy_and:nnTF
9081                     { \sys_if_engine_xetex_p: }
9082                     {
9083                       \int_compare_p:nNn
9084                         { \exp_after:wN \use_none:n \tex_XeTeXrevision:D }
9085                           < { 99999 }
9086                         }
9087                         { 0 }
9088                         { \tex_primitive:D #1 }
9089                     }
9090                   { 0 }
9091                 }
9092               }
9093     \int_const:Nn \c_sys_minute_int
9094       { \int_mod:nn { \__sys_tmp:w \time } { 60 } }
9095     \int_const:Nn \c_sys_hour_int
9096       { \int_div_truncate:nn { \__sys_tmp:w \time } { 60 } }
9097     \int_const:Nn \c_sys_day_int { \__sys_tmp:w \day }
9098     \int_const:Nn \c_sys_month_int { \__sys_tmp:w \month }
9099     \int_const:Nn \c_sys_year_int { \__sys_tmp:w \year }
9100   \group_end:
9101 }
```

(End definition for `\c_sys_minute_int` and others. These variables are documented on page 72.)

### 47.2.3 Random numbers

`\sys_rand_seed:`

Unpack the primitive. When random numbers are not available, we return zero after an error (and incidentally make sure the number of expansions needed is the same as with random numbers available).

```
9102 \__sys_everyjob:n
9103   {
9104     \sys_if_rand_exist:TF
9105       { \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D } }
9106       {
9107         \cs_new:Npn \sys_rand_seed:
9108           {
9109             \int_value:w
```

```

9110     \msg_expandable_error:nnn { kernel } { fp-no-random }
9111         { \sys_rand_seed: }
9112         \c_zero_int
9113     }
9114 }
9115 }
```

(End definition for `\sys_rand_seed:`. This function is documented on page 74.)

`\sys_gset_rand_seed:n` The primitive always assigns the seed globally.

```

9116 \__sys_everyjob:n
9117 {
9118     \sys_if_rand_exist:TF
9119     {
9120         \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9121             { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
9122     }
9123     {
9124         \cs_new_protected:Npn \sys_gset_rand_seed:n #1
9125             {
9126                 \msg_error:nnn { kernel } { fp-no-random }
9127                 { \sys_gset_rand_seed:n {#1} }
9128             }
9129     }
9130 }
```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 74.)

`\sys_timer:` In LuaTeX, create a pseudo-primitive, otherwise try to locate the real primitive. The elapsed time will be available if this succeeds.

```

9131 </tex>
9132 <*lua>
9133 local gettimeofday = os.gettimeofday
9134 local epoch = gettimeofday() - os.clock()
9135 local write = tex.write
9136 local tointeger = math.tointeger
9137 luacmd'__sys_elapsedtime:', function()
9138     write(tointeger((gettimeofday() - epoch)*65536 // 1))
9139 end, 'global')
9140 </lua>
9141 <*tex>
9142 \sys_if_engine_luatex:TF
9143 {
9144     \cs_new:Npn \sys_timer:
9145         { __sys_elapsedtime: }
9146 }
9147 {
9148     \cs_if_exist:NTF \tex_elapsedtime:D
9149         {
9150             \cs_new:Npn \sys_timer:
9151                 { \int_value:w \tex_elapsedtime:D }
9152         }
9153     {
9154         \cs_new:Npn \sys_timer:
```

```

9155     {
9156         \int_value:w
9157         \msg_expandable_error:nnn { kernel } { no-elapsed-time }
9158             { \sys_timer: }
9159             \c_zero_int
9160     }
9161 }
9162 }
9163 \__sys_const:nn { sys_if_timer_exist }
9164     { \cs_if_exist_p:N \tex_elapsedtime:D || \cs_if_exist_p:N \__sys_elapsedtime: }

(End definition for \sys_timer:, \__sys_elapsedtime:, and \sys_if_timer_exist:TF. These functions
are documented on page 73.)

```

#### 47.2.4 Access to the shell

\c\_sys\_shell\_escape\_int Expose the engine's shell escape status to the user.

```

9165 \__sys_everyjob:n
9166 {
9167     \int_const:Nn \c_sys_shell_escape_int
9168     {
9169         \sys_if_engine_luatex:TF
9170         {
9171             \tex_directlua:D
9172             { \tex.sprint(status.shell_escape~or~os.execute()) }
9173         }
9174         { \tex_shellescape:D }
9175     }
9176 }

```

(End definition for \c\_sys\_shell\_escape\_int. This variable is documented on page 75.)

\sys\_if\_shell\_p: \sys\_if\_shell:TF \sys\_if\_shell\_unrestricted\_p: \sys\_if\_shell\_unrestricted:TF \sys\_if\_shell\_restricted\_p: \sys\_if\_shell\_restricted:TF

```

9177 \__sys_everyjob:n
9178 {
9179     \__sys_const:nn { sys_if_shell }
9180     { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
9181     \__sys_const:nn { sys_if_shell_unrestricted }
9182     { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
9183     \__sys_const:nn { sys_if_shell_restricted }
9184     { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }
9185 }

```

(End definition for \sys\_if\_shell:TF, \sys\_if\_shell\_unrestricted:TF, and \sys\_if\_shell\_restricted:TF. These functions are documented on page 75.)

#### 47.2.5 Held over from l3file

\g\_file\_curr\_name\_str See comments about \c\_sys\_jobname\_str: here, as soon as there is file input/output, things get “tided up”.

```

9186 \__sys_everyjob:n
9187 { \cs_gset_eq:NN \g_file_curr_name_str \tex_jobname:D }

```

(End definition for \g\_file\_curr\_name\_str. This variable is documented on page 93.)

## 47.3 Last-minute code

**\sys\_finalise:** A simple hook to finalise the system-dependent layer. This is forced by the backend loader, which is forced by the main loader, so we do not need to include that here.

```

9188 \cs_new_protected:Npn \sys_finalise:
9189   {
9190     \sys_everyjob:
9191     \tl_use:N \g__sys_finalise_tl
9192     \tl_gclear:N \g__sys_finalise_tl
9193   }
9194 \cs_new_protected:Npn \__sys_finalise:n #1
9195   { \tl_gput_right:Nn \g__sys_finalise_tl {#1} }
9196 \tl_new:N \g__sys_finalise_tl

```

(End definition for `\sys_finalise:`, `\__sys_finalise:n`, and `\g__sys_finalise_tl`. This function is documented on page 76.)

### 47.3.1 Detecting the output

This is a simple enough concept: the two views here are complementary.

```

9197 \__sys_finalise:n
9198   {
9199     \str_const:Nx \c_sys_output_str
9200       {
9201         \int_compare:nNnTF
9202           { \cs_if_exist_use:NF \tex_pdfoutput:D { 0 } } > { 0 }
9203           { pdf }
9204           { dvi }
9205       }
9206     \__sys_const:nn { sys_if_output_dvi }
9207       { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
9208     \__sys_const:nn { sys_if_output_pdf }
9209       { \str_if_eq_p:Vn \c_sys_output_str { pdf } }
9210   }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page 73.)

### 47.3.2 Configurations

As the backend has to be checked and possibly adjusted, the approach here is to create a variable and use that in a one-shot to set a constant.

```

9211 \tl_new:N \g__sys_backend_tl
9212 \__sys_finalise:n
9213   {
9214     \__kernel_tl_gset:Nx \g__sys_backend_tl
9215       {
9216         \sys_if_engine_xetex:TF
9217           { xetex }
9218           {
9219             \sys_if_output_pdf:TF
9220               {
9221                 \sys_if_engine_pdftex:TF
9222                   { pdftex }

```

```

9223         { luatex }
9224     }
9225     { dvips }
9226   }
9227 }
9228 }
```

If there is a class option set, and recognised, we pick it up: these will over-ride anything set automatically but will themselves be over-written if there is a package option.

```

9229 \__sys_finalise:n
9230 {
9231   \cs_if_exist:NT \c@classoptionslist
9232   {
9233     \cs_if_eq:NNF \c@classoptionslist \scan_stop:
9234     {
9235       \clist_map_inline:Nn \c@classoptionslist
9236       {
9237         \str_case:nnT {\#1}
9238         {
9239           { dvipdfmx }
9240           { \t1_gset:Nn \g__sys_backend_tl { dvipdfmx } }
9241           { dvips }
9242           { \t1_gset:Nn \g__sys_backend_tl { dvips } }
9243           { dvisvgm }
9244           { \t1_gset:Nn \g__sys_backend_tl { dvisvgm } }
9245           { pdftex }
9246           { \t1_gset:Nn \g__sys_backend_tl { pdfmode } }
9247           { xetex }
9248           { \t1_gset:Nn \g__sys_backend_tl { xdvipdfmx } }
9249         }
9250         { \clist_remove_all:Nn \c@unusedoptionlist {\#1} }
9251       }
9252     }
9253   }
9254 }
```

(End definition for `\g__sys_backend_tl`.)

```

9255 </tex>
9256 </package>
```

# Chapter 48

## I3msg implementation

```
9257  <*package>
9258  <@@=msg>
\l__msg_internal_tl A general scratch for the module.
9259  \tl_new:N \l__msg_internal_tl
(End definition for \l__msg_internal_tl.)
```

```
\l__msg_name_str Used to save module info when creating messages.
\l__msg_text_str 9260 \str_new:N \l__msg_name_str
9261 \str_new:N \l__msg_text_str
(End definition for \l__msg_name_str and \l__msg_text_str.)
```

### 48.1 Internal auxiliaries

```
\s__msg_mark Internal scan marks.
\s__msg_stop 9262 \scan_new:N \s__msg_mark
9263 \scan_new:N \s__msg_stop
(End definition for \s__msg_mark and \s__msg_stop.)
```

```
\_msg_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
9264 \cs_new:Npn \_msg_use_none_delimit_by_s_stop:w #1 \s__msg_stop { }
(End definition for \_msg_use_none_delimit_by_s_stop:w.)
```

### 48.2 Creating messages

Messages are created and used separately, so there two parts to the code here. First, a mechanism for creating message text. This is pretty simple, as there is not actually a lot to do.

```
\c__msg_text_prefix_tl Locations for the text of messages.
\c__msg_more_text_prefix_tl 9265 \tl_const:Nn \c__msg_text_prefix_tl      { msg-text->~ }
9266 \tl_const:Nn \c__msg_more_text_prefix_tl { msg-extra-text->~ }
```

(End definition for `\c__msg_text_prefix_tl` and `\c__msg_more_text_prefix_tl`.)

`\msg_if_exist_p:nn`  
`\msg_if_exist:nnTF`

Test whether the control sequence containing the message text exists or not.

```
9267 \prg_new_conditional:Npnn \msg_if_exist:nn #1#2 { p , T , F , TF }
9268   {
9269     \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
9270     { \prg_return_true: } { \prg_return_false: }
9271   }
```

(End definition for `\msg_if_exist:nnTF`. This function is documented on page 78.)

`\__msg_chk_if_free:nn`

This auxiliary is similar to `\__kernel_chk_if_free:cs:N`, and is used when defining messages with `\msg_new:nnnn`.

```
9272 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
9273   {
9274     \msg_if_exist:nnT {#1} {#2}
9275     {
9276       \msg_error:nnxx { msg } { already-defined }
9277       {#1} {#2}
9278     }
9279   }
```

(End definition for `\__msg_chk_if_free:nn`.)

`\msg_new:nnnn`  
`\msg_new:nn`

Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
9280 \cs_new_protected:Npn \msg_new:nnnn #1#2
9281   {
9282     \__msg_chk_free:nn {#1} {#2}
9283     \msg_gset:nnnn {#1} {#2}
9284   }
9285 \cs_new_protected:Npn \msg_new:nnn #1#2#3
9286   { \msg_new:nnnn {#1} {#2} {#3} { } }
9287 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
9288   {
9289     \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
9290     ##1##2##3##4 {#3}
9291     \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9292     ##1##2##3##4 {#4}
9293   }
9294 \cs_new_protected:Npn \msg_set:nnn #1#2#3
9295   { \msg_set:nnnn {#1} {#2} {#3} { } }
9296 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
9297   {
9298     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
9299     ##1##2##3##4 {#3}
9300     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9301     ##1##2##3##4 {#4}
9302   }
9303 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
9304   { \msg_gset:nnnn {#1} {#2} {#3} { } }
```

(End definition for `\msg_new:nnnn` and others. These functions are documented on page 78.)

## 48.3 Messages: support functions and text

```
\c__msg_error_text_tl
\c__msg_continue_text_tl
\c__msg_critical_text_tl
\c__msg_fatal_text_tl
\c__msg_help_text_tl
\c__msg_no_info_text_tl
\c__msg_on_line_text_tl
\c__msg_return_text_tl
\c__msg_trouble_text_tl

9305 \tl_const:Nn \c__msg_error_text_tl
9306 {
9307   This-is-a-coding-error.
9308   \\ \\
9309 }
9310 \tl_const:Nn \c__msg_continue_text_tl
9311 { Type-<return>-to-continue }
9312 \tl_const:Nn \c__msg_critical_text_tl
9313 { Reading-the-current-file-'g_file_curr_name_str'-will-stop. }
9314 \tl_const:Nn \c__msg_fatal_text_tl
9315 { This-is-a-fatal-error:-LaTeX-will-abort. }
9316 \tl_const:Nn \c__msg_help_text_tl
9317 { For-immediate-help-type-H-<return> }
9318 \tl_const:Nn \c__msg_no_info_text_tl
9319 {
9320   LaTeX-does-not-know-anything-more-about-this-error,-sorry.
9321   \c__msg_return_text_tl
9322 }
9323 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
9324 \tl_const:Nn \c__msg_return_text_tl
9325 {
9326   \\ \\
9327   Try-typing-<return>-to-proceed.
9328   \\
9329   If-that-doesn't-work,-type-X-<return>-to-quit.
9330 }
9331 \tl_const:Nn \c__msg_trouble_text_tl
9332 {
9333   \\ \\
9334   More-errors-will-almost-certainly-follow: \\
9335   the-LaTeX-run-should-be-aborted.
9336 }
```

(End definition for `\c__msg_error_text_tl` and others.)

`\msg_line_number:` For writing the line number nicely. `\msg_line_context:` was set up earlier, so this is not new.

```
9337 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inlineno:D }
9338 \cs_gset:Npn \msg_line_context:
9339 {
9340   \c__msg_on_line_text_tl
9341   \c_space_tl
9342   \msg_line_number:
9343 }
```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 79.)

## 48.4 Showing messages: low level mechanism

```
\_\_msg_interrupt:Nnnn
\_\_msg_no_more_text:nnnn
```

The low-level interruption macro is rather opaque, unfortunately. Depending on the availability of more information there is a choice of how to set up the further help. We feed the extra help text and the message itself to a wrapping auxiliary, in this order because we must first setup TeX's `\errhelp` register before issuing an `\errmessage`. To deal with the various cases of critical or fatal errors with and without help text, there is a bit of argument-passing to do.

```
9344 \cs_new_protected:Npn \_\_msg_interrupt:NnnnN #1#2#3#4#5
9345   {
9346     \str_set:Nx \l_\_msg_text_str { #1 {#2} }
9347     \str_set:Nx \l_\_msg_name_str { \msg_module_name:n {#2} }
9348     \cs_if_eq:cNTF
9349       { \c_\_msg_more_text_prefix_tl #2 / #3 }
9350       \_\_msg_no_more_text:nnnn
9351     {
9352       \_\_msg_interrupt_wrap:nnn
9353         { \use:c { \c_\_msg_text_prefix_tl #2 / #3 } #4 }
9354         { \c_\_msg_continue_text_tl }
9355         {
9356           \c_\_msg_no_info_text_tl
9357           \tl_if_empty:NF #5
9358           { \\ \\ #5 }
9359         }
9360     }
9361   {
9362     \_\_msg_interrupt_wrap:nnn
9363       { \use:c { \c_\_msg_text_prefix_tl #2 / #3 } #4 }
9364       { \c_\_msg_help_text_tl }
9365       {
9366         \use:c { \c_\_msg_more_text_prefix_tl #2 / #3 } #4
9367         \tl_if_empty:NF #5
9368         { \\ \\ #5 }
9369       }
9370     }
9371   }
9372 \cs_new:Npn \_\_msg_no_more_text:nnnn #1#2#3#4 { }
```

(End definition for `\_\_msg_interrupt:Nnnn` and `\_\_msg_no_more_text:nnnn`.)

```
\_\_msg_interrupt_wrap:nnn
\_\_msg_interrupt_text:n
\_\_msg_interrupt_more_text:n
```

First setup TeX's `\errhelp` register with the extra help #1, then build a nice-looking error message with #2. Everything is done using x-type expansion as the new line markers are different for the two type of text and need to be correctly set up. The auxiliary `\_\_msg_interrupt_more_text:n` receives its argument as a line-wrapped string, which is thus unaffected by expansion. We ave to split the main text into two parts as only the "message" itself is wrapped with a leader: the generic help is wrapped at full width. We also have to allow for the two characters used by `\errmessage` itself.

```
9373 \cs_new_protected:Npn \_\_msg_interrupt_wrap:nnn #1#2#3
9374   {
9375     \iow_wrap:nnnN { \\ #3 } { } { } \_\_msg_interrupt_more_text:n
9376     \group_begin:
9377       \int_sub:Nn \l_iow_line_count_int { 2 }
9378       \iow_wrap:nxnN { \l_\_msg_text_str : ~ #1 }
```

```

9379      {
9380        ( \l__msg_name_str )
9381        \prg_replicate:nn
9382        {
9383          \str_count:N \l__msg_text_str
9384          - \str_count:N \l__msg_name_str
9385          + 2
9386        }
9387        { ~ }
9388      }
9389      { } \__msg_interrupt_text:n
9390      \iow_wrap:nnnN { \l__msg_internal_tl \\ \\ #2 } { } { }
9391      \__msg_interrupt:n
9392    }
9393    \cs_new_protected:Npn \__msg_interrupt_text:n #1
9394    {
9395      \group_end:
9396      \tl_set:Nn \l__msg_internal_tl {#1}
9397    }
9398    \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
9399    { \exp_args:Nx \tex_errhelp:D { #1 \iow_newline: } }

```

(End definition for `\__msg_interrupt_wrap:nnn`, `\__msg_interrupt_text:n`, and `\__msg_interrupt-more_text:n`.)

`\__msg_interrupt:n` The business end of the process starts by producing some visual separation of the message from the main part of the log. The error message needs to be printed with everything made “invisible”: TeX’s own information involves the macro in which `\errmessage` is called, and the end of the argument of the `\errmessage`, including the closing brace. We use an active ! to call the `\errmessage` primitive, and end its argument with `\use-none:n {<spaces>}` which fills the output with spaces. Two trailing closing braces are turned into spaces to hide them as well. The group in which we alter the definition of the active ! is closed before producing the message: this ensures that tokens inserted by typing I in the command-line are inserted after the message is entirely cleaned up.

The `\_kernel_iow_with:Nnn` auxiliary, defined in `l3file`, expects an `<integer variable>`, an integer `<value>`, and some `<code>`. It runs the `<code>` after ensuring that the `<integer variable>` takes the given `<value>`, then restores the former value of the `<integer variable>` if needed. We use it to ensure that the `\newlinechar` is 10, as needed for `\iow_newline:` to work, and that `\errorcontextlines` is `-1`, to avoid showing irrelevant context. Note that restoring the former value of these integers requires inserting tokens after the `\errmessage`, which go in the way of tokens which could be inserted by the user. This is unavoidable.

```

9400 \group_begin:
9401   \char_set_lccode:nn { 38 } { 32 } % &
9402   \char_set_lccode:nn { 46 } { 32 } % .
9403   \char_set_lccode:nn { 123 } { 32 } % {
9404   \char_set_lccode:nn { 125 } { 32 } % }
9405   \char_set_catcode_active:N \&
9406 \tex_lowercase:D
9407 {
9408   \group_end:
9409   \cs_new_protected:Npn \__msg_interrupt:n #1
9410   {

```

```

9411     \iow_term:n { }
9412     \_kernel_iow_with:Nnn \tex_newlinechar:D { '\^J }
9413     {
9414         \_kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9415         {
9416             \group_begin:
9417             \cs_set_protected:Npn &
9418             {
9419                 \tex_errmessage:D
9420                 {
9421                     #1
9422                     \use_none:n
9423                     { ..... }
9424                 }
9425             }
9426             \exp_after:wN
9427             \group_end:
9428             &
9429         }
9430     }
9431 }
9432 }
```

(End definition for `\_msg_interrupt:n`.)

## 48.5 Displaying messages

L<sup>A</sup>T<sub>E</sub>X is handling error messages and so the T<sub>E</sub>X ones are disabled.

```
9433 \int_gset:Nn \tex_errorcontextlines:D { -1 }
```

`\msg_fatal_text:n`  
`\msg_critical_text:n`  
`\msg_error_text:n`  
`\msg_warning_text:n`  
`\msg_info_text:n`

A function for issuing messages: both the text and order could in principle vary. The module name may be empty for kernel messages, hence the slightly contorted code path for a space.

```

9434 \cs_new:Npn \msg_fatal_text:n #1
9435 {
9436     Fatal ~
9437     \msg_error_text:n {#1}
9438 }
9439 \cs_new:Npn \msg_critical_text:n #1
9440 {
9441     Critical ~
9442     \msg_error_text:n {#1}
9443 }
9444 \cs_new:Npn \msg_error_text:n #1
9445 { \_msg_text:nn {#1} { Error } }
9446 \cs_new:Npn \msg_warning_text:n #1
9447 { \_msg_text:nn {#1} { Warning } }
9448 \cs_new:Npn \msg_info_text:n #1
9449 { \_msg_text:nn {#1} { Info } }
9450 \cs_new:Npn \_msg_text:nn #1#2
9451 {
9452     \exp_args:Nf \_msg_text:n { \msg_module_type:n {#1} }
9453     \exp_args:Nf \_msg_text:n { \msg_module_name:n {#1} }
```

```

9454      #2
9455    }
9456 \cs_new:Npn \__msg_text:n #1
9457  {
9458    \tl_if_blank:nF {#1}
9459    { #1 ~ }
9460  }

```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 79.)

`\g_msg_module_name_prop`  
`\g_msg_module_type_prop`

For storing public module information: the kernel data is set up in advance.

```

9461 \prop_new:N \g_msg_module_name_prop
9462 \prop_new:N \g_msg_module_type_prop
9463 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX } { }

```

(End definition for `\g_msg_module_name_prop` and `\g_msg_module_type_prop`. These variables are documented on page 78.)

`\msg_module_type:n`

Contextual footer information, with the potential to give modules an alternative name.

```

9464 \cs_new:Npn \msg_module_type:n #1
9465  {
9466    \prop_if_in:NnTF \g_msg_module_type_prop {#1}
9467    { \prop_item:Nn \g_msg_module_type_prop {#1} }
9468    { Package }
9469  }

```

(End definition for `\msg_module_type:n`. This function is documented on page 78.)

`\msg_module_name:n`  
`\msg_see_documentation_text:n`

Contextual footer information, with the potential to give modules an alternative name.

```

9470 \cs_new:Npn \msg_module_name:n #1
9471  {
9472    \prop_if_in:NnTF \g_msg_module_name_prop {#1}
9473    { \prop_item:Nn \g_msg_module_name_prop {#1} }
9474    {#1}
9475  }
9476 \cs_new:Npn \msg_see_documentation_text:n #1
9477  {
9478    See~the~ \msg_module_name:n {#1} ~
9479    documentation~for~further~information.
9480  }

```

(End definition for `\msg_module_name:n` and `\msg_see_documentation_text:n`. These functions are documented on page 78.)

`\__msg_class_new:nn`

```

9481 \group_begin:
9482   \cs_set_protected:Npn \__msg_class_new:nn #1#2
9483   {
9484     \prop_new:c { l__msg_redirect_ #1 _prop }
9485     \cs_new_protected:cpn { __msg_ #1 _code:nnnnnn }
9486     ##1##2##3##4##5##6 {#2}
9487     \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9488     {
9489       \use:x
9490       {

```

```

9491     \exp_not:n { \_msg_use:nnnnnnn {#1} {##1} {##2} }
9492         { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9493         { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9494     }
9495 }
9496 \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
9497     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9498 \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
9499     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9500 \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3##3
9501     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9502 \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
9503     { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } { } }
9504 \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
9505     {
9506         \use:x
9507         {
9508             \exp_not:N \exp_not:n
9509                 { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
9510                 {##3} {##4} {##5} {##6}
9511         }
9512     }
9513 \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
9514     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} {##5} { } }
9515 \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
9516     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
9517 \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3##3
9518     { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
9519 }

```

(End definition for `\_msg_class_new:nn`.)

`\msg_fatal:nnnnn`  
`\msg_fatal:nnxxxx`  
`\msg_fatal:nnnn`

For fatal errors, after the error message TeX bails out. We force a bail out rather than using `\end` as this means it does not matter if we are in a context where normally the run cannot end.

```

9520     \_msg_class_new:nn { fatal }
9521     {
9522         \_msg_interrupt:NnnnN
9523             \msg_fatal_text:n {#1} {#2}
9524             {##3} {##4} {##5} {##6} }
9525             \c__msg_fatal_text_tl
9526             \_msg_fatal_exit:
9527     }
9528 \cs_new_protected:Npn \_msg_fatal_exit:
9529     {
9530         \tex_batchmode:D
9531         \tex_read:D -1 to \l__msg_internal_tl
9532     }

```

(End definition for `\msg_fatal:nnnnn` and others. These functions are documented on page 80.)

`\msg_critical:nnnnn`  
`\msg_critical:nnxxxx`  
`\msg_critical:nnnn`

Not quite so bad: just end the current file.

```

9533     \_msg_class_new:nn { critical }
9534     {

```

```

9535   \__msg_interrupt:NnnnN
9536     \msg_critical_text:n {#1} {#2}
9537     { {#3} {#4} {#5} {#6} }
9538     \c__msg_critical_text_tl
9539     \tex_endinput:D
9540   }

```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 81.)

```

\msg_error:nnnnn
\msg_error:nnxxx
\msg_error:nnnn
\msg_error:nnxx
\msg_error:nnn
\msg_error:nn
\msg_error:nnx
\msg_error:nn

```

```

9541   \cs_undefine:N \msg_error:nnxx
9542   \cs_undefine:N \msg_error:nnx
9543   \cs_undefine:N \msg_error:nn
9544   \__msg_class_new:nn { error }
9545   {
9546     \__msg_interrupt:NnnnN
9547       \msg_error_text:n {#1} {#2}
9548       { {#3} {#4} {#5} {#6} }
9549       \c_empty_tl
9550   }

```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 81.)

```

\__msg_info_aux>NNnnnnnn
\msg_warning:nnnnn
\msg_warning:nnxxx
\msg_warning:nnnn
\msg_warning:nnxx
\msg_warning:nnn
\msg_warning:nn
\msg_warning:nnx
\msg_warning:nn
\msg_note:nnnnnn
\msg_note:nnxxxx
\msg_note:nnnn
\msg_note:nnxx
\msg_note:nnn
\msg_note:nn
\msg_note:nnx
\msg_note:nn
\msg_info:nnnnn
\msg_info:nnxxxx
\msg_info:nnnn
\msg_info:nnxx
\msg_info:nnn
\msg_info:nn
\msg_info:nnx
\msg_info:nn

```

```

9551   \cs_new_protected:Npn \__msg_info_aux:NNnnnnnn #1#2#3#4#5#6#7#8
9552   {
9553     \str_set:Nx \l__msg_text_str { #2 {#3} }
9554     \str_set:Nx \l__msg_name_str { \msg_module_name:n {#3} }
9555     #1 { }
9556     \iow_wrap:nxnN
9557     {
9558       \l__msg_text_str : ~
9559       \use:c { \c__msg_text_prefix_tl #3 / #4 } {#5} {#6} {#7} {#8}
9560     }
9561     {
9562       ( \l__msg_name_str )
9563       \prg_replicate:nn
9564         {
9565           \str_count:N \l__msg_text_str
9566             - \str_count:N \l__msg_name_str
9567         }
9568         { ~ }
9569     }
9570     { } #1
9571     #1 { }
9572   }
9573   \__msg_class_new:nn { warning }
9574   {
9575     \__msg_info_aux:NNnnnnnn \iow_term:n \msg_warning_text:n
9576       {#1} {#2} {#3} {#4} {#5} {#6}
9577   }

```

```

9578     \_\_msg\_class\_new:nn { note }
9579     {
9580         \_\_msg\_info\_aux:NNnnnnnn \iow\_term:n \msg\_info\_text:n
9581             {#1} {#2} {#3} {#4} {#5} {#6}
9582     }
9583     \_\_msg\_class\_new:nn { info }
9584     {
9585         \_\_msg\_info\_aux:NNnnnnnn \iow\_log:n \msg\_info\_text:n
9586             {#1} {#2} {#3} {#4} {#5} {#6}
9587     }

```

(End definition for `\_\_msg\_info\_aux:NNnnnnnn` and others. These functions are documented on page [81.](#))

`\msg\_log:nnnnn`  
`\msg\_log:nnxxx`  
`\msg\_log:nnnn`

`\msg\_log:nnxx`

`\msg\_log:nnn`

`\msg\_log:nnx`

`\msg\_log:nn`

`\msg\_term:nnnnn`

`\msg\_term:nnxxx`

`\msg\_term:nnnn`

`\msg\_term:nnxx`

`\msg\_term:nnn`

`\msg\_term:nnx`

`\msg\_term:nn`

`\msg\_term:nnx`

`\msg\_term:nnnn`

`\msg\_none:nnnnn`

`\msg\_term:nn`

`\msg\_none:nnnn`

`\msg\_none:nnxxx`

`\msg\_none:nnnn`

`\msg\_none:nnxx`

`\msg\_none:nnn`

`\msg\_none:nnx`

`\msg\_none:nn`

`\msg\_show:nnnnn`

`\msg\_show:nnxxx`

`\msg\_show:nnnn`

`\msg\_show:nnxx`

`\msg\_show:nnn`

`\msg\_show:nnx`

`\msg\_show:nn`

`\msg\_show:nnx`

`\msg\_show:n`

`\msg\_show:w`

`\msg\_show_dot:w`

`\msg\_show:nn`

“Log” data is very similar to information, but with no extras added. “Term” is used for communicating with the user through the terminal, like diagnostic messages, and debugging. This is similar to “log” messages, but uses the terminal output.

```

9588     \_\_msg\_class\_new:nn { log }
9589     {
9590         \iow\_wrap:nnN
9591             { \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9592             { } { } \iow\_log:n
9593     }
9594     \_\_msg\_class\_new:nn { term }
9595     {
9596         \iow\_wrap:nnN
9597             { \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9598             { } { } \iow\_term:n
9599     }

```

(End definition for `\msg\_log:nnnnnn` and others. These functions are documented on page [82.](#))

The `none` message type is needed so that input can be gobbled.

```

9600     \_\_msg\_class\_new:nn { none } { }

```

(End definition for `\msg\_none:nnnnnn` and others. These functions are documented on page [82.](#))

The `show` message type is used for `\seq\_show:N` and similar complicated data structures. Wrap the given text with a trailing dot (important later) then pass it to `\_\_msg\_show:n`. If there is `\>` (or if the whole thing starts with `>`) we split there, print the first part and show the second part using `\showtokens` (the `\exp\_after:wN` ensure a nice display). Note that this primitive adds a leading `>` and trailing dot. That is why we included a trailing dot before wrapping and removed it afterwards. If there is no `\>` do the same but with an empty second part which adds a spurious but inevitable `>`.

```

9601     \_\_msg\_class\_new:nn { show }
9602     {
9603         \iow\_wrap:nnN
9604             { \use:c { \c\_msg\_text\_prefix\_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9605             { } { } \_\_msg\_show:n
9606     }
9607     \cs\_new\_protected:Npn \_\_msg\_show:n #1
9608     {
9609         \tl\_if\_in:nnTF { ^~J #1 } { ^~J > ~ }
9610         {

```

```

9611     \tl_if_in:nnTF { #1 \s__msg_mark } { . \s__msg_mark }
9612         { \_msg_show_dot:w } { \_msg_show:w }
9613         ^^J #1 \s__msg_stop
9614     }
9615     { \__msg_show:nn { ? #1 } { } }
9616 }
9617 \cs_new:Npn \_msg_show_dot:w #1 ^^J > ~ #2 . \s__msg_stop
9618     { \_msg_show:nn {#1} {#2} }
9619 \cs_new:Npn \_msg_show:w #1 ^^J > ~ #2 \s__msg_stop
9620     { \_msg_show:nn {#1} {#2} }
9621 \cs_new_protected:Npn \_msg_show:nn #1#2
9622 {
9623     \tl_if_empty:nF {#1}
9624         { \exp_args:No \iow_term:n { \use_none:n #1 } }
9625     \tl_set:Nn \l__msg_internal_tl {#2}
9626     \kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
9627     {
9628         \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9629         {
9630             \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9631                 { \exp_after:wN \l__msg_internal_tl }
9632         }
9633     }
9634 }

```

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 83.)

End the group to eliminate `\__msg_class_new:nn`.

```
9635 \group_end:
```

`\__msg_class_chk_exist:nT` Checking that a message class exists. We build this from `\cs_if_free:cTF` rather than `\cs_if_exist:cTF` because that avoids reading the second argument earlier than necessary.

```

9636 \cs_new:Npn \__msg_class_chk_exist:nT #1
9637 {
9638     \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
9639         { \msg_error:nnx { msg } { class-unknown } {#1} }
9640 }

```

(End definition for `\__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.

```

9641 \tl_new:N \l__msg_class_tl
9642 \tl_new:N \l__msg_current_class_tl

```

(End definition for `\l__msg_class_tl` and `\l__msg_current_class_tl`.)

`\l__msg_redirect_prop` For redirection of individually-named messages

```
9643 \prop_new:N \l__msg_redirect_prop
```

(End definition for `\l__msg_redirect_prop`.)

`\l__msg_hierarchy_seq` During redirection, split the message name into a sequence: `{/module/submodule}`, `{/module}`, and `{}`.

```
9644 \seq_new:N \l__msg_hierarchy_seq
```

(End definition for \l\_msg\_hierarchy\_seq.)

\l\_msg\_class\_loop\_seq Classes encountered when following redirections to check for loops.

9645 \seq\_new:N \l\_msg\_class\_loop\_seq

(End definition for \l\_msg\_class\_loop\_seq.)

Actually using a message is a multi-step process. First, some safety checks on the message and class requested. The code and arguments are then stored to avoid passing them around. The assignment to \msg\_use\_code: is similar to \tl\_set:Nn. The message is eventually produced with whatever \l\_msg\_class\_tl is when \msg\_use\_code: is called. Here is also a good place to suppress tracing output if the trace package is loaded since all (non-expandable) messages go through this auxiliary.

```
9646 \cs_new_protected:Npn \msg_use:nnnnnn #1#2#3#4#5#6#7
9647 {
9648     \cs_if_exist_use:N \conditionally@traceoff
9649     \msg_if_exist:nnTF {#2} {#3}
9650     {
9651         \msg_class_chk_exist:nT {#1}
9652         {
9653             \tl_set:Nn \l_msg_current_class_tl {#1}
9654             \cs_set_protected:Npx \msg_use_code:
9655             {
9656                 \exp_not:n
9657                 {
9658                     \use:c { _msg_ \l_msg_class_tl _code:nnnnnn }
9659                     {#2} {#3} {#4} {#5} {#6} {#7}
9660                 }
9661             }
9662             \msg_use_redirect_name:n { #2 / #3 }
9663         }
9664     }
9665     { \msg_error:nnxx { msg } { unknown } {#2} {#3} }
9666     \cs_if_exist_use:N \conditionally@traceon
9667 }
9668 \cs_new_protected:Npn \msg_use_code: { }
```

The first check is for a individual message redirection. If this applies then no further redirection is attempted. Otherwise, split the message name into *<module>*, *<submodule>* and *<message>* (with an arbitrary number of slashes), and store {*/module/submodule*}, {*/module*} and {} into \l\_msg\_hierarchy\_seq. We then map through this sequence, applying the most specific redirection.

```
9669 \cs_new_protected:Npn \msg_use_redirect_name:n #1
9670 {
9671     \prop_get:NnNTF \l_msg_redirect_prop { / #1 } \l_msg_class_tl
9672     { \msg_use_code: }
9673     {
9674         \seq_clear:N \l_msg_hierarchy_seq
9675         \msg_use_hierarchy:nwwN { }
9676         #1 \s_msg_mark \msg_use_hierarchy:nwwN
9677         / \s_msg_mark \msg_use_none_delimit_by_s_stop:w
9678         \s_msg_stop
9679         \msg_use_redirect_module:n { }
9680     }
```

```

9681     }
9682 \cs_new_protected:Npn \__msg_use_hierarchy:nwwN #1#2 / #3 \s__msg_mark #4
9683 {
9684     \seq_put_left:Nn \l__msg_hierarchy_seq {#1}
9685     #4 { #1 / #2 } #3 \s__msg_mark #4
9686 }

```

At this point, the items of `\l__msg_hierarchy_seq` are the various levels at which we should look for a redirection. Redirections which are less specific than the argument of `\__msg_use_redirect_module:n` are not attempted. This argument is empty for a class redirection, `/module` for a module redirection, *etc.* Loop through the sequence to find the most specific redirection, with module `##1`. The loop is interrupted after testing for a redirection for `##1` equal to the argument `#1` (least specific redirection allowed). When a redirection is found, break the mapping, then if the redirection targets the same class, output the code with that class, and otherwise set the target as the new current class, and search for further redirections. Those redirections should be at least as specific as `##1`.

```

9687 \cs_new_protected:Npn \__msg_use_redirect_module:n #1
9688 {
9689     \seq_map_inline:Nn \l__msg_hierarchy_seq
9690     {
9691         \prop_get:cnNTF { l__msg_redirect_ \l__msg_current_class_tl _prop }
9692             {##1} \l__msg_class_tl
9693             {
9694                 \seq_map_break:n
9695                 {
9696                     \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_class_tl
9697                         { \__msg_use_code: }
9698                         {
9699                             \tl_set_eq:NN \l__msg_current_class_tl \l__msg_class_tl
9700                             \__msg_use_redirect_module:n {##1}
9701                         }
9702                     }
9703                 }
9704             {
9705                 \str_if_eq:nnT {##1} {#1}
9706                 {
9707                     \tl_set_eq:NN \l__msg_class_tl \l__msg_current_class_tl
9708                     \seq_map_break:n { \__msg_use_code: }
9709                 }
9710             }
9711         }
9712     }

```

(End definition for `\__msg_use:nnnnnnn` and others.)

`\msg_redirect_name:nnn` Named message always use the given class even if that class is redirected further. An empty target class cancels any existing redirection for that message.

```

9713 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9714 {
9715     \tl_if_empty:nTF {#3}
9716     {
9717         \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 }
9718         {
9719             \__msg_class_chk_exist:nT {#3}

```

```

9719     { \prop_put:Nnn \l__msg_redirect_prop { / #1 / #2 } {#3} }
9720   }
9721 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 84.)

```

\msg_redirect_class:nn
\msg_redirect_module:nnn
\__msg_redirect:nnn

```

If the target class is empty, eliminate the corresponding redirection. Otherwise, add the redirection. We must then check for a loop: as an initialization, we start by storing the initial class in `\l__msg_current_class_tl`.

```

9722 \cs_new_protected:Npn \msg_redirect_class:nn
9723   { \__msg_redirect:nnn { } }
9724 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9725   { \__msg_redirect:nnn { / #1 } }
9726 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9727   {
9728     \__msg_class_chk_exist:nT {#2}
9729     {
9730       \tl_if_empty:nTF {#3}
9731         { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9732       {
9733         \__msg_class_chk_exist:nT {#3}
9734         {
9735           \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9736           \tl_set:Nn \l__msg_current_class_tl {#2}
9737           \seq_clear:N \l__msg_class_loop_seq
9738           \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9739         }
9740       }
9741     }
9742   }

```

Since multiple redirections can only happen with increasing specificity, a loop requires that all steps are of the same specificity. The new redirection can thus only create a loop with other redirections for the exact same module, #1, and not submodules. After some initialization above, follow redirections with `\l__msg_current_class_tl`, and keep track in `\l__msg_class_loop_seq` of the various classes encountered. A redirection from a class to itself, or the absence of redirection both mean that there is no loop. A redirection to the initial class marks a loop. To break it, we must decide which redirection to cancel. The user most likely wants the newly added redirection to hold with no further redirection. We thus remove the redirection starting from #2, target of the new redirection. Note that no message is emitted by any of the underlying functions: otherwise we may get an infinite loop because of a message from the message system itself.

```

9743 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9744   {
9745     \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9746     \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_current_class_tl
9747     {
9748       \str_if_eq:VnF \l__msg_current_class_tl {#1}
9749       {
9750         \tl_if_eq:NNTF \l__msg_current_class_tl \l__msg_current_class_tl
9751         {
9752           \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
9753           \msg_warning:nnxxxx
9754             { msg } { redirect-loop }

```

```

9755 { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9756 { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
9757 {#3}
9758 {
9759     \seq_map_function:NN \l__msg_class_loop_seq
9760         \__msg_redirect_loop_list:n
9761         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9762     }
9763 }
9764 { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
9765 }
9766 }
9767 }
9768 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
9769 \cs_new:Npn \__msg_redirect_loop_list:n #1 { {#1} ~ => ~ }

(End definition for \msg_redirect_class:nn and others. These functions are documented on page 84.)

```

## 48.6 Kernel-specific functions

These are all retained purely for older `xparse` support.

```

\__kernel_msg_new:nnnn
\__kernel_msg_new:nnn
9770 \cs_new_protected:Npn \__kernel_msg_new:nnnn #1
9771     { \msg_new:nnnn { LaTeX / #1 } }
9772 \cs_new_protected:Npn \__kernel_msg_new:nnn #1
9773     { \msg_new:nnn { LaTeX / #1 } }

(End definition for \__kernel_msg_new:nnnn and \__kernel_msg_new:nnn.)

```

```

\__kernel_msg_info:nnxx
\__kernel_msg_warning:nnx
\__kernel_msg_warning:nnxx
\__kernel_msg_error:nnx
\__kernel_msg_error:nnxx
\__kernel_msg_error:nnxxx
9774 \cs_new_protected:Npn \__kernel_msg_info:nnxx #1
9775     { \msg_info:nnxx { LaTeX / #1 } }
9776 \cs_new_protected:Npn \__kernel_msg_warning:nnx #1
9777     { \msg_warning:nnx { LaTeX / #1 } }
9778 \cs_new_protected:Npn \__kernel_msg_warning:nnxx #1
9779     { \msg_warning:nnxx { LaTeX / #1 } }
9780 \cs_new_protected:Npn \__kernel_msg_error:nnx #1
9781     { \msg_error:nnx { LaTeX / #1 } }
9782 \cs_new_protected:Npn \__kernel_msg_error:nnxx #1
9783     { \msg_error:nnxx { LaTeX / #1 } }
9784 \cs_new_protected:Npn \__kernel_msg_error:nnxxx #1
9785     { \msg_error:nnxxx { LaTeX / #1 } }

(End definition for \__kernel_msg_info:nnxx and others.)

```

```

\__kernel_msg_expandable_error:nn
\__kernel_msg_expandable_error:nnf
\__kernel_msg_expandable_error:nnff
9786 \cs_new:Npn \__kernel_msg_expandable_error:nnn #1
9787     { \msg_expandable_error:nnn { LaTeX / #1 } }
9788 \cs_new:Npn \__kernel_msg_expandable_error:nnf #1
9789     { \msg_expandable_error:nnf { LaTeX / #1 } }
9790 \cs_new:Npn \__kernel_msg_expandable_error:nnff #1
9791     { \msg_expandable_error:nnff { LaTeX / #1 } }

(End definition for \__kernel_msg_expandable_error:nn and \__kernel_msg_expandable_error:nnff.)

```

## 48.7 Internal messages

Error messages needed to actually implement the message system itself.

```

9792 \msg_new:nnnn { msg } { already-defined }
9793   { Message~'#2'~for~module~'#1'~already~defined. }
9794   {
9795     \c__msg_coding_error_text_tl
9796     LaTeX~was~asked~to~define~a~new~message~called~'#2'\\
9797     by~the~module~'#1':~this~message~already~exists.
9798     \c__msg_return_text_tl
9799   }
9800 \msg_new:nnnn { msg } { unknown }
9801   { Unknown~message~'#2'~for~module~'#1'. }
9802   {
9803     \c__msg_coding_error_text_tl
9804     LaTeX~was~asked~to~display~a~message~called~'#2'\\
9805     by~the~module~'#1':~this~message~does~not~exist.
9806     \c__msg_return_text_tl
9807   }
9808 \msg_new:nnnn { msg } { class-unknown }
9809   { Unknown~message~class~'#1'. }
9810   {
9811     LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
9812     this~was~never~defined.
9813     \c__msg_return_text_tl
9814   }
9815 \msg_new:nnnn { msg } { redirect-loop }
9816   {
9817     Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
9818     \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
9819   }
9820   {
9821     Adding~the~message~redirection~ {#1} ~=>~ {#2}
9822     \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
9823     created~an~infinite~loop\\\\
9824     \iow_indent:n { #4 \\\\ }
9825   }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

9826 \msg_new:nnnn { kernel } { bad-number-of-arguments }
9827   { Function~'#1'~cannot~be~defined~with~#2~arguments. }
9828   {
9829     \c__msg_coding_error_text_tl
9830     LaTeX~has~been~asked~to~define~a~function~'#1'~with~
9831     #2~arguments.~
9832     TeX~allows~between~0~and~9~arguments~for~a~single~function.
9833   }
9834 \msg_new:nnnn { kernel } { command-already-defined }
9835   { Control~sequence~'#1'~already~defined. }
9836   {
9837     \c__msg_coding_error_text_tl
9838     LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
9839     but~this~name~has~already~been~used~elsewhere. \\ \\
9840     The~current~meaning~is:\\

```

```

9841      \\ #2
9842    }
9843 \msg_new:nnnn { kernel } { command-not-defined }
9844   { Control-sequence~#1~undefined. }
9845   {
9846     \c__msg_coding_error_text_tl
9847     LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\\
9848     this~has~not~been~defined~yet.
9849   }
9850 \msg_new:nnnn { kernel } { empty-search-pattern }
9851   { Empty~search~pattern. }
9852   {
9853     \c__msg_coding_error_text_tl
9854     LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
9855     would~lead~to~an~infinite~loop!
9856   }
9857 \cs_if_exist:NF \tex_elapsedtime:D
9858   {
9859     \msg_new:nnnn { kernel } { noelapsed-time }
9860       { No~clock~detected~for~#1. }
9861       { The~current~engine~provides~no~way~to~access~the~system~time. }
9862   }
9863 \msg_new:nnnn { kernel } { non-base-function }
9864   { Function~'#1'~is~not~a~base~function }
9865   {
9866     \c__msg_coding_error_text_tl
9867     Functions~defined~through~\iow_char:N\cs_new:Nn~must~have~
9868     a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
9869     The~signature~'#2'~of~'#1'~contains~other~arguments~'#3'.~
9870     To~define~variants~use~\iow_char:N\cs_generate_variant:Nn~
9871     and~to~define~other~functions~use~\iow_char:N\cs_new:Npn.
9872   }
9873 \msg_new:nnnn { kernel } { missing-colon }
9874   { Function~'#1'~contains~no~':'. }
9875   {
9876     \c__msg_coding_error_text_tl
9877     Code-level~functions~must~contain~':~to~separate~the~
9878     argument~specification~from~the~function~name.~This~is~
9879     needed~when~defining~conditionals~or~variants,~or~when~building~a~
9880     parameter~text~from~the~number~of~arguments~of~the~function.
9881   }
9882 \msg_new:nnnn { kernel } { overflow }
9883   { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
9884   {
9885     An~attempt~was~made~to~store~#3~
9886     \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'.~
9887     The~largest~allowed~value~#4~will~be~used~instead.
9888   }
9889 \msg_new:nnnn { kernel } { out-of-bounds }
9890   { Access~to~an~entry~beyond~an~array's~bounds. }
9891   {
9892     An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
9893     array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
9894   }

```

```

9895 \msg_new:nnn { kernel } { protected-predicate }
9896   { Predicate~'#1'~must~be~expandable. }
9897   {
9898     \c__msg_coding_error_text_tl
9899     LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
9900     Only~expandable~tests~can~have~a~predicate~version.
9901   }
9902 \msg_new:nnn { kernel } { randint-backward-range }
9903   { Wrong~order~of~bounds~in~\iow_char:N\int_rand:nn{#1}{#2}. }
9904 \msg_new:nnn { kernel } { conditional-form-unknown }
9905   { Conditional~form~'#1'~for~function~'#2'~unknown. }
9906   {
9907     \c__msg_coding_error_text_tl
9908     LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
9909     the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
9910   }
9911 \msg_new:nnn { kernel } { variant-too-long }
9912   { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
9913   {
9914     \c__msg_coding_error_text_tl
9915     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9916     with~a~signature~starting~with~'#1',~but~that~is~longer~than~
9917     the~signature~(part~after~the~colon)~of~'#2'.
9918   }
9919 \msg_new:nnn { kernel } { invalid-variant }
9920   { Variant~form~'#1'~invalid~for~base~form~'#2'. }
9921   {
9922     \c__msg_coding_error_text_tl
9923     LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
9924     with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
9925     from~type~'#3'~to~type~'#4'.
9926   }
9927 \msg_new:nnn { kernel } { invalid-exp-args }
9928   { Invalid~variant~specifier~'#1'~in~'#2'. }
9929   {
9930     \c__msg_coding_error_text_tl
9931     LaTeX~has~been~asked~to~create~an~\iow_char:N\exp_args:N...~
9932     function~with~signature~'N#2'~but~'#1'~is~not~a~valid~argument~
9933     specifier.
9934   }
9935 \msg_new:nnn { kernel } { deprecated-variant }
9936   {
9937     Variant~form~'#1'~deprecated~for~base~form~'#2'.~
9938     One~should~not~change~an~argument~from~type~'#3'~to~type~'#4',
9939     \str_case:nnF {#3}
9940     {
9941       { n } { :~use~a~'\token_if_eqCharCode:NNTF #4 c v V'~variant? }
9942       { N } { :~base~form~only~accepts~a~single~token~argument. }
9943       {#4} { :~base~form~is~already~a~variant. }
9944     } { . }
9945   }
9946 \msg_new:nnn { char } { active }
9947   { Cannot~generate~active~chars. }
9948 \msg_new:nnn { char } { invalid-catcode }

```

```

9949   { Invalid~catcode~for~char~generation. }
9950   \msg_new:nnn { char } { null-space }
9951   { Cannot~generate~null~char~as~a~space. }
9952   \msg_new:nnn { char } { out-of-range }
9953   { Charcode~requested~out~of~engine~range. }
9954   \msg_new:nnnn { kernel } { quote-in-shell }
9955   { Quotes~in~shell~command~'#1'. }
9956   { Shell~commands~cannot~contain~quotes~("). }
9957   \msg_new:nnnn { keys } { no-property }
9958   { No~property~given~in~definition~of~key~'#1'. }
9959   {
9960     \c_msg_error_text_tl
9961     Inside~\keys_define:nn each~key~name~
9962     needs~a~property: \\ \\
9963     \iow_indent:n { #1 .<property>} \\ \\
9964     LaTeX~did~not~find~a~'.~to~indicate~the~start~of~a~property.
9965   }
9966   \msg_new:nnnn { keys } { property-boolean-values-only }
9967   { The~property~'#1'~accepts~boolean~values~only. }
9968   {
9969     \c_msg_error_text_tl
9970     The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
9971   }
9972   \msg_new:nnnn { keys } { property-requires-value }
9973   { The~property~'#1'~requires~a~value. }
9974   {
9975     \c_msg_error_text_tl
9976     LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
9977     No~value~was~given~for~the~property,~and~one~is~required.
9978   }
9979   \msg_new:nnnn { keys } { property-unknown }
9980   { The~key~property~'#1'~is~unknown. }
9981   {
9982     \c_msg_error_text_tl
9983     LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
9984     this~property~is~not~defined.
9985   }
9986   \msg_new:nnnn { quark } { invalid-function }
9987   { Quark~test~function~'#1'~is~invalid. }
9988   {
9989     \c_msg_error_text_tl
9990     LaTeX~has~been~asked~to~create~quark~test~function~'#1'~
9991     \tl_if_empty:nTF {#2}
9992     { but~that~name~}
9993     { with~signature~'#2',~but~that~signature~}
9994     is~not~valid.
9995   }
9996   \__kernel_msg_new:nnn { quark } { invalid }
9997   { Invalid~quark~variable~'#1'. }
9998   \msg_new:nnnn { scanmark } { already-defined }
9999   { Scan~mark~'#1~already~defined. }
10000   {
10001     \c_msg_error_text_tl
10002     LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~

```

```

10003     but~this~name~has~already~been~used~for~a~scan~mark.
10004 }
10005 \msg_new:nnnn { seq } { shuffle-too-large }
10006   { The~sequence~#1~is~too~long~to~be~shuffled~by~TeX. }
10007 {
10008   TeX-has~ \int_eval:n { \c_max_register_int + 1 } ~
10009   toks-registers:~this~only~allows~to~shuffle~up~to~
10010   \int_use:N \c_max_register_int \ items.~
10011   The~list~will~not~be~shuffled.
10012 }
10013 \msg_new:nnnn { kernel } { variable-not-defined }
10014   { Variable~#1~undefined. }
10015 {
10016   \c__msg_coding_error_text_tl
10017   LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
10018   been~defined~yet.
10019 }
10020 \msg_new:nnnn { kernel } { bad-type }
10021   { Variable~'#1'~is~not~a~valid~#3. }
10022 {
10023   \c__msg_coding_error_text_tl
10024   The~variable~'#1'~with~\tl_if_empty:nTF {#4} {meaning} {value}\\\\\
10025   \iow_indent:n {#2}\\\\\
10026   should~be~a~#3~variable,~but~
10027   \tl_if_empty:nTF {#4}
10028     { it~is~not~\str_if_eq:nnF {#3} { bool } { ~a~short~macro } . }
10029   {
10030     it~does~not~have~the~correct~
10031     \str_if_eq:nnTF {#2} {#4}
10032       { category~codes. }
10033       { internal~structure:\\\\\iow_indent:n {#4} }
10034   }
10035 }
10036 \msg_new:nnnn { clist } { non-clist }
10037   { Variable~'#1'~is~not~a~valid~clist. }
10038 {
10039   \c__msg_coding_error_text_tl
10040   The~variable~'#1'~with~value\\\\\
10041   \iow_indent:n {#2}\\\\\
10042   should~be~a~clist~variable,~but~it~includes~empty~or~blank~items~
10043   without~braces.
10044 }

```

Some errors are only needed in package mode if debugging is enabled by one of the options `enable-debug`, `check-declarations`, `log-functions`, or on the contrary if debugging is turned off. In format mode the error is somewhat different.

```

10045 \msg_new:nnnn { debug } { enable-debug }
10046   { To~use~'#1'~set~the~'enable-debug'~option. }
10047 {
10048   The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
10049   some~internal~functions~in~expl3~have~been~appropriately~
10050   defined.~This~only~happens~if~one~of~the~options~
10051   'enable-debug',~'check-declarations'~or~'log-functions'~was~
10052   given~as~an~option:~see~the~main~expl3~documentation.

```

10053 }

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```
10054 \msg_new:nnn { kernel } { bad-exp-end-f }
10055   { Misused~\exp_end_continue_f:w or~:nw }
10056 \msg_new:nnn { kernel } { bad-variable }
10057   { Erroneous~variable~#1 used! }
10058 \msg_new:nnn { seq } { misused }
10059   { A~sequence~was~misused. }
10060 \msg_new:nnn { prop } { misused }
10061   { A~property~list~was~misused. }
10062 \msg_new:nnn { prg } { negative-replication }
10063   { Negative~argument~for~\iow_char:N\prg_replicate:nn. }
10064 \msg_new:nnn { prop } { prop-keyval }
10065   { Missing~'=~in~#1~(in~..._keyval:Nn') }
10066 \msg_new:nnn { kernel } { unknown-comparison }
10067   { Relation~#1~not~among~=,<,>,==,!~,<=,>=. }
10068 \msg_new:nnn { kernel } { zero-step }
10069   { Zero~step~size~for~function~#1. }
10070 \cs_if_exist:NF \tex_expanded:D
10071   {
10072     \msg_new:nnn { kernel } { e-type }
10073     { #1 ~ in~e-type~argument }
10074   }
```

Messages used by the "show" functions.

```
10075 \msg_new:nnn { clist } { show }
10076   {
10077     The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
10078     \tl_if_empty:nTF {#2}
10079       { is~empty \\>~ . }
10080       { contains~the~items~(without~outer~braces): #2 . }
10081   }
10082 \msg_new:nnn { intarray } { show }
10083   { The~integer~array~#1~contains~#2~items: \\ #3 . }
10084 \msg_new:nnn { prop } { show }
10085   {
10086     The~property~list~#1~
10087     \tl_if_empty:nTF {#2}
10088       { is~empty \\>~ . }
10089       { contains~the~pairs~(without~outer~braces): #2 . }
10090   }
10091 \msg_new:nnn { seq } { show }
10092   {
10093     The~sequence~#1~
10094     \tl_if_empty:nTF {#2}
10095       { is~empty \\>~ . }
10096       { contains~the~items~(without~outer~braces): #2 . }
10097   }
10098 \msg_new:nnn { kernel } { show-streams }
10099   {
10100     \tl_if_empty:nTF {#2} { No~ } { The~following~ }
10101     \str_case:nn {#1}
10102       {
```

```

10103      { ior } { input ~ }
10104      { iow } { output ~ }
10105      }
10106      streams~are~
10107      \tl_if_empty:nTF {#2} { open } { in-use: #2 . }
10108  }

System layer messages
10109 \msg_new:nnnn { sys } { backend-set }
10110   { Backend~configuration~already~set. }
10111   {
10112     Run-time~backend~selection~may~only~be~carried~out~once~during~a~run.~
10113     This~second~attempt~to~set~them~will~be~ignored.
10114   }
10115 \msg_new:nnnn { sys } { wrong-backend }
10116   { Backend~request~inconsistent~with~engine:~using~'#2'~backend. }
10117   {
10118     You~have~requested~backend~'#1',~but~this~is~not~suitable~for~use~with~the~
10119     active~engine.~LaTeX~will~use~the~'#2'~backend~instead.
10120   }

```

## 48.8 Expandable errors

`\__msg_expandable_error:nn` In expansion only context, we cannot use the normal means of reporting errors. Instead, we rely on a low-level TeX error caused by expanding a macro `\???` with parameter text “?” (this could be any token) which we used followed by something else (here, a space). This shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \???
          ! mypkg Error: The error message.

```

In other words, TeX is processing the argument of `\use:n`, which is `\???` *<space>* ! *<error type>* : *<error message>*.

```

10121 \cs_set_protected:Npn \__msg_tmp:w #1
10122   {
10123     \cs_new:Npn #1 ? { }
10124     \cs_new:Npn \__msg_expandable_error:nn ##1##2
10125     {
10126       \exp_after:wN \exp_after:wN
10127       \exp_after:wN \__msg_use_none_delimit_by_s_stop:w
10128       \use:n { #1 ~ ! ~ ##2 : ~ ##1 } \s__msg_stop
10129     }
10130   }
10131 \exp_args:Nc \__msg_tmp:w { ??? }

```

(End definition for `\__msg_expandable_error:nn`.)

The command built from the csname `\c__msg_text_prefix_tl #1 / #2` takes four arguments and builds the error text, which is fed to `\__msg_expandable_error:n` with appropriate expansion: just as for usual messages the arguments are first turned to strings, then the message is fully expanded. The module name also has to be determined.

```

10132 \exp_args_generate:n { oooo }
10133 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6

```

```

10134 {
10135   \exp_args:Nee \__msg_expandable_error:nn
10136   {
10137     \exp_args:Nc \exp_args:Noooo
10138     { \c__msg_text_prefix_tl #1 / #2 }
10139     { \tl_to_str:n {#3} }
10140     { \tl_to_str:n {#4} }
10141     { \tl_to_str:n {#5} }
10142     { \tl_to_str:n {#6} }
10143   }
10144   { \msg_error_text:n {#1} }
10145 }
10146 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
10147   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
10148 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
10149   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
10150 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
10151   { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
10152 \cs_new:Npn \msg_expandable_error:nn #1#2
10153   { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
10154 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
10155 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnfff }
10156 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
10157 \cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }

```

(End definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 83.)

## 48.9 Message formatting

```

10158 \prop_gput:Nnn \g_msg_module_name_prop { kernel } { LaTeX }
10159 \prop_gput:Nnn \g_msg_module_type_prop { kernel } { }
10160 \clist_map_inline:nn
10161   {
10162     char ,clist ,coffin ,debug ,deprecation ,msg ,
10163     quark ,prg ,prop ,scanmark ,seq ,sys
10164   }
10165   {
10166     \prop_gput:Nnn \g_msg_module_name_prop {#1} { LaTeX }
10167     \prop_gput:Nnn \g_msg_module_type_prop {#1} { }
10168   }
10169 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / cmd } { LaTeX }
10170 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / cmd } { }
10171 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX / ltcmd } { LaTeX }
10172 \prop_gput:Nnn \g_msg_module_type_prop { LaTeX / ltcmd } { }
10173 </package>

```

# Chapter 49

## I3file implementation

The following test files are used for this code: *m3file001*.

10174 ⟨\*package⟩

### 49.1 Input operations

10175 ⟨@@=ior⟩

#### 49.1.1 Variables and constants

\l\_\_ior\_internal\_tl Used as a short-term scratch variable.

10176 \tl\_new:N \l\_\_ior\_internal\_tl

(End definition for \l\_\_ior\_internal\_tl.)

\c\_\_ior\_term\_ior Reading from the terminal (with a prompt) is done using a positive but non-existent stream number. Unlike writing, there is no concept of reading from the log.

10177 \int\_const:Nn \c\_\_ior\_term\_ior { 16 }

(End definition for \c\_\_ior\_term\_ior.)

\g\_\_ior\_streams\_seq A list of the currently-available input streams to be used as a stack.

10178 \seq\_new:N \g\_\_ior\_streams\_seq

(End definition for \g\_\_ior\_streams\_seq.)

\l\_\_ior\_stream\_tl Used to recover the raw stream number from the stack.

10179 \tl\_new:N \l\_\_ior\_stream\_tl

(End definition for \l\_\_ior\_stream\_tl.)

\g\_\_ior\_streams\_prop The name of the file attached to each stream is tracked in a property list. To get the correct number of reserved streams in package mode the underlying mechanism needs to be queried. For L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> and plain T<sub>E</sub>X this data is stored in \count16: with the etex package loaded we need to subtract 1 as the register holds the number of the next stream to use. In ConT<sub>E</sub>Xt, we need to look at \count38 but there is no subtraction: like the original plain T<sub>E</sub>X/L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> mechanism it holds the value of the *last* stream allocated.

10180 \prop\_new:N \g\_\_ior\_streams\_prop

```

10181 \int_step_inline:nnn
10182   { 0 }
10183   {
10184     \cs_if_exist:NTF \contextversion
10185       { \tex_count:D 38 ~ }
10186       {
10187         \tex_count:D 16 ~ %
10188         \cs_if_exist:NT \loccount { - 1 }
10189       }
10190   }
10191   {
10192     \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
10193   }

```

(End definition for `\g__ior_streams_prop`.)

### 49.1.2 Stream management

**\ior\_new:N** Reserving a new stream is done by defining the name as equal to using the terminal.

```

\ior_new:c 10194 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c__ior_term_ior }
10195 \cs_generate_variant:Nn \ior_new:N { c }

```

(End definition for `\ior_new:N`. This function is documented on page 86.)

**\g\_tmpa\_ior** The usual scratch space.

```

\g_tmpb_ior 10196 \ior_new:N \g_tmpa_ior
10197 \ior_new:N \g_tmpb_ior

```

(End definition for `\g_tmpa_ior` and `\g_tmpb_ior`. These variables are documented on page 93.)

**\ior\_open:Nn** Use the conditional version, with an error if the file is not found.

```

\ior_open:cn 10198 \cs_new_protected:Npn \ior_open:Nn #1#2
10199   { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
10200 \cs_generate_variant:Nn \ior_open:Nn { c }

```

(End definition for `\ior_open:Nn`. This function is documented on page 86.)

**\l\_\_ior\_file\_name\_tl** Data storage.

```

10201 \tl_new:N \l__ior_file_name_tl

```

(End definition for `\l__ior_file_name_tl`.)

**\ior\_open:NnTF** An auxiliary searches for the file in the  $\text{\TeX}$ ,  $\text{\LaTeX}\text{2}_\varepsilon$  and  $\text{\LaTeX}3$  paths. Then pass the file found to the lower-level function which deals with streams. The `full_name` is empty when the file is not found.

```

10202 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10203   {
10204     \file_get_full_name:nNTF {#2} \l__ior_file_name_tl
10205     {
10206       \__kernel_ior_open:No #1 \l__ior_file_name_tl
10207       \prg_return_true:
10208     }
10209     { \prg_return_false: }
10210   }
10211 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }

```

(End definition for `\ior_open:NnTF`. This function is documented on page 86.)

`\__ior_new:N` Streams are reserved using `\newread` before they can be managed by `ior`. To prevent `ior` from being affected by redefinitions of `\newread` (such as done by the third-party package `morewrites`), this macro is saved here under a private name. The complicated code ensures that `\__ior_new:N` is not `\outer` despite plain TeX's `\newread` being `\outer`. For ConTeXt, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```
10212 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10213   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
10214 \cs_if_exist:NT \contextversion
10215   {
10216     \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
10217     \cs_gset_protected:Npn \__ior_new:N #1
10218     {
10219       \cs_undefine:N #1
10220       \__ior_new_aux:N #1
10221     }
10222 }
```

(End definition for `\__ior_new:N`.)

`\__kernel_ior_open:Nn` `\__kernel_ior_open:Nn` `\__ior_open_stream:Nn` The stream allocation itself uses the fact that there is a list of all of those available. Life gets more complex as it's important to keep things in sync. That is done using a two-part approach: any streams that have already been taken up by `ior` but are now free are tracked, so we first try those. If that fails, ask plain TeX or L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  for a new stream and use that number (after a bit of conversion).

```
10223 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
10224   {
10225     \ior_close:N #1
10226     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10227     { \__ior_open_stream:Nn #1 {#2} }
10228     {
10229       \__ior_new:N #1
10230       \__kernel_tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10231       \__ior_open_stream:Nn #1 {#2}
10232     }
10233   }
10234 \cs_generate_variant:Nn \__kernel_ior_open:Nn { No }
```

Here, we act defensively in case LuaTeX is in use with an extensionless file name.

```
10235 \cs_new_protected:Npx \__ior_open_stream:Nn #1#2
10236   {
10237     \tex_global:D \tex_chardef:D #1 = \exp_not:N \l__ior_stream_tl \scan_stop:
10238     \prop_gput:NVn \exp_not:N \g__ior_streams_prop #1 {#2}
10239     \tex_openin:D #1
10240     \sys_if_engine_luatex:TF
10241     { {#2} }
10242     { \exp_not:N \__kernel_file_name_quote:n {#2} \scan_stop: }
10243 }
```

(End definition for `\__kernel_ior_open:Nn` and `\__ior_open_stream:Nn`.)

**\ior\_close:N** Closing a stream means getting rid of it at the TeX level and removing from the various data structures. Unless the name passed is an invalid stream number (outside the range [0, 15]), it can be closed. On the other hand, it only gets added to the stack if it was not already there, to avoid duplicates building up.

```

10244 \cs_new_protected:Npn \ior_close:N #1
10245   {
10246     \int_compare:nT { -1 < #1 < \c__ior_term_ior }
10247     {
10248       \tex_closein:D #1
10249       \prop_gremove:NV \g__ior_streams_prop #1
10250       \seq_if_in:NVF \g__ior_streams_seq #1
10251         { \seq_gpush:NV \g__ior_streams_seq #1 }
10252       \cs_gset_eq:NN #1 \c__ior_term_ior
10253     }
10254   }
10255 \cs_generate_variant:Nn \ior_close:N { c }
```

(End definition for `\ior_close:N`. This function is documented on page 86.)

**\ior\_show:N**  
**\ior\_log:N** Seek the stream in the `\g__ior_streams_prop` list, then show the stream as open or closed accordingly.

```

\__ior_show>NN
10256 \cs_new_protected:Npn \ior_show:N { \__ior_show:NN \tl_show:n }
10257 \cs_generate_variant:Nn \ior_show:N { c }
10258 \cs_new_protected:Npn \ior_log:N { \__ior_show:NN \tl_log:n }
10259 \cs_generate_variant:Nn \ior_log:N { c }
10260 \cs_new_protected:Npn \__ior_show:NN #1#2
10261   {
10262     \__kernel_chk_defined:NT #2
10263     {
10264       \prop_get:NVNTF \g__ior_streams_prop #2 \l__ior_internal_tl
10265         {
10266           \exp_args:Nx #1
10267             { \token_to_str:N #2 ~ open: ~ \l__ior_internal_tl }
10268         }
10269         { \exp_args:Nx #1 { \token_to_str:N #2 ~ closed } }
10270     }
10271   }
```

(End definition for `\ior_show:N`, `\ior_log:N`, and `\__ior_show:NN`. These functions are documented on page 86.)

**\ior\_show\_list:** Show the property lists, but with some “pretty printing”. See the `l3msg` module. The first argument of the message is `ior` (as opposed to `iow`) and the second is empty if no read stream is open and non-empty (the list of streams formatted using `\msg_show-item_unbraced:nn`) otherwise. The code of the message `show-streams` takes care of translating `ior/iow` to English.

```

10272 \cs_new_protected:Npn \ior_show_list: { \__ior_list:N \msg_show:nnxxxx }
10273 \cs_new_protected:Npn \ior_log_list: { \__ior_list:N \msg_log:nnxxxx }
10274 \cs_new_protected:Npn \__ior_list:N #1
10275   {
10276     #1 { kernel } { show-streams }
10277     { ior }
10278     {
10279       \prop_map_function:NN \g__ior_streams_prop
```

```

10280           \msg_show_item_unbraced:nn
10281       }
10282   { } { }
10283 }

```

(End definition for `\ior_show_list:`, `\ior_log_list:`, and `\_ior_list:N`. These functions are documented on page 87.)

### 49.1.3 Reading input

`\if_eof:w` The primitive conditional

```
10284 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for `\if_eof:w`. This function is documented on page 93.)

`\ior_if_eof_p:N` To test if some particular input stream is exhausted the following conditional is provided.  
`\ior_if_eof:NTF` The primitive test can only deal with numbers in the range [0, 15] so we catch outliers (they are exhausted).

```

10285 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
10286 {
10287     \if_int_compare:w -1 < #1
10288     \if_int_compare:w #1 < \c__ior_term_ior
10289         \if_eof:w #1
10290             \prg_return_true:
10291         \else:
10292             \prg_return_false:
10293         \fi:
10294     \else:
10295         \prg_return_true:
10296     \fi:
10297     \else:
10298         \prg_return_true:
10299     \fi:
10300 }

```

(End definition for `\ior_if_eof:NTF`. This function is documented on page 90.)

`\ior_get:NN` And here we read from files.

```

\__ior_get:NN
10301 \cs_new_protected:Npn \ior_get:NN #1#2
10302     { \ior_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10303 \cs_new_protected:Npn \__ior_get:NN #1#2
10304     { \tex_read:D #1 to #2 }
10305 \prg_new_protected_conditional:Npnn \ior_get:NN #1#2 { T , F , TF }
10306 {
10307     \ior_if_eof:NTF #1
10308     { \prg_return_false: }
10309     {
10310         \__ior_get:NN #1 #2
10311         \prg_return_true:
10312     }
10313 }

```

(End definition for `\ior_get:NN`, `\__ior_get:NN`, and `\ior_get:NTF`. These functions are documented on page 87.)

**\ior\_str\_get:NN** **\\_\_ior\_str\_get:NN** Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

```

10314 \cs_new_protected:Npn \ior_str_get:NN #1#2
10315   { \ior_str_get:NNF #1 #2 { \tl_set:Nn #2 { \q_no_value } } }
10316 \cs_new_protected:Npn \__ior_str_get:NN #1#2
10317   {
10318     \exp_args:Nno \use:n
10319     {
10320       \int_set:Nn \tex_endlinechar:D { -1 }
10321       \tex_readline:D #1 to #2
10322       \int_set:Nn \tex_endlinechar:D
10323     } { \int_use:N \tex_endlinechar:D }
10324   }
10325 \prg_new_protected_conditional:Npnn \ior_str_get:NN #1#2 { T , F , TF }
10326   {
10327     \ior_if_eof:NTF #1
10328     { \prg_return_false: }
10329     {
10330       \__ior_str_get:NN #1 #2
10331       \prg_return_true:
10332     }
10333   }

```

(End definition for `\ior_str_get:NN`, `\__ior_str_get:NN`, and `\ior_str_get:NNTF`. These functions are documented on page 88.)

**\c\_\_ior\_term\_noprompt\_ior** For reading without a prompt.

```
10334 \int_const:Nn \c__ior_term_noprompt_ior { -1 }
```

(End definition for `\c__ior_term_noprompt_ior`.)

**\ior\_get\_term:nN** Getting from the terminal is better with pretty-printing.

```

10335 \cs_new_protected:Npn \ior_get_term:nN #1#2
10336   { \__ior_get_term:NnN \__ior_get>NN {#1} #2 }
10337 \cs_new_protected:Npn \ior_str_get_term:nN #1#2
10338   { \__ior_get_term:NnN \__ior_str_get>NN {#1} #2 }
10339 \cs_new_protected:Npn \__ior_get_term:NnN #1#2#3
10340   {
10341     \group_begin:
10342     \tex_escapechar:D = -1 \scan_stop:
10343     \tl_if_blank:nTF {#2}
10344     { \exp_args:NNe #1 \c__ior_term_noprompt_ior }
10345     { \exp_args:NNe #1 \c__ior_term_ior }
10346     {#2}
10347     \exp_args:NNNv \group_end:
10348     \tl_set:Nn #3 {#2}
10349   }

```

(End definition for `\ior_get_term:nN`, `\ior_str_get_term:nN`, and `\__ior_get_term:NnN`. These functions are documented on page 307.)

**\ior\_map\_break:** Usual map breaking functions.

```

10350 \cs_new:Npn \ior_map_break:
10351   { \prg_map_break:Nn \ior_map_break: { } }
10352 \cs_new:Npn \ior_map_break:n
10353   { \prg_map_break:Nn \ior_map_break: }
```

(End definition for `\ior_map_break`: and `\ior_map_break:n`. These functions are documented on page 89.)

`\ior_map_inline:Nn`  
`\ior_str_map_inline:Nn`  
`\_ior_map_inline>NNn`  
`\_ior_map_inline>NNNn`  
`\_ior_map_inline_loop>NNN`

Mapping over an input stream can be done on either a token or a string basis, hence the set up. Within that, there is a check to avoid reading past the end of a file, hence the two applications of `\ior_if_eof:N` and its lower-level analogue `\if_eof:w`. This mapping cannot be nested with twice the same stream, as the stream has only one “current line”.

```

10354 \cs_new_protected:Npn \ior_map_inline:Nn
10355   { \_ior_map_inline>NNn \ior_get:NN }
10356 \cs_new_protected:Npn \ior_str_map_inline:Nn
10357   { \_ior_map_inline>NNn \_ior_str_get:NN }
10358 \cs_new_protected:Npn \_ior_map_inline>NNn
10359   {
10360     \int_gincr:N \g_kernel_prg_map_int
10361     \exp_args:Nc \_ior_map_inline>NNNn
10362       { \_ior_map_ \int_use:N \g_kernel_prg_map_int :n }
10363   }
10364 \cs_new_protected:Npn \_ior_map_inline>NNNn #1#2#3#4
10365   {
10366     \cs_gset_protected:Npn #1 ##1 {[#4]
10367     \ior_if_eof:NF #3 { \_ior_map_inline_loop>NNN #1#2#3 }
10368     \prg_break_point:Nn \ior_map_break:
10369       { \int_gdecr:N \g_kernel_prg_map_int }
10370   }
10371 \cs_new_protected:Npn \_ior_map_inline_loop>NNN #1#2#3
10372   {
10373     #2 #3 \l_ior_internal_tl
10374     \if_eof:w #3
10375       \exp_after:wN \ior_map_break:
10376     \fi:
10377     \exp_args:No #1 \l_ior_internal_tl
10378     \_ior_map_inline_loop>NNN #1#2#3
10379   }

```

(End definition for `\ior_map_inline:Nn` and others. These functions are documented on page 88.)

Since the TeX primitive (`\read` or `\readline`) assigns the tokens read in the same way as a token list assignment, we simply call the appropriate primitive. The end-of-loop is checked using the primitive conditional for speed.

`\ior_map_variable>NNn`  
`\ior_str_map_variable>NNn`  
`\_ior_map_variable>NNNn`  
`\_ior_map_variable_loop>NNNn`

```

10380 \cs_new_protected:Npn \ior_map_variable>NNn
10381   { \_ior_map_variable>NNNn \ior_get:NN }
10382 \cs_new_protected:Npn \ior_str_map_variable>NNn
10383   { \_ior_map_variable>NNNn \ior_str_get:NN }
10384 \cs_new_protected:Npn \_ior_map_variable>NNNn #1#2#3#4
10385   {
10386     \ior_if_eof:NF #2 { \_ior_map_variable_loop>NNNn #1#2#3 {[#4] }
10387     \prg_break_point:Nn \ior_map_break: { }
10388   }
10389 \cs_new_protected:Npn \_ior_map_variable_loop>NNNn #1#2#3#4
10390   {
10391     #1 #2 #3
10392     \if_eof:w #2
10393       \exp_after:wN \ior_map_break:
10394     \fi:

```

```

10395      #4
10396      \__ior_map_variable_loop:NNn #1#2#3 {#4}
10397  }

```

(End definition for `\ior_map_variable:NNn` and others. These functions are documented on page 88.)

## 49.2 Output operations

```
10398 <@@=iow>
```

There is a lot of similarity here to the input operations, at least for many of the basics. Thus quite a bit is copied from the earlier material with minor alterations.

### 49.2.1 Variables and constants

`\l__iow_internal_tl` Used as a short-term scratch variable.

```
10399 \tl_new:N \l__iow_internal_tl
```

(End definition for `\l__iow_internal_tl`.)

`\c_log_iow`  
`\c_term_iow` Here we allocate two output streams for writing to the transcript file only (`\c_log_iow`) and to both the terminal and transcript file (`\c_term_iow`). Recent LuaTeX provide 128 write streams; we also use `\c_term_iow` as the first non-allowed write stream so its value depends on the engine.

```

10400 \int_const:Nn \c_log_iow { -1 }
10401 \int_const:Nn \c_term_iow
10402 {
10403   \bool_lazy_and:nnTF
10404     { \sys_if_engine_luatex_p: }
10405     { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
10406     { 128 }
10407     { 16 }
10408 }

```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 93.)

`\g__iow_streams_seq` A list of the currently-available output streams to be used as a stack.

```
10409 \seq_new:N \g__iow_streams_seq
```

(End definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
10410 \tl_new:N \l__iow_stream_tl
```

(End definition for `\l__iow_stream_tl`.)

`\g__iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```

10411 \prop_new:N \g__iow_streams_prop
10412 \int_step_inline:nnn
10413   { 0 }
10414   {
10415     \cs_if_exist:NTF \contextversion
10416       { \tex_count:D 39 ~ }
10417       {
10418         \tex_count:D 17 ~

```

```

10419          \cs_if_exist:NT \loccount { - 1 }
10420      }
10421  }
10422  {
10423      \prop_gput:Nnn \g__iow_streams_prop {#1} { Reserved-by-format }
10424  }

```

(End definition for `\g__iow_streams_prop`.)

### 49.2.2 Internal auxiliaries

`\s__iow_mark` Internal scan marks.

```

\scans_new:N \s__iow_mark
10425 \scans_new:N \s__iow_stop
10426

```

(End definition for `\s__iow_mark` and `\s__iow_stop`.)

`\_iow_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```

10427 \cs_new:Npn \_iow_use_i_delimit_by_s_stop:nw #1 #2 \s__iow_stop {#1}

```

(End definition for `\_iow_use_i_delimit_by_s_stop:nw`.)

`\q__iow_nil` Internal quarks.

```

10428 \quark_new:N \q__iow_nil

```

(End definition for `\q__iow_nil`.)

## 49.3 Stream management

`\iow_new:N` Reserving a new stream is done by defining the name as equal to writing to the terminal:  
`\iow_new:c` odd but at least consistent.

```

10429 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10430 \cs_generate_variant:Nn \iow_new:N { c }

```

(End definition for `\iow_new:N`. This function is documented on page 86.)

`\g_tmpa_iow` The usual scratch space.

```

\g_tmpb_iow
10431 \iow_new:N \g_tmpa_iow
10432 \iow_new:N \g_tmpb_iow

```

(End definition for `\g_tmpa_iow` and `\g_tmpb_iow`. These variables are documented on page 93.)

`\_iow_new:N` As for read streams, copy `\newwrite`, making sure that it is not `\outer`. For ConTeXt, we have to deal with the fact that `\newwrite` works like our own: it actually checks before altering definition.

```

10433 \exp_args:NNf \cs_new_protected:Npn \_iow_new:N
10434   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
10435 \cs_if_exist:NT \contextversion
10436   {
10437     \cs_new_eq:NN \_iow_new_aux:N \_iow_new:N
10438     \cs_gset_protected:Npn \_iow_new:N #1
10439     {
10440       \cs_undefine:N #1
10441       \_iow_new_aux:N #1
10442     }
10443   }

```

(End definition for `\_iow_new:N`.)

`\l_iow_file_name_tl` Data storage.

```
10444 \tl_new:N \l_iow_file_name_tl
```

(End definition for `\l_iow_file_name_tl`.)

**\iow\_open:Nn** **\iow\_open:cN** The same idea as for reading, but without the path and without the need to allow for a conditional version.

```
10445 \cs_new_protected:Npn \iow_open:Nn #1#2
10446   {
10447     \_kernel_tl_set:Nx \l_iow_file_name_tl
10448       { \_kernel_file_name_sanitize:n {#2} }
10449     \iow_close:N #1
10450     \seq_gpop:NNTF \g_iow_streams_seq \l_iow_stream_tl
10451       { \_iow_open_stream:NV #1 \l_iow_file_name_tl }
10452       {
10453         \_iow_new:N #1
10454         \_kernel_tl_set:Nx \l_iow_stream_tl { \int_eval:n {#1} }
10455         \_iow_open_stream:NV #1 \l_iow_file_name_tl
10456       }
10457   }
10458 \cs_generate_variant:Nn \iow_open:Nn { c }
10459 \cs_new_protected:Npn \_iow_open_stream:Nn #1#2
10460   {
10461     \tex_global:D \tex_chardef:D #1 = \l_iow_stream_tl \scan_stop:
10462     \prop_gput:NVn \g_iow_streams_prop #1 {#2}
10463     \tex_immediate:D \tex_openout:D
10464       #1 \_kernel_file_name_quote:n {#2} \scan_stop:
10465   }
10466 \cs_generate_variant:Nn \_iow_open_stream:Nn { NV }
```

(End definition for `\iow_open:Nn` and `\_iow_open_stream:Nn`. This function is documented on page 86.)

**\iow\_close:N** **\iow\_close:c** Closing a stream is not quite the reverse of opening one. First, the close operation is easier than the open one, and second as the stream is actually a number we can use it directly to show that the slot has been freed up.

```
10467 \cs_new_protected:Npn \iow_close:N #1
10468   {
10469     \int_compare:nT { \c_log_iow < #1 < \c_term_iow }
10470       {
10471         \tex_immediate:D \tex_closeout:D #1
10472         \prop_gremove:NV \g_iow_streams_prop #1
10473         \seq_if_in:NVF \g_iow_streams_seq #1
10474           { \seq_gpush:NV \g_iow_streams_seq #1 }
10475         \cs_gset_eq:NN #1 \c_term_iow
10476       }
10477   }
10478 \cs_generate_variant:Nn \iow_close:N { c }
```

(End definition for `\iow_close:N`. This function is documented on page 86.)

```

\iow_show:N Seek the stream in the \g__iow_streams_prop list, then show the stream as open or
\iow_log:N closed accordingly.
 $\_\_iow\_show:NN$ 
10479 \cs_new_protected:Npn \iow_show:N { \_\_iow_show:NN \tl_show:n }
10480 \cs_generate_variant:Nn \iow_show:N { c }
10481 \cs_new_protected:Npn \iow_log:N { \_\_iow_show:NN \tl_log:n }
10482 \cs_generate_variant:Nn \iow_log:N { c }
10483 \cs_new_protected:Npn \_\_iow_show:NN #1#2
10484 {
10485     \_\_kernel_chk_defined:NT #2
10486     {
10487         \prop_get:NVNTF \g__iow_streams_prop #2 \l__iow_internal_tl
10488         {
10489             \exp_args:Nx #1
10490             { \token_to_str:N #2 ~ open: ~ \l__iow_internal_tl }
10491         }
10492         { \exp_args:Nx #1 { \token_to_str:N #2 ~ closed } }
10493     }
10494 }

```

(End definition for \iow\_show:N, \iow\_log:N, and \\_\\_iow\_show:NN. These functions are documented on page 86.)

*\iow\_show\_list:* Done as for input, but with a copy of the auxiliary so the name is correct.

```

\iow_log_list:
 $\_\_iow\_list:N$ 
10495 \cs_new_protected:Npn \iow_show_list: { \_\_iow_list:N \msg_show:nxxxxx }
10496 \cs_new_protected:Npn \iow_log_list: { \_\_iow_list:N \msg_log:nxxxxx }
10497 \cs_new_protected:Npn \_\_iow_list:N #1
10498 {
10499     #1 { kernel } { show-streams }
10500     { iow }
10501     {
10502         \prop_map_function:NN \g__iow_streams_prop
10503         \msg_show_item_unbraced:nn
10504     }
10505     { } { }
10506 }

```

(End definition for \iow\_show\_list:, \iow\_log\_list:, and \\_\\_iow\_list:N. These functions are documented on page 87.)

### 49.3.1 Deferred writing

*\iow\_shipout\_x:Nn* First the easy part, this is the primitive, which expects its argument to be braced.

```

 $\iow\_shipout\_x:Nx$ 
10507 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
10508 { \tex_write:D #1 {#2} }
10509 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }

```

(End definition for \iow\_shipout\_x:Nn. This function is documented on page 90.)

*\iow\_shipout:Nn* With  $\varepsilon$ - $\text{\TeX}$  available deferred writing without expansion is easy.

```

 $\iow\_shipout:Nx$ 
10510 \cs_new_protected:Npn \iow_shipout:Nn #1#2
10511 { \tex_write:D #1 { \exp_not:n {#2} } }
10512 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }

```

(End definition for \iow\_shipout:Nn. This function is documented on page 90.)

### 49.3.2 Immediate writing

```
\__kernel_iow_with:Nnn
\__iow_with:nNnn
```

If the integer #1 is equal to #2, just leave #3 in the input stream. Otherwise, pass the old value to an auxiliary, which sets the integer to the new value, runs the code, and restores the integer.

```
10513 \cs_new_protected:Npn \__kernel_iow_with:Nnn #1#2
10514 {
10515     \int_compare:nNnTF {#1} = {#2}
10516     { \use:n }
10517     { \exp_args:No \__iow_with:nNnn { \int_use:N #1 } #1 {#2} }
10518 }
10519 \cs_new_protected:Npn \__iow_with:nNnn #1#2#3#4
10520 {
10521     \int_set:Nn #2 {#3}
10522     #4
10523     \int_set:Nn #2 {#1}
10524 }
```

(End definition for `\__kernel_iow_with:Nnn` and `\__iow_with:nNnn`.)

```
\iow_now:Nn
\iow_now:Nx
\iow_now:cn
\iow_now:cx
```

This routine writes the second argument onto the output stream without expansion. If this stream isn't open, the output goes to the terminal instead. If the first argument is no output stream at all, we get an internal error. We don't use the expansion done by `\write` to get the Nx variant, because it differs in subtle ways from x-expansion, namely, macro parameter characters would not need to be doubled. We set the `\newlinechar` to 10 using `\__kernel_iow_with:Nnn` to support formats such as plain TeX: otherwise, `\iow_newline:` would not work. We do not do this for `\iow_shipout:Nn` or `\iow_shipout_x:Nn`, as TeX looks at the value of the `\newlinechar` at shipout time in those cases.

```
10525 \cs_new_protected:Npn \iow_now:Nn #1#2
10526 {
10527     \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^J }
10528     { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10529 }
10530 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
```

(End definition for `\iow_now:Nn`. This function is documented on page 90.)

```
\iow_log:n
\iow_log:x
\iow_term:n
\iow_term:x
```

Writing to the log and the terminal directly are relatively easy.

```
10531 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
10532 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
10533 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
10534 \cs_new_protected:Npn \iow_term:n { \iow_now:Nn \c_term_iow }
```

(End definition for `\iow_log:n` and `\iow_term:n`. These functions are documented on page 90.)

### 49.3.3 Special characters for writing

```
\iow_newline:
```

Global variable holding the character that forces a new line when something is written to an output stream.

```
10535 \cs_new:Npn \iow_newline: { ^J }
```

(End definition for `\iow_newline:`. This function is documented on page 91.)

**\iow\_char:N** Function to write any escaped char to an output stream.

```
10536 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for `\iow_char:N`. This function is documented on page 91.)

#### 49.3.4 Hard-wrapping lines to a character count

The code here implements a generic hard-wrapping function. This is used by the messaging system, but is designed such that it is available for other uses.

**\l\_iow\_line\_count\_int** This is the “raw” number of characters in a line which can be written to the terminal. The standard value is the line length typically used by TeX Live and MiKTeX.

```
10537 \int_new:N \l_iow_line_count_int  
10538 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 92.)

**\l\_\_iow\_newline\_tl** The token list inserted to produce a new line, with the *<run-on text>*.

```
10539 \tl_new:N \l__iow_newline_tl
```

(End definition for `\l__iow_newline_tl`.)

**\l\_\_iow\_line\_target\_int** This stores the target line count: the full number of characters in a line, minus any part for a leader at the start of each line.

```
10540 \int_new:N \l__iow_line_target_int
```

(End definition for `\l__iow_line_target_int`.)

**\\_iow\_set\_indent:n** The `_indent` variables hold one indentation marker and its length. The `\_iow_unindent:w` auxiliary removes one indentation. The function `\_iow_set_indent:n` (that could possibly be public) sets the indentation in a consistent way. We set it to four spaces by default.

```
10541 \tl_new:N \l__iow_one_indent_tl  
10542 \int_new:N \l__iow_one_indent_int  
10543 \cs_new:Npn \_iow_unindent:w { }  
10544 \cs_new_protected:Npn \_iow_set_indent:n #1  
10545 {  
10546     \_kernel_tl_set:Nx \l__iow_one_indent_tl  
10547     { \exp_args:No \_kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }  
10548     \int_set:Nn \l__iow_one_indent_int  
10549     { \str_count:N \l__iow_one_indent_tl }  
10550     \exp_last_unbraced:NNo  
10551     \cs_set:Npn \_iow_unindent:w \l__iow_one_indent_tl { }  
10552 }  
10553 \exp_args:Nx \_iow_set_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(End definition for `\_iow_set_indent:n` and others.)

**\l\_\_iow\_indent\_tl** The current indentation (some copies of `\l__iow_one_indent_tl`) and its number of characters.

```
10554 \tl_new:N \l__iow_indent_tl  
10555 \int_new:N \l__iow_indent_int
```

(End definition for `\l__iow_indent_tl` and `\l__iow_indent_int`.)

<code>\l_iow_line_tl</code>	These hold the current line of text and a partial line to be added to it, respectively.
<code>\l_iow_line_part_tl</code>	<pre>10556 \tl_new:N \l_iow_line_tl 10557 \tl_new:N \l_iow_line_part_tl</pre> <p>(End definition for <code>\l_iow_line_tl</code> and <code>\l_iow_line_part_tl</code>.)</p>
<code>\l_iow_line_break_bool</code>	Indicates whether the line was broken precisely at a chunk boundary.
	<pre>10558 \bool_new:N \l_iow_line_break_bool</pre> <p>(End definition for <code>\l_iow_line_break_bool</code>.)</p>
<code>\l_iow_wrap_tl</code>	Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.
	<pre>10559 \tl_new:N \l_iow_wrap_tl</pre> <p>(End definition for <code>\l_iow_wrap_tl</code>.)</p>
<code>\c_iow_wrap_marker_tl</code> <code>\c_iow_wrap_end_marker_tl</code> <code>\c_iow_wrap_newline_marker_tl</code> <code>\c_iow_wrap_allow_break_marker_tl</code> <code>\c_iow_wrap_indent_marker_tl</code> <code>\c_iow_wrap_unindent_marker_tl</code>	Every special action of the wrapping code starts with the same recognizable string, <code>\c_iow_wrap_marker_tl</code> . Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of <code>\escapechar</code> here is not very important, but makes <code>\c_iow_wrap_marker_tl</code> look marginally nicer.
	<pre>10560 \group_begin: 10561   \int_set:Nn \tex_escapechar:D { -1 } 10562   \tl_const:Nx \c_iow_wrap_marker_tl 10563     { \tl_to_str:n { \^I \^O \^W \^_ \^W \^R \^A \^P } } 10564 \group_end: 10565 \tl_map_inline:nn 10566   { { end } { newline } { allow_break } { indent } { unindent } } 10567   { 10568     \tl_const:cx { c_iow_wrap_ #1 _marker_tl } 10569     { 10570       \c_iow_wrap_marker_tl 10571       #1 10572       \c_catcode_other_space_tl 10573     } 10574 }</pre> <p>(End definition for <code>\c_iow_wrap_marker_tl</code> and others.)</p>
<code>\iow_allow_break:</code> <code>\_iow_allow_break:</code> <code>\_\_iow_allow_break_error:</code>	We set <code>\iow_allow_break:n</code> to produce an error when outside messages. Within wrapped message, it is set to <code>\_iow_allow_break:</code> when valid and otherwise to <code>\_\_iow_allow_break_error:</code> . The second produces an error expandably.
	<pre>10575 \cs_new_protected:Npn \iow_allow_break: 10576   { 10577     \msg_error:nnnn { kernel } { iow-indent } 10578     { \iow_wrap:nnN } { \iow_allow_break: } 10579   } 10580 \cs_new:Npx \_iow_allow_break: { \c_iow_wrap_allow_break_marker_tl } 10581 \cs_new:Npn \_\_iow_allow_break_error: 10582   { 10583     \msg_expandable_error:nnnn { kernel } { iow-indent } 10584     { \iow_wrap:nnN } { \iow_allow_break: } 10585 }</pre>

(End definition for `\iow_allow_break:`, `\_iow_allow_break:`, and `\\_iow_allow_break_error:`. This function is documented on page 307.)

`\iow_indent:n` We set `\iow_indent:n` to produce an error when outside messages. Within wrapped message, it is set to `\_iow_indent:n` when valid and otherwise to `\_iow_indent_error:n`. The first places the instruction for increasing the indentation before its argument, and the instruction for unindenting afterwards. The second produces an error expandably. Note that there are no forced line-break, so the indentation only changes when the next line is started.

```

10586 \cs_new_protected:Npn \iow_indent:n #1
10587   {
10588     \msg_error:nnnn { kernel } { iow-indent }
10589     { \iow_wrap:nnN } { \iow_indent:n } {#1}
10590     #1
10591   }
10592 \cs_new:Npx \_iow_indent:n #1
10593   {
10594     \c_iow_wrap_indent_marker_tl
10595     #1
10596     \c_iow_wrap_unindent_marker_tl
10597   }
10598 \cs_new:Npn \_iow_indent_error:n #1
10599   {
10600     \msg_expandable_error:nnnn { kernel } { iow-indent }
10601     { \iow_wrap:nnN } { \iow_indent:n } {#1}
10602     #1
10603   }

```

(End definition for `\iow_indent:n`, `\_iow_indent:n`, and `\_iow_indent_error:n`. This function is documented on page 92.)

`\iow_wrap:nnnN` The main wrapping function works as follows. First give `\\\`, `\_` and other formatting commands the correct definition for messages and perform the given setup #3. The definition of `\_` uses an “other” space rather than a normal space, because the latter might be absorbed by TeX to end a number or other f-type expansions. Use `\conditionally@traceoff` if defined; it is introduced by the `trace` package and suppresses uninteresting tracing of the wrapping code.

```

10604 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
10605   {
10606     \group_begin:
10607       \cs_if_exist_use:N \conditionally@traceoff
10608       \int_set:Nn \tex_escapechar:D { -1 }
10609       \cs_set:Npx \{ { \token_to_str:N \{ }
10610       \cs_set:Npx \# { \token_to_str:N \# }
10611       \cs_set:Npx \} { \token_to_str:N \} }
10612       \cs_set:Npx \% { \token_to_str:N \% }
10613       \cs_set:Npx \~ { \token_to_str:N \~ }
10614       \int_set:Nn \tex_escapechar:D { 92 }
10615       \cs_set_eq:NN \\ \iow_newline:
10616       \cs_set_eq:NN \ \c_catcode_other_space_tl
10617       \cs_set_eq:NN \iow_allow_break: \_iow_allow_break:
10618       \cs_set_eq:NN \iow_indent:n \_iow_indent:n
10619     #3

```

Then fully-expand the input: in package mode, the expansion uses L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub> 's `\protect` mechanism in the same way as `\typeout`. In generic mode this setting is useless but harmless. As soon as the expansion is done, reset `\iow_indent:n` to its error definition: it only works in the first argument of `\iow_wrap:nnN`.

```
10620      \cs_set_eq:NN \protect \token_to_str:N
10621      \_kernel_tl_set:Nx \l_iow_wrap_tl {#1}
10622      \cs_set_eq:NN \iow_allow_break: \_iow_allow_break_error:
10623      \cs_set_eq:NN \iow_indent:n \_iow_indent_error:n
```

Afterwards, set the newline marker (two assignments to fully expand, then convert to a string) and initialize the target count for lines (the first line has target count `\l_iow_line_count_int` instead).

```
10624      \_kernel_tl_set:Nx \l_iow_newline_tl { \iow_newline: #2 }
10625      \_kernel_tl_set:Nx \l_iow_newline_tl { \tl_to_str:N \l_iow_newline_tl }
10626      \int_set:Nn \l_iow_line_target_int
10627      { \l_iow_line_count_int - \str_count:N \l_iow_newline_tl + 1 }
```

Sanity check.

```
10628      \int_compare:nNnT { \l_iow_line_target_int } < 0
10629      {
10630          \tl_set:Nn \l_iow_newline_tl { \iow_newline: }
10631          \int_set:Nn \l_iow_line_target_int
10632          { \l_iow_line_count_int + 1 }
10633      }
```

There is then a loop over the input, which stores the wrapped result in `\l_iow_wrap_tl`. After the loop, the resulting text is passed on to the function which has been given as a post-processor. The `\tl_to_str:N` step converts the “other” spaces back to normal spaces. The f-expansion removes a leading space from `\l_iow_wrap_tl`.

```
10634      \_iow_wrap_do:
10635      \exp_args:Nnf \group_end:
10636      #4 { \tl_to_str:N \l_iow_wrap_tl }
10637  }
10638 \cs_generate_variant:Nn \iow_wrap:nnN { nx }
```

(End definition for `\iow_wrap:nnN`. This function is documented on page 92.)

`\_iow_wrap_do:` Escape spaces and change newlines to `\c_iow_wrap_newline_marker_tl`. Set up a few variables, in particular the initial value of `\l_iow_wrap_tl`: the space stops the f-expansion of the main wrapping function and `\use_none:n` removes a newline marker inserted by later code. The main loop consists of repeatedly calling the `chunk` auxiliary to wrap chunks delimited by (newline or indentation) markers.

```
10639 \cs_new_protected:Npn \_iow_wrap_do:
10640  {
10641      \_kernel_tl_set:Nx \l_iow_wrap_tl
10642      {
10643          \exp_args:No \_kernel_str_to_other_fast:n \l_iow_wrap_tl
10644          \c_iow_wrap_end_marker_tl
10645      }
10646      \_kernel_tl_set:Nx \l_iow_wrap_tl
10647      {
10648          \exp_after:wN \_iow_wrap_fix_newline:w \l_iow_wrap_tl
10649          ^^J \q_iow_nil ^^J \s_iow_stop
10650      }
```

```

10651      \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
10652    }
10653 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
10654  {
10655    #1
10656    \if_meaning:w \q__iow_nil #2
10657      \__iow_use_i_delimit_by_s_stop:nw
10658    \fi:
10659    \c__iow_wrap_newline_marker_tl
10660    \__iow_wrap_fix_newline:w #2 ^^J
10661  }
10662 \cs_new_protected:Npn \__iow_wrap_start:w
10663  {
10664    \bool_set_false:N \l__iow_line_break_bool
10665    \tl_clear:N \l__iow_line_tl
10666    \tl_clear:N \l__iow_line_part_tl
10667    \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
10668    \int_zero:N \l__iow_indent_int
10669    \tl_clear:N \l__iow_indent_tl
10670    \__iow_wrap_chunk:nw { \l_iow_line_count_int }
10671  }

```

(End definition for `\__iow_wrap_do:`, `\__iow_wrap_fix_newline:w`, and `\__iow_wrap_start:w`.)

`\__iow_wrap_chunk:nw`  
`\__iow_wrap_next:nw`

The `chunk` and `next` auxiliaries are defined indirectly to obtain the expansions of `\c_catcode_other_space_tl` and `\c__iow_wrap_marker_tl` in their definition. The `next` auxiliary calls a function corresponding to the type of marker (its `##2`), which can be `newline` or `indent` or `unindent` or `end`. The first argument of the `chunk` auxiliary is a target number of characters and the second is some string to wrap. If the chunk is empty simply call `next`. Otherwise, set up a call to `\__iow_wrap_line:nw`, including the indentation if the current line is empty, and including a trailing space (#1) before the `\__iow_wrap_end_chunk:w` auxiliary.

```

10672 \cs_set_protected:Npn \__iow_tmp:w #1#2
10673  {
10674    \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
10675    {
10676      \tl_if_empty:nTF {##2}
10677      {
10678        \tl_clear:N \l__iow_line_part_tl
10679        \__iow_wrap_next:nw {##1}
10680      }
10681    {
10682      \tl_if_empty:NTF \l__iow_line_tl
10683      {
10684        \__iow_wrap_line:nw
10685        { \l__iow_indent_tl }
10686        ##1 - \l__iow_indent_int ;
10687      }
10688      { \__iow_wrap_line:nw { } ##1 ; }
10689      ##2 #1
10690      \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \s__iow_stop
10691    }
10692  }
10693 \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1

```

```

10694     { \use:c { __iow_wrap_##2:n } {##1} }
10695   }
10696 \exp_args:NVV \__iow_tmp:w \c_catcode_other_space_tl \c__iow_wrap_marker_tl
(End definition for \__iow_wrap_chunk:nw and \__iow_wrap_next:nw.)

```

```

\__iow_wrap_line:nw
\__iow_wrap_line_loop:w
\__iow_wrap_line_aux:Nw
  \__iow_wrap_line_seven:nnnnnn
  \__iow_wrap_line_end:Nnnnnnnn
\__iow_wrap_line_end:nw
\__iow_wrap_end_chunk:w

```

This is followed by `{⟨string⟩} ⟨intexpr⟩ ;`. It stores the `⟨string⟩` and up to `⟨intexpr⟩` characters from the current chunk into `\l__iow_line_part_tl`. Characters are grabbed 8 at a time and left in `\l__iow_line_part_tl` by the `line_loop` auxiliary. When  $k < 8$  remain to be found, the `line_aux` auxiliary calls the `line_end` auxiliary followed by (the single digit)  $k$ , then  $7 - k$  empty brace groups, then the chunk's remaining characters. The `line_end` auxiliary leaves  $k$  characters from the chunk in the line part, then ends the assignment. Ignore the `\use:none:nnnnnn` line for now. If the next character is a space the line can be broken there: store what we found into the result and get the next line. Otherwise some work is needed to find a break-point. So far we have ignored what happens if the chunk is shorter than the requested number of characters: this is dealt with by the `end_chunk` auxiliary, which gets treated like a character by the rest of the code. It ends up being called either as one of the arguments #2–#9 of the `line_loop` auxiliary or as one of the arguments #2–#8 of the `line_end` auxiliary. In both cases stop the assignment and work out how many characters are still needed. Notice that when we have exactly seven arguments to clean up, a `\exp_stop_f:` has to be inserted to stop the `\exp:w`. The weird `\use:none:nnnnnn` ensures that the required data is in the right place.

```

10697 \cs_new_protected:Npn \__iow_wrap_line:nw #
10698   {
10699     \tex_eodef:D \l__iow_line_part_tl { \if_false: } \fi:
10700     #1
10701     \exp_after:wN \__iow_wrap_line_loop:w
10702     \int_value:w \int_eval:w
10703   }
10704 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
10705   {
10706     \if_int_compare:w #1 < 8 \exp_stop_f:
10707       \__iow_wrap_line_aux:Nw #1
10708     \fi:
10709     #2 #3 #4 #5 #6 #7 #8 #9
10710     \exp_after:wN \__iow_wrap_line_loop:w
10711     \int_value:w \int_eval:w #1 - 8 ;
10712   }
10713 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
10714   {
10715     #2
10716     \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
10717     \exp_after:wN #1
10718     \exp:w \exp_end_continue_f:w
10719     \exp_after:wN \exp_after:wN
10720     \if_case:w #1 \exp_stop_f:
10721       \prg_do_nothing:
10722     \or: \use:none:n
10723     \or: \use:none:nn
10724     \or: \use:none:nnn
10725     \or: \use:none:nnnn
10726     \or: \use:none:nnnnn

```

```

10727     \or: \use_none:nnnnnn
10728     \or: \__iow_wrap_line_seven:nnnnnnn
10729     \fi:
10730     { } { } { } { } { } { } { } { } #3
10731   }
10732 \cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
10733 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnN #1#2#3#4#5#6#7#8#9
10734   {
10735     #2 #3 #4 #5 #6 #7 #8
10736     \use_none:nnnnn \int_eval:w 8 - ; #9
10737     \token_if_eq_charcode:NNTF \c_space_token #9
10738     { \__iow_wrap_line_end:nw { } }
10739     { \if_false: { \fi: } \__iow_wrap_break:w #9 }
10740   }
10741 \cs_new:Npn \__iow_wrap_line_end:nw #1
10742   {
10743     \if_false: { \fi: }
10744     \__iow_wrap_store_do:n {#1}
10745     \__iow_wrap_next_line:w
10746   }
10747 \cs_new:Npn \__iow_wrap_end_chunk:w
10748   #1 \int_eval:w #2 - #3 ; #4#5 \s_iow_stop
10749   {
10750     \if_false: { \fi: }
10751     \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
10752   }

```

(End definition for `\__iow_wrap_line:nw` and others.)

`\__iow_wrap_break:w`, `\__iow_wrap_break_first:w`, `\__iow_wrap_break_none:w`, `\__iow_wrap_break_loop:w`, `\__iow_wrap_break_end:w` Functions here are defined indirectly: `\__iow_tmp:w` is eventually called with an “other” space as its argument. The goal is to remove from `\l__iow_line_part_tl` the part after the last space. In most cases this is done by repeatedly calling the `break_loop` auxiliary, which leaves “words” (delimited by spaces) until it hits the trailing space: then its argument `#3` is `? \__iow_wrap_break_end:w` instead of a single token, and that `break_end` auxiliary leaves in the assignment the line until the last space, then calls `\__iow_wrap_line_end:nw` to finish up the line and move on to the next. If there is no space in `\l__iow_line_part_tl` then the `break_first` auxiliary calls the `break_none` auxiliary. In that case, if the current line is empty, the complete word (including `#4`, characters beyond what we had grabbed) is added to the line, making it over-long. Otherwise, the word is used for the following line (and the last space of the line so far is removed because it was inserted due to the presence of a marker).

```

10753 \cs_set_protected:Npn \__iow_tmp:w #1
10754   {
10755     \cs_new:Npn \__iow_wrap_break:w
10756       {
10757         \tex_eodef:D \l__iow_line_part_tl
10758         \if_false: } \fi:
10759         \exp_after:wn \__iow_wrap_break_first:w
10760         \l__iow_line_part_tl
10761         #1
10762         { ? \__iow_wrap_break_end:w }
10763         \s_iow_mark
10764       }

```

```

10765 \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
10766 {
10767     \use_none:nn ##2 \__iow_wrap_break_none:w
10768     \__iow_wrap_break_loop:w ##1 #1 ##2
10769 }
10770 \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \s__iow_mark ##4 #1
10771 {
10772     \tl_if_empty:NTF \l__iow_line_tl
10773         { ##2 ##4 \__iow_wrap_line_end:nw { } }
10774         { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
10775 }
10776 \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
10777 {
10778     \use_none:n ##3
10779     ##1 #
10780     \__iow_wrap_break_loop:w ##2 #1 ##3
10781 }
10782 \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \s__iow_mark
10783     { ##1 \__iow_wrap_line_end:nw { } ##3 }
10784 }
10785 \exp_args:NV \__iow_tmp:w \c_catcode_other_space_tl

```

(End definition for `\__iow_wrap_break:w` and others.)

`\__iow_wrap_next_line:w` The special case where the end of a line coincides with the end of a chunk is detected here, to avoid a spurious empty line. Otherwise, call `\__iow_wrap_line:nw` to find characters for the next line (remembering to account for the indentation).

```

10786 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \s__iow_stop
10787 {
10788     \tl_clear:N \l__iow_line_tl
10789     \token_if_eq_meaning:NTF #1 \__iow_end_chunk:w
10790     {
10791         \tl_clear:N \l__iow_line_part_tl
10792         \bool_set_true:N \l__iow_line_break_bool
10793         \__iow_wrap_next:nw { \l__iow_line_target_int }
10794     }
10795     {
10796         \__iow_wrap_line:nw
10797             { \l__iow_indent_tl }
10798             \l__iow_line_target_int - \l__iow_indent_int ;
10799             #1 #2 \s__iow_stop
10800     }
10801 }

```

(End definition for `\__iow_wrap_next_line:w`.)

`\__iow_wrap_allow_break:n` This is called after a chunk has been wrapped. The `\l__iow_line_part_tl` typically ends with a space (except at the beginning of a line?), which we remove since the `allow_break` marker should not insert a space. Then move on with the next chunk, making sure to adjust the target number of characters for the line in case we did remove a space.

```

10802 \cs_new_protected:Npn \__iow_wrap_allow_break:n #1
10803 {
10804     \kernel_tl_set:Nx \l__iow_line_tl
10805         { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }

```

```

10806      \bool_set_false:N \l__iow_line_break_bool
10807      \tl_if_empty:NTF \l__iow_line_part_tl
10808          { \__iow_wrap_chunk:nw {#1} }
10809          { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
10810      }

```

(End definition for `\__iow_wrap_allow_break:n`.)

```
\__iow_wrap_indent:n  
\__iow_wrap_unindent:n
```

These functions are called after a chunk has been wrapped, when encountering `indent/unindent` markers. Add the line part (last line part of the previous chunk) to the line so far and reset a boolean denoting the presence of a line-break. Most importantly, add or remove one indent from the current indent (both the integer and the token list). Finally, continue wrapping.

```

10811 \cs_new_protected:Npn \__iow_wrap_indent:n #1
10812 {
10813     \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
10814     \bool_set_false:N \l__iow_line_break_bool
10815     \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10816     \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
10817     \__iow_wrap_chunk:nw {#1}
10818 }
10819 \cs_new_protected:Npn \__iow_wrap_unindent:n #1
10820 {
10821     \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
10822     \bool_set_false:N \l__iow_line_break_bool
10823     \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10824     \__kernel_tl_set:Nx \l__iow_indent_tl
10825         { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
10826     \__iow_wrap_chunk:nw {#1}
10827 }

```

(End definition for `\__iow_wrap_indent:n` and `\__iow_wrap_unindent:n`.)

```
\__iow_wrap_newline:n  
\__iow_wrap_end:n
```

These functions are called after a chunk has been line-wrapped, when encountering a `newline/end` marker. Unless we just took a line-break, store the line part and the line so far into the whole `\l__iow_wrap_tl`, trimming a trailing space. In the `newline` case look for a new line (of length `\l__iow_line_target_int`) in a new chunk.

```

10828 \cs_new_protected:Npn \__iow_wrap_newline:n #1
10829 {
10830     \bool_if:NF \l__iow_line_break_bool
10831         { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
10832     \bool_set_false:N \l__iow_line_break_bool
10833     \__iow_wrap_chunk:nw { \l__iow_line_target_int }
10834 }
10835 \cs_new_protected:Npn \__iow_wrap_end:n #1
10836 {
10837     \bool_if:NF \l__iow_line_break_bool
10838         { \__iow_wrap_store_do:n { \__iow_wrap_trim:N } }
10839     \bool_set_false:N \l__iow_line_break_bool
10840 }

```

(End definition for `\__iow_wrap_newline:n` and `\__iow_wrap_end:n`.)

\\_\\_iow\\_wrap\\_store\\_do:n First add the last line part to the line, then append it to \l\\_\\_iow\\_wrap\\_tl with the appropriate new line (with “run-on” text), possibly with its last space removed (#1 is empty or \\_\\_iow\\_wrap\\_trim:N).

```

10841 \cs_new_protected:Npn \_\_iow_wrap_store_do:n #1
10842 {
10843     \__kernel_tl_set:Nx \l\_\_iow_line_tl
10844     { \l\_\_iow_line_tl \l\_\_iow_line_part_tl }
10845     \__kernel_tl_set:Nx \l\_\_iow_wrap_tl
10846     {
10847         \l\_\_iow_wrap_tl
10848         \l\_\_iow_newline_tl
10849         #1 \l\_\_iow_line_tl
10850     }
10851     \tl_clear:N \l\_\_iow_line_tl
10852 }
```

(End definition for \\_\\_iow\\_wrap\\_store\\_do:n.)

\\_\\_iow\\_wrap\\_trim:N Remove one trailing “other” space from the argument if present.

```

10853 \cs_set_protected:Npn \_\_iow_tmp:w #1
10854 {
10855     \cs_new:Npn \_\_iow_wrap_trim:N ##1
10856     { \exp_after:wN \_\_iow_wrap_trim:w ##1 \s\_\_iow_mark #1 \s\_\_iow_mark \s\_\_iow_stop }
10857     \cs_new:Npn \_\_iow_wrap_trim:w ##1 #1 \s\_\_iow_mark
10858     { \_\_iow_wrap_trim_aux:w ##1 \s\_\_iow_mark }
10859     \cs_new:Npn \_\_iow_wrap_trim_aux:w ##1 \s\_\_iow_mark ##2 \s\_\_iow_stop {##1}
10860 }
10861 \exp_args:NV \_\_iow_tmp:w \c_catcode_other_space_tl
```

(End definition for \\_\\_iow\\_wrap\\_trim:N, \\_\\_iow\\_wrap\\_trim:w, and \\_\\_iow\\_wrap\\_trim\\_aux:w.)

10862 ⟨@=file⟩

## 49.4 File operations

\l\\_\\_file\\_internal\\_tl Used as a short-term scratch variable.

10863 \tl\_new:N \l\\_\\_file\_internal\_tl

(End definition for \l\\_\\_file\\_internal\\_tl.)

\g\\_\\_file\\_\\_curr\\_\\_dir\\_\\_str The name of the current file should be available at all times: the name itself is set dynamically.

\g\\_\\_file\\_\\_curr\\_\\_ext\\_\\_str  
\g\\_\\_file\\_\\_curr\\_\\_name\\_\\_str

```

10864 \str_new:N \g\_\_file\_\_curr\_\_dir\_\_str
10865 \str_new:N \g\_\_file\_\_curr\_\_ext\_\_str
10866 \str_new:N \g\_\_file\_\_curr\_\_name\_\_str
```

(End definition for \g\\_\\_file\\_\\_curr\\_\\_dir\\_\\_str, \g\\_\\_file\\_\\_curr\\_\\_ext\\_\\_str, and \g\\_\\_file\\_\\_curr\\_\\_name\\_\\_str. These variables are documented on page 93.)

\g\\_\\_file\\_\\_stack\\_\\_seq The input list of files is stored as a sequence stack. In package mode we can recover the information from the details held by L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  (we must be in the preamble and loaded using \usepackage or \RequirePackage). As L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  doesn’t store directory and name separately, we stick to the same convention here. In pre-loading, \@currnamestack is empty so is skipped.

```

10867 \seq_new:N \g__file_stack_seq
10868 \group_begin:
10869   \cs_set_protected:Npn \__file_tmp:w #1#2#3
10870   {
10871     \tl_if_blank:nTF {#1}
10872     {
10873       \cs_set:Npn \__file_tmp:w ##1 " ##2 " ##3 \s__file_stop
10874       { { } {##2} { } }
10875       \seq_gput_right:Nx \g__file_stack_seq
10876       {
10877         \exp_after:wN \__file_tmp:w \tex_jobname:D
10878         " \tex_jobname:D " \s__file_stop
10879       }
10880     }
10881     {
10882       \seq_gput_right:Nn \g__file_stack_seq { { } {#1} {#2} }
10883       \__file_tmp:w
10884     }
10885   }
10886   \cs_if_exist:NT \currnamestack
10887   {
10888     \tl_if_empty:NF \currnamestack
10889     { \exp_after:wN \__file_tmp:w \currnamestack }
10890   }
10891 \group_end:

```

(End definition for `\g__file_stack_seq`.)

#### `\g__file_record_seq`

The total list of files used is recorded separately from the current file stack, as nothing is ever popped from this list. The current file name should be included in the file list! We will eventually copy the contents of `\@filelist`.

```
10892 \seq_new:N \g__file_record_seq
```

(End definition for `\g__file_record_seq`.)

#### `\l__file_base_name_tl`

For storing the basename and full path whilst passing data internally.

```
10893 \tl_new:N \l__file_base_name_tl
10894 \tl_new:N \l__file_full_name_tl
```

(End definition for `\l__file_base_name_tl` and `\l__file_full_name_tl`.)

#### `\l__file_dir_str`

Used in parsing a path into parts: in contrast to the above, these are never used outside of the current module.

```
10895 \str_new:N \l__file_dir_str
10896 \str_new:N \l__file_ext_str
10897 \str_new:N \l__file_name_str
```

(End definition for `\l__file_dir_str`, `\l__file_ext_str`, and `\l__file_name_str`.)

#### `\l_file_search_path_seq`

The current search path.

```
10898 \seq_new:N \l_file_search_path_seq
```

(End definition for `\l_file_search_path_seq`. This variable is documented on page 94.)

#### `\l__file_tmp_seq`

Scratch space for comma list conversion.

```
10899 \seq_new:N \l__file_tmp_seq
```

(End definition for `\l__file_tmp_seq`.)

#### 49.4.1 Internal auxiliaries

\s\_\_file\_stop Internal scan marks.

```
10900 \scan_new:N \s__file_stop
```

(End definition for \s\_\_file\_stop.)

\q\_\_file\_nil Internal quarks.

```
10901 \quark_new:N \q__file_nil
```

(End definition for \q\_\_file\_nil.)

\\_\_file\_quark\_if\_nil\_p:n Branching quark conditional.

```
10902 \__kernel_quark_new_conditional:Nn \__file_quark_if_nil:n { TF }
```

(End definition for \\_\_file\_quark\_if\_nil:nTF.)

\q\_\_file\_recursion\_tail Internal recursion quarks.

```
10903 \quark_new:N \q__file_recursion_tail
```

```
10904 \quark_new:N \q__file_recursion_stop
```

(End definition for \q\_\_file\_recursion\_tail and \q\_\_file\_recursion\_stop.)

\\_\_file\_if\_recursion\_tail\_break:NN Functions to query recursion quarks.

```
10905 \__kernel_quark_new_test:N \__file_if_recursion_tail_stop:N
```

```
10906 \__kernel_quark_new_test:N \__file_if_recursion_tail_stop_do:nn
```

(End definition for \\_\_file\_if\_recursion\_tail\_break:NN and \\_\_file\_if\_recursion\_tail\_stop\_do:nn.)

Expanding the file name uses a \csname-based approach, and relies on active characters (for example from UTF-8 characters) being properly set up to expand to a expansion-safe version using \ifcsname. This is less conservative than the token-by-token approach used before, but it is much faster.

```
10907 \cs_new:Npn \__kernel_file_name_sanitize:n #1
```

```
10908 {
```

```
10909   \exp_args:Ne \__file_name_trim_spaces:n
```

```
10910   {
```

```
10911     \exp_args:Ne \__file_name_strip_quotes:n
```

```
10912     { \__file_name_expand:n {#1} }
```

```
10913   }
```

```
10914 }
```

We'll use \cs:w to start expanding the file name, and to avoid creating csnames equal to \relax with "common" names, there's a prefix \_\_file\_name= to the csname. There's also a guard token at the end so we can check if there was an error during the process and (try to) clean up gracefully.

```
10915 \cs_new:Npn \__file_name_expand:n #1
```

```
10916 {
```

```
10917   \exp_after:wN \__file_name_expand_cleanup:Nw
```

```
10918   \cs:w __file_name = #1 \cs_end:
```

```
10919   \__file_name_expand_end:
```

```
10920 }
```

With the csname built, we grab it, and grab the remaining tokens delimited by `\__file_name_expand_end`. If there are any remaining tokens, something bad happened, so we'll call the error procedure `\__file_name_expand_error:Nw`. If everything went according to plan, then use `\token_to_str:N` on the csname built, and call `\__file_name_expand_cleanup:w` to remove the prefix we added a while back. `\__file_name_expand_cleanup:w` takes a leading argument so we don't have to bother about the value of `\tex_escapechar:D`.

```

10921 \cs_new:Npn \__file_name_expand_cleanup:Nw #1 #2 \__file_name_expand_end:
10922   {
10923     \tl_if_empty:nF {#2}
10924       { \__file_name_expand_error:Nw #2 \__file_name_expand_end: }
10925     \exp_after:wN \__file_name_expand_cleanup:w \token_to_str:N #1
10926   }
10927 \exp_last_unbraced:NNNNo
10928 \cs_new:Npn \__file_name_expand_cleanup:w #1 \tl_to_str:n { __file_name = } { }
```

In non-error cases `\__file_name_expand_end:` should not expand. It will only do so in case there is a `\csname` too much in the file name, so it will throw an error (while expanding), then insert the missing `\cs_end:` and yet another `\__file_name_expand_end:` that will be used as a delimiter by `\__file_name_expand_cleanup:Nw` (or that will expand again if yet another `\endcsname` is missing).

```

10929 \cs_new:Npn \__file_name_expand_end:
10930   {
10931     \msg_expandable_error:nn
10932       { kernel } { filename-missing-endcsname }
10933     \cs_end: \__file_name_expand_end:
10934   }
```

Now to the error case. `\__file_name_expand_error:Nw` adds an extra `\cs_end:` so that in case there was an extra `\csname` in the file name, then `\__file_name_expand_error_aux:Nw` throws the error.

```

10935 \cs_new:Npn \__file_name_expand_error:Nw #1 #2 \__file_name_expand_end:
10936   { \__file_name_expand_error_aux:Nw #1 #2 \cs_end: \__file_name_expand_end: }
10937 \cs_new:Npn \__file_name_expand_error_aux:Nw #1 #2 \cs_end: #3
10938   \__file_name_expand_end:
10939   {
10940     \msg_expandable_error:nnff
10941       { kernel } { filename-chars-lost }
10942       { \token_to_str:N #1 } { \exp_stop_f: #2 }
10943   }
```

Quoting file name uses basically the same approach as for `luaquotejobname`: count the " tokens and remove them.

```

10944 \cs_new:Npn \__file_name_strip_quotes:n #1
10945   {
10946     \__file_name_strip_quotes:nw { 0 }
10947       #1 " \q_file_recursion_tail " \q_file_recursion_stop {#1}
10948   }
10949 \cs_new:Npn \__file_name_strip_quotes:nw #1#2 "
10950   {
10951     \if_meaning:w \q_file_recursion_tail #2
10952       \__file_name_strip_quotes_end:wnw
10953     \fi:
10954     #2
```

```

10955      \__file_name_strip_quotes:nw { #1 + 1 }
10956    }
10957 \cs_new:Npn \__file_name_strip_quotes_end:wnwn \fi: #1
10958   \__file_name_strip_quotes:nw #2 \q_file_recursion_stop #3
10959  {
10960    \fi:
10961    \int_if_odd:nT {#2}
10962    {
10963      \msg_expandable_error:nnn
10964      { kernel } { unbalanced-quote-in-filename } {#3}
10965    }
10966  }

```

Spaces need to be trimmed from the start of the name and from the end of any extension. However, the name we are passed might not have an extension: that means we have to look for one. If there is no extension, we still use the standard trimming function but deliberately prevent any spaces being removed at the end.

```

10967 \cs_new:Npn \__file_name_trim_spaces:n #1
10968  { \__file_name_trim_spaces:nw {#1} #1 . \q_file_nil . \s_file_stop }
10969 \cs_new:Npn \__file_name_trim_spaces:nw #1#2 . #3 . #4 \s_file_stop
10970  {
10971    \__file_quark_if_nil:nTF {#3}
10972    {
10973      \tl_trim_spaces_apply:nN {#1} \s_file_stop
10974      \__file_name_trim_spaces_aux:n
10975    }
10976    { \tl_trim_spaces:n {#1} }
10977  }
10978 \cs_new:Npn \__file_name_trim_spaces_aux:n #1
10979  { \__file_name_trim_spaces_aux:w #1 }
10980 \cs_new:Npn \__file_name_trim_spaces_aux:w #1 \s_file_stop {#1}

```

(End definition for `\__kernel_file_name_sanitize:n` and others.)

```

\__kernel_file_name_quote:n
\__file_name_quote:nw
10981 \cs_new:Npn \__kernel_file_name_quote:n #1
10982  { \__file_name_quote:nw {#1} #1 ~ \q_file_nil \s_file_stop }
10983 \cs_new:Npn \__file_name_quote:nw #1 #2 ~ #3 \s_file_stop
10984  {
10985    \__file_quark_if_nil:nTF {#3}
10986    { #1 }
10987    { "#1" }
10988  }

```

(End definition for `\__kernel_file_name_quote:n` and `\__file_name_quote:nw`.)

`\c_file_marker_tl` The same idea as the marker for rescanning token lists: this pair of tokens cannot appear in a file that is being input.

```
10989 \tl_const:Nx \c_file_marker_tl { : \token_to_str:N : }
```

(End definition for `\c_file_marker_tl`.)

`\file_get:nnNTF` `\file_get:nnN` The approach here is similar to that for `\tl_set_rescan:Nnn`. The file contents are grabbed as an argument delimited by `\c_file_marker_tl`. A few subtleties: braces in `\if_false: ... \fi:` to deal with possible alignment tabs, `\tracingnesting` to avoid

a warning about a group being closed inside the `\scantokens`, and `\prg_return_true:` is placed after the end-of-file marker.

```

10990 \cs_new_protected:Npn \file_get:nnN #1#2#3
10991 {
10992     \file_get:nnNF {#1} {#2} #3
10993     { \tl_set:Nn #3 { \q_no_value } }
10994 }
10995 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
10996 {
10997     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
10998     {
10999         \exp_args:NV \__file_get_aux:nnN
11000         \l__file_full_name_tl
11001         {#2} #3
11002         \prg_return_true:
11003     }
11004     { \prg_return_false: }
11005 }
11006 \cs_new_protected:Npx \__file_get_aux:nnN #1#2#3
11007 {
11008     \exp_not:N \if_false: { \exp_not:N \fi:
11009     \group_begin:
11010         \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
11011         \exp_not:N \exp_args:No \tex_everyeof:D
11012         { \exp_not:N \c__file_marker_tl }
11013         #2 \scan_stop:
11014         \exp_not:N \exp_after:wN \exp_not:N \__file_get_do:Nw
11015         \exp_not:N \exp_after:wN #3
11016         \exp_not:N \exp_after:wN \exp_not:N \prg_do_nothing:
11017         \exp_not:N \tex_input:D
11018         \sys_if_engine_luatex:TF
11019         { {#1} }
11020         { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11021         \exp_not:N \if_false: } \exp_not:N \fi:
11022     }
11023 \exp_args:Nno \use:nn
11024     { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
11025     { \c__file_marker_tl }
11026     {
11027         \group_end:
11028         \tl_set:No #1 {#2}
11029     }

```

(End definition for `\file_get:nnNTF` and others. These functions are documented on page 94.)

`\__file_size:n` A copy of the primitive where it's available.

```
11030 \cs_new_eq:NN \__file_size:n \tex_filesize:D
```

(End definition for `\__file_size:n`.)

### `\file_full_name:n`

```

\__file_full_name:n
\__file_full_name_aux:nn
\__file_full_name_aux:Nn
\__file_full_name_aux:nN
\__file_full_name_aux:nnN
    \__file_name_cleanup:w
        \__file_name_end:
\__file_name_ext_check:nn
\__file_name_ext_check:nw
\__file_name_ext_check:nnw
\__file_name_ext_check:nnn
\__file_name_ext_check:nnn
\__file_name_ext_check:nnnn

```

File searching can be carried out if the `\pdffilesize` primitive or an equivalent is available. That of course means we need to arrange for everything else to here to be done by expansion too. We start off by sanitizing the name and quoting if required: we may need to remove those quotes, so the raw name is passed too.

```

11031 \cs_new:Npn \file_full_name:n #1
11032 {
11033     \exp_args:Ne \__file_full_name:n
11034     { \__kernel_file_name_sanitize:n {#1} }
11035 }

```

First, we check if the file is just here: no mapping so we do not need the break part of the broader auxiliary. We are using the fact that the primitive here returns nothing if the file is entirely absent. To avoid unnecessary filesystem lookups, the result of \pdffilesize is kept available as an argument. For package mode, \input@path is a token list not a sequence.

```

11036 \cs_new:Npn \__file_full_name:n #1
11037 {
11038     \tl_if_blank:nF {#1}
11039     { \exp_args:Nne \__file_full_name_aux:nn {#1} { \__file_size:n {#1} } }
11040 }
11041 \cs_new:Npn \__file_full_name_aux:nn #1 #2
11042 {
11043     \tl_if_blank:nTF {#2}
11044     {
11045         \seq_map_tokens:Nn \l_file_search_path_seq
11046         { \__file_full_name_aux:Nnn \seq_map_break:n {#1} }
11047         \cs_if_exist:NT \input@path
11048         {
11049             \tl_map_tokens:Nn \input@path
11050             { \__file_full_name_aux:Nnn \tl_map_break:n {#1} }
11051         }
11052         \__file_name_end:
11053     }
11054     { \__file_ext_check:nn {#1} {#2} }
11055 }

```

Two pars to the auxiliary here so we can avoid doing quoting twice in the event we find the right file.

```

11056 \cs_new:Npn \__file_full_name_aux:Nnn #1#2#3
11057     { \exp_args:Ne \__file_full_name_aux:nN { \tl_to_str:n {#3} / #2 } #1 }
11058 \cs_new:Npn \__file_full_name_aux:nN #1
11059     { \exp_args:Nne \__file_full_name_aux:nnN {#1} { \__file_size:n {#1} } }
11060 \cs_new:Npn \__file_full_name_aux:nnN #1 #2 #3
11061 {
11062     \tl_if_blank:nF {#2}
11063     {
11064         #3
11065         {
11066             \__file_ext_check:nn {#1} {#2}
11067             \__file_name_cleanup:w
11068         }
11069     }
11070 }
11071 \cs_new:Npn \__file_name_cleanup:w #1 \__file_name_end: { }
11072 \cs_new:Npn \__file_name_end: { }

```

As TeX automatically adds .tex if there is no extension, there is a little clean up to do here. First, make sure we are not in the directory part, saving that. Then check for an extension.

```

11073 \cs_new:Npn \__file_ext_check:nn #1 #2
11074 { \__file_ext_check:nnw {#2} { / } #1 / \q_file_nil / \s_file_stop }
11075 \cs_new:Npn \__file_ext_check:nnw #1 #2 #3 / #4 / #5 \s_file_stop
11076 {
11077     \__file_quark_if_nil:nTF {#4}
11078 {
11079     \exp_args:No \__file_ext_check:nnnw
11080         { \use_none:n #2 } {#1} {#3} #3 . \q_file_nil . \s_file_stop
11081     }
11082     { \__file_ext_check:nnw {#1} { #2 #3 / } #4 / #5 \s_file_stop }
11083 }
11084 \cs_new:Npx \__file_ext_check:nnnw #1#2#3#4 . #5 . #6 \s_file_stop
11085 {
11086     \exp_not:N \__file_quark_if_nil:nTF {#5}
11087 {
11088     \exp_not:N \__file_ext_check:nnn
11089         { #1 #3 \tl_to_str:n { .tex } } { #1 #3 } {#2}
11090     }
11091     { #1 #3 }
11092 }
11093 \cs_new:Npn \__file_ext_check:nnn #1
11094 { \exp_args:Nne \__file_ext_check:nnnn {#1} { \__file_size:n {#1} } }
11095 \cs_new:Npn \__file_ext_check:nnnn #1#2#3#4
11096 {
11097     \tl_if_blank:nTF {#2}
11098     {#3}
11099 {
11100     \int_compare:nNnTF
11101         {#4} = {#2}
11102         {#1}
11103         {#3}
11104     }
11105 }

```

Deal with the fact that the primitive might not be available.

```

11106 \cs_if_exist:NF \tex_filesize:D
11107 {
11108     \cs_gset:Npn \file_full_name:n #1
11109 {
11110     \msg_expandable_error:nnn
11111         { kernel } { primitive-not-available }
11112         { \pdf filesize }
11113     }
11114 }
11115 \msg_new:nnnn { kernel } { primitive-not-available }
11116 { Primitive~\token_to_str:N #1 not~available }
11117 {
11118     The~version~of~your~TeX~engine~does~not~provide~functionality~equivalent~to~
11119     the~#1~primitive.
11120 }

```

(End definition for `\file_full_name:n` and others. This function is documented on page 94.)

`\file_get_full_name:nN`  
`\file_get_full_name:vN`  
`\file_get_full_name:nNTF`  
`\file_get_full_name:vNTF`  
`\__file_get_full_name_search:nN`

These functions pre-date using `\tex_filesize:D` for file searching, so are `get` functions with protection. To avoid having different search set ups, they are simply wrappers

around the code above.

```

11121 \cs_new_protected:Npn \file_get_full_name:nN #1#2
11122 {
11123     \file_get_full_name:nNF {#1} #2
11124     { \tl_set:Nn #2 { \q_no_value } }
11125 }
11126 \cs_generate_variant:Nn \file_get_full_name:nN { V }
11127 \prg_new_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
11128 {
11129     \__kernel_tl_set:Nx #2
11130     { \file_full_name:n {#1} }
11131     \tl_if_empty:NTF #2
11132     { \prg_return_false: }
11133     { \prg_return_true: }
11134 }
11135 \cs_generate_variant:Nn \file_get_full_name:nNT { V }
11136 \cs_generate_variant:Nn \file_get_full_name:nNF { V }
11137 \cs_generate_variant:Nn \file_get_full_name:nNTF { V }

```

If `\tex_filesize:D` is not available, the way to test if a file exists is to try to open it: if it does not exist then TeX reports end-of-file. A search is made looking at each potential path in turn (starting from the current directory). The first location is of course treated as the correct one: this is done by jumping to `\prg_break_point:`. If nothing is found, #2 is returned empty. A special case when there is no extension is that once the first location is found we test the existence of the file with `.tex` extension in that directory, and if it exists we include the `.tex` extension in the result.

```

11138 \cs_if_exist:NF \tex_filesize:D
11139 {
11140     \prg_set_protected_conditional:Npnn \file_get_full_name:nN #1#2 { T , F , TF }
11141     {
11142         \__kernel_tl_set:Nx \l__file_base_name_tl
11143         { \__kernel_file_name_sanitize:n {#1} }
11144         \__file_get_full_name_search:nN { } \use:n
11145         \seq_map_inline:Nn \l_file_search_path_seq
11146         { \__file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
11147         \cs_if_exist:NT \input@path
11148         {
11149             \tl_map_inline:Nn \input@path
11150             { \__file_get_full_name_search:nN { ##1 } \tl_map_break:n }
11151         }
11152         \tl_set:Nn \l__file_full_name_tl { \q_no_value }
11153         \prg_break_point:
11154         \quark_if_no_value:NTF \l__file_full_name_tl
11155         {
11156             \ior_close:N \g__file_internal_ior
11157             \prg_return_false:
11158         }
11159         {
11160             \file_parse_full_name:VNNN \l__file_full_name_tl
11161             \l__file_dir_str \l__file_name_str \l__file_ext_str
11162             \str_if_empty:NT \l__file_ext_str
11163             {
11164                 \__kernel_ior_open:No \g__file_internal_ior
11165                 { \l__file_full_name_tl .tex }

```

```

11166          \ior_if_eof:NF \g__file_internal_ior
11167              { \tl_put_right:Nn \l__file_full_name_tl { .tex } }
11168      }
11169      \ior_close:N \g__file_internal_ior
11170          \tl_set_eq:NN #2 \l__file_full_name_tl
11171          \prg_return_true:
11172      }
11173  }
11174 \cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
11175 {
11176     \__kernel_tl_set:Nx \l__file_full_name_tl
11177         { \tl_to_str:n {#1} \l__file_base_name_tl }
11178     \__kernel_ior_open:No \g__file_internal_ior \l__file_full_name_tl
11179         \ior_if_eof:NF \g__file_internal_ior { #2 { \prg_break: } }
11180     }
11181 }

```

(End definition for `\file_get_full_name:nN`, `\file_get_full_name:nNTF`, and `\__file_get_full_name_search:nN`. These functions are documented on page 94.)

`\g__file_internal_ior` A reserved stream to test for file existence (if required), and for opening a shell.

```
11182 \ior_new:N \g__file_internal_ior
```

(End definition for `\g__file_internal_ior`.)

`\file_mdfive_hash:n` Getting file details by expansion is relatively easy if a bit repetitive. As the MD5 function has a slightly different syntax from the other commands, there is a little cleaning up to do.

```

11183 \cs_new:Npn \file_size:n #1
11184     { \__file_details:nn {#1} { size } }
11185 \cs_new:Npn \file_timestamp:n #1
11186     { \__file_details:nn {#1} { moddate } }
11187 \cs_new:Npn \__file_details:nn #1#2
11188     {
11189         \exp_args:Ne \__file_details_aux:nn
11190             { \file_full_name:n {#1} } {#2}
11191     }
11192 \cs_new:Npn \__file_details_aux:nn #1#2
11193     {
11194         \tl_if_blank:nF {#1}
11195             { \use:c { tex_file #2 :D } {#1} }
11196     }
11197 \cs_new:Npn \file_mdfive_hash:n #1
11198     { \exp_args:Ne \__file_mdfive_hash:n { \file_full_name:n {#1} } }
11199 \cs_new:Npn \__file_mdfive_hash:n #1
11200     { \tex_md5fivesum:D file {#1} }

```

(End definition for `\file_mdfive_hash:n` and others. These functions are documented on page 96.)

`\file_hex_dump:nnn`

These are separate as they need multiple arguments or the file size. For LuaTeX, the emulation does not need the file size so we save a little on expansion.

```

11201 \cs_new:Npn \file_hex_dump:nnn #1#2#3
11202     {
11203         \exp_args:Neee \__file_hex_dump_auxi:nnn
11204             { \file_full_name:n {#1} }

```

```

11205      { \int_eval:n {#2} }
11206      { \int_eval:n {#3} }
11207    }
11208  \cs_new:Npn \__file_hex_dump_auxi:nnn #1#2#3
11209  {
11210    \bool_lazy_any:nF
11211    {
11212      { \tl_if_blank_p:n {#1} }
11213      { \int_compare_p:nNn {#2} = 0 }
11214      { \int_compare_p:nNn {#3} = 0 }
11215    }
11216    {
11217      \exp_args:Ne \__file_hex_dump_auxii:nnnn
11218      { \__file_details_aux:nn {#1} { size } }
11219      {#1} {#2} {#3}
11220    }
11221  }
11222 \cs_new:Npn \__file_hex_dump_auxii:nnnn #1#2#3#4
11223 {
11224   \int_compare:nNnTF {#3} > 0
11225   { \__file_hex_dump_auxiii:nnnn {#3} }
11226   {
11227     \exp_args:Ne \__file_hex_dump_auxiii:nnnn
11228     { \int_eval:n { #1 + #3 } }
11229   }
11230   {#1} {#2} {#4}
11231 }
11232 \cs_new:Npn \__file_hex_dump_auxiii:nnnn #1#2#3#4
11233 {
11234   \int_compare:nNnTF {#4} > 0
11235   { \__file_hex_dump_auxiv:nnn {#4} }
11236   {
11237     \exp_args:Ne \__file_hex_dump_auxiv:nnn
11238     { \int_eval:n { #2 + #4 } }
11239   }
11240   {#1} {#3}
11241 }
11242 \cs_new:Npn \__file_hex_dump_auxiv:nnn #1#2#3
11243 {
11244   \tex_filedump:D
11245   offset ~ \int_eval:n { #2 - 1 } ~
11246   length ~ \int_eval:n { #1 - #2 + 1 }
11247   {#3}
11248 }
11249 \cs_new:Npn \file_hex_dump:n #1
11250   { \exp_args:Ne \__file_hex_dump:n { \file_full_name:n {#1} } }
11251 \sys_if_engine_luatex:TF
11252 {
11253   \cs_new:Npn \__file_hex_dump:n #1
11254   {
11255     \tl_if_blank:nF {#1}
11256     { \tex_filedump:D whole {#1} {#1} }
11257   }
11258 }

```

```

11259     {
11260         \cs_new:Npn \__file_hex_dump:n #1
11261         {
11262             \tl_if_blank:nF {#1}
11263                 { \tex_filedump:D length \tex_filesize:D {#1} {#1} }
11264         }
11265     }

```

(End definition for `\file_hex_dump:nnn` and others. These functions are documented on page 95.)

```

\file_get_hex_dump:nN
\file_get_hex_dump:nNTF
le_get_mdfive_hash:nN\file_get_size:nN
_get_mdfive_hash:nN\file_get_size:nNTF
\file_get_timestamp:nN
\file_get_timestamp:nNTF
\__file_get_details:nnN

```

Non-expandable wrappers around the above in the case where appropriate primitive support exists.

```

11266 \cs_new_protected:Npn \file_get_hex_dump:nN #1#2
11267     { \file_get_hex_dump:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11268 \cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
11269     { \file_get_mdfive_hash:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11270 \cs_new_protected:Npn \file_get_size:nN #1#2
11271     { \file_get_size:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11272 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
11273     { \file_get_timestamp:nNF {#1} #2 { \tl_set:Nn #2 { \q_no_value } } }
11274 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nN #1#2 { T , F , TF }
11275     { \__file_get_details:nnN {#1} { hex_dump } #2 }
11276 \prg_new_protected_conditional:Npnn \file_get_mdfive_hash:nN #1#2 { T , F , TF }
11277     { \__file_get_details:nnN {#1} { mdfive_hash } #2 }
11278 \prg_new_protected_conditional:Npnn \file_get_size:nN #1#2 { T , F , TF }
11279     { \__file_get_details:nnN {#1} { size } #2 }
11280 \prg_new_protected_conditional:Npnn \file_get_timestamp:nN #1#2 { T , F , TF }
11281     { \__file_get_details:nnN {#1} { timestamp } #2 }
11282 \cs_new_protected:Npn \__file_get_details:nnN #1#2#3
11283     {
11284         \__kernel_tl_set:Nx #3
11285             { \use:c { file_ #2 :n } {#1} }
11286         \tl_if_empty:NTF #3
11287             { \prg_return_false: }
11288             { \prg_return_true: }
11289     }

```

Where the primitive is not available, issue an error: this is a little more conservative than absolutely needed, but does work.

```

11290 \cs_if_exist:NF \tex_filesize:D
11291     {
11292         \cs_set_protected:Npn \__file_get_details:nnN #1#2#3
11293         {
11294             \tl_clear:N #3
11295             \msg_error:nnx
11296                 { kernel } { primitive-not-available }
11297             {
11298                 \token_to_str:N \pdf_file
11299                 \str_case:nn {#2}
11300                     {
11301                         { hex_dump } { dump }
11302                         { mdfive_hash } { md5sum }
11303                         { timestamp } { moddate }
11304                         { size } { size }
11305                     }

```

```

11306         }
11307     \prg_return_false:
11308   }
11309 }
```

(End definition for `\file_get_hex_dump:nNtf` and others. These functions are documented on page 95.)

`\file_get_hex_dump:nNn` Custom code due to the additional arguments.

```

11310 \cs_new_protected:Npn \file_get_hex_dump:nNn #1#2#3#4
11311 {
11312   \file_get_hex_dump:nNnf {#1} {#2} {#3} #4
11313   { \tl_set:Nn #4 { \q_no_value } }
11314 }
11315 \prg_new_protected_conditional:Npnn \file_get_hex_dump:nNn #1#2#3#4
11316 { T , F , TF }
11317 {
11318   \__kernel_tl_set:Nx #4
11319   { \file_hex_dump:nNn {#1} {#2} {#3} }
11320   \tl_if_empty:NTF #4
11321   { \prg_return_false: }
11322   { \prg_return_true: }
11323 }
```

(End definition for `\file_get_hex_dump:nNtf`. This function is documented on page 95.)

`\__file_str_cmp:nn`

As we are doing a fixed-length “big” integer comparison, it is easiest to use the low-level behavior of string comparisons.

```
11324 \cs_new_eq:NN \__file_str_cmp:nn \tex_strcmp:D
```

(End definition for `\__file_str_cmp:nn`.)

`\file_compare_timestamp:p:nNn`

Comparison of file date can be done by using the low-level nature of the string comparison functions.

```

11325 \prg_new_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
11326 { p , T , F , TF }
11327 {
11328   \exp_args:Nee \__file_compare_timestamp:nNn
11329   { \file_full_name:n {#1} }
11330   { \file_full_name:n {#3} }
11331   #2
11332 }
11333 \cs_new:Npn \__file_compare_timestamp:nnN #1#2#3
11334 {
11335   \tl_if_blank:nTF {#1}
11336   {
11337     \if_charcode:w #3 <
11338     \prg_return_true:
11339   \else:
11340     \prg_return_false:
11341   \fi:
11342 }
11343 {
11344   \tl_if_blank:nTF {#2}
11345   {
11346     \if_charcode:w #3 >
```

```

11347           \prg_return_true:
11348       \else:
11349           \prg_return_false:
11350       \fi:
11351   }
11352   {
11353       \if_int_compare:w
11354           \__file_str_cmp:nn
11355               { \__file_timestamp:n {#1} }
11356               { \__file_timestamp:n {#2} }
11357               #3 \c_zero_int
11358           \prg_return_true:
11359       \else:
11360           \prg_return_false:
11361       \fi:
11362   }
11363 }
11364 }
11365 \cs_new_eq:NN \__file_timestamp:n \tex_filemoddate:D
11366 \cs_if_exist:NF \__file_timestamp:n
11367 {
11368     \prg_set_conditional:Npnn \file_compare_timestamp:nNn #1#2#3
11369         { p , T , F , TF }
11370     {
11371         \msg_expandable_error:nnn
11372             { kernel } { primitive-not-available }
11373             { \pdf filemoddate }
11374         \prg_return_false:
11375     }
11376 }

```

(End definition for `\file_compare_timestamp:nNnTF`, `\__file_compare_timestamp:nnN`, and `\__file_timestamp:n`. This function is documented on page 97.)

**\file\_if\_exist:nTF** The test for the existence of a file is a wrapper around the function to add a path to a file. If the file was found, the path contains something, whereas if the file was not located then the return value is empty.

```

11377 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
11378 {
11379     \file_get_full_name:nNTF {#1} \l__file_full_name_tl
11380     { \prg_return_true: }
11381     { \prg_return_false: }
11382 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 94.)

**\file\_if\_exist\_input:n** Input of a file with a test for existence. We do not define the T or TF variants because the most useful place to place the `<true code>` would be inconsistent with other conditionals.

```

11383 \cs_new_protected:Npn \file_if_exist_input:n #1
11384 {
11385     \file_get_full_name:nNT {#1} \l__file_full_name_tl
11386     { \__file_input:V \l__file_full_name_tl }
11387 }
11388 \cs_new_protected:Npn \file_if_exist_input:nF #1#2

```

```

11389     {
11390         \file_get_full_name:nNTF {#1} \l__file_full_name_t1
11391         { \__file_input:V \l__file_full_name_t1 }
11392         {#2}
11393     }

```

(End definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 97.)

**\file\_input\_stop:** A simple rename.

```
11394 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(End definition for `\file_input_stop:`. This function is documented on page 97.)

**\\_\_kernel\_file\_missing:n** An error message for a missing file, also used in `\ior_open:Nn`.

```

11395 \cs_new_protected:Npn \__kernel_file_missing:n #1
11396     {
11397         \msg_error:nnx { kernel } { file-not-found }
11398         { \__kernel_file_name_sanitize:n {#1} }
11399     }

```

(End definition for `\__kernel_file_missing:n`.)

**\file\_input:n** Loading a file is done in a safe way, checking first that the file exists and loading only if it does. Push the file name on the `\g_file_stack_seq`, and add it to the file list, either `\g_file_record_seq`, or `\@filelist` in package mode.

```

11400 \cs_new_protected:Npn \file_input:n #1
11401     {
11402         \file_get_full_name:nNTF {#1} \l__file_full_name_t1
11403         { \__file_input:V \l__file_full_name_t1 }
11404         { \__kernel_file_missing:n {#1} }
11405     }
11406 \cs_new_protected:Npx \__file_input:n #1
11407     {
11408         \exp_not:N \clist_if_exist:NTF \exp_not:N \@filelist
11409         { \exp_not:N \addtofilelist {#1} }
11410         { \seq_gput_right:Nn \exp_not:N \g_file_record_seq {#1} }
11411         \exp_not:N \__file_input_push:n {#1}
11412         \exp_not:N \tex_input:D
11413         \sys_if_engine_luatex:TF
11414         { {#1} }
11415         { \exp_not:N \__kernel_file_name_quote:n {#1} \scan_stop: }
11416         \exp_not:N \__file_input_pop:
11417     }
11418 \cs_generate_variant:Nn \__file_input:n { V }

```

Keeping a track of the file data is easy enough: we store the separated parts so we do not need to parse them twice.

```

11419 \cs_new_protected:Npn \__file_input_push:n #1
11420     {
11421         \seq_gpush:Nx \g_file_stack_seq
11422         {
11423             { \g_file_curr_dir_str }
11424             { \g_file_curr_name_str }
11425             { \g_file_curr_ext_str }

```

```

11426      }
11427      \file_parse_full_name:nNNN {#1}
11428          \l_file_dir_str \l_file_name_str \l_file_ext_str
11429          \str_gset_eq:NN \g_file_curr_dir_str \l_file_dir_str
11430          \str_gset_eq:NN \g_file_curr_name_str \l_file_name_str
11431          \str_gset_eq:NN \g_file_curr_ext_str \l_file_ext_str
11432      }
11433 \cs_new_eq:NN \_kernel_file_input_push:n \_file_input_push:n
11434 \cs_new_protected:Npn \_file_input_pop:
11435 {
11436     \seq_gpop>NN \g_file_stack_seq \l_file_internal_tl
11437     \exp_after:wN \_file_input_pop:nnn \l_file_internal_tl
11438 }
11439 \cs_new_eq:NN \_kernel_file_input_pop: \_file_input_pop:
11440 \cs_new_protected:Npn \_file_input_pop:nnn #1#2#3
11441 {
11442     \str_gset:Nn \g_file_curr_dir_str {#1}
11443     \str_gset:Nn \g_file_curr_name_str {#2}
11444     \str_gset:Nn \g_file_curr_ext_str {#3}
11445 }

```

(End definition for `\file_input:n` and others. This function is documented on page 97.)

`\file_parse_full_name:n`  
`\file_parse_full_name_apply:nN` The main parsing macro `\file_parse_full_name_apply:nN` passes the file name #1 through `\_kernel_file_name_sanitze:n` so that we have a single normalised way to treat files internally. `\file_parse_full_name:n` uses the former, with `\prg_do_nothing:` to leave each part of the name within a pair of braces.

```

11446 \cs_new:Npn \file_parse_full_name:n #1
11447 {
11448     \file_parse_full_name_apply:nN {#1}
11449     \prg_do_nothing:
11450 }
11451 \cs_new:Npn \file_parse_full_name_apply:nN #1
11452 {
11453     \exp_args:Ne \_file_parse_full_name_auxi:nN
11454     { \_kernel_file_name_sanitze:n {#1} }
11455 }

```

`\_file_parse_full_name_area:nw` splits the file name into chunks separated by /, until the last one is reached. The last chunk is the file name plus the extension, and everything before that is the path. When `\_file_parse_full_name_area:nw` is done, it leaves the path within braces after the scan mark `\s_file_stop` and proceeds parsing the actual file name.

```

11456 \cs_new:Npn \_file_parse_full_name_auxi:nN #1
11457 {
11458     \_file_parse_full_name_area:nw { } #1
11459     / \s_file_stop
11460 }
11461 \cs_new:Npn \_file_parse_full_name_area:nw #1 #2 / #3 \s_file_stop
11462 {
11463     \tl_if_empty:nTF {#3}
11464     { \_file_parse_full_name_base:nw { } #2 . \s_file_stop {#1} }
11465     { \_file_parse_full_name_area:nw { #1 / #2 } #3 \s_file_stop }
11466 }

```

`\__file_parse_full_name_base:nw` does roughly the same as above, but it separates the chunks at each period. However here there's some extra complications: In case #1 is empty, it is assumed that the extension is actually empty, and the file name is #2. Besides, an extra `.` has to be added to #2 because it is later removed in `\__file_parse_full_name_tidy:nnnN`. In any case, if there's an extension, it is returned with a leading `..`

```

11467 \cs_new:Npn \__file_parse_full_name_base:nw #1 #2 . #3 \s__file_stop
11468 {
11469     \tl_if_empty:nTF {#3}
11470     {
11471         \tl_if_empty:nTF {#1}
11472         {
11473             \tl_if_empty:nTF {#2}
11474             { \__file_parse_full_name_tidy:nnnN { } { } }
11475             { \__file_parse_full_name_tidy:nnnN { .#2 } { } }
11476         }
11477         { \__file_parse_full_name_tidy:nnnN {#1} { .#2 } }
11478     }
11479     { \__file_parse_full_name_base:nw { #1 . #2 } #3 \s__file_stop }
11480 }
```

Now we just need to tidy some bits left loose before. The loop used in the two macros above start with a leading `/` and `.` in the file path an name, so here we need to remove them, except in the path, if it is a single `/`, in which case it's left as is. After all's done, pass to #4.

```

11481 \cs_new:Npn \__file_parse_full_name_tidy:nnnN #1 #2 #3 #4
11482 {
11483     \exp_args:Nee #4
11484     {
11485         \str_if_eq:nnF {#3} { / } { \use_none:n }
11486         #3 \prg_do_nothing:
11487     }
11488     { \use_none:n #1 \prg_do_nothing: }
11489     {#2}
11490 }
```

(End definition for `\file_parse_full_name:n` and others. These functions are documented on page 95.)

### `\file_parse_full_name:nNNN`

### `\file_parse_full_name:VNNN`

```

11491 \cs_new_protected:Npn \file_parse_full_name:nNNN #1 #2 #3 #4
11492 {
11493     \file_parse_full_name_apply:nN {#1}
11494     \__file_full_name_assign:nnnNNN #2 #3 #4
11495 }
11496 \cs_new_protected:Npn \__file_full_name_assign:nnnNNN #1 #2 #3 #4 #5 #6
11497 {
11498     \str_set:Nn #4 {#1}
11499     \str_set:Nn #5 {#2}
11500     \str_set:Nn #6 {#3}
11501 }
11502 \cs_generate_variant:Nn \file_parse_full_name:nNNN { V }
```

(End definition for `\file_parse_full_name:nNNN`. This function is documented on page 95.)

```

\file_show_list: A function to list all files used to the log, without duplicates. In package mode, if
\file_log_list: \@filelist is still defined, we need to take this list of file names into account (we
\__file_list:N capture it \AtBeginDocument into \g__file_record_seq), turning it to a string (this
\__file_list_aux:n does not affect the commas of this comma list).

11503 \cs_new_protected:Npn \file_show_list: { \__file_list:N \msg_show:nnxxxx }
11504 \cs_new_protected:Npn \file_log_list: { \__file_list:N \msg_log:nnxxxx }
11505 \cs_new_protected:Npn \__file_list:N #1
11506   {
11507     \seq_clear:N \l__file_tmp_seq
11508     \clist_if_exist:NT \@filelist
11509     {
11510       \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
11511         { \tl_to_str:N \@filelist }
11512     }
11513     \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g__file_record_seq
11514     \seq_remove_duplicates:N \l__file_tmp_seq
11515     #1 { kernel } { file-list }
11516     { \seq_map_function:NN \l__file_tmp_seq \__file_list_aux:n }
11517     { } { } { }
11518   }
11519 \cs_new:Npn \__file_list_aux:n #1 { \iow_newline: #1 }

(End definition for \file_show_list: and others. These functions are documented on page 97.)

```

When used as a package, there is a need to hold onto the standard file list as well as the new one here. File names recorded in \@filelist must be turned to strings before being added to \g\_\_file\_record\_seq.

```

11520 \cs_if_exist:NT \@filelist
11521   {
11522     \AtBeginDocument
11523     {
11524       \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
11525         { \tl_to_str:N \@filelist }
11526       \seq_gconcat:NNN
11527         \g__file_record_seq
11528         \g__file_record_seq
11529         \l__file_tmp_seq
11530     }
11531   }

```

## 49.5 GetIdInfo

### \GetIdInfo

\\_\_file\_id\_info\_auxi:w \\_\_file\_id\_info\_auxii:w \\_\_file\_id\_info\_auxiii:w As documented in expl3.dtx this function extracts file name etc from an SVN Id line. This used to be how we got version number and so on in all modules, so it had to be defined in l3bootstrap. Now it's more convenient to define it after we have set up quite a lot of tools, and l3file seems the least unreasonable place for it.

The idea here is to extract out the information needed from a standard SVN Id line, but to avoid a line that would get changed when the file is checked in. Hence the fact that none of the lines here include both a dollar sign and the Id keyword!

```

11532 \cs_new_protected:Npn \GetIdInfo
11533   {
11534     \tl_clear_new:N \ExplFileVersion
11535     \tl_clear_new:N \ExplFileDate

```

```

11536      \tl_clear_new:N \ExplFileName
11537      \tl_clear_new:N \ExplFileExtension
11538      \tl_clear_new:N \ExplFileVersion
11539      \group_begin:
11540      \char_set_catcode_space:n { 32 }
11541      \exp_after:wN
11542      \group_end:
11543      \__file_id_info_auxi:w
11544  }

```

A first check for a completely empty SVN field. If that is not the case, there is a second case when a file created using `svn cp` but has not been checked in. That leaves a special marker `-1` version, which has no further data. Dealing correctly with that is the reason for the space in the line to use `\__file_id_info_auxi:w`.

```

11545  \cs_new_protected:Npn \__file_id_info_auxi:w #1 #2
11546  {
11547      \tl_set:Nn \ExplFileVersion {#2}
11548      \str_if_eq:nnTF {#1} { Id }
11549      {
11550          \tl_set:Nn \ExplFileDate { 0000/00/00 }
11551          \tl_set:Nn \ExplFileName { [unknown] }
11552          \tl_set:Nn \ExplFileExtension { [unknown-extension] }
11553          \tl_set:Nn \ExplFileVersion {-1}
11554      }
11555      { \__file_id_info_auxii:w #1 ~ \s__file_stop }
11556  }

```

Here, `#1` is `Id`, `#2` is the file name, `#3` is the extension, `#4` is the version, `#5` is the check in date and `#6` is the check in time and user, plus some trailing spaces. If `#4` is the marker `-1` value then `#5` and `#6` are empty.

```

11557  \cs_new_protected:Npn \__file_id_info_auxii:w
11558      #1 ~ #2.#3 ~ #4 ~ #5 ~ #6 \s__file_stop
11559  {
11560      \tl_set:Nn \ExplFileName {#2}
11561      \tl_set:Nn \ExplFileExtension {#3}
11562      \tl_set:Nn \ExplFileVersion {#4}
11563      \str_if_eq:nnTF {#4} {-1}
11564      { \tl_set:Nn \ExplFileDate { 0000/00/00 } }
11565      { \__file_id_info_auxiii:w #5 - 0 - 0 - \s__file_stop }
11566  }

```

Convert an SVN-style date into a L<sup>A</sup>T<sub>E</sub>X-style one.

```

11567  \cs_new_protected:Npn \__file_id_info_auxiii:w #1 - #2 - #3 - #4 \s__file_stop
11568      { \tl_set:Nn \ExplFileDate { #1/#2/#3 } }

```

(End definition for `\GetIdInfo` and others. This function is documented on page 10.)

## 49.6 Checking the version of kernel dependencies

This function is responsible for checking if dependencies of the L<sup>A</sup>T<sub>E</sub>X3 kernel match the version preloaded in the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  kernel. If versions don't match, the function attempts to tell why by searching for a possible stray format file.

The function starts by checking that the kernel date is defined, and if not zero is used to force the error route. The kernel date is then compared with the argument requested

date (usually the packaging date of the dependency). If the kernel date is less than the required date, it's an error and the loading should abort.

```

11569 \cs_new_protected:Npn \__kernel_dependency_check:Nn #1
11570   { \exp_args:NV \__kernel_dependency_check:nn #1 }
11571 \cs_new_protected:Npn \__kernel_dependency_check:nn #1
11572   {
11573     \cs_if_exist:NTF \c_kernel_expl_date_tl
11574     {
11575       \exp_args:NV \__file_kernel_dependency_compare:nnn
11576         \c_kernel_expl_date_tl {#1}
11577     }
11578     { \__file_kernel_dependency_compare:nnn { 0000-00-00 } {#1} }
11579   }
11580 \cs_new_protected:Npn \__file_kernel_dependency_compare:nnn #1 #2 #3
11581   {
11582     \int_compare:nNnT
11583       { \__file_parse_version:w #1 \s_file_stop } <
11584       { \__file_parse_version:w #2 \s_file_stop }
11585       { \__file_mismatched_dependency_error:nn {#2} {#3} }
11586   }
11587 \cs_new:Npn \__file_parse_version:w #1 - #2 - #3 \s_file_stop {#1#2#3}

```

If the versions differ, then we try to give the user some guidance. This function starts by taking the engine name `\c_sys_engine_str` and replacing `tex` by `latex`, then building a command of the form: `kpsewhich --all --engine=<engine> <format>[-dev].fmt` to query the format files available. A shell is opened and each line is read into a sequence.

```

11588 \cs_new_protected:Npn \__file_mismatched_dependency_error:nn #1 #2
11589   {
11590     \exp_args:NNx \ior_shell_open:Nn \g_file_internal_ior
11591     {
11592       kpsewhich ~ --all ~
11593         --engine = \c_sys_engine_exec_str
11594         \c_space_tl \c_sys_engine_format_str
11595         \bool_lazy_and:nnT
11596           { \tl_if_exist_p:N \development@branch@name }
11597           { ! \tl_if_empty_p:N \development@branch@name }
11598           { -dev } .fmt
11599     }
11600     \seq_clear:N \l_file_tmp_seq
11601     \ior_map_inline:Nn \g_file_internal_ior
11602       { \seq_put_right:Nn \l_file_tmp_seq {##1} }
11603     \ior_close:N \g_file_internal_ior
11604     \msg_error:nnnn { kernel } { mismatched-support-file }
11605       {#1} {#2}

```

And finish by ending the current file.

```

11606   \tex_endinput:D
11607   }
11608 %   \begin{macrocode}
11609 %
11610 % Now define the actual error message:
11611 %   \begin{macrocode}
11612 \msg_new:nnnn { kernel } { mismatched-support-file }
11613   {

```

```

11614      Mismatched-LaTeX-support-files-detected. \\
11615      Loading #'2' aborted!

```

\c\_kernel\_expl\_date\_tl may not exist, due to an older format, so only print the dates when the sentinel token list exists:

```

11616      \tl_if_exist:NT \c_kernel_expl_date_tl
11617      {
11618          \\
11619          The-L3-programming-layer-in-the-LaTeX-format \\
11620          is-dated-\c_kernel_expl_date_tl,~but-in-your-TeX-
11621          tree-the-files-require \\ at-least-#1.
11622      }
11623  }
11624  {

```

The sequence containing the format files should have exactly one item: the format file currently being run. If that's the case, the cause of the error is not that, so print a generic help with some possible causes. If more than one format file was found, then print the list to the user, with appropriate indications of what's in the system and what's in the user tree.

```

11625      \int_compare:nNnTF { \seq_count:N \l_file_tmp_seq } > 1
11626      {
11627          The-cause-seems-to-be-an-old-format-file-in-the-user-tree. \\
11628          LaTeX-found-these-files:
11629          \seq_map_tokens:Nn \l_file_tmp_seq { \\---\use:n } \\
11630          Try-deleting-the-file-in-the-user-tree-then-run-LaTeX-again.
11631      }
11632  {
11633      The-most-likely-causes-are:
11634      \\---A-recent-format-generation-failed;
11635      \\---A-stray-format-file-in-the-user-tree-which-needs-
11636          to-be-removed-or-rebuilt;
11637      \\---You-are-running-a-manually-installed-version-of-#2 \\
11638          \\ which-is-incompatible-with-the-version-in-LaTeX. \\
11639      }
11640  \\
11641      LaTeX-will-abort-loading-the-incompatible-support-files-
11642      but-this-may-lead-to \\ later-errors.-Please-ensure-that-
11643      your-LaTeX-format-is-correctly-regenerated.
11644  }

```

(End definition for \\_kernel\_dependency\_version\_check:Nn and others.)

## 49.7 Messages

```

11645  \msg_new:nnnn { kernel } { file-not-found }
11646  {
11647      File-'#1'-not-found.
11648      The-requested-file-could-not-be-found-in-the-current-directory,~
11649      in-the-TeX-search-path-or-in-the-LaTeX-search-path.
11650  }
11651  \msg_new:nnn { kernel } { file-list }
11652  {
11653      >-File-List-<

```

```

11654     #1 \\
11655     .....
11656 }
11657 \msg_new:nnn { kernel } { filename-chars-lost }
11658   { #1-invalid~in~file~name.~Lost:-#2. }
11659   {
11660     There~was~an~invalid~token~in~the~file~name~that~caused~
11661     the~characters~following~it~to~be~lost.
11662 }
11663 \msg_new:nnn { kernel } { filename-missing-endcsname }
11664   { Missing~\iow_char:N\endcsname~inserted~in~filename. }
11665   {
11666     The~file~name~had~more~\iow_char:N\csname~commands~than~
11667     \iow_char:N\endcsname~ones.~LaTeX~will~add~the~missing~
11668     \iow_char:N\endcsname~and~try~to~continue~as~best~as~it~can.
11669 }
11670 \msg_new:nnn { kernel } { unbalanced-quote-in-filename }
11671   { Unbalanced~quotes~in~file~name~'#1'. }
11672   {
11673     File~names~must~contain~balanced~numbers~of~quotes~(").
11674 }
11675 \msg_new:nnn { kernel } { iow-indent }
11676   { Only~#1~allows~#2 }
11677   {
11678     The~command~#2~can~only~be~used~in~messages~
11679     which~will~be~wrapped~using~#1.
11680     \tl_if_empty:nF {#3} { ~ It~was~called~with~argument~'#3'. }
11681 }

```

## 49.8 Functions delayed from earlier modules

<@@@=sys>

### \c\_sys\_platform\_str

Detecting the platform on LuaTeX is easy: for other engines, we use the fact that the two common cases have special null files. It is possible to probe further (see package `platform`), but that requires shell escape and seems unlikely to be useful. This is set up here as it requires file searching.

```

11682 \sys_if_engine_luatex:TF
11683   {
11684     \str_const:Nx \c_sys_platform_str
11685       { \tex_directlua:D { tex.print(os.type) } }
11686   }
11687   {
11688     \file_if_exist:nTF { nul: }
11689       {
11690         \file_if_exist:nF { /dev/null }
11691           { \str_const:Nn \c_sys_platform_str { windows } }
11692       }
11693       {
11694         \file_if_exist:nT { /dev/null }
11695           { \str_const:Nn \c_sys_platform_str { unix } }
11696       }
11697   }
11698 \cs_if_exist:NF \c_sys_platform_str

```

```
11699 { \str_const:Nn \c_sys_platform_str { unknown } }
```

(End definition for `\c_sys_platform_str`. This variable is documented on page 74.)

`\sys_if_platform_unix:p:`

```
11700 \clist_map_inline:nn { unix , windows }
```

```
11701 {
```

```
11702     \__file_const:nn { sys_if_platform_ #1 }
```

```
11703     { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
```

```
11704 }
```

(End definition for `\sys_if_platform_unix:TF` and `\sys_if_platform_windows:TF`. These functions are documented on page 74.)

```
11705 </package>
```

# Chapter 50

## I3luatex implementation

11706 ⟨\*package⟩

### 50.1 Breaking out to Lua

11707 ⟨\*tex⟩

11708 ⟨@=lua⟩

⟨\_\_lua\_escape:n

  ⟨\_\_lua\_now:n⟩ ⟨cs\_new\_eq>NN ⟨\_\_lua\_escape:n⟩ ⟨tex\_luaescapestring:D

  ⟨cs\_new\_eq>NN ⟨\_\_lua\_now:n⟩ ⟨tex\_directlua:D

  ⟨cs\_new\_eq>NN ⟨\_\_lua\_shipout:n⟩ ⟨tex\_latelua:D

(End definition for ⟨\_\_lua\_escape:n⟩, ⟨\_\_lua\_now:n⟩, and ⟨\_\_lua\_shipout:n⟩.)

These functions are set up in I3str for bootstrapping: we want to replace them with a “proper” version at this stage, so clean up.

11712 ⟨cs\_undefine:N⟩ ⟨lua\_escape:e

11713 ⟨cs\_undefine:N⟩ ⟨lua\_now:e

⟨lua\_now:n

  ⟨lua\_now:e⟩ ⟨cs\_new:Npn⟩ ⟨lua\_now:e⟩ #1 { ⟨\_\_lua\_now:n⟩ {#1} }

  ⟨cs\_new:Npn⟩ ⟨lua\_now:n⟩ #1 { ⟨lua\_now:e⟩ { \exp\_not:n {#1} } }

  ⟨lua\_shipout\_e:n

  ⟨lua\_shipout:n

  ⟨lua\_escape:n

  ⟨lua\_escape:e⟩ ⟨cs\_new:Npn⟩ ⟨lua\_escape:e⟩ #1 { ⟨\_\_lua\_escape:n⟩ {#1} }

  ⟨cs\_new:Npn⟩ ⟨lua\_escape:n⟩ #1 { ⟨lua\_escape:e⟩ { \exp\_not:n {#1} } }

(End definition for ⟨lua\_now:n⟩ and others. These functions are documented on page 98.)

⟨lua\_load\_module:n

Wrapper around `require'⟨module⟩'`.

11721 ⟨str\_new:N⟩ ⟨\_\_lua\_err\_msg\_str

11722 ⟨cs\_generate\_variant:Nn⟩ ⟨msg\_error:nnnn⟩ { nnnV }

11723 ⟨cs\_new\_protected:Npn⟩ ⟨lua\_load\_module:n⟩ #1

  {

  ⟨bool\_if:nF⟩ { ⟨\_\_lua\_load\_module\_p:n⟩ { #1 } }

  {

  ⟨msg\_error:nnnV

  { luatex } { module-not-found } { #1 } { \\_\_lua\_err\_msg\_str }

```

11729     }
11730   }

```

(End definition for `\lua_load_module:n`. This function is documented on page 99.)

As with engines other than LuaTeX these have to be macros, we give them the same status in all cases. When LuaTeX is not in use, simply give an error message/

```

11731 \sys_if_engine_luatex:F
11732   {
11733     \clist_map_inline:nn
11734       {
11735         \lua_escape:n , \lua_escape:e ,
11736         \lua_now:n , \lua_now:e
11737       }
11738     {
11739       \cs_set:Npn #1 ##1
11740         {
11741           \msg_expandable_error:nnn
11742             { luatex } { luatex-required } { #1 }
11743         }
11744     }
11745   \clist_map_inline:nn
11746     {
11747       \lua_shipout_e:n , \lua_shipout:n, \lua_load_module:n
11748     }
11749     {
11750       \cs_set_protected:Npn #1 ##1
11751         {
11752           \msg_error:nnn
11753             { luatex } { luatex-required } { #1 }
11754         }
11755     }
11756   }

```

## 50.2 Messages

```

11755 \msg_new:nnnn { luatex } { luatex-required }
11756   { LuATeX-engine-not-in-use!~Ignoring~#1. }
11757   {
11758     The~feature~you~are~using~is~only~available~
11759     with~the~LuATeX~engine.~LaTeX3~ignored~'#1'.
11760   }
11761
11762 \msg_new:nnnn { luatex } { module-not-found }
11763   { Lu-module-'#1'-not-found. }
11764   {
11765     The~file~'#1.lua'~could~not~be~found.~Please~ensure~
11766     that~the~file~was~properly~installed~and~that~the~
11767     filename~database~is~current. \\ \\
11768     The~Lua~loader~provided~this~additional~information: \\
11769     #2
11770   }
11771
11772 \prop_gput:Nnn \g_msg_module_name_prop { luatex } { LaTeX }
11773 \prop_gput:Nnn \g_msg_module_type_prop { luatex } { }
11774 </tex>

```

### 50.3 Lua functions for internal use

```
11775  <*lua>
```

Most of the emulation of pdfTEX here is based heavily on Heiko Oberdiek's pdftex-cmds package.

**ltx.utils** Create a table for the kernel's own use.

```
11776 ltx = ltx or {utils={}}
11777 ltx.utils = ltx.utils or {}
11778 local ltxutils = ltx.utils
```

(End definition for `ltx.utils`. This function is documented on page 99.)

Local copies of global tables.

```
11779 local io      = io
11780 local kpse    = kpse
11781 local lfs     = lfs
11782 local math    = math
11783 local md5    = md5
11784 local os      = os
11785 local string  = string
11786 local tex     = tex
11787 local texio   = texio
11788 local tonumber = tonumber
```

Local copies of standard functions.

```
11789 local abs      = math.abs
11790 local byte     = string.byte
11791 local floor    = math.floor
11792 local format   = string.format
11793 local gsub     = string.gsub
11794 local lfs_attr  = lfs.attributes
11795 local open     = io.open
11796 local os_date  = os.date
11797 local setcatcode = tex.setcatcode
11798 local sprint   = tex.sprint
11799 local cprint   = tex.cprint
11800 local write    = tex.write
11801 local write_nl = texio.write_nl
11802 local utf8_char = utf8.char
11803 local package_loaded = package.loaded
11804 local package_searchers = package.searchers
11805 local table_concat = table.concat
11806
11807 local scan_int   = token.scan_int or token.scan_integer
11808 local scan_string = token.scan_string
11809 local scan_keyword = token.scan_keyword
11810 local put_next   = token.put_next
11811 local token_create = token.create
11812 local token_new   = token.new
11813 local set_macro  = token.set_macro
```

Since `token.create` only returns useful values after the tokens has been added to TeX's hash table, we define a variant which defines it first if necessary.

```
11814 local token_create_safe
11815 do
```

```

11816 local is_defined = token.is_defined
11817 local set_char   = token.set_char
11818 local runtoks    = tex.runtoks
11819 local let_token  = token_create'let'
11820
11821 function token_create_safe(s)
11822   local orig_token = token_create(s)
11823   if is_defined(s, true) then
11824     return orig_token
11825   end
11826   set_char(s, 0)
11827   local new_token = token_create(s)
11828   runtoks(function()
11829     put_next(let_token, new_token, orig_token)
11830   end)
11831   return new_token
11832 end
11833 end
11834
11835 local true_tok      = token_create_safe'prg_return_true:'
11836 local false Tok      = token_create_safe'prg_return_false:'

```

In ConTeXt lmtx token.command\_id does not exist, but it can easily be emulated with ConTeXt's tokens.commands.

```

11837 local command_id   = token.command_id
11838 if not command_id and tokens and tokens.commands then
11839   local id_map = tokens.commands
11840   function command_id(name)
11841     return id_map[name]
11842   end
11843 end

```

Deal with ConTeXt: doesn't use kpse library.

```

11844 local kpse_find = (resolvers and resolvers.findfile) or kpse.find_file

```

`escapehex` An internal auxiliary to convert a string to the matching hex escape. This works on a byte basis: extension to handled UTF-8 input is covered in pdftexcmds but is not currently required here.

```

11845 local function escapehex(str)
11846   return (gsub(str, ".", 
11847     function (ch) return format("%02X", byte(ch)) end))
11848 end

```

(End definition for `escapehex`.)

`ltx.utils.filiedump` Similar comments here to the next function: read the file in binary mode to avoid any line-end weirdness.

```

11849 local function filiedump(name,offset,length)
11850   local file = kpse_find(name,"tex",true)
11851   if not file then return end
11852   local f = open(file,"rb")
11853   if not f then return end
11854   if offset and offset > 0 then
11855     f:seek("set", offset)
11856   end

```

```

11857     local data = f:read(length or 'a')
11858     f:close()
11859     return escapehex(data)
11860   end
11861 ltxutils.filiedump = filiedump

```

(End definition for `ltx.utils.filiedump`. This function is documented on page [99](#).)

- md5.HEX** Hash a string and return the hash in uppercase hexadecimal format. In some engines, this is build-in. For traditional LuaTeX, the conversion to hexadecimal has to be done by us.

```

11862 local md5_HEX = md5.HEX
11863 if not md5_HEX then
11864   local md5_sum = md5.sum
11865   function md5_HEX(data)
11866     return escapehex(md5_sum(data))
11867   end
11868   md5.HEX = md5_HEX
11869 end

```

(End definition for `md5.HEX`. This function is documented on page [??](#).)

- ltx.utils.filemd5sum** Read an entire file and hash it: the hash function itself is a built-in. As Lua is byte-based there is no work needed here in terms of UTF-8 (see `pdftexcmds` and how it handles strings that have passed through LuaTeX). The file is read in binary mode so that no line ending normalisation occurs.

```

11870 local function filemd5sum(name)
11871   local file = kpse_find(name, "tex", true) if not file then return end
11872   local f = open(file, "rb") if not f then return end
11873
11874   local data = f:read("*a")
11875   f:close()
11876   return md5_HEX(data)
11877 end
11878 ltxutils.filemd5sum = filemd5sum

```

(End definition for `ltx.utils.filemd5sum`. This function is documented on page [99](#).)

- ltx.utils.filemoddate** There are two cases: If the C standard library is C99 compliant, we can use `%z` to get the timezone in almost the right format. We only have to add primes and replace a zero or missing offset with Z.

Of course this would be boring, so Windows does things differently. There we have to manually calculate the offset. See procedure `makepdftime` in `utils.c` of pdfTeX.

```

11879 local filemoddate
11880 if os_date'%z':match'^[+-]%'d%d%d%d$' then
11881   local pattern = lpeg.Cs(16 *
11882     (lpeg.Cg(lpeg.S'+' * '0000' * lpeg.Cc'Z') +
11883     + 3 * lpeg.Cc'" * 2 * lpeg.Cc'" +
11884     + lpeg.Cc'Z') *
11885     * -1)
11886   function filemoddate(name)
11887     local file = kpse_find(name, "tex", true)
11888     if not file then return end
11889     local date = lfs_attr(file, "modification")

```

```

11890     if not date then return end
11891     return pattern:match(os_date("D:%Y%m%d%H%M%S%z", date))
11892   end
11893 else
11894   local function filemoddate(name)
11895     local file = kpse_find(name, "tex", true)
11896     if not file then return end
11897     local date = lfs_attr(file, "modification")
11898     if not date then return end
11899     local d = os_date("*t", date)
11900     local u = os_date("!*t", date)
11901     local off = 60 * (d.hour - u.hour) + d.min - u.min
11902     if d.year ~= u.year then
11903       if d.year > u.year then
11904         off = off + 1440
11905       else
11906         off = off - 1440
11907       end
11908     elseif d.yday ~= u.yday then
11909       if d.yday > u.yday then
11910         off = off + 1440
11911       else
11912         off = off - 1440
11913       end
11914     end
11915     local timezone
11916     if off == 0 then
11917       timezone = "Z"
11918     else
11919       if off < 0 then
11920         timezone = "-"
11921         off = -off
11922       else
11923         timezone = "+"
11924       end
11925       timezone = format("%s%02d%02d", timezone, hours // 60, hours % 60)
11926     end
11927     return format("D:%04d%02d%02d%02d%02d%s",
11928                   d.year, d.month, d.day, d.hour, d.min, d.sec, timezone)
11929   end
11930 end
11931 ltxutils.filemoddate = filemoddate

```

(End definition for `ltx.utils.filemoddate`. This function is documented on page 99.)

`ltx.utils.filesize` A simple disk lookup.

```

11932 local function filesize(name)
11933   local file = kpse_find(name, "tex", true)
11934   if file then
11935     local size = lfs_attr(file, "size")
11936     if size then
11937       return size
11938     end
11939   end

```

```

11940   end
11941 ltxutils.filesize = filesize

```

(End definition for `ltx.utils.filesize`. This function is documented on page 100.)

- `luadef` An internal function for defining control sequences from Lua which behave like primitives. This acts as a wrapper around `token.set_lua` which accepts a function instead of an index into the functions table.

```

11942 local luacmd do
11943   local set_lua = token.set_lua
11944   local undefined_cs = command_id'undefined_cs'
11945
11946   if not context and not luatexbase then require'ltxbase' end
11947   if luatexbase then
11948     local new_luafunction = luatexbase.new_luafunction
11949     local functions = lua.get_functions_table()
11950     function luacmd(name, func, ...)
11951       local id
11952       local tok = token_create(name)
11953       if tok.command == undefined_cs then
11954         id = new_luafunction(name)
11955         set_lua(name, id, ...)
11956       else
11957         id = tok.index or tok.mode
11958       end
11959       functions[id] = func
11960     end
11961   elseif context then
11962     local register = context.functions.register
11963     local functions = context.functions.known
11964     function luacmd(name, func, ...)
11965       local tok = token_create(name)
11966       if tok.command == undefined_cs then
11967         token.set_lua(name, register(func), ...)
11968       else
11969         functions[tok.index or tok.mode] = func
11970       end
11971     end
11972   end
11973 end

```

(End definition for `luadef`.)

- `try_require` Loads a Lua module. This function loads the module similarly to the standard Lua global function `require`, with a few differences. On success, `try_require` returns `true, module`. If the module cannot be found, it returns `false, err_msg`. If the module is found, but something goes wrong when loading it, the function throws an error.

```

11974 local function try_require(name)
11975   if package_loaded[name] then
11976     return true, package_loaded[name]
11977   end
11978
11979   local failure_details = {}
11980   for _, searcher in ipairs(package_searchers) do

```

```

11981     local loader, data = searcher(name)
11982     if type(loader) == 'function' then
11983         package_loaded[name] = loader(name, data) or true
11984         return true, package_loaded[name]
11985     elseif type(loader) == 'string' then
11986         failure_details[#failure_details + 1] = loader
11987     end
11988 end
11989
11990     return false, table_concat(failure_details, '\n')
11991 end

```

(End definition for `try_require`.)

`\_lua_load_module_p:n` Check to see if we can load a module using `require`. If we can load the module, then we load it immediately. Otherwise, we save the error message in `\_lua_err_msg_str`.

```

11992 local char_given = command_id'char_given'
11993 local c_true_bool = token_create(1, char_given)
11994 local c_false_bool = token_create(0, char_given)
11995 local c_str_cctab = token_create('c_str_cctab').mode
11996
11997 luacmd('\_lua_load_module_p:n', function()
11998     local success, result = try_require(scan_string())
11999     if success then
12000         set_macro(c_str_cctab, '\_lua_err_msg_str', '')
12001         put_next(c_true_bool)
12002     else
12003         set_macro(c_str_cctab, '\_lua_err_msg_str', result)
12004         put_next(c_false_bool)
12005     end
12006 end)

```

(End definition for `\_lua_load_module_p:n`.)

## 50.4 Preserving iniTeX Lua data for runs

`\@=lua`

The Lua state is not dumped when a forat is written, therefore any Lua variables filled doing format building need to be restored in order to be accessible during normal runs.

We provide some kernel-internal helpers for this. They will only be available if `luatexbase` is available. This is not a big restriction though, because ConTeXt (which does not use `luatexbase`) does not load `expl3` in the format.

```

12008 local register_luadata, get_luadata
12009
12010 if luatexbase then
12011     local register = token_create'@expl@luadata@bytecode'.index
12012     if status.ini_version then

```

`register_luadata` `register_luadata` is only available during format generation. It accept a string which uniquely identifies the data object and has to be provided to retrieve it later. Additionally it accepts a function which is called in the `pre_dump` callback and which has to return a string that evaluates to a valid Lua object to be preserved.

```

12013     local luadata, luadata_order = {}, {}
12014
12015     function register_luadata(name, func)
12016         if luadata[name] then
12017             error(format("LaTeX error: data name %q already in use", name))
12018         end
12019         luadata[name] = func
12020         luadata_order[#luadata_order + 1] = func and name
12021     end

```

(End definition for `register_luadata`. This function is documented on page ??.)

The actual work is done in `pre_dump`. The `luadata_order` is used to ensure that the order is consistent over multiple runs.

```

12022     luatexbase.add_to_callback("pre_dump", function()
12023         if next(luadata) then
12024             local str = "return {"
12025             for i=1, #luadata_order do
12026                 local name = luadata_order[i]
12027                 str = format('%s[%q]=%s,', str, name, luadata[name]())
12028             end
12029             lua.bytecode[register] = assert(load(str .. "})")
12030         end
12031     end, "ltx.luadata")
12032     else

```

`get_luadata` `get_luadata` is only available if data should be restored. It accept the identifier which was used when the data object was registered and returns the associated object. Every object can only be retrieved once.

```

12033     local luadata = lua.bytecode[register]
12034     if luadata then
12035         lua.bytecode[register] = nil
12036         luadata = luadata()
12037     end
12038     function get_luadata(name)
12039         if not luadata then return end
12040         local data = luadata[name]
12041         luadata[name] = nil
12042         return data
12043     end
12044 end
12045 end

```

(End definition for `get_luadata`. This function is documented on page ??.)

```

12046 </lua>
12047 </package>

```

# Chapter 51

## I3legacy Implementation

```
12048  <*package>
12049  @@@=legacy>
```

\legacy\_if\_p:n  
\legacy\_if:nTF

A friendly wrapper. We need to use the \if:w approach here, rather than testing against \iftrue/\iffalse as the latter approach fails for primitive conditionals such as \ifmmode. The \reverse\_if:N here means that we get a slightly more useful error if the name is undefined.

```
12050  \prg_new_conditional:Npnn \legacy_if:n #1 { p , T , F , TF }
12051  {
12052      \exp_after:wN \reverse_if:N
12053      \cs:w if#1 \cs_end:
12054      \prg_return_false:
12055      \else:
12056      \prg_return_true:
12057      \fi:
12058  }
```

(End definition for \legacy\_if:nTF. This function is documented on page 101.)

\legacy\_if\_set\_true:n

\legacy\_if\_set\_false:n

\legacy\_if\_gset\_true:n

\legacy\_if\_gset\_false:n

A friendly wrapper.

```
12059  \cs_new_protected:Npn \legacy_if_set_true:n #1
12060  { \cs_set_eq:cN { if#1 } \if_true: }
12061  \cs_new_protected:Npn \legacy_if_set_false:n #1
12062  { \cs_set_eq:cN { if#1 } \if_false: }
12063  \cs_new_protected:Npn \legacy_if_gset_true:n #1
12064  { \cs_gset_eq:cN { if#1 } \if_true: }
12065  \cs_new_protected:Npn \legacy_if_gset_false:n #1
12066  { \cs_gset_eq:cN { if#1 } \if_false: }
```

(End definition for \legacy\_if\_set\_true:n and others. These functions are documented on page 101.)

\legacy\_if\_set:nn

\legacy\_if\_gset:nn

A more elaborate wrapper.

```
12067  \cs_new_protected:Npn \legacy_if_set:nn #1#2
12068  {
12069      \bool_if:nTF {#2} \legacy_if_set_true:n \legacy_if_set_false:n
12070      {#1}
12071  }
12072  \cs_new_protected:Npn \legacy_if_gset:nn #1#2
```

```
12073  {
12074    \bool_if:nTF {#2} \legacy_if_gset_true:n \legacy_if_gset_false:n
12075    {#1}
12076 }
```

(End definition for `\legacy_if_set:nn` and `\legacy_if_gset:nn`. These functions are documented on page [101](#).)

```
12077 </package>
```

# Chapter 52

## I3tl implementation

```
12078  <*package>
12079  <@@=tl>
```

A token list variable is a T<sub>E</sub>X macro that holds tokens. By using the  $\varepsilon$ -T<sub>E</sub>X primitive `\unexpanded` inside a T<sub>E</sub>X `\edef` it is possible to store any tokens, including `#`, in this way.

### 52.1 Functions

`\__kernel_tl_set:Nx` These two are supplied to get better performance for macros which would otherwise use `\tl_set:Nx` or `\tl_gset:Nx` internally.

```
12080  \cs_new_eq:NN \__kernel_tl_set:Nx  \cs_set_nopar:Npx
12081  \cs_new_eq:NN \__kernel_tl_gset:Nx \cs_gset_nopar:Npx
```

(End definition for `\__kernel_tl_set:Nx` and `\__kernel_tl_gset:Nx`.)

**`\tl_new:N`** Creating new token list variables is a case of checking for an existing definition and doing the definition.

```
12082  \cs_new_protected:Npn \tl_new:N #1
12083  {
12084      \__kernel_chk_if_free_cs:N #1
12085      \cs_gset_eq:NN #1 \c_empty_tl
12086  }
12087  \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for `\tl_new:N`. This function is documented on page 103.)

**`\tl_const:Nn`** Constants are also easy to generate. They use `\cs_gset_nopar:Npx` instead of `\__kernel_tl_gset:Nx` so that the correct scope checking is applied if `I3debug` is used.

```
12088  \cs_new_protected:Npn \tl_const:Nn #1#2
12089  {
12090      \__kernel_chk_if_free_cs:N #1
12091      \cs_gset_nopar:Npx #1 f \__kernel_exp_not:w {#2} }
12092  }
12093  \cs_new_protected:Npn \tl_const:Nx #1#2
12094  {
12095      \__kernel_chk_if_free_cs:N #1
12096      \cs_gset_nopar:Npx #1 {#2}
```

```

12097     }
12098 \cs_generate_variant:Nn \tl_const:Nn { c }
12099 \cs_generate_variant:Nn \tl_const:Nx { c }

```

(End definition for `\tl_const:Nn`. This function is documented on page 104.)

**\tl\_clear:N** Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```

12100 \cs_new_protected:Npn \tl_clear:N #1
12101   { \tex_let:D #1 = ~ \c_empty_tl }
12102 \cs_new_protected:Npn \tl_gclear:N #1
12103   { \tex_global:D \tex_let:D #1 ~ \c_empty_tl }
12104 \cs_generate_variant:Nn \tl_clear:N { c }
12105 \cs_generate_variant:Nn \tl_gclear:N { c }

```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 104.)

**\tl\_clear\_new:N** Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```

12106 \cs_new_protected:Npn \tl_clear_new:N #1
12107   { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
12108 \cs_new_protected:Npn \tl_gclear_new:N #1
12109   { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
12110 \cs_generate_variant:Nn \tl_clear_new:N { c }
12111 \cs_generate_variant:Nn \tl_gclear_new:N { c }

```

(End definition for `\tl_clear_new:N` and `\tl_gclear_new:N`. These functions are documented on page 104.)

**\tl\_set\_eq:NN** For setting token list variables equal to each other. To allow for patching, the arguments have to be explicit. In addition this ensures that a braced second argument will not cause problems.

```

12112 \cs_new_protected:Npn \tl_set_eq:NN #1#2
12113   { \tex_let:D #1 = ~ #2 }
12114 \cs_new_protected:Npn \tl_gset_eq:NN #1#2
12115   { \tex_global:D \tex_let:D #1 = ~ #2 }
12116 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
12117 \cs_generate_variant:Nn \tl_gset_eq:NN { cN, Nc, cc }

```

(End definition for `\tl_set_eq:NN` and `\tl_gset_eq:NN`. These functions are documented on page 104.)

**\tl\_concat:NNN** Concatenating token lists is easy. When checking is turned on, all three arguments must be checked: a token list #2 or #3 equal to `\scan_stop`: would lead to problems later on.

```

12118 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
12119   {
12120     \__kernel_tl_set:Nx #1
12121     {
12122       \__kernel_exp_not:w \exp_after:wN {#2}
12123       \__kernel_exp_not:w \exp_after:wN {#3}
12124     }
12125   }
12126 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
12127   {
12128     \__kernel_tl_gset:Nx #1
12129   }

```

```

12130      \__kernel_exp_not:w \exp_after:wN {#2}
12131      \__kernel_exp_not:w \exp_after:wN {#3}
12132  }
12133 }
12134 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
12135 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

(End definition for \tl_concat:NNN and \tl_gconcat:NNN. These functions are documented on page 104.)
```

**\tl\_if\_exist\_p:N** Copies of the `cs` functions defined in `l3basics`.

```

12136 \prg_new_eq_conditional:NNN \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
12137 \prg_new_eq_conditional:NNN \tl_if_exist:c \cs_if_exist:c { TF , T , F , p }
\tl_if_exist:cTF
```

(End definition for `\tl_if_exist:NTF`. This function is documented on page 104.)

## 52.2 Constant token lists

**\c\_empty\_tl** Never full. We need to define that constant before using `\tl_new:N`.

```

12138 \tl_const:Nn \c_empty_tl { }
```

(End definition for `\c_empty_tl`. This variable is documented on page 119.)

**\c\_novalue\_tl** A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```

12139 \group_begin:
12140   \tex_catcode:D ` - = 11 ~
12141   \tl_const:Nx \c_novalue_tl { - NoValue \token_to_str:N - }
12142 \group_end:
```

(End definition for `\c_novalue_tl`. This variable is documented on page 119.)

**\c\_space\_tl** A space as a token list (as opposed to as a character).

```

12143 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for `\c_space_tl`. This variable is documented on page 119.)

## 52.3 Adding to token list variables

**\tl\_set:Nn** By using `\exp_not:n` token list variables can contain # tokens, which makes the token list registers provided by TeX more or less redundant. The `\tl_set:No` version is done by hand as it is used quite a lot.

```

12144 \cs_new_protected:Npn \tl_set:Nn #1#2
12145   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w {#2} } }
12146 \cs_new_protected:Npn \tl_set:No #1#2
12147   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
12148 \cs_new_protected:Npn \tl_set:Nx #1#2
12149   { \__kernel_tl_set:Nx #1 {#2} }
12150 \cs_new_protected:Npn \tl_gset:Nn #1#2
12151   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w {#2} } }
12152 \cs_new_protected:Npn \tl_gset:No #1#2
12153   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#2} } }
\tl_gset:Nn
\tl_gset:NV
\tl_gset:Nv
\tl_gset:No
\tl_gset:Nf
\tl_gset:Nx
\tl_gset:cn
\tl_gset:cV
\tl_gset:cV
\tl_gset:co
\tl_gset:cf
```

```

12155     { \__kernel_tl_gset:Nx #1 {#2} }
12156     \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
12157     \cs_generate_variant:Nn \tl_set:Nn { c, cV , cv , cf }
12158     \cs_generate_variant:Nn \tl_set:Nx { c }
12159     \cs_generate_variant:Nn \tl_set:No { c }
12160     \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
12161     \cs_generate_variant:Nn \tl_gset:Nn { c, cV , cv , cf }
12162     \cs_generate_variant:Nn \tl_gset:Nx { c }
12163     \cs_generate_variant:Nn \tl_gset:No { c }

```

(End definition for `\tl_set:Nn` and `\tl_gset:Nn`. These functions are documented on page 104.)

Adding to the left is done directly to gain a little performance.

```

\tl_put_left:Nn
\tl_put_left:NV
\tl_put_left:No
\tl_put_left:Nx
\tl_put_left:cN
\tl_put_left:cV
\tl_put_left:co
\tl_put_left:cx
\tl_gput_left:Nn
\tl_gput_left:NV
\tl_gput_left:No
\tl_gput_left:Nx
\tl_gput_left:cN
\tl_gput_left:cV
\tl_gput_left:co
\tl_gput_left:cx
12164 \cs_new_protected:Npn \tl_put_left:Nn #1#2
12165 {
12166     \__kernel_tl_set:Nx #1
12167     { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12168 }
12169 \cs_new_protected:Npn \tl_put_left:NV #1#2
12170 {
12171     \__kernel_tl_set:Nx #1
12172     { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
12173 }
12174 \cs_new_protected:Npn \tl_put_left:No #1#2
12175 {
12176     \__kernel_tl_set:Nx #1
12177     {
12178         \__kernel_exp_not:w \exp_after:wN {#2}
12179         \__kernel_exp_not:w \exp_after:wN {#1}
12180     }
12181 }
12182 \cs_new_protected:Npn \tl_put_left:Nx #1#2
12183 { \__kernel_tl_set:Nx #1 { #2 \__kernel_exp_not:w \exp_after:wN {#1} } }
12184 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
12185 {
12186     \__kernel_tl_gset:Nx #1
12187     { \__kernel_exp_not:w {#2} \__kernel_exp_not:w \exp_after:wN {#1} }
12188 }
12189 \cs_new_protected:Npn \tl_gput_left:NV #1#2
12190 {
12191     \__kernel_tl_gset:Nx #1
12192     { \exp_not:V #2 \__kernel_exp_not:w \exp_after:wN {#1} }
12193 }
12194 \cs_new_protected:Npn \tl_gput_left:No #1#2
12195 {
12196     \__kernel_tl_gset:Nx #1
12197     {
12198         \__kernel_exp_not:w \exp_after:wN {#2}
12199         \__kernel_exp_not:w \exp_after:wN {#1}
12200     }
12201 }
12202 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
12203 { \__kernel_tl_gset:Nx #1 { #2 \__kernel_exp_not:w \exp_after:wN {#1} } }
12204 \cs_generate_variant:Nn \tl_put_left:Nn { c }

```

```

12205 \cs_generate_variant:Nn \tl_put_left:NV { c }
12206 \cs_generate_variant:Nn \tl_put_left:No { c }
12207 \cs_generate_variant:Nn \tl_put_left:Nx { c }
12208 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
12209 \cs_generate_variant:Nn \tl_gput_left:NV { c }
12210 \cs_generate_variant:Nn \tl_gput_left:No { c }
12211 \cs_generate_variant:Nn \tl_gput_left:Nx { c }

```

(End definition for `\tl_put_left:Nn` and `\tl_gput_left:Nn`. These functions are documented on page 104.)

**\tl\_put\_right:Nn** The same on the right.

```

\tl_put_right:NV
\tl_put_right:No
\tl_put_right:Nx
\tl_put_right:cn
\tl_put_right:cV
\tl_put_right:co
\tl_put_right:cx
\tl_gput_right:Nn
\tl_gput_right:NV
\tl_gput_right:No
\tl_gput_right:Nx
\tl_gput_right:cn
\tl_gput_right:cV
\tl_gput_right:co
\tl_gput_right:cx
12212 \cs_new_protected:Npn \tl_put_right:Nn #1#2
12213   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
12214 \cs_new_protected:Npn \tl_put_right:NV #1#2
12215   {
12216     \__kernel_tl_set:Nx #1
12217     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
12218   }
12219 \cs_new_protected:Npn \tl_put_right:No #1#2
12220   {
12221     \__kernel_tl_set:Nx #1
12222     {
12223       \__kernel_exp_not:w \exp_after:wN {#1}
12224       \__kernel_exp_not:w \exp_after:wN {#2}
12225     }
12226   }
12227 \cs_new_protected:Npn \tl_put_right:Nx #1#2
12228   { \__kernel_tl_set:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#1} #2 } }
12229 \cs_new_protected:Npn \tl_gput_right:Nn #1#2
12230   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN { #1 #2 } } }
12231 \cs_new_protected:Npn \tl_gput_right:NV #1#2
12232   {
12233     \__kernel_tl_gset:Nx #1
12234     { \__kernel_exp_not:w \exp_after:wN {#1} \exp_not:V #2 }
12235   }
12236 \cs_new_protected:Npn \tl_gput_right:No #1#2
12237   {
12238     \__kernel_tl_gset:Nx #1
12239     {
12240       \__kernel_exp_not:w \exp_after:wN {#1}
12241       \__kernel_exp_not:w \exp_after:wN {#2}
12242     }
12243   }
12244 \cs_new_protected:Npn \tl_gput_right:Nx #1#2
12245   { \__kernel_tl_gset:Nx #1 { \__kernel_exp_not:w \exp_after:wN {#1} #2 } }
12246 \cs_generate_variant:Nn \tl_put_right:Nn { c }
12247 \cs_generate_variant:Nn \tl_put_right:NV { c }
12248 \cs_generate_variant:Nn \tl_put_right:No { c }
12249 \cs_generate_variant:Nn \tl_put_right:Nx { c }
12250 \cs_generate_variant:Nn \tl_gput_right:Nn { c }
12251 \cs_generate_variant:Nn \tl_gput_right:NV { c }
12252 \cs_generate_variant:Nn \tl_gput_right:No { c }
12253 \cs_generate_variant:Nn \tl_gput_right:Nx { c }

```

(End definition for `\tl_put_right:Nn` and `\tl_gput_right:Nn`. These functions are documented on page 105.)

## 52.4 Internal quarks and quark-query functions

```
\q_tl_nil Internal quarks.  
\q_tl_mark  
\q_tl_stop  
12254 \quark_new:N \q_tl_nil  
12255 \quark_new:N \q_tl_mark  
12256 \quark_new:N \q_tl_stop
```

(End definition for `\q_tl_nil`, `\q_tl_mark`, and `\q_tl_stop`.)

```
\q_tl_recursion_tail Internal recursion quarks.  
\q_tl_recursion_stop  
12257 \quark_new:N \q_tl_recursion_tail  
12258 \quark_new:N \q_tl_recursion_stop
```

(End definition for `\q_tl_recursion_tail` and `\q_tl_recursion_stop`.)

```
\_tl_if_recursion_tail_break:nN Functions to query recursion quarks.  
\_tl_if_recursion_tail_stop_p:n  
\_tl_if_recursion_tail_stop:nTF  
12259 \__kernel_quark_new_test:N \_tl_if_recursion_tail_break:nN  
12260 \__kernel_quark_new_conditional:Nn \_tl_quark_if_nil:n { TF }
```

(End definition for `\_tl_if_recursion_tail_break:nN` and `\_tl_if_recursion_tail_stop:nTF`.)

## 52.5 Reassigning token list category codes

```
\c_tl_rescan_marker_tl The rescanning code needs a special token list containing the same character (chosen here to be a colon) with two different category codes: it cannot appear in the tokens being rescanned since all colons have the same category code.
```

```
12261 \tl_const:Nx \c_tl_rescan_marker_tl { : \token_to_str:N : }
```

(End definition for `\c_tl_rescan_marker_tl`.)

```
\tl_set_rescan:Nnn In a group, after some initial setup explained below and the user setup #3 (followed by  
\scan_stop: to be safe), there is a call to \_tl_set_rescan:nNN. This shared auxiliary  
defined later distinguishes single-line and multi-line “files”. In the simplest case of multi-  
line files, it calls (with the same arguments) \_tl_set_rescan_multi:nNN, whose code  
is included here to help understand the approach. This function rescans its argument #1,  
closes the group, and performs the assignment.
```

One difficulty when rescanning is that `\scantokens` treats the argument as a file,  
and without the correct settings a T<sub>E</sub>X error occurs:

```
! File ended while scanning definition of ...
```

A related minor issue is a warning due to opening a group before the `\scantokens` and  
closing it inside that temporary file; we avoid that by setting `\tracingnesting`. The  
standard solution to the “File ended” error is to grab the rescanned tokens as a delimited  
argument of an auxiliary, here `\_tl_rescan>NNw`, that performs the assignment, then let  
T<sub>E</sub>X “execute” the end of file marker. As usual in delimited arguments we use `\prg_do_-  
nothing`: to avoid stripping an outer set braces: this is removed by using o-expanding  
assignments. The delimiter cannot appear within the rescanned token list because it  
contains twice the same character, with different catcodes.

For `\tl_rescan:nn` we cannot simply call `\_tl_set_rescan:Nnn \prg_do_nothing: \use:n` because that would leave the end-of-file marker *after* the result of rescanning. If that rescanned result is code that looks further in the input stream for arguments, it would break.

For multi-line files the only subtlety is that `\newlinechar` should be equal to `\endlinechar` because `\newlinechar` characters become new lines and then become `\endlinechar` characters when writing to an abstract file and reading back. This equality is ensured by setting `\newlinechar` equal to `\endlinechar`. Prior to this, `\endlinechar` is set to  $-1$  if it was  $32$  (in particular true after `\ExplSyntaxOn`) to avoid unreasonable line-breaks at every space for instance in error messages triggered by the user setup. Another side effect of reading back from the file is that spaces (catcode  $10$ ) are ignored at the beginning of lines, and spaces and tabs (character code  $32$  and  $9$ ) are ignored at the end of lines.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning. We put the “opening” one after `\group_begin:` so that if one accidentally f-expands `\tl_set_rescan:Nnn` braces remain balanced. This is essential in e-type arguments when `\expanded` is not available.

```

12262 \cs_new_protected:Npn \tl_rescan:nn #1#2
12263   {
12264     \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
12265     \exp_after:wN \_tl_rescan_aux:
12266     \l__tl_internal_a_tl
12267   }
12268 \exp_args:NNo \cs_new_protected:Npn \_tl_rescan_aux:
12269   { \tl_clear:N \l__tl_internal_a_tl }
12270 \cs_new_protected:Npn \tl_set_rescan:Nnn
12271   { \_tl_set_rescan>NNnn \tl_set:No }
12272 \cs_new_protected:Npn \tl_gset_rescan:Nnn
12273   { \_tl_set_rescan>NNnn \tl_gset:No }
12274 \cs_new_protected:Npn \_tl_set_rescan>NNnn #1#2#3#4
12275   {
12276     \group_begin:
12277       \if_false: { \fi:
12278         \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
12279         \int_compare:nNnT \tex_endlinechar:D = { 32 }
12280           { \int_set:Nn \tex_endlinechar:D { -1 } }
12281         \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
12282           #3 \scan_stop:
12283         \exp_args:No \_tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
12284       \if_false: } \fi:
12285   }
12286 \cs_new_protected:Npn \_tl_set_rescan_multi:nNN #1#2#3
12287   {
12288     \tex_everyeof:D \exp_after:wN { \c__tl_rescan_marker_tl }
12289     \exp_after:wN \_tl_rescan:NNw
12290     \exp_after:wN #2
12291     \exp_after:wN #3
12292     \exp_after:wN \prg_do_nothing:
12293     \tex_scantokens:D {#1}
12294   }
12295 \exp_args:Nno \use:nn
12296   { \cs_new:Npn \_tl_rescan:NNw #1#2#3 } \c__tl_rescan_marker_tl

```

```

12297  {
12298  \group_end:
12299  #1 #2 {#3}
12300 }
12301 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
12302 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
12303 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
12304 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { c , cno }

```

(End definition for `\tl_set_rescan:Nnn` and others. These functions are documented on page 118.)

```

\__tl_set_rescan:nNN
\__tl_set_rescan_single:nnNN
\__tl_set_rescan_single_aux:nnNN
\__tl_set_rescan_single_aux:w

```

The function `\__tl_set_rescan:nNN` calls `\__tl_set_rescan_multi:nNN` or `\__tl_set_rescan_single:nnNN` depending on whether its argument is a single-line fragment of code/data or is made of multiple lines by testing for the presence of a `\newlinechar` character. If `\newlinechar` is out of range, the argument is assumed to be a single line.

For a single line, no `\endlinechar` should be added, so it is set to `-1`, and spaces should not be removed. Trailing spaces and tabs are a difficult matter, as TeX removes these at a very low level. The only way to preserve them is to rescan not the argument but the argument followed by a character with a reasonable category code. Here, 11 (letter) and 12 (other) are accepted, as these are convenient, suitable for delimiting an argument, and it is very unlikely that none of the ASCII characters are in one of these categories. To avoid selecting one particular character to put at the end, whose category code may have been modified, there is a loop through characters from `'` (ASCII 39) to `~` (ASCII 127). The choice of starting point was made because this is the start of a very long range of characters whose standard category is letter or other, thus minimizing the number of steps needed by the loop (most often just a single one). If no valid character is found (very rare), fall-back on `\__tl_set_rescan_multi:nNN`.

Otherwise, once a valid character is found (let us use `'` in this explanation) run some code very similar to `\__tl_set_rescan_multi:nNN` but with `'` added at both ends of the input. Of course, we need to define the auxiliary `\__tl_set_rescan_single>NNww` on the fly to remove the additional `'` that is just before `::` (by which we mean `\c_tl_rescan_marker_t1`). Note that the argument must be delimited by `'` with the current catcode; this is done thanks to `\char_generate:nn`. Yet another issue is that the rescanned token list may contain a comment character, in which case the `'` we expected is not there. We fix this as follows: rather than just `::` we set `\everyeof` to `::{\langle code1\rangle} ::{\langle code2\rangle} \s__tl_stop`. The auxiliary `\__tl_set_rescan_single>NNww` runs the `\o-expanding assignment`, expanding either `\langle code1\rangle` or `\langle code2\rangle` before its the main argument `#3`. In the typical case without comment character, `\langle code1\rangle` is expanded, removing the leading `'`. In the rarer case with comment character, `\langle code2\rangle` is expanded, calling `\__tl_set_rescan_single_aux:w`, which removes the trailing `::{\langle code1\rangle}` and the leading `'`.

```

12305 \cs_new_protected:Npn \__tl_set_rescan:nNN #1
12306  {
12307  \int_compare:nNnTF \tex_newlinechar:D < 0
12308  { \use_ii:nn }
12309  {
12310  \exp_args:Nnf \tl_if_in:nnTF {\#1}
12311  { \char_generate:nn { \tex_newlinechar:D } { 12 } }
12312  }
12313  { \__tl_set_rescan_multi:nNN }
12314  {

```

```

12315     \int_set:Nn \tex_endlinechar:D { -1 }
12316     \__tl_set_rescan_single:nnNN { ' }
12317   }
12318   {#1}
12319 }
12320 \cs_new_protected:Npn \__tl_set_rescan_single:nnNN #1
12321 {
12322   \int_compare:nNnTF
12323   { \char_value_catcode:n {#1} / 2 } = 6
12324   {
12325     \exp_args:Nof \__tl_set_rescan_single_aux:nnnNN
12326     \c__tl_rescan_marker_tl
12327     { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
12328   }
12329   {
12330     \int_compare:nNnTF {#1} < { '\~ }
12331     {
12332       \exp_args:Nf \__tl_set_rescan_single:nnNN
12333       { \int_eval:n { #1 + 1 } }
12334     }
12335     { \__tl_set_rescan_multi:nNN }
12336   }
12337 }
12338 \cs_new_protected:Npn \__tl_set_rescan_single_aux:nnnNN #1#2#3#4#5
12339 {
12340   \tex_everyeof:D
12341   {
12342     #1 \use_none:n
12343     #2 #1 { \exp:w \__tl_set_rescan_single_aux:w }
12344     \s__tl_stop
12345   }
12346   \cs_set:Npn \__tl_rescan>NNw ##1##2##3 #2 #1 ##4 ##5 \s__tl_stop
12347   {
12348     \group_end:
12349     ##1 ##2 { ##4 ##3 }
12350   }
12351   \exp_after:wN \__tl_rescan>NNw
12352   \exp_after:wN #4
12353   \exp_after:wN #5
12354   \tex_scantokens:D { #2 #3 #2 }
12355 }
12356 \exp_args:Nno \use:nn
12357 { \cs_new:Npn \__tl_set_rescan_single_aux:w #1 }
12358 \c__tl_rescan_marker_tl #2
12359 { \use_i:nn \exp_end: #1 }

```

(End definition for `\__tl_set_rescan:nNN` and others.)

## 52.6 Modifying token list variables

`\tl_replace_all:Nnn`  
`\tl_replace_all:cnn`  
`\tl_greplace_all:Nnn`  
`\tl_greplace_all:cnn`  
`\tl_replace_once:Nnn`  
`\tl_replace_once:cnn`  
`\tl_greplace_once:Nnn`  
`\tl_greplace_once:cnn`

All of the `replace` functions call `\__tl_replace:NnNNNnn` with appropriate arguments. The first two arguments are explained later. The next controls whether the replacement function calls itself (`\__tl_replace_next:w`) or stops (`\__tl_replace_wrap:w`) after

the first replacement. Next comes an  $x$ -type assignment function  $\text{\_}\text{tl\_set:Nx}$  or  $\text{\_}\text{tl\_gset:Nx}$  for local or global replacements. Finally, the three arguments  $\langle tl\ var \rangle \{ \langle pattern \rangle \} \{ \langle replacement \rangle \}$  provided by the user. When describing the auxiliary functions below, we denote the contents of the  $\langle tl\ var \rangle$  by  $\langle token\ list \rangle$ .

```

12360 \cs_new_protected:Npn \tl_replace_once:Nnn
12361   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_set:Nx }
12362 \cs_new_protected:Npn \tl_greplace_once:Nnn
12363   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_wrap:w \__kernel_tl_gset:Nx }
12364 \cs_new_protected:Npn \tl_replace_all:Nnn
12365   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_set:Nx }
12366 \cs_new_protected:Npn \tl_greplace_all:Nnn
12367   { \__tl_replace:NnNNNnn \q__tl_mark ? \__tl_replace_next:w \__kernel_tl_gset:Nx }
12368 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
12369 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
12370 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
12371 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }

```

(End definition for  $\text{\_}\text{tl\_replace\_all:Nnn}$  and others. These functions are documented on page 117.)

To implement the actual replacement auxiliary  $\text{\_}\text{tl\_replace\_auxii:nNNNnn}$  we need a  $\langle delimiter \rangle$  with the following properties:

- all occurrences of the  $\langle pattern \rangle$  #6 in “ $\langle token\ list \rangle \langle delimiter \rangle$ ” belong to the  $\langle token\ list \rangle$  and have no overlap with the  $\langle delimiter \rangle$ ,
- the first occurrence of the  $\langle delimiter \rangle$  in “ $\langle token\ list \rangle \langle delimiter \rangle$ ” is the trailing  $\langle delimiter \rangle$ .

We first find the building blocks for the  $\langle delimiter \rangle$ , namely two tokens  $\langle A \rangle$  and  $\langle B \rangle$  such that  $\langle A \rangle$  does not appear in #6 and #6 is not  $\langle B \rangle$  (this condition is trivial if #6 has more than one token). Then we consider the delimiters “ $\langle A \rangle$ ” and “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ”, for  $n \geq 1$ , where  $\langle A \rangle^n$  denotes  $n$  copies of  $\langle A \rangle$ , and we choose as our  $\langle delimiter \rangle$  the first one which is not in the  $\langle token\ list \rangle$ .

Every delimiter in the set obeys the first condition: #6 does not contain  $\langle A \rangle$  hence cannot be overlapping with the  $\langle token\ list \rangle$  and the  $\langle delimiter \rangle$ , and it cannot be within the  $\langle delimiter \rangle$  since it would have to be in one of the two  $\langle B \rangle$  hence be equal to this single token (or empty, but this is an error case filtered separately). Given the particular form of these delimiters, for which no prefix is also a suffix, the second condition is actually a consequence of the weaker condition that the  $\langle delimiter \rangle$  we choose does not appear in the  $\langle token\ list \rangle$ . Additionally, the set of delimiters is such that a  $\langle token\ list \rangle$  of  $n$  tokens can contain at most  $O(n^{1/2})$  of them, hence we find a  $\langle delimiter \rangle$  with at most  $O(n^{1/2})$  tokens in a time at most  $O(n^{3/2})$ . Bear in mind that these upper bounds are reached only in very contrived scenarios: we include the case “ $\langle A \rangle$ ” in the list of delimiters to try, so that the  $\langle delimiter \rangle$  is simply  $\text{\_}\text{q__tl_mark}$  in the most common situation where neither the  $\langle token\ list \rangle$  nor the  $\langle pattern \rangle$  contains  $\text{\_}\text{q__tl_mark}$ .

Let us now ahead, optimizing for this most common case. First, two special cases: an empty  $\langle pattern \rangle$  #6 is an error, and if #1 is absent from both the  $\langle token\ list \rangle$  #5 and the  $\langle pattern \rangle$  #6 then we can use it as the  $\langle delimiter \rangle$  through  $\text{\_}\text{tl_replace\_auxii:nNNNnn} \{ \#1 \}$ . Otherwise, we end up calling  $\text{\_}\text{tl_replace:NnNNNnn}$  repeatedly with the first two arguments  $\text{\_}\text{q__tl_mark} \{ ? \}$ ,  $\text{\_}\text{?} \{ ?? \}$ ,  $\text{\_}\text{??} \{ ??? \}$ , and so on, until #6 does not contain the control sequence #1, which we take as our  $\langle A \rangle$ . The argument #2 only serves to collect ? characters for #1. Note that the order of the tests means that the

first two are done every time, which is wasteful (for instance, we repeatedly test for the emptiness of #6). However, this is rare enough not to matter. Finally, choose  $\langle B \rangle$  to be  $\backslash q\_tl\_nil$  or  $\backslash q\_tl\_stop$  such that it is not equal to #6.

The  $\backslash \_tl\_replace\_auxi:NnnNNNnn$  auxiliary receives  $\{\langle A \rangle\}$  and  $\{\langle A \rangle^n \langle B \rangle\}$  as its arguments, initially with  $n = 1$ . If “ $\langle A \rangle \langle A \rangle^n \langle B \rangle \langle A \rangle^n \langle B \rangle$ ” is in the  $\langle token\ list \rangle$  then increase  $n$  and try again. Once it is not anymore in the  $\langle token\ list \rangle$  we take it as our  $\langle delimiter \rangle$  and pass this to the  $\text{auxii}$  auxiliary.

```

12372 \cs_new_protected:Npn \_tl_replace:NnNNNnn #1#2#3#4#5#6#7
12373 {
12374     \tl_if_empty:nTF {#6}
12375     {
12376         \msg_error:nnx { kernel } { empty-search-pattern }
12377         { \tl_to_str:n {#7} }
12378     }
12379     {
12380         \tl_if_in:onTF { #5 #6 } {#1}
12381         {
12382             \tl_if_in:nnTF {#6} {#1}
12383             { \exp_args:Nc \_tl_replace:NnNNNnn {#2} {#2?} }
12384             {
12385                 \_tl_quark_if_nil:nTF {#6}
12386                 { \_tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_tl_stop } }
12387                 { \_tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_tl_nil } }
12388             }
12389         }
12390         { \_tl_replace_auxii:nNNNnn {#1} }
12391         #3#4#5 {#6} {#7}
12392     }
12393 }
12394 \cs_new_protected:Npn \_tl_replace_auxi:NnnNNNnn #1#2#3
12395 {
12396     \tl_if_in:NnTF #1 { #2 #3 #3 }
12397     { \_tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2} }
12398     { \_tl_replace_auxii:nNNNnn { #2 #3 #3 } }
12399 }
```

The auxiliary  $\backslash \_tl\_replace\_auxii:nNNNnn$  receives the following arguments:

```
{⟨delimiter⟩} {⟨function⟩} {⟨assignment⟩}
{⟨tl var⟩} {⟨pattern⟩} {⟨replacement⟩}
```

All of its work is done between  $\backslash \text{group\_align\_safe\_begin:}$  and  $\backslash \text{group\_align\_safe\_end:}$  to avoid issues in alignments. It does the actual replacement within #3 #4 {...}, an x-expanding  $\langle assignment \rangle$  #3 to the  $\langle tl\ var \rangle$  #4. The auxiliary  $\backslash \_tl\_replace\_next:w$  is called, followed by the  $\langle token\ list \rangle$ , some tokens including the  $\langle delimiter \rangle$  #1, followed by the  $\langle pattern \rangle$  #5. This auxiliary finds an argument delimited by #5 (the presence of a trailing #5 avoids runaway arguments) and calls  $\backslash \_tl\_replace\_wrap:w$  to test whether this #5 is found within the  $\langle token\ list \rangle$  or is the trailing one.

If on the one hand it is found within the  $\langle token\ list \rangle$ , then #1 cannot contain the  $\langle delimiter \rangle$  #1 that we worked so hard to obtain, thus  $\backslash \_tl\_replace\_wrap:w$  gets ##1 as its own argument ##1, and protects it against the x-expanding assignment. It also finds  $\backslash \exp_{not}:n$  as ##2 and does nothing to it, thus letting through  $\backslash \exp_{not}:n$  {⟨replacement⟩} into the assignment. Note that  $\backslash \_tl\_replace\_next:w$  and  $\backslash \_tl\_replace\_wrap:w$  are always called followed by two empty brace groups. These are safe

because no delimiter can match them. They prevent losing braces when grabbing delimited arguments, but require the use of `\exp_not:o` and `\use_none:nn`, rather than simply `\exp_not:n`. Afterwards, `\_tl_replace_next:w` is called to repeat the replacement, or `\_tl_replace_wrap:w` if we only want a single replacement. In this second case, `##1` is the *<remaining tokens>* in the *<token list>* and `##2` is some *<ending code>* which ends the assignment and removes the trailing tokens `#5` using some `\if_false: { \fi: }` trickery because `#5` may contain any delimiter.

If on the other hand the argument `##1` of `\_tl_replace_next:w` is delimited by the trailing *<pattern>* `#5`, then `##1` is “`{ } { } <token list> <delimiter> {<ending code>}`”, hence `\_tl_replace_wrap:w` finds “`{ } { } <token list>`” as `##1` and the *<ending code>* as `##2`. It leaves the *<token list>* into the assignment and unbrates the *<ending code>* which removes what remains (essentially the *<delimiter>* and *<replacement>*).

```

12400 \cs_new_protected:Npn \_tl_replace_auxii:nNNNnn #1#2#3#4#5#6
12401   {
12402     \group_align_safe_begin:
12403     \cs_set:Npn \_tl_replace_wrap:w ##1 #1 ##2
12404       { \_kernel_exp_not:w \exp_after:wN { \use_none:nn ##1 } ##2 }
12405     \cs_set:Npx \_tl_replace_next:w ##1 #5
12406       {
12407         \exp_not:N \_tl_replace_wrap:w ##1
12408         \exp_not:n { #1 }
12409         \exp_not:n { \exp_not:n {#6} }
12410         \exp_not:n { #2 { } { } }
12411       }
12412     #3 #4
12413       {
12414         \exp_after:wN \_tl_replace_next_aux:w
12415         #4
12416         #1
12417         {
12418           \if_false: { \fi: }
12419           \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12420         }
12421         #5
12422       }
12423     \group_align_safe_end:
12424   }
12425 \cs_new:Npn \_tl_replace_next_aux:w { \_tl_replace_next:w { } { } }
12426 \cs_new_eq:NN \_tl_replace_wrap:w ?
12427 \cs_new_eq:NN \_tl_replace_next:w ?
```

(End definition for `\_tl_replace:NnNNNnn` and others.)

`\tl_remove_once:Nn` Removal is just a special case of replacement.

```

\tl_remove_once:cn
12428 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
12429   { \tl_replace_once:Nnn #1 {#2} { } }
12430 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
12431   { \tl_greplace_once:Nnn #1 {#2} { } }
12432 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
12433 \cs_generate_variant:Nn \tl_gremove_once:Nn { c }
```

(End definition for `\tl_remove_once:Nn` and `\tl_gremove_once:Nn`. These functions are documented on page 117.)

```

\tl_remove_all:Nn Removal is just a special case of replacement.
\tl_remove_all:cn 12434 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_gremove_all:Nn 12435 { \tl_replace_all:Nnn #1 {\#2} { } }
\tl_gremove_all:cn 12436 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
12437 { \tl_greplace_all:Nnn #1 {\#2} { } }
12438 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
12439 \cs_generate_variant:Nn \tl_gremove_all:Nn { c }

(End definition for \tl_remove_all:Nn and \tl_gremove_all:Nn. These functions are documented on page 118.)

```

## 52.7 Token list conditionals

\tl\_if\_empty\_p:N These functions check whether the token list in the argument is empty and execute the proper code from their argument(s).

```

\tl_if_empty_p:c 12440 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\tl_if_empty:cTF 12441 {
12442     \if_meaning:w #1 \c_empty_tl
12443         \prg_return_true:
12444     \else:
12445         \prg_return_false:
12446     \fi:
12447 }
12448 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
12449 { c } { p , T , F , TF }

(End definition for \tl_if_empty:NTF. This function is documented on page 105.)

```

\tl\_if\_empty\_p:n The \if:w triggers the expansion of \tl\_to\_str:n which converts the argument to a string: this is empty if and only if the argument is. Then \if:w \scan\_stop: ... \scan\_stop: is true if and only if the string ... is empty. It could be tempting to use \if:w \scan\_stop: #1 \scan\_stop: directly. But this fails on a token list expanding to anything starting with \scan\_stop: leaving everything that follows in the input stream.

```

\tl_if_empty_p:v 12450 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
\tl_if_empty:nTF 12451 {
12452     \if:w \scan_stop: \tl_to_str:n {#1} \scan_stop:
12453         \prg_return_true:
12454     \else:
12455         \prg_return_false:
12456     \fi:
12457 }
12458 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
12459 { V } { p , TF , T , F }

(End definition for \tl_if_empty:nTF. This function is documented on page 105.)

```

\tl\_if\_empty\_p:o The auxiliary function \\_\_tl\_if\_empty\_if:o is for use in various token list conditionals which reduce to testing if a given token list is empty after applying a simple function to it. The test for emptiness is based on \tl\_if\_empty:nTF, but the expansion is hard-coded for efficiency, as this auxiliary function is used in several places. We don't put \prg\_return\_true: and so on in the definition of the auxiliary, because that would prevent an optimization applied to conditionals that end with this code. Also the

\@\_if\_empty\_if:o is expanded once in \tl\_if\_empty:oTF for efficiency as well (and to reduce code doubling).

```

12460 \cs_new:Npn \__tl_if_empty_if:o #1
12461   {
12462     \if:w \scan_stop: \__kernel_tl_to_str:w \exp_after:wN {#1} \scan_stop:
12463   }
12464 \exp_args:Nno \use:n
12465   { \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F } }
12466   {
12467     \__tl_if_empty_if:o {#1}
12468     \prg_return_true:
12469     \else:
12470     \prg_return_false:
12471   \fi:
12472 }
```

(End definition for \tl\_if\_empty:nTF and \\_\_tl\_if\_empty\_if:o. This function is documented on page 105.)

\tl\_if\_blank\_p:n TEX skips spaces when reading a non-delimited arguments. Thus, a  $\langle token\ list\rangle$  is blank if and only if \use\_none:n  $\langle token\ list\rangle$  ? is empty after one expansion. The auxiliary \\_\_tl\_if\_empty\_if:o is a fast emptiness test, converting its argument to a string (after one expansion) and using the test \if:w \scan\_stop: ... \scan\_stop:.

```

12473 \exp_args:Nno \use:n
12474   { \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF } }
12475   {
12476     \__tl_if_empty_if:o { \use_none:n #1 ? }
12477     \prg_return_true:
12478     \else:
12479     \prg_return_false:
12480   \fi:
12481 }
12482 \prg_generate_conditional_variant:Nnn \tl_if_blank:n
12483   { e , V , o } { p , T , F , TF }
```

(End definition for \tl\_if\_blank:nTF and \\_\_tl\_if\_blank\_p:NNw. This function is documented on page 105.)

\tl\_if\_eq\_p:NN Returns \c\_true\_bool if and only if the two token list variables are equal.

```

12484 \prg_new_conditional:Npnn \tl_if_eq:NN #1#2 { p , T , F , TF }
12485   {
12486     \if_meaning:w #1 #2
12487     \prg_return_true:
12488     \else:
12489     \prg_return_false:
12490   \fi:
12491 }
12492 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
12493   { Nc , c , cc } { p , TF , T , F }
```

(End definition for \tl\_if\_eq:NNTF. This function is documented on page 105.)

\l\_\_tl\_internal\_a\_tl Temporary storage.

```

12494 \tl_new:N \l__tl_internal_a_tl
12495 \tl_new:N \l__tl_internal_b_tl
```

(End definition for `\l__tl_internal_a_tl` and `\l__tl_internal_b_tl`.)

**\tl\_if\_eq:NnTF** A simple store and compare routine.

```
12496 \prg_new_protected_conditional:Npnn \tl_if_eq:Nn #1#2 { T , F , TF }
12497   {
12498     \group_begin:
12499       \tl_set:Nn \l__tl_internal_b_tl {#2}
12500       \exp_after:wN
12501     \group_end:
12502     \if_meaning:w #1 \l__tl_internal_b_tl
12503       \prg_return_true:
12504     \else:
12505       \prg_return_false:
12506     \fi:
12507   }
12508 \prg_generate_conditional_variant:Nnm \tl_if_eq:Nn { c } { TF , T , F }
```

(End definition for `\tl_if_eq:NnTF`. This function is documented on page 105.)

**\tl\_if\_eq:nnTF** A simple store and compare routine.

```
12509 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
12510   {
12511     \group_begin:
12512       \tl_set:Nn \l__tl_internal_a_tl {#1}
12513       \tl_set:Nn \l__tl_internal_b_tl {#2}
12514       \exp_after:wN
12515     \group_end:
12516     \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
12517       \prg_return_true:
12518     \else:
12519       \prg_return_false:
12520     \fi:
12521   }
```

(End definition for `\tl_if_eq:nnTF`. This function is documented on page 106.)

**\tl\_if\_in:NnTF** See `\tl_if_in:nnTF` for further comments. Here we simply expand the token list variable  
**\tl\_if\_in:cnTF** and pass it to `\tl_if_in:nnTF`.

```
12522 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
12523 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
12524 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
12525 \prg_generate_conditional_variant:Nnm \tl_if_in:Nn
12526   { c } { T , F , TF }
```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 106.)

**\tl\_if\_in:nnTF** Once more, the test relies on the emptiness test for robustness. The function `\_tl_`  
**\tl\_if\_in:VnTF** `\tmp:w` removes tokens until the first occurrence of #2. If this does not appear in #1, then  
**\tl\_if\_in:onTF** the final #2 is removed, leaving an empty token list. Otherwise some tokens remain, and  
**\tl\_if\_in:notF** the test is `false`. See `\tl_if_empty:nTF` for details on the emptiness test.

Treating correctly cases like `\tl_if_in:nnTF {a state}{states}`, where #1#2 contains #2 before the end, requires special care. To cater for this case, we insert {}{} between the two token lists. This marker may not appear in #2 because of TeX limitations on what can delimit a parameter, hence we are safe. Using two brace groups makes the

test work also for empty arguments. The `\if_false:` constructions are a faster way to do `\group_align_safe_begin:` and `\group_align_safe_end:`. The `\scan_stop:` ensures that f-expanding `\tl_if_in:nnTF` does not lead to unbalanced braces.

```

12527 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
12528   {
12529     \scan_stop:
12530     \if_false: { \fi:
12531       \cs_set:Npn \__tl_tmp:w ##1 #2 { }
12532       \tl_if_empty:oTF { \__tl_tmp:w #1 {} {} #2 }
12533         { \prg_return_false: } { \prg_return_true: }
12534       \if_false: } \fi:
12535     }
12536 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
12537   { V , o , no } { T , F , TF }

```

(End definition for `\tl_if_in:nnTF`. This function is documented on page 106.)

**\tl\_if\_novalue\_p:n** Tests whether `##1` matches `-NoValue-` exactly (with suitable catcodes): this is similar to `\quark_if_nil:nTF`. The first argument of `\__tl_if_novalue:w` is empty if and only if `##1` starts with `-NoValue-`, while the second argument is empty if `##1` is exactly `-NoValue-` or if it has a question mark just following `-NoValue-`. In this second case, however, the material after the first `?`! remains and makes the emptiness test return `false`.

```

12538 \cs_set_protected:Npn \__tl_tmp:w #1
12539   {
12540     \prg_new_conditional:Npnn \tl_if_novalue:n ##1
12541       { p , T , F , TF }
12542       {
12543         \__tl_if_empty_if:o { \__tl_if_novalue:w {} ##1 {} ? ! #1 ? ? ! }
12544           \prg_return_true:
12545         \else:
12546           \prg_return_false:
12547         \fi:
12548       }
12549     \cs_new:Npn \__tl_if_novalue:w ##1 #1 ##2 ? ##3 ? ! { ##1 ##2 }
12550   }
12551 \exp_args:No \__tl_tmp:w { \c_novalue_tl }

```

(End definition for `\tl_if_novalue:nTF` and `\__tl_if_novalue:w`. This function is documented on page 106.)

**\tl\_if\_single\_p:N** Expand the token list and feed it to `\tl_if_single:nTF`.

**\tl\_if\_single:nTF**

```

12552 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
12553 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
12554 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
12555 \cs_new:Npn \tl_if_single:NTF { \exp_args:No \tl_if_single:nTF }

```

(End definition for `\tl_if_single:NTF`. This function is documented on page 106.)

**\tl\_if\_single\_p:n** This test is similar to `\tl_if_empty:nTF`. Expanding `\use_none:nn #1 ??` once yields an empty result if `#1` is blank, a single `?` if `#1` has a single item, and otherwise yields some tokens ending with `??`. Then, `\__kernel_tl_to_str:w` makes sure there are no odd category codes. An earlier version would compare the result to a single `?` using string comparison, but the Lua call is slow in LuaTeX. Instead, `\__tl_if_single:nnw` picks

the second token in front of it. If #1 is empty, this token is the trailing ? and the \if:w test yields **false**. If #1 has a single item, the token is \scan\_stop: and the \if:w test yields **true**. Otherwise, it is one of the characters resulting from \tl\_to\_str:n, and the \if:w test yields **false**. Note that \if:w and \\_kernel\_tl\_to\_str:w are primitives that take care of expansion.

```

12556 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
12557   {
12558     \if:w \scan_stop: \exp_after:wn \_tl_if_single:nnw
12559       \_kernel_tl_to_str:w
12560         \exp_after:wn { \use_none:nn #1 ?? } \scan_stop: ? \s__tl_stop
12561       \prg_return_true:
12562     \else:
12563       \prg_return_false:
12564     \fi:
12565   }
12566 \cs_new:Npn \_tl_if_single:nnw #1#2#3 \s__tl_stop {#2}

```

(End definition for \tl\_if\_single:nTF and \\_tl\_if\_single:nnw. This function is documented on page 106.)

### \tl\_if\_single\_token\_p:n \tl\_if\_single\_token:nTF

There are four cases: empty token list, token list starting with a normal token, with a brace group, or with a space token. If the token list starts with a normal token, remove it and check for emptiness. For the next case, an empty token list is not a single token. Finally, we have a non-empty token list starting with a space or a brace group. Applying f-expansion yields an empty result if and only if the token list is a single space.

```

12567 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
12568   {
12569     \tl_if_head_is_N_type:nTF {#1}
12570       { \_tl_if_empty_if:o { \use_none:n #1 } }
12571       {
12572         \tl_if_empty:nTF {#1}
12573           { \if_false: }
12574             { \_tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
12575       }
12576       \prg_return_true:
12577     \else:
12578       \prg_return_false:
12579     \fi:
12580   }

```

(End definition for \tl\_if\_single\_token:nTF. This function is documented on page 106.)

### \tl\_case:Nn

\tl\_case:cn

\tl\_case:NnTF

\tl\_case:cNTF

\\_tl\_case:nNFT

\\_tl\_case:Nw

\\_tl\_case\_end:nw

```

12581 \cs_new:Npn \tl_case:Nn #1#2
12582   {
12583     \exp:w
12584       \_tl_case:NnTF #1 {#2} { } { }
12585   }
12586 \cs_new:Npn \tl_case:NnT #1#2#3

```

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the **true** or **false** branch code is inserted.

```

12587   {
12588     \exp:w
12589     \__tl_case:NnTF #1 {#2} {#3} { }
12590   }
12591 \cs_new:Npn \tl_case:NnF #1#2#3
12592   {
12593     \exp:w
12594     \__tl_case:NnTF #1 {#2} { } {#3}
12595   }
12596 \cs_new:Npn \tl_case:NnTF #1#2
12597   {
12598     \exp:w
12599     \__tl_case:NnTF #1 {#2}
12600   }
12601 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
12602   { \__tl_case:Nw #1 #2 #1 { } \s__tl_mark {#3} \s__tl_mark {#4} \s__tl_stop }
12603 \cs_new:Npn \__tl_case:Nw #1#2#3
12604   {
12605     \tl_if_eq:NNTF #1 #2
12606     { \__tl_case_end:nw {#3} }
12607     { \__tl_case:Nw #1 }
12608   }
12609 \cs_generate_variant:Nn \tl_case:Nn { c }
12610 \prg_generate_conditional_variant:Nnn \tl_case:Nn
12611   { c } { T , F , TF }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare **\s\_\_tl\_mark** and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first **\s\_\_tl\_mark** and so #4 is the **false** code (the **true** code is mopped up by #3).

```

12612 \cs_new:Npn \__tl_case_end:nw #1#2#3 \s__tl_mark #4#5 \s__tl_stop
12613   { \exp_end: #1 #4 }

```

(End definition for **\tl\_case:NnTF** and others. This function is documented on page 107.)

## 52.8 Mapping over token lists

**\tl\_map\_function:nN**  
**\tl\_map\_function:NN**  
**\tl\_map\_function:cN**  
**\\_\_tl\_map\_function:Nnnnnnnnn**  
**\\_\_tl\_map\_function\_end:w**  
**\\_\_tl\_use\_none\_delimit\_by\_s\_stop:w**

Expandable loop macro for token lists. We use the internal scan mark **\s\_\_tl\_stop** (defined later), which is not allowed to show up in the token list #1 since it is internal to **\tl**. This allows us a very fast test of whether some **\langle item \rangle** is the end-marker **\s\_\_tl\_stop**, namely call **\\_\_tl\_use\_none\_delimit\_by\_s\_stop:w** **\langle item \rangle** **\langle function \rangle** **\s\_\_tl\_stop**, which calls **\langle function \rangle** if the **\langle item \rangle** is the end-marker. To speed up the loop even more, only test one out of eight items, and once we hit one of the eight end-markers, go more slowly through the last few items of the list using **\\_\_tl\_map\_function\_end:w**.

```

12614 \cs_new:Npn \tl_map_function:nN #1#2
12615   {
12616     \__tl_map_function:Nnnnnnnnn #2 #1
12617     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12618     \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop

```

```

12619      \prg_break_point:Nn \tl_map_break: { }
12620  }
12621 \cs_new:Npn \tl_map_function:NN
12622  { \exp_args:No \tl_map_function:nN }
12623 \cs_generate_variant:Nn \tl_map_function:NN { c }
12624 \cs_new:Npn \__tl_map_function:Nnnnnnnnn #1#2#3#4#5#6#7#8#9
12625  {
12626      \__tl_use_none_delimit_by_s_stop:w
12627          #9 \__tl_map_function_end:w \s__tl_stop
12628          #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
12629          \__tl_map_function:Nnnnnnnnn #1
12630  }
12631 \cs_new:Npn \__tl_map_function_end:w \s__tl_stop #1#2
12632  {
12633      \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12634          #1 {#2}
12635          \__tl_map_function_end:w \s__tl_stop
12636  }
12637 \cs_new:Npn \__tl_use_none_delimit_by_s_stop:w #1 \s__tl_stop { }

(End definition for \tl_map_function:nN and others. These functions are documented on page 111.)
```

\tl\_map\_inline:nn      The inline functions are straight forward by now. We use a little trick with the counter \g\_kernel\_prg\_map\_int to make them nestable. We can also make use of \\_\_tl\_map\_function:Nnnnnnnnn from before.

```

12638 \cs_new_protected:Npn \tl_map_inline:nn #1#2
12639  {
12640      \int_gincr:N \g_kernel_prg_map_int
12641      \cs_gset_protected:cpn
12642          { __tl_map_ \int_use:N \g_kernel_prg_map_int :w } ##1 {#2}
12643      \exp_args:Nc \__tl_map_function:Nnnnnnnnn
12644          { __tl_map_ \int_use:N \g_kernel_prg_map_int :w }
12645          #1
12646          \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12647          \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12648      \prg_break_point:Nn \tl_map_break:
12649          { \int_gdecr:N \g_kernel_prg_map_int }
12650  }
12651 \cs_new_protected:Npn \tl_map_inline:Nn
12652  { \exp_args:No \tl_map_inline:nn }
12653 \cs_generate_variant:Nn \tl_map_inline:Nn { c }

(End definition for \tl_map_inline:nn and \tl_map_inline:Nn. These functions are documented on page 111.)
```

\tl\_map\_tokens:nn      Much like the function mapping.

```

12654 \cs_new:Npn \tl_map_tokens:nn #1#2
12655  {
12656      \__tl_map_tokens:nnnnnnnnn {#2} #1
12657          \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12658          \s__tl_stop \s__tl_stop \s__tl_stop \s__tl_stop
12659      \prg_break_point:Nn \tl_map_break: { }
12660  }
12661 \cs_new:Npn \tl_map_tokens:Nn
```

```

12662 { \exp_args:No \tl_map_tokens:nn }
12663 \cs_generate_variant:Nn \tl_map_tokens:Nn { c }
12664 \cs_new:Npn \__tl_map_tokens:nnnnnnnn #1#2#3#4#5#6#7#8#9
12665 {
12666     \__tl_use_none_delimit_by_s_stop:w
12667     #9 \__tl_map_tokens_end:w \s__tl_stop
12668     \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
12669     \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
12670     \__tl_map_tokens:nnnnnnnn {#1}
12671 }
12672 \cs_new:Npn \__tl_map_tokens_end:w \s__tl_stop \use:n #1#2
12673 {
12674     \__tl_use_none_delimit_by_s_stop:w #2 \tl_map_break: \s__tl_stop
12675     #1 {#2}
12676     \__tl_map_tokens_end:w \s__tl_stop
12677 }

```

(End definition for `\tl_map_tokens:nn` and others. These functions are documented on page 112.)

`\tl_map_variable:nNn`  
`\tl_map_variable:NNn`  
`\tl_map_variable:cNn`

```

\__tl_map_variable:Nnn
12678 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
12679 { \tl_map_tokens:nn {#1} { \__tl_map_variable:Nnn #2 {#3} } }
12680 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
12681 { \tl_set:Nn #1 {#3} #2 }
12682 \cs_new_protected:Npn \tl_map_variable:NNn
12683 { \exp_args:No \tl_map_variable:nNn }
12684 \cs_generate_variant:Nn \tl_map_variable:NNn { c }

```

(End definition for `\tl_map_variable:nNn`, `\tl_map_variable:NNn`, and `\__tl_map_variable:Nnn`. These functions are documented on page 112.)

`\tl_map_break:`

`\tl_map_break:n`

```

12685 \cs_new:Npn \tl_map_break:
12686 { \prg_map_break:Nn \tl_map_break: { } }
12687 \cs_new:Npn \tl_map_break:n
12688 { \prg_map_break:Nn \tl_map_break: }

```

(End definition for `\tl_map_break:` and `\tl_map_break:n`. These functions are documented on page 112.)

## 52.9 Using token lists

`\tl_to_str:n` Another name for a primitive: defined in `l3basics`.  
`\tl_to_str:V`

(End definition for `\tl_to_str:n`. This function is documented on page 108.)

`\tl_to_str:N` These functions return the replacement text of a token list as a string.

`\tl_to_str:c`

```

12690 \cs_new:Npn \tl_to_str:N #1 { \__kernel_tl_to_str:w \exp_after:wN {#1} }
12691 \cs_generate_variant:Nn \tl_to_str:N { c }

```

(End definition for `\tl_to_str:N`. This function is documented on page 108.)

**\tl\_use:N** Token lists which are simply not defined give a clear TeX error here. No such luck for ones equal to \scan\_stop: so instead a test is made and if there is an issue an error is forced.

```

12692 \cs_new:Npn \tl_use:N #1
12693 {
12694     \tl_if_exist:NTF #1 {#1}
12695     {
12696         \msg_expandable_error:nnn
12697             { kernel } { bad-variable } {#1}
12698     }
12699 }
12700 \cs_generate_variant:Nn \tl_use:N { c }
```

(End definition for \tl\_use:N. This function is documented on page 109.)

## 52.10 Working with the contents of token lists

**\tl\_count:n** Count number of elements within a token list or token list variable. Brace groups within the list are read as a single element. Spaces are ignored. \\_\_tl\_count:n grabs the element and replaces it by +1. The 0 ensures that it works on an empty list.

```

12701 \cs_new:Npn \tl_count:n #1
12702 {
12703     \int_eval:n
12704     { 0 \tl_map_function:nN {#1} \__tl_count:n }
12705 }
12706 \cs_new:Npn \tl_count:N #1
12707 {
12708     \int_eval:n
12709     { 0 \tl_map_function:NN #1 \__tl_count:n }
12710 }
12711 \cs_new:Npn \__tl_count:n #1 { + 1 }
12712 \cs_generate_variant:Nn \tl_count:n { V , o }
12713 \cs_generate_variant:Nn \tl_count:N { c }
```

(End definition for \tl\_count:n, \tl\_count:N, and \\_\_tl\_count:n. These functions are documented on page 109.)

**\tl\_count\_tokens:n** The token count is computed through an \int\_eval:n construction. Each 1+ is output to the left, into the integer expression, and the sum is ended by the \exp\_end: inserted by \\_\_tl\_act\_end:wn (which is technically implemented as \c\_zero\_int). Somewhat a hack!

```

12714 \cs_new:Npn \tl_count_tokens:n #1
12715 {
12716     \int_eval:n
12717     {
12718         \__tl_act>NNNn
12719         \__tl_act_count_normal:N
12720         \__tl_act_count_group:n
12721         \__tl_act_count_space:
12722         {#1}
12723     }
12724 }
12725 \cs_new:Npn \__tl_act_count_normal:N #1 { 1 + }
```

```

12726 \cs_new:Npn \__tl_act_count_space: { 1 + }
12727 \cs_new:Npn \__tl_act_count_group:n #1 { 2 + \tl_count_tokens:n {#1} + }

```

(End definition for `\tl_count_tokens:n` and others. This function is documented on page 109.)

**\tl\_reverse\_items:n**  
`\__tl_reverse_items:nwNwn` Reversal of a token list is done by taking one item at a time and putting it after `\s__-tl_stop`.

```

12728 \cs_new:Npn \tl_reverse_items:n #1
12729   {
12730     \__tl_reverse_items:nwNwn #1 ?
12731       \s__tl_mark \__tl_reverse_items:nwNwn
12732       \s__tl_mark \__tl_reverse_items:wn
12733       \s__tl_stop { }
12734   }
12735 \cs_new:Npn \__tl_reverse_items:nwNwn #1 #2 \s__tl_mark #3 #4 \s__tl_stop #5
12736   {
12737     #3 #2
12738       \s__tl_mark \__tl_reverse_items:nwNwn
12739       \s__tl_mark \__tl_reverse_items:wn
12740       \s__tl_stop { {#1} #5 }
12741   }
12742 \cs_new:Npn \__tl_reverse_items:wn #1 \s__tl_stop #2
12743   { \__kernel_exp_not:w \exp_after:wN { \use_none:nn #2 } }

```

(End definition for `\tl_reverse_items:n`, `\__tl_reverse_items:nwNwn`, and `\__tl_reverse_items:wn`. This function is documented on page 110.)

**\tl\_trim\_spaces:n**  
`\tl_trim_spaces:o`  
`\tl_trim_spaces_apply:nN`  
`\tl_trim_spaces_apply:oN`  
`\tl_trim_spaces:N`  
`\tl_trim_spaces:c`  
`\tl_gtrim_spaces:N`  
`\tl_gtrim_spaces:c` Trimming spaces from around the input is deferred to an internal function whose first argument is the token list to trim, augmented by an initial `\__tl_trim_mark:`, and whose second argument is a *<continuation>*, which receives as a braced argument `\__tl_trim_mark: <trimmed token list>`. The control sequence `\__tl_trim_mark:` expands to nothing in a single expansion. In the case at hand, we take `\__kernel_exp_not:w \exp_after:wN` as our continuation, so that space trimming behaves correctly within an x-type expansion.

```

12744 \cs_new:Npn \tl_trim_spaces:n #1
12745   {
12746     \__tl_trim_spaces:nn
12747       { \__tl_trim_mark: #1 }
12748       { \__kernel_exp_not:w \exp_after:wN }
12749   }
12750 \cs_generate_variant:Nn \tl_trim_spaces:n { o }
12751 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
12752   { \__tl_trim_spaces:nn { \__tl_trim_mark: #1 } { \exp_args:No #2 } }
12753 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }
12754 \cs_new_protected:Npn \tl_trim_spaces:N #1
12755   { \__kernel_tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
12756 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
12757   { \__kernel_tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
12758 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
12759 \cs_generate_variant:Nn \tl_gtrim_spaces:N { c }

```

Trimming spaces from around the input is done using delimited arguments and quarks, and to get spaces at odd places in the definitions, we nest those in `\__tl_tmp:w`, which then receives a single space as its argument: #1 is `\_`. Removing leading spaces is done with `\__tl_trim_spaces_auxi:w`, which loops until `\__tl_trim_mark:\_` matches the

```

\__tl_trim_spaces:nn
\__tl_trim_spaces_auxi:w
\__tl_trim_spaces_auxii:w
\__tl_trim_spaces_auxiii:w
\__tl_trim_spaces_auxiv:w
  \__tl_trim_mark:

```

end of the token list: then `##1` is the token list and `##3` is `\_tl_trim_spaces_auxii:w`. This hands the relevant tokens to the loop `\_tl_trim_spaces_auxiii:w`, responsible for trimming trailing spaces. The end is reached when `\s__tl_nil` matches the one present in the definition of `\tl_trim_spaces:n`. Then `\_tl_trim_spaces_auxiv:w` puts the token list into a group, with a lingering `\_tl_trim_mark:` at the start (which will expand to nothing in one step of expansion), and feeds this to the *(continuation)*.

```

12760 \cs_set_protected:Npn \_tl_tmp:w #1
12761 {
12762     \cs_new:Npn \_tl_trim_spaces:nn ##1
12763     {
12764         \_tl_trim_spaces_auxi:w
12765         ##1
12766         \s__tl_nil
12767         \_tl_trim_mark: #1 { }
12768         \_tl_trim_mark: \_tl_trim_spaces_auxii:w
12769         \_tl_trim_spaces_auxiii:w
12770         #1 \s__tl_nil
12771         \_tl_trim_spaces_auxiv:w
12772         \s__tl_stop
12773     }
12774     \cs_new:Npn
12775         \_tl_trim_spaces_auxi:w ##1 \_tl_trim_mark: #1 ##2 \_tl_trim_mark: ##3
12776     {
12777         ##3
12778         \_tl_trim_spaces_auxi:w
12779         \_tl_trim_mark:
12780         ##2
12781         \_tl_trim_mark: #1 {##1}
12782     }
12783     \cs_new:Npn \_tl_trim_spaces_auxii:w
12784         \_tl_trim_spaces_auxi:w \_tl_trim_mark: \_tl_trim_mark: ##1
12785     {
12786         \_tl_trim_spaces_auxiii:w
12787         ##1
12788     }
12789     \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \s__tl_nil ##2
12790     {
12791         ##2
12792         ##1 \s__tl_nil
12793         \_tl_trim_spaces_auxiii:w
12794     }
12795     \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \s__tl_nil ##2 \s__tl_stop ##3
12796     { ##3 { ##1 } }
12797     \cs_new:Npn \_tl_trim_mark: {}
12798 }
12799 \_tl_tmp:w { ~ }
```

*(End definition for `\tl_trim_spaces:n` and others. These functions are documented on page 110.)*

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn`

`\tl_gsort:Nn`

`\tl_gsort:cn`

`\tl_sort:nN`

*(End definition for `\tl_sort:Nn`, `\tl_gsort:Nn`, and `\tl_sort:cn`. These functions are documented on page 117.)*

## 52.11 The first token from a token list

```

\tl_head:N Finding the head of a token list expandably always strips braces, which is fine as this is
\tl_head:n consistent with for example mapping over a list. The empty brace groups in \tl_head:n
\tl_head:V ensure that a blank argument gives an empty result. The result is returned within the
\tl_head:v \unexpanded primitive. The approach here is to use \if_false: to allow us to use } as
\tl_head:f the closing delimiter: this is the only safe choice, as any other token would not be able
\_tl_head_auxi:nw to parse it's own code. If the \expanded primitive is available it is used to get a fast and
\_tl_head_auxi:n safe code variant in which we don't have to ensure that the left-most token is an internal
\tl_head:w to not break in an f-type expansion. If \expanded isn't available, using a marker, we
\_tl_tl_head:w can see if what we are grabbing is exactly the marker, or there is anything else to deal
\tl_tail:N with. If there is, there is a loop. If not, tidy up and leave the item in the output stream.
\tl_tail:n More detail in http://tex.stackexchange.com/a/70168.
\tl_tail:V
12800 \cs_if_exist:NTF \tex_expanded:D
\tl_tail:v
12801 {
12802     \cs_new:Npn \tl_head:n #1
12803     {
12804         \__kernel_exp_not:w \tex_expanded:D
12805         { { \if_false: { \fi: \_tl_head_auxi:n #1 { } } } }
12806     }
12807     \cs_new:Npn \_tl_head_auxi:n #1
12808     {
12809         \__kernel_exp_not:w {#1}
12810         \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12811     }
12812 }
12813 {
12814     \cs_new:Npn \tl_head:n #1
12815     {
12816         \__kernel_exp_not:w
12817         \if_false: { \fi: \_tl_head_auxi:nw #1 { } \s__tl_stop }
12818     }
12819     \cs_new:Npn \_tl_head_auxi:nw #1#2 \s__tl_stop
12820     {
12821         \exp_after:wN \_tl_head_auxi:n \exp_after:wN {
12822             \if_false: } \fi: {#1}
12823     }
12824     \exp_args:Nno \use:n
12825     { \cs_new:Npn \_tl_head_auxi:n #1 }
12826     {
12827         \__tl_if_empty_if:o { \use_none:n #1 }
12828         \exp_after:wN \use_ii:nnn
12829         \fi:
12830         \use_ii:nn
12831         {#1}
12832         { \if_false: { \fi: \_tl_head_auxi:nw #1 } }
12833     }
12834 }
12835 \cs_generate_variant:Nn \tl_head:n { V , v , f }
12836 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
12837 \cs_new:Npn \_tl_tl_head:w #1#2 \s__tl_stop {#1}
12838 \cs_new:Npn \tl_head:N { \exp_args:No \tl_head:n }

```

To correctly leave the tail of a token list, it's important *not* to absorb any of the tail part

as an argument. For example, the simple definition

```
\cs_new:Npn \tl_tail:n #1 { \tl_tail:w #1 \q_stop }
\cs_new:Npn \tl_tail:w #1#2 \q_stop
```

would give the wrong result for `\tl_tail:n { a { bc } }` (the braces would be stripped). Thus the only safe way to proceed is to first check that there is an item to grab (*i.e.* that the argument is not blank) and assuming there is to dispose of the first item. As with `\tl_head:n`, the result is protected from further expansion by `\unexpanded`. While we could optimise the test here, this would leave some tokens “banned” in the input, which we do not have with this definition.

```
12839 \exp_args:Nno \use:n { \cs_new:Npn \tl_tail:n #1 }
12840   {
12841     \exp_after:wN \__kernel_exp_not:w
12842     \tl_if_blank:nTF {#1}
12843     { {} }
12844     { \exp_after:wN { \use:none:n #1 } }
12845   }
12846 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
12847 \cs_new:Npn \tl_tail:N { \exp_args:No \tl_tail:n }
```

(End definition for `\tl_head:N` and others. These functions are documented on page 113.)

```
\tl_if_head_eq_meaning:p:nN
\tl_if_head_eq_meaning:nNTF
\tl_if_head_eq_charcode:p:nN
\tl_if_head_eq_charcode:nNTF
\tl_if_head_eq_charcode:p:fN
\tl_if_head_eq_charcode:fNTF
\tl_if_head_eq_catcode:p:nN
\tl_if_head_eq_catcode:nNTF
\tl_if_head_eq_catcode:p:oN
\tl_if_head_eq_catcode:oNTF
  \__tl_head_exp_not:w
  \__tl_if_head_eq_empty_arg:w
```

Accessing the first token of a token list is tricky in three cases: when it has category code 1 (begin-group token), when it is an explicit space, with category code 10 and character code 32, or when the token list is empty (obviously).

Forgetting temporarily about this issue we would use the following test in `\tl_if_head_eq_charcode:nN`. Here, `\tl_head:w` yields the first token of the token list, then passed to `\exp_not:N`.

```
\if_charcode:w
  \exp_after:wN \exp_not:N \tl_head:w #1 \q_nil \q_stop
  \exp_not:N #2
```

The two first special cases are detected by testing if the token list starts with an N-type token (the extra ? sends empty token lists to the `true` branch of this test). In those cases, the first token is a character, and since we only care about its character code, we can use `\str_head:n` to access it (this works even if it is a space character). An empty argument results in `\tl_head:w` leaving two token: ^ and `\__tl_if_head_eq_empty_arg:w` which will result in the `\if_charcode:w` test being false and remove `\exp_not:N` and #2.

```
12848 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
12849   {
12850     \if_charcode:w
12851       \tl_if_head_is_N_type:nTF { #1 ? }
12852       { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
12853       { \str_head:n {#1} }
12854       \exp_not:N #2
12855       \prg_return_true:
12856     \else:
12857       \prg_return_false:
12858     \fi:
12859   }
12860 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_charcode:nN
12861   { f } { p , TF , T , F }
```

For `\tl_if_head_eq_catcode:nN`, again we detect special cases with a `\tl_if_head_is_N_type:n`. Then we need to test if the first token is a begin-group token or an explicit space token, and produce the relevant token, either `\c_group_begin_token` or `\c_space_token`. Again, for an empty argument, a hack is used, removing the token given by the user and leaving two tokens in the input stream which will make the `\if_catcode:w` test return `false`.

```

12862 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
12863   {
12864     \if_catcode:w
12865       \tl_if_head_is_N_type:nTF { #1 ? }
12866         { \__tl_head_exp_not:w #1 { ^ \__tl_if_head_eq_empty_arg:w } \s__tl_stop }
12867       {
12868         \tl_if_head_is_group:nTF {#1}
12869           \c_group_begin_token
12870           \c_space_token
12871         }
12872       \exp_not:N #2
12873     \prg_return_true:
12874   \else:
12875     \prg_return_false:
12876   \fi:
12877 }
12878 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
12879   { o } { p , TF , T , F }
```

For `\tl_if_head_eq_meaning:nN`, again, detect special cases. In the normal case, use `\tl_head:w`, with no `\exp_not:N` this time, since `\if_meaning:w` causes no expansion. With an empty argument, the test is `true`, and `\use_none:nnn` removes `#2` and `\prg_return_true:` and `\else:` (it is safe this way here as in this case `\prg_new_conditional:Npnn` didn't optimize these two away). In the special cases, we know that the first token is a character, hence `\if_charcode:w` and `\if_catcode:w` together are enough. We combine them in some order, hopefully faster than the reverse. Tests are not nested because the arguments may contain unmatched primitive conditionals.

```

12880 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
12881   {
12882     \tl_if_head_is_N_type:nTF { #1 ? }
12883       \__tl_if_head_eq_meaning_normal:nN
12884       \__tl_if_head_eq_meaning_special:nN
12885       {#1} #2
12886   }
12887 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
12888   {
12889     \exp_after:wN \if_meaning:w
12890       \__tl_tl_head:w #1 { ?? \use_none:nnn } \s__tl_stop #2
12891     \prg_return_true:
12892   \else:
12893     \prg_return_false:
12894   \fi:
12895 }
12896 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2
12897   {
12898     \if_charcode:w \str_head:n {#1} \exp_not:N #2
12899     \exp_after:wN \use_ii:nn
```

```

12900     \else:
12901         \prg_return_false:
12902     \fi:
12903     \use_none:n
12904     {
12905         \if_catcode:w \exp_not:N #2
12906             \tl_if_head_is_group:nTF {#1}
12907                 { \c_group_begin_token }
12908                 { \c_space_token }
12909             \prg_return_true:
12910         \else:
12911             \prg_return_false:
12912         \fi:
12913     }
12914 }
```

Both `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` will need to get the first token of their argument and apply `\exp_not:N` to it. `\_tl_head_exp_not:w` does exactly that.

```

12915 \cs_new:Npn \_tl_head_exp_not:w #1 #2 \s__tl_stop
12916     { \exp_not:N #1 }
```

If the argument of `\tl_if_head_eq_charcode:nN` and `\tl_if_head_eq_catcode:nN` was empty `\_tl_if_head_eq_empty_arg:w` will be left in the input stream. This macro has to remove `\exp_not:N` and the following token from the input stream to make sure no unbalanced if-construct is created and leave tokens there which make the two tests return `false`.

```

12917 \cs_new:Npn \_tl_if_head_eq_empty_arg:w \exp_not:N #1
12918     { ? }
```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 107.)

`\tl_if_head_is_N_type_p:n` A token list can be empty, can start with an explicit space character (catcode 10 and charcode 32), can start with a begin-group token (catcode 1), or start with an N-type argument. In the first two cases, the line involving `\_tl_if_head_is_N_type_auxi:w` produces `f` (and otherwise nothing). In the third case (begin-group token), the lines involving `\token_to_str:N` produce a single closing brace. The category code test is thus true exactly in the fourth case, which is what we want. One cannot optimize by moving one of the `\scan_stop:` to the beginning: if #1 contains primitive conditionals, all of its occurrences must be dealt with before the `\if:w` tries to skip the `true` branch of the conditional.

```

12919 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
12920     {
12921         \if:w
12922             \if_false: { \fi: \_tl_if_head_is_N_type_auxi:w \prg_do_nothing: #1 ~ }
12923             { \exp_after:wN { \token_to_str:N #1 } }
12924             \scan_stop: \scan_stop:
12925         \prg_return_true:
12926     \else:
12927         \prg_return_false:
12928     \fi:
12929 }
```

```
12930 \exp_args:Nno \use:n { \cs_new:Npn \_tl_if_head_is_N_type_auxi:w #1 ~ }
```

```

12931   {
12932     \tl_if_empty:cTF { #1 }
12933     { f \exp_after:wN \use_none:n }
12934     { \exp_after:wN \_tl_if_head_is_N_type_auxii:n }
12935     \exp_after:wN { \if_false: } \fi:
12936   }
12937 \cs_new:Npn \_tl_if_head_is_N_type_auxii:n #1
12938   { \exp_after:wN \use_none:n \exp_after:wN }

```

(End definition for `\tl_if_head_is_N_type:nTF` and others. This function is documented on page 108.)

**\tl\_if\_head\_is\_group\_p:n** Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance. The extra ? caters for an empty argument. This could be made faster, but we need all brace tricks to happen in one step of expansion, keeping the token list brace balanced at all times.

```

12939 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
12940   {
12941     \if:w
12942       \exp_after:wN \use_none:n
12943       \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
12944       \scan_stop: \scan_stop:
12945       \_tl_if_head_is_group_fi_false:w
12946     \fi:
12947     \if_true:
12948       \prg_return_true:
12949     \else:
12950       \prg_return_false:
12951     \fi:
12952   }
12953 \cs_new:Npn \_tl_if_head_is_group_fi_false:w \fi: \if_true: { \fi: \if_false: }

```

(End definition for `\tl_if_head_is_group:nTF` and `\_tl_if_head_is_group_fi_false:w`. This function is documented on page 107.)

**\tl\_if\_head\_is\_space\_p:n** The auxiliary's argument is all that is before the first explicit space in `\prg_do_nothing:#1?~`. If that is a single `\prg_do_nothing:` the test yields `true`. Otherwise, that is more than one token, and the test yields `false`. The work is done within braces (with an `\if_false: { \fi: ... }` construction) both to hide potential alignment tab characters from TeX in a table, and to allow for removing what remains of the token list after its first space. The use of `\if:w` ensures that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```

12954 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
12955   {
12956     \if:w
12957       \if_false: { \fi: \_tl_if_head_is_space:w \prg_do_nothing: #1 ? ~ }
12958       \scan_stop: \scan_stop:
12959       \prg_return_true:
12960     \else:
12961       \prg_return_false:
12962     \fi:
12963   }
12964 \exp_args:Nno \use:n { \cs_new:Npn \_tl_if_head_is_space:w #1 ~ }
12965   {
12966     \_tl_if_empty_if:o {#1} \else: f \fi:

```

```

12967     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
12968 }

```

(End definition for `\_tl_if_head_is_space:nTF` and `\_tl_if_head_is_space:w`. This function is documented on page 108.)

## 52.12 Token by token changes

`\s\_tl_act_stop`

The `\_tl_act_...` functions may be applied to any token list. Hence, we use a private quark, to allow any token, even quarks, in the token list. Only `\s\_tl_act_stop` may not appear in the token lists manipulated by `\_tl_act:NNNn` functions.

```

12969 \scan_new:N \s\_tl_act_stop

```

(End definition for `\s\_tl_act_stop`.)

```

\_\_tl_act:NNNn
\_\_tl_act_output:n
\_\_tl_act_reverse_output:n
\_\_tl_act_loop:w
\_\_tl_act_normal:NwNNN
\_\_tl_act_group:nwNNN
\_\_tl_act_space:wwNNN
\_\_tl_act_end:wn
\_\_tl_act_if_head_is_space:nTF
\_\_tl_act_if_head_is_space:w
\_\_tl_act_if_head_is_space_true:w
\_\_tl_use_none_delimit_by_q_act_stop:w

```

To help control the expansion, `\_tl_act:NNNn` should always be preceded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. This way no internal token of it can be accidentally end up in the input stream. Because `\s\_tl_act_stop` can't appear without braces around it in the argument #1 of `\_tl_act_loop:w`, we can use this marker to set up a fast test for leading spaces.

```

12970 \cs_set_protected:Npn \_\_tl_tmp:w #1
12971 {
12972     \cs_new:Npn \_\_tl_act_if_head_is_space:nTF ##1
12973     {
12974         \_\_tl_act_if_head_is_space:w
12975         \s\_tl_act_stop ##1 \s\_tl_act_stop \_\_tl_act_if_head_is_space_true:w
12976         \s\_tl_act_stop #1 \s\_tl_act_stop \use_i:nn
12977     }
12978     \cs_new:Npn \_\_tl_act_if_head_is_space:w
12979         ##1 \s\_tl_act_stop #1 ##2 \s\_tl_act_stop
12980     {}
12981     \cs_new:Npn \_\_tl_act_if_head_is_space_true:w
12982         \s\_tl_act_stop #1 \s\_tl_act_stop \use_i:nn ##1 ##2
12983     {##1}
12984 }
12985 \_\_tl_tmp:w { ~ }

```

(We expand the definition `\_tl_act_if_head_is_space:nTF` when setting up `\_tl_act_loop:w`, so we can then undefine the auxiliary.) In the loop, we check how the token list begins and act accordingly. In the “group” case, we may have reached `\s\_tl_act_stop`, the end of the list. Then leave `\exp_end:` and the result in the input stream, to terminate the expansion of `\exp:w`. Otherwise, apply the relevant function to the “arguments”, #3 and to the head of the token list. Then repeat the loop. The scheme is the same if the token list starts with an N-type or with a space, making sure that `\_tl_act_space:wwNNN` gobble the space.

```

12986 \exp_args:Nnx \use:n { \cs_new:Npn \_\_tl_act_loop:w #1 \s\_tl_act_stop }
12987 {
12988     \exp_not:o { \_\_tl_act_if_head_is_space:nTF {#1} }
12989     \exp_not:N \_\_tl_act_space:wwNNN
12990 {
12991     \exp_not:o { \tl_if_head_is_group:nTF {#1} }
12992     \exp_not:N \_\_tl_act_group:nwNNN
12993     \exp_not:N \_\_tl_act_normal:NwNNN

```

```

12994      }
12995      \exp_not:n {#1} \s__tl_act_stop
12996  }
12997 \cs_undefine:N \__tl_act_if_head_is_space:nTF
12998 \cs_new:Npn \__tl_act_normal:NwNNN #1 #2 \s__tl_act_stop #3
12999  {
13000      #3 #1
13001      \__tl_act_loop:w #2 \s__tl_act_stop
13002      #3
13003  }
13004 \cs_new:Npn \__tl_use_none_delimit_by_s_act_stop:w #1 \s__tl_act_stop { }
13005 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
13006  { \group_align_safe_end: \exp_end: #2 }
13007 \cs_new:Npn \__tl_act_group:nwNNN #1 #2 \s__tl_act_stop #3#4#5
13008  {
13009      \__tl_use_none_delimit_by_s_act_stop:w #1 \__tl_act_end:wn \s__tl_act_stop
13010      #5 {#1}
13011      \__tl_act_loop:w #2 \s__tl_act_stop
13012      #3 #4 #5
13013  }
13014 \exp_last_unbraced:NNo
13015 \cs_new:Npn \__tl_act_space:wwNNN \c_space_tl #1 \s__tl_act_stop #2#3
13016  {
13017      #3
13018      \__tl_act_loop:w #1 \s__tl_act_stop
13019      #2 #3
13020  }

\__tl_act:NNNn loops over tokens, groups, and spaces in #4. {\s_@@_act_stop} serves
as the end of token list marker, the ? after it avoids losing outer braces. The result is
stored as an argument for the dummy function \__tl_act_result:n.

13021 \cs_new:Npn \__tl_act:NNNn #1#2#3#4
13022  {
13023      \group_align_safe_begin:
13024      \__tl_act_loop:w #4 { \s__tl_act_stop } ? \s__tl_act_stop
13025      #1 #3 #2
13026      \__tl_act_result:n { }
13027  }

```

Typically, the output is done to the right of what was already output, using \\_\_tl\_act\_output:n, but for the \\_\_tl\_act\_reverse functions, it should be done to the left.

```

13028 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
13029  { #2 \__tl_act_result:n { #3 #1 } }
13030 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
13031  { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for \\_\_tl\_act:NNNn and others.)

#### \tl\_reverse:n

The goal here is to reverse without losing spaces nor braces. This is done using the general internal function \\_\_tl\_act:NNNn. Spaces and “normal” tokens are output on the left of the current output. Grouped tokens are output to the left but without any reversal within the group.

```

13032 \cs_new:Npn \tl_reverse:n #1
13033  {
13034      \__kernel_exp_not:w \exp_after:wN

```

```

13035   {
13036     \exp:w
13037     \_\_tl_act:NNNn
13038       \_\_tl_reverse_normal:N
13039       \_\_tl_reverse_group_preserve:n
13040       \_\_tl_reverse_space:
13041       {#1}
13042   }
13043 }
13044 \cs_generate_variant:Nn \tl_reverse:n { o , V }
13045 \cs_new:Npn \_\_tl_reverse_normal:N
13046   { \_\_tl_act_reverse_output:n }
13047 \cs_new:Npn \_\_tl_reverse_group_preserve:n #1
13048   { \_\_tl_act_reverse_output:n { {#1} } }
13049 \cs_new:Npn \_\_tl_reverse_space:
13050   { \_\_tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. This function is documented on page 109.)

<code>\tl_reverse:N</code>	This reverses the list, leaving <code>\exp_stop_f:</code> in front, which stops the f-expansion.
<code>\tl_reverse:c</code>	<code>\cs_new_protected:Npn \tl_reverse:N #1</code>
<code>\tl_greverse:N</code>	<code>{ \_\_kernel_tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }</code>
<code>\tl_greverse:c</code>	<code>\cs_new_protected:Npn \tl_greverse:N #1</code>
	<code>{ \_\_kernel_tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }</code>
	<code>\cs_generate_variant:Nn \tl_reverse:N { c }</code>
	<code>\cs_generate_variant:Nn \tl_greverse:N { c }</code>

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 109.)

## 52.13 Using a single item

`\tl_item:nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then `\_\_tl_if_recursion_tail_break:nN` terminates the loop, and returns nothing at all.

```

\_\_tl_item_aux:nn
\_\_tl_item:nn
13057 \cs_new:Npn \tl_item:nn #1#2
13058 {
13059   \exp_args:Nf \_\_tl_item:nn
13060   { \exp_args:Nf \_\_tl_item_aux:nn { \int_eval:n {#2} } {#1} }
13061   #1
13062   \q_\_tl_recursion_tail
13063   \prg_break_point:
13064 }
13065 \cs_new:Npn \_\_tl_item_aux:nn #1#2
13066 {
13067   \int_compare:nNnTF {#1} < 0
13068   { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
13069   {#1}
13070 }
13071 \cs_new:Npn \_\_tl_item:nn #1#2
13072 {
13073   \_\_tl_if_recursion_tail_break:nN {#2} \prg_break:
13074   \int_compare:nNnTF {#1} = 1
13075   { \prg_break:n { \exp_not:n {#2} } }

```

```

13076      { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
13077  }
13078 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
13079 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for `\tl_item:nn` and others. These functions are documented on page 115.)

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

```

13080 \cs_new:Npn \tl_rand_item:n #1
13081 {
13082     \tl_if_blank:nF {#1}
13083     { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
13084 }
13085 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
13086 \cs_generate_variant:Nn \tl_rand_item:N { c }

```

(End definition for `\tl_rand_item:n` and `\tl_rand_item:N`. These functions are documented on page 115.)

`\tl_range:Nnn` To avoid checking for the end of the token list at every step, start by counting the number  $l$  of items and “normalizing” the bounds, namely clamping them to the interval  $[0, l]$  and dealing with negative indices. More precisely, `\__tl_range_items:nnNn` receives the number of items to skip at the beginning of the token list, the index of the last item to keep, a function which is either `\__tl_range:w` or the token list itself. If nothing should be kept, leave {}: this stops the f-expansion of `\tl_head:f` and that function produces an empty result. Otherwise, repeatedly call `\__tl_range_skip:w` to delete #1 items from the input stream (the extra brace group avoids an off-by-one shift). For the braced version `\__tl_range_braced:w` sets up `\__tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. Depending on the first token of the tail, either just move it (if it is a space) or also decrement the number of items left to find. Eventually, the result is a brace group followed by the rest of the token list, and `\tl_head:f` cleans up and gives the result in `\exp_not:n`.

```

13087 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
13088 \cs_generate_variant:Nn \tl_range:Nnn { c }
13089 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
13090 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
13091 {
13092     \tl_head:f
13093     {
13094         \exp_args:Nf \__tl_range:nnnNn
13095         { \tl_count:n {#2} } {#3} {#4} #1 {#2}
13096     }
13097 }
13098 \cs_new:Npn \__tl_range:nnnNn #1#2#3
13099 {
13100     \exp_args:Nff \__tl_range:nnNn
13101     {
13102         \exp_args:Nf \__tl_range_normalize:nn
13103         { \int_eval:n { #2 - 1 } } {#1}
13104     }
13105 {
13106         \exp_args:Nf \__tl_range_normalize:nn
13107         { \int_eval:n {#3} } {#1}
13108 }

```

```

13109    }
13110 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
13111 {
13112     \if_int_compare:w #2 > #1 \exp_stop_f: \else:
13113         \exp_after:wN { \exp_after:wN }
13114     \fi:
13115     \exp_after:wN #3
13116     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
13117     \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
13118 }
13119 \cs_new:Npn \__tl_range_skip:w #1 ; #2
13120 {
13121     \if_int_compare:w #1 > \c_zero_int
13122         \exp_after:wN \__tl_range_skip:w
13123         \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
13124     \else:
13125         \exp_after:wN \exp_end:
13126     \fi:
13127 }
13128 \cs_new:Npn \__tl_range:w #1 ; #2
13129 {
13130     \exp_args:Nf \__tl_range_collect:nn
13131     { \__tl_range_skip_spaces:n {#2} } {#1}
13132 }
13133 \cs_new:Npn \__tl_range_skip_spaces:n #1
13134 {
13135     \tl_if_head_is_space:nTF {#1}
13136     { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
13137     { { } #1 }
13138 }
13139 \cs_new:Npn \__tl_range_collect:nn #1#2
13140 {
13141     \int_compare:nNnTF {#2} = 0
13142     {#1}
13143 {
13144     \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
13145     {
13146         \exp_args:Nf \__tl_range_collect:nn
13147         { \__tl_range_collect_space:nw #1 }
13148         {#2}
13149     }
13150     {
13151         \__tl_range_collect:ff
13152         {
13153             \exp_args:No \tl_if_head_is_N_type:nTF { \use_none:n #1 }
13154             { \__tl_range_collect_N:nN }
13155             { \__tl_range_collect_group:nn }
13156             #1
13157         }
13158         { \int_eval:n { #2 - 1 } }
13159     }
13160 }
13161 }
13162 \cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }

```

```

13163 \cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
13164 \cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
13165 \cs_generate_variant:Nn \__tl_range_collect:nn { ff }

```

(End definition for `\_tl_range:Nnn` and others. These functions are documented on page 116.)

`\__tl_range_normalize:nn`

This function converts an `<index>` argument into an explicit position in the token list (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index>` #1 and the string count #2. If #1 is negative, replace it by `#1 + #2 + 1`, then limit to the range [0, #2].

```

13166 \cs_new:Npn \__tl_range_normalize:nn #1#2
13167 {
13168     \int_eval:n
13169     {
13170         \if_int_compare:w #1 < \c_zero_int
13171             \if_int_compare:w #1 < -#2 \exp_stop_f:
13172                 0
13173             \else:
13174                 #1 + #2 + 1
13175             \fi:
13176         \else:
13177             \if_int_compare:w #1 < #2 \exp_stop_f:
13178                 #1
13179             \else:
13180                 #2
13181             \fi:
13182         \fi:
13183     }
13184 }

```

(End definition for `\_tl_range_normalize:nn`.)

## 52.14 Viewing token lists

```

\tl_show:N Showing token list variables is done after checking that the variable is defined (see
\tl_show:c \__kernel_register_show:N).
\tl_log:N 13185 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
\tl_log:c 13186 \cs_generate_variant:Nn \tl_show:N { c }
\__tl_show:NN 13187 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
13188 \cs_generate_variant:Nn \tl_log:N { c }
13189 \cs_new_protected:Npn \__tl_show:NN #1#2
13190 {
13191     \__kernel_chk_defined:NT #2
13192     {
13193         \exp_args:Nf \tl_if_empty:nTF
13194             { \cs_prefix_spec:N #2 \cs_parameter_spec:N #2 }
13195             {
13196                 \exp_args:Ne #1
13197                     { \token_to_str:N #2 = \__kernel_exp_not:w \exp_after:wN {#2} }
13198             }
13199             {
13200                 \msg_error:nnxxxx { kernel } { bad-type }
13201                     { \token_to_str:N #2 } { \token_to_meaning:N #2 } { tl }

```

```

13202         }
13203     }
13204 }
```

(End definition for `\tl_show:N`, `\tl_log:N`, and `\_tl_show:NN`. These functions are documented on page 110.)

**\tl\_show:n** Many `show` functions are based on `\tl_show:n`. The argument of `\tl_show:n` is line-wrapped using `\iow_wrap:nnN` but with a leading `>~` and trailing period, both removed before passing the wrapped text to the `\showtokens` primitive. This primitive shows the result with a leading `>~` and trailing period.

The token list `\l__tl_internal_a_tl` containing the result of all these manipulations is displayed to the terminal using `\tex_showtokens:D` and an odd `\exp_after:wN` which expand the closing brace to improve the output slightly. The calls to `\_kernel_iow_with:Nnn` ensure that the `\newlinechar` is set to 10 so that the `\iow_newline:` inserted by the line-wrapping code are correctly recognized by TeX, and that `\errorcontextlines` is `-1` to avoid printing irrelevant context.

```

13205 \cs_new_protected:Npn \tl_show:n #1
13206   { \iow_wrap:nnN { >~ \tl_to_str:n {#1} . } { } { } \_tl_show:n }
13207 \cs_new_protected:Npn \_tl_show:n #1
13208   {
13209     \tl_set:Nf \l__tl_internal_a_tl { \_tl_show:w #1 \s__tl_stop }
13210     \_kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
13211     {
13212       \_kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
13213       {
13214         \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
13215         { \exp_after:wN \l__tl_internal_a_tl }
13216       }
13217     }
13218   }
13219 \cs_new:Npn \_tl_show:w #1 > #2 . \s__tl_stop {#2}
```

(End definition for `\tl_show:n`, `\_tl_show:n`, and `\_tl_show:w`. This function is documented on page 110.)

**\tl\_log:n** Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```

13220 \cs_new_protected:Npn \tl_log:n #1
13221   { \iow_wrap:nnN { >~ \tl_to_str:n {#1} . } { } { } \iow_log:n }
```

(End definition for `\tl_log:n`. This function is documented on page 111.)

**\\_kernel\_chk\_tl\_type:NnnT** Helper for checking that `#1` has the correct internal structure to be of a certain type. Make sure that it is defined and that it is a token list, namely a macro with no `\long` nor `\protected` prefix. Then compare `#1` to an attempt at reconstructing a valid structure of the given type using `#2` (see implementation of `\seq_show:N` for instance). If that is successful run the requested code `#4`.

```

13222 \cs_new_protected:Npn \_kernel_chk_tl_type:NnnT #1#2#3#4
13223   {
13224     \_kernel_chk_defined:NT #1
13225     {
13226       \exp_args:Nf \tl_if_empty:nTF
13227         { \cs_prefix_spec:N #1 \cs_parameter_spec:N #1 }
```

```

13228   {
13229     \tl_set:Nx \l__tl_internal_a_tl {#3}
13230     \tl_if_eq:NNTF #1 \l__tl_internal_a_tl
13231       {#4}
13232       {
13233         \msg_error:nnxxxx { kernel } { bad-type }
13234           { \token_to_str:N #1 } { \tl_to_str:N #1 }
13235           {#2} { \tl_to_str:N \l__tl_internal_a_tl }
13236       }
13237     }
13238     {
13239       \msg_error:nnxxxx { kernel } { bad-type }
13240           { \token_to_str:N #1 } { \token_to_meaning:N #1 } {#2}
13241     }
13242   }
13243 }
```

(End definition for `\_kernel_chk_tl_type:NnnT`.)

## 52.15 Internal scan marks

`\s__tl_nil` Internal scan marks. These are defined here at the end because the code for `\scan_new:N` depends on some `\tl` functions.

`\s__tl_mark`

`\s__tl_stop`

```

13244 \scan_new:N \s__tl_nil
13245 \scan_new:N \s__tl_mark
13246 \scan_new:N \s__tl_stop
```

(End definition for `\s__tl_nil`, `\s__tl_mark`, and `\s__tl_stop`.)

## 52.16 Scratch token lists

`\g_tmpa_tl` Global temporary token list variables. They are supposed to be set and used immediately, with no delay between the definition and the use because you can't count on other macros not to redefine them from under you.

```

13247 \tl_new:N \g_tmpa_tl
13248 \tl_new:N \g_tmpb_tl
```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 120.)

`\l_tmpa_tl` These are local temporary token list variables. Be sure not to assume that the value you put into them will survive for long—see discussion above.

```

13249 \tl_new:N \l_tmpa_tl
13250 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_tl` and `\l_tmpb_tl`. These variables are documented on page 119.)

We finally clean up a temporary control sequence that we have used at various points to set up some definitions.

```

13251 \cs_undefine:N \_tl_tmp:w
13252 </package>
```

# Chapter 53

## I3str implementation

```
13253  <*package>
13254  <@@=str>
```

### 53.1 Internal auxiliaries

\s\_\_str\_mark Internal scan marks.

```
13255  \scan_new:N \s__str_mark
13256  \scan_new:N \s__str_stop
```

(End definition for \s\_\_str\_mark and \s\_\_str\_stop.)

\\_str\_use\_none\_delimit\_by\_s\_stop:w Functions to gobble up to a scan mark.

```
13257  \cs_new:Npn \_str_use_none_delimit_by_s_stop:w #1 \s__str_stop { }
13258  \cs_new:Npn \_str_use_i_delimit_by_s_stop:nw #1 #2 \s__str_stop {#1}
```

(End definition for \\_str\_use\_none\_delimit\_by\_s\_stop:w and \\_str\_use\_i\_delimit\_by\_s\_stop:nw.)

\q\_\_str\_recursion\_tail Internal recursion quarks.

```
13259  \quark_new:N \q__str_recursion_tail
13260  \quark_new:N \q__str_recursion_stop
```

(End definition for \q\_\_str\_recursion\_tail and \q\_\_str\_recursion\_stop.)

\\_str\_if\_recursion\_tail\_break:NN Functions to query recursion quarks.

```
13261  \__kernel_quark_new_test:N \_str_if_recursion_tail_break:NN
13262  \__kernel_quark_new_test:N \_str_if_recursion_tail_stop_do:Nn
```

(End definition for \\_str\_if\_recursion\_tail\_break:NN and \\_str\_if\_recursion\_tail\_stop\_do:Nn.)

## 53.2 Creating and setting string variables

```

\str_new:N A string is simply a token list. The full mapping system isn't set up yet so do things by
\str_new:c hand.

\str_use:N 13263 \group_begin:
\str_use:c 13264   \cs_set_protected:Npn \__str_tmp:n #1
\str_clear:N 13265   {
\str_clear:c 13266     \tl_if_blank:nF {#1}
\str_gclear:N 13267     {
\str_gclear:c 13268       \cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }
\str_clear_new:N 13269       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }
\str_clear_new:c 13270       \__str_tmp:n
\str_gclear_new:N 13271     }
\str_gclear_new:c 13272   }
\str_set_eq:NN 13273   \__str_tmp:n
\str_set_eq:cN 13274     { new }
\str_set_eq:Nc 13275     { use }
\str_set_eq:cc 13276     { clear }
\str_set_eq:cc 13277     { gclear }
\str_gset_eq:NN 13278     { clear_new }
\str_gset_eq:cN 13279     { gclear_new }
\str_gset_eq:Nc 13280     { }
\str_gset_eq:cc 13281 \group_end:
\str_concat:NNN 13282   \cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN
\str_concat:ccc 13283   \cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN
\str_gconcat:NNN 13284   \cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }
\str_gconcat:NNN 13285   \cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }
\str_gconcat:ccc 13286   \cs_new_eq:NN \str_concat:NNN \tl_concat:NNN
\str_gconcat:ccc 13287   \cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN
\str_gconcat:ccc 13288   \cs_generate_variant:Nn \str_concat:NNN { ccc }
\str_gconcat:ccc 13289   \cs_generate_variant:Nn \str_gconcat:NNN { ccc }

(End definition for \str_new:N and others. These functions are documented on page 122.)

```

\str\_set:Nn Simply convert the token list inputs to *⟨strings⟩*.

```

\str_set:NV 13290 \group_begin:
\str_set:Nx 13291   \cs_set_protected:Npn \__str_tmp:n #1
\str_set:cn 13292   {
\str_set:cV 13293     \tl_if_blank:nF {#1}
\str_set:cx 13294     {
\str_gset:Nn 13295       \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
\str_gset:NV 13296       {
\str_gset:Nx 13297         \exp_not:c { tl_ #1 :Nx } ##1
\str_gset:cn 13298         { \exp_not:N \tl_to_str:n {##2} }
\str_gset:cV 13299       }
\str_gset:cx 13300       \cs_generate_variant:cn { str_ #1 :Nn } { NV , Nx , cn , cV , cx }
\str_gset:cx 13301       \__str_tmp:n
\str_const:Nn 13302     }
\str_const:NV 13303   }
\str_const:Nx 13304   \__str_tmp:n
\str_const:cn 13305     { set }
\str_const:cV 13306     { gset }
\str_const:cx 13307     { const }
\str_const:cx 13308     { put_left }

\str_put_left:Nn
\str_put_left:NV
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cV
\str_put_left:cx

\str_gput_left:Nn
\str_gput_left:NV
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cV

```

```

13309 { gput_left }
13310 { put_right }
13311 { gput_right }
13312 { }
13313 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 123.)

### 53.3 Modifying string variables

Start by applying `\tl_to_str:n` to convert the old and new token lists to strings, and also apply `\tl_to_str:N` to avoid any issues if we are fed a token list variable. Then the code is a much simplified version of the token list code because neither the delimiter nor the replacement can contain macro parameters or braces. The delimiter `\s__str_-mark` cannot appear in the string to edit so it is used in all cases. Some x-expansion is unnecessary. There is no need to avoid losing braces nor to protect against expansion. The ending code is much simplified and does not need to hide in braces.

```

13314 \cs_new_protected:Npn \str_replace_once:Nnn
13315 { \__str_replace>NNNnn \prg_do_nothing: \__kernel_tl_set:Nx }
13316 \cs_new_protected:Npn \str_greplace_once:Nnn
13317 { \__str_replace>NNNnn \prg_do_nothing: \__kernel_tl_gset:Nx }
13318 \cs_new_protected:Npn \str_replace_all:Nnn
13319 { \__str_replace>NNNnn \__str_replace_next:w \__kernel_tl_set:Nx }
13320 \cs_new_protected:Npn \str_greplace_all:Nnn
13321 { \__str_replace>NNNnn \__str_replace_next:w \__kernel_tl_gset:Nx }
13322 \cs_generate_variant:Nn \str_replace_once:Nnn { c }
13323 \cs_generate_variant:Nn \str_greplace_once:Nnn { c }
13324 \cs_generate_variant:Nn \str_replace_all:Nnn { c }
13325 \cs_generate_variant:Nn \str_greplace_all:Nnn { c }
13326 \cs_new_protected:Npn \__str_replace>NNNnn #1#2#3#4#5
13327 {
13328     \tl_if_empty:nTF {#4}
13329     {
13330         \msg_error:nnx { kernel } { empty-search-pattern } {#5}
13331     }
13332     {
13333         \use:x
13334         {
13335             \exp_not:n { \__str_replace_aux>NNNnnn #1 #2 #3 }
13336             { \tl_to_str:N #3 }
13337             { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
13338         }
13339     }
13340 }
13341 \cs_new_protected:Npn \__str_replace_aux>NNNnnn #1#2#3#4#5#6
13342 {
13343     \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
13344     #2 #3
13345     {
13346         \__str_replace_next:w
13347         #4
13348         \__str_use_none_delimit_by_s_stop:w
13349         #5

```

```

13350           \s__str_stop
13351       }
13352   }
13353 \cs_new_eq:NN \__str_replace_next:w ?

```

(End definition for `\str_replace_all:Nnn` and others. These functions are documented on page 130.)

`\str_remove_once:Nn` Removal is just a special case of replacement.

```

\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn
13354 \cs_new_protected:Npn \str_remove_once:Nn #1#2
13355   { \str_replace_once:Nnn #1 {#2} { } }
13356 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
13357   { \str_greplace_once:Nnn #1 {#2} { } }
13358 \cs_generate_variant:Nn \str_remove_once:Nn { c }
13359 \cs_generate_variant:Nn \str_gremove_once:Nn { c }

```

(End definition for `\str_remove_once:Nn` and `\str_gremove_once:Nn`. These functions are documented on page 130.)

`\str_remove_all:Nn` Removal is just a special case of replacement.

```

\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn
13360 \cs_new_protected:Npn \str_remove_all:Nn #1#2
13361   { \str_replace_all:Nnn #1 {#2} { } }
13362 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
13363   { \str_greplace_all:Nnn #1 {#2} { } }
13364 \cs_generate_variant:Nn \str_remove_all:Nn { c }
13365 \cs_generate_variant:Nn \str_gremove_all:Nn { c }

```

(End definition for `\str_remove_all:Nn` and `\str_gremove_all:Nn`. These functions are documented on page 130.)

## 53.4 String comparisons

`\str_if_empty_p:N` More copy-paste!

```

\str_if_empty_p:c
\str_if_empty_p:NTF
\str_if_empty:cTF
\str_if_empty_p:n
\str_if_empty:nTF
\str_if_exist_p:N
\str_if_exist_p:c
\str_if_exist:NTF
\str_if_exist:cTF
13366 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
13367   { p , T , F , TF }
13368 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
13369   { p , T , F , TF }
13370 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
13371   { p , T , F , TF }
13372 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
13373   { p , T , F , TF }
13374 \prg_new_eq_conditional:NNn \str_if_empty:n \tl_if_empty:n
13375   { p , T , F , TF }

```

(End definition for `\str_if_empty:NTF`, `\str_if_empty:nTF`, and `\str_if_exist:NTF`. These functions are documented on page 123.)

`\__str_if_eq:nn` String comparisons rely on the primitive `\(pdf)strcmp`, so we define a new name for it.

```
13376 \cs_new_eq:NN \__str_if_eq:nn \tex_strcmp:D
```

(End definition for `\__str_if_eq:nn`.)

**\str\_compare\_p:nNn** Simply rely on `\_str_if_eq:nn`, which expands to -1, 0 or 1. The `ee` version is created directly because it is more efficient.

```

13377 \prg_new_conditional:Npnn \str_compare:nNn #1#2#3 { p , T , F , TF }
13378 {
13379     \if_int_compare:w
13380         \_str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#3} }
13381         #2 \c_zero_int
13382         \prg_return_true: \else: \prg_return_false: \fi:
13383     }
13384 \prg_new_conditional:Npnn \str_compare:eNe #1#2#3 { p , T , F , TF }
13385 {
13386     \if_int_compare:w \_str_if_eq:nn {#1} {#3} #2 \c_zero_int
13387         \prg_return_true: \else: \prg_return_false: \fi:
13388 }

```

(End definition for `\str_compare:nNnTF`. This function is documented on page [125](#).)

**\str\_if\_eq\_p:nn** Modern engines provide a direct way of comparing two token lists, but returning a number. This set of conditionals therefore makes life a bit clearer. The `nn` and `ee` versions are created directly as this is most efficient. Since `\_str_if_eq:nn` will expand to 0 as an explicit character with category 12 if the two lists match (and either -1 or 1 if they don't) we can use `\if:w` here which is faster than using `\if_int_compare:w`.

```

13389 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
13390 {
13391     \if:w 0 \_str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
13392         \prg_return_true: \else: \prg_return_false: \fi:
13393     }
13394 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
13395     { V , v , o , nV , no , VV , nv } { p , T , F , TF }
13396 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
13397 {
13398     \if:w 0 \_str_if_eq:nn {#1} {#2}
13399         \prg_return_true: \else: \prg_return_false: \fi:
13400 }

```

(End definition for `\str_if_eq:nnTF`. This function is documented on page [124](#).)

**\str\_if\_eq\_p:NN** Note that `\str_if_eq:NNTF` is different from `\tl_if_eq:NNTF` because it needs to ignore category codes.

```

13401 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
13402 {
13403     \if:w 0 \_str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
13404         \prg_return_true: \else: \prg_return_false: \fi:
13405     }
13406 \prg_generate_conditional_variant:Nnn \str_if_eq:NN
13407     { c , Nc , cc } { T , F , TF , p }

```

(End definition for `\str_if_eq:NNTF`. This function is documented on page [123](#).)

**\str\_if\_in:NnTF** Everything here needs to be detokenized but beyond that it is a simple token list test. **\str\_if\_in:cNTF** It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of **\str\_if\_in:nnTF** directly but that takes more code.

```

13408 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
13409 {

```

```
13410 \use:x
13411   { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
13412   { \prg_return_true: } { \prg_return_false: }
13413 }
13414 \prg_generate_conditional_variant:Nnn \str_if_in:Nn
13415   { c } { T , F , TF }
13416 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
13417 {
13418   \use:x
13419     { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
13420     { \prg_return_true: } { \prg_return_false: }
13421 }
```

(End definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 124.)

```

\str_case:nn Much the same as \tl_case:nnTF here: just a change in the internal comparison.
\str_case:Vn
\str_case:on
\str_case:nV
\str_case:nv
\str_case:nnTF
\str_case:VnTF
\str_case:onTF
\str_case:nVT
\str_case:nvTF
    \str_case:Nn
\str_case:NnTF
\str_case_e:nn
\str_case_e:nnTF
\__str_case:nnTF
\__str_case_e:nnTF
    \__str_case:nw
\__str_case_e:nw
\__str_case_end:nw

13422 \cs_new:Npn \str_case:nn #1#2
13423 {
13424     \exp:w
13425     \__str_case:nnTF {#1} {#2} { } { }
13426 }
13427 \cs_new:Npn \str_case:nnT #1#2#3
13428 {
13429     \exp:w
13430     \__str_case:nnTF {#1} {#2} {#3} { }
13431 }
13432 \cs_new:Npn \str_case:nnF #1#2
13433 {
13434     \exp:w
13435     \__str_case:nnTF {#1} {#2} { }
13436 }
13437 \cs_new:Npn \str_case:nnTF #1#2
13438 {
13439     \exp:w
13440     \__str_case:nnTF {#1} {#2}
13441 }
13442 \cs_new:Npn \__str_case:nnTF #1#2#3#4
13443 { \__str_case:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13444 \cs_generate_variant:Nn \str_case:nn { V , o , nV , nv }
13445 \prg_generate_conditional_variant:Nnn \str_case:nn
13446 { V , o , nV , nv } { T , F , TF }
13447 \cs_new_eq:NN \str_case:Nn \str_case:Vn
13448 \cs_new_eq:NN \str_case:NnT \str_case:VnT
13449 \cs_new_eq:NN \str_case:NnF \str_case:VnF
13450 \cs_new_eq:NN \str_case:NnTF \str_case:VnTF
13451 \cs_new:Npn \__str_case:nw #1#2#3
13452 {
13453     \str_if_eq:nnTF {#1} {#2}
13454     { \__str_case_end:nw {#3} }
13455     { \__str_case:nw {#1} }
13456 }
13457 \cs_new:Npn \str_case_e:nn #1#2
13458 {

```

```

13459      \exp:w
13460      \__str_case_e:nnTF {#1} {#2} { } { }
13461    }
13462 \cs_new:Npn \str_case_e:nnT #1#2#3
13463  {
13464    \exp:w
13465    \__str_case_e:nnTF {#1} {#2} {#3} { }
13466  }
13467 \cs_new:Npn \str_case_e:nnF #1#2
13468  {
13469    \exp:w
13470    \__str_case_e:nnTF {#1} {#2} { }
13471  }
13472 \cs_new:Npn \str_case_e:nnTF #1#2
13473  {
13474    \exp:w
13475    \__str_case_e:nnTF {#1} {#2}
13476  }
13477 \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
13478  { \__str_case_e:nw {#1} #2 {#1} { } \s__str_mark {#3} \s__str_mark {#4} \s__str_stop }
13479 \cs_new:Npn \__str_case_e:nw #1#2#3
13480  {
13481    \str_if_eq:eeTF {#1} {#2}
13482    { \__str_case_end:nw {#3} }
13483    { \__str_case_e:nw {#1} }
13484  }
13485 \cs_new:Npn \__str_case_end:nw #1#2#3 \s__str_mark #4#5 \s__str_stop
13486  { \exp_end: #1 #4 }

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 124.)

## 53.5 Mapping over strings

```

\str_map_function:NN
\str_map_function:cN
\str_map_function:nN
  \str_map_inline:Nn
  \str_map_inline:cn
  \str_map_inline:nn
\str_map_variable:NNn
\str_map_variable:cNn
\str_map_variable:nNn
  \str_map_break:
  \str_map_break:n
\__str_map_function:w
\__str_map_function:nn
  \__str_map_inline>NN
\__str_map_variable:NnN

```

The inline and variable mappings are similar to the usual token list mappings but start out by turning the argument to an “other string”. Doing the same for the expandable function mapping would require `\_kernel_str_to_other:n`, quadratic in the string length. To deal with spaces in that case, `\_str_map_function:w` replaces the following space by a braced space and a further call to itself. These are received by `\_str_map_function:nn`, which passes the space to `#1` and calls `\_str_map_function:w` to deal with the next space. The space before the braced space allows to optimize the `\q__str_recursion_tail` test. Of course we need to include a trailing space (the question mark is needed to avoid losing the space when TeX tokenizes the line). At the cost of about three more auxiliaries this code could get a 9 times speed up by testing only every 9-th character for whether it is `\q__str_recursion_tail` (also by converting 9 spaces at a time in the `\str_map_function:nN` case).

For the `map_variable` functions we use a string assignment to store each character because spaces are made catcode 12 before the loop.

```

13487 \cs_new:Npn \str_map_function:nN #1#
13488  {
13489    \exp_after:wN \__str_map_function:w
13490    \exp_after:wN \__str_map_function:nn \exp_after:wN #2
13491      \_kernel_tl_to_str:w {#1}

```

```

13492      \q__str_recursion_tail ? ~
13493      \prg_break_point:Nn \str_map_break: { }
13494  }
13495 \cs_new:Npn \str_map_function:NN
13496  { \exp_args:No \str_map_function:nN }
13497 \cs_new:Npn \__str_map_function:w #1 ~
13498  { #1 { ~ { ~ } \__str_map_function:w } }
13499 \cs_new:Npn \__str_map_function:nn #1#2
13500  {
13501      \if_meaning:w \q__str_recursion_tail #2
13502      \exp_after:wN \str_map_break:
13503      \fi:
13504      #1 #2 \__str_map_function:nn {#1}
13505  }
13506 \cs_generate_variant:Nn \str_map_function:NN { c }
13507 \cs_new_protected:Npn \str_map_inline:nn #1#2
13508  {
13509      \int_gincr:N \g__kernel_prg_map_int
13510      \cs_gset_protected:cpn
13511      { \__str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
13512      \use:x
13513      {
13514          \exp_not:N \__str_map_inline:NN
13515          \exp_not:c { \__str_map_ \int_use:N \g__kernel_prg_map_int :w }
13516          \__kernel_str_to_other_fast:n {#1}
13517      }
13518      \q__str_recursion_tail
13519      \prg_break_point:Nn \str_map_break:
13520      { \int_gdecr:N \g__kernel_prg_map_int }
13521  }
13522 \cs_new_protected:Npn \str_map_inline:Nn
13523  { \exp_args:No \str_map_inline:nn }
13524 \cs_generate_variant:Nn \str_map_inline:Nn { c }
13525 \cs_new:Npn \__str_map_inline:NN #1#2
13526  {
13527      \__str_if_recursion_tail_break:NN #2 \str_map_break:
13528      \exp_args:No #1 { \token_to_str:N #2 }
13529      \__str_map_inline:NN #1
13530  }
13531 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
13532  {
13533      \use:x
13534      {
13535          \exp_not:n { \__str_map_variable:NnN #2 {#3} }
13536          \__kernel_str_to_other_fast:n {#1}
13537      }
13538      \q__str_recursion_tail
13539      \prg_break_point:Nn \str_map_break: { }
13540  }
13541 \cs_new_protected:Npn \str_map_variable:NNn
13542  { \exp_args:No \str_map_variable:nNn }
13543 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
13544  {
13545      \__str_if_recursion_tail_break:NN #3 \str_map_break:

```

```

13546     \str_set:Nn #1 {#3}
13547     \use:n {#2}
13548     \__str_map_variable:NnN #1 {#2}
13549 }
13550 \cs_generate_variant:Nn \str_map_variable:NNn { c }
13551 \cs_new:Npn \str_map_break:
13552   { \prg_map_break:Nn \str_map_break: { } }
13553 \cs_new:Npn \str_map_break:n
13554   { \prg_map_break:Nn \str_map_break: }

```

(End definition for `\str_map_function>NN` and others. These functions are documented on page 125.)

`\str_map_tokens:Nn`  
`\str_map_tokens:cn`  
`\str_map_tokens:nn`

Uses an auxiliary of `\str_map_function>NN`.

```

13555 \cs_new:Npn \str_map_tokens:nn #1#2
13556 {
13557   \exp_args:Nno \use:nn
13558   { \__str_map_function:w \__str_map_function:nn {#2} }
13559   { \__kernel_tl_to_str:w {#1} }
13560   \q_recursion_tail ? ~
13561   \prg_break_point:Nn \str_map_break: { }
13562 }
13563 \cs_new:Npn \str_map_tokens:Nn { \exp_args:No \str_map_tokens:nn }
13564 \cs_generate_variant:Nn \str_map_tokens:Nn { c }

```

(End definition for `\str_map_tokens:Nn` and `\str_map_tokens:nn`. These functions are documented on page 126.)

## 53.6 Accessing specific characters in a string

`\__kernel_str_to_other:n`  
`\__str_to_other_loop:w`  
`\__str_to_other_end:w`

First apply `\tl_to_str:n`, then replace all spaces by “other” spaces, 8 at a time, storing the converted part of the string between the `\s__str_mark` and `\s__str_stop` markers. The end is detected when `\__str_to_other_loop:w` finds one of the trailing A, distinguished from any contents of the initial token list by their category. Then `\__str_to_other_end:w` is called, and finds the result between `\s__str_mark` and the first A (well, there is also the need to remove a space).

```

13565 \cs_new:Npn \__kernel_str_to_other:n #1
13566 {
13567   \exp_after:wN \__str_to_other_loop:w
13568   \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_mark \s__str_stop
13569 }
13570 \group_begin:
13571 \tex_lccode:D ‘\* = ‘\ %
13572 \tex_lccode:D ‘\A = ‘\A %
13573 \tex_lowercase:D
13574 {
13575   \group_end:
13576   \cs_new:Npn \__str_to_other_loop:w
13577   #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \s__str_stop
13578   {
13579     \if_meaning:w A #8
13580       \__str_to_other_end:w
13581     \fi:
13582     \__str_to_other_loop:w

```

```

13583     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \s__str_stop
13584 }
13585 \cs_new:Npn \__str_to_other_end:w {fi: #1 \s__str_mark #2 * A #3 \s__str_stop
13586 { \fi: #2 }
13587 }

(End definition for \__kernel_str_to_other:n, \__str_to_other_loop:w, and \__str_to_other-
end:w.)

```

The difference with \\_\_kernel\_str\_to\_other:n is that the converted part is left in the input stream, making these commands only restricted-expandable.

```

13588 \cs_new:Npn \__kernel_str_to_other_fast:n #
13589 {
13590     \exp_after:wN \__str_to_other_fast_loop:w \tl_to_str:n {#1} ~
13591     A ~ A ~ A ~ A ~ A ~ A ~ A ~ \s__str_stop
13592 }
13593 \group_begin:
13594 \tex_lccode:D '\* = '\ %
13595 \tex_lccode:D '\A = '\A %
13596 \tex_lowercase:D
13597 {
13598     \group_end:
13599     \cs_new:Npn \__str_to_other_fast_loop:w
13600     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
13601     {
13602         \if_meaning:w A #9
13603             \__str_to_other_fast_end:w
13604             \fi:
13605             #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
13606             \__str_to_other_fast_loop:w *
13607     }
13608     \cs_new:Npn \__str_to_other_fast_end:w #1 * A #2 \s__str_stop {#1}
13609 }

```

(End definition for \\_\_kernel\_str\_to\_other\_fast:n, \\_\_kernel\_str\_to\_other\_fast\_loop:w, and \\_\_-
str\_to\_other\_fast\_end:w.)

### \str\_item:Nn

### \str\_item:cN

### \str\_item:nn

### \str\_item\_ignore\_spaces:nn

### \\_\_str\_item:nn

### \\_\_str\_item:w

The \str\_item:nn hands its argument with spaces escaped to \\_\_str\_item:nn, and makes sure to turn the result back into a proper string (with category code 10 spaces) eventually. The \str\_item\_ignore\_spaces:nn function does not escape spaces, which are thus ignored by \\_\_str\_item:nn since everything else is done with undelimited arguments. Evaluate the *<index>* argument #2 and count characters in the string, passing those two numbers to \\_\_str\_item:w for further analysis. If the *<index>* is negative, shift it by the *<count>* to know the how many character to discard, and if that is still negative give an empty result. If the *<index>* is larger than the *<count>*, give an empty result, and otherwise discard *<index>* - 1 characters before returning the following one. The shift by -1 is obtained by inserting an empty brace group before the string in that case: that brace group also covers the case where the *<index>* is zero.

```

13610 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }
13611 \cs_generate_variant:Nn \str_item:Nn { c }
13612 \cs_new:Npn \str_item:nn #1#2
13613 {
13614     \exp_args:Nf \tl_to_str:n
13615     {

```

```

13616      \exp_args:Nf \__str_item:nn
13617          { \__kernel_str_to_other:n {#1} } {#2}
13618      }
13619  }
13620 \cs_new:Npn \str_item_ignore_spaces:nn #1
13621  { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
13622 \cs_new:Npn \__str_item:nn #1#2
13623  {
13624      \exp_after:wN \__str_item:w
13625      \int_value:w \int_eval:n {#2} \exp_after:wN ;
13626      \int_value:w \__str_count:n {#1} ;
13627      #1 \s__str_stop
13628  }
13629 \cs_new:Npn \__str_item:w #1; #2;
13630  {
13631      \int_compare:nNnTF {#1} < 0
13632      {
13633          \int_compare:nNnTF {#1} < {-#2}
13634          { \__str_use_none_delimit_by_s_stop:w }
13635          {
13636              \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13637              \exp:w \exp_after:wN \__str_skip_exp_end:w
13638              \int_value:w \int_eval:n { #1 + #2 } ;
13639          }
13640      }
13641  {
13642      \int_compare:nNnTF {#1} > {#2}
13643      { \__str_use_none_delimit_by_s_stop:w }
13644      {
13645          \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13646          \exp:w \__str_skip_exp_end:w #1 ; { }
13647      }
13648  }
13649 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 128.)

`\__str_skip_exp_end:w`  
`\__str_skip_loop:wNNNNNNNN`  
`\__str_skip_end:w`  
`\__str_skip_end>NNNNNNNN`

Removes `max(#1,0)` characters from the input stream, and then leaves `\exp_end:`. This should be expanded using `\exp:w`. We remove characters 8 at a time until there are at most 8 to remove. Then we do a dirty trick: the `\if_case:w` construction leaves between 0 and 8 times the `\or:` control sequence, and those `\or:` become arguments of `\__str_skip_end>NNNNNNNN`. If the number of characters to remove is 6, say, then there are two `\or:` left, and the 8 arguments of `\__str_skip_end>NNNNNNNN` are the two `\or:`, and 6 characters from the input stream, exactly what we wanted to remove. Then close the `\if_case:w` conditional with `\fi:`, and stop the initial expansion with `\exp_end:` (see places where `\__str_skip_exp_end:w` is called).

```

13650 \cs_new:Npn \__str_skip_exp_end:w #1;
13651  {
13652      \if_int_compare:w #1 > 8 \exp_stop_f:
13653          \exp_after:wN \__str_skip_loop:wNNNNNNNN
13654      \else:
13655          \exp_after:wN \__str_skip_end:w
13656          \int_value:w \int_eval:w
13657      \fi:

```

```

13658      #1 ;
13659    }
13660 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
13661  {
13662    \exp_after:wN \__str_skip_exp_end:w
13663    \int_value:w \int_eval:n { #1 - 8 } ;
13664  }
13665 \cs_new:Npn \__str_skip_end:w #1 ;
13666  {
13667    \exp_after:wN \__str_skip_end:NNNNNNNN
13668    \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or: \or:
13669  }
13670 \cs_new:Npn \__str_skip_end:NNNNNNNN #1#2#3#4#5#6#7#8 { \fi: \exp_end: }
```

(End definition for `\__str_skip_exp_end:w` and others.)

```

\str_range:Nnn
\str_range:nnn
\str_range_ignore_spaces:nnn
\__str_range:nnn
\__str_range:w
\__str_range:nnw
```

Sanitize the string. Then evaluate the arguments. At this stage we also decrement the `<start index>`, since our goal is to know how many characters should be removed. Then limit the range to be non-negative and at most the length of the string (this avoids needing to check for the end of the string when grabbing characters), shifting negative numbers by the appropriate amount. Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```

13671 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
13672 \cs_generate_variant:Nn \str_range:Nnn { c }
13673 \cs_new:Npn \str_range:nnn #1#2#3
13674  {
13675    \exp_args:Nf \tl_to_str:n
13676    {
13677      \exp_args:Nf \__str_range:nnn
13678      { \__kernel_str_to_other:n {#1} } {#2} {#3}
13679    }
13680  }
13681 \cs_new:Npn \str_range_ignore_spaces:nnn #1
13682  { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
13683 \cs_new:Npn \__str_range:nnn #1#2#3
13684  {
13685    \exp_after:wN \__str_range:w
13686    \int_value:w \__str_count:n {#1} \exp_after:wN ;
13687    \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
13688    \int_value:w \int_eval:n {#3} ;
13689    #1 \s__str_stop
13690  }
13691 \cs_new:Npn \__str_range:w #1; #2; #3;
13692  {
13693    \exp_args:Nf \__str_range:nnw
13694    { \__str_range_normalize:nn {#2} {#1} }
13695    { \__str_range_normalize:nn {#3} {#1} }
13696  }
13697 \cs_new:Npn \__str_range:nnw #1#2
13698  {
13699    \exp_after:wN \__str_collect_delimit_by_q_stop:w
13700    \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
13701    \exp:w \__str_skip_exp_end:w #1 ;
13702  }
```

(End definition for `\str_range:Nnn` and others. These functions are documented on page 129.)

`\__str_range_normalize:nn`

This function converts an `<index>` argument into an explicit position in the string (a result of 0 denoting “out of bounds”). Expects two explicit integer arguments: the `<index>` #1 and the string count #2. If #1 is negative, replace it by `#1 + #2 + 1`, then limit to the range [0, #2].

```
13703 \cs_new:Npn \__str_range_normalize:nn #1#2
13704 {
13705     \int_eval:n
13706     {
13707         \if_int_compare:w #1 < \c_zero_int
13708             \if_int_compare:w #1 < -#2 \exp_stop_f:
13709                 0
13710             \else:
13711                 #1 + #2 + 1
13712             \fi:
13713         \else:
13714             \if_int_compare:w #1 < #2 \exp_stop_f:
13715                 #1
13716             \else:
13717                 #2
13718             \fi:
13719         \fi:
13720     }
13721 }
```

(End definition for `\__str_range_normalize:nn`.)

`\__str_collect_delimit_by_q_stop:w`  
`\__str_collect_loop:wn`  
  `\__str_collect_loop:wnNNNNNNNN`  
`\__str_collect_end:wn`  
`\__str_collect_end:nnnnnnnnnn`

Collects `max(#1, 0)` characters, and removes everything else until `\s__str_stop`. This is somewhat similar to `\__str_skip_exp_end:w`, but accepts integer expression arguments. This time we can only grab 7 characters at a time. At the end, we use an `\if_case:w` trick again, so that the 8 first arguments of `\__str_collect_end:nnnnnnnnnn` are some `\or:`, followed by an `\fi:`, followed by #1 characters from the input stream. Simply leaving this in the input stream closes the conditional properly and the `\or:` disappear.

```
13722 \cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
13723     { \__str_collect_loop:wn #1 ; { } }
13724 \cs_new:Npn \__str_collect_loop:wn #1 ;
13725     {
13726         \if_int_compare:w #1 > 7 \exp_stop_f:
13727             \exp_after:wN \__str_collect_loop:wnNNNNNNNN
13728         \else:
13729             \exp_after:wN \__str_collect_end:wn
13730         \fi:
13731         #1 ;
13732     }
13733 \cs_new:Npn \__str_collect_loop:wnNNNNNNNN #1; #2 #3#4#5#6#7#8#9
13734     {
13735         \exp_after:wN \__str_collect_loop:wn
13736         \int_value:w \int_eval:n { #1 - 7 } ;
13737         { #2 #3#4#5#6#7#8#9 }
13738     }
13739 \cs_new:Npn \__str_collect_end:wn #1 ;
13740     {
```

```

13741     \exp_after:wN \__str_collect_end:nnnnnnnnw
13742     \if_case:w \if_int_compare:w #1 > \c_zero_int
13743         #1 \else: 0 \fi: \exp_stop_f:
13744         \or: \or: \or: \or: \or: \or: \or: \or: \or:
13745     }
13746 \cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \s__str_stop
13747     { #1#2#3#4#5#6#7#8 }

```

(End definition for `\__str_collect_delimit_by_q_stop:w` and others.)

### 53.7 Counting characters

`\str_count_spaces:N`  
`\str_count_spaces:c`  
`\str_count_spaces:n`  
`\__str_count_spaces_loop:w`

To speed up this function, we grab and discard 9 space-delimited arguments in each iteration of the loop. The loop stops when the last argument is one of the trailing `X<number>`, and that `<number>` is added to the sum of 9 that precedes, to adjust the result.

```

13748 \cs_new:Npn \str_count_spaces:N
13749     { \exp_args:No \str_count_spaces:n }
13750 \cs_generate_variant:Nn \str_count_spaces:N { c }
13751 \cs_new:Npn \str_count_spaces:n #1
13752     {
13753         \int_eval:n
13754         {
13755             \exp_after:wN \__str_count_spaces_loop:w
13756             \tl_to_str:n {#1} ~
13757             X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
13758             \s__str_stop
13759         }
13760     }
13761 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
13762     {
13763         \if_meaning:w X #9
13764             \__str_use_i_delimit_by_s_stop:nw
13765         \fi:
13766         9 + \__str_count_spaces_loop:w
13767     }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `\__str_count_spaces_loop:w`. These functions are documented on page 127.)

`\str_count:N`  
`\str_count:c`  
`\str_count:n`  
`\str_count_ignore_spaces:n`  
`\__str_count:n`  
`\__str_count_aux:n`  
`\__str_count_loop:NNNNNNNNN`

To count characters in a string we could first escape all spaces using `\__kernel_str_to_other:n`, then pass the result to `\tl_count:n`. However, the escaping step would be quadratic in the number of characters in the string, and we can do better. Namely, sum the number of spaces (`\str_count_spaces:n`) and the result of `\tl_count:n`, which ignores spaces. Since strings tend to be longer than token lists, we use specialized functions to count characters ignoring spaces. Namely, loop, grabbing 9 non-space characters at each step, and end as soon as we reach one of the 9 trailing items. The internal function `\__str_count:n`, used in `\str_item:nn` and `\str_range:nnn`, is similar to `\str_count_ignore_spaces:n` but expects its argument to already be a string or a string with spaces escaped.

```

13768 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
13769 \cs_generate_variant:Nn \str_count:N { c }

```

```

13770 \cs_new:Npn \str_count:n #1
13771 {
13772     \__str_count_aux:n
13773     {
13774         \str_count_spaces:n {#1}
13775         + \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1}
13776     }
13777 }
13778 \cs_new:Npn \__str_count:n #1
13779 {
13780     \__str_count_aux:n
13781     { \__str_count_loop:NNNNNNNN #1 }
13782 }
13783 \cs_new:Npn \str_count_ignore_spaces:n #1
13784 {
13785     \__str_count_aux:n
13786     { \exp_after:wN \__str_count_loop:NNNNNNNN \tl_to_str:n {#1} }
13787 }
13788 \cs_new:Npn \__str_count_aux:n #1
13789 {
13790     \int_eval:n
13791     {
13792         #1
13793         { X 8 } { X 7 } { X 6 }
13794         { X 5 } { X 4 } { X 3 }
13795         { X 2 } { X 1 } { X 0 }
13796         \s__str_stop
13797     }
13798 }
13799 \cs_new:Npn \__str_count_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
13800 {
13801     \if_meaning:w X #9
13802     \exp_after:wN \__str_use_none_delimit_by_s_stop:w
13803     \fi:
13804     9 + \__str_count_loop:NNNNNNNN
13805 }

```

(End definition for `\str_count:N` and others. These functions are documented on page 127.)

## 53.8 The first character in a string

<code>\str_head:N</code>	The <code>_ignore_spaces</code> variant applies <code>\tl_to_str:n</code> then grabs the first item, thus skipping spaces. As usual, <code>\str_head:N</code> expands its argument and hands it to <code>\str_head:n</code> .
<code>\str_head:c</code>	
<code>\str_head:n</code>	To circumvent the fact that TeX skips spaces when grabbing undelimited macro parameters, <code>\__str_head:w</code> takes an argument delimited by a space. If #1 starts with a non-space character, <code>\__str_use_i_delimit_by_s_stop:nw</code> leaves that in the input stream. On the other hand, if #1 starts with a space, the <code>\__str_head:w</code> takes an empty argument, and the single (initially braced) space in the definition of <code>\__str_head:w</code> makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of <code>\str_head:n</code> gives an empty result after passing through <code>\__str_use_i_delimit_by_s_stop:nw</code> .
<code>\str_head_ignore_spaces:n</code>	
<code>\__str_head:w</code>	

```
13806 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
```

```

13807 \cs_generate_variant:Nn \str_head:N { c }
13808 \cs_new:Npn \str_head:n #1
13809 {
13810     \exp_after:wN \__str_head:w
13811     \tl_to_str:n {#1}
13812     { f } ~ \s__str_stop
13813 }
13814 \cs_new:Npn \__str_head:w #1 ~ %
13815 { \__str_use_i_delimit_by_s_stop:nw #1 { ~ } }
13816 \cs_new:Npn \str_head_ignore_spaces:n #1
13817 {
13818     \exp_after:wN \__str_use_i_delimit_by_s_stop:nw
13819     \tl_to_str:n {#1} { } \s__str_stop
13820 }

```

(End definition for `\str_head:N` and others. These functions are documented on page 128.)

**\str\_tail:N** Getting the tail is a little bit more convoluted than the head of a string. We hit the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:`. This removes the first character, and necessarily makes the test true, since the character cannot match `\scan_stop:`. The auxiliary function then inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input stream. The details are such that an empty string has an empty tail (this requires in particular that the end-marker X be unexpandable and not a control sequence). The `_ignore_spaces` is rather simpler: after converting the input to a string, `\__str_tail_auxii:w` removes one undelimited argument and leaves everything else until an end-marker `\s__str_mark`. One can check that an empty (or blank) string yields an empty tail.

```

13821 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
13822 \cs_generate_variant:Nn \str_tail:N { c }
13823 \cs_new:Npn \str_tail:n #1
13824 {
13825     \exp_after:wN \__str_tail_auxi:w
13826     \reverse_if:N \if_charcode:w
13827         \scan_stop: \tl_to_str:n {#1} X X \s__str_stop
13828 }
13829 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \s__str_stop { \fi: #1 }
13830 \cs_new:Npn \str_tail_ignore_spaces:n #1
13831 {
13832     \exp_after:wN \__str_tail_auxii:w
13833     \tl_to_str:n {#1} \s__str_mark \s__str_mark \s__str_stop
13834 }
13835 \cs_new:Npn \__str_tail_auxii:w #1 #2 \s__str_mark #3 \s__str_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 128.)

## 53.9 String manipulation

**\str\_casetofold:n** Case changing for programmatic reasons is done by first detokenizing input then doing a simple loop that only has to worry about spaces and everything else. The output is detokenized to allow data sharing with text-based case changing. Similarly, for 8-bit engines the multi-byte information is shared.

```

13836 \cs_new:Npn \str_casetofold:n #1 { \__str_change_case:nn {#1} { casetofold } }

```

```

\__str_change_case:nn
\__str_change_case_aux:nn
\__str_change_case_result:n
\__str_change_case_output:nw
\__str_change_case_output:fw
\__str_change_case_end:nw
\__str_change_case_loop:nw
\__str_change_case_space:n
\__str_change_case_char:nN
    \str_change_case_char:nw

```

```

13837 \cs_new:Npn \str_lowercase:n #1 { \__str_change_case:nn {#1} { lowercase } }
13838 \cs_new:Npn \str_uppercase:n #1 { \__str_change_case:nn {#1} { uppercase } }
13839 \cs_generate_variant:Nn \str_casetof:n { V }
13840 \cs_generate_variant:Nn \str_lowercase:n { f }
13841 \cs_generate_variant:Nn \str_uppercase:n { f }
13842 \cs_new:Npn \__str_change_case:nn #1
13843 {
13844     \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
13845         { \tl_to_str:n {#1} }
13846 }
13847 \cs_new:Npn \__str_change_case_aux:nn #1#2
13848 {
13849     \__str_change_case_loop:nw {#2} #1 \q__str_recursion_tail \q__str_recursion_stop
13850         \__str_change_case_result:n { }
13851 }
13852 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
13853 { #2 \__str_change_case_result:n { #3 #1 } }
13854 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
13855 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
13856 { \tl_to_str:n {#2} }
13857 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q__str_recursion_stop
13858 {
13859     \tl_if_head_is_space:nTF {#2}
13860         { \__str_change_case_space:n }
13861         { \__str_change_case_char:nN }
13862     {#1} #2 \q__str_recursion_stop
13863 }
13864 \exp_last_unbraced:NNNNo
13865 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
13866 {
13867     \__str_change_case_output:nw { ~ }
13868     \__str_change_case_loop:nw {#1}
13869 }
13870 \cs_new:Npn \__str_change_case_char:nN #1#2
13871 {
13872     \__str_if_recursion_tail_stop_do:Nn #2
13873         { \__str_change_case_end:wn }
13874         \__str_change_case_codepoint:nN {#1} #2
13875 }
13876 \if_int_compare:w 0
13877     \cs_if_exist:NT \tex_XeTeXversion:D { 1 }
13878     \cs_if_exist:NT \tex_luatexversion:D { 1 }
13879     > 0 \exp_stop_f:
13880     \cs_new:Npn \__str_change_case_codepoint:nN #1#2
13881         { \__str_change_case_char:fnn { \int_eval:n {'#2} } {#1} {#2} }
13882 \else:
13883     \cs_new:Npn \__str_change_case_codepoint:nN #1#2
13884 {
13885     \int_compare:nNnTF {'#2} > { "80 }
13886     {
13887         \int_compare:nNnTF {'#2} < { "E0 }
13888         { \__str_change_case_codepoint:nNN }
13889     {
13890         \int_compare:nNnTF {'#2} < { "F0 }

```

```

13891           { \__str_change_case_codepoint:nNNN }
13892           { \__str_change_case_codepoint:nNNNNN }
13893       }
13894   }
13895   { \__str_change_case_char_aux:nN }
13896   {#1} #2
13897 }
13898 \cs_new:Npn \__str_change_case_char_aux:nN #1#2
13899   { \__str_change_case_char:fnn { \int_eval:n {`#2} } {#1} {#2} }
13900 \cs_new:Npn \__str_change_case_codepoint:nNN #1#2#3
13901   {
13902     \__str_change_case_char:fnn
13903     { \int_eval:n { (`#2 - "CO) * "40 + `#3 - "80 } }
13904     {#1} {#2#3}
13905   }
13906 \cs_new:Npn \__str_change_case_codepoint:nNNN #1#2#3#4
13907   {
13908     \__str_change_case_char:fnn
13909     {
13910       \int_eval:n
13911       { (`#2 - "E0) * "1000 + (`#3 - "80) * "40 + `#4 - "80 }
13912     }
13913     {#1} {#2#3#4}
13914   }
13915 \cs_new:Npn \__str_change_case_codepoint:nNNNN #1#2#3#4#5
13916   {
13917     \__str_change_case_char:fnn
13918     {
13919       \int_eval:n
13920       {
13921         (`#2 - "F0) * "40000
13922         + (`#3 - "80) * "1000
13923         + (`#4 - "80) * "40
13924         + `#5 - "80
13925       }
13926     }
13927     {#1} {#2#3#4#5}
13928   }
13929 \fi:
13930 \cs_new:Npn \__str_change_case_char:nnn #1#2#3
13931   {
13932     \__str_change_case_output:fw
13933     {
13934       \exp_args:Ne \__str_change_case_char_aux:nnn
13935       { \__kernel_codepoint_case:nn {#2} {#1} } {#1} {#3}
13936     }
13937     \__str_change_case_loop:nw {#2}
13938   }
13939 \cs_generate_variant:Nn \__str_change_case_char:nnn { f }
13940 \cs_new:Npn \__str_change_case_char_aux:nnn #1#2#3
13941   {
13942     \use:e { \__str_change_case_char:nnnn #1 {#2} {#3} }
13943   }
13944 \cs_new:Npn \__str_change_case_char:nnnn #1#2#3#4#5

```

```

13945      {
13946        \int_compare:nNnTF {#1} = {#4}
13947          { \tl_to_str:n {#5} }
13948          {
13949            \codepoint_str_generate:n {#1}
13950            \tl_if_blank:nF {#2}
13951              {
13952                \codepoint_str_generate:n {#2}
13953                \tl_if_blank:nF {#3}
13954                  { \codepoint_str_generate:n {#3} }
13955              }
13956            }
13957        }

```

(End definition for `\str_casfold:n` and others. These functions are documented on page 132.)

`\c_ampersand_str`  
`\c_atsign_str`  
`\c_backslash_str`  
`\c_left_brace_str`  
`\c_right_brace_str`  
`\c_circumflex_str`  
`\c_colon_str`  
`\c_dollar_str`  
`\c_hash_str`  
`\c_percent_str`  
`\c_tilde_str`  
`\c_underscore_str`  
`\c_zero_str`

For all of those strings, use `\cs_to_str:N` to get characters with the correct category code without worries

```

13958 \str_const:Nx \c_ampersand_str  { \cs_to_str:N \& }
13959 \str_const:Nx \c_atsign_str   { \cs_to_str:N \@ }
13960 \str_const:Nx \c_backslash_str { \cs_to_str:N \\ }
13961 \str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }
13962 \str_const:Nx \c_right_brace_str { \cs_to_str:N \} }
13963 \str_const:Nx \c_circumflex_str { \cs_to_str:N \^ }
13964 \str_const:Nx \c_colon_str    { \cs_to_str:N \: }
13965 \str_const:Nx \c_dollar_str   { \cs_to_str:N \$ }
13966 \str_const:Nx \c_hash_str    { \cs_to_str:N \# }
13967 \str_const:Nx \c_percent_str { \cs_to_str:N \% }
13968 \str_const:Nx \c_tilde_str   { \cs_to_str:N \~ }
13969 \str_const:Nx \c_underscore_str { \cs_to_str:N \_ }
13970 \str_const:Nx \c_zero_str    { 0 }

```

(End definition for `\c_ampersand_str` and others. These variables are documented on page 133.)

`\l_tmpa_str` Scratch strings.

`\l_tmpb_str`  
`\g_tmpa_str`  
`\g_tmpb_str`

```

13971 \str_new:N \l_tmpa_str
13972 \str_new:N \l_tmpb_str
13973 \str_new:N \g_tmpa_str
13974 \str_new:N \g_tmpb_str

```

(End definition for `\l_tmpa_str` and others. These variables are documented on page 133.)

## 53.10 Viewing strings

`\str_show:n` Displays a string on the terminal.

`\str_show:N`  
`\str_show:c`  
`\str_log:n`  
`\str_log:N`  
`\str_log:c`

```

13975 \cs_new_eq:NN \str_show:n \tl_show:n
13976 \cs_new_protected:Npn \str_show:N #1
13977   {
13978     \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
13979       { \tl_show:N #1 }
13980   }
13981 \cs_generate_variant:Nn \str_show:N { c }
13982 \cs_new_eq:NN \str_log:n \tl_log:n

```

```
13983 \cs_new_protected:Npn \str_log:N #1
13984   {
13985     \__kernel_chk_tl_type:NnnT #1 { str } { \tl_to_str:N #1 }
13986     { \tl_log:N #1 }
13987   }
13988 \cs_generate_variant:Nn \str_log:N { c }
```

(End definition for `\str_show:n` and others. These functions are documented on page 132.)

```
13989 </package>
```

# Chapter 54

## I3str-convert implementation

```
13990  <*package>
13991  <@@=str>
```

### 54.1 Helpers

#### 54.1.1 Variables and constants

<code>\__str_tmp:w</code>	Internal scratch space for some functions.
<code>\l__str_internal_t1</code>	<pre>13992 \cs_new_protected:Npn \__str_tmp:w { } 13993 \tl_new:N \l__str_internal_t1</pre> <p>(End definition for <code>\__str_tmp:w</code> and <code>\l__str_internal_t1</code>.)</p>
<code>\g__str_result_t1</code>	The <code>\g__str_result_t1</code> variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.
	<pre>13994 \tl_new:N \g__str_result_t1</pre> <p>(End definition for <code>\g__str_result_t1</code>.)</p>
<code>\c__str_replacement_char_int</code>	When converting, invalid bytes are replaced by the Unicode replacement character "FFFD".
	<pre>13995 \int_const:Nn \c__str_replacement_char_int { "FFFD }</pre> <p>(End definition for <code>\c__str_replacement_char_int</code>.)</p>
<code>\c__str_max_byte_int</code>	The maximal byte number.
	<pre>13996 \int_const:Nn \c__str_max_byte_int { 255 }</pre> <p>(End definition for <code>\c__str_max_byte_int</code>.)</p>
<code>\s__str</code>	Internal scan marks.
	<pre>13997 \scan_new:N \s__str</pre> <p>(End definition for <code>\s__str</code>.)</p>

\q\_str\_nil	Internal quarks.
	13998 \quark_new:N \q\_str\_nil
	(End definition for \q\_str\_nil.)
\g\_str\_alias\_prop	To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.
	<pre> 13999 \prop_new:N \g\_str_alias_prop 14000 \prop_gput:Nnn \g\_str_alias_prop { latin1 } { iso88591 } 14001 \prop_gput:Nnn \g\_str_alias_prop { latin2 } { iso88592 } 14002 \prop_gput:Nnn \g\_str_alias_prop { latin3 } { iso88593 } 14003 \prop_gput:Nnn \g\_str_alias_prop { latin4 } { iso88594 } 14004 \prop_gput:Nnn \g\_str_alias_prop { latin5 } { iso88599 } 14005 \prop_gput:Nnn \g\_str_alias_prop { latin6 } { iso885910 } 14006 \prop_gput:Nnn \g\_str_alias_prop { latin7 } { iso885913 } 14007 \prop_gput:Nnn \g\_str_alias_prop { latin8 } { iso885914 } 14008 \prop_gput:Nnn \g\_str_alias_prop { latin9 } { iso885915 } 14009 \prop_gput:Nnn \g\_str_alias_prop { latin10 } { iso885916 } 14010 \prop_gput:Nnn \g\_str_alias_prop { utf16le } { utf16 } 14011 \prop_gput:Nnn \g\_str_alias_prop { utf16be } { utf16 } 14012 \prop_gput:Nnn \g\_str_alias_prop { utf32le } { utf32 } 14013 \prop_gput:Nnn \g\_str_alias_prop { utf32be } { utf32 } 14014 \prop_gput:Nnn \g\_str_alias_prop { hexadecimal } { hex } 14015 \bool_lazy_any:nTF 14016 { 14017   \sys_if_engine_luatex_p: 14018   \sys_if_engine_xetex_p: 14019 } 14020 { 14021   \prop_gput:Nnn \g\_str_alias_prop { default } { } 14022 } 14023 { 14024   \prop_gput:Nnn \g\_str_alias_prop { default } { utf8 } 14025 }</pre>
	(End definition for \g\_str\_alias\_prop.)
\g\_str\_error\_bool	In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.
	14026 \bool_new:N \g\_str_error_bool
	(End definition for \g\_str_error_bool.)
str\_byte str\_error	Conversions from one <i>encoding</i> / <i>escaping</i> pair to another are done within x-expanding assignments. Errors are signalled by raising the relevant flag.
	14027 \flag_new:n { str_byte } 14028 \flag_new:n { str_error }
	(End definition for str\_byte and str\_error. These variables are documented on page ??.)

## 54.2 String conditionals

```
\_\_str_if_contains_char:NnT
\_\_str_if_contains_char:NnTF
\_\_str_if_contains_char:nntF
    \_\_str_if_contains_char_aux:nN
\_\_str_if_contains_char_auxi:nN
\_\_str_if_contains_char_true:
```

`\_\_str_if_contains_char:nnTF {<token list>} <char>`  
 Expects the *<token list>* to be an *<other string>*: the caller is responsible for ensuring that no (too-)special catcodes remain. Loop over the characters of the string, comparing character codes. The loop is broken if character codes match. Otherwise we return “false”.

```
14029 \prg_new_conditional:Npnn \_\_str_if_contains_char:Nn #1#2 { T , TF }
14030   {
14031     \exp_after:wN \_\_str_if_contains_char_aux:nn \exp_after:wN {#1} {#2}
14032     { \prg_break:n { ? \fi: } }
14033     \prg_break_point:
14034     \prg_return_false:
14035   }
14036 \cs_new:Npn \_\_str_if_contains_char_auxi:nn #1#2
14037   { \_\_str_if_contains_char_auxi:nN {#2} #1 }
14038 \prg_new_conditional:Npnn \_\_str_if_contains_char:nn #1#2 { TF }
14039   {
14040     \_\_str_if_contains_char_auxi:nN {#2} #1 { \prg_break:n { ? \fi: } }
14041     \prg_break_point:
14042     \prg_return_false:
14043   }
14044 \cs_new:Npn \_\_str_if_contains_char_auxi:nN #1#2
14045   {
14046     \if_charcode:w #1 #2
14047       \exp_after:wN \_\_str_if_contains_char_true:
14048       \fi:
14049       \_\_str_if_contains_char_auxi:nN {#1}
14050   }
14051 \cs_new:Npn \_\_str_if_contains_char_true:
14052   { \prg_break:n { \prg_return_true: \use_none:n } }
```

(End definition for `\_\_str_if_contains_char:NnT` and others.)

```
\_\_str_octal_use:NTF
```

`\_\_str_octal_use:NTF <token> {<true code>} {<false code>}`

If the *<token>* is an octal digit, it is left in the input stream, *followed* by the *<true code>*. Otherwise, the *<false code>* is left in the input stream.

**TEXhackers note:** This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. TEX dutifully detects octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is ’1#1, greater than 1. Otherwise, the right-hand side stops as ’1, and the conditional takes the `false` branch.

```
14053 \prg_new_conditional:Npnn \_\_str_octal_use:N #1 { TF }
14054   {
14055     \if_int_compare:w 1 < '1 \token_to_str:N #1 \exp_stop_f:
14056       #1 \prg_return_true:
14057     \else:
14058       \prg_return_false:
14059     \fi:
14060   }
```

(End definition for `\_\_str_octal_use:NTF`.)

`\__str_hexadecimal_use:NTF` T<sub>E</sub>X detects uppercase hexadecimal digits for us (see `\__str_octal_use:NTF`), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

14061 \prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
14062 {
14063   \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
14064     #1 \prg_return_true:
14065   \else:
14066     \if_case:w \int_eval:n { \exp_after:wN ` \token_to_str:N #1 - 'a }
14067       A
14068       \or: B
14069       \or: C
14070       \or: D
14071       \or: E
14072       \or: F
14073     \else:
14074       \prg_return_false:
14075       \exp_after:wN \use_none:n
14076     \fi:
14077     \prg_return_true:
14078   \fi:
14079 }
```

(End definition for `\__str_hexadecimal_use:NTF`.)

## 54.3 Conversions

### 54.3.1 Producing one byte or character

For each integer  $N$  in the range  $[0, 255]$ , we create a constant token list which holds three character tokens with category code other: the character with character code  $N$ , followed by the representation of  $N$  as two hexadecimal digits. The value  $-1$  is given a default token list which ensures that later functions give an empty result for the input  $-1$ .

```

14080 \group_begin:
14081   \kernel_tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
14082   \tl_map_inline:Nn \l__str_internal_tl
14083   {
14084     \tl_map_inline:Nn \l__str_internal_tl
14085     {
14086       \tl_const:cx { c__str_byte_ \int_eval:n {"#1##1} _tl }
14087       { \char_generate:nn { "#1##1" } { 12 } #1 ##1 }
14088     }
14089   }
14090 \group_end:
14091 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }
```

(End definition for `\c__str_byte_0_tl` and others.)

`\__str_output_byte:n` `\__str_output_byte:w` `\__str_output_hexadecimal:n` `\__str_output_end:` Those functions must be used carefully: feeding them a value outside the range  $[-1, 255]$  will attempt to use the undefined token list variable `\c__str_byte_{number}_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value  $-1$  produces an empty result in both cases.

```

14092 \cs_new:Npn \__str_output_byte:n #1
14093   { \__str_output_byte:w #1 \__str_output_end: }
14094 \cs_new:Npn \__str_output_byte:w
14095   {
14096     \exp_after:wN \exp_after:wN
14097     \exp_after:wN \use_i:nnn
14098     \cs:w c__str_byte_ \int_eval:w
14099   }
14100 \cs_new:Npn \__str_output_hexadecimal:n #1
14101   {
14102     \exp_after:wN \exp_after:wN
14103     \exp_after:wN \use_none:n
14104     \cs:w c__str_byte_ \int_eval:n {#1} _tl \cs_end:
14105   }
14106 \cs_new:Npn \__str_output_end:
14107   { \scan_stop: _tl \cs_end: }

(End definition for \__str_output_byte:n and others.)

```

\\_\_str\_output\_byte\_pair\_be:n  
\\_\_str\_output\_byte\_pair\_le:n  
\\_\_str\_output\_byte\_pair:nnN

Convert a number in the range [0,65535] to a pair of bytes, either big-endian or little-endian.

```

14108 \cs_new:Npn \__str_output_byte_pair_be:n #1
14109   {
14110     \exp_args:Nf \__str_output_byte_pair:nnN
14111       { \int_div_truncate:nn { #1 } { "100" } {#1} \use:nn
14112     }
14113 \cs_new:Npn \__str_output_byte_pair_le:n #1
14114   {
14115     \exp_args:Nf \__str_output_byte_pair:nnN
14116       { \int_div_truncate:nn { #1 } { "100" } {#1} \use_i:i:nn
14117     }
14118 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
14119   {
14120     #3
14121       { \__str_output_byte:n { #1 } }
14122       { \__str_output_byte:n { #2 - #1 * "100" } }
14123   }

(End definition for \__str_output_byte_pair_be:n, \__str_output_byte_pair_le:n, and \__str_output_byte_pair:nnN.)

```

### 54.3.2 Mapping functions for conversions

\\_\_str\_convert\_gmap:N  
\\_\_str\_convert\_gmap\_loop>NN

This maps the function #1 over all characters in \g\_\_str\_result\_tl, which should be a byte string in most cases, sometimes a native string.

```

14124 \cs_new_protected:Npn \__str_convert_gmap:N #1
14125   {
14126     \__kernel_tl_gset:Nx \g__str_result_tl
14127     {
14128       \exp_after:wN \__str_convert_gmap_loop>NN
14129       \exp_after:wN #1
14130       \g__str_result_tl { ? \prg_break: }
14131       \prg_break_point:
14132     }

```

```

14133     }
14134 \cs_new:Npn \__str_convert_gmap_loop:NN #1#2
14135 {
14136     \use_none:n #2
14137     #1#2
14138     \__str_convert_gmap_loop:NN #1
14139 }
```

(End definition for `\__str_convert_gmap:N` and `\__str_convert_gmap:NN`.)

This maps the function #1 over all character codes in `\g__str_result_tl`, which must be in the internal representation.

```

14140 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
14141 {
14142     \__kernel_tl_gset:Nx \g__str_result_tl
14143 {
14144     \exp_after:wN \__str_convert_gmap_internal_loop:Nww
14145     \exp_after:wN #1
14146     \g__str_result_tl \s__str \s__str_stop \prg_break: \s__str
14147     \prg_break_point:
14148 }
14149 }
14150 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__str #3 \s__str
14151 {
14152     \__str_use_none_delimit_by_s_stop:w #3 \s__str_stop
14153     #1 {#3}
14154     \__str_convert_gmap_internal_loop:Nww #1
14155 }
```

(End definition for `\__str_convert_gmap_internal:N` and `\__str_convert_gmap_internal_loop:Nw`.)

### 54.3.3 Error-reporting during conversion

When converting using the function `\str_set_convert:Nnnn`, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically `\@@_error`), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions `\str_set_convert:NnnnTF`, errors should be suppressed. This is done by changing `\__str_if_flag_error:nnx` into `\__str_if_flag_no_error:nnx` locally.

```

14156 \cs_new_protected:Npn \__str_if_flag_error:nnx #1
14157 {
14158     \flag_if_raised:nTF {#1}
14159     { \msg_error:nnx { str } }
14160     { \use_none:nn }
14161 }
14162 \cs_new_protected:Npn \__str_if_flag_no_error:nnx #1#2#3
14163 { \flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }
```

(End definition for `\__str_if_flag_error:nnx` and `\__str_if_flag_no_error:nnx`.)

At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```

14164 \cs_new:Npn \__str_if_flag_times:nT #1#2
14165   { \flag_if_raised:nT {#1} { #2~(x \flag_height:n {#1} ) } }

```

(End definition for `\__str_if_flag_times:nT`.)

#### 54.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of TEX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

`(bytes) \s__str ⟨Unicode code point⟩ \s__str`

where we have collected the `(bytes)` which combined to form this particular Unicode character, and the `⟨Unicode code point⟩` is in the range [0, "10FFFF].

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

`\str_set_convert:Nnnn`  
`\str_gset_convert:Nnnn`  
`\str_set_convert:NnnnTF`  
`\str_gset_convert:NnnnTF`  
`\__str_convert:nNNnnn`

```

14166 \cs_new_protected:Npn \str_set_convert:Nnnn
14167   { \__str_convert:nNNnnn { } \tl_set_eq:NN }
14168 \cs_new_protected:Npn \str_gset_convert:Nnnn
14169   { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
14170 \prg_new_protected_conditional:Npnn
14171   \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
14172   {
14173     \bool_gset_false:N \g__str_error_bool
14174     \__str_convert:nNNnnn
14175       { \cs_set_eq:NN \__str_if_flag_error:nmx \__str_if_flag_no_error:nmx }
14176       \tl_set_eq:NN #1 {#2} {#3} {#4}
14177     \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
14178   }
14179 \prg_new_protected_conditional:Npnn
14180   \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }

```

```

14181  {
14182      \bool_gset_false:N \g__str_error_bool
14183      \__str_convert:nNNnnn
14184      { \cs_set_eq:NN \__str_if_flag_error:nnx \__str_if_flag_no_error:nnx }
14185      \tl_gset_eq:NN #1 {#2} {#3} {#4}
14186      \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
14187  }
14188 \cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
14189  {
14190      \group_begin:
14191          #1
14192          \kernel_tl_gset:Nx \g__str_result_tl { \kernel_str_to_other_fast:n {#4} }
14193          \exp_after:wN \__str_convert:wwnn
14194          \tl_to_str:n {#5} /// \s__str_stop
14195          { decode } { unescape }
14196          \prg_do_nothing:
14197          \__str_convert_decode_:
14198          \exp_after:wN \__str_convert:wwnn
14199          \tl_to_str:n {#6} /// \s__str_stop
14200          { encode } { escape }
14201          \use_i:i:nn
14202          \__str_convert_encode_:
14203      \group_end:
14204      #2 #3 \g__str_result_tl
14205  }

```

(End definition for `\str_set_convert:Nnnn` and others. These functions are documented on page 136.)

`\__str_convert:wwnn`  
`\__str_convert:NNnnn`

The task of `\__str_convert:wwnn` is to split `<encoding>/<escaping>` pairs into their components, #1 and #2. Calls to `\__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- #1 is the encoding conversion function;
- #2 is the escaping function;
- #3 is the escaping name for use in an error message;
- #4 is `\prg_do_nothing:` for unescaping/decoding, and `\use_i:i:nn` for encoding/escaping;
- #5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do #2#1 in this order, and in the encoding/escaping phase, the order should be reversed: #4#2#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, #1, should be performed.

```

14206 \cs_new_protected:Npn \__str_convert:wwnn
14207     #1 / #2 // #3 \s__str_stop #4#5
14208  {
14209      \__str_convert:nnn {enc} {#4} {#1}
14210      \__str_convert:nnn {esc} {#5} {#2}
14211      \exp_args:Ncc \__str_convert:NNnnn

```

```

14212      { __str_convert_#4_#1: } { __str_convert_#5_#2: } {#2}
14213  }
14214 \cs_new_protected:Npn __str_convert>NNnNN #1#2#3#4#5
14215  {
14216      \if_meaning:w #1 #5
14217          \tl_if_empty:nF {#3}
14218              { \msg_error:nnx { str } { native-escaping } {#3} }
14219          #1
14220      \else:
14221          #4 #2 #1
14222      \fi:
14223  }

```

(End definition for `\__str_convert:wwnn` and `\__str_convert:NNnNN`.)

`\__str_convert:nnn`

The arguments of `\__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `\__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```

14224 \cs_new_protected:Npn __str_convert:nnn #1#2#3
14225  {
14226      \cs_if_exist:cF { __str_convert_#2_#3: }
14227  {
14228      \exp_args:Nx __str_convert:nnnn
14229          { __str_convert_lowercase_alphanum:n {#3} }
14230          {#1} {#2} {#3}
14231  }
14232  }
14233 \cs_new_protected:Npn __str_convert:nnnn #1#2#3#4
14234  {
14235      \cs_if_exist:cF { __str_convert_#3_#1: }
14236  {
14237      \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
14238          { \tl_set:Nn \l__str_internal_tl {#1} }
14239      \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
14240          {
14241          \file_if_exist:nTF { 13str-#2- \l__str_internal_tl .def }

```

```

14242 {
14243   \group_begin:
14244     \cctab_select:N \c_code_cctab
14245     \file_input:n { l3str-#2- \l__str_internal_tl .def }
14246     \group_end:
14247   }
14248   {
14249     \tl_clear:N \l__str_internal_tl
14250     \msg_error:n{str} { unknown-#2 } {#4} {#1}
14251   }
14252 }
14253 \cs_if_exist:cF { __str_convert_#3_#1: }
14254 {
14255   \cs_gset_eq:cc { __str_convert_#3_#1: }
14256   { __str_convert_#3_ \l__str_internal_tl : }
14257 }
14258 }
14259 \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
14260 }

```

(End definition for `\__str_convert:nnn` and `\__str_convert:nnnn`.)

This function keeps only letters and digits, with upper case letters converted to lower case.

```

14261 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
14262   {
14263     \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
14264     \tl_to_str:n {#1} { ? \prg_break: }
14265     \prg_break_point:
14266   }
14267 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
14268   {
14269     \use_none:n #1
14270     \if_int_compare:w '#1 > 'Z \exp_stop_f:
14271       \if_int_compare:w '#1 > 'z \exp_stop_f: \else:
14272         \if_int_compare:w '#1 < 'a \exp_stop_f: \else:
14273           #1
14274           \fi:
14275           \fi:
14276     \else:
14277       \if_int_compare:w '#1 < 'A \exp_stop_f:
14278         \if_int_compare:w 1 < 1#1 \exp_stop_f:
14279           #1
14280           \fi:
14281         \else:
14282           \__str_output_byte:n { '#1 + 'a - 'A }
14283           \fi:
14284         \fi:
14285       \__str_convert_lowercase_alphanum_loop:N
14286   }

```

(End definition for `\__str_convert_lowercase_alphanum:n` and `\__str_convert_lowercase_alphanum_loop:N`.)

### 54.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

In the case of 8-bit engines, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `str_byte`. Spaces have already been given the correct category code when this function is called.

```

14287 \bool_lazy_any:nTF
14288 {
14289   \sys_if_engine_luatex_p:
14290   \sys_if_engine_xetex_p:
14291 }
14292 {
14293   \cs_new:Npn \__str_filter_bytes:n #1
14294   {
14295     \__str_filter_bytes_aux:N #1
14296     { ? \prg_break: }
14297     \prg_break_point:
14298   }
14299   \cs_new:Npn \__str_filter_bytes_aux:N #1
14300   {
14301     \use_none:n #1
14302     \if_int_compare:w '#1 < 256 \exp_stop_f:
14303       #1
14304     \else:
14305       \flag_raise:n { str_byte }
14306     \fi:
14307     \__str_filter_bytes_aux:N
14308   }
14309 }
14310 { \cs_new_eq:NN \__str_filter_bytes:n \use:n }
```

(End definition for `\__str_filter_bytes:n` and `\__str_filter_bytes_aux:N`.)

`\__str_convert_unescape_:` The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

```

14311 \bool_lazy_any:nTF
14312 {
14313   \sys_if_engine_luatex_p:
14314   \sys_if_engine_xetex_p:
14315 }
14316 {
14317   \cs_new_protected:Npn \__str_convert_unescape_:
14318   {
14319     \flag_clear:n { str_byte }
14320     \__kernel_tl_gset:Nx \g__str_result_tl
14321     { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }
14322     \__str_if_flag_error:nnx { str_byte } { non-byte } { bytes }
14323   }
14324 }
14325 { \cs_new_protected:Npn \__str_convert_unescape_:
14326 \cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:
```

(End definition for `\__str_convert_unescape_:` and `\__str_convert_unescape_bytes:..`)

`\_\_str\_convert\_escape_:` The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

```
14327 \cs_new_protected:Npn \_\_str\_convert\_escape_ { }
```

```
14328 \cs_new_eq:NN \_\_str\_convert\_escape_bytes: \_\_str\_convert\_escape_:
```

(*End definition for \\_\\_str\\_convert\\_escape\_ and \\_\\_str\\_convert\\_escape\\_bytes::*)

### 54.3.6 Native strings

`\_\_str\_convert\_decode_:` Convert each character to its character code, one at a time.

```
14329 \cs_new_protected:Npn \_\_str\_convert\_decode_:
14330   { \_\_str\_convert_gmap:N \_\_str\_decode_native_char:N }
14331 \cs_new:Npn \_\_str\_decode_native_char:N #1
14332   { #1 \s\_str \int_value:w '#1 \s\_str }
```

(*End definition for \\_\\_str\\_convert\\_decode\_ and \\_\\_str\\_decode\_native\_char:N.*)

`\_\_str\_convert\_encode_:` The conversion from an internal string to native character tokens basically maps `\char_generate:nn` through the code-points, but in non-Unicode-aware engines we use a fall-back character ? rather than nothing when given a character code outside [0, 255]. We detect the presence of bad characters using a flag and only produce a single error after the x-expanding assignment.

```
14333 \bool_lazy_any:nTF
14334   {
14335     \sys_if_engine_luatex_p:
14336     \sys_if_engine_xetex_p:
14337   }
14338   {
14339     \cs_new_protected:Npn \_\_str\_convert\_encode_:
14340       { \_\_str\_convert_gmap_internal:N \_\_str\_encode_native_char:n }
14341     \cs_new:Npn \_\_str\_encode_native_char:n #1
14342       { \char_generate:nn {#1} {12} }
14343   }
14344   {
14345     \cs_new_protected:Npn \_\_str\_convert\_encode_:
14346       {
14347         \flag_clear:n { str_error }
14348         \_\_str\_convert_gmap_internal:N \_\_str\_encode_native_char:n
14349         \_\_str_if_flag_error:nnx { str_error }
14350           { native-overflow } { }
14351       }
14352     \cs_new:Npn \_\_str\_encode_native_char:n #1
14353       {
14354         \if_int_compare:w #1 > \c\_str_max_byte_int
14355           \flag_raise:n { str_error }
14356           ?
14357         \else:
14358           \char_generate:nn {#1} {12}
14359         \fi:
14360       }
14361     \msg_new:nnnn { str } { native-overflow }
14362       { Character~code~too~large~for~this~engine. }
14363   }
```

```

14364     This~engine~only~support~8-bit~characters:~
14365     valid~character~codes~are~in~the~range~[0,255].~
14366     To~manipulate~arbitrary~Unicode,~use~LuaTeX~or~XeTeX.
14367   }
14368 }

(End definition for \__str_convert_encode_ : and \__str_encode_native_char:n.)
```

### 54.3.7 clist

\\_\_str\_convert\_decode\_clist:  
\\_\_str\_decode\_clist\_char:n Convert each integer to the internal form. We first turn \g\_\_str\_result\_t1 into a clist variable, as this avoids problems with leading or trailing commas.

```

14369 \cs_new_protected:Npn \__str_convert_decode_clist:
14370   {
14371     \clist_gset:Nn \g__str_result_t1 \g__str_result_t1
14372     \__kernel_tl_gset:Nx \g__str_result_t1
14373     {
14374       \exp_args:No \clist_map_function:nN
14375         \g__str_result_t1 \__str_decode_clist_char:n
14376     }
14377   }
14378 \cs_new:Npn \__str_decode_clist_char:n #1
14379   { #1 \s__str \int_eval:n {#1} \s__str }

(End definition for \__str_convert_decode_clist: and \__str_decode_clist_char:n.)
```

\\_\_str\_convert\_encode\_clist:  
\\_\_str\_encode\_clist\_char:n Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since \tl\_tail:N does not trigger an error in this case).

```

14380 \cs_new_protected:Npn \__str_convert_encode_clist:
14381   {
14382     \__str_convert_gmap_internal:N \__str_encode_clist_char:n
14383     \__kernel_tl_gset:Nx \g__str_result_t1 { \tl_tail:N \g__str_result_t1 }
14384   }
14385 \cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }

(End definition for \__str_convert_encode_clist: and \__str_encode_clist_char:n.)
```

### 54.3.8 8-bit encodings

It is not clear in what situations 8-bit encodings are used, hence it is not clear what should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings.

The data needed to support a given 8-bit encoding is stored in a file that consists of a single function call

```
\__str_declare_eight_bit_encoding:nnnn {\name} {\modulo} {\mapping} {\missing}
```

This declares the encoding *(name)* to map bytes to Unicode characters according to the *(mapping)*, and map those bytes which are not mentioned in the *(mapping)* either to the replacement character (if they appear in *(missing)*), or to themselves. The *(mapping)* argument is a token list of pairs {\i<byte>} {\i<Unicode>} expressed in uppercase hexadecimal notation. The *(missing)* argument is a token list of {\i<byte>}. Every *<byte>* which does

not appear in the  $\langle mapping \rangle$  nor the  $\langle missing \rangle$  lists maps to itself in Unicode, so for instance the `latin1` encoding has empty  $\langle mapping \rangle$  and  $\langle missing \rangle$  lists. The  $\langle modulo \rangle$  is a (decimal) integer between 256 and 558 inclusive, modulo which all Unicode code points supported by the encodings must be different.

We use two integer arrays per encoding. When decoding we only use the `decode` integer array, with entry  $n + 1$  (offset needed because integer array indices start at 1) equal to the Unicode code point that corresponds to the  $n$ -th byte in the encoding under consideration, or  $-1$  if the given byte is invalid in this encoding. When encoding we use both arrays: upon seeing a code point  $n$ , we look up the entry (1 plus)  $n$  modulo some number  $M$  in the `encode` array, which tells us the byte that might encode the given Unicode code point, then we check in the `decode` array that indeed this byte encodes the Unicode code point we want. Here,  $M$  is an encoding-dependent integer between 256 and 558 (it turns out), chosen so that among the Unicode code points that can be validly represented in the given encoding, no pair of code points have the same value modulo  $M$ .

Loop through both lists of bytes to fill in the `decode` integer array, then fill the `encode` array accordingly. For bytes that are invalid in the given encoding, store  $-1$  in the `decode` array.

```

14386 \cs_new_protected:Npn \__str_declare_eight_bit_encoding:nnnn #1
14387  {
14388      \tl_set:Nn \l__str_internal_tl {\#1}
14389      \cs_new_protected:cpn { __str_convert_decode_#1: }
14390          { \__str_convert_decode_eight_bit:n {\#1} }
14391      \cs_new_protected:cpn { __str_convert_encode_#1: }
14392          { \__str_convert_encode_eight_bit:n {\#1} }
14393      \exp_args:Ncc \__str_declare_eight_bit_aux:NNnnn
14394          { g__str_decode_#1_intarray } { g__str_encode_#1_intarray }
14395  }
14396 \cs_new_protected:Npn \__str_declare_eight_bit_aux:NNnnn #1#2#3#4#5
14397  {
14398      \intarray_new:Nn #1 { 256 }
14399      \int_step_inline:nnn { 0 } { 255 }
14400          { \intarray_gset:Nnn #1 { 1 + ##1 } {##1} }
14401      \__str_declare_eight_bit_loop:Nnn #1
14402          #4 { \s__str_stop \prg_break: } { }
14403      \prg_break_point:
14404      \__str_declare_eight_bit_loop:Nn #1
14405          #5 { \s__str_stop \prg_break: }
14406      \prg_break_point:
14407      \intarray_new:Nn #2 {#3}
14408      \int_step_inline:nnn { 0 } { 255 }
14409      {
14410          \int_compare:nNnF { \intarray_item:Nn #1 { 1 + ##1 } } = { -1 }
14411          {
14412              \intarray_gset:Nnn #2
14413                  {
14414                      1 +
14415                      \int_mod:nn { \intarray_item:Nn #1 { 1 + ##1 } }
14416                          { \intarray_count:N #2 }
14417                  }
14418                  {##1}
14419          }
14420      }

```

```

14421    }
14422 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nnn #1#2#3
14423 {
14424     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14425     \intarray_gset:Nnn #1 { 1 + "#2 } { "#3 }
14426     \__str_declare_eight_bit_loop:Nnn #1
14427 }
14428 \cs_new_protected:Npn \__str_declare_eight_bit_loop:Nn #1#2
14429 {
14430     \__str_use_none_delimit_by_s_stop:w #2 \s__str_stop
14431     \intarray_gset:Nnn #1 { 1 + "#2 } { -1 }
14432     \__str_declare_eight_bit_loop:Nn #1
14433 }

```

(End definition for `\__str_declare_eight_bit_encoding:nnnn` and others.)

```
\__str_convert_decode_eight_bit:n
\__str_decode_eight_bit_aux:n
\__str_decode_eight_bit_aux:Nn
```

The map from bytes to Unicode code points is in the `decode` array corresponding to the given encoding. Define `\__str_tmp:w` and pass it successively all bytes in the string. It produces an internal representation with suitable `\s__str` inserted, and the corresponding code point is obtained by looking it up in the integer array. If the entry is  $-1$  then issue a replacement character and raise the flag indicating that there was an error.

```

14434 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
14435 {
14436     \cs_set:Npx \__str_tmp:w
14437     {
14438         \exp_not:N \__str_decode_eight_bit_aux:Nn
14439         \exp_not:c { g__str_decode_#1_intarray }
14440     }
14441     \flag_clear:n { str_error }
14442     \__str_convert_gmap:N \__str_tmp:w
14443     \__str_if_flag_error:nnx { str_error } { decode-8-bit } { #1 }
14444 }
14445 \cs_new:Npn \__str_decode_eight_bit_aux:Nn #1#2
14446 {
14447     #2 \s__str
14448     \exp_args:Nf \__str_decode_eight_bit_aux:n
14449     { \intarray_item:Nn #1 { 1 + '#2 } }
14450     \s__str
14451 }
14452 \cs_new:Npn \__str_decode_eight_bit_aux:n #1
14453 {
14454     \if_int_compare:w #1 < \c_zero_int
14455         \flag_raise:n { str_error }
14456         \int_value:w \c__str_replacement_char_int
14457     \else:
14458         #1
14459     \fi:
14460 }

```

(End definition for `\__str_convert_decode_eight_bit:n`, `\__str_decode_eight_bit_aux:n`, and `\__str_decode_eight_bit_aux:Nn`.)

```
\__str_convert_encode_eight_bit:n
\__str_encode_eight_bit_aux:nnN
\__str_encode_eight_bit_aux:NNn
```

It is not practical to make an integer array with indices in the full Unicode range, so we work modulo some number, which is simply the size of the `encode` integer array for the

given encoding. This gives us a candidate byte for representing a given Unicode code point. Of course taking the modulo leads to collisions so we check in the `decode` array that the byte we got is indeed correct. Otherwise the Unicode code point we started from is simply not representable in the given encoding.

```

14461 \int_new:N \l__str_modulo_int
14462 \cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
14463 {
14464     \cs_set:Npx \__str_tmp:w
14465     {
14466         \exp_not:N \__str_encode_eight_bit_aux:NNn
14467         \exp_not:c { g__str_encode_#1_intarray }
14468         \exp_not:c { g__str_decode_#1_intarray }
14469     }
14470     \flag_clear:n { str_error }
14471     \__str_convert_gmap_internal:N \__str_tmp:w
14472     \__str_if_flag_error:nnx { str_error } { encode-8-bit } {#1}
14473 }
14474 \cs_new:Npn \__str_encode_eight_bit_aux:NNn #1#2#3
14475 {
14476     \exp_args:Nf \__str_encode_eight_bit_aux:nnN
14477     {
14478         \intarray_item:Nn #1
14479         { 1 + \int_mod:nn {#3} { \intarray_count:N #1 } }
14480     }
14481     {#3}
14482     #2
14483 }
14484 \cs_new:Npn \__str_encode_eight_bit_aux:nnN #1#2#3
14485 {
14486     \int_compare:nNnTF { \intarray_item:Nn #3 { 1 + #1 } } = {#2}
14487     { \__str_output_byte:n {#1} }
14488     { \flag_raise:n { str_error } }
14489 }

```

(End definition for `\__str_convert_encode_eight_bit:n`, `\__str_encode_eight_bit_aux:nnN`, and `\__str_encode_eight_bit_aux:NNn`.)

## 54.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```

14490 \msg_new:nnn { str } { unknown-esc }
14491     { Escaping-scheme-'#1'~(filtered:'#2')~unknown. }
14492 \msg_new:nnn { str } { unknown-enc }
14493     { Encoding-scheme-'#1'~(filtered:'#2')~unknown. }
14494 \msg_new:nnnn { str } { native-escaping }
14495     { The-'native'-encoding-scheme-does-not-support-any-escaping. }
14496     {
14497         Since-native-strings-do-not-consist-in-bytes,~
14498         none-of-the-escaping-methods-make-sense.~
14499         The-specified-escaping,'#1',will be ignored.
14500     }
14501 \msg_new:nnn { str } { file-not-found }

```

```

14502 { File~'l3str-#1.def'~not~found. }

Message used when the “bytes” unescaping fails because the string given to \str-
set_convert:Nnnn contains a non-byte. This cannot happen for the -8-bit engines.
Messages used for other escapings and encodings are defined in each definition file.

14503 \bool_lazy_any:nT
14504 {
14505   \sys_if_engine_luatex_p:
14506   \sys_if_engine_xetex_p:
14507 }
14508 {
14509   \msg_new:nnnn { str } { non-byte }
14510   { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
14511   {
14512     Some~characters~in~the~string~you~asked~to~convert~are~not~
14513     8-bit~characters.~Perhaps~the~string~is~a~native'~Unicode~string?~
14514     If~it~is,~try~using\\
14515     \\
14516     \iow_indent:n
14517     {
14518       \iow_char:N\\str_set_convert:Nnnn \\
14519       \\ \ <str-var>-\{~<string>~\}~\{~native~\}~\{~target~encoding>~\}
14520     }
14521   }
14522 }

```

Those messages are used when converting to and from 8-bit encodings.

```

14523 \msg_new:nnnn { str } { decode-8-bit }
14524   { Invalid~string~in~encoding~'#1'. }
14525   {
14526     LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
14527     any~character~in~the~encoding~'#1'.
14528   }
14529 \msg_new:nnnn { str } { encode-8-bit }
14530   { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
14531   {
14532     The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
14533     LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
14534     string~contains~a~character~that~'#1'~does~not~support.
14535 }

```

## 54.5 Escaping definitions

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);
- **hex** or **hexadecimal**, as per the pdftEX primitive \pdfescapehex
- **name**, as per the pdftEX primitive \pdfescapename
- **string**, as per the pdftEX primitive \pdfescapestring
- **url**, as per the percent encoding of urls.

### 54.5.1 Unescape methods

`\__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte. Anything else than hexadecimal digits is ignored, raising the flag. A string which contains an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending a 0 in all cases, and dropping it if it is alone.

```

14536 \cs_new_protected:Npn \__str_convert_unescape_hex:
14537 {
14538     \group_begin:
14539         \flag_clear:n { str_error }
14540         \int_set:Nn \tex_escapechar:D { 92 }
14541         \__kernel_tl_gset:Nx \g__str_result_tl
14542         {
14543             \__str_output_byte:w "
14544             \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
14545                 { \tl_to_str:N \g__str_result_tl }
14546                 0 { ? 0 - 1 \prg_break: }
14547                 \prg_break_point:
14548             \__str_output_end:
14549         }
14550         \__str_if_flag_error:n { str_error } { unescape-hex } { }
14551     \group_end:
14552 }
14553 \cs_new:Npn \__str_unescape_hex_auxi:N #1
14554 {
14555     \use_none:n #1
14556     \__str_hexadecimal_use:NTF #1
14557         { \__str_unescape_hex_auxii:N }
14558         {
14559             \flag_raise:n { str_error }
14560             \__str_unescape_hex_auxi:N
14561         }
14562     }
14563 \cs_new:Npn \__str_unescape_hex_auxii:N #1
14564 {
14565     \use_none:n #1
14566     \__str_hexadecimal_use:NTF #1
14567         {
14568             \__str_output_end:
14569             \__str_output_byte:w " \__str_unescape_hex_auxi:N
14570         }
14571         {
14572             \flag_raise:n { str_error }
14573             \__str_unescape_hex_auxii:N
14574         }
14575     }
14576 \msg_new:nnnn { str } { unescape-hex }
14577     { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
14578     {
14579         Some~characters~in~the~string~you~asked~to~convert~are~not~
14580         hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
14581     }

```

(End definition for `\__str_convert_unescape_hex:`, `\__str_unescape_hex_auxi:N`, and `\__str_unescape_hex_auxii:N`.)

```
\_\_str\_convert\_unescape\_name:  
\_\_str\_unescape\_name\_loop:wNN  
\_\_str\_convert\_unescape\_url:  
\_\_str\_unescape\_url\_loop:wNN
```

The `\_\_str\_convert\_unescape\_name:` function replaces each occurrence of # followed by two hexadecimal digits in `\g\_\_str\_result\_tl` by the corresponding byte. The `url` function is identical, with escape character % instead of #. Thus we define the two together. The arguments of `\_\_str\_tmp:w` are the character code of # or % in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test `\_\_str\_hexadecimal\_use:NTF` leaves the upper-case digit in the input stream, hence we surround the test with `\_\_str\_output\_byte:w` and `\_\_str\_output\_end:..`. If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed "#1 to `\_\_str\_output\_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```
14582 \cs_set_protected:Npn \_\_str\_tmp:w #1#2#3  
14583 {  
14584     \cs_new_protected:cpn { \_\_str\_convert\_unescape\_#2: }  
14585 {  
14586     \group_begin:  
14587         \flag_clear:n { str\_byte }  
14588         \flag_clear:n { str\_error }  
14589         \int_set:Nn \tex_escapechar:D { 92 }  
14590         \_\_kernel_tl_gset:Nx \g\_\_str\_result\_tl  
14591         {  
14592             \exp_after:wN #3 \g\_\_str\_result\_tl  
14593             #1 ? { ? \prg_break: }  
14594             \prg_break_point:  
14595         }  
14596         \_\_str_if_flag_error:nnx { str\_byte } { non-byte } { #2 }  
14597         \_\_str_if_flag_error:nnx { str\_error } { unescape-#2 } { }  
14598     \group_end:  
14599 }  
14600 \cs_new:Npn #3 ##1#1##2##3  
14601 {  
14602     \_\_str_filter_bytes:n {##1}  
14603     \use_none:n ##3  
14604     \_\_str_output_byte:w "  
14605     \_\_str_hexadecimal_use:NTF ##2  
14606     {  
14607         \_\_str_hexadecimal_use:NTF ##3  
14608         { }  
14609         {  
14610             \flag_raise:n { str_error }  
14611             * 0 + '#1 \use_i:nn  
14612         }  
14613     }  
14614     {  
14615         \flag_raise:n { str_error }  
14616         0 + '#1 \use_i:nn  
14617     }  
14618     \_\_str_output_end:  
14619     \use_i:nnn #3 ##2##3  
14620 }
```

```

14621     \msg_new:nnnn { str } { unescape-#2 }
14622     { String~invalid~in~escaping-'#2'. }
14623     {
14624         LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
14625         two~hexadecimal~digits.~This~is~invalid~in~the~escaping-'#2'.
14626     }
14627 }
14628 \exp_after:wN \__str_tmp:w \c_hash_str { name }
14629     \__str_unescape_name_loop:wNN
14630 \exp_after:wN \__str_tmp:w \c_percent_str { url }
14631     \__str_unescape_url_loop:wNN

```

(End definition for `\__str_convert_unescape_name:` and others.)

```

\__str_convert_unescape_string:
\__str_unescape_string_newlines:wN
\__str_unescape_string_loop:wNNN
\__str_unescape_string_repeat>NNNNN

```

The `string` escaping is somewhat similar to the `name` and `url` escapings, with escape character `\`. The first step is to convert all three line endings, `^J`, `^M`, and `^M^J` to the common `^J`, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

- `\n` Line feed (10)
- `\r` Carriage return (13)
- `\t` Horizontal tab (9)
- `\b` Backspace (8)
- `\f` Form feed (12)
- `\(` Left parenthesis
- `\)` Right parenthesis
- `\\"` Backslash

`\ddd` (backslash followed by 1 to 3 octal digits) Byte `ddd` (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```

14632 \group_begin:
14633   \char_set_catcode_other:N \^^J
14634   \char_set_catcode_other:N \^^M
14635   \cs_set_protected:Npn \__str_tmp:w #1
14636   {
14637     \cs_new_protected:Npn \__str_convert_unescape_string:
14638     {
14639       \group_begin:
14640         \flag_clear:n { str_byte }
14641         \flag_clear:n { str_error }
14642         \int_set:Nn \tex_escapechar:D { 92 }
14643         \__kernel_tl_gset:Nx \g__str_result_tl
14644         {
14645           \exp_after:wN \__str_unescape_string_newlines:wN
14646             \g__str_result_tl \prg_break: ^M ?
14647             \prg_break_point:

```

```

14648 }
14649 \__kernel_tl_gset:Nx \g__str_result_tl
14650 {
14651     \exp_after:wN \__str_unescape_string_loop:wNNN
14652         \g__str_result_tl #1 ?? { ? \prg_break: }
14653         \prg_break_point:
14654     }
14655     \__str_if_flag_error:nnx { str_byte } { non-byte } { string }
14656         \__str_if_flag_error:nnx { str_error } { unescape-string } { }
14657     \group_end:
14658 }
14659 }
14660 \exp_args:No \__str_tmp:w { \c_backslash_str }
14661 \exp_last_unbraced:NNNNo
14662     \cs_new:Npn \__str_unescape_string_loop:wNNN #1 \c_backslash_str #2#3#4
14663 {
14664     \__str_filter_bytes:n {#1}
14665     \use_none:n #4
14666     \__str_output_byte:w ,
14667         \__str_octal_use:NTF #2
14668     {
14669         \__str_octal_use:NTF #3
14670         {
14671             \__str_octal_use:NTF #4
14672             {
14673                 \if_int_compare:w #2 > 3 \exp_stop_f:
14674                     - 256
14675                     \fi:
14676                     \__str_unescape_string_repeat:NNNNNN
14677                 }
14678                 { \__str_unescape_string_repeat:NNNNNN ? }
14679             }
14680             { \__str_unescape_string_repeat:NNNNNN ?? }
14681         }
14682     {
14683         \str_case_e:nnF {#2}
14684     {
14685         { \c_backslash_str } { 134 }
14686         { ( } { 50 }
14687         { ) } { 51 }
14688         { r } { 15 }
14689         { f } { 14 }
14690         { n } { 12 }
14691         { t } { 11 }
14692         { b } { 10 }
14693         { ^J } { 0 - 1 }
14694     }
14695     {
14696         \flag_raise:n { str_error }
14697         0 - 1 \use_i:nn
14698     }
14699 }
14700 \__str_output_end:
14701 \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4

```

```

14702      }
14703  \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
14704    { \__str_output_end: \__str_unescape_string_loop:wNNN }
14705  \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
14706    {
14707      #1
14708      \if_charcode:w ^^J #2 \else: ^^J \fi:
14709      \__str_unescape_string_newlines:wN #2
14710    }
14711  \msg_new:nnnn { str } { unescape-string }
14712  { String~invalid~in~escaping~'string'. }
14713  {
14714    LaTeX~came~across~an~escape~character~'\c_backslash_str'~
14715    not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'('~,~')',~'
14716    '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
14717    of~a~line.
14718  }
14719 \group_end:

```

(End definition for `\__str_convert_unescape_string:` and others.)

### 54.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

`\__str_convert_escape_hex:`

```

\__str_escape_hex_char:N
14720 \cs_new_protected:Npn \__str_convert_escape_hex:
14721  { \__str_convert_gmap:N \__str_escape_hex_char:N }
14722  \cs_new:Npn \__str_escape_hex_char:N #1
14723  { \__str_output_hexadecimal:n { '#1' } }

```

(End definition for `\__str_convert_escape_hex:` and `\__str_escape_hex_char:N`.)

`\__str_convert_escape_name:`

```

\__str_escape_name_char:n
\__str_if_escape_name:nTF
\c__str_escape_name_str
\c__str_escape_name_not_str

```

For each byte, test whether it should be output as is, or be “hash-encoded”. Roughly, bytes outside the range ["2A, "7E] are hash-encoded. We keep two lists of exceptions: characters in `\c__str_escape_name_not_str` are not hash-encoded, and characters in the `\c__str_escape_name_str` are encoded.

```

14724 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$%
14725 \str_const:Nn \c__str_escape_name_str { {} / > [] }
14726 \cs_new_protected:Npn \__str_convert_escape_name:
14727  { \__str_convert_gmap:N \__str_escape_name_char:n }
14728 \cs_new:Npn \__str_escape_name_char:n #1
14729  {
14730    \__str_if_escape_name:nTF { '#1' } { '#1' }
14731    { \c_hash_str \__str_output_hexadecimal:n { '#1' } }
14732  }
14733 \prg_new_conditional:Npnn \__str_if_escape_name:n #1 { TF }
14734  {
14735    \if_int_compare:w '#1 < "2A \exp_stop_f:
14736    \__str_if_contains_char:NnTF \c__str_escape_name_not_str { '#1' }
14737    \prg_return_true: \prg_return_false:
14738  \else:
14739    \if_int_compare:w '#1 > "7E \exp_stop_f:

```

```

14740           \prg_return_false:
14741     \else:
14742       \__str_if_contains_char:NnTF \c__str_escape_name_str {#1}
14743         \prg_return_false: \prg_return_true:
14744     \fi:
14745   \fi:
14746 }

```

(End definition for `\__str_convert_escape_name:` and others.)

Any character below (and including) space, and any character above (and including) `del`, are converted to octal. One backslash is added before each parenthesis and backslash.

```

14747 \str_const:Nx \c__str_escape_string_str
14748   { \c_backslash_str ( ) }
14749 \cs_new_protected:Npn \__str_convert_escape_string:
14750   { \__str_convert_gmap:N \__str_escape_string_char:N }
14751 \cs_new:Npn \__str_escape_string_char:N #1
14752   {
14753     \__str_if_escape_string:NTF #1
14754     {
14755       \__str_if_contains_char:NnT
14756         \c__str_escape_string_str {#1}
14757         { \c_backslash_str }
14758       #1
14759     }
14760   {
14761     \c_backslash_str
14762     \int_div_truncate:nn {'#1} {64}
14763     \int_mod:nn { \int_div_truncate:nn {'#1} { 8 } } { 8 }
14764     \int_mod:nn {'#1} { 8 }
14765   }
14766 }
14767 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
14768   {
14769     \if_int_compare:w '#1 < "27 \exp_stop_f:
14770       \prg_return_false:
14771     \else:
14772       \if_int_compare:w '#1 > "7A \exp_stop_f:
14773         \prg_return_false:
14774       \else:
14775         \prg_return_true:
14776       \fi:
14777     \fi:
14778   }

```

(End definition for `\__str_convert_escape_string:` and others.)

`\__str_convert_escape_url:` This function is similar to `\__str_convert_escape_name:`, escaping different characters.

```

14779 \cs_new_protected:Npn \__str_convert_escape_url:
14780   { \__str_convert_gmap:N \__str_escape_url_char:n }
14781 \cs_new:Npn \__str_escape_url_char:n #1
14782   {
14783     \__str_if_escape_url:nTF {#1} {#1}
14784     { \c_percent_str \__str_output_hexadecimal:n { '#1 } }

```

```

14785      }
14786  \prg_new_conditional:Npnn \__str_if_escape_url:n #1 { TF }
14787  {
14788      \if_int_compare:w '#1 < "41 \exp_stop_f:
14789          \__str_if_contains_char:nnTF { "-.<>" } {#1}
14790          \prg_return_true: \prg_return_false:
14791      \else:
14792          \if_int_compare:w '#1 > "7E \exp_stop_f:
14793              \prg_return_false:
14794          \else:
14795              \__str_if_contains_char:nnTF { [ ] } {#1}
14796                  \prg_return_false: \prg_return_true:
14797          \fi:
14798      \fi:
14799  }

```

(End definition for `\__str_convert_escape_url:`, `\__str_escape_url_char:n`, and `\__str_if_escape_url:nTF`.)

## 54.6 Encoding definitions

The `native` encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the nonexistent ISO 8859-12.

### 54.6.1 utf-8 support

```
\__str_convert_encode_utf8:
    \__str_encode_utf_viii_char:n
\__str_encode_utf_viii_loop:wwnnw
```

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0, 15], which we output shifted by 224. The last range, [65536, 1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `\__str_encode_utf_vii_loop:wwnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in

the ranges [0, 127], [192, 223], [224, 239], and [240, 247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient #1 is less than the limit #3 for that range, output the leading byte (#1 shifted by #4) and stop. Otherwise, we need one more step: use the quotient of #1 by 64, and #1 as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder #2 – 64#1 + 128. The bizarre construction – 1 + 0 \* removes the spurious initial continuation byte (better methods welcome).

```

14800 \cs_new_protected:cpn { __str_convert_encode_utf8: }
14801   { __str_convert_gmap_internal:N __str_encode_utf_viii_char:n }
14802 \cs_new:Npn __str_encode_utf_viii_char:n #1
14803   {
14804     __str_encode_utf_viii_loop:wwnnw #1 ; - 1 + 0 * ;
14805     { 128 } { 0 }
14806     { 32 } { 192 }
14807     { 16 } { 224 }
14808     { 8 } { 240 }
14809     \s__str_stop
14810   }
14811 \cs_new:Npn __str_encode_utf_viii_loop:wwnnw #1; #2; #3#4 #5 \s__str_stop
14812   {
14813     \if_int_compare:w #1 < #3 \exp_stop_f:
14814       __str_output_byte:n { #1 + #4 }
14815       \exp_after:wN __str_use_none_delimit_by_s_stop:w
14816     \fi:
14817     \exp_after:wN __str_encode_utf_viii_loop:wwnnw
14818       \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
14819       #5 \s__str_stop
14820     __str_output_byte:n { #2 - 64 * ( #1 - 2 ) }
14821   }

(End definition for __str_convert_encode_utf8:, __str_encode_utf_viii_char:n, and __str-
encode_utf_viii_loop:wwnnw.)
```

\l\_\_str\_missing\_flag  
\l\_\_str\_extra\_flag  
\l\_\_str\_overlong\_flag  
\l\_\_str\_overflow\_flag

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using \flag\_clear\_new:n rather than \flag\_new:n, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.
- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L<sup>A</sup>T<sub>E</sub>X3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

14822 \flag_clear_new:n { str_missing }
14823 \flag_clear_new:n { str_extra }
14824 \flag_clear_new:n { str_overlong }
14825 \flag_clear_new:n { str_overflow }
14826 \msg_new:nnnn { str } { utf8-decode }
14827 {
14828     Invalid~UTF-8~string:
14829     \exp_last_unbraced:Nf \use_none:n
14830     {
14831         \_\_str_if_flag_times:nT { str_missing } { ,~missing~continuation~byte }
14832         \_\_str_if_flag_times:nT { str_extra } { ,~extra~continuation~byte }
14833         \_\_str_if_flag_times:nT { str_overlong } { ,~overlong~form }
14834         \_\_str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
14835     }
14836     .
14837 }
14838 {
14839     In~the~UTF-8~encoding,~each~Unicode~character~consists~in~
14840     1-to~4~bytes,~with~the~following~bit~pattern: \\\
14841     \iow_indent:n
14842     {
14843         Code~point~\ \ \ \ <~128:~0xxxxxxxx \\\
14844         Code~point~\ \ \ <~2048:~110xxxxx~10xxxxxx \\\
14845         Code~point~\ \ <~65536:~1110xxxx~10xxxxxx~10xxxxxx \\\
14846         Code~point~ <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \\\
14847     }
14848 Bytes~of~the~form~10xxxxxx~are~called~continuation~bytes.
14849 \flag_if_raised:nT { str_missing }
14850 {
14851     \\\\
14852     A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
14853     the~appropriate~number~of~continuation~bytes.
14854 }
14855 \flag_if_raised:nT { str_extra }
14856 {
14857     \\\\
14858     LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
14859 }
14860 \flag_if_raised:nT { str_overlong }
14861 {
14862     \\\\
14863     Every~Unicode~code~point~must~be~expressed~in~the~shortest~
14864     possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
14865     representation~for~the~code~point~3.
14866 }
14867 \flag_if_raised:nT { str_overflow }
14868 {
14869     \\\\
14870     Unicode~limits~code~points~to~the~range~[0,1114111].
14871 }
14872 }
14873 \prop_gput:Nnn \g_msg_module_name_prop { str } { LaTeX }
14874 \prop_gput:Nnn \g_msg_module_type_prop { str } { }

```

(End definition for \l\_\_str\_missing\_flag and others.)

```

\__str_convert_decode_utf8:
    \__str_decode_utf_viii_start:N
\__str_decode_utf_viii_continuation:wNwN
    \__str_decode_utf_viii_aux:wNnwN
    \__str_decode_utf_viii_overflow:w
\__str_decode_utf_viii_end:

```

Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L<sup>A</sup>T<sub>E</sub>X3 error, as explained above). We expect successive multi-byte sequences of the form *<start byte> <continuation bytes>*. The *\_start* auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the *\_continuation* auxiliary. We expect #3 to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to ="C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the *\_start* auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the *\_aux* function.

The *\_aux* function tests whether we should look for more continuation bytes or not. If the number it receives as #1 is less than the maximum #4 for the current range, then we are done: check for an overlong representation by comparing #1 with the maximum #3 for the previous range. Otherwise, we call the *\_continuation* auxiliary again, after shifting the “current code point” by #4 (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the *\_start* auxiliary leaves its first argument in the input stream: the end-marker begins with `\prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the *\_end* auxiliary, which raises the appropriate error flag before ending the mapping.

```

14875 \cs_new_protected:cpn { __str_convert_decode_utf8: }
14876   {
14877     \flag_clear:n { str_error }
14878     \flag_clear:n { str_missing }
14879     \flag_clear:n { str_extra }
14880     \flag_clear:n { str_overlong }
14881     \flag_clear:n { str_overflow }
14882     \__kernel_tl_gset:Nx \g__str_result_tl
14883   {
14884     \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
14885       { \prg_break: \__str_decode_utf_viii_end: }
14886     \prg_break_point:
14887   }
14888   \__str_if_flag_error:nnx { str_error } { utf8-decode } { }

```

```

14889     }
14890 \cs_new:Npn \__str_decode_utf_viii_start:N #1
14891 {
14892     #1
14893     \if_int_compare:w '#1 < "C0 \exp_stop_f:
14894         \s__str
14895         \if_int_compare:w '#1 < "80 \exp_stop_f:
14896             \int_value:w '#1
14897         \else:
14898             \flag_raise:n { str_extra }
14899             \flag_raise:n { str_error }
14900             \int_use:N \c__str_replacement_char_int
14901         \fi:
14902     \else:
14903         \exp_after:wN \__str_decode_utf_viii_continuation:wwN
14904         \int_value:w \int_eval:n { '#1 - "C0 } \exp_after:wN
14905     \fi:
14906     \s__str
14907     \__str_use_none_delimit_by_s_stop:w {"80} {"800} {"10000} {"110000} \s__str_stop
14908     \__str_decode_utf_viii_start:N
14909 }
14910 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
14911     #1 \s__str #2 \__str_decode_utf_viii_start:N #3
14912 {
14913     \use_none:n #3
14914     \if_int_compare:w '#3 <
14915         \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
14916             "C0 \exp_stop_f:
14917             #3
14918             \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
14919             \int_value:w \int_eval:n { #1 * "40 + '#3 - "80 } \exp_after:wN
14920     \else:
14921         \s__str
14922         \flag_raise:n { str_missing }
14923         \flag_raise:n { str_error }
14924         \int_use:N \c__str_replacement_char_int
14925     \fi:
14926     \s__str
14927     #2
14928     \__str_decode_utf_viii_start:N #3
14929 }
14930 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
14931     #1 \s__str #2#3#4 #5 \__str_decode_utf_viii_start:N #6
14932 {
14933     \if_int_compare:w #1 < #4 \exp_stop_f:
14934         \s__str
14935         \if_int_compare:w #1 < #3 \exp_stop_f:
14936             \flag_raise:n { str_overlong }
14937             \flag_raise:n { str_error }
14938             \int_use:N \c__str_replacement_char_int
14939         \else:
14940             #1
14941         \fi:
14942     \else:

```

```

14943   \if_meaning:w \s__str_stop #5
14944     \__str_decode_utf_viii_overflow:w #1
14945   \fi:
14946   \exp_after:wN \__str_decode_utf_viii_continuation:wwN
14947   \int_value:w \int_eval:n { #1 - #4 } \exp_after:wN
14948   \fi:
14949   \s__str
14950   #2 {\#4} #5
14951   \__str_decode_utf_viii_start:N
14952 }
14953 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
14954 {
14955   \fi: \fi:
14956   \flag_raise:n { str_overflow }
14957   \flag_raise:n { str_error }
14958   \int_use:N \c__str_replacement_char_int
14959 }
14960 \cs_new:Npn \__str_decode_utf_viii_end:
14961 {
14962   \s__str
14963   \flag_raise:n { str_missing }
14964   \flag_raise:n { str_error }
14965   \int_use:N \c__str_replacement_char_int \s__str
14966   \prg_break:
14967 }

```

(End definition for `\__str_convert_decode_utf8:` and others.)

### 54.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

14968 \group_begin:
14969   \char_set_catcode_other:N \^^fe
14970   \char_set_catcode_other:N \^^ff

```

When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

- [0, "D7FF]: converted to two bytes;
- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 - "10000/"400.

For the duration of this operation, `\__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to `\__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

14971   \cs_new_protected:cpn { __str_convert_encode_utf16: }
14972   {
14973     \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
14974     \tl_gput_left:Nx \g__str_result_tl { ^~fe ^~ff }
14975   }
14976 \cs_new_protected:cpn { __str_convert_encode_utf16be: }
14977   { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
14978 \cs_new_protected:cpn { __str_convert_encode_utf16le: }
14979   { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
14980 \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
14981   {
14982     \flag_clear:n { str_error }
14983     \cs_set_eq:NN \__str_tmp:w #1
14984     \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
14985     \__str_if_flag_error:n nx { str_error } { utf16-encode } { }
14986   }
14987 \cs_new:Npn \__str_encode_utf_xvi_char:n #1
14988   {
14989     \if_int_compare:w #1 < "D800 \exp_stop_f:
14990       \__str_tmp:w {#1}
14991     \else:
14992       \if_int_compare:w #1 < "10000 \exp_stop_f:
14993         \if_int_compare:w #1 < "E000 \exp_stop_f:
14994           \flag_raise:n { str_error }
14995           \__str_tmp:w { \c__str_replacement_char_int }
14996         \else:
14997           \__str_tmp:w {#1}
14998         \fi:
14999       \else:
15000         \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
15001         \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
15002         \fi:
15003       \fi:
15004     }

```

(End definition for `\__str_convert_encode_utf16:` and others.)

```
\l__str_missing_flag
\l__str_extra_flag
\l__str_end_flag
```

When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800, "DFFF], corresponding to surrogates, cannot be encoded. We use the all-purpose flag `@@error` to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

15005   \flag_clear_new:n { str_missing }
15006   \flag_clear_new:n { str_extra }
15007   \flag_clear_new:n { str_end }
15008   \msg_new:nnnn { str } { utf16-encode }
15009     { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
15010     {
15011       Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
15012       can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
15013       but~not~in~the~UTF-16~encoding.
15014     }
15015   \msg_new:nnnn { str } { utf16-decode }
```

```

15016    {
15017      Invalid~UTF-16~string:
15018      \exp_last_unbraced:Nf \use_none:n
15019      {
15020        \__str_if_flag_times:nT { str_missing } { ,~missing~trail~surrogate }
15021        \__str_if_flag_times:nT { str_extra } { ,~extra~trail~surrogate }
15022        \__str_if_flag_times:nT { str_end } { ,~odd~number~of~bytes }
15023      }
15024      .
15025    }
15026    {
15027      In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
15028      2~or~4~bytes: \\ 
15029      \iow_indent:n
15030      {
15031        Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
15032        Code~point~in~[U+D800,~U+DFFF]:~illegal \\
15033        Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
15034        Code~point~in~[U+10000,~U+10FFFF]:~ 
15035          a~lead~surrogate~and~a~trail~surrogate \\
15036      }
15037      Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
15038      and~trail~surrogates~are~in~the~range~[0xDC00,~0xDFFF].
15039      \flag_if_raised:nT { str_missing }
15040      {
15041        \\\\
15042          A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
15043      }
15044      \flag_if_raised:nT { str_extra }
15045      {
15046        \\\\
15047          LaTeX-came-across-a-trail-surrogate-when-it-was-not-expected.
15048      }
15049      \flag_if_raised:nT { str_end }
15050      {
15051        \\\\
15052          The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~ 
15053          the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
15054      }
15055    }

```

(End definition for `\l__str_missing_flag`, `\l__str_extra_flag`, and `\l__str_end_flag`.)

```

\__str_convert_decode_utf16:
  \__str_convert_decode_utf16be:
  \__str_convert_decode_utf16le:
\__str_decode_utf_xvi_bom:NN
\__str_decode_utf_xvi:Nw

```

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__str_stop`, is expanded once (the string may be long; passing `\g__str_result_t1` as an argument before expansion is cheaper).

The `\__str_decode_utf_xvi:Nw` function defines `\__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `\__str_decode_utf_xvi_pair>NN` described below.

```

15056   \cs_new_protected:cpn { __str_convert_decode_utf16be: }
15057     { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__str_stop }
15058   \cs_new_protected:cpn { __str_convert_decode_utf16le: }
15059     { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__str_stop }
15060   \cs_new_protected:cpn { __str_convert_decode_utf16: }
15061     {
15062       \exp_after:wN \__str_decode_utf_xvi_bom>NN
15063         \g__str_result_tl \s__str_stop \s__str_stop \s__str_stop
15064     }
15065   \cs_new_protected:Npn \__str_decode_utf_xvi_bom>NN #1#2
15066     {
15067       \str_if_eq:mnTF { #1#2 } { ^ff ^fe }
15068         { \__str_decode_utf_xvi:Nw 2 }
15069         {
15070           \str_if_eq:nnTF { #1#2 } { ^fe ^ff }
15071             { \__str_decode_utf_xvi:Nw 1 }
15072             { \__str_decode_utf_xvi:Nw 1 #1#2 }
15073         }
15074     }
15075   \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__str_stop
15076     {
15077       \flag_clear:n { str_error }
15078       \flag_clear:n { str_missing }
15079       \flag_clear:n { str_extra }
15080       \flag_clear:n { str_end }
15081       \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
15082       \__kernel_tl_gset:Nx \g__str_result_tl
15083       {
15084         \exp_after:wN \__str_decode_utf_xvi_pair>NN
15085           #2 \q__str_nil \q__str_nil
15086           \prg_break_point:
15087       }
15088       \__str_if_flag_error:n { str_error } { utf16-decode } { }
15089     }

```

(End definition for `\__str_convert_decode_utf16:` and others.)

`\__str_decode_utf_xvi_pair:NN`  
`\__str_decode_utf_xvi_quad:NNNN`  
`\__str_decode_utf_xvi_pair_end:Nw`

`\__str_decode_utf_xvi_error:nNN`  
`\__str_decode_utf_xvi_extra:NNw`

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 ( $\varepsilon$ -TEX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1, #2, #4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range ["DC, "DF"] (trail surrogate). The test is similar to the test used for continuation bytes

in the UTF-8 decoding functions. In the case where #4#5 is indeed a trail surrogate, leave #1#2#4#5 \s\_\_str ⟨code point⟩ \s\_\_str, and remove the pair #4#5 before looping with \\_\_str\_decode\_utf\_xvi\_pair:NN. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7\*"400 = "D800\*"400+"DC00—"10000.

Every time we read a pair of bytes, we test for the end-marker \q\_\_str\_nil. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

15090   \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
15091   {
15092     \if_meaning:w \q__str_nil #2
15093       \__str_decode_utf_xvi_pair_end:Nw #1
15094     \fi:
15095     \if_case:w
15096       \int_eval:n { ( \__str_tmp:w #1#2 - "D6 ) / 4 } \scan_stop:
15097     \or: \exp_after:wN \__str_decode_utf_xvi_quad>NNwNN
15098     \or: \exp_after:wN \__str_decode_utf_xvi_extra>NNw
15099     \fi:
15100     #1#2 \s__str
15101     \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__str
15102     \__str_decode_utf_xvi_pair:NN
15103   }
15104   \cs_new:Npn \__str_decode_utf_xvi_quad>NNwNN
15105     #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
15106   {
15107     \if_meaning:w \q__str_nil #5
15108       \__str_decode_utf_xvi_error:nNN { missing } #1#2
15109       \__str_decode_utf_xvi_pair_end:Nw #4
15110     \fi:
15111     \if_int_compare:w
15112       \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
15113         0 = 1
15114       \else:
15115         \__str_tmp:w #4#5 < "E0
15116       \fi:
15117       \exp_stop_f:
15118     #1 #2 #4 #5 \s__str
15119     \int_eval:n
15120     {
15121       ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
15122       + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
15123     }
15124     \s__str
15125     \exp_after:wN \use_i:nnn
15126   \else:
15127     \__str_decode_utf_xvi_error:nNN { missing } #1#2
15128   \fi:
15129   \__str_decode_utf_xvi_pair:NN #4#5
15130 }
15131 \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
15132 {
15133   \fi:
15134   \if_meaning:w \q__str_nil #1

```

```

15135     \else:
15136         \_\_str\_decode\_utf\_xvi\_error:nNN { end } #1 \prg\_do\_nothing:
15137     \fi:
15138     \prg\_break:
15139 }
15140 \cs_new:Npn \_\_str\_decode\_utf\_xvi\_extra:NNw #1#2 \s__str #3 \s__str
15141     { \_\_str\_decode\_utf\_xvi\_error:nNN { extra } #1#2 }
15142 \cs_new:Npn \_\_str\_decode\_utf\_xvi\_error:nNN #1#2#3
15143 {
15144     \flag_raise:n { str_error }
15145     \flag_raise:n { str_#1 }
15146     #2 #3 \s__str
15147     \int_use:N \c__str_replacement_char_int \s__str
15148 }

```

(End definition for `\_\_str\_decode\_utf\_xvi\_pair:NN` and others.)

Restore the original catcodes of bytes 254 and 255.

```
15149 \group_end:
```

### 54.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```

15150 \group_begin:
15151     \char_set_catcode_other:N \^00
15152     \char_set_catcode_other:N \^fe
15153     \char_set_catcode_other:N \^ff

```

`\_\_str\_convert\_encode\_utf32:`  
`\_\_str\_convert\_encode\_utf32be:`  
`\_\_str\_convert\_encode\_utf32le:`

Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `\_\_str\_output\_byte:n` instructions are reversed.

```

15154 \cs_new_protected:cpn { __str_convert_encode_utf32: }
15155 {
15156     \_\_str_convert_gmap_internal:N \_\_str_encode_utf_xxxii_be:n
15157     \tl_gput_left:Nx \g__str_result_tl { ^00 ^00 ^fe ^ff }
15158 }
15159 \cs_new_protected:cpn { __str_convert_encode_utf32be: }
15160     { \_\_str_convert_gmap_internal:N \_\_str_encode_utf_xxxii_be:n }
15161 \cs_new_protected:cpn { __str_convert_encode_utf32le: }
15162     { \_\_str_convert_gmap_internal:N \_\_str_encode_utf_xxxii_le:n }
15163 \cs_new:Npn \_\_str_encode_utf_xxxii_be:n #1
15164 {
15165     \exp_args:Nf \_\_str_encode_utf_xxxii_be_aux:nn
15166     { \int_div_truncate:nn {#1} { "100" } {#1} }
15167 }
15168 \cs_new:Npn \_\_str_encode_utf_xxxii_be_aux:nn #1#2
15169 {
15170     ^00
15171     \_\_str_output_byte_pair_be:n {#1}
15172     \_\_str_output_byte:n { #2 - #1 * "100" }
15173 }
15174 \cs_new:Npn \_\_str_encode_utf_xxxii_le:n #1
15175 {

```

```

15176     \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
15177         { \int_div_truncate:nn {#1} { "100" } } {#1}
15178     }
15179 \cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
15180 {
15181     \__str_output_byte:n { #2 - #1 * "100" }
15182     \__str_output_byte_pair_le:n {#1}
15183     ^~00
15184 }

```

(End definition for `\__str_convert_encode_utf32:` and others.)

`str_overflow`  
`str_end`

There can be no error when encoding in UTF-32. When decoding, the string may not have length  $4n$ , or it may contain code points larger than "10FFFF". The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```

15185     \flag_clear_new:n { str_overflow }
15186     \flag_clear_new:n { str_end }
15187     \msg_new:nnnn { str } { utf32-decode }
15188     {
15189         Invalid-UTF-32-string:
15190         \exp_last_unbraced:Nf \use_none:n
15191         {
15192             \__str_if_flag_times:nT { str_overflow } { ,code-point-too-large }
15193             \__str_if_flag_times:nT { str_end } { ,truncated-string }
15194         }
15195     }
15196     {
15197         In-the-UTF-32-encoding, every-Unicode-character-
15198         (in-the-range-[U+0000,~U+10FFFF]) is-encoded-as-4-bytes.
15199         \flag_if_raised:nT { str_overflow }
15200         {
15201             \\\\
15202                 LaTeX-came-across-a-code-point-larger-than-1114111,~
15203                 the-maximum-code-point-defined-by-Unicode.~
15204                 Perhaps-the-string-was-not-encoded-in-the-UTF-32-encoding?
15205             }
15206             \flag_if_raised:nT { str_end }
15207             {
15208                 \\\\
15209                 The-length-of-the-string-is-not-a-multiple-of-4.~
15210                 Perhaps-the-string-was-truncated?
15211             }
15212         }
15213     }

```

(End definition for `str_overflow` and `str_end`. These variables are documented on page ??.)

```

\__str_convert_decode_utf32:
    \__str_convert_decode_utf32e:
    \__str_convert_decode_utf32le:
    \__str_decode_utf_xxxii_bom:NNNN
\__str_decode_utf_xxxii:Nw
    \__str_decode_utf_xxxii_loop:NNNN
    \__str_decode_utf_xxxii_end:w

```

The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `\s_stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `\__str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets

`\__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an `x`-expanding assignment to `\g__str_result_tl`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__str_stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the *(4 bytes)* `\s__str` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__str_stop`. Break the map.

```

15214     \cs_new_protected:cpn { __str_convert_decode_utf32be: }
15215         { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__str_stop }
15216     \cs_new_protected:cpn { __str_convert_decode_utf32le: }
15217         { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__str_stop }
15218     \cs_new_protected:cpn { __str_convert_decode_utf32: }
15219         {
15220             \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
15221                 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
15222         }
15223     \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
15224         {
15225             \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
15226                 { \__str_decode_utf_xxxii:Nw 2 }
15227                 {
15228                     \str_if_eq:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
15229                         { \__str_decode_utf_xxxii:Nw 1 }
15230                         { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
15231                 }
15232         }
15233     \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__str_stop
15234         {
15235             \flag_clear:n { str_overflow }
15236             \flag_clear:n { str_end }
15237             \flag_clear:n { str_error }
15238             \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
15239             \kernel_tl_gset:Nx \g__str_result_tl
15240             {
15241                 \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
15242                     #2 \s__str_stop \s__str_stop \s__str_stop \s__str_stop
15243                     \prg_break_point:
15244             }
15245             \__str_if_flag_error:nnx { str_error } { utf32-decode } { }
15246         }
15247     \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
15248         {
15249             \if_meaning:w \s__str_stop #4
15250                 \exp_after:wN \__str_decode_utf_xxxii_end:w
15251             \fi:
15252             #1#2#3#4 \s__str
15253             \if_int_compare:w \__str_tmp:w #1#4 > \c_zero_int
15254                 \flag_raise:n { str_overflow }
15255                 \flag_raise:n { str_error }
15256                 \int_use:N \c__str_replacement_char_int
15257             \else:

```

```

15258 \if_int_compare:w \__str_tmp:w #2#3 > 16 \exp_stop_f:
15259   \flag_raise:n { str_overflow }
15260   \flag_raise:n { str_error }
15261   \int_use:N \c__str_replacement_char_int
15262 \else:
15263   \int_eval:n
15264     { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
15265 \fi:
15266 \fi:
15267 \s__str
15268   \__str_decode_utf_xxxii_loop:NNNN
15269 }
15270 \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__str_stop
15271 {
15272   \tl_if_empty:nF {#1}
15273   {
15274     \flag_raise:n { str_end }
15275     \flag_raise:n { str_error }
15276     #1 \s__str
15277     \int_use:N \c__str_replacement_char_int \s__str
15278   }
15279 \prg_break:
15280 }

```

(End definition for `\__str_convert_decode_utf32:` and others.)

Restore the original catcodes of bytes 0, 254 and 255.

```
15281 \group_end:
```

## 54.7 PDF names and strings by expansion

```
\str_convert_pdfname:n
\__str_convert_pdfname:n
  \__str_convert_pdfname_bytes:n
  \__str_convert_pdfname_bytes_aux:n
\__str_convert_pdfname_bytes_aux:nm
```

To convert to PDF names by expansion, we work purely on UTF-8 input. The first step is to make a string with “other” spaces, after which we use a simple token-by-token approach. In Unicode engines, we break down everything before one-byte codepoints, but for 8-bit engines there is no need to worry. Actual escaping is covered by the same code as used in the non-expandable route.

```

15282 \cs_new:Npn \str_convert_pdfname:n #1
15283   {
15284     \exp_args:Ne \tl_to_str:n
15285       { \str_map_function:n {#1} \__str_convert_pdfname:n }
15286   }
15287 \bool_lazy_or:nnTF
15288   { \sys_if_engine_luatex_p: }
15289   { \sys_if_engine_xetex_p: }
15290   {
15291     \cs_new:Npn \__str_convert_pdfname:n #1
15292     {
15293       \int_compare:nNnTF { '#1 } > { "7F }
15294         { \__str_convert_pdfname_bytes:n {#1} }
15295         { \__str_escape_name_char:n {#1} }
15296     }
15297   \cs_new:Npn \__str_convert_pdfname_bytes:n #1
15298   {

```

```

15299     \exp_args:Nn \__str_convert_pdfname_bytes_aux:n
15300         { \__kernel_codepoint_to_bytes:n {'#1} }
15301     }
15302     \cs_new:Npn \__str_convert_pdfname_bytes_aux:n #1
15303         { \__str_convert_pdfname_bytes_aux:nnnn #1 }
15304     \cs_new:Npx \__str_convert_pdfname_bytes_aux:nnnn #1#2#3#4
15305         {
15306             \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#1}
15307             \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#2}
15308             \exp_not:N \tl_if_blank:nF {#3}
15309                 {
15310                     \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#3}
15311                     \exp_not:N \tl_if_blank:nF {#4}
15312                         {
15313                             \c_hash_str \exp_not:N \__str_output_hexadecimal:n {#4}
15314                         }
15315                     }
15316                 }
15317             }
15318         { \cs_new_eq:NN \__str_convert_pdfname:n \__str_escape_name_char:n }

(End definition for \str_convert_pdfname:n and others. This function is documented on page 136.)
15319     </package>

```

#### 54.7.1 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```

15320     (*iso88591)
15321     \__str_declare_eight_bit_encoding:nnnn { iso88591 } { 256 }
15322     {
15323     }
15324     {
15325     }
15326     </iso88591>
15327     (*iso88592)
15328     \__str_declare_eight_bit_encoding:nnnn { iso88592 } { 399 }
15329     {
15330         { A1 } { 0104 }
15331         { A2 } { 02D8 }
15332         { A3 } { 0141 }
15333         { A5 } { 013D }
15334         { A6 } { 015A }
15335         { A9 } { 0160 }
15336         { AA } { 015E }
15337         { AB } { 0164 }
15338         { AC } { 0179 }
15339         { AE } { 017D }
15340         { AF } { 017B }
15341         { B1 } { 0105 }
15342         { B2 } { 02DB }
15343         { B3 } { 0142 }

```

```

15344      { B5 } { 013E }
15345      { B6 } { 015B }
15346      { B7 } { 02C7 }
15347      { B9 } { 0161 }
15348      { BA } { 015F }
15349      { BB } { 0165 }
15350      { BC } { 017A }
15351      { BD } { 02DD }
15352      { BE } { 017E }
15353      { BF } { 017C }
15354      { CO } { 0154 }
15355      { C3 } { 0102 }
15356      { C5 } { 0139 }
15357      { C6 } { 0106 }
15358      { C8 } { 010C }
15359      { CA } { 0118 }
15360      { CC } { 011A }
15361      { CF } { 010E }
15362      { DO } { 0110 }
15363      { D1 } { 0143 }
15364      { D2 } { 0147 }
15365      { D5 } { 0150 }
15366      { D8 } { 0158 }
15367      { D9 } { 016E }
15368      { DB } { 0170 }
15369      { DE } { 0162 }
15370      { EO } { 0155 }
15371      { E3 } { 0103 }
15372      { E5 } { 013A }
15373      { E6 } { 0107 }
15374      { E8 } { 010D }
15375      { EA } { 0119 }
15376      { EC } { 011B }
15377      { EF } { 010F }
15378      { FO } { 0111 }
15379      { F1 } { 0144 }
15380      { F2 } { 0148 }
15381      { F5 } { 0151 }
15382      { F8 } { 0159 }
15383      { F9 } { 016F }
15384      { FB } { 0171 }
15385      { FE } { 0163 }
15386      { FF } { 02D9 }
15387      }
15388      {
15389      }
15390  
```

```

15391  
```

```

15392  
```

```

15393  
```

```

15394  
```

```

15395  
```

```

15396  
```

```

15397  
```

```

15398 { AA } { 015E }
15399 { AB } { 011E }
15400 { AC } { 0134 }
15401 { AF } { 017B }
15402 { B1 } { 0127 }
15403 { B6 } { 0125 }
15404 { B9 } { 0131 }
15405 { BA } { 015F }
15406 { BB } { 011F }
15407 { BC } { 0135 }
15408 { BF } { 017C }
15409 { C5 } { 010A }
15410 { C6 } { 0108 }
15411 { D5 } { 0120 }
15412 { D8 } { 011C }
15413 { DD } { 016C }
15414 { DE } { 015C }
15415 { E5 } { 010B }
15416 { E6 } { 0109 }
15417 { F5 } { 0121 }
15418 { F8 } { 011D }
15419 { FD } { 016D }
15420 { FE } { 015D }
15421 { FF } { 02D9 }
15422 }
15423 {
15424 { A5 }
15425 { AE }
15426 { BE }
15427 { C3 }
15428 { DO }
15429 { E3 }
15430 { FO }
15431 }
15432 </iso88593>
15433 <*iso88594>
15434 \__str_declare_eight_bit_encoding:nnnn { iso88594 } { 383 }
15435 {
15436 { A1 } { 0104 }
15437 { A2 } { 0138 }
15438 { A3 } { 0156 }
15439 { A5 } { 0128 }
15440 { A6 } { 013B }
15441 { A9 } { 0160 }
15442 { AA } { 0112 }
15443 { AB } { 0122 }
15444 { AC } { 0166 }
15445 { AE } { 017D }
15446 { B1 } { 0105 }
15447 { B2 } { 02DB }
15448 { B3 } { 0157 }
15449 { B5 } { 0129 }
15450 { B6 } { 013C }
15451 { B7 } { 02C7 }

```

```

15452      { B9 } { 0161 }
15453      { BA } { 0113 }
15454      { BB } { 0123 }
15455      { BC } { 0167 }
15456      { BD } { 014A }
15457      { BE } { 017E }
15458      { BF } { 014B }
15459      { CO } { 0100 }
15460      { C7 } { 012E }
15461      { C8 } { 010C }
15462      { CA } { 0118 }
15463      { CC } { 0116 }
15464      { CF } { 012A }
15465      { DO } { 0110 }
15466      { D1 } { 0145 }
15467      { D2 } { 014C }
15468      { D3 } { 0136 }
15469      { D9 } { 0172 }
15470      { DD } { 0168 }
15471      { DE } { 016A }
15472      { EO } { 0101 }
15473      { E7 } { 012F }
15474      { E8 } { 010D }
15475      { EA } { 0119 }
15476      { EC } { 0117 }
15477      { EF } { 012B }
15478      { FO } { 0111 }
15479      { F1 } { 0146 }
15480      { F2 } { 014D }
15481      { F3 } { 0137 }
15482      { F9 } { 0173 }
15483      { FD } { 0169 }
15484      { FE } { 016B }
15485      { FF } { 02D9 }
15486  }
15487  {
15488  }
15489 </iso88594>
15490 <*iso88595>
15491 \__str_declare_eight_bit_encoding:nnnn { iso88595 } { 374 }
15492  {
15493      { A1 } { 0401 }
15494      { A2 } { 0402 }
15495      { A3 } { 0403 }
15496      { A4 } { 0404 }
15497      { A5 } { 0405 }
15498      { A6 } { 0406 }
15499      { A7 } { 0407 }
15500      { A8 } { 0408 }
15501      { A9 } { 0409 }
15502      { AA } { 040A }
15503      { AB } { 040B }
15504      { AC } { 040C }
15505      { AE } { 040E }

```

15506	{ AF } { 040F }
15507	{ B0 } { 0410 }
15508	{ B1 } { 0411 }
15509	{ B2 } { 0412 }
15510	{ B3 } { 0413 }
15511	{ B4 } { 0414 }
15512	{ B5 } { 0415 }
15513	{ B6 } { 0416 }
15514	{ B7 } { 0417 }
15515	{ B8 } { 0418 }
15516	{ B9 } { 0419 }
15517	{ BA } { 041A }
15518	{ BB } { 041B }
15519	{ BC } { 041C }
15520	{ BD } { 041D }
15521	{ BE } { 041E }
15522	{ BF } { 041F }
15523	{ CO } { 0420 }
15524	{ C1 } { 0421 }
15525	{ C2 } { 0422 }
15526	{ C3 } { 0423 }
15527	{ C4 } { 0424 }
15528	{ C5 } { 0425 }
15529	{ C6 } { 0426 }
15530	{ C7 } { 0427 }
15531	{ C8 } { 0428 }
15532	{ C9 } { 0429 }
15533	{ CA } { 042A }
15534	{ CB } { 042B }
15535	{ CC } { 042C }
15536	{ CD } { 042D }
15537	{ CE } { 042E }
15538	{ CF } { 042F }
15539	{ DO } { 0430 }
15540	{ D1 } { 0431 }
15541	{ D2 } { 0432 }
15542	{ D3 } { 0433 }
15543	{ D4 } { 0434 }
15544	{ D5 } { 0435 }
15545	{ D6 } { 0436 }
15546	{ D7 } { 0437 }
15547	{ D8 } { 0438 }
15548	{ D9 } { 0439 }
15549	{ DA } { 043A }
15550	{ DB } { 043B }
15551	{ DC } { 043C }
15552	{ DD } { 043D }
15553	{ DE } { 043E }
15554	{ DF } { 043F }
15555	{ EO } { 0440 }
15556	{ E1 } { 0441 }
15557	{ E2 } { 0442 }
15558	{ E3 } { 0443 }
15559	{ E4 } { 0444 }

```

15560    { E5 } { 0445 }
15561    { E6 } { 0446 }
15562    { E7 } { 0447 }
15563    { E8 } { 0448 }
15564    { E9 } { 0449 }
15565    { EA } { 044A }
15566    { EB } { 044B }
15567    { EC } { 044C }
15568    { ED } { 044D }
15569    { EE } { 044E }
15570    { EF } { 044F }
15571    { F0 } { 2116 }
15572    { F1 } { 0451 }
15573    { F2 } { 0452 }
15574    { F3 } { 0453 }
15575    { F4 } { 0454 }
15576    { F5 } { 0455 }
15577    { F6 } { 0456 }
15578    { F7 } { 0457 }
15579    { F8 } { 0458 }
15580    { F9 } { 0459 }
15581    { FA } { 045A }
15582    { FB } { 045B }
15583    { FC } { 045C }
15584    { FD } { 00A7 }
15585    { FE } { 045E }
15586    { FF } { 045F }
15587    }
15588    {
15589    }
15590 </iso88595>
15591 <*iso88596>
15592 \__str_declare_eight_bit_encoding:nmmm { iso88596 } { 344 }
15593 {
15594    { AC } { 060C }
15595    { BB } { 061B }
15596    { BF } { 061F }
15597    { C1 } { 0621 }
15598    { C2 } { 0622 }
15599    { C3 } { 0623 }
15600    { C4 } { 0624 }
15601    { C5 } { 0625 }
15602    { C6 } { 0626 }
15603    { C7 } { 0627 }
15604    { C8 } { 0628 }
15605    { C9 } { 0629 }
15606    { CA } { 062A }
15607    { CB } { 062B }
15608    { CC } { 062C }
15609    { CD } { 062D }
15610    { CE } { 062E }
15611    { CF } { 062F }
15612    { DO } { 0630 }
15613    { D1 } { 0631 }

```

```

15614      { D2 } { 0632 }
15615      { D3 } { 0633 }
15616      { D4 } { 0634 }
15617      { D5 } { 0635 }
15618      { D6 } { 0636 }
15619      { D7 } { 0637 }
15620      { D8 } { 0638 }
15621      { D9 } { 0639 }
15622      { DA } { 063A }
15623      { EO } { 0640 }
15624      { E1 } { 0641 }
15625      { E2 } { 0642 }
15626      { E3 } { 0643 }
15627      { E4 } { 0644 }
15628      { E5 } { 0645 }
15629      { E6 } { 0646 }
15630      { E7 } { 0647 }
15631      { E8 } { 0648 }
15632      { E9 } { 0649 }
15633      { EA } { 064A }
15634      { EB } { 064B }
15635      { EC } { 064C }
15636      { ED } { 064D }
15637      { EE } { 064E }
15638      { EF } { 064F }
15639      { F0 } { 0650 }
15640      { F1 } { 0651 }
15641      { F2 } { 0652 }
15642      }
15643      {
15644      { A1 }
15645      { A2 }
15646      { A3 }
15647      { A5 }
15648      { A6 }
15649      { A7 }
15650      { A8 }
15651      { A9 }
15652      { AA }
15653      { AB }
15654      { AE }
15655      { AF }
15656      { BO }
15657      { B1 }
15658      { B2 }
15659      { B3 }
15660      { B4 }
15661      { B5 }
15662      { B6 }
15663      { B7 }
15664      { B8 }
15665      { B9 }
15666      { BA }
15667      { BC }

```

```

15668      { BD }
15669      { BE }
15670      { CO }
15671      { DB }
15672      { DC }
15673      { DD }
15674      { DE }
15675      { DF }
15676      }
15677  </iso88596>
15678 <*iso88597>
15679 \__str_declare_eight_bit_encoding:nnnn { iso88597 } { 498 }
15680  {
15681      { A1 } { 2018 }
15682      { A2 } { 2019 }
15683      { A4 } { 20AC }
15684      { A5 } { 20AF }
15685      { AA } { 037A }
15686      { AF } { 2015 }
15687      { B4 } { 0384 }
15688      { B5 } { 0385 }
15689      { B6 } { 0386 }
15690      { B8 } { 0388 }
15691      { B9 } { 0389 }
15692      { BA } { 038A }
15693      { BC } { 038C }
15694      { BE } { 038E }
15695      { BF } { 038F }
15696      { CO } { 0390 }
15697      { C1 } { 0391 }
15698      { C2 } { 0392 }
15699      { C3 } { 0393 }
15700      { C4 } { 0394 }
15701      { C5 } { 0395 }
15702      { C6 } { 0396 }
15703      { C7 } { 0397 }
15704      { C8 } { 0398 }
15705      { C9 } { 0399 }
15706      { CA } { 039A }
15707      { CB } { 039B }
15708      { CC } { 039C }
15709      { CD } { 039D }
15710      { CE } { 039E }
15711      { CF } { 039F }
15712      { DO } { 03A0 }
15713      { D1 } { 03A1 }
15714      { D3 } { 03A3 }
15715      { D4 } { 03A4 }
15716      { D5 } { 03A5 }
15717      { D6 } { 03A6 }
15718      { D7 } { 03A7 }
15719      { D8 } { 03A8 }
15720      { D9 } { 03A9 }
15721      { DA } { 03AA }

```

```

15722 { DB } { 03AB }
15723 { DC } { 03AC }
15724 { DD } { 03AD }
15725 { DE } { 03AE }
15726 { DF } { 03AF }
15727 { EO } { 03B0 }
15728 { E1 } { 03B1 }
15729 { E2 } { 03B2 }
15730 { E3 } { 03B3 }
15731 { E4 } { 03B4 }
15732 { E5 } { 03B5 }
15733 { E6 } { 03B6 }
15734 { E7 } { 03B7 }
15735 { E8 } { 03B8 }
15736 { E9 } { 03B9 }
15737 { EA } { 03BA }
15738 { EB } { 03BB }
15739 { EC } { 03BC }
15740 { ED } { 03BD }
15741 { EE } { 03BE }
15742 { EF } { 03BF }
15743 { FO } { 03C0 }
15744 { F1 } { 03C1 }
15745 { F2 } { 03C2 }
15746 { F3 } { 03C3 }
15747 { F4 } { 03C4 }
15748 { F5 } { 03C5 }
15749 { F6 } { 03C6 }
15750 { F7 } { 03C7 }
15751 { F8 } { 03C8 }
15752 { F9 } { 03C9 }
15753 { FA } { 03CA }
15754 { FB } { 03CB }
15755 { FC } { 03CC }
15756 { FD } { 03CD }
15757 { FE } { 03CE }
15758 }
15759 {
15760 { AE }
15761 { D2 }
15762 }
15763 </iso88597>
15764 <*iso88598>
15765 \__str_declare_eight_bit_encoding:nmmm { iso88598 } { 308 }
15766 {
15767 { AA } { 00D7 }
15768 { BA } { 00F7 }
15769 { DF } { 2017 }
15770 { EO } { 05D0 }
15771 { E1 } { 05D1 }
15772 { E2 } { 05D2 }
15773 { E3 } { 05D3 }
15774 { E4 } { 05D4 }
15775 { E5 } { 05D5 }

```

```

15776 { E6 } { 05D6 }
15777 { E7 } { 05D7 }
15778 { E8 } { 05D8 }
15779 { E9 } { 05D9 }
15780 { EA } { 05DA }
15781 { EB } { 05DB }
15782 { EC } { 05DC }
15783 { ED } { 05DD }
15784 { EE } { 05DE }
15785 { EF } { 05DF }
15786 { FO } { 05EO }
15787 { F1 } { 05E1 }
15788 { F2 } { 05E2 }
15789 { F3 } { 05E3 }
15790 { F4 } { 05E4 }
15791 { F5 } { 05E5 }
15792 { F6 } { 05E6 }
15793 { F7 } { 05E7 }
15794 { F8 } { 05E8 }
15795 { F9 } { 05E9 }
15796 { FA } { 05EA }
15797 { FD } { 200E }
15798 { FE } { 200F }

15799 }
15800 {
15801 { A1 }
15802 { BF }
15803 { CO }
15804 { C1 }
15805 { C2 }
15806 { C3 }
15807 { C4 }
15808 { C5 }
15809 { C6 }
15810 { C7 }
15811 { C8 }
15812 { C9 }
15813 { CA }
15814 { CB }
15815 { CC }
15816 { CD }
15817 { CE }
15818 { CF }
15819 { DO }
15820 { D1 }
15821 { D2 }
15822 { D3 }
15823 { D4 }
15824 { D5 }
15825 { D6 }
15826 { D7 }
15827 { D8 }
15828 { D9 }
15829 { DA }

```

```

15830      { DB }
15831      { DC }
15832      { DD }
15833      { DE }
15834      { FB }
15835      { FC }
15836      }
15837 </iso88598>
15838 <*iso88599>
15839 \__str_declare_eight_bit_encoding:nmnm { iso88599 } { 352 }
15840 {
15841     { DO } { 011E }
15842     { DD } { 0130 }
15843     { DE } { 015E }
15844     { FO } { 011F }
15845     { FD } { 0131 }
15846     { FE } { 015F }
15847 }
15848 {
15849 }
15850 </iso88599>
15851 <*iso885910>
15852 \__str_declare_eight_bit_encoding:nmnm { iso885910 } { 383 }
15853 {
15854     { A1 } { 0104 }
15855     { A2 } { 0112 }
15856     { A3 } { 0122 }
15857     { A4 } { 012A }
15858     { A5 } { 0128 }
15859     { A6 } { 0136 }
15860     { A8 } { 013B }
15861     { A9 } { 0110 }
15862     { AA } { 0160 }
15863     { AB } { 0166 }
15864     { AC } { 017D }
15865     { AE } { 016A }
15866     { AF } { 014A }
15867     { B1 } { 0105 }
15868     { B2 } { 0113 }
15869     { B3 } { 0123 }
15870     { B4 } { 012B }
15871     { B5 } { 0129 }
15872     { B6 } { 0137 }
15873     { B8 } { 013C }
15874     { B9 } { 0111 }
15875     { BA } { 0161 }
15876     { BB } { 0167 }
15877     { BC } { 017E }
15878     { BD } { 2015 }
15879     { BE } { 016B }
15880     { BF } { 014B }
15881     { CO } { 0100 }
15882     { C7 } { 012E }

```

```

15883 { C8 } { 010C }
15884 { CA } { 0118 }
15885 { CC } { 0116 }
15886 { D1 } { 0145 }
15887 { D2 } { 014C }
15888 { D7 } { 0168 }
15889 { D9 } { 0172 }
15890 { EO } { 0101 }
15891 { E7 } { 012F }
15892 { E8 } { 010D }
15893 { EA } { 0119 }
15894 { EC } { 0117 }
15895 { F1 } { 0146 }
15896 { F2 } { 014D }
15897 { F7 } { 0169 }
15898 { F9 } { 0173 }
15899 { FF } { 0138 }
15900 }
15901 {
15902 }
15903 </iso885910>
15904 <*iso885911>
15905 \__str_declare_eight_bit_encoding:nnnn { iso885911 } { 369 }
15906 {
15907 { A1 } { OE01 }
15908 { A2 } { OE02 }
15909 { A3 } { OE03 }
15910 { A4 } { OE04 }
15911 { A5 } { OE05 }
15912 { A6 } { OE06 }
15913 { A7 } { OE07 }
15914 { A8 } { OE08 }
15915 { A9 } { OE09 }
15916 { AA } { OEOA }
15917 { AB } { OEOB }
15918 { AC } { OEOC }
15919 { AD } { OEOD }
15920 { AE } { OEOE }
15921 { AF } { OEOF }
15922 { B0 } { OE10 }
15923 { B1 } { OE11 }
15924 { B2 } { OE12 }
15925 { B3 } { OE13 }
15926 { B4 } { OE14 }
15927 { B5 } { OE15 }
15928 { B6 } { OE16 }
15929 { B7 } { OE17 }
15930 { B8 } { OE18 }
15931 { B9 } { OE19 }
15932 { BA } { OE1A }
15933 { BB } { OE1B }
15934 { BC } { OE1C }
15935 { BD } { OE1D }
15936 { BE } { OE1E }

```

15937	{ BF } { OE1F }
15938	{ CO } { OE20 }
15939	{ C1 } { OE21 }
15940	{ C2 } { OE22 }
15941	{ C3 } { OE23 }
15942	{ C4 } { OE24 }
15943	{ C5 } { OE25 }
15944	{ C6 } { OE26 }
15945	{ C7 } { OE27 }
15946	{ C8 } { OE28 }
15947	{ C9 } { OE29 }
15948	{ CA } { OE2A }
15949	{ CB } { OE2B }
15950	{ CC } { OE2C }
15951	{ CD } { OE2D }
15952	{ CE } { OE2E }
15953	{ CF } { OE2F }
15954	{ DO } { OE30 }
15955	{ D1 } { OE31 }
15956	{ D2 } { OE32 }
15957	{ D3 } { OE33 }
15958	{ D4 } { OE34 }
15959	{ D5 } { OE35 }
15960	{ D6 } { OE36 }
15961	{ D7 } { OE37 }
15962	{ D8 } { OE38 }
15963	{ D9 } { OE39 }
15964	{ DA } { OE3A }
15965	{ DF } { OE3F }
15966	{ EO } { OE40 }
15967	{ E1 } { OE41 }
15968	{ E2 } { OE42 }
15969	{ E3 } { OE43 }
15970	{ E4 } { OE44 }
15971	{ E5 } { OE45 }
15972	{ E6 } { OE46 }
15973	{ E7 } { OE47 }
15974	{ E8 } { OE48 }
15975	{ E9 } { OE49 }
15976	{ EA } { OE4A }
15977	{ EB } { OE4B }
15978	{ EC } { OE4C }
15979	{ ED } { OE4D }
15980	{ EE } { OE4E }
15981	{ EF } { OE4F }
15982	{ FO } { OE50 }
15983	{ F1 } { OE51 }
15984	{ F2 } { OE52 }
15985	{ F3 } { OE53 }
15986	{ F4 } { OE54 }
15987	{ F5 } { OE55 }
15988	{ F6 } { OE56 }
15989	{ F7 } { OE57 }
15990	{ F8 } { OE58 }

```

15991 { F9 } { 0E59 }
15992 { FA } { 0E5A }
15993 { FB } { 0E5B }
15994 }
15995 {
15996 { DB }
15997 { DC }
15998 { DD }
15999 { DE }
16000 }
16001 </iso885911>
16002 <*iso885913>
16003 \__str_declare_eight_bit_encoding:nnnn { iso885913 } { 399 }
16004 {
16005 { A1 } { 201D }
16006 { A5 } { 201E }
16007 { A8 } { 00D8 }
16008 { AA } { 0156 }
16009 { AF } { 00C6 }
16010 { B4 } { 201C }
16011 { B8 } { 00F8 }
16012 { BA } { 0157 }
16013 { BF } { 00E6 }
16014 { C0 } { 0104 }
16015 { C1 } { 012E }
16016 { C2 } { 0100 }
16017 { C3 } { 0106 }
16018 { C6 } { 0118 }
16019 { C7 } { 0112 }
16020 { C8 } { 010C }
16021 { CA } { 0179 }
16022 { CB } { 0116 }
16023 { CC } { 0122 }
16024 { CD } { 0136 }
16025 { CE } { 012A }
16026 { CF } { 013B }
16027 { DO } { 0160 }
16028 { D1 } { 0143 }
16029 { D2 } { 0145 }
16030 { D4 } { 014C }
16031 { D8 } { 0172 }
16032 { D9 } { 0141 }
16033 { DA } { 015A }
16034 { DB } { 016A }
16035 { DD } { 017B }
16036 { DE } { 017D }
16037 { EO } { 0105 }
16038 { E1 } { 012F }
16039 { E2 } { 0101 }
16040 { E3 } { 0107 }
16041 { E6 } { 0119 }
16042 { E7 } { 0113 }
16043 { E8 } { 010D }
16044 { EA } { 017A }

```

```

16045      { EB } { 0117 }
16046      { EC } { 0123 }
16047      { ED } { 0137 }
16048      { EE } { 012B }
16049      { EF } { 013C }
16050      { FO } { 0161 }
16051      { F1 } { 0144 }
16052      { F2 } { 0146 }
16053      { F4 } { 014D }
16054      { F8 } { 0173 }
16055      { F9 } { 0142 }
16056      { FA } { 015B }
16057      { FB } { 016B }
16058      { FD } { 017C }
16059      { FE } { 017E }
16060      { FF } { 2019 }

16061    }
16062    {
16063    }
16064 〈/iso885913〉

16065 〈*iso885914〉
16066 \__str_declare_eight_bit_encoding:nmnm { iso885914 } { 529 }
16067  {
16068      { A1 } { 1EO2 }
16069      { A2 } { 1EO3 }
16070      { A4 } { 010A }
16071      { A5 } { 010B }
16072      { A6 } { 1EOA }
16073      { A8 } { 1E80 }
16074      { AA } { 1E82 }
16075      { AB } { 1EOB }
16076      { AC } { 1EF2 }
16077      { AF } { 0178 }
16078      { BO } { 1E1E }
16079      { B1 } { 1E1F }
16080      { B2 } { 0120 }
16081      { B3 } { 0121 }
16082      { B4 } { 1E40 }
16083      { B5 } { 1E41 }
16084      { B7 } { 1E56 }
16085      { B8 } { 1E81 }
16086      { B9 } { 1E57 }
16087      { BA } { 1E83 }
16088      { BB } { 1E60 }
16089      { BC } { 1EF3 }
16090      { BD } { 1E84 }
16091      { BE } { 1E85 }
16092      { BF } { 1E61 }
16093      { DO } { 0174 }
16094      { D7 } { 1E6A }
16095      { DE } { 0176 }
16096      { FO } { 0175 }
16097      { F7 } { 1E6B }
16098      { FE } { 0177 }

```

```

16099    }
16100    {
16101    }
16102  </iso885914>
16103  <*iso885915>
16104  \__str_declare_eight_bit_encoding:nnnn { iso885915 } { 383 }
16105  {
16106    { A4 } { 20AC }
16107    { A6 } { 0160 }
16108    { A8 } { 0161 }
16109    { B4 } { 017D }
16110    { B8 } { 017E }
16111    { BC } { 0152 }
16112    { BD } { 0153 }
16113    { BE } { 0178 }
16114  }
16115  {
16116  }
16117  </iso885915>
16118  <*iso885916>
16119  \__str_declare_eight_bit_encoding:nnnn { iso885916 } { 558 }
16120  {
16121    { A1 } { 0104 }
16122    { A2 } { 0105 }
16123    { A3 } { 0141 }
16124    { A4 } { 20AC }
16125    { A5 } { 201E }
16126    { A6 } { 0160 }
16127    { A8 } { 0161 }
16128    { AA } { 0218 }
16129    { AC } { 0179 }
16130    { AE } { 017A }
16131    { AF } { 017B }
16132    { B2 } { 010C }
16133    { B3 } { 0142 }
16134    { B4 } { 017D }
16135    { B5 } { 201D }
16136    { B8 } { 017E }
16137    { B9 } { 010D }
16138    { BA } { 0219 }
16139    { BC } { 0152 }
16140    { BD } { 0153 }
16141    { BE } { 0178 }
16142    { BF } { 017C }
16143    { C3 } { 0102 }
16144    { C5 } { 0106 }
16145    { D0 } { 0110 }
16146    { D1 } { 0143 }
16147    { D5 } { 0150 }
16148    { D7 } { 015A }
16149    { D8 } { 0170 }
16150    { DD } { 0118 }
16151    { DE } { 021A }

```

```
16152      { E3 } { 0103 }
16153      { E5 } { 0107 }
16154      { F0 } { 0111 }
16155      { F1 } { 0144 }
16156      { F5 } { 0151 }
16157      { F7 } { 015B }
16158      { F8 } { 0171 }
16159      { FD } { 0119 }
16160      { FE } { 021B }
16161      }
16162      {
16163      }
16164 </iso885916>
```

# Chapter 55

## l3quark implementation

The following test files are used for this code: *m3quark001.lvt*.

16165 ⟨\*package⟩

### 55.1 Quarks

16166 ⟨@=quark⟩

＼quark\_new:N Allocate a new quark.

```
16167 \cs_new_protected:Npn \quark_new:N #1
16168 {
16169     \__kernel_chk_if_free_cs:N #1
16170     \cs_gset_nopar:Npn #1 {#1}
16171 }
```

(End definition for *\quark\_new:N*. This function is documented on page 139.)

＼q\_nil Some “public” quarks. *\q\_stop* is an “end of argument” marker, *\q\_nil* is a empty value and *\q\_no\_value* marks an empty argument.

```
16172 \quark_new:N \q_nil
16173 \quark_new:N \q_mark
16174 \quark_new:N \q_no_value
16175 \quark_new:N \q_stop
```

(End definition for *\q\_nil* and others. These variables are documented on page 139.)

＼q\_recursion\_tail Quarks for ending recursions. Only ever used there! *\q\_recursion\_tail* is appended to whatever list structure we are doing recursion on, meaning it is added as a proper list item with whatever list separator is in use. *\q\_recursion\_stop* is placed directly after the list.

```
16176 \quark_new:N \q_recursion_tail
16177 \quark_new:N \q_recursion_stop
```

(End definition for *\q\_recursion\_tail* and *\q\_recursion\_stop*. These variables are documented on page 140.)

＼s\_\_quark Private scan mark used in l3quark. We don’t have l3scan yet, so we declare the scan mark here and add it to the scan mark pool later.

```
16178 \cs_new_eq:NN \s__quark \scan_stop:
```

(End definition for `\s_quark`.)

`\q_quark_nil` Private quark use for some tests.

16179   `\quark_new:N \q_quark_nil`

(End definition for `\q_quark_nil`.)

`\quark_if_recursion_tail_stop:N`  
`\quark_if_recursion_tail_stop_do:Nn`

When doing recursions, it is easy to spend a lot of time testing if the end marker has been found. To avoid this, a dedicated end marker is used each time a recursion is set up. Thus if the marker is found everything can be wrapped up and finished off. The simple case is when the test can guarantee that only a single token is being tested. In this case, there is just a dedicated copy of the standard quark test. Both a gobbling version and one inserting end code are provided.

```
16180  \cs_new:Npn \quark_if_recursion_tail_stop:N #1
16181   {
16182     \if_meaning:w \q_recursion_tail #1
16183       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
16184     \fi:
16185   }
16186  \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
16187   {
16188     \if_meaning:w \q_recursion_tail #1
16189       \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
16190     \else:
16191       \exp_after:wN \use_none:n
16192     \fi:
16193 }
```

(End definition for `\quark_if_recursion_tail_stop:N` and `\quark_if_recursion_tail_stop_do:Nn`. These functions are documented on page 140.)

`\quark_if_recursion_tail_stop:`  
`\quark_if_recursion_tail_stop:o`  
`\quark_if_recursion_tail_stop:nn`

`\quark_if_recursion_tail_stop:on`  
`\_quark_if_recursion_tail:w`

See `\quark_if_nil:nTF` for the details. Expanding `\_quark_if_recursion_tail:w` once in front of the tokens chosen here gives an empty result if and only if #1 is exactly `\q_recursion_tail`.

```
16194  \cs_new:Npn \quark_if_recursion_tail_stop:n #1
16195   {
16196     \tl_if_empty:oTF
16197       { \_quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
16198       { \use_none_delimit_by_q_recursion_stop:w }
16199       { }
16200     }
16201  \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
16202   {
16203     \tl_if_empty:oTF
16204       { \_quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
16205       { \use_i_delimit_by_q_recursion_stop:nw }
16206       { \use_none:n }
16207     }
16208  \cs_new:Npn \_quark_if_recursion_tail:w
16209     #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
16210  \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
16211  \cs_generate_variant:Nn \quark_if_recursion_tail_stop_do:nn { o }
```

(End definition for `\quark_if_recursion_tail_stop:n`, `\quark_if_recursion_tail_stop_do:nn`, and `\_quark_if_recursion_tail:w`. These functions are documented on page 140.)

`\quark_if_recursion_tail_break:NN` Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```

16212 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
16213 {
16214     \if_meaning:w \q_recursion_tail #1
16215         \exp_after:wN #2
16216     \fi:
16217 }
16218 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
16219 {
16220     \tl_if_empty:oT
16221     { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
16222     {#2}
16223 }
```

(End definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 141.)

`\quark_if_nil_p:N` Here we test if we found a special quark as the first argument. We better start with `\q_no_value` as the first argument since the whole thing may otherwise loop if #1 is wrongly given a string like `aabc` instead of a single token.<sup>9</sup>

```

16224 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
16225 {
16226     \if_meaning:w \q_nil #1
16227         \prg_return_true:
16228     \else:
16229         \prg_return_false:
16230     \fi:
16231 }
16232 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
16233 {
16234     \if_meaning:w \q_no_value #1
16235         \prg_return_true:
16236     \else:
16237         \prg_return_false:
16238     \fi:
16239 }
16240 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
16241 { c } { p , T , F , TF }
```

(End definition for `\quark_if_nil:NTF` and `\quark_if_no_value:NTF`. These functions are documented on page 139.)

`\quark_if_nil_p:n` Let us explain `\quark_if_nil:nTF`. Expanding `\__quark_if_nil:w` once is safe thanks to the trailing `\q_nil ??!`. The result of expanding once is empty if and only if both delimited arguments #1 and #2 are empty and #3 is delimited by the last tokens `?!`. Thanks to the leading `{}`, the argument #1 is empty if and only if the argument of `\quark_if_nil:n` starts with `\q_nil`. The argument #2 is empty if and only if this `\q_nil` is followed immediately by `?` or by `{?}`, coming either from the trailing tokens in the definition of `\quark_if_nil:n`, or from its argument. In the first case, `\__quark_if_nil:w` is followed by `{?}\q_nil {?}?! \q_nil ??!`, hence #3 is delimited by the final `?!`, and the test returns `true` as wanted. In the second case, the result is not empty since

---

<sup>9</sup>It may still loop in special circumstances however!

the first `?!` in the definition of `\quark_if_nil:n` stop #3. The auxiliary here is the same as `\__tl_if_empty_if:o`, with the same comments applying.

```

16242 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p, T , F , TF }
16243   {
16244     \__quark_if_empty_if:o
16245       { \__quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
16246       \prg_return_true:
16247     \else:
16248       \prg_return_false:
16249     \fi:
16250   }
16251 \cs_new:Npn \__quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
16252 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p, T , F , TF }
16253   {
16254     \__quark_if_empty_if:o
16255       { \__quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
16256       \prg_return_true:
16257     \else:
16258       \prg_return_false:
16259     \fi:
16260   }
16261 \cs_new:Npn \__quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
16262 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
16263   { V , o } { p , TF , T , F }
16264 \cs_new:Npn \__quark_if_empty_if:o #1
16265   {
16266     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
16267       \__kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
16268   }

```

(End definition for `\quark_if_nil:nTF` and others. These functions are documented on page 139.)

`\__kernel_quark_new_test:N` The function `\__kernel_quark_new_test:N` defines #1 in a similar way as `\quark_if_recursion_tail_...` functions (as described below), using `\q_{namespace}_recursion_tail` as the test quark and `\q_{namespace}_recursion_stop` as the delimiter quark, where the `<namespace>` is determined as the first `_`-delimited part in #1.

There are six possible function types which this function can define, and which is defined depends on the signature of the function being defined:

- `:n` gives an analogue of `\quark_if_recursion_tail_stop:n`
- `:nn` gives an analogue of `\quark_if_recursion_tail_stop_do:nn`
- `:nN` gives an analogue of `\quark_if_recursion_tail_break:nN`
- `:N` gives an analogue of `\quark_if_recursion_tail_stop:N`
- `:Nn` gives an analogue of `\quark_if_recursion_tail_stop_do:Nn`
- `:NN` gives an analogue of `\quark_if_recursion_tail_break:NN`

Any other signature causes an error, as does a function without signature.

```
\_\_kernel_quark_new_conditional:Nn
```

Similar to `\_\_kernel_quark_new_test:N`, but defines quark branching conditionals like `\quark_if_nil:nTF` that test for the quark `\q\_\<namespace>\_\<name>`. The `<namespace>` and `<name>` are determined from the conditional #1, which must take the rather rigid form `\_\<namespace>\quark_if_<name>:(arg spec)`. There are only two cases for the `(arg spec)` here:

`:n` gives an analogue of `\quark_if_nil:nTF`

`:N` gives an analogue of `\quark_if_nil:NTF`

Any other signature causes an error, as does a function without signature. We use low-level emptiness tests as `|3tl` is not available yet when these functions are used; thankfully we only care about whether strings are empty so a simple `\if_meaning:w \q_nil <string> \q_nil` suffices.

```
16269 \cs_new_protected:Npn \_\_kernel_quark_new_test:N #1
16270   { \_\_quark_new_test_aux:Nx #1 { \_\_quark_module_name:N #1 } }
16271 \cs_new_protected:Npn \_\_quark_new_test_aux:Nn #1 #2
16272   {
16273     \if_meaning:w \q_nil #2 \q_nil
16274       \msg_error:nnx { quark } { invalid-function }
16275         { \token_to_str:N #1 }
16276     \else:
16277       \_\_quark_new_test:Nccn #1
16278         { q\_\#2_recursion_tail } { q\_\#2_recursion_stop } { __#2 }
16279     \fi:
16280   }
16281 \cs_generate_variant:Nn \_\_quark_new_test_aux:Nn { Nx }
16282 \cs_new_protected:Npn \_\_quark_new_test:NNNn #1
16283   {
16284     \exp_last_unbraced:Nf \_\_quark_new_test_aux:nnNNnnnn
16285       { \cs_split_function:N #1 }
16286       #1 { test }
16287   }
16288 \cs_generate_variant:Nn \_\_quark_new_test:NNNn { Ncc }
16289 \cs_new_protected:Npn \_\_kernel_quark_new_conditional:Nn #1
16290   {
16291     \_\_quark_new_conditional:Nxxn #1
16292       { \_\_quark_quark_conditional_name:N #1 }
16293       { \_\_quark_module_name:N #1 }
16294   }
16295 \cs_new_protected:Npn \_\_quark_new_conditional:Nnnn #1#2#3#4
16296   {
16297     \if_meaning:w \q_nil #2 \q_nil
16298       \msg_error:nnx { quark } { invalid-function }
16299         { \token_to_str:N #1 }
16300     \else:
16301       \if_meaning:w \q_nil #3 \q_nil
16302         \msg_error:nnx { quark } { invalid-function }
16303           { \token_to_str:N #1 }
16304     \else:
16305       \exp_last_unbraced:Nf \_\_quark_new_test_aux:nnNNnnnn
16306         { \cs_split_function:N #1 }
```

```

16307         #1 { conditional }
16308         {#2} {#3} {#4}
16309     \fi:
16310     \fi:
16311   }
16312 \cs_generate_variant:Nn \__quark_new_conditional:Nnnn { Nxx }
16313 \cs_new_protected:Npn \__quark_new_test_aux:nnNNnnn #1 #2 #3 #4 #5
16314   {
16315     \cs_if_exist_use:cTF { __quark_new_#5_#2:Nnnn } { #4 }
16316     {
16317       \msg_error:nxxx { quark } { invalid-function }
16318       { \token_to_str:N #4 } {#2}
16319       \use_none:nnn
16320     }
16321   }

```

(End definition for `\__kernel_quark_new_test:N` and others.)

These macros implement the six possibilities mentioned above, passing the right arguments to `\__quark_new_test_aux_do:nNNnnnnNNn`, which defines some auxiliaries, and then to `\__quark_new_test_define_tl:nNnNNn` (`:n(n)` variants) or to `\__quark_new_test_define_ifx:nNnNNn` (`:N(n)`) which define the main conditionals.

```

16322 \cs_new_protected:Npn \__quark_new_test_n:Nnnn #1 #2 #3 #4
16323   {
16324     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
16325     \__quark_new_test_define_tl:nNnNNn #1 { }
16326   }
16327 \cs_new_protected:Npn \__quark_new_test_nn:Nnnn #1 #2 #3 #4
16328   {
16329     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16330     \__quark_new_test_define_tl:nNnNNn #1 { \use_none:n }
16331   }
16332 \cs_new_protected:Npn \__quark_new_test_nN:Nnnn #1 #2 #3 #4
16333   {
16334     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16335     \__quark_new_test_define_break_tl:nNNNNn #1 { }
16336   }
16337 \cs_new_protected:Npn \__quark_new_test_N:Nnnn #1 #2 #3 #4
16338   {
16339     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { none } { } { } { }
16340     \__quark_new_test_define_ifx:nNnNNn #1 { }
16341   }
16342 \cs_new_protected:Npn \__quark_new_test_Nn:Nnnn #1 #2 #3 #4
16343   {
16344     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16345     \__quark_new_test_define_ifx:nNnNNn #1
16346     { \else: \exp_after:wN \use_none:n }
16347   }
16348 \cs_new_protected:Npn \__quark_new_test_NN:Nnnn #1 #2 #3 #4
16349   {
16350     \__quark_new_test_aux_do:nNNnnnnNNn {#4} #2 #3 { i } { n } {##1} {##2}
16351     \__quark_new_test_define_break_ifx:nNNNNn #1 { }
16352   }

```

(End definition for `\__quark_new_test_n:Nnnn` and others.)

```
\_\_quark_new_test_aux_do:nNNnnnnNNn
\_\_quark_test_define_aux:NNNNnnNNn
```

`\_\_quark_new_test_aux_do:nNNnnnnNNn` makes the control sequence names which will be used by `\_\_quark_test_define_aux:NNNNnnNNn`, and then later by `\_\_quark_new-test_define_t1:nNnNNn` or `\_\_quark_new_test_define_ifx:nNnNNn`. The control sequences defined here are analogous to `\_\_quark_if_recursion_tail:w` and to `\use_(none|i)_delimit_by_q_recursion_stop:(|n)w`.

The name is composed by the name-space and the name of the quarks. Suppose `\_\_kernel_quark_new_test:N` was used with:

```
\_\_kernel_quark_new_test:N \_\_test_quark_tail:n
```

then the first auxiliary will be `\_\_test_quark_recursion_tail:w`, and the second one will be `\_\_test_use_none_delimit_by_q_recursion_stop:w`.

Note that the actual quarks are *not* defined here. They should be defined separately using `\quark_new:N`.

```
16353 \cs_new_protected:Npn \_\_quark_new_test_aux_do:nNNnnnnNNn #1 #2 #3 #4 #5
16354   {
16355     \exp_args:Ncc \_\_quark_test_define_aux:NNNNnnNNn
16356       { #1 _quark_recursion_tail:w }
16357       { #1 _use_ #4 _delimit_by_q_recursion_stop: #5 w }
16358       #2 #3
16359   }
16360 \cs_new_protected:Npn \_\_quark_test_define_aux:NNNNnnNNn #1 #2 #3 #4 #5 #6 #7
16361   {
16362     \cs_gset:Npn #1 ##1 #3 ##2 ? ##3 ?! { ##1 ##2 }
16363     \cs_gset:Npn #2 ##1 #6 #4 {##5}
16364     #7 {##1} #1 #2 #3
16365   }
```

(End definition for `\_\_quark_new_test_aux_do:nNNnnnnNNn` and `\_\_quark_test_define_aux:NNNNnnNNn`.)

Finally, these two macros define the main conditional function using what's been set up before.

```
\_\_quark_new_test_define_t1:nNnNNn
\_\_quark_new_test_define_ifx:nNnNNn
```

```
\_\_quark_new_test_define_break_t1:nNNNNn
\_\_quark_new_test_define_break_ifx:nNNNNn
```

```
16366 \cs_new_protected:Npn \_\_quark_new_test_define_t1:nNnNNn #1 #2 #3 #4 #5 #6
16367   {
16368     \cs_new:Npn #5 #1
16369       {
16370         \tl_if_empty:oTF
16371           { #2 {} ##1 {} ?! #4 ??! }
16372           {##3} {##6}
16373       }
16374   }
16375 \cs_new_protected:Npn \_\_quark_new_test_define_ifx:nNnNNn #1 #2 #3 #4 #5 #6
16376   {
16377     \cs_new:Npn #5 #1
16378       {
16379         \if_meaning:w #4 ##1
16380           \exp_after:wN #3
16381           #6
16382           \fi:
16383       }
16384   }
16385 \cs_new_protected:Npn \_\_quark_new_test_define_break_t1:nNNNNn #1 #2 #3
16386   { \_\_quark_new_test_define_t1:nNnNNn {##1##2} #2 {##2} }
16387 \cs_new_protected:Npn \_\_quark_new_test_define_break_ifx:nNNNNn #1 #2 #3
```

```
16388 { \__quark_new_test_define_ifx:nNnNNn {##1##2} #2 {##2} }
```

(End definition for `\__quark_new_test_define_t1:nNnNNn` and others.)

```
\__quark_new_conditional_n:Nnnn  
\__quark_new_conditional_N:Nnnn
```

These macros implement the two possibilities for branching quark conditionals, passing the right arguments to `\__quark_new_conditional_aux_do:NNnnn`, which defines some auxiliaries and defines the main conditionals.

```
16389 \cs_new_protected:Npn \__quark_new_conditional_n:Nnnn  
16390 { \__quark_new_conditional_aux_do:NNnnn \use_i:nn }  
16391 \cs_new_protected:Npn \__quark_new_conditional_N:Nnnn  
16392 { \__quark_new_conditional_aux_do:NNnnn \use_ii:nn }
```

(End definition for `\__quark_new_conditional_n:Nnnn` and `\__quark_new_conditional_N:Nnnn`.)

Similar to the previous macros, but branching conditionals only require one auxiliary, so we take a shortcut. In `\__quark_new_conditional_define:NNNNn`, #4 is `\use_i:nn` to define the n-type function (which needs an auxiliary) and is `\use_ii:nn` to define the N-type function.

```
16393 \cs_new_protected:Npn \__quark_new_conditional_aux_do:NNnnn #1 #2 #3 #4  
16394 {  
16395     \exp_args:Ncc \__quark_new_conditional_define:NNNNn  
16396     { __ #4 _if_quark_ #3 :w } { q__ #4 _ #3 } #2 #1  
16397 }  
16398 \cs_new_protected:Npn \__quark_new_conditional_define:NNNNn #1 #2 #3 #4 #5  
16399 {  
16400     #4 { \cs_gset:Npn #1 ##1 #2 ##2 ? ##3 ?! { ##1 ##2 } } { }  
16401     \exp_args:Nno \use:n { \prg_new_conditional:Npnn #3 ##1 {#5} }  
16402     {  
16403         #4 { \__quark_if_empty_if:o { #1 {} ##1 {} ?! #2 ??! } }  
16404         { \if_meaning:w #2 ##1 }  
16405         \prg_return_true: \else: \prg_return_false: \fi:  
16406     }  
16407 }
```

(End definition for `\__quark_new_conditional_aux_do:NNnnn` and `\__quark_new_conditional_define:NNNNn`.)

`\__quark_module_name:N` takes a control sequence and returns its *<module>* name, determined as the first non-empty non-single-character word, separated by `_` or `:`. These rules give the correct result for public functions `\<module>_...`, private functions `\__-<module>_...`, and variables such as `\l_<module>_...`. If no valid module is found the result is an empty string. The approach is to first cut off everything after the (first) `:` if any is present, then repeatedly grab `_`-delimited words until finding one of length at least 2 (we use low-level tests as `\l3tl` is not fully available when `\__kernel_quark_new-test:N` is first used. If no *<module>* is found (such as in `\:::n`) we get the trailing marker `\use_none:n {}`, which expands to nothing.

```
16408 \cs_set:Npn \__quark_tmp:w #1#2  
16409 {  
16410     \cs_new:Npn \__quark_module_name:N ##1  
16411     {  
16412         \exp_last_unbraced:Nf \__quark_module_name:w  
16413         { \cs_to_str:N ##1 } #1 \s__quark  
16414     }  
16415     \cs_new:Npn \__quark_module_name:w ##1 #1 ##2 \s__quark  
16416     { \__quark_module_name_loop:w ##1 #2 \use_none:n {} #2 \s__quark }
```

```

16417 \cs_new:Npn \__quark_module_name_loop:w ##1 #2
16418 {
16419     \use_i_if:nnn \if_meaning:w \prg_do_nothing:
16420         ##1 \prg_do_nothing: \prg_do_nothing:
16421         \exp_after:wN \__quark_module_name_loop:w
16422     \else:
16423         \__quark_module_name_end:w ##1
16424     \fi:
16425 }
16426 \cs_new:Npn \__quark_module_name_end:w
16427     ##1 \fi: ##2 \s__quark { \fi: ##1 }
16428 }
16429 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _ }
```

(End definition for `\__quark_module_name:N` and others.)

`\__quark_quark_conditional_name:N`  
`\__quark_quark_conditional_name:w`

`\__quark_quark_conditional_name:N` determines the quark name that the quark conditional function `##1` queries, as the part of the function name between `_quark_if_` and the trailing `::`. Again we define it through `\__quark_tmp:w`, which receives `:` as `#1` and `_quark_if_` as `#2`. The auxiliary `\__quark_quark_conditional_name:w` returns the part between the first `_quark_if_` and the next `:`, and we apply this auxiliary to the function name followed by `:` (in case the function name is lacking a signature), and `_quark_if_:` so that `\__quark_quark_conditional_name:N` returns an empty string if `_quark_if_` is not present.

```

16430 \cs_set:Npn \__quark_tmp:w #1 #2 \s__quark
16431 {
16432     \cs_new:Npn \__quark_quark_conditional_name:N ##1
16433     {
16434         \exp_last_unbraced:Nf \__quark_quark_conditional_name:w
16435             { \cs_to_str:N ##1 } #1 #2 #1 \s__quark
16436     }
16437     \cs_new:Npn \__quark_quark_conditional_name:w
16438         ##1 #2 ##2 #1 ##3 \s__quark {##2}
16439 }
16440 \exp_after:wN \__quark_tmp:w \tl_to_str:n { : _quark_if_ } \s__quark
```

(End definition for `\__quark_quark_conditional_name:N` and `\__quark_quark_conditional_name:w`.)

## 55.2 Scan marks

16441 `<@@=scan>`

`\scan_new:N` Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```

16442 \cs_new_protected:Npn \scan_new:N #1
16443 {
16444     \tl_if_in:NnTF \g__scan_marks_tl { #1 }
16445     {
16446         \msg_error:nnx { scanmark } { already-defined }
16447             { \token_to_str:N #1 }
16448     }
16449     {
16450         \tl_gput_right:Nn \g__scan_marks_tl {#1}
```

```

16451           \cs_new_eq:NN #1 \scan_stop:
16452       }
16453   }

```

(End definition for `\scan_new:N`. This function is documented on page 142.)

### `\s_stop` `\g__scan_marks_tl`

We only declare one scan mark here, more can be defined by specific modules. Can't use `\scan_new:N` yet because `\I3tl` isn't loaded, so define `\s_stop` by hand and add it to `\g__scan_marks_tl`. We also add the scan marks declared earlier to the pool here. Since they lives in a different namespace, a little `\I3docstrip` cheating is necessary.

```

16454 \cs_new_eq:NN \s_stop \scan_stop:
16455 \cs_gset_nopar:Npn \g__scan_marks_tl
16456 {
16457     \s_stop
16458     ⟨@=quark⟩
16459     \s__quark
16460     ⟨@=cs⟩
16461     \s__cs_mark
16462     \s__cs_stop
16463     ⟨@=scan⟩
16464 }

```

(End definition for `\s_stop` and `\g__scan_marks_tl`. This variable is documented on page 142.)

### `\use_none_delimit_by_s_stop:w`

Similar to `\use_none_delimit_by_q_stop:w`.

```
16465 \cs_new:Npn \use_none_delimit_by_s_stop:w #1 \s_stop { }
```

(End definition for `\use_none_delimit_by_s_stop:w`. This function is documented on page 142.)

```
16466 ⟨/package⟩
```

# Chapter 56

## I3seq implementation

The following test files are used for this code: *m3seq002*, *m3seq003*.

16467 `<*package>`

16468 `<@@=seq>`

A sequence is a control sequence whose top-level expansion is of the form “`\s__-seq \__seq_item:n {\langle item_1 \rangle} ... \__seq_item:n {\langle item_n \rangle}`”, with a leading scan mark followed by *n* items of the same form. An earlier implementation used the structure “`\seq_elt:w \langle item_1 \rangle \seq_elt_end: ... \seq_elt:w \langle item_n \rangle \seq_elt_end:`”. This allowed rapid searching using a delimited function, but was not suitable for items containing {, } and # tokens, and also lead to the loss of surrounding braces around items

---

`\__seq_item:n *` `\__seq_item:n {\langle item \rangle}`

The internal token used to begin each sequence entry. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

---

`\__seq_push_item_def:n` `\__seq_push_item_def:n {\langle code \rangle}`

`\__seq_push_item_def:x` Saves the definition of `\__seq_item:n` and redefines it to accept one parameter and expand to `\langle code \rangle`. This function should always be balanced by use of `\__seq_pop_item_def:`.

---

`\__seq_pop_item_def:` `\__seq_pop_item_def:`

Restores the definition of `\__seq_item:n` most recently saved by `\__seq_push_item_def:n`. This function should always be used in a balanced pair with `\__seq_push_item_def:n`.

`\s__seq` This private scan mark.

16469 `\scan_new:N \s__seq`

(End definition for `\s__seq`.)

`\s__seq_mark` Private scan marks.

`\s__seq_stop` 16470 `\scan_new:N \s__seq_mark`

16471 `\scan_new:N \s__seq_stop`

(End definition for `\s__seq_mark` and `\s__seq_stop`.)

`\_seq_item:n` The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.

```
16472 \cs_new:Npn \_seq_item:n
16473 {
16474     \msg_expandable_error:nn { seq } { misused }
16475     \use_none:n
16476 }
```

(End definition for `\_seq_item:n`.)

`\l_seq_internal_a_tl` Scratch space for various internal uses.

`\l_seq_internal_b_tl`   
 16477 `\tl_new:N \l_seq_internal_a_tl`  
 16478 `\tl_new:N \l_seq_internal_b_tl`

(End definition for `\l_seq_internal_a_tl` and `\l_seq_internal_b_tl`.)

`\_seq_tmp:w` Scratch function for internal use.

```
16479 \cs_new_eq:NN \_seq_tmp:w ?
```

(End definition for `\_seq_tmp:w`.)

`\c_empty_seq` A sequence with no item, following the structure mentioned above.

```
16480 \tl_const:Nn \c_empty_seq { \s_seq }
```

(End definition for `\c_empty_seq`. This variable is documented on page 154.)

## 56.1 Allocation and initialisation

`\seq_new:N` Sequences are initialized to `\c_empty_seq`.

`\seq_new:c`   
 16481 `\cs_new_protected:Npn \seq_new:N #1`  
 16482 {  
 16483 `\_kernel_chk_if_free_cs:N #1`  
 16484 `\cs_gset_eq:NN #1 \c_empty_seq`  
 16485 }  
 16486 `\cs_generate_variant:Nn \seq_new:N { c }`

(End definition for `\seq_new:N`. This function is documented on page 143.)

`\seq_clear:N` Clearing a sequence is similar to setting it equal to the empty one.

`\seq_clear:c`   
 16487 `\cs_new_protected:Npn \seq_clear:N #1`  
 16488 { `\seq_set_eq:NN #1 \c_empty_seq` }  
`\seq_gclear:N`   
 16489 `\cs_generate_variant:Nn \seq_clear:N { c }`  
 16490 `\cs_new_protected:Npn \seq_gclear:N #1`  
 16491 { `\seq_gset_eq:NN #1 \c_empty_seq` }  
 16492 `\cs_generate_variant:Nn \seq_gclear:N { c }`

(End definition for `\seq_clear:N` and `\seq_gclear:N`. These functions are documented on page 143.)

`\seq_clear_new:N` Once again we copy code from the token list functions.

`\seq_clear_new:c`   
 16493 `\cs_new_protected:Npn \seq_clear_new:N #1`  
 16494 { `\seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 }`  }  
`\seq_gclear_new:N`   
 16495 `\cs_generate_variant:Nn \seq_clear_new:N { c }`  
 16496 `\cs_new_protected:Npn \seq_gclear_new:N #1`  
 16497 { `\seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 }`  }  
 16498 `\cs_generate_variant:Nn \seq_gclear_new:N { c }`

(End definition for `\seq_clear_new:N` and `\seq_gclear_new:N`. These functions are documented on page 143.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.

```
16499 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
16500 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
16501 \cs_new_eq:NN \seq_set_eq:cN \tl_set_eq:cN
16502 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
16503 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
16504 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
16505 \cs_new_eq:NN \seq_gset_eq:cN \tl_gset_eq:cN
16506 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
```

(End definition for `\seq_set_eq:NN` and `\seq_gset_eq:NN`. These functions are documented on page 143.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.

```
16507 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
16508 {
16509     \__kernel_tl_set:Nx #1
16510     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
16511 }
16512 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
16513 {
16514     \__kernel_tl_set:Nx #1
16515     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
16516 }
16517 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
16518 {
16519     \__kernel_tl_gset:Nx #1
16520     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
16521 }
16522 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
16523 {
16524     \__kernel_tl_gset:Nx #1
16525     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
16526 }
16527 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
16528 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
16529 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c }
16530 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
16531 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
16532 \cs_generate_variant:Nn \seq_gset_from_clist:Nn { c }
```

(End definition for `\seq_set_from_clist:NN` and others. These functions are documented on page 144.)

`\seq_const_from_clist:Nn` Almost identical to `\seq_set_from_clist:Nn`.

```
16533 \cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
16534 {
16535     \tl_const:Nx #1
16536     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
16537 }
16538 \cs_generate_variant:Nn \seq_const_from_clist:Nn { c }
```

(End definition for `\seq_const_from_clist:Nn`. This function is documented on page 144.)

```
\seq_set_split:Nnn  
\seq_set_split:NnV  
\seq_gset_split:Nnn  
\seq_gset_split:NnV  
\seq_set_split_keep_spaces:Nnn  
\seq_set_split_keep_spaces:NnV  
\seq_gset_split_keep_spaces:Nnn  
\seq_gset_split_keep_spaces:NnV  
\_seq_set_split:NNnn  
  \_\_seq_set_split:Nw  
    \_\_seq_set_split:w  
\_\_seq_set_split_end:
```

When the separator is empty, everything is very simple, just map `\_seq_wrap_item:` through the items of the last argument. For non-trivial separators, the goal is to split a given token list at the marker, strip spaces from each item, and remove one set of outer braces if after removing leading and trailing spaces the item is enclosed within braces. After `\tl_replace_all:Nnn`, the token list `\l__seq_internal_a_t1` is a repetition of the pattern `\_seq_set_split_auxi:w \prg_do_nothing: <item with spaces> \_seq_set_split_end:`. Then, x-expansion causes `\_seq_set_split_auxi:w` to trim spaces, and leaves its result as `\_seq_set_split_auxii:w <trimmed item> \_seq_set_split_end:`. This is then converted to the `\l3seq` internal structure by another x-expansion. In the first step, we insert `\prg_do_nothing:` to avoid losing braces too early: that would cause space trimming to act within those lost braces. The second step is solely there to strip braces which are outermost after space trimming.

```

16539 \cs_new_protected:Npn \seq_set_split:Nnn
16540   { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \tl_trim_spaces:n }
16541 \cs_new_protected:Npn \seq_gset_split:Nnn
16542   { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \tl_trim_spaces:n }
16543 \cs_new_protected:Npn \seq_set_split_keep_spaces:Nnn
16544   { \__seq_set_split:NNNnn \__kernel_tl_set:Nx \exp_not:n }
16545 \cs_new_protected:Npn \seq_gset_split_keep_spaces:Nnn
16546   { \__seq_set_split:NNNnn \__kernel_tl_gset:Nx \exp_not:n }
16547 \cs_new_protected:Npn \__seq_set_split:NNNnn #1#2#3#4#5
16548 {
16549   \tl_if_empty:nTF {#4}
16550   {
16551     \tl_set:Nn \l__seq_internal_a_tl
16552       { \tl_map_function:nN {#5} \__seq_wrap_item:n }
16553   }
16554   {
16555     \tl_set:Nn \l__seq_internal_a_tl
16556       {
16557         \__seq_set_split:Nw #2 \prg_do_nothing:
16558         #5
16559         \__seq_set_split_end:
16560       }
16561     \tl_replace_all:Nnn \l__seq_internal_a_tl {#4}
16562   }
16563   \__seq_set_split_end:
16564   \__seq_set_split:Nw #2 \prg_do_nothing:
16565   }
16566   \__kernel_tl_set:Nx \l__seq_internal_a_tl { \l__seq_internal_a_tl }
16567   }
16568 #1 #3 { \s__seq \l__seq_internal_a_tl }
16569 }
16570 \cs_new:Npn \__seq_set_split:Nw #1#2 \__seq_set_split_end:
16571 {
16572   \exp_not:N \__seq_set_split:w
16573   \exp_args:No #1 {#2}
16574   \exp_not:N \__seq_set_split_end:
16575 }
16576 \cs_new:Npn \__seq_set_split:w #1 \__seq_set_split_end:
16577   { \__seq_wrap_item:n {#1} }
16578 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
16579 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

```

16580 \cs_generate_variant:Nn \seq_set_split_keep_spaces:Nnn { NnV }
16581 \cs_generate_variant:Nn \seq_gset_split_keep_spaces:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 144.)

`\seq_concat:NNN` When concatenating sequences, one must remove the leading `\s__seq` of the second sequence. The result starts with `\s__seq` (of the first sequence), which stops f-expansion.

```

16582 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
16583 { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
16584 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
16585 { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
16586 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
16587 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 144.)

`\seq_if_exist_p:N` Copies of the `cs` functions defined in `l3basics`.

```

16588 \prg_new_eq_conditional>NNn \seq_if_exist:N \cs_if_exist:N
16589 { TF , T , F , p }
16590 \prg_new_eq_conditional>NNn \seq_if_exist:c \cs_if_exist:c
16591 { TF , T , F , p }

```

(End definition for `\seq_if_exist:NTF`. This function is documented on page 145.)

## 56.2 Appending data to either end

`\seq_put_left:Nn` When adding to the left of a sequence, remove `\s__seq`. This is done by `\__seq_put_left_aux:w`, which also stops f-expansion.

```

16592 \cs_new_protected:Npn \seq_put_left:Nn #1#2
16593 {
16594     \__kernel_tl_set:Nx #1
16595     {
16596         \exp_not:n { \s__seq \__seq_item:n {#2} }
16597         \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
16598     }
16599 }
16600 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
16601 {
16602     \__kernel_tl_gset:Nx #1
16603     {
16604         \exp_not:n { \s__seq \__seq_item:n {#2} }
16605         \exp_not:f { \exp_after:wN \__seq_put_left_aux:w #1 }
16606     }
16607 }
16608 \cs_new:Npn \__seq_put_left_aux:w \s__seq { \exp_stop_f: }
16609 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
16610 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
16611 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
16612 \cs_generate_variant:Nn \seq_gput_left:Nn { c , cV , cv , co , cx }

```

(End definition for `\seq_put_left:Nn`, `\seq_gput_left:Nn`, and `\__seq_put_left_aux:w`. These functions are documented on page 145.)

`\seq_put_right:Nn` Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `\_seq_item:n`.

```

16613 \cs_new_protected:Npn \seq_put_right:Nn #1#2
16614   { \tl_put_right:Nn #1 { \_seq_item:n {#2} } }
16615   \cs_new_protected:Npn \seq_gput_right:Nn #1#2
16616   { \tl_gput_right:Nn #1 { \_seq_item:n {#2} } }
16617 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
16618 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cv , cv , co , cx }
16619 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
16620 \cs_generate_variant:Nn \seq_put_right:Nn { c , cv , cv , co , cx }
```

(End definition for `\seq_put_right:Nn` and `\seq_gput_right:Nn`. These functions are documented on page 145.)

### 56.3 Modifying sequences

This function converts its argument to a proper sequence item in an x-expansion context.

```
16621 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \_seq_item:n {#1} } }
```

(End definition for `\__seq_wrap_item:n`.)

An internal sequence for the removal routines.

```
16622 \seq_new:N \l__seq_remove_seq
```

(End definition for `\l__seq_remove_seq`.)

`\seq_remove_duplicates:N` Removing duplicates means making a new list then copying it.

```

16623 \cs_new_protected:Npn \seq_remove_duplicates:N
16624   { \__seq_remove_duplicates>NN \seq_set_eq:NN }
16625 \cs_new_protected:Npn \seq_gremove_duplicates:N
16626   { \__seq_remove_duplicates>NN \seq_gset_eq:NN }
16627 \cs_new_protected:Npn \__seq_remove_duplicates>NN #1#2
16628   {
16629     \seq_clear:N \l__seq_remove_seq
16630     \seq_map_inline:Nn #2
16631     {
16632       \seq_if_in:NnF \l__seq_remove_seq {##1}
16633         { \seq_put_right:Nn \l__seq_remove_seq {##1} }
16634     }
16635     #1 #2 \l__seq_remove_seq
16636   }
16637 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
16638 \cs_generate_variant:Nn \seq_gremove_duplicates:N { c }
```

(End definition for `\seq_remove_duplicates:N`, `\seq_gremove_duplicates:N`, and `\__seq_remove_duplicates>NN`. These functions are documented on page 148.)

`\seq_remove_all:Nn` The idea of the code here is to avoid a relatively expensive addition of items one at a time to an intermediate sequence. The approach taken is therefore similar to that in `\__seq_pop_right:NNN`, using a “flexible” x-type expansion to do most of the work. As `\tl_if_eq:nnT` is not expandable, a two-part strategy is needed. First, the x-type expansion uses `\str_if_eq:nnT` to find potential matches. If one is found, the expansion is halted and the necessary set up takes place to use the `\tl_if_eq:NNT` test. The x-type is started

again, including all of the items copied already. This happens repeatedly until the entire sequence has been scanned. The code is set up to avoid needing an intermediate scratch list: the lead-off x-type expansion (#1 #2 {#2}) ensures that nothing is lost.

```

16639 \cs_new_protected:Npn \seq_remove_all:Nn
16640   { \__seq_remove_all_aux:NNn \__kernel_tl_set:Nx }
16641 \cs_new_protected:Npn \seq_gremove_all:Nn
16642   { \__seq_remove_all_aux:NNn \__kernel_tl_gset:Nx }
16643 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
16644   {
16645     \__seq_push_item_def:n
16646     {
16647       \str_if_eq:nnT {##1} {#3}
16648       {
16649         \if_false: { \fi: }
16650         \tl_set:Nn \l__seq_internal_b_tl {##1}
16651         #1 #2
16652         { \if_false: } \fi:
16653         \exp_not:o {#2}
16654         \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
16655         { \use_none:nn }
16656       }
16657       \__seq_wrap_item:n {##1}
16658     }
16659     \tl_set:Nn \l__seq_internal_a_tl {#3}
16660     #1 #2 {#2}
16661     \__seq_pop_item_def:
16662   }
16663 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
16664 \cs_generate_variant:Nn \seq_gremove_all:Nn { c }

(End definition for \seq_remove_all:Nn, \seq_gremove_all:Nn, and \__seq_remove_all_aux:NNn.
These functions are documented on page 148.)
```

\seq\_reverse:N

\seq\_reverse:c

\seq\_greverse:N

\seq\_greverse:c

\\_\_seq\_reverse>NN

\\_\_seq\_reverse\_item:nwn

Previously, \seq\_reverse:N was coded by collecting the items in reverse order after an \exp\_stop\_f: marker.

```

\cs_new_protected:Npn \seq_reverse:N #1
  {
    \cs_set_eq:NN \@@_item:n \@@_reverse_item:nw
    \tl_set:Nf #2 { #2 \exp_stop_f: }
  }
\cs_new:Npn \@@_reverse_item:nw #1 #2 \exp_stop_f:
  {
    #2 \exp_stop_f:
    \@@_item:n {#1}
  }
```

At first, this seems optimal, since we can forget about each item as soon as it is placed after \exp\_stop\_f:. Unfortunately, TeX's usual tail recursion does not take place in this case: since the following \\_\_seq\_reverse\_item:nw only reads tokens until \exp\_stop\_f:, and never reads the \@@\_item:n {#1} left by the previous call, TeX cannot remove that previous call from the stack, and in particular must retain the various macro parameters in memory, until the end of the replacement text is reached. The stack is thus

only flushed after all the `\__seq_reverse_item:nw` are expanded. Keeping track of the arguments of all those calls uses up a memory quadratic in the length of the sequence. TeX can then not cope with more than a few thousand items.

Instead, we collect the items in the argument of `\exp_not:n`. The previous calls are cleanly removed from the stack, and the memory consumption becomes linear.

```

16665 \cs_new_protected:Npn \seq_reverse:N
16666   { \__seq_reverse>NN \__kernel_tl_set:Nx }
16667 \cs_new_protected:Npn \seq_greverse:N
16668   { \__seq_reverse>NN \__kernel_tl_gset:Nx }
16669 \cs_new_protected:Npn \__seq_reverse>NN #1 #2
16670   {
16671     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
16672     \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
16673     #1 #2 { #2 \exp_not:n { } }
16674     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
16675   }
16676 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
16677   {
16678     #2
16679     \exp_not:n { \__seq_item:n {#1} #3 }
16680   }
16681 \cs_generate_variant:Nn \seq_reverse:N { c }
16682 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 148.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

`\seq_gsort:Nn`

`\seq_gsort:cn`

## 56.4 Sequence conditionals

Similar to token lists, we compare with the empty sequence.

```

16683 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
16684   {
16685     \if_meaning:w #1 \c_empty_seq
16686       \prg_return_true:
16687     \else:
16688       \prg_return_false:
16689     \fi:
16690   }
16691 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
16692   { c } { p , T , F , TF }

```

(End definition for `\seq_if_empty:NTF`. This function is documented on page 148.)

`\seq_shuffle:N`

`\seq_shuffle:c`

`\seq_gshuffle:N`

`\seq_gshuffle:c`

`\__seq_shuffle:NN`

`\__seq_shuffle_item:n`

`\g__seq_internal_seq`

We apply the Fisher–Yates shuffle, storing items in `\toks` registers. We use the primitive `\tex_uniformdeviate:D` for speed reasons. Its non-uniformity is of order its argument divided by  $2^{28}$ , not too bad for small lists. For sequences with more than 13 elements there are more possible permutations than possible seeds ( $13! > 2^{28}$ ) so the question of uniformity is somewhat moot. The integer variables are declared in `l3int`: load-order issues.

```

16693 \cs_if_exist:NTF \tex_uniformdeviate:D

```

```

16694  {
16695    \seq_new:N \g__seq_internal_seq
16696    \cs_new_protected:Npn \seq_shuffle:N { \__seq_shuffle>NN \seq_set_eq:NN }
16697    \cs_new_protected:Npn \seq_gshuffle:N { \__seq_shuffle>NN \seq_gset_eq:NN }
16698    \cs_new_protected:Npn \__seq_shuffle>NN #1#2
16699    {
16700      \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
16701      {
16702        \msg_error:nnx { seq } { shuffle-too-large }
16703        { \token_to_str:N #2 }
16704      }
16705      {
16706        \group_begin:
16707          \int_zero:N \l__seq_internal_a_int
16708          \__seq_push_item_def:
16709          \cs_gset_eq:NN \__seq_item:n \__seq_shuffle_item:n
16710          #2
16711          \__seq_pop_item_def:
16712          \seq_gset_from_inline_x:Nnn \g__seq_internal_seq
16713          { \int_step_function:nN { \l__seq_internal_a_int } }
16714          { \tex_the:D \tex_toks:D ##1 }
16715        \group_end:
16716        #1 #2 \g__seq_internal_seq
16717        \seq_gclear:N \g__seq_internal_seq
16718      }
16719    }
16720    \cs_new_protected:Npn \__seq_shuffle_item:n
16721    {
16722      \int_incr:N \l__seq_internal_a_int
16723      \int_set:Nn \l__seq_internal_b_int
16724      { 1 + \tex_uniformdeviate:D \l__seq_internal_a_int }
16725      \tex_toks:D \l__seq_internal_a_int
16726      = \tex_toks:D \l__seq_internal_b_int
16727      \tex_toks:D \l__seq_internal_b_int
16728    }
16729  }
16730  {
16731    \cs_new_protected:Npn \seq_shuffle:N #1
16732    {
16733      \msg_error:nnn { kernel } { fp-no-random }
16734      { \seq_shuffle:N #1 }
16735    }
16736    \cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
16737  }
16738  \cs_generate_variant:Nn \seq_shuffle:N { c }
16739  \cs_generate_variant:Nn \seq_gshuffle:N { c }

```

(End definition for `\seq_shuffle:N` and others. These functions are documented on page 148.)

`\seq_if_in:NnTF` The approach here is to define `\__seq_item:n` to compare its argument with the test sequence. If the two items are equal, the mapping is terminated and `\group_end: \prg_return_true:` is inserted after skipping over the rest of the recursion. On the other hand, if there is no match then the loop breaks, returning `\prg_return_false:`. Everything is inside a group so that `\__seq_item:n` is preserved in nested situations.

`\seq_if_in:cNTF`  
`\seq_if_in:cVTF`  
`\seq_if_in:cVTF`  
`\seq_if_in:cOTF`  
`\seq_if_in:cXTF`  
`\__seq_if_in:`

```

16740 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
16741   { T , F , TF }
16742   {
16743     \group_begin:
16744       \tl_set:Nn \l__seq_internal_a_tl {##2}
16745       \cs_set_protected:Npn \__seq_item:n ##1
16746       {
16747         \tl_set:Nn \l__seq_internal_b_tl {##1}
16748         \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
16749           \exp_after:wN \__seq_if_in:
16750         \fi:
16751       }
16752       #1
16753     \group_end:
16754     \prg_return_false:
16755     \prg_break_point:
16756   }
16757 \cs_new:Npn \__seq_if_in:
16758   { \prg_break:n { \group_end: \prg_return_true: } }
16759 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
16760   { NV , Nv , No , Nx , c , cV , cv , co , cx } { T , F , TF }

```

(End definition for `\seq_if_in:NnTF` and `\__seq_if_in:`. This function is documented on page 149.)

## 56.5 Recovering data from sequences

`\__seq_pop:NNNN`  
`\__seq_pop_TF:NNNN`

The two pop functions share their emptiness tests. We also use a common emptiness test for all branching get and pop functions.

```

16761 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
16762   {
16763     \if_meaning:w #3 \c_empty_seq
16764       \tl_set:Nn #4 { \q_no_value }
16765     \else:
16766       #1#2#3#4
16767     \fi:
16768   }
16769 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
16770   {
16771     \if_meaning:w #3 \c_empty_seq
16772       % \tl_set:Nn #4 { \q_no_value }
16773     \prg_return_false:
16774     \else:
16775       #1#2#3#4
16776     \prg_return_true:
16777     \fi:
16778   }

```

(End definition for `\__seq_pop:NNNN` and `\__seq_pop_TF:NNNN`.)

`\seq_get_left:NN`  
`\seq_get_left:cN`  
`\__seq_get_left:wnw`

Getting an item from the left of a sequence is pretty easy: just trim off the first item after `\__seq_item:n` at the start. We append a `\q_no_value` item to cover the case of an empty sequence

```

16779 \cs_new_protected:Npn \seq_get_left:NN #1#2

```

```

16780    {
16781        \__kernel_tl_set:Nx #2
16782        {
16783            \exp_after:wN \__seq_get_left:wnw
16784            #1 \__seq_item:n { \q_no_value } \s__seq_stop
16785        }
16786    }
16787 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \s__seq_stop
16788    { \exp_not:n {#2} }
16789 \cs_generate_variant:Nn \seq_get_left:NN { c }

(End definition for \seq_get_left:NN and \__seq_get_left:wnw. This function is documented on page 145.)

```

### \seq\_pop\_left:NN

### \seq\_pop\_left:cN

### \seq\_gpop\_left:NN

### \seq\_gpop\_left:cN

### \\_\_seq\_pop\_left:NNN

### \\_\_seq\_pop\_left:wnwNNN

The approach to popping an item is pretty similar to that to get an item, with the only difference being that the sequence itself has to be redefined. This makes it more sensible to use an auxiliary function for the local and global cases.

```

16790 \cs_new_protected:Npn \seq_pop_left:NN
16791    { \__seq_pop>NNNN \__seq_pop_left>NNN \tl_set:Nn }
16792 \cs_new_protected:Npn \seq_gpop_left:NN
16793    { \__seq_pop>NNNN \__seq_pop_left>NNN \tl_gset:Nn }
16794 \cs_new_protected:Npn \__seq_pop_left>NNN #1#2#3
16795    { \exp_after:wN \__seq_pop_left:wnwNNN #2 \s__seq_stop #1#2#3 }
16796 \cs_new_protected:Npn \__seq_pop_left:wnwNNN
16797    #1 \__seq_item:n #2#3 \s__seq_stop #4#5#6
16798    {
16799        #4 #5 { #1 #3 }
16800        \tl_set:Nn #6 {#2}
16801    }
16802 \cs_generate_variant:Nn \seq_pop_left>NN { c }
16803 \cs_generate_variant:Nn \seq_gpop_left>NN { c }

(End definition for \seq_pop_left:NN and others. These functions are documented on page 145.)

```

### \seq\_get\_right:NN

### \seq\_get\_right:cN

### \\_\_seq\_get\_right\_loop:nw

### \\_\_seq\_get\_right\_end:NnN

First remove \s\_\_seq and prepend \q\_no\_value. The first argument of \\_\_seq\_get\_right\_loop:nw is the last item found, and the second argument is empty until the end of the loop, where it is code that applies \exp\_not:n to the last item and ends the loop.

```

16804 \cs_new_protected:Npn \seq_get_right:NN #1#2
16805    {
16806        \__kernel_tl_set:Nx #2
16807        {
16808            \exp_after:wN \use_i:nnn
16809            \exp_after:wN \__seq_get_right_loop:nw
16810            \exp_after:wN \q_no_value
16811            #1
16812            \__seq_get_right_end:NnN \__seq_item:n
16813        }
16814    }
16815 \cs_new:Npn \__seq_get_right_loop:nw #1#2 \__seq_item:n
16816    {
16817        #2 \use_none:n {#1}
16818        \__seq_get_right_loop:nw
16819    }
16820 \cs_new:Npn \__seq_get_right_end:NnN #1#2#3 { \exp_not:n {#2} }
16821 \cs_generate_variant:Nn \seq_get_right>NN { c }


```

(End definition for `\seq_get_right:NN`, `\_seq_get_right_loop:nw`, and `\_seq_get_right_end:NnN`. This function is documented on page 145.)

```
\seq_pop_right:NN
\seq_pop_right:cN
\seq_gpop_right:NN
\seq_gpop_right:cN
\_\seq_pop_right:NNN
\_\seq_pop_right_loop:nn
```

The approach to popping from the right is a bit more involved, but does use some of the same ideas as getting from the right. What is needed is a “flexible length” way to set a token list variable. This is supplied by the `{ \if_false: } \fi: ... \if_false: { \fi: }` construct. Using an x-type expansion and a “non-expanding” definition for `\_seq_item:n`, the left-most  $n - 1$  entries in a sequence of  $n$  items are stored back in the sequence. That needs a loop of unknown length, hence using the strange `\if_false:` way of including braces. When the last item of the sequence is reached, the closing brace for the assignment is inserted, and `\tl_set:Nn #3` is inserted in front of the final entry. This therefore does the pop assignment. One more iteration is performed, with an empty argument and `\use_none:nn`, which finally stops the loop.

```
16822 \cs_new_protected:Npn \seq_pop_right:NN
16823   { \_\seq_pop>NNNN \_\seq_pop_right>NNN \_kernel_tl_set:Nx }
16824   \cs_new_protected:Npn \seq_gpop_right:NN
16825   { \_\seq_pop>NNNN \_\seq_pop_right>NNN \_kernel_tl_gset:Nx }
16826   \cs_new_protected:Npn \_\seq_pop_right>NNN #1#2#3
16827   {
16828     \cs_set_eq:NN \_\seq_tmp:w \_\seq_item:n
16829     \cs_set_eq:NN \_\seq_item:n \scan_stop:
16830     #1 #2
16831     { \if_false: } \fi: \s_seq
16832     \exp_after:wN \use_i:nnn
16833     \exp_after:wN \_\seq_pop_right_loop:nn
16834     #2
16835     {
16836       \if_false: { \fi: }
16837       \_kernel_tl_set:Nx #3
16838     }
16839     { } \use_none:nn
16840     \cs_set_eq:NN \_\seq_item:n \_\seq_tmp:w
16841   }
16842 \cs_new:Npn \_\seq_pop_right_loop:nn #1#2
16843   {
16844     #2 { \exp_not:n {#1} }
16845     \_\seq_pop_right_loop:nn
16846   }
16847 \cs_generate_variant:Nn \seq_pop_right:NN { c }
16848 \cs_generate_variant:Nn \seq_gpop_right:NN { c }
```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 146.)

```
\seq_get_left:NNTF
\seq_get_left:cNTF
```

Getting from the left or right with a check on the results. The first argument to `\_seq_pop_TF:NNNN` is left unused.

```
\seq_get_right:NNTF
\seq_get_right:cNTF
```

```
16849 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
16850   { \_\seq_pop_TF>NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
16851 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
16852   { \_\seq_pop_TF>NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
16853 \prg_generate_conditional_variant:Nnn \seq_get_left:NN
16854   { c } { T , F , TF }
16855 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
16856   { c } { T , F , TF }
```

(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 147.)

`\seq_pop_left:NNTF` More or less the same for popping.

```
16857 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
16858   { T , F , TF }
16859   { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_set:Nn #1 #2 }
16860 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
16861   { T , F , TF }
16862   { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
16863 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
16864   { T , F , TF }
16865   { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_set:Nx #1 #2 }
16866 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
16867   { T , F , TF }
16868   { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \__kernel_tl_gset:Nx #1 #2 }
16869 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
16870   { T , F , TF }
16871 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
16872   { T , F , TF }
16873 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
16874   { T , F , TF }
16875 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
16876   { T , F , TF }
```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 147.)

`\seq_item:Nn` The idea here is to find the offset of the item from the left, then use a loop to grab the correct item. If the resulting offset is too large, then the argument delimited by `\__seq_item:n` is `\prg_break:` instead of being empty, terminating the loop and returning nothing at all.

```
16877 \cs_new:Npn \seq_item:Nn #1
16878   { \exp_after:wN \__seq_item:wNn #1 \s__seq_stop #1 }
16879 \cs_new:Npn \__seq_item:wNn \s__seq #1 \s__seq_stop #2#3
16880   {
16881     \exp_args:Nf \__seq_item:nwn
16882     { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
16883     #1
16884     \prg_break: \__seq_item:n { }
16885     \prg_break_point:
16886   }
16887 \cs_new:Npn \__seq_item:nN #1#2
16888   {
16889     \int_compare:nNnTF {#1} < 0
16890     { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
16891     {#1}
16892   }
16893 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
16894   {
16895     #2
16896     \int_compare:nNnTF {#1} = 1
16897     { \prg_break:n { \exp_not:n {#3} } }
16898     { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
16899   }
16900 \cs_generate_variant:Nn \seq_item:Nn { c }
```

(End definition for `\seq_item:Nn` and others. This function is documented on page 146.)

`\seq_rand_item:N` Importantly, `\seq_item:Nn` only evaluates its argument once.

```
16901 \cs_new:Npn \seq_rand_item:N #1
16902 {
16903     \seq_if_empty:NF #1
16904     { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }
16905 }
16906 \cs_generate_variant:Nn \seq_rand_item:N { c }
```

(End definition for `\seq_rand_item:N`. This function is documented on page 146.)

## 56.6 Mapping over sequences

`\seq_map_break:` To break a function, the special token `\prg_break_point:Nn` is used to find the end of the code. Any ending code is then inserted before the return value of `\seq_map_break:n` is inserted.

```
16907 \cs_new:Npn \seq_map_break:
16908     { \prg_map_break:Nn \seq_map_break: { } }
16909 \cs_new:Npn \seq_map_break:n
16910     { \prg_map_break:Nn \seq_map_break: }
```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 150.)

`\seq_map_function>NN` The idea here is to apply the code of #2 to each item in the sequence without altering the definition of `\_seq_item:n`. The even-numbered arguments of `\_seq_map_function:Nw` delimited by `\_seq_item:n` are almost always empty, except at the end of the loop where it is `\prg_break:..`. This allows to break the loop without needing to do a (relatively-expensive) quark test.

```
16911 \cs_new:Npn \seq_map_function>NN #1#2
16912 {
16913     \exp_after:wN \use_i_i:nnn
16914     \exp_after:wN \_seq_map_function:Nw
16915     \exp_after:wN #2
16916     #1
16917     \prg_break:
16918     \_seq_item:n { } \_seq_item:n { } \_seq_item:n { } \_seq_item:n { }
16919     \prg_break_point:
16920     \prg_break_point:Nn \seq_map_break: { }
16921 }
16922 \cs_new:Npn \_seq_map_function:Nw #1
16923     #2 \_seq_item:n #3
16924     #4 \_seq_item:n #5
16925     #6 \_seq_item:n #7
16926     #8 \_seq_item:n #9
16927 {
16928     #2 #1 {#3}
16929     #4 #1 {#5}
16930     #6 #1 {#7}
16931     #8 #1 {#9}
16932     \_seq_map_function:Nw #1
16933 }
16934 \cs_generate_variant:Nn \seq_map_function>NN { c }
```

(End definition for `\seq_map_function:NN` and `\_seq_map_function:Nw`. This function is documented on page 149.)

`\_seq_push_item_def:n` The definition of `\_seq_item:n` needs to be saved and restored at various points within the mapping and manipulation code. That is handled here: as always, this approach uses global assignments.

```

16935 \cs_new_protected:Npn \_seq_push_item_def:n
16936 {
16937     \_seq_push_item_def:
16938     \cs_gset:Npn \_seq_item:n ##1
16939 }
16940 \cs_new_protected:Npn \_seq_push_item_def:x
16941 {
16942     \_seq_push_item_def:
16943     \cs_gset:Npx \_seq_item:n ##1
16944 }
16945 \cs_new_protected:Npn \_seq_push_item_def:
16946 {
16947     \int_gincr:N \g_kernel_prg_map_int
16948     \cs_gset_eq:cN { _seq_map_ \int_use:N \g_kernel_prg_map_int :w }
16949     \_seq_item:n
16950 }
16951 \cs_new_protected:Npn \_seq_pop_item_def:
16952 {
16953     \cs_gset_eq:Nc \_seq_item:n
16954     { _seq_map_ \int_use:N \g_kernel_prg_map_int :w }
16955     \int_gdecr:N \g_kernel_prg_map_int
16956 }
```

(End definition for `\_seq_push_item_def:n`, `\_seq_push_item_def:x`, and `\_seq_pop_item_def:..`)

`\seq_map_inline:Nn` The idea here is that `\_seq_item:n` is already “applied” to each item in a sequence, and so an in-line mapping is just a case of redefining `\_seq_item:n`.

```

16957 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
16958 {
16959     \_seq_push_item_def:n {#2}
16960     #1
16961     \prg_break_point:Nn \seq_map_break: { \_seq_pop_item_def: }
16962 }
16963 \cs_generate_variant:Nn \seq_map_inline:Nn { c }
```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 149.)

`\seq_map_tokens:Nn` This is based on the function mapping but using the same tricks as described for `\prop_map_tokens:Nn`. The idea is to remove the leading `\s_seq` and apply the tokens such that they are safe with the break points, hence the `\use:n`.

```

16964 \cs_new:Npn \seq_map_tokens:Nn #1#2
16965 {
16966     \exp_last_unbraced:Nno
16967     \use_i:nn { \_seq_map_tokens:nw {#2} } #1
16968     \prg_break:
16969     \_seq_item:n { } \_seq_item:n { } \_seq_item:n { } \_seq_item:n { }
16970     \prg_break_point:
16971     \prg_break_point:Nn \seq_map_break: { }
```

```

16972     }
16973 \cs_generate_variant:Nn \seq_map_tokens:Nn { c }
16974 \cs_new:Npn \__seq_map_tokens:nw #1
16975     #2 \__seq_item:n #3
16976     #4 \__seq_item:n #5
16977     #6 \__seq_item:n #7
16978     #8 \__seq_item:n #9
16979 {
16980     #2 \use:n {#1} {#3}
16981     #4 \use:n {#1} {#5}
16982     #6 \use:n {#1} {#7}
16983     #8 \use:n {#1} {#9}
16984     \__seq_map_tokens:nw {#1}
16985 }

```

(End definition for `\seq_map_tokens:Nn` and `\__seq_map_tokens:nw`. This function is documented on page 149.)

`\seq_map_variable>NNn`  
`\seq_map_variable:Ncn`  
`\seq_map_variable:cNn`  
`\seq_map_variable:cCNn`

This is just a specialised version of the in-line mapping function, using an x-type expansion for the code set up so that the number of # tokens required is as expected.

```

16986 \cs_new_protected:Npn \seq_map_variable>NNn #1#2#3
16987 {
16988     \__seq_push_item_def:x
16989     {
16990         \tl_set:Nn \exp_not:N #2 {##1}
16991         \exp_not:n {#3}
16992     }
16993     #1
16994     \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
16995 }
16996 \cs_generate_variant:Nn \seq_map_variable:NNn { Nc }
16997 \cs_generate_variant:Nn \seq_map_variable>NNn { c , cc }

```

(End definition for `\seq_map_variable>NNn`. This function is documented on page 149.)

`\seq_map_indexed_function>NN`  
`\seq_map_indexed_inline:Nn`

```

\__seq_map_indexed:nNN
\__seq_map_indexed:Nw
16998 \cs_new:Npn \seq_map_indexed_function:NN #1#2
16999 {
17000     \__seq_map_indexed:NN #1#2
17001     \prg_break_point:Nn \seq_map_break: { }
17002 }
17003 \cs_new_protected:Npn \seq_map_indexed_inline:Nn #1#2
17004 {
17005     \int_gincr:N \g_kernel_prg_map_int
17006     \cs_gset_protected:cpn
17007     { \__seq_map_ \int_use:N \g_kernel_prg_map_int :w } ##1##2 {#2}
17008     \exp_args:NNc \__seq_map_indexed:NN #1
17009     { \__seq_map_ \int_use:N \g_kernel_prg_map_int :w }
17010     \prg_break_point:Nn \seq_map_break:
17011     { \int_gdecr:N \g_kernel_prg_map_int }
17012 }
17013 \cs_new:Npn \__seq_map_indexed:NN #1#2
17014 {

```

```

17015      \exp_after:wN \__seq_map_indexed:Nw
17016      \exp_after:wN #2
17017      \int_value:w 1
17018      \exp_after:wN \use_i:nn
17019      \exp_after:wN ;
17020      #1
17021      \prg_break: \__seq_item:n { } \prg_break_point:
17022  }
17023 \cs_new:Npn \__seq_map_indexed:Nw #1#2 ; #3 \__seq_item:n #4
17024  {
17025      #3
17026      #1 {#2} {#4}
17027      \exp_after:wN \__seq_map_indexed:Nw
17028      \exp_after:wN #1
17029      \int_value:w \int_eval:w 1 + #2 ;
17030  }

```

(End definition for `\seq_map_indexed_function:NN` and others. These functions are documented on page 149.)

`\seq_set_map_x:NNn`  
`\seq_gset_map_x:NNn`

Very similar to `\seq_set_filter:NNn`. We could actually merge the two within a single function, but it would have weird semantics.

```

\__seq_set_map_x:NNNn
17031 \cs_new_protected:Npn \seq_set_map_x:NNn
17032  { \__seq_set_map_x:NNNn \__kernel_tl_set:Nx }
17033 \cs_new_protected:Npn \seq_gset_map_x:NNn
17034  { \__seq_set_map_x:NNNn \__kernel_tl_gset:Nx }
17035 \cs_new_protected:Npn \__seq_set_map_x:NNNn #1#2#3#4
17036  {
17037      \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
17038      #1 #2 { #3 }
17039      \__seq_pop_item_def:
17040  }

```

(End definition for `\seq_set_map_x:NNn`, `\seq_gset_map_x:NNn`, and `\__seq_set_map_x:NNNn`. These functions are documented on page 151.)

`\seq_set_map>NNn`  
`\seq_gset_map>NNn`

Similar to `\seq_set_map_x:NNn`, but prevents expansion of the <inline function>.

```

17041 \cs_new_protected:Npn \seq_set_map>NNn
17042  { \__seq_set_map:NNNn \__kernel_tl_set:Nx }
17043 \cs_new_protected:Npn \seq_gset_map>NNn
17044  { \__seq_set_map:NNNn \__kernel_tl_gset:Nx }
17045 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
17046  {
17047      \__seq_push_item_def:n { \exp_not:n { \__seq_item:n {#4} } }
17048      #1 #2 { #3 }
17049      \__seq_pop_item_def:
17050  }

```

(End definition for `\seq_set_map>NNn`, `\seq_gset_map>NNn`, and `\__seq_set_map:NNNn`. These functions are documented on page 151.)

`\seq_count:N`  
`\seq_count:c`  
`\__seq_count:w`  
`\__seq_count_end:w`

Since counting the items in a sequence is quite common, we optimize it by grabbing 8 items at a time and correspondingly adding 8 to an integer expression. At the end of the loop, #9 is `\__seq_count_end:w` instead of being empty. It removes `8+` and instead

places the number of `\_seq_item:n` that `\_seq_count:w` grabbed before reaching the end of the sequence.

```

17051 \cs_new:Npn \seq_count:N #1
17052 {
17053     \int_eval:n
17054     {
17055         \exp_after:wN \use_i:nn
17056         \exp_after:wN \_seq_count:w
17057         #1
17058         \_seq_count_end:w \_seq_item:n 7
17059         \_seq_count_end:w \_seq_item:n 6
17060         \_seq_count_end:w \_seq_item:n 5
17061         \_seq_count_end:w \_seq_item:n 4
17062         \_seq_count_end:w \_seq_item:n 3
17063         \_seq_count_end:w \_seq_item:n 2
17064         \_seq_count_end:w \_seq_item:n 1
17065         \_seq_count_end:w \_seq_item:n 0
17066         \prg_break_point:
17067     }
17068 }
17069 \cs_new:Npn \_seq_count:w
17070     #1 \_seq_item:n #2 \_seq_item:n #3 \_seq_item:n #4 \_seq_item:n
17071     #5 \_seq_item:n #6 \_seq_item:n #7 \_seq_item:n #8 #9 \_seq_item:n
17072     { #9 8 + \_seq_count:w }
17073 \cs_new:Npn \_seq_count_end:w 8 + \_seq_count:w #1#2 \prg_break_point: {#1}
17074 \cs_generate_variant:Nn \seq_count:N { c }

```

(End definition for `\seq_count:N`, `\_seq_count:w`, and `\_seq_count_end:w`. This function is documented on page [151](#).)

## 56.7 Using sequences

`\seq_use:Nnnn`

See `\clist_use:Nnnn` for a general explanation. The main difference is that we use `\_seq_item:n` as a delimiter rather than commas. We also need to add `\_seq_item:n` at various places, and `\s_seq`.

```

17075 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
17076 {
17077     \seq_if_exist:NTF #1
17078     {
17079         \int_case:nnF { \seq_count:N #1 }
17080         {
17081             { 0 } { }
17082             { 1 } { \exp_after:wN \_seq_use:NNnNnn #1 ? { } { } }
17083             { 2 } { \exp_after:wN \_seq_use:NNnNnn #1 {#2} }
17084         }
17085         {
17086             \exp_after:wN \_seq_use_setup:w #1 \_seq_item:n
17087             \s_seq_mark { \_seq_use:nwwwnwn {#3} }
17088             \s_seq_mark { \_seq_use:nwwn {#4} }
17089             \s_seq_stop { }
17090         }
17091     }
17092 }

```

```

17093         \msg_expandable_error:nnn
17094             { kernel } { bad-variable } {#1}
17095     }
17096 }
17097 \cs_generate_variant:Nn \seq_use:Nnnn { c }
17098 \cs_new:Npn \__seq_use:NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
17099 \cs_new:Npn \__seq_use_setup:w \s__seq { \__seq_use:nwwwnwn { } }
17100 \cs_new:Npn \__seq_use:nwwwnwn
17101     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
17102     \s__seq_mark #6#7 \s__seq_stop #8
17103 {
17104     #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
17105     \s__seq_mark {#6} #7 \s__seq_stop { #8 #1 #2 }
17106 }
17107 \cs_new:Npn \__seq_use:nwwn #1 \__seq_item:n #2 #3 \s__seq_stop #4
17108 { \exp_not:n { #4 #1 #2 } }
17109 \cs_new:Npn \seq_use:Nn #1#2
17110 { \seq_use:Nnnn #1 {#2} {#2} {#2} }
17111 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 151.)

## 56.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

**\seq\_push:Nn**

Pushing to a sequence is the same as adding on the left.

```

17112 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
17113 \cs_new_eq:NN \seq_push:NV \seq_put_left:NV
17114 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
17115 \cs_new_eq:NN \seq_push:No \seq_put_left:No
17116 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
17117 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
17118 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
17119 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
17120 \cs_new_eq:NN \seq_push:co \seq_put_left:co
17121 \cs_new_eq:NN \seq_push:cX \seq_put_left:cX
\seq_gpush:Nn
\seq_gpush:NV
\seq_gpush:NV
\seq_gpush:Nv
\seq_gpush:No
\seq_gpush:Nx
\seq_gpush:cn
\seq_gpush:cV
\seq_gpush:cV
\seq_gpush:co
\seq_gpush:cX

```

(End definition for `\seq_push:Nn` and `\seq_gpush:Nn`. These functions are documented on page 153.)

**\seq\_get:NN** In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.

```

\seq_pop:NN
\seq_pop:cN
\seq_gpop:NN
\seq_gpop:cN

```

```

17133 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
17134 \cs_new_eq:NN \seq_pop:NN \seq_pop_left:NN
17135 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
17136 \cs_new_eq:NN \seq_gpop:NN \seq_gpop_left:NN
17137 \cs_new_eq:NN \seq_gpop:cN \seq_gpop_left:cN

```

(End definition for `\seq_get:NN`, `\seq_pop:NN`, and `\seq_gpop:NN`. These functions are documented on page 152.)

`\seq_get:NNTF` More copies.

```

\seq_get:cNNTF 17138 \prg_new_eq_conditional:Nnn \seq_get:NN \seq_get_left:NN { T , F , TF }
\seq_pop:NNTF 17139 \prg_new_eq_conditional:Nnn \seq_get:cN \seq_get_left:cN { T , F , TF }
\seq_pop:cNNTF 17140 \prg_new_eq_conditional:Nnn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
\seq_gpop:NNTF 17141 \prg_new_eq_conditional:Nnn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
\seq_gpop:cNNTF 17142 \prg_new_eq_conditional:Nnn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
17143 \prg_new_eq_conditional:Nnn \seq_gpop:cN \seq_gpop_left:cN { T , F , TF }

```

(End definition for `\seq_get:NNTF`, `\seq_pop:NNTF`, and `\seq_gpop:NNTF`. These functions are documented on page 152.)

## 56.9 Viewing sequences

`\seq_show:N` Apply the general `\_kernel_chk_tl_type:NnnT`.

```

\seq_show:c 17144 \cs_new_protected:Npn \seq_show:N { \_seq_show:NN \msg_show:nxxxxx }
\seq_log:N 17145 \cs_generate_variant:Nn \seq_show:N { c }
\seq_log:c 17146 \cs_new_protected:Npn \seq_log:N { \_seq_show:NN \msg_log:nxxxxx }
\_\seq_show>NN 17147 \cs_generate_variant:Nn \seq_log:N { c }
\_\seq_show_validate:nn 17148 \cs_new_protected:Npn \_\seq_show:NN #1#2
17149 {
17150     \_kernel_chk_tl_type:NnnT #2 { seq }
17151     {
17152         \s__seq
17153         \exp_after:wN \use_i:nn \exp_after:wN \_\seq_show_validate:nn #2
17154         \q_recursion_tail \q_recursion_tail \q_recursion_stop
17155     }
17156     {
17157         #1 { seq } { show }
17158         { \token_to_str:N #2 }
17159         { \seq_map_function:NN #2 \msg_show_item:n }
17160         { } { }
17161     }
17162 }
17163 \cs_new:Npn \_\seq_show_validate:nn #1#2
17164 {
17165     \quark_if_recursion_tail_stop:n {#2}
17166     \_\seq_wrap_item:n {#2}
17167     \_\seq_show_validate:nn
17168 }

```

(End definition for `\seq_show:N` and others. These functions are documented on page 155.)

## 56.10 Scratch sequences

\l\_tmpa\_seq      Temporary comma list variables.  
\\l\_tmpb\_seq  
\g\_tmpa\_seq  
\g\_tmpb\_seq

17169 \seq\_new:N \\l\_tmpa\_seq  
17170 \seq\_new:N \\l\_tmpb\_seq  
17171 \seq\_new:N \\g\_tmpa\_seq  
17172 \seq\_new:N \\g\_tmpb\_seq

(End definition for \l\_tmpa\_seq and others. These variables are documented on page 155.)  
17173 </package>

# Chapter 57

## I3int implementation

```
17174  <*package>
17175  <@@=int>
```

*The following test files are used for this code: m3int001,m3int002,m3int03.*

**\c\_max\_register\_int** Done in I3basics.

(End definition for \c\_max\_register\_int. This variable is documented on page 167.)

**\\_\_int\_to\_roman:w** Done in I3basics.

**\if\_int\_compare:w**

(End definition for \\_\_int\_to\_roman:w and \if\_int\_compare:w. This function is documented on page 168.)

**\or:** Done in I3basics.

(End definition for \or:. This function is documented on page 168.)

**\int\_value:w** Here are the remaining primitives for number comparisons and expressions.

```
17176  \cs_new_eq:NN \int_value:w      \tex_number:D
17177  \cs_new_eq:NN \__int_eval:w     \tex_numexpr:D
17178  \cs_new_eq:NN \__int_eval_end:   \tex_relax:D
17179  \cs_new_eq:NN \if_int_odd:w    \tex_ifodd:D
17180  \cs_new_eq:NN \if_case:w       \tex_ifcase:D
```

(End definition for \int\_value:w and others. These functions are documented on page 168.)

**\s\_\_int\_mark** Scan marks used throughout the module.

```
17181  \scan_new:N \s__int_mark
17182  \scan_new:N \s__int_stop
```

(End definition for \s\_\_int\_mark and \s\_\_int\_stop.)

**\\_\_int\_use\_none\_delimit\_by\_s\_stop:w** Function to gobble until a scan mark.

```
17183  \cs_new:Npn \__int_use_none_delimit_by_s_stop:w #1 \s__int_stop { }
```

(End definition for \\_\_int\_use\_none\_delimit\_by\_s\_stop:w.)

**\q\_\_int\_recursion\_tail** Quarks for recursion.

```
17184  \quark_new:N \q__int_recursion_tail
17185  \quark_new:N \q__int_recursion_stop
```

(End definition for `\q_int_recursion_tail` and `\q_int_recursion_stop`.)

`\_int_if_recursion_tail_stop:Fn`

```
17186 \_kernel_quark_new_test:N \_int_if_recursion_tail_stop:Fn  
17187 \_kernel_quark_new_test:N \_int_if_recursion_tail_stop:N
```

(End definition for `\_int_if_recursion_tail_stop:Fn` and `\_int_if_recursion_tail_stop:N`.)

## 57.1 Integer expressions

`\int_eval:n` Wrapper for `\_int_eval:w`: can be used in an integer expression or directly in the input stream. It is very slightly faster to use `\the` rather than `\number` to turn the expression to a number. When debugging, we introduce parentheses to catch early termination (see `\l3debug`).

```
17188 \cs_new:Npn \int_eval:n #1  
17189 { \tex_the:D \_int_eval:w #1 \_int_eval_end: }  
17190 \cs_new:Npn \int_eval:w { \tex_the:D \_int_eval:w }
```

(End definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 157.)

`\int_sign:n` See `\int_abs:n`. Evaluate the expression once (and when debugging is enabled, check that the expression is well-formed), then test the first character to determine the sign. This is wrapped in `\int_value:w ... \exp_stop_f:` to ensure a fixed number of expansions and to avoid dealing with closing the conditionals.

```
17191 \cs_new:Npn \int_sign:n #1  
17192 {  
17193 \int_value:w \exp_after:wN \_int_sign:Nw  
17194 \int_value:w \_int_eval:w #1 \_int_eval_end: ;  
17195 \exp_stop_f:  
17196 }  
17197 \cs_new:Npn \_int_sign:Nw #1#2 ;  
17198 {  
17199 \if_meaning:w 0 #1  
17200 0  
17201 \else:  
17202 \if_meaning:w - #1 - \fi: 1  
17203 \fi:  
17204 }
```

(End definition for `\int_sign:n` and `\_int_sign:Nw`. This function is documented on page 157.)

`\int_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is obtained by removing a leading sign if any. All three functions expand in two steps.

```
17205 \cs_new:Npn \int_abs:n #1  
17206 {  
17207 \int_value:w \exp_after:wN \_int_abs:N  
17208 \int_value:w \_int_eval:w #1 \_int_eval_end:  
17209 \exp_stop_f:  
17210 }  
17211 \cs_new:Npn \_int_abs:N #1  
17212 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }  
17213 \cs_set:Npn \int_max:nn #1#2  
17214 {
```

```

17215     \int_value:w \exp_after:wN \__int_maxmin:wwN
17216     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17217     \int_value:w \__int_eval:w #2 ;
17218     >
17219     \exp_stop_f:
17220   }
17221 \cs_set:Npn \int_min:nn #1#2
17222   {
17223     \int_value:w \exp_after:wN \__int_maxmin:wwN
17224     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17225     \int_value:w \__int_eval:w #2 ;
17226     <
17227     \exp_stop_f:
17228   }
17229 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
17230   {
17231     \if_int_compare:w #1 #3 #2 ~
17232       #1
17233     \else:
17234       #2
17235     \fi:
17236   }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 157.)

`\int_div_truncate:nn`  
`\int_div_round:nn`  
`\int_mod:nn`

`\__int_div_truncate:NwNw`  
`\__int_mod:ww`

As `\__int_eval:w` rounds the result of a division we also provide a version that truncates the result. We use an auxiliary to make sure numerator and denominator are only evaluated once: this comes in handy when those are more expensive to evaluate (e.g., `\tl_count:n`). If the numerator `#1#2` is 0, then we divide 0 by the denominator (this ensures that 0/0 is correctly reported as an error). Otherwise, shift the numerator `#1#2` towards 0 by  $(|#3#4|-1)/2$ , which we round away from zero. It turns out that this quantity exactly compensates the difference between  $\varepsilon$ -TEX's rounding and the truncating behaviour that we want. The details are thanks to Heiko Oberdiek: getting things right in all cases is not so easy.

```

17237 \cs_new:Npn \int_div_truncate:nn #1#2
17238   {
17239     \int_value:w \__int_eval:w
17240       \exp_after:wN \__int_div_truncate:NwNw
17241     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17242     \int_value:w \__int_eval:w #2 ;
17243     \__int_eval_end:
17244   }
17245 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
17246   {
17247     \if_meaning:w 0 #1
17248       0
17249     \else:
17250       (
17251         #1#2
17252         \if_meaning:w - #1 + \else: - \fi:
17253         ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
17254       )
17255     \fi:
17256     / #3#4

```

```
17257 }
```

For the sake of completeness:

```
17258 \cs_new:Npn \int_div_round:nn #1#2
17259   { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }
```

Finally there's the modulus operation.

```
17260 \cs_new:Npn \int_mod:nn #1#2
17261   {
17262     \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
17263     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17264     \int_value:w \__int_eval:w #2 ;
17265     \__int_eval_end:
17266   }
17267 \cs_new:Npn \__int_mod:ww #1; #2;
17268   { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }
```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 157.)

`\__kernel_int_add:nnn` Equivalent to `\int_eval:n {#1+#2+#3}` except that overflow only occurs if the final result overflows  $[-2^{31} + 1, 2^{31} - 1]$ . The idea is to choose the order in which the three numbers are added together. If #1 and #2 have opposite signs (one is in  $[-2^{31} + 1, -1]$  and the other in  $[0, 2^{31} - 1]$ ) then #1+#2 cannot overflow so we compute the result as #1+#2+#3. If they have the same sign, then either #3 has the same sign and the order does not matter, or #3 has the opposite sign and any order in which #3 is not last will work. We use #1+#3+#2.

```
17269 \cs_new:Npn \__kernel_int_add:nnn #1#2#3
17270   {
17271     \int_value:w \__int_eval:w #1
17272     \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
17273     \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
17274     \__int_eval_end:
17275   }
```

(End definition for `\__kernel_int_add:nnn`.)

## 57.2 Creating and initialising integers

`\int_new:N` Two ways to do this: one for the format and one for the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package. In plain T<sub>E</sub>X, `\newcount` (and other allocators) are `\outer`: to allow the code here to work in “generic” mode this is therefore accessed by name. (The same applies to `\newbox`, `\newdimen` and so on.)

```
17276 \cs_new_protected:Npn \int_new:N #1
17277   {
17278     \__kernel_chk_if_free_cs:N #1
17279     \cs:w newcount \cs_end: #1
17280   }
17281 \cs_generate_variant:Nn \int_new:N { c }
```

(End definition for `\int_new:N`. This function is documented on page 158.)

```
\int_const:Nn
\int_const:cn
\__int_const:nN
\__int_constdef:Nw
\c__int_max_constdef_int
```

As stated, most constants can be defined as `\chardef` or `\mathchardef` but that's engine dependent. As a result, there is some set up code to determine what can be done. No full engine testing just yet so everything is a little awkward. We cannot use `\int_gset:Nn` because (when `check-declarations` is enabled) this runs some checks that constants would fail.

```
17282 \cs_new_protected:Npn \int_const:Nn #1#2
17283   { \exp_args:Nx \__int_const:nN { \int_eval:n {#2} } #1 }
17284 \cs_new_protected:Npn \__int_const:nN #1#2
17285   {
17286     \int_compare:nNnTF {#1} < \c_zero_int
17287       {
17288         \int_new:N #2
17289         \tex_global:D
17290       }
17291     {
17292       \int_compare:nNnTF {#1} > \c__int_max_constdef_int
17293         {
17294           \int_new:N #2
17295           \tex_global:D
17296         }
17297       {
17298         \__kernel_chk_if_free_cs:N #2
17299         \tex_global:D \__int_constdef:Nw
17300       }
17301     }
17302     #2 = \__int_eval:w #1 \__int_eval_end:
17303   }
17304 \cs_generate_variant:Nn \int_const:Nn { c }
17305 \if_int_odd:w 0
17306   \cs_if_exist:NT \tex_luatexversion:D { 1 }
17307   \cs_if_exist:NT \tex_omathchardef:D { 1 }
17308   \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
17309   \cs_if_exist:NTF \tex_omathchardef:D
17310     { \cs_new_eq:NN \__int_constdef:Nw \tex_omathchardef:D }
17311     { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
17312   \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
17313 \else:
17314   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
17315   \tex_mathchardef:D \c__int_max_constdef_int 32767 ~
17316 \fi:
```

(End definition for `\int_const:Nn` and others. This function is documented on page 158.)

```
\int_zero:N Functions that reset an <integer> register to zero.
\int_zero:c
\int_zero:N
\int_gzero:N
\int_gzero:c
\int_gzero:N
```

- 17317 \cs\_new\_protected:Npn \int\_zero:N #1 { #1 = \c\_zero\_int }
- 17318 \cs\_new\_protected:Npn \int\_gzero:N #1 { \tex\_global:D #1 = \c\_zero\_int }
- 17319 \cs\_generate\_variant:Nn \int\_zero:N { c }
- 17320 \cs\_generate\_variant:Nn \int\_gzero:N { c }

(End definition for `\int_zero:N` and `\int_gzero:N`. These functions are documented on page 158.)

```
\int_zero_new:N Create a register if needed, otherwise clear it.
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c
```

- 17321 \cs\_new\_protected:Npn \int\_zero\_new:N #1
- 17322 { \int\_if\_exist:NTF #1 { \int\_zero:N #1 } { \int\_new:N #1 } }

```

17323 \cs_new_protected:Npn \int_gzero_new:N #1
17324   { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
17325 \cs_generate_variant:Nn \int_zero_new:N { c }
17326 \cs_generate_variant:Nn \int_gzero_new:N { c }

(End definition for \int_zero_new:N and \int_gzero_new:N. These functions are documented on page 158.)

```

**\int\_set\_eq:NN** Setting equal means using one integer inside the set function of another. Check that assigned integer is local/global. No need to check that the other one is defined as T<sub>E</sub>X does it for us.

```

17327 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
17328 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
17329 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
17330 \cs_generate_variant:Nn \int_gset_eq:NN { c , Nc , cc }

(End definition for \int_set_eq:NN and \int_gset_eq:NN. These functions are documented on page 158.)

```

**\int\_if\_exist\_p:N** Copies of the cs functions defined in l3basics.

```

17331 \prg_new_eq_conditional:Nnn \int_if_exist:N \cs_if_exist:N
17332   { TF , T , F , p }
17333 \prg_new_eq_conditional:Nnn \int_if_exist:c \cs_if_exist:c
17334   { TF , T , F , p }

(End definition for \int_if_exist:NTF. This function is documented on page 158.)

```

### 57.3 Setting and incrementing integers

**\int\_add:Nn** Adding and subtracting to and from a counter. Including here the optional *by* would slow down these operations by a few percent.

```

17335 \cs_new_protected:Npn \int_add:Nn #1#2
17336   { \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
17337 \cs_new_protected:Npn \int_sub:Nn #1#2
17338   { \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
17339 \cs_new_protected:Npn \int_gadd:Nn #1#2
17340   { \tex_global:D \tex_advance:D #1 \__int_eval:w #2 \__int_eval_end: }
17341 \cs_new_protected:Npn \int_gsub:Nn #1#2
17342   { \tex_global:D \tex_advance:D #1 - \__int_eval:w #2 \__int_eval_end: }
17343 \cs_generate_variant:Nn \int_add:Nn { c }
17344 \cs_generate_variant:Nn \int_gadd:Nn { c }
17345 \cs_generate_variant:Nn \int_sub:Nn { c }
17346 \cs_generate_variant:Nn \int_gsub:Nn { c }

(End definition for \int_add:Nn and others. These functions are documented on page 159.)

```

**\int\_incr:N** Incrementing and decrementing of integer registers is done with the following functions.

```

17347 \cs_new_protected:Npn \int_incr:N #1
17348   { \tex_advance:D #1 \c_one_int }
17349 \cs_new_protected:Npn \int_decr:N #1
17350   { \tex_advance:D #1 - \c_one_int }
17351 \cs_new_protected:Npn \int_gincr:N #1
17352   { \tex_global:D \tex_advance:D #1 \c_one_int }
17353 \cs_new_protected:Npn \int_gdecr:N #1


```

```

17354   { \tex_global:D \tex_advance:D #1 - \c_one_int }
17355   \cs_generate_variant:Nn \int_incr:N { c }
17356   \cs_generate_variant:Nn \int_decr:N { c }
17357   \cs_generate_variant:Nn \int_gincr:N { c }
17358   \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for `\int_incr:N` and others. These functions are documented on page 159.)

`\int_set:Nn`  
`\int_set:cN`  
`\int_gset:Nn`

As integers are register-based TeX issues an error if they are not defined. While the = sign is optional, this version with = is slightly quicker than without, while adding the optional space after = slows things down minutely.

`\int_gset:cn`

```

17359   \cs_new_protected:Npn \int_set:Nn #1#2
17360     { #1 = \__int_eval:w #2 \__int_eval_end: }
17361   \cs_new_protected:Npn \int_gset:Nn #1#2
17362     { \tex_global:D #1 = \__int_eval:w #2 \__int_eval_end: }
17363   \cs_generate_variant:Nn \int_set:Nn { c }
17364   \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for `\int_set:Nn` and `\int_gset:Nn`. These functions are documented on page 159.)

## 57.4 Using integers

`\int_use:N`

Here is how counters are accessed. We hand-code the c variant for some speed gain.

`\int_use:c`

```

17365   \cs_new_eq:NN \int_use:N \tex_the:D
17366   \cs_new:Npn \int_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\int_use:N`. This function is documented on page 159.)

## 57.5 Integer expression conditionals

`\__int_compare_error:`  
`\__int_compare_error:Nw`

Those functions are used for comparison tests which use a simple syntax where only one set of braces is required and additional operators such as != and >= are supported. The tests first evaluate their left-hand side, with a trailing `\__int_compare_error:`. This marker is normally not expanded, but if the relation symbol is missing from the test's argument, then the marker inserts = (and itself) after triggering the relevant TeX error. If the first token which appears after evaluating and removing the left-hand side is not a known relation symbol, then a judiciously placed `\__int_compare_error:Nw` gets expanded, cleaning up the end of the test and telling the user what the problem was.

```

17367   \cs_new_protected:Npn \__int_compare_error:
17368     {
17369       \if_int_compare:w \c_zero_int \c_zero_int \fi:
17370       =
17371       \__int_compare_error:
17372     }
17373   \cs_new:Npn \__int_compare_error:Nw
17374     #1#2 \s__int_stop
17375   {
17376     {}
17377     \c_zero_int \fi:
17378     \msg_expandable_error:nnn
17379       { kernel } { unknown-comparison } {#1}
17380     \prg_return_false:
17381   }

```

(End definition for `\_int_compare_error:` and `\_int_compare_error:Nw.`)

```
\int_compare_p:n
\int_compare:nTF
  \_int_compare:w
  \_int_compare:Nw
  \_int_compare:NNw
  \_int_compare:nnN
\_int_compare_end_=:NNw
  \_int_compare_=:NNw
  \_int_compare_<:NNw
  \_int_compare_>:NNw
  \_int_compare_==:NNw
  \_int_compare_!=:NNw
  \_int_compare_<=:NNw
  \_int_compare_>=:NNw
```

Comparison tests using a simple syntax where only one set of braces is required, additional operators such as `!=` and `>=` are supported, and multiple comparisons can be performed at once, for instance `0 < 5 <= 1`. The idea is to loop through the argument, finding one operand at a time, and comparing it to the previous one. The looping auxiliary `\_int_compare:Nw` reads one `<operand>` and one `<comparison>` symbol, and leaves roughly

```
<operand> \prg_return_false: \fi:
\reverse_if:N \if_int_compare:w <operand> <comparison>
\_int_compare:Nw
```

in the input stream. Each call to this auxiliary provides the second operand of the last call's `\if_int_compare:w`. If one of the `<comparisons>` is `false`, the `true` branch of the TeX conditional is taken (because of `\reverse_if:N`), immediately returning `false` as the result of the test. There is no TeX conditional waiting the first operand, so we add an `\if_false:` and expand by hand with `\int_value:w`, thus skipping `\prg_return_false:` on the first iteration.

Before starting the loop, the first step is to make sure that there is at least one relation symbol. We first let TeX evaluate this left hand side of the (in)equality using `\_int_eval:w`. Since the relation symbols `<`, `>`, `=` and `!` are not allowed in integer expressions, they would terminate the expression. If the argument contains no relation symbol, `\_int_compare_error:` is expanded, inserting `=` and itself after an error. In all cases, `\_int_compare:w` receives as its argument an integer, a relation symbol, and some more tokens. We then setup the loop, which is ended by the two odd-looking items `e` and `{=nd_}`, with a trailing `\s__int_stop` used to grab the entire argument when necessary.

```
17382 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
17383   {
17384     \exp_after:wN \_int_compare:w
17385     \int_value:w \_int_eval:w #1 \_int_compare_error:
17386   }
17387 \cs_new:Npn \_int_compare:w #1 \_int_compare_error:
17388   {
17389     \exp_after:wN \if_false: \int_value:w
17390     \_int_compare:Nw #1 e { = nd_ } \s__int_stop
17391   }
```

The goal here is to find an `<operand>` and a `<comparison>`. The `<operand>` is already evaluated, but we cannot yet grab it as an argument. To access the following relation symbol, we remove the number by applying `\_int_to_roman:w`, after making sure that the argument becomes non-positive: its roman numeral representation is then empty. Then probe the first two tokens with `\_int_compare:NNw` to determine the relation symbol, building a control sequence from it (`\token_to_str:N` gives better errors if `#1` is not a character). All the extended forms have an extra `=` hence the test for that as a second token. If the relation symbol is unknown, then the control sequence is turned by TeX into `\scan_stop:`, ignored thanks to `\unexpanded`, and `\_int_compare_error:Nw` raises an error.

```
17392 \cs_new:Npn \_int_compare:Nw #1#2 \s__int_stop
17393   {
17394     \exp_after:wN \_int_compare:NNw
17395     \_int_to_roman:w - 0 #2 \s__int_mark
```

```

17396      #1#2 \s__int_stop
17397    }
17398 \cs_new:Npn \__int_compare:NNw #1#2#3 \s__int_mark
17399  {
17400    \__kernel_exp_not:w
17401    \use:c
17402    {
17403      __int_compare_ \token_to_str:N #1
17404      \if_meaning:w = #2 = \fi:
17405      :NNw
17406    }
17407    \__int_compare_error:Nw #1
17408  }

```

When the last  $\langle operand \rangle$  is seen, `\__int_compare:NNw` receives `e` and `=nd_` as arguments, hence calling `\__int_compare_end_=:NNw` to end the loop: return the result of the last comparison (involving the operand that we just found). When a normal relation is found, the appropriate auxiliary calls `\__int_compare:nnN` where `#1` is `\if_int_compare:w` or `\reverse_if:N \if_int_compare:w`, `#2` is the  $\langle operand \rangle$ , and `#3` is one of `<`, `=`, or `>`. As announced earlier, we leave the  $\langle operand \rangle$  for the previous conditional. If this conditional is true the result of the test is known, so we remove all tokens and return `false`. Otherwise, we apply the conditional `#1` to the  $\langle operand \rangle$  `#2` and the comparison `#3`, and call `\__int_compare:Nw` to look for additional operands, after evaluating the following expression.

```

17409 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \s__int_stop
17410  {
17411    {#3} \exp_stop_f:
17412    \prg_return_false: \else: \prg_return_true: \fi:
17413  }
17414 \cs_new:Npn \__int_compare:nnN #1#2#3
17415  {
17416    {#2} \exp_stop_f:
17417    \prg_return_false: \exp_after:wN \__int_use_none_delimit_by_s_stop:w
17418    \fi:
17419    #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
17420  }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding `\__int_compare_error:Nw`  $\langle token \rangle$  responsible for error detection.

```

17421 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
17422  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
17423 \cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
17424  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
17425 \cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
17426  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
17427 \cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
17428  { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
17429 \cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
17430  { \__int_compare:nnN { \if_int_compare:w } {#3} = }
17431 \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
17432  { \__int_compare:nnN { \if_int_compare:w } {#3} > }
17433 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
17434  { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for `\int_compare:nTF` and others. This function is documented on page 160.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```
17435 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
17436   {
17437     \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
17438       \prg_return_true:
17439     \else:
17440       \prg_return_false:
17441     \fi:
17442   }
```

(End definition for `\int_compare:nNnTF`. This function is documented on page 160.)

`\int_case:nn`  
`\int_case:mnTF`

For integer cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\tl_case:nnTF` as described in l3tl.

```
17443 \cs_new:Npn \int_case:nnTF #1
17444   {
17445     \exp:w
17446     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
17447   }
17448 \cs_new:Npn \int_case:nnT #1#2#3
17449   {
17450     \exp:w
17451     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
17452   }
17453 \cs_new:Npn \int_case:nnF #1#2
17454   {
17455     \exp:w
17456     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
17457   }
17458 \cs_new:Npn \int_case:nn #1#2
17459   {
17460     \exp:w
17461     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
17462   }
17463 \cs_new:Npn \__int_case:nnTF #1#2#3#4
17464   { \__int_case:nw {#1} #2 {#1} { } \s__int_mark {#3} \s__int_mark {#4} \s__int_stop }
17465 \cs_new:Npn \__int_case:nw #1#2#3
17466   {
17467     \int_compare:nNnTF {#1} = {#2}
17468     { \__int_case_end:nw {#3} }
17469     { \__int_case:nw {#1} }
17470   }
17471 \cs_new:Npn \__int_case_end:nw #1#2#3 \s__int_mark #4#5 \s__int_stop
17472   { \exp_end: #1 #4 }
```

(End definition for `\int_case:nnTF` and others. This function is documented on page 161.)

`\int_if_odd_p:n` A predicate function.

```
17473 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF }
17474   {
17475     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
17476       \prg_return_true:
```

```

17477     \else:
17478         \prg_return_false:
17479     \fi:
17480 }
17481 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF}
17482 {
17483     \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
17484         \prg_return_true:
17485     \else:
17486         \prg_return_false:
17487     \fi:
17488 }

```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 161.)

## 57.6 Integer expression loops

`\int_while_do:nn`  
`\int_until_do:nn`

These are quite easy given the above functions. The `while` versions test first and then execute the body. The `do_while` does it the other way round.

```

17489 \cs_new:Npn \int_while_do:nn #1#2
17490 {
17491     \int_compare:nT {#1}
17492     {
17493         #2
17494         \int_while_do:nn {#1} {#2}
17495     }
17496 }
17497 \cs_new:Npn \int_until_do:nn #1#2
17498 {
17499     \int_compare:nF {#1}
17500     {
17501         #2
17502         \int_until_do:nn {#1} {#2}
17503     }
17504 }
17505 \cs_new:Npn \int_do_while:nn #1#2
17506 {
17507     #2
17508     \int_compare:nT {#1}
17509     { \int_do_while:nn {#1} {#2} }
17510 }
17511 \cs_new:Npn \int_do_until:nn #1#2
17512 {
17513     #2
17514     \int_compare:nF {#1}
17515     { \int_do_until:nn {#1} {#2} }
17516 }

```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 162.)

`\int_while_do:nNnn`  
`\int_until_do:nNnn`  
`\int_do_while:nNnn`  
`\int_do_until:nNnn`

As above but not using the more natural syntax.

```
17517 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
```

```

17518   {
17519     \int_compare:nNnT {#1} #2 {#3}
17520     {
17521       #4
17522       \int_while_do:nNnn {#1} #2 {#3} {#4}
17523     }
17524   }
17525 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
17526   {
17527     \int_compare:nNnF {#1} #2 {#3}
17528     {
17529       #4
17530       \int_until_do:nNnn {#1} #2 {#3} {#4}
17531     }
17532   }
17533 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
17534   {
17535     #4
17536     \int_compare:nNnT {#1} #2 {#3}
17537     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
17538   }
17539 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
17540   {
17541     #4
17542     \int_compare:nNnF {#1} #2 {#3}
17543     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
17544   }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 162.)

## 57.7 Integer step functions

```

\int_step_function:nnnN
  \__int_step:wwwN
  \__int_step:NwnnN
\int_step_function:nN
\int_step_function:nnN

```

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

17545 \cs_new:Npn \int_step_function:nnnN #1#2#3
17546   {
17547     \exp_after:wN \__int_step:wwwN
17548     \int_value:w \__int_eval:w #1 \exp_after:wN ;
17549     \int_value:w \__int_eval:w #2 \exp_after:wN ;
17550     \int_value:w \__int_eval:w #3 ;
17551   }
17552 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
17553   {
17554     \int_compare:nNnTF {#2} > \c_zero_int
17555     { \__int_step:NwnnN > }
17556     {
17557       \int_compare:nNnTF {#2} = \c_zero_int
17558       {
17559         \msg_expandable_error:nnn
17560           { kernel } { zero-step } {#4}

```

```

17561           \prg_break:
17562       }
17563   { \__int_step:NwnnN < }
17564 }
17565 #1 ; {#2} {#3} #4
17566 \prg_break_point:
17567 }
17568 \cs_new:Npn \__int_step:NwnnN #1#2 ; #3#4#5
17569 {
17570     \if_int_compare:w #2 #1 #4 \exp_stop_f:
17571         \prg_break:n
17572     \fi:
17573     #5 {#2}
17574     \exp_after:wN \__int_step:Nwnnn
17575     \exp_after:wN #1
17576     \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
17577 }
17578 \cs_new:Npn \int_step_function:nN
17579   { \int_step_function:nnnN { 1 } { 1 } }
17580 \cs_new:Npn \int_step_function:nnN #1
17581   { \int_step_function:nnnN {#1} { 1 } }

```

(End definition for `\int_step_function:nnnN` and others. These functions are documented on page [163](#).)

`\int_step_inline:nn`  
`\int_step_inline:nnn`  
`\int_step_inline:nnnn`  
`\int_step_variable:nNn`  
`\int_step_variable:nnNn`  
`\int_step_variable:nnnNn`

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\int_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g_kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

17582 \cs_new_protected:Npn \int_step_inline:nn
17583   { \int_step_inline:nnnn { 1 } { 1 } }
17584 \cs_new_protected:Npn \int_step_inline:nnn #1
17585   { \int_step_inline:nnnn {#1} { 1 } }
17586 \cs_new_protected:Npn \int_step_inline:nnnn
17587   {
17588     \int_gincr:N \g_kernel_prg_map_int
17589     \exp_args:NNc \__int_step>NNnnnn
17590     \cs_gset_protected:Npn
17591     { \__int_map_ \int_use:N \g_kernel_prg_map_int :w }
17592   }
17593 \cs_new_protected:Npn \int_step_variable:nNn
17594   { \int_step_variable:nnNn { 1 } { 1 } }
17595 \cs_new_protected:Npn \int_step_variable:nnNn #1
17596   { \int_step_variable:nnNn {#1} { 1 } }
17597 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
17598   {
17599     \int_gincr:N \g_kernel_prg_map_int
17600     \exp_args:NNc \__int_step>NNnnnn
17601     \cs_gset_protected:Npx
17602     { \__int_map_ \int_use:N \g_kernel_prg_map_int :w }
17603     {#1}{#2}{#3}
17604   }

```

```

17605      \tl_set:Nn \exp_not:N #4 {##1}
17606      \exp_not:n {#5}
17607  }
17608 }
17609 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
17610 {
17611     #1 #2 ##1 {#6}
17612     \int_step_function:nnnN {#3} {#4} {#5} #2
17613     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
17614 }
```

(End definition for `\int_step_inline:nn` and others. These functions are documented on page 163.)

## 57.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```
17615 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(End definition for `\int_to_arabic:n`. This function is documented on page 164.)

`\int_to_symbols:nnn` `\__int_to_symbols:nnnn` For conversion of integers to arbitrary symbols the method is in general as follows. The input number (#1) is compared to the total number of symbols available at each place (#2). If the input is larger than the total number of symbols available then the modulus is needed, with one added so that the positions don't have to number from zero. Using an f-type expansion, this is done so that the system is recursive. The actual conversion function therefore gets a ‘nice’ number at each stage. Of course, if the initial input was small enough then there is no problem and everything is easy.

```

17616 \cs_new:Npn \int_to_symbols:nnn #1#2#3
17617 {
17618     \int_compare:nNnTF {#1} > {#2}
17619     {
17620         \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
17621         {
17622             \int_case:nn
17623                 { 1 + \int_mod:nn { #1 - 1 } {#2} }
17624                 {#3}
17625             }
17626             {#1} {#2} {#3}
17627         }
17628         { \int_case:nn {#1} {#3} }
17629     }
17630 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
17631 {
17632     \exp_args:Nf \int_to_symbols:nnn
17633     { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
17634     #1
17635 }
```

(End definition for `\int_to_symbols:nnn` and `\__int_to_symbols:nnnn`. This function is documented on page 164.)

`\int_to_alpha:n` `\int_to_Alpha:n` These both use the above function with input functions that make sense for the alphabet in English.

```

17636 \cs_new:Npn \int_to_alpha:n #1
17637   {
17638     \int_to_symbols:n {#1} { 26 }
17639   {
17640     { 1 } { a }
17641     { 2 } { b }
17642     { 3 } { c }
17643     { 4 } { d }
17644     { 5 } { e }
17645     { 6 } { f }
17646     { 7 } { g }
17647     { 8 } { h }
17648     { 9 } { i }
17649     { 10 } { j }
17650     { 11 } { k }
17651     { 12 } { l }
17652     { 13 } { m }
17653     { 14 } { n }
17654     { 15 } { o }
17655     { 16 } { p }
17656     { 17 } { q }
17657     { 18 } { r }
17658     { 19 } { s }
17659     { 20 } { t }
17660     { 21 } { u }
17661     { 22 } { v }
17662     { 23 } { w }
17663     { 24 } { x }
17664     { 25 } { y }
17665     { 26 } { z }
17666   }
17667 }
17668 \cs_new:Npn \int_to_Alph:n #1
17669   {
17670     \int_to_symbols:n {#1} { 26 }
17671   {
17672     { 1 } { A }
17673     { 2 } { B }
17674     { 3 } { C }
17675     { 4 } { D }
17676     { 5 } { E }
17677     { 6 } { F }
17678     { 7 } { G }
17679     { 8 } { H }
17680     { 9 } { I }
17681     { 10 } { J }
17682     { 11 } { K }
17683     { 12 } { L }
17684     { 13 } { M }
17685     { 14 } { N }
17686     { 15 } { O }
17687     { 16 } { P }
17688     { 17 } { Q }
17689     { 18 } { R }

```

```

17690      { 19 } { S }
17691      { 20 } { T }
17692      { 21 } { U }
17693      { 22 } { V }
17694      { 23 } { W }
17695      { 24 } { X }
17696      { 25 } { Y }
17697      { 26 } { Z }
17698  }
17699 }

```

(End definition for `\int_to_alpha:n` and `\int_to_Alpha:n`. These functions are documented on page 164.)

`\int_to_base:nn`  
`\int_to_Base:nn`  
`\_int_to_base:nn`  
`\_int_to_Base:nn`  
`\_int_to_base:nnN`  
`\_int_to_Base:nnN`  
`\_int_to_base:nnnN`  
`\_int_to_Base:nnnN`  
`\_int_to_letter:n`  
`\_int_to_Letter:n`

Converting from base ten (#1) to a second base (#2) starts with computing #1: if it is a complicated calculation, we shouldn't perform it twice. Then check the sign, store it, either - or `\c_empty_t1`, and feed the absolute value to the next auxiliary function.

```

17700 \cs_new:Npn \int_to_base:nn #1
17701   { \exp_args:Nf \_int_to_base:nn { \int_eval:n {#1} } }
17702 \cs_new:Npn \int_to_Base:nn #1
17703   { \exp_args:Nf \_int_to_Base:nn { \int_eval:n {#1} } }
17704 \cs_new:Npn \_int_to_base:nn #1#2
17705   {
17706     \int_compare:nNnTF {#1} < 0
17707       { \exp_args:No \_int_to_base:nnN { \use_none:n #1 } {#2} - }
17708       { \_int_to_base:nnN {#1} {#2} \c_empty_t1 }
17709   }
17710 \cs_new:Npn \_int_to_Base:nn #1#2
17711   {
17712     \int_compare:nNnTF {#1} < 0
17713       { \exp_args:No \_int_to_Base:nnN { \use_none:n #1 } {#2} - }
17714       { \_int_to_Base:nnN {#1} {#2} \c_empty_t1 }
17715   }

```

Here, the idea is to provide a recursive system to deal with the input. The output is built up after the end of the function. At each pass, the value in #1 is checked to see if it is less than the new base (#2). If it is, then it is converted directly, putting the sign back in front. On the other hand, if the value to convert is greater than or equal to the new base then the modulus and remainder values are found. The modulus is converted to a symbol and put on the right, and the remainder is carried forward to the next round.

```

17716 \cs_new:Npn \_int_to_base:nnN #1#2#3
17717   {
17718     \int_compare:nNnTF {#1} < {#2}
17719       { \exp_last_unbraced:Nf #3 { \_int_to_letter:n {#1} } }
17720       {
17721         \exp_args:Nf \_int_to_base:nnnN
17722           { \_int_to_letter:n { \int_mod:nn {#1} {#2} } }
17723           {#1}
17724           {#2}
17725           #3
17726       }
17727   }
17728 \cs_new:Npn \_int_to_base:nnnN #1#2#3#4
17729   {
17730     \exp_args:Nf \_int_to_base:nnN

```

```

17731     { \int_div_truncate:nn {#2} {#3} }
17732     {#3}
17733     #4
17734     #1
17735   }
17736 \cs_new:Npn \__int_to_Base:nnN #1#2#3
17737   {
17738     \int_compare:nNnTF {#1} < {#2}
17739     { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
17740     {
17741       \exp_args:Nf \__int_to_Base:nnN
17742         { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
17743         {#1}
17744         {#2}
17745         #3
17746     }
17747   }
17748 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
17749   {
17750     \exp_args:Nf \__int_to_Base:nnN
17751       { \int_div_truncate:nn {#2} {#3} }
17752       {#3}
17753       #4
17754     #1
17755   }

```

Convert to a letter only if necessary, otherwise simply return the value unchanged. It would be cleaner to use `\int_case:nn`, but in our case, the cases are contiguous, so it is forty times faster to use the `\if_case:w` primitive. The first `\exp_after:wN` expands the conditional, jumping to the correct case, the second one expands after the resulting character to close the conditional. Since `#1` might be an expression, and not directly a single digit, we need to evaluate it properly, and expand the trailing `\fi:`.

```

17756 \cs_new:Npn \__int_to_letter:n #1
17757   {
17758     \exp_after:wN \exp_after:wN
17759     \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
17760       a
17761     \or: b
17762     \or: c
17763     \or: d
17764     \or: e
17765     \or: f
17766     \or: g
17767     \or: h
17768     \or: i
17769     \or: j
17770     \or: k
17771     \or: l
17772     \or: m
17773     \or: n
17774     \or: o
17775     \or: p
17776     \or: q
17777     \or: r

```

```

17778      \or: s
17779      \or: t
17780      \or: u
17781      \or: v
17782      \or: w
17783      \or: x
17784      \or: y
17785      \or: z
17786      \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
17787      \fi:
17788  }
17789 \cs_new:Npn \__int_to_Letter:n #1
17790  {
17791      \exp_after:wN \exp_after:wN
17792      \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
17793          A
17794          \or: B
17795          \or: C
17796          \or: D
17797          \or: E
17798          \or: F
17799          \or: G
17800          \or: H
17801          \or: I
17802          \or: J
17803          \or: K
17804          \or: L
17805          \or: M
17806          \or: N
17807          \or: O
17808          \or: P
17809          \or: Q
17810          \or: R
17811          \or: S
17812          \or: T
17813          \or: U
17814          \or: V
17815          \or: W
17816          \or: X
17817          \or: Y
17818          \or: Z
17819      \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
17820      \fi:
17821  }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 165.)

`\int_to_bin:n` Wrappers around the generic function.

```

17822 \cs_new:Npn \int_to_bin:n #1
17823  { \int_to_base:nn {#1} { 2 } }
\int_to_Hex:n
17824 \cs_new:Npn \int_to_hex:n #1
17825  { \int_to_base:nn {#1} { 16 } }
\int_to_oct:n
17826 \cs_new:Npn \int_to_Hex:n #1
17827  { \int_to_Base:nn {#1} { 16 } }

```

```

17828 \cs_new:Npn \int_to_oct:n #1
17829   { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 165.)

```

\int_to_roman:n
\int_to_Roman:n
\_int_to_roman:N
\_int_to_roman:N
\_int_to_roman_i:w
\_int_to_roman_v:w
\_int_to_roman_x:w
\_int_to_roman_l:w
\_int_to_roman_c:w
\_int_to_roman_d:w
\_int_to_roman_m:w
\_int_to_roman_Q:w
\_int_to_Roman_i:w
\_int_to_Roman_v:w
\_int_to_Roman_x:w
\_int_to_Roman_l:w
\_int_to_Roman_c:w
\_int_to_Roman_d:w
\_int_to_Roman_m:w
\_int_to_Roman_Q:w

```

```

17830 \cs_new:Npn \int_to_roman:n #1
17831   {
17832     \exp_after:wN \_int_to_roman:N
17833     \_int_to_roman:w \int_eval:n {#1} Q
17834   }
17835 \cs_new:Npn \_int_to_roman:N #1
17836   {
17837     \use:c { _int_to_roman_ #1 :w }
17838     \_int_to_roman:N
17839   }
17840 \cs_new:Npn \int_to_Roman:n #1
17841   {
17842     \exp_after:wN \_int_to_Roman_aux:N
17843     \_int_to_roman:w \int_eval:n {#1} Q
17844   }
17845 \cs_new:Npn \_int_to_Roman_aux:N #1
17846   {
17847     \use:c { _int_to_Roman_ #1 :w }
17848     \_int_to_Roman_aux:N
17849   }
17850 \cs_new:Npn \_int_to_roman_i:w { i }
17851 \cs_new:Npn \_int_to_roman_v:w { v }
17852 \cs_new:Npn \_int_to_roman_x:w { x }
17853 \cs_new:Npn \_int_to_roman_l:w { l }
17854 \cs_new:Npn \_int_to_roman_c:w { c }
17855 \cs_new:Npn \_int_to_roman_d:w { d }
17856 \cs_new:Npn \_int_to_roman_m:w { m }
17857 \cs_new:Npn \_int_to_roman_Q:w #1 { }
17858 \cs_new:Npn \_int_to_Roman_i:w { I }
17859 \cs_new:Npn \_int_to_Roman_v:w { V }
17860 \cs_new:Npn \_int_to_Roman_x:w { X }
17861 \cs_new:Npn \_int_to_Roman_l:w { L }
17862 \cs_new:Npn \_int_to_Roman_c:w { C }
17863 \cs_new:Npn \_int_to_Roman_d:w { D }
17864 \cs_new:Npn \_int_to_Roman_m:w { M }
17865 \cs_new:Npn \_int_to_Roman_Q:w #1 { }

```

(End definition for `\int_to_roman:n` and others. These functions are documented on page 165.)

## 57.9 Converting from other formats to integers

```

\_int_pass_signs:wn
\_int_pass_signs_end:wn

```

Called as `\_int_pass_signs:wn` *(signs and digits)* `\s_\int_stop` {*(code)*}, this function leaves in the input stream any sign it finds, then inserts the *(code)* before the first

non-sign token (and removes `\s__int_stop`). More precisely, it deletes any `+` and passes any `-` to the input stream, hence should be called in an integer expression.

```

17866 \cs_new:Npn \__int_pass_signs:wn #1
17867 {
17868     \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
17869     \exp_after:wN \__int_pass_signs:wn
17870     \else:
17871         \exp_after:wN \__int_pass_signs_end:wn
17872         \exp_after:wN #1
17873     \fi:
17874 }
17875 \cs_new:Npn \__int_pass_signs_end:wn #1 \s__int_stop #2 { #2 #1 }

(End definition for \__int_pass_signs:wn and \__int_pass_signs_end:wn.)
```

`\int_from_alpha:n`  
`\__int_from_alpha:nN`  
`\__int_from_alpha:N`

First take care of signs then loop through the input using the `recursion quarks`. The `\__int_from_alpha:nN` auxiliary collects in its first argument the value obtained so far, and the auxiliary `\__int_from_alpha:N` converts one letter to an expression which evaluates to the correct number.

```

17876 \cs_new:Npn \int_from_alpha:n #1
17877 {
17878     \int_eval:n
17879     {
17880         \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
17881         \s__int_stop { \__int_from_alpha:nN { 0 } }
17882         \q__int_recursion_tail \q__int_recursion_stop
17883     }
17884 }
17885 \cs_new:Npn \__int_from_alpha:nN #1#2
17886 {
17887     \__int_if_recursion_tail_stop_do:Nn #2 {#1}
17888     \exp_args:Nf \__int_from_alpha:nN
17889     { \int_eval:n { #1 * 26 + \__int_from_alpha:N #2 } }
17890 }
17891 \cs_new:Npn \__int_from_alpha:N #1
17892 { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 64 } { 96 } }
```

(End definition for `\int_from_alpha:n`, `\__int_from_alpha:nN`, and `\__int_from_alpha:N`. This function is documented on page 165.)

`\int_from_base:nn`  
`\__int_from_base:nnN`  
`\__int_from_base:N`

Leave the signs into the integer expression, then loop through characters, collecting the value found so far in the first argument of `\__int_from_base:nnN`. To convert a single character, `\__int_from_base:N` checks first for digits, then distinguishes lower from upper case letters, turning them into the appropriate number. Note that this auxiliary does not use `\int_eval:n`, hence is not safe for general use.

```

17893 \cs_new:Npn \int_from_base:nn #1#2
17894 {
17895     \int_eval:n
17896     {
17897         \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
17898         \s__int_stop { \__int_from_base:nnN { 0 } {#2} }
17899         \q__int_recursion_tail \q__int_recursion_stop
17900     }
17901 }
```

```

17902 \cs_new:Npn \__int_from_base:nnN #1#2#3
17903 {
17904     \__int_if_recursion_tail_stop_do:Nn #3 {#1}
17905     \exp_args:Nf \__int_from_base:nnN
17906     { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
17907     {#2}
17908 }
17909 \cs_new:Npn \__int_from_base:N #1
17910 {
17911     \int_compare:nNnTF { '#1 } < { 58 }
17912     {#1}
17913     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
17914 }

```

(End definition for `\int_from_base:nn`, `\__int_from_base:nnN`, and `\__int_from_base:N`. This function is documented on page 166.)

`\int_from_bin:n`

`\int_from_hex:n`

```

17915 \cs_new:Npn \int_from_bin:n #1
17916     { \int_from_base:nn {#1} { 2 } }
17917 \cs_new:Npn \int_from_hex:n #1
17918     { \int_from_base:nn {#1} { 16 } }
17919 \cs_new:Npn \int_from_oct:n #1
17920     { \int_from_base:nn {#1} { 8 } }

```

(End definition for `\int_from_bin:n`, `\int_from_hex:n`, and `\int_from_oct:n`. These functions are documented on page 166.)

`\c_int_from_roman_i_int`

```

\c_int_from_roman_v_int
\c_int_from_roman_x_int
\c_int_from_roman_l_int
\c_int_from_roman_c_int
\c_int_from_roman_d_int
\c_int_from_roman_m_int
\c_int_from_roman_I_int
\c_int_from_roman_V_int
\c_int_from_roman_X_int
\c_int_from_roman_L_int
\c_int_from_roman_C_int
\c_int_from_roman_D_int
\c_int_from_roman_M_int

```

```

17921 \int_const:cn { \c_int_from_roman_i_int } { 1 }
17922 \int_const:cn { \c_int_from_roman_v_int } { 5 }
17923 \int_const:cn { \c_int_from_roman_x_int } { 10 }
17924 \int_const:cn { \c_int_from_roman_l_int } { 50 }
17925 \int_const:cn { \c_int_from_roman_c_int } { 100 }
17926 \int_const:cn { \c_int_from_roman_d_int } { 500 }
17927 \int_const:cn { \c_int_from_roman_m_int } { 1000 }
17928 \int_const:cn { \c_int_from_roman_I_int } { 1 }
17929 \int_const:cn { \c_int_from_roman_V_int } { 5 }
17930 \int_const:cn { \c_int_from_roman_X_int } { 10 }
17931 \int_const:cn { \c_int_from_roman_L_int } { 50 }
17932 \int_const:cn { \c_int_from_roman_C_int } { 100 }
17933 \int_const:cn { \c_int_from_roman_D_int } { 500 }
17934 \int_const:cn { \c_int_from_roman_M_int } { 1000 }

```

(End definition for `\c_int_from_roman_i_int` and others.)

`\int_from_roman:n`

`\__int_from_roman:NN`

`\__int_from_roman_error:w`

The method here is to iterate through the input, finding the appropriate value for each letter and building up a sum. This is then evaluated by TeX. If any unknown letter is found, skip to the closing parenthesis and insert `*0-1` afterwards, to replace the value by `-1`.

```

17935 \cs_new:Npn \int_from_roman:n #1
17936 {
17937     \int_eval:n
17938     {
17939         (

```

```

17940          0
17941          \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
17942          \q__int_recursion_tail \q__int_recursion_tail \q__int_recursion_stop
17943      )
17944  }
17945 }
17946 \cs_new:Npn \__int_from_roman:NN #1#2
17947 {
17948     \__int_if_recursion_tail_stop:N #1
17949     \int_if_exist:cF { c__int_from_roman_ #1 _int }
17950     { \__int_from_roman_error:w }
17951     \__int_if_recursion_tail_stop_do:Nn #2
17952     { + \use:c { c__int_from_roman_ #1 _int } }
17953     \int_if_exist:cF { c__int_from_roman_ #2 _int }
17954     { \__int_from_roman_error:w }
17955     \int_compare:nNnTF
17956     { \use:c { c__int_from_roman_ #1 _int } }
17957     <
17958     { \use:c { c__int_from_roman_ #2 _int } }
17959     {
17960         + \use:c { c__int_from_roman_ #2 _int }
17961         - \use:c { c__int_from_roman_ #1 _int }
17962         \__int_from_roman:NN
17963     }
17964     {
17965         + \use:c { c__int_from_roman_ #1 _int }
17966         \__int_from_roman:NN #2
17967     }
17968 }
17969 \cs_new:Npn \__int_from_roman_error:w #1 \q__int_recursion_stop #2
17970 { #2 * 0 - 1 }

```

(End definition for `\int_from_roman:n`, `\__int_from_roman:NN`, and `\__int_from_roman_error:w`. This function is documented on page 166.)

## 57.10 Viewing integer

`\int_show:N` Diagnostics.

```

\int_show:c 17971 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
\__int_show:nN 17972 \cs_generate_variant:Nn \int_show:N { c }

```

(End definition for `\int_show:N` and `\__int_show:nN`. This function is documented on page 167.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

17973 \cs_new_protected:Npn \int_show:n
17974 { \msg_show_eval:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 167.)

`\int_log:N` Diagnostics.

```

\int_log:c 17975 \cs_new_eq:NN \int_log:N \__kernel_register_log:N
17976 \cs_generate_variant:Nn \int_log:N { c }

```

(End definition for `\int_log:N`. This function is documented on page 167.)

`\int_log:n` Similar to `\int_show:n`.

```
17977 \cs_new_protected:Npn \int_log:n
17978   { \msg_log_eval:Nn \int_eval:n }
```

(End definition for `\int_log:n`. This function is documented on page 167.)

## 57.11 Random integers

`\int_rand:nn` Defined in `l3fp-random`.

(End definition for `\int_rand:nn`. This function is documented on page 166.)

## 57.12 Constant integers

`\c_zero_int` The zero is defined in `l3basics`.

```
17979 \int_const:Nn \c_zero_int { 1 }
```

(End definition for `\c_zero_int` and `\c_one_int`. These variables are documented on page 167.)

`\c_max_int` The largest number allowed is  $2^{31} - 1$

```
17980 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 167.)

`\c_max_char_int` The largest character code is 1114111 (hexadecimal 10FFFF) in X<sub>E</sub>T<sub>E</sub>X and LuaT<sub>E</sub>X and 255 in other engines. In many places pT<sub>E</sub>X and upT<sub>E</sub>X support larger character codes but for instance the values of `\lccode` are restricted to [0, 255].

```
17981 \int_const:Nn \c_max_char_int
17982   {
17983     \if_int_odd:w 0
17984       \cs_if_exist:NT \tex_luatexversion:D { 1 }
17985       \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
17986       "10FFFF
17987     \else:
17988       "FF
17989     \fi:
17990   }
```

(End definition for `\c_max_char_int`. This variable is documented on page 167.)

## 57.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

```
17991 \int_new:N \l_tmpa_int
17992 \int_new:N \l_tmpb_int
17993 \int_new:N \g_tmpa_int
17994 \int_new:N \g_tmpb_int
```

(End definition for `\l_tmpa_int` and others. These variables are documented on page 167.)

## 57.14 Integers for earlier modules

<@@=seq>

```
\l__int_internal_a_int
\l__int_internal_b_int
 17995 \int_new:N \l__int_internal_a_int
 17996 \int_new:N \l__int_internal_b_int
(End definition for \l__int_internal_a_int and \l__int_internal_b_int.)
 17997 ⟨/package⟩
```

# Chapter 58

## I3flag implementation

```
17998  <*package>
17999  <@@=flag>
```

*The following test files are used for this code: m3flag001.*

### 58.1 Non-expandable flag commands

The height  $h$  of a flag (initially zero) is stored by setting control sequences of the form  $\text{\flag } \langle\text{name}\rangle \langle\text{integer}\rangle$  to  $\text{\relax}$  for  $0 \leq \langle\text{integer}\rangle < h$ . When a flag is raised, a “trap” function  $\text{\flag } \langle\text{name}\rangle$  is called. The existence of this function is also used to test for the existence of a flag.

**\flag\_new:n** For each flag, we define a “trap” function, which by default simply increases the flag by 1 by letting the appropriate control sequence to  $\text{\relax}$ . This can be done expandably!

```
18000 \cs_new_protected:Npn \flag_new:n #1
18001  {
18002    \cs_new:cpn { flag~#1 } ##1 ;
18003    { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
18004 }
```

(End definition for  $\text{\flag\_new:n}$ . This function is documented on page 170.)

**\flag\_clear:n**  $\_\_$ **\flag\_clear:wn** Undefine control sequences, starting from the 0 flag, upwards, until reaching an undefined control sequence. We don’t use  $\text{\cs_undefine:c}$  because that would act globally. When the option `check-declarations` is used, check for the function defined by  $\text{\flag_new:n}$ .

```
18005 \cs_new_protected:Npn \flag_clear:n #1 { \_\_flag_clear:wn 0 ; {#1} }
18006 \cs_new_protected:Npn \_\_flag_clear:wn #1 ; #2
18007  {
18008    \if_cs_exist:w flag~#2~#1 \cs_end:
18009      \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
18010      \exp_after:wN \_\_flag_clear:wn
18011      \int_value:w \int_eval:w 1 + #1
18012    \else:
18013      \use_i:nnn
18014    \fi:
18015    ; {#2}
18016 }
```

(End definition for `\flag_clear:n` and `\_flag_clear:wn`. This function is documented on page 170.)

`\flag_clear_new:n` As for other datatypes, clear the `\flag` or create a new one, as appropriate.

```
18017 \cs_new_protected:Npn \flag_clear_new:n #1
18018 { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }
```

(End definition for `\flag_clear_new:n`. This function is documented on page 171.)

`\flag_show:n` Show the height (terminal or log file) using appropriate l3msg auxiliaries.

```
\flag_log:n
\_flag_show:Nn
18019 \cs_new_protected:Npn \flag_show:n { \_flag_show:Nn \tl_show:n }
18020 \cs_new_protected:Npn \flag_log:n { \_flag_show:Nn \tl_log:n }
18021 \cs_new_protected:Npn \_flag_show:Nn #1#2
18022 {
18023     \exp_args:Nc \_kernel_chk_defined:NT { flag~#2 }
18024     {
18025         \exp_args:Nx #1
18026         { \tl_to_str:n { flag~#2-height } = \flag_height:n {#2} }
18027     }
18028 }
```

(End definition for `\flag_show:n`, `\flag_log:n`, and `\_flag_show:Nn`. These functions are documented on page 171.)

## 58.2 Expandable flag commands

`\flag_if_exist_p:n` A flag exist if the corresponding trap `\flag <flag name>:n` is defined.

```
\flag_if_exist:nTF
18029 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
18030 {
18031     \cs_if_exist:cTF { flag~#1 }
18032     { \prg_return_true: } { \prg_return_false: }
18033 }
```

(End definition for `\flag_if_exist:nTF`. This function is documented on page 171.)

`\flag_if_raised_p:n` Test if the flag has a non-zero height, by checking the 0 control sequence.

```
\flag_if_raised:nTF
18034 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
18035 {
18036     \if_cs_exist:w flag~#1~0 \cs_end:
18037     \prg_return_true:
18038     \else:
18039     \prg_return_false:
18040     \fi:
18041 }
```

(End definition for `\flag_if_raised:nTF`. This function is documented on page 171.)

`\flag_height:n` Extract the value of the flag by going through all of the control sequences starting from 0.

```
\_flag_height_loop:wn
\_flag_height_end:wn
18042 \cs_new:Npn \flag_height:n #1 { \_flag_height_loop:wn 0; {#1} }
18043 \cs_new:Npn \_flag_height_loop:wn #1 ; #2
18044 {
18045     \if_cs_exist:w flag~#2~#1 \cs_end:
18046     \exp_after:wN \_flag_height_loop:wn \int_value:w \int_eval:w 1 +
18047     \else:
```

```

18048      \exp_after:wN \__flag_height_end:wn
18049      \fi:
18050      #1 ; {#2}
18051  }
18052 \cs_new:Npn \__flag_height_end:wn #1 ; #2 {#1}

```

(End definition for `\flag_height:n`, `\__flag_height_loop:wn`, and `\__flag_height_end:wn`. This function is documented on page [171](#).)

**\flag\_raise:n** Simply apply the trap to the height, after expanding the latter.

```

18053 \cs_new:Npn \flag_raise:n #1
18054 {
18055   \cs:w flag~#1 \exp_after:wN \cs_end:
18056   \int_value:w \flag_height:n {#1} ;
18057 }

```

(End definition for `\flag_raise:n`. This function is documented on page [171](#).)

18058 ⟨/package⟩

# Chapter 59

## I3clist implementation

The following test files are used for this code: *m3clist002*.

18059 ⟨\*package⟩

18060 ⟨@=clist⟩

**\c\_empty\_clist** An empty comma list is simply an empty token list.

18061 \cs\_new\_eq:NN \c\_empty\_clist \c\_empty\_tl

(End definition for *\c\_empty\_clist*. This variable is documented on page 181.)

**\l\_\_clist\_internal\_clist** Scratch space for various internal uses. This comma list variable cannot be declared as such because it comes before *\clist\_new:N*

18062 \tl\_new:N \l\_\_clist\_internal\_clist

(End definition for *\l\_\_clist\_internal\_clist*.)

**\s\_\_clist\_mark** Internal scan marks.

**\s\_\_clist\_stop**

18063 \scan\_new:N \s\_\_clist\_mark

18064 \scan\_new:N \s\_\_clist\_stop

(End definition for *\s\_\_clist\_mark* and *\s\_\_clist\_stop*.)

**\\_clist\_use\_none\_delimit\_by\_s\_mark:w** Functions to gobble up to a scan mark.

18065 \cs\_new:Npn \\_clist\_use\_none\_delimit\_by\_s\_mark:w #1 \s\_\_clist\_mark { }

18066 \cs\_new:Npn \\_clist\_use\_none\_delimit\_by\_s\_stop:w #1 \s\_\_clist\_stop { }

18067 \cs\_new:Npn \\_clist\_use\_i\_delimit\_by\_s\_stop:nw #1 #2 \s\_\_clist\_stop {#1}

(End definition for *\\_clist\_use\_none\_delimit\_by\_s\_mark:w*, *\\_clist\_use\_none\_delimit\_by\_s\_stop:w*, and *\\_clist\_use\_i\_delimit\_by\_s\_stop:nw*.)

**\\_clist\_tmp:w** A temporary function for various purposes.

18068 \cs\_new\_protected:Npn \\_clist\_tmp:w { }

(End definition for *\\_clist\_tmp:w*.)

## 59.1 Removing spaces around items

`\__clist_trim_next:w` Called as `\exp:w \__clist_trim_next:w \prg_do_nothing: <comma list> ...` it expands to `{(trimmed item)}` where the `<trimmed item>` is the first non-empty result from removing spaces from both ends of comma-delimited items in the `<comma list>`. The `\prg_do_nothing:` marker avoids losing braces. The test for blank items is a somewhat optimized `\tl_if_empty:oTF` construction; if blank, another item is sought, otherwise trim spaces.

```

18069 \cs_new:Npn \__clist_trim_next:w #1 ,
18070   {
18071     \tl_if_empty:oTF { \use_none:nn #1 ? }
18072     { \__clist_trim_next:w \prg_do_nothing: }
18073     { \tl_trim_spaces_apply:oN {#1} \exp_end: }
18074   }

```

(End definition for `\__clist_trim_next:w`.)

`\__clist_sanitize:n` The auxiliary `\__clist_sanitize:Nn` receives a delimiter (`\c_empty_tl` the first time, afterwards a comma) and that item as arguments. Unless we are done with the loop it calls `\__clist_wrap_item:w` to unbrace the item (using a comma delimiter is safe since `#2` came from removing spaces from an argument delimited by a comma) and possibly re-brace it if needed.

```

18075 \cs_new:Npn \__clist_sanitize:n #1
18076   {
18077     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
18078     \exp:w \__clist_trim_next:w \prg_do_nothing:
18079     #1 , \s__clist_stop \prg_break: , \prg_break_point:
18080   }
18081 \cs_new:Npn \__clist_sanitize:Nn #1#2
18082   {
18083     \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18084     #1 \__clist_wrap_item:w #2 ,
18085     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
18086     \exp:w \__clist_trim_next:w \prg_do_nothing:
18087   }

```

(End definition for `\__clist_sanitize:n` and `\__clist_sanitize:Nn`.)

`\__clist_if_wrap:nTF` True if the argument must be wrapped to avoid getting altered by some clist operations.  
`\__clist_if_wrap:w` That is the case whenever the argument

- starts or end with a space or contains a comma,
- is empty, or
- consists of a single braced group.

If the argument starts or ends with a space or contains a comma then one of the three arguments of `\__clist_if_wrap:w` will have its end delimiter (partly) in one of the three copies of `#1` in `\__clist_if_wrap:nTF`; this has a knock-on effect meaning that the result of the expansion is not empty; in that case, wrap. Otherwise, the argument is safe unless it starts with a brace group (or is empty) and it is empty or consists of a single `n`-type argument.

```
18088 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
```

```

18089    {
18090      \tl_if_empty:oTF
18091      {
18092        \__clist_if_wrap:w
18093        \s__clist_mark ? #1 ~ \s__clist_mark ? ~ #1
18094        \s__clist_mark , ~ \s__clist_mark #1 ,
18095      }
18096      {
18097        \tl_if_head_is_group:nTF { #1 { } }
18098        {
18099          \tl_if_empty:nTF {#1}
18100          { \prg_return_true: }
18101          {
18102            \tl_if_empty:oTF { \use_none:n #1}
18103            { \prg_return_true: }
18104            { \prg_return_false: }
18105          }
18106        }
18107        { \prg_return_false: }
18108      }
18109      { \prg_return_true: }
18110    }
18111 \cs_new:Npn \__clist_if_wrap:w #1 \s__clist_mark ? ~ #2 ~ \s__clist_mark #3 , { }

(End definition for \__clist_if_wrap:nTF and \__clist_if_wrap:w.)
```

\\_\_clist\_wrap\_item:w Safe items are put in \exp\_not:n, otherwise we put an extra set of braces.

```

18112 \cs_new:Npn \__clist_wrap_item:w #1 ,
18113   { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }
```

(End definition for \\_\_clist\_wrap\_item:w.)

## 59.2 Allocation and initialisation

**\clist\_new:N** Internally, comma lists are just token lists.

```

\clist_new:c
  18114 \cs_new_eq:NN \clist_new:N \tl_new:N
  18115 \cs_new_eq:NN \clist_new:c \tl_new:c
```

(End definition for \clist\_new:N. This function is documented on page 173.)

**\clist\_const:Nn** Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

\clist_const:Nx
  18116 \cs_new_protected:Npn \clist_const:Nn #1#2
  18117   { \tl_const:Nx #1 { \__clist_sanitize:n {#2} } }
  18118 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }
```

(End definition for \clist\_const:Nn. This function is documented on page 173.)

**\clist\_clear:N** Clearing comma lists is just the same as clearing token lists.

```

\clist_clear:c
  18119 \cs_new_eq:NN \clist_clear:N \tl_clear:N
\clist_gclear:N
  18120 \cs_new_eq:NN \clist_clear:c \tl_clear:c
\clist_gclear:c
  18121 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
  18122 \cs_new_eq:NN \clist_gclear:c \tl_gclear:c
```

(End definition for `\clist_clear:N` and `\clist_gclear:N`. These functions are documented on page 173.)

`\clist_clear_new:N` Once again a copy from the token list functions.  
18123 `\cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N`  
`\clist_gclear_new:N` 18124 `\cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c`  
`\clist_gclear_new:c` 18125 `\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N`  
18126 `\cs_new_eq:NN \clist_gclear_new:c \tl_gclear_new:c`

(End definition for `\clist_clear_new:N` and `\clist_gclear_new:N`. These functions are documented on page 173.)

`\clist_set_eq:NN` Once again, these are simple copies from the token list functions.  
18127 `\cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN`  
`\clist_set_eq:Nc` 18128 `\cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc`  
`\clist_set_eq:cc` 18129 `\cs_new_eq:NN \clist_set_eq:cN \tl_set_eq:cN`  
`\clist_gset_eq:NN` 18130 `\cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc`  
`\clist_gset_eq:cN` 18131 `\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN`  
`\clist_gset_eq:Nc` 18132 `\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc`  
`\clist_gset_eq:cc` 18133 `\cs_new_eq:NN \clist_gset_eq:cN \tl_gset_eq:cN`  
18134 `\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc`

(End definition for `\clist_set_eq:NN` and `\clist_gset_eq:NN`. These functions are documented on page 173.)

`\clist_set_from_seq:NN` Setting a comma list from a comma-separated list is done using a simple mapping. Safe items are put in `\exp_not:n`, otherwise we put an extra set of braces. The first comma must be removed, except in the case of an empty comma-list.

18135 `\cs_new_protected:Npn \clist_set_from_seq:NN`  
18136 { `\__clist_set_from_seq:NNNN \clist_clear:N \__kernel_tl_set:Nx` }  
18137 `\cs_new_protected:Npn \clist_gset_from_seq:NN`  
18138 { `\__clist_set_from_seq:NNNN \clist_gclear:N \__kernel_tl_gset:Nx` }  
18139 `\cs_new_protected:Npn \__clist_set_from_seq:NNNN #1#2#3#4`  
18140 {  
18141 `\seq_if_empty:NTF #4`  
18142 { `#1 #3` }  
18143 {  
18144 `#2 #3`  
18145 {  
18146 `\exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w`  
18147 `\seq_map_function:NN #4 \__clist_set_from_seq:n`  
18148 }  
18149 }  
18150 }  
18151 `\cs_new:Npn \__clist_set_from_seq:n #1`  
18152 {  
18153 ,  
18154 `\__clist_if_wrap:nTF {#1}`  
18155 { `\exp_not:n { {#1} }` }  
18156 { `\exp_not:n { {#1} }` }  
18157 }  
18158 `\cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }`  
18159 `\cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }`  
18160 `\cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }`  
18161 `\cs_generate_variant:Nn \clist_gset_from_seq:NN { c , cc }`

(End definition for `\clist_set_from_seq:NN` and others. These functions are documented on page 173.)

`\clist_concat:NNN` Concatenating comma lists is not quite as easy as it seems, as there needs to be the correct addition of a comma to the output. So a little work to do.

```
18162 \cs_new_protected:Npn \clist_concat:NNN
18163   { \__clist_concat:NNNN \__kernel_tl_set:Nx }
18164 \cs_new_protected:Npn \clist_gconcat:NNN
18165   { \__clist_concat:NNNN \__kernel_tl_gset:Nx }
18166 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
18167   {
18168     #1 #2
18169   {
18170     \exp_not:o #3
18171     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
18172     \exp_not:o #4
18173   }
18174 }
18175 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
18176 \cs_generate_variant:Nn \clist_gconcat:NNN { ccc }
```

(End definition for `\clist_concat:NNN`, `\clist_gconcat:NNN`, and `\__clist_concat:NNNN`. These functions are documented on page 174.)

`\clist_if_exist_p:N` Copies of the `cs` functions defined in l3basics.

```
18177 \prg_new_eq_conditional:NNn \clist_if_exist:N \cs_if_exist:N
18178   { TF , T , F , p }
18179 \prg_new_eq_conditional:NNn \clist_if_exist:c \cs_if_exist:c
18180   { TF , T , F , p }
```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 174.)

## 59.3 Adding data to comma lists

`\clist_set:Nn`  
`\clist_set:NV`  
`\clist_set:No`  
`\clist_set:Nx`  
`\clist_set:cn`  
`\clist_set:cV`  
`\clist_set:co`  
`\clist_set:cx`  
`\clist_gset:Nn`  
`\clist_lpat_gset:NW`  
`\clist_lpat_gset:Nv`  
`\clist_lpat_gset:Nx`  
`\clist_lpat_gset:Nn`  
`\clist_lpat_gset:CW`  
`\clist_lpat_gset:cV`  
`\clist_lpat_gset:cS`  
`\clist_put_left:Nn`  
`\clist_gput_left:NV`  
`\clist_gput_left:No`  
`\clist_gput_left:Nx`  
`\clist_gput_left:cn`  
`\clist_gput_left:cV`  
`\clist_gput_left:co`  
`\clist_gput_left:cx`  
`\__clist_put_left>NNNn`  
(End definition for `\clist_set:Nn` and `\clist_gset:Nn`. These functions are documented on page 174.)

`\clist_lpat_gset:NW` Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```
18181 \cs_new_protected:Npn \clist_set:Nn #1#2
18182   { \__kernel_tl_set:Nx #1 { \__clist_sanitize:n {#2} } }
18183 \cs_new_protected:Npn \clist_gset:Nn #1#2
18184   { \__kernel_tl_gset:Nx #1 { \__clist_sanitize:n {#2} } }
18185 \cs_generate_variant:Nn \clist_set:Nn { NV , No , Nx , c , cV , co , cx }
18186 \cs_generate_variant:Nn \clist_gset:Nn { NV , No , Nx , c , cV , co , cx }
18187 \cs_new_protected:Npn \clist_put_left:Nn
18188   { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
18189 \cs_new_protected:Npn \clist_gput_left:Nn
18190   { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
18191 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
18192   {
18193     #2 \l__clist_internal_clist {#4}
18194     #1 #3 \l__clist_internal_clist #3
```

```

18195      }
18196  \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
18197  \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
18198  \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
18199  \cs_generate_variant:Nn \clist_gput_left:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_left:Nn`, `\clist_gput_left:Nn`, and `\_clist_put_left:NNNn`. These functions are documented on page 174.)

```

\clist_put_right:Nn
\clist_put_right:NV
\clist_put_right:No
\clist_put_right:Nx
\clist_put_right:cN
\clist_put_right:cV
\clist_put_right:co
\clist_put_right:cx
\clist_gput_right:Nn
\clist_gput_right:NV
\clist_gput_right:No
\clist_gput_right:Nx
\clist_gput_right:cN
\clist_gput_right:cV
\clist_gput_right:co
\clist_gput_right:cx
\_\clist_put_right:NNNn

```

```

18200 \cs_new_protected:Npn \clist_put_right:Nn
18201   { \_clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
18202 \cs_new_protected:Npn \clist_gput_right:Nn
18203   { \_clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
18204 \cs_new_protected:Npn \_clist_put_right:NNNn #1#2#3#4
18205   {
18206     #2 \l__clist_internal_clist {#4}
18207     #1 #3 #3 \l__clist_internal_clist
18208   }
18209 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
18210 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
18211 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
18212 \cs_generate_variant:Nn \clist_gput_right:Nn { c , cV , co , cx }

```

(End definition for `\clist_put_right:Nn`, `\clist_gput_right:Nn`, and `\_clist_put_right:NNNn`. These functions are documented on page 174.)

## 59.4 Comma lists as stacks

```

\clist_get:NN
\clist_get:cN
\_\clist_get:wN

```

Getting an item from the left of a comma list is pretty easy: just trim off the first item using the comma. No need to trim spaces as comma-list *variables* are assumed to have “cleaned-up” items. (Note that grabbing a comma-delimited item removes an outer pair of braces if present, exactly as needed to uncover the underlying item.)

```

18213 \cs_new_protected:Npn \clist_get:NN #1#2
18214   {
18215     \if_meaning:w #1 \c_empty_clist
18216       \tl_set:Nn #2 { \q_no_value }
18217     \else:
18218       \exp_after:wN \_\clist_get:wN #1 , \s__clist_stop #2
18219     \fi:
18220   }
18221 \cs_new_protected:Npn \_\clist_get:wN #1 , #2 \s__clist_stop #3
18222   { \tl_set:Nn #3 {#1} }
18223 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `\_clist_get:wN`. This function is documented on page 179.)

```

\clist_pop:NN
\clist_pop:cN
\clist_gpop:NN
\clist_gpop:cN
\_\clist_pop:NNN
\_\clist_pop:wwNNN
\_\clist_pop:wN

```

An empty clist leads to `\q_no_value`, otherwise grab until the first comma and assign to the variable. The second argument of `\_clist_pop:wwNNN` is a comma list ending in a comma and `\s__clist_mark`, unless the original clist contained exactly one item: then the argument is just `\s__clist_mark`. The next auxiliary picks either `\exp_not:n` or `\use_none:n` as #2, ensuring that the result can safely be an empty comma list.

```

18224 \cs_new_protected:Npn \clist_pop:NN

```

```

18225 { \__clist_pop:NNN \__kernel_tl_set:Nx }
18226 \cs_new_protected:Npn \clist_gpop:NN
18227 { \__clist_pop:NNN \__kernel_tl_gset:Nx }
18228 \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
18229 {
18230     \if_meaning:w #2 \c_empty_clist
18231         \tl_set:Nn #3 { \q_no_value }
18232     \else:
18233         \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
18234     \fi:
18235 }
18236 \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \s__clist_stop #3#4#5
18237 {
18238     \tl_set:Nn #5 {#1}
18239     #3 #4
18240     {
18241         \__clist_pop:wN \prg_do_nothing:
18242             #2 \exp_not:o
18243                 , \s__clist_mark \use_none:n
18244                 \s__clist_stop
18245             }
18246     }
18247 \cs_new:Npn \__clist_pop:wN #1 , \s__clist_mark #2 #3 \s__clist_stop { #2 {#1} }
18248 \cs_generate_variant:Nn \clist_pop:NN { c }
18249 \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and others. These functions are documented on page 179.)

**`\clist_get:NNTF`** The same, as branching code: very similar to the above.

```

\clist_get:cNTF
18250 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
\clist_pop:NNTF
18251 {
\clist_pop:cNTF
18252     \if_meaning:w #1 \c_empty_clist
\clist_gpop:NNTF
18253     \prg_return_false:
\clist_gpop:cNTF
18254     \else:
\__clist_pop_TF:NNN
18255         \exp_after:wN \__clist_get:wN #1 , \s__clist_stop #2
18256         \prg_return_true:
18257     \fi:
18258 }
\prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
18259 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
18260     { \__clist_pop_TF:NNN \__kernel_tl_set:Nx #1 #2 }
18261     \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
18262     { \__clist_pop_TF:NNN \__kernel_tl_gset:Nx #1 #2 }
18263 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
18264 {
18265     \if_meaning:w #2 \c_empty_clist
18266         \prg_return_false:
18267     \else:
18268         \exp_after:wN \__clist_pop:wwNNN #2 , \s__clist_mark \s__clist_stop #1#2#3
18269         \prg_return_true:
18270     \fi:
18271 }
\prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
18272 \prg_generate_conditional_variant:Nnn \clist_gpop:NN { c } { T , F , TF }

```

(End definition for `\clist_get:NNTF` and others. These functions are documented on page 179.)

```
\clist_push:Nn Pushing to a comma list is the same as adding on the left.  
18275 \cs_new_eq:NN \clist_push:Nn \clist_put_left:Nn  
18276 \cs_new_eq:NN \clist_push:NV \clist_put_left:NV  
18277 \cs_new_eq:NN \clist_push:No \clist_put_left:No  
18278 \cs_new_eq:NN \clist_push:Nx \clist_put_left:Nx  
18279 \cs_new_eq:NN \clist_push:cn \clist_put_left:cn  
18280 \cs_new_eq:NN \clist_push:cV \clist_put_left:cV  
18281 \cs_new_eq:NN \clist_push:co \clist_put_left:co  
18282 \cs_new_eq:NN \clist_push:cx \clist_put_left:cx  
18283 \cs_new_eq:NN \clist_gpush:Nn \clist_gput_left:Nn  
18284 \cs_new_eq:NN \clist_gpush:NV \clist_gput_left:NV  
18285 \cs_new_eq:NN \clist_gpush:No \clist_gput_left:No  
18286 \cs_new_eq:NN \clist_gpush:Nx \clist_gput_left:Nx  
18287 \cs_new_eq:NN \clist_gpush:cn \clist_gput_left:cn  
18288 \cs_new_eq:NN \clist_gpush:cV \clist_gput_left:cV  
18289 \cs_new_eq:NN \clist_gpush:co \clist_gput_left:co  
18290 \cs_new_eq:NN \clist_gpush:cx \clist_gput_left:cx
```

(End definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 180.)

## 59.5 Modifying comma lists

An internal comma list and a sequence for the removal routines.

```
\l__clist_internal_remove_clist  
\l__clist_internal_remove_seq  
18291 \clist_new:N \l__clist_internal_remove_clist  
18292 \seq_new:N \l__clist_internal_remove_seq
```

(End definition for `\l__clist_internal_remove_clist` and `\l__clist_internal_remove_seq`.)

```
\clist_remove_duplicates:N Removing duplicates means making a new list then copying it.  
\clist_remove_duplicates:c  
\clist_gremove_duplicates:N  
\clist_gremove_duplicates:c  
  \l__clist_remove_duplicates:NN  
18293 \cs_new_protected:Npn \clist_remove_duplicates:N  
18294 { \l__clist_remove_duplicates:NN \clist_set_eq:NN }  
18295 \cs_new_protected:Npn \clist_gremove_duplicates:N  
18296 { \l__clist_remove_duplicates:NN \clist_gset_eq:NN }  
18297 \cs_new_protected:Npn \l__clist_remove_duplicates:NN #1#2  
18298 {  
18299   \clist_clear:N \l__clist_internal_remove_clist  
18300   \clist_map_inline:Nn #2  
18301   {  
18302     \clist_if_in:NnF \l__clist_internal_remove_clist {##1}  
18303     {  
18304       \tl_put_right:Nx \l__clist_internal_remove_clist  
18305       {  
18306         \clist_if_empty:NF \l__clist_internal_remove_clist { , }  
18307         \l__clist_if_wrap:nTF {##1} { \exp_not:n {##1} } { \exp_not:n {##1} }  
18308       }  
18309     }  
18310   }  
18311   #1 #2 \l__clist_internal_remove_clist  
18312 }  
18313 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }  
18314 \cs_generate_variant:Nn \clist_gremove_duplicates:N { c }
```

(End definition for `\clist_remove_duplicates:N`, `\clist_gremove_duplicates:N`, and `\_clist_remove_duplicates>NN`. These functions are documented on page 175.)

```
\clist_remove_all:Nn
\clist_remove_all:cn
\clist_remove_all:NV
\clist_remove_all:cV
\clist_gremove_all:Nn
\clist_gremove_all:cn
\clist_gremove_all:NV
\clist_gremove_all:cV
\__clist_remove_all:NNNn
\__clist_remove_all:w
```

The method used here for safe items is very similar to `\tl_replace_all:Nnn`. However, if the item contains commas or leading/trailing spaces, or is empty, or consists of a single brace group, we know that it can only appear within braces so the code would fail; instead just convert to a sequence and do the removal with `\bseq` code (it involves somewhat elaborate code to do most of the work expandably but the final token list comparisons non-expandably).

For “safe” items, build a function delimited by the `<item>` that should be removed, surrounded with commas, and call that function followed by the expanded comma list, and another copy of the `<item>`. The loop is controlled by the argument grabbed by `\__clist_remove_all:w`: when the item was found, the `\s__clist_mark` delimiter used is the one inserted by `\__clist_tmp:w`, and `\__clist_use_none_delimit_by_s_stop:w` is deleted. At the end, the final `<item>` is grabbed, and the argument of `\__clist_tmp:w` contains `\s__clist_mark`: in that case, `\__clist_remove_all:w` removes the second `\s__clist_mark` (inserted by `\__clist_tmp:w`), and lets `\__clist_use_none_delimit_by_s_stop:w` act.

No brace is lost because items are always grabbed with a leading comma. The result of the first assignment has an extra leading comma, which we remove in a second assignment. Two exceptions: if the clist lost all of its elements, the result is empty, and we shouldn’t remove anything; if the clist started up empty, the first step happens to turn it into a single comma, and the second step removes it.

```
18315 \cs_new_protected:Npn \clist_remove_all:Nn
18316   { \__clist_remove_all:NNNn \clist_set_from_seq:NN \__kernel_tl_set:Nx }
18317   \cs_new_protected:Npn \clist_gremove_all:Nn
18318   { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \__kernel_tl_gset:Nx }
18319   \cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
18320   {
18321     \__clist_if_wrap:nTF {#4}
18322     {
18323       \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
18324       \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
18325       #1 #3 \l__clist_internal_remove_seq
18326     }
18327     {
18328       \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
18329       {
18330         ##1
18331         , \s__clist_mark , \__clist_use_none_delimit_by_s_stop:w ,
18332         \__clist_remove_all:
18333       }
18334       #2 #3
18335       {
18336         \exp_after:wN \__clist_remove_all:
18337         #3 , \s__clist_mark , #4 , \s__clist_stop
18338       }
18339       \clist_if_empty:NF #3
18340       {
18341         #2 #3
18342         {
18343           \exp_args:No \exp_not:o
```

```

18344           { \exp_after:wN \use_none:n #3 }
18345       }
18346   }
18347 }
18348 }
18349 \cs_new:Npn \__clist_remove_all:
18350   { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
18351 \cs_new:Npn \__clist_remove_all:w #1 , \s__clist_mark , #2 , { \exp_not:n {#1} }
18352 \cs_generate_variant:Nn \clist_remove_all:Nn { c , NV , cV }
18353 \cs_generate_variant:Nn \clist_gremove_all:Nn { c , NV , cV }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 175.)

**\clist\_reverse:N** Use `\clist_reverse:n` in an x-expanding assignment. The extra work that `\clist_reverse:n` does to preserve braces and spaces would not be needed for the well-controlled case of N-type comma lists, but the slow-down is not too bad.

**\clist\_greverse:c**

```

18354 \cs_new_protected:Npn \clist_reverse:N #1
18355   { \__kernel_tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
18356 \cs_new_protected:Npn \clist_greverse:N #1
18357   { \__kernel_tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
18358 \cs_generate_variant:Nn \clist_reverse:N { c }
18359 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 175.)

**\clist\_reverse:n** The reversed token list is built one item at a time, and stored between `\s__clist_stop` and `\s__clist_mark`, in the form of ? followed by zero or more instances of “`\item`”,. We start from a comma list “`\item_1`, ..., `\item_n`”. During the loop, the auxiliary `\__clist_reverse:wwNww` receives “?`\item_i`” as #1, “`\item_{i+1}`, ..., `\item_n`” as #2, `\__clist_reverse:wwNww` as #3, what remains until `\s__clist_stop` as #4, and “`\item_{i-1}`, ..., `\item_1`,” as #5. The auxiliary moves #1 just before #5, with a comma, and calls itself (#3). After the last item is moved, `\__clist_reverse:wwNww` receives “`\s__clist_mark \__clist_reverse:wwNww !`” as its argument #1, thus `\__clist_reverse_end:ww` as its argument #3. This second auxiliary cleans up until the marker !, removes the trailing comma (introduced when the first item was moved after `\s__clist_stop`), and leaves its argument #1 within `\exp_not:n`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

18360 \cs_new:Npn \clist_reverse:n #1
18361   {
18362     \__clist_reverse:wwNww ? #1 ,
18363     \s__clist_mark \__clist_reverse:wwNww ! ,
18364     \s__clist_mark \__clist_reverse_end:ww
18365     \s__clist_stop ? \s__clist_mark
18366   }
18367 \cs_new:Npn \__clist_reverse:wwNww
18368   #1 , #2 \s__clist_mark #3 #4 \s__clist_stop ? #5 \s__clist_mark
18369   { #3 ? #2 \s__clist_mark #3 #4 \s__clist_stop #1 , #5 \s__clist_mark }
18370 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \s__clist_mark
18371   { \exp_not:o { \use_none:n #2 } }

```

(End definition for `\clist_reverse:n`, `\__clist_reverse:wwNww`, and `\__clist_reverse_end:ww`. This function is documented on page 175.)

`\clist_sort:Nn` Implemented in `\l3sort`.  
`\clist_sort:cn`  
`\clist_gsort:Nn`  
`\clist_gsort:cn`  
*(End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 176.)*

## 59.6 Comma list conditionals

`\clist_if_empty_p:N` Simple copies from the token list variable material.

```

18372 \prg_new_eq_conditional:Nnn \clist_if_empty:N \tl_if_empty:N
18373   { p , T , F , TF }
18374 \prg_new_eq_conditional:Nnn \clist_if_empty:c \tl_if_empty:c
18375   { p , T , F , TF }

```

*(End definition for `\clist_if_empty:NTF`. This function is documented on page 176.)*

`\clist_if_empty_p:n`  
`\clist_if_empty:nTF`  
`\__clist_if_empty_n:w`  
`\__clist_if_empty_n:wNw`  
As usual, we insert a token (here `?`) before grabbing any argument: this avoids losing braces. The argument of `\tl_if_empty:oTF` is empty if #1 is `?` followed by blank spaces (besides, this particular variant of the emptiness test is optimized). If the item of the comma list is blank, grab the next one. As soon as one item is non-blank, exit: the second auxiliary grabs `\prg_return_false:` as #2, unless every item in the comma list was blank and the loop actually got broken by the trailing `\s__clist_stop \prg_return_false:` item.

```

18376 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
18377   {
18378     \__clist_if_empty_n:w ? #1
18379     , \s__clist_stop \prg_return_false:
18380     , \s__clist_stop \prg_return_true:
18381     \s__clist_stop
18382   }
18383 \cs_new:Npn \__clist_if_empty_n:w #1 ,
18384   {
18385     \tl_if_empty:oTF { \use_none:nn #1 ? }
18386     { \__clist_if_empty_n:w ? }
18387     { \__clist_if_empty_n:wNw }
18388   }
18389 \cs_new:Npn \__clist_if_empty_n:wNw #1 \s__clist_stop #2#3 \s__clist_stop {#2}

```

*(End definition for `\clist_if_empty:nTF`, `\__clist_if_empty_n:w`, and `\__clist_if_empty_n:wNw`. This function is documented on page 176.)*

`\clist_if_in:NnTF` For “safe” items, we simply surround the comma list, and the item, with commas, then  
`\clist_if_in:NVT` use the same code as for `\tl_if_in:Nn`. For “unsafe” items we follow the same route as  
`\clist_if_in:NotF` `\seq_if_in:Nn`, mapping through the list a comparison function. If found, return `true`  
`\clist_if_in:cNTF` and remove `\prg_return_false:`.

```

18390 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
18391   {
18392     \exp_args:No \__clist_if_in_return:nnN #1 {#2} #1
18393   }
18394 \prg_new_protected_conditional:Npnn \clist_if_in:nn #1#2 { T , F , TF }
18395   {
18396     \clist_set:Nn \l__clist_internal_clist {#1}
18397     \exp_args:No \__clist_if_in_return:nnN \l__clist_internal_clist {#2}
18398     \l__clist_internal_clist

```

```

18399    }
18400  \cs_new_protected:Npn \__clist_if_in_return:nnN #1#2#3
18401  {
18402      \__clist_if_wrap:nTF {#2}
18403      {
18404          \cs_set:Npx \__clist_tmp:w ##1
18405          {
18406              \exp_not:N \tl_if_eq:nnt {##1}
18407              \exp_not:n
18408              {
18409                  {#2}
18410                  { \clist_map_break:n { \prg_return_true: \use_none:n } }
18411              }
18412          }
18413          \clist_map_function:NN #3 \__clist_tmp:w
18414          \prg_return_false:
18415      }
18416      {
18417          \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
18418          \tl_if_empty:oTF
18419              { \__clist_tmp:w ,#1, {} {} ,#2, }
18420              { \prg_return_false: } { \prg_return_true: }
18421      }
18422  }
18423  \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
18424  { NV , No , c , cV , co } { T , F , TF }
18425  \prg_generate_conditional_variant:Nnn \clist_if_in:nn
18426  { nV , no } { T , F , TF }

```

(End definition for `\clist_if_in:NnTF`, `\clist_if_in:nnTF`, and `\__clist_if_in_return:nnN`. These functions are documented on page [176](#).)

## 59.7 Mapping over comma lists

### `\clist_map_function:NN`

### `\clist_map_function:cN`

### `\__clist_map_function:Nw`

### `\__clist_map_function_end:w`

If the variable is empty, the mapping is skipped (otherwise, that comma-list would be seen as consisting of one empty item). Then loop over the comma-list, grabbing eight comma-delimited items at a time. The end is marked by `\s__clist_stop`, which may not appear in any of the items. Once the last group of eight items has been reached, we go through them more slowly using `\__clist_map_function_end:w`. The auxiliary function `\__clist_map_function:Nw` is also used in some otherclist mappings.

```

18427 \cs_new:Npn \clist_map_function:NN #1#2
18428  {
18429      \clist_if_empty:NF #1
18430      {
18431          \exp_after:wN \__clist_map_function:Nw \exp_after:wN #2 #1 ,
18432          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18433          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18434          \prg_break_point:Nn \clist_map_break: { }
18435      }
18436  }
18437 \cs_new:Npn \__clist_map_function:Nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
18438  {
18439      \__clist_use_none_delimit_by_s_stop:w

```

```

18440      #9 \__clist_map_function_end:w \s__clist_stop
18441      #1 {#2} #1 {#3} #1 {#4} #1 {#5} #1 {#6} #1 {#7} #1 {#8} #1 {#9}
18442      \__clist_map_function:Nw #1
18443  }
18444 \cs_new:Npn \__clist_map_function_end:w \s__clist_stop #1#2
18445  {
18446      \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
18447      #1 {#2}
18448      \__clist_map_function_end:w \s__clist_stop
18449  }
18450 \cs_generate_variant:Nn \clist_map_function:NN { c }

(End definition for \clist_map_function:NN, \__clist_map_function:Nw, and \__clist_map_function_end:w. This function is documented on page 177.)

```

**\clist\_map\_function:nN**  
**\\_\_clist\_map\_function\_n:Nn**  
**\\_\_clist\_map\_unbrace:wn**

The n-type mapping function is a bit more awkward, since spaces must be trimmed from each item. Space trimming is again based on \\_\_clist\_trim\_next:w. The auxiliary \\_\_clist\_map\_function\_n:Nn receives as arguments the function, and the next non-empty item (after space trimming but before brace removal). One level of braces is removed by \\_\_clist\_map\_unbrace:wn.

```

18451 \cs_new:Npn \clist_map_function:nN #1#2
18452  {
18453      \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
18454      \exp:w \__clist_trim_next:w \prg_do_nothing: #1 ,
18455      \s__clist_stop \clist_map_break: ,
18456      \prg_break_point:Nn \clist_map_break: {}
18457  }
18458 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
18459  {
18460      \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18461      \__clist_map_unbrace:wn #2 , #1
18462      \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
18463      \exp:w \__clist_trim_next:w \prg_do_nothing:
18464  }
18465 \cs_new:Npn \__clist_map_unbrace:wn #1, #2 { #2 {#1} }

(End definition for \clist_map_function:nN, \__clist_map_function_n:Nn, and \__clist_map_unbrace:wn. This function is documented on page 177.)

```

**\clist\_map\_inline:Nn**  
**\clist\_map\_inline:cn**  
**\clist\_map\_inline:nn**

Inline mapping is done by creating a suitable function “on the fly”: this is done globally to avoid any issues with T<sub>E</sub>X’s groups. We use a different function for each level of nesting.

Since the mapping is non-expandable, we can perform the space-trimming needed by the n version simply by storing the comma-list in a variable. We don’t need a different comma-list for each nesting level: the comma-list is expanded before the mapping starts.

```

18466 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
18467  {
18468      \clist_if_empty:NF #1
18469  {
18470      \int_gincr:N \g__kernel_prg_map_int
18471      \cs_gset_protected:cpn
18472      { __clist_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
18473      \exp_last_unbraced:Nco \__clist_map_function:Nw
18474      { __clist_map_ \int_use:N \g__kernel_prg_map_int :w }


```

```

18475      #1 ,
18476      \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18477      \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18478      \prg_break_point:Nn \clist_map_break:
18479      { \int_gdecr:N \g__kernel_prg_map_int }
18480  }
18481  }
18482 \cs_new_protected:Npn \clist_map_inline:nn #1
18483  {
18484      \clist_set:Nn \l__clist_internal_clist {#1}
18485      \clist_map_inline:Nn \l__clist_internal_clist
18486  }
18487 \cs_generate_variant:Nn \clist_map_inline:Nn { c }

```

(End definition for `\clist_map_inline:Nn` and `\clist_map_inline:nn`. These functions are documented on page [177](#).)

`\clist_map_variable:NNn`  
`\clist_map_variable:cNn`  
`\clist_map_variable:nNn`  
`\__clist_map_variable:Nnn`

The N-type version is a straightforward application of `\clist_map_tokens:Nn`, calling `\__clist_map_variable:Nnn` for each item to assign the variable and run the user's code. The n-type version is *not* implemented in terms of the n-type function `\clist_map_tokens:Nn`, because here we are allowed to clean up the n-type comma list non-expandably.

```

18488 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
18489  { \clist_map_tokens:Nn #1 { \__clist_map_variable:Nnn #2 {#3} } }
18490 \cs_generate_variant:Nn \clist_map_variable:NNn { c }
18491 \cs_new_protected:Npn \__clist_map_variable:Nnn #1#2#3
18492  { \tl_set:Nn #1 {#3} #2 }
18493 \cs_new_protected:Npn \clist_map_variable:nNn #1
18494  {
18495      \clist_set:Nn \l__clist_internal_clist {#1}
18496      \clist_map_variable:NNn \l__clist_internal_clist
18497  }

```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `\__clist_map_variable:Nnn`. These functions are documented on page [177](#).)

`\clist_map_tokens:Nn`  
`\clist_map_tokens:cn`  
`\__clist_map_tokens:nw`  
`\__clist_map_tokens_end:w`

Essentially a copy of `\clist_map_function>NN` with braces added.

```

18498 \cs_new:Npn \clist_map_tokens:Nn #1#2
18499  {
18500      \clist_if_empty:NF #1
18501      {
18502          \exp_last_unbraced:Nno \__clist_map_tokens:nw {#2} #1 ,
18503          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18504          \s__clist_stop , \s__clist_stop , \s__clist_stop , \s__clist_stop ,
18505          \prg_break_point:Nn \clist_map_break: { }
18506      }
18507  }
18508 \cs_new:Npn \__clist_map_tokens:nw #1 #2, #3, #4, #5, #6, #7, #8, #9,
18509  {
18510      \__clist_use_none_delimit_by_s_stop:w
18511      #9 \__clist_map_tokens_end:w \s__clist_stop
18512      \use:n {#1} {#2} \use:n {#1} {#3} \use:n {#1} {#4} \use:n {#1} {#5}
18513      \use:n {#1} {#6} \use:n {#1} {#7} \use:n {#1} {#8} \use:n {#1} {#9}
18514      \__clist_map_tokens:nw {#1}

```

```

18515     }
18516 \cs_new:Npn \__clist_map_tokens_end:w \s__clist_stop \use:n #1#2
18517 {
18518     \__clist_use_none_delimit_by_s_stop:w #2 \clist_map_break: \s__clist_stop
18519     #1 {#2}
18520     \__clist_map_tokens_end:w \s__clist_stop
18521 }
18522 \cs_generate_variant:Nn \clist_map_tokens:Nn { c }

```

(End definition for `\clist_map_tokens:Nn`, `\__clist_map_tokens:nw`, and `\__clist_map_tokens_end:w`. This function is documented on page 177.)

**\clist\_map\_tokens:nn**  
`\__clist_map_tokens_n:nw`

```

18523 \cs_new:Npn \clist_map_tokens:nn #1#2
18524 {
18525     \__clist_map_tokens_n:nw {#2}
18526     \prg_do_nothing: #1 , \s__clist_stop \clist_map_break: ,
18527     \prg_break_point:Nn \clist_map_break: { }
18528 }
18529 \cs_new:Npn \__clist_map_tokens_n:nw #1#2 ,
18530 {
18531     \tl_if_empty:oF { \use_none:nn #2 ? }
18532     {
18533         \__clist_use_none_delimit_by_s_stop:w #2 \s__clist_stop
18534         \tl_trim_spaces_apply:oN {#2} \use_i_i:nn
18535         \__clist_map_unbrace:wn , {#1}
18536     }
18537     \__clist_map_tokens_n:nw {#1} \prg_do_nothing:
18538 }

```

(End definition for `\clist_map_tokens:nn` and `\__clist_map_tokens_n:nw`. This function is documented on page 177.)

**\clist\_map\_break:** The break statements use the general `\prg_map_break:Nn` mechanism.

**\clist\_map\_break:n**

```

18539 \cs_new:Npn \clist_map_break:
18540     { \prg_map_break:Nn \clist_map_break: { } }
18541 \cs_new:Npn \clist_map_break:n
18542     { \prg_map_break:Nn \clist_map_break: }

```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page 177.)

**\clist\_count:N** Counting the items in a comma list is done using the same approach as for other token count functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics. In the case of an n-type comma-list, we could of course use `\clist_map_function:nN`, but that is very slow, because it carefully removes spaces. Instead, we loop manually, and skip blank items (but not `{}`, hence the extra spaces).

```

18543 \cs_new:Npn \clist_count:N #1
18544 {
18545     \int_eval:n
18546     {
18547         0
18548         \clist_map_function:NN #1 \__clist_count:n
18549     }

```

```

18550      }
18551  \cs_generate_variant:Nn \clist_count:N { c }
18552  \cs_new:Npn \__clist_count:n #1 { + 1 }
18553  \cs_set_protected:Npn \__clist_tmp:w #1
18554  {
18555      \cs_new:Npn \clist_count:n ##1
18556  {
18557      \int_eval:n
18558  {
18559      0
18560      \__clist_count:w #1
18561      ##1 , \s__clist_stop \prg_break: , \prg_break_point:
18562  }
18563  }
18564  \cs_new:Npn \__clist_count:w ##1 ,
18565  {
18566      \__clist_use_none_delimit_by_s_stop:w ##1 \s__clist_stop
18567      \tl_if_blank:nF {##1} { + 1 }
18568      \__clist_count:w #1
18569  }
18570  }
18571  \exp_args:No \__clist_tmp:w \c_space_tl

```

(End definition for `\clist_count:N` and others. These functions are documented on page 178.)

## 59.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
\__clist_use:wwn
\__clist_use:nwwwnwn
\__clist_use:nwnw
\__clist_use:Nn
\clist_use:cn

```

First check that the variable exists. Then count the items in the comma list. If it has none, output nothing. If it has one item, output that item, brace stripped (note that space-trimming has already been done when the comma list was assigned). If it has two, place the *<separator between two>* in the middle.

Otherwise, `\__clist_use:nwwwnwn` takes the following arguments; 1: a *<separator>*, 2, 3, 4: three items from the comma list (or quarks), 5: the rest of the comma list, 6: a *<continuation>* function (`use_ii` or `use_iii` with its *<separator>* argument), 7: junk, and 8: the temporary result, which is built in a brace group following `\s__clist_stop`. The *<separator>* and the first of the three items are placed in the result, then we use the *<continuation>*, placing the remaining two items after it. When we begin this loop, the three items really belong to the comma list, the first `\s__clist_mark` is taken as a delimiter to the `use_ii` function, and the continuation is `use_ii` itself. When we reach the last two items of the original token list, `\s__clist_mark` is taken as a third item, and now the second `\s__clist_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

18572  \cs_new:Npn \clist_use:Nnnn #1#2#3#4
18573  {
18574      \clist_if_exist:NTF #1
18575  {
18576      \int_case:nnF { \clist_count:N #1 }
18577  {
18578      { 0 } { }
18579      { 1 } { \exp_after:wN \__clist_use:wwn #1 , , { } }
18580      { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
18581  }

```

```

18582 {
18583     \exp_after:wN \__clist_use:nwwwnwn
18584     \exp_after:wN { \exp_after:wN } #1 ,
18585     \s__clist_mark , { \__clist_use:nwwwnwn {#3} }
18586     \s__clist_mark , { \__clist_use:nwnn {#4} }
18587     \s__clist_stop { }
18588 }
18589 }
18590 {
18591     \msg_expandable_error:nnn
18592     { kernel } { bad-variable } {#1}
18593 }
18594 }
18595 \cs_generate_variant:Nn \clist_use:Nnnn { c }
18596 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
18597 \cs_new:Npn \__clist_use:nwwwnwn
18598     #1#2 , #3 , #4 , #5 \s__clist_mark , #6#7 \s__clist_stop #8
18599     { #6 {#3} , {#4} , #5 \s__clist_mark , {#6} #7 \s__clist_stop { #8 #1 #2 } }
18600 \cs_new:Npn \__clist_use:nwnn #1#2 , #3 \s__clist_stop #4
18601     { \exp_not:n { #4 #1 #2 } }
18602 \cs_new:Npn \clist_use:Nn #1#2
18603     { \clist_use:Nnnn #1 {#2} {#2} {#2} }
18604 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and others. These functions are documented on page 178.)

`\clist_use:nnnn`  
`\clist_use:nn`  
`\__clist_use:Nw`  
`\__clist_use_one:w`  
`\__clist_use_end:w`  
`\__clist_use_more:w`

Items are grabbed by `\__clist_use:Nw`, which detects blank items with a `\tl_if_empty:OTF` test (in which case it recurses). Non-blank items are either the end of the list, in which case the argument `#1` of `\__clist_use:Nw` is used to properly end the list, or are normal items, which must be trimmed and properly unbraced. As we find successive items, the long list of `\__clist_use:Nw` calls gets shortened and we end up calling `\__clist_use_more:w` once we have found 3 items. This auxiliary leaves the first-found item and the general separator, and calls `\__clist_use:Nw` to find more items. A subtlety is that we use `\__clist_use_end:w` both in the case of a two-item list and for the last two items of a general list: to get the correct separator, `\__clist_use_more:w` replaces the separator-of-two by the last-separator when called, namely as soon as we have found three items.

```

18605 \cs_new:Npn \clist_use:nnnn #1#2#3#4
18606 {
18607     \__clist_use:Nw \__clist_use_none_delimit_by_s_stop:w
18608     \__clist_use:Nw \__clist_use_one:w
18609     \__clist_use:Nw \__clist_use_end:w
18610     \__clist_use_more:w ;
18611     {#2} {#3} {#4} ;
18612     \prg_do_nothing: #1 , \s__clist_mark ,
18613     \s__clist_stop
18614 }
18615 \cs_new:Npn \__clist_use:Nw #1#2 ; #3 ; #4 ,
18616 {
18617     \tl_if_empty:OTF { \use_none:nn #4 ? }
18618     { \__clist_use:Nw #1#2 ; }
18619     {
18620         \__clist_use_none_delimit_by_s_mark:w #4 #1 \s__clist_mark

```

```

18621      \tl_trim_spaces_apply:oN {#4} \use_i:i:nn
18622      \__clist_map_unbrace:wn , { #2 ; }
18623      }
18624      #3 ; \prg_do_nothing:
18625      }
18626 \cs_new:Npn \__clist_use_one:w \s__clist_mark #1 , #2#3#4 \s__clist_stop
18627  { \exp_not:n {#3} }
18628 \cs_new:Npn \__clist_use_end:w
18629  \s__clist_mark #1 , #2#3#4#5#6 \s__clist_stop
18630  { \exp_not:n { #4 #5 #3 } }
18631 \cs_new:Npn \__clist_use_more:w ; #1#2#3#4#5#6 ;
18632  {
18633  \exp_not:n { #3 #5 }
18634  \__clist_use:Nw \__clist_use_end:w \__clist_use_more:w ;
18635  {#1} {#2} {#6} {#5} {#6} ;
18636  }
18637 \cs_new:Npn \clist_use:nnn #1#2 { \clist_use:nnnn {#1} {#2} {#2} {#2} }

(End definition for \clist_use:nnnn and others. These functions are documented on page 179.)
```

## 59.9 Using a single item

**\clist\_item:Nn**  
**\clist\_item:cn**  
**\\_\_clist\_item:nnN**  
**\\_\_clist\_item:ffoN**  
**\\_\_clist\_item:ffnN**

To avoid needing to test the end of the list at each step, we first compute the  $\langle length \rangle$  of the list. If the item number is 0, less than  $-\langle length \rangle$ , or more than  $\langle length \rangle$ , the result is empty. If it is negative, but not less than  $-\langle length \rangle$ , add  $\langle length \rangle + 1$  to the item number before performing the loop. The loop itself is very simple, return the item if the counter reached 1, otherwise, decrease the counter and repeat.

```

18638 \cs_new:Npn \clist_item:Nn #1#2
18639  {
18640  \__clist_item:ffoN
18641  { \clist_count:N #1 }
18642  { \int_eval:n {#2} }
18643  #1
18644  \__clist_item_N_loop:nw
18645  }
18646 \cs_new:Npn \__clist_item:nnN #1#2#3#4
18647  {
18648  \int_compare:nNnTF {#2} < 0
18649  {
18650  \int_compare:nNnTF {#2} < { - #1 }
18651  { \__clist_use_none_delimit_by_s_stop:w }
18652  { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
18653  }
18654  {
18655  \int_compare:nNnTF {#2} > {#1}
18656  { \__clist_use_none_delimit_by_s_stop:w }
18657  { #4 {#2} }
18658  }
18659  { } , #3 , \s__clist_stop
18660  }
18661 \cs_generate_variant:Nn \__clist_item:nnN { ffo, ff }
18662 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
18663  {
```

```

18664     \int_compare:nNnTF {#1} = 0
18665     { \__clist_use_i_delimit_by_s_stop:nw { \exp_not:n {#2} } }
18666     { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }
18667   }
18668 \cs_generate_variant:Nn \clist_item:Nn { c }

```

(End definition for `\clist_item:Nn`, `\__clist_item:nnN`, and `\__clist_item_N_loop:nw`. This function is documented on page 180.)

### `\clist_item:nn`

```

\__clist_item_n:nw
\__clist_item_n_loop:nw
\__clist_item_n_end:n
\__clist_item_n_strip:n
\__clist_item_n_strip:w

```

This starts in the same way as `\clist_item:Nn` by counting the items of the comma list. The final item should be space-trimmed before being brace-stripped, hence we insert a couple of odd-looking `\prg_do_nothing:` to avoid losing braces. Blank items are ignored.

```

18669 \cs_new:Npn \clist_item:nn #1#2
18670   {
18671     \__clist_item:ffnN
18672     { \clist_count:n {#1} }
18673     { \int_eval:n {#2} }
18674     {#1}
18675     \__clist_item_n:nw
18676   }
18677 \cs_new:Npn \__clist_item_n:nw #1
18678   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
18679 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
18680   {
18681     \exp_args:No \tl_if_blank:nTF {#2}
18682     { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
18683     {
18684       \int_compare:nNnTF {#1} = 0
18685       { \exp_args:No \__clist_item_n_end:n {#2} }
18686       {
18687         \exp_args:Nf \__clist_item_n_loop:nw
18688         { \int_eval:n { #1 - 1 } }
18689         \prg_do_nothing:
18690       }
18691     }
18692   }
18693 \cs_new:Npn \__clist_item_n_end:n #1 #2 \s__clist_stop
18694   { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
18695 \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
18696 \cs_new:Npn \__clist_item_n_strip:w #1 , { \exp_not:n {#1} }

```

(End definition for `\clist_item:nn` and others. This function is documented on page 180.)

```

\clist_rand_item:n
\clist_rand_item:N
\clist_rand_item:c
\__clist_rand_item:nn

```

The N-type function is not implemented through the n-type function for efficiency: for instance comma-list variables do not require space-trimming of their items. Even testing for emptiness of an n-type comma-list is slow, so we count items first and use that both for the emptiness test and the pseudo-random integer. Importantly, `\clist_item:Nn` and `\clist_item:nn` only evaluate their argument once.

```

18697 \cs_new:Npn \clist_rand_item:n #1
18698   { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
18699 \cs_new:Npn \__clist_rand_item:nn #1#2
18700   {
18701     \int_compare:nNnF {#1} = 0
18702     { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }

```

```

18703     }
18704     \cs_new:Npn \clist_rand_item:N #1
18705     {
18706         \clist_if_empty:NF #1
18707             { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
18708     }
18709 \cs_generate_variant:Nn \clist_rand_item:N { c }

```

(End definition for `\clist_rand_item:n`, `\clist_rand_item:N`, and `\_clist_rand_item:nn`. These functions are documented on page 181.)

## 59.10 Viewing comma lists

**\clist\_show:N** Apply the general `\_kernel_chk_tl_type:NnnT` with `\exp_not:o #2` serving as a dummy code to prevent a check performed by this auxiliary.

```

\clist_log:N
\clist_log:c
\__clist_show:NN
18710 \cs_new_protected:Npn \clist_show:N { \__clist_show:NN \msg_show:nnxxxx }
18711 \cs_generate_variant:Nn \clist_show:N { c }
18712 \cs_new_protected:Npn \clist_log:N { \__clist_show:NN \msg_log:nnxxxx }
18713 \cs_generate_variant:Nn \clist_log:N { c }
18714 \cs_new_protected:Npn \__clist_show:NN #1#2
18715     {
18716         \_kernel_chk_tl_type:NnnT #2 { clist } { \exp_not:o #2 }
18717         {
18718             \int_compare:nNnTF { \clist_count:N #2 }
18719                 = { \exp_args:No \clist_count:n #2 }
18720                 {
18721                     #1 { clist } { show }
18722                         { \token_to_str:N #2 }
18723                         { \clist_map_function:NN #2 \msg_show_item:n }
18724                         { } { }
18725                 }
18726                 {
18727                     \msg_error:nnxx { clist } { non-clist }
18728                         { \token_to_str:N #2 } { \tl_to_str:N #2 }
18729                 }
18730             }
18731     }

```

(End definition for `\clist_show:N`, `\clist_log:N`, and `\_clist_show:NN`. These functions are documented on page 181.)

**\clist\_show:n** A variant of the above: no existence check, empty first argument for the message.

```

\clist_log:n
\__clist_show:Nn
18732 \cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nnxxxx }
18733 \cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nnxxxx }
18734 \cs_new_protected:Npn \__clist_show:Nn #1#2
18735     {
18736         #1 { clist } { show }
18737             { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
18738     }

```

(End definition for `\clist_show:n`, `\clist_log:n`, and `\_clist_show:Nn`. These functions are documented on page 181.)

## 59.11 Scratch comma lists

```
\l_tmpa_clist  Temporary comma list variables.  
\l_tmpb_clist  
\g_tmpa_clist  
\g_tmpb_clist  
18739 \clist_new:N \l_tmpa_clist  
18740 \clist_new:N \l_tmpb_clist  
18741 \clist_new:N \g_tmpa_clist  
18742 \clist_new:N \g_tmpb_clist
```

(End definition for `\l_tmpa_clist` and others. These variables are documented on page 181.)

```
18743 </package>
```

# Chapter 60

## I3token implementation

```
18744  <*package>
18745  <*tex>
18746  <@@=char>
```

### 60.1 Internal auxiliaries

\s\_\_char\_stop Internal scan mark.

```
18747  \scan_new:N \s__char_stop
```

(End definition for \s\_\_char\_stop.)

\q\_\_char\_no\_value Internal recursion quarks.

```
18748  \quark_new:N \q__char_no_value
```

(End definition for \q\_\_char\_no\_value.)

\\_\\_char\_quark\_if\_no\_value\_p:N Functions to query recursion quarks.

```
18749  \_\_kernel_quark_new_conditional:Nn \_\_char_quark_if_no_value:N { TF }
```

(End definition for \\_\\_char\_quark\_if\_no\_value:NTF.)

### 60.2 Manipulating and interrogating character tokens

\char\_set\_catcode:nn Simple wrappers around the primitives.

```
18750  \cs_new_protected:Npn \char_set_catcode:nn #1#2
18751    { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18752    \cs_new:Npn \char_value_catcode:n #1
18753      { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
18754    \cs_new_protected:Npn \char_show_value_catcode:n #1
18755      { \exp_args:Nf \tl_show:n { \char_value_catcode:n {#1} } }
```

(End definition for \char\_set\_catcode:nn, \char\_value\_catcode:n, and \char\_show\_value\_catcode:n. These functions are documented on page 185.)

```

\char_set_catcode_escape:N
  \char_set_catcode_group_begin:N
    \char_set_catcode_group_end:N
      \char_set_catcode_math_toggle:N
        \char_set_catcode_alignment:N
\char_set_catcode_end_line:N
  \char_set_catcode_parameter:N
\char_set_catcode_math_superscript:N
  \char_set_catcode_math_subscript:N
\char_set_catcode_ignore:N
  \char_set_catcode_space:N
\char_set_catcode_letter:N
  \char_set_catcode_other:N
\char_set_catcode_active:N
\char_set_catcode_comment:N
\char_set_catcode_invalid:N
  18756 \cs_new_protected:Npn \char_set_catcode_escape:N #1
  18757   { \char_set_catcode:nn { '#1 } { 0 } }
  18758 \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
  18759   { \char_set_catcode:nn { '#1 } { 1 } }
  18760 \cs_new_protected:Npn \char_set_catcode_group_end:N #1
  18761   { \char_set_catcode:nn { '#1 } { 2 } }
  18762 \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
  18763   { \char_set_catcode:nn { '#1 } { 3 } }
  18764 \cs_new_protected:Npn \char_set_catcode_alignment:N #1
  18765   { \char_set_catcode:nn { '#1 } { 4 } }
  18766 \cs_new_protected:Npn \char_set_catcode_end_line:N #1
  18767   { \char_set_catcode:nn { '#1 } { 5 } }
  18768 \cs_new_protected:Npn \char_set_catcode_parameter:N #1
  18769   { \char_set_catcode:nn { '#1 } { 6 } }
  18770 \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
  18771   { \char_set_catcode:nn { '#1 } { 7 } }
  18772 \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
  18773   { \char_set_catcode:nn { '#1 } { 8 } }
  18774 \cs_new_protected:Npn \char_set_catcode_ignore:N #1
  18775   { \char_set_catcode:nn { '#1 } { 9 } }
  18776 \cs_new_protected:Npn \char_set_catcode_space:N #1
  18777   { \char_set_catcode:nn { '#1 } { 10 } }
  18778 \cs_new_protected:Npn \char_set_catcode_letter:N #1
  18779   { \char_set_catcode:nn { '#1 } { 11 } }
  18780 \cs_new_protected:Npn \char_set_catcode_other:N #1
  18781   { \char_set_catcode:nn { '#1 } { 12 } }
  18782 \cs_new_protected:Npn \char_set_catcode_active:N #1
  18783   { \char_set_catcode:nn { '#1 } { 13 } }
  18784 \cs_new_protected:Npn \char_set_catcode_comment:N #1
  18785   { \char_set_catcode:nn { '#1 } { 14 } }
  18786 \cs_new_protected:Npn \char_set_catcode_invalid:N #1
  18787   { \char_set_catcode:nn { '#1 } { 15 } }

```

(End definition for `\char_set_catcode_escape:N` and others. These functions are documented on page 184.)

```

\char_set_catcode_escape:n
  \char_set_catcode_group_begin:n
    \char_set_catcode_group_end:n
\char_set_catcode_math_toggle:n
  \char_set_catcode_alignment:n
\char_set_catcode_end_line:n
  \char_set_catcode_parameter:n
\char_set_catcode_math_superscript:n
  \char_set_catcode_math_subscript:n
\char_set_catcode_ignore:n
  \char_set_catcode_space:n
\char_set_catcode_letter:n
  \char_set_catcode_other:n
\char_set_catcode_active:n
\char_set_catcode_comment:n
\char_set_catcode_invalid:n
  18788 \cs_new_protected:Npn \char_set_catcode_escape:n #1
  18789   { \char_set_catcode:nn {#1} { 0 } }
  18790 \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
  18791   { \char_set_catcode:nn {#1} { 1 } }
  18792 \cs_new_protected:Npn \char_set_catcode_group_end:n #1
  18793   { \char_set_catcode:nn {#1} { 2 } }
  18794 \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
  18795   { \char_set_catcode:nn {#1} { 3 } }
  18796 \cs_new_protected:Npn \char_set_catcode_alignment:n #1
  18797   { \char_set_catcode:nn {#1} { 4 } }
  18798 \cs_new_protected:Npn \char_set_catcode_end_line:n #1
  18799   { \char_set_catcode:nn {#1} { 5 } }
  18800 \cs_new_protected:Npn \char_set_catcode_parameter:n #1
  18801   { \char_set_catcode:nn {#1} { 6 } }
  18802 \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
  18803   { \char_set_catcode:nn {#1} { 7 } }

```

```

18804 \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
18805   { \char_set_catcode:nn {#1} { 8 } }
18806 \cs_new_protected:Npn \char_set_catcode_ignore:n #1
18807   { \char_set_catcode:nn {#1} { 9 } }
18808 \cs_new_protected:Npn \char_set_catcode_space:n #1
18809   { \char_set_catcode:nn {#1} { 10 } }
18810 \cs_new_protected:Npn \char_set_catcode_letter:n #1
18811   { \char_set_catcode:nn {#1} { 11 } }
18812 \cs_new_protected:Npn \char_set_catcode_other:n #1
18813   { \char_set_catcode:nn {#1} { 12 } }
18814 \cs_new_protected:Npn \char_set_catcode_active:n #1
18815   { \char_set_catcode:nn {#1} { 13 } }
18816 \cs_new_protected:Npn \char_set_catcode_comment:n #1
18817   { \char_set_catcode:nn {#1} { 14 } }
18818 \cs_new_protected:Npn \char_set_catcode_invalid:n #1
18819   { \char_set_catcode:nn {#1} { 15 } }

```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 184.)

Pretty repetitive, but necessary!

```

18820 \cs_new_protected:Npn \char_set_mathcode:nn #1#2
18821   { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18822 \cs_new:Npn \char_value_mathcode:n #1
18823   { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
18824 \cs_new_protected:Npn \char_show_value_mathcode:n #1
18825   { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
18826 \cs_new_protected:Npn \char_set_lccode:nn #1#2
18827   { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18828 \cs_new:Npn \char_value_lccode:n #1
18829   { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
18830 \cs_new_protected:Npn \char_show_value_lccode:n #1
18831   { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
18832 \cs_new_protected:Npn \char_set_uccode:nn #1#2
18833   { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18834 \cs_new:Npn \char_value_uccode:n #1
18835   { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }
18836 \cs_new_protected:Npn \char_show_value_uccode:n #1
18837   { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
18838 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
18839   { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
18840 \cs_new:Npn \char_value_sfcode:n #1
18841   { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
18842 \cs_new_protected:Npn \char_show_value_sfcode:n #1
18843   { \exp_args:Nf \tl_show:n { \char_value_sfcode:n {#1} } }

```

(End definition for `\char_set_mathcode:nn` and others. These functions are documented on page 186.)

`\l_char_active_seq`  
`\l_char_special_seq`

Two sequences for dealing with special characters. The first is characters which may be active, the second longer list is for “special” characters more generally. Both lists are escaped so that for example bulk code assignments can be carried out. In both cases, the order is by ASCII character code (as is done in for example `\ExplSyntaxOn`).

```

18844 \seq_new:N \l_char_special_seq
18845 \seq_set_split:Nnn \l_char_special_seq { }

```

```

18846   { \ " \# \$ % & \\ ^ _ { } ~ }
18847 \seq_new:N \l_char_active_seq
18848 \seq_set_split:Nnn \l_char_active_seq { }
18849   { " \$ & ^ _ ~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 186.)

## 60.3 Creating character tokens

`\char_set_active_eq:NN`

`\char_set_active_eq:Nc`

`\char_gset_active_eq:NN`

`\char_gset_active_eq:Nc`

`\char_set_active_eq:nN`

`\char_set_active_eq:nc`

`\char_gset_active_eq:nN`

`\char_gset_active_eq:nc`

Four simple functions with very similar definitions, so set up using an auxiliary. These are similar to LuaTeX's `\letcharcode` primitive.

```

18850 \group_begin:
18851   \char_set_catcode_active:N ^@_
18852   \cs_set_protected:Npn \__char_tmp:nN #1#2
18853   {
18854     \cs_new_protected:cpn { #1 :nN } ##1
18855     {
18856       \group_begin:
18857         \char_set_lccode:nn { '^@_ } { ##1 }
18858         \tex_lowercase:D { \group_end: #2 ^@_ }
18859     }
18860     \cs_new_protected:cpn { #1 :NN } ##1
18861     { \exp_not:c { #1 : nN } { ##1 } }
18862   }
18863   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
18864   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
18865 \group_end:
18866 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
18867 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
18868 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
18869 \cs_generate_variant:Nn \char_gset_active_eq:nN { nc }

```

(End definition for `\char_set_active_eq:NN` and others. These functions are documented on page 183.)

`\__char_int_to_roman:w`

For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```
18870 \cs_new_eq:NN \__char_int_to_roman:w \tex_roman numeral:D
```

(End definition for `\__char_int_to_roman:w`.)

`\char_generate:nn`

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (XeTeX, LuaTeX). There are though various issues which are covered below. At the interface layer, turn the two arguments into integers up-front so this is only done once.

```

18871 \cs_new:Npn \char_generate:nn #1#2
18872   {
18873     \exp:w \exp_after:wN \__char_generate_aux:w
18874     \int_value:w \int_eval:n {#1} \exp_after:wN ;
18875     \int_value:w \int_eval:n {#2} ;
18876   }

```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaTEX emulation only makes normal (charcode 32 spaces). However,  $\wedge\wedge 0$  is filtered out separately as that can't be done with macro emulation either, so is flagged up separately. That done, hand off to the engine-dependent part.

```

18877 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
18878   {
18879     \if_int_odd:w 0
18880       \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
18881       \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
18882       \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
18883       \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
18884       \msg_expandable_error:nn { char }
18885         { invalid-catcode }
18886   \else:
18887     \if_int_odd:w 0
18888       \if_int_compare:w #1 < \c_zero_int 1 \fi:
18889       \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
18890       \msg_expandable_error:nn { char }
18891         { out-of-range }
18892   \else:
18893     \if_int_compare:w #2#1 = 100 \exp_stop_f:
18894       \msg_expandable_error:nn { char } { null-space }
18895     \else:
18896       \__char_generate_aux:nnw {#1} {#2}
18897     \fi:
18898   \fi:
18899   \fi:
18900   \exp_end:
18901 }
18902 \tl_new:N \l__char_tmp_tl

```

Engine-dependent definitions are now needed for the implementation. For LuaTEX and X<sub>E</sub>T<sub>E</sub>X there is engine-level support. They can do cases that macro emulation can't. All of those are filtered out here using a primitive-based boolean expression to avoid fixing the category code of the null character used in the false branch (for 8-bit engines). The final level is the basic definition at the engine level: the arguments here are integers so there is no need to worry about them too much. Older versions of X<sub>E</sub>T<sub>E</sub>X cannot generate active characters so we filter that: at some future stage that may change: the slightly odd ordering of auxiliaries reflects that.

```

18903 \group_begin:
18904   \if_int_odd:w 0
18905     \sys_if_engine_luatex:T { 1 }
18906     \sys_if_engine_xetex:T { 1 } \exp_stop_f:
18907     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
18908     {
18909       #3
18910       \exp_after:wN \exp_end:
18911       \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
18912     }
18913   \cs_if_exist:NF \tex_expanded:D
18914   {
18915     \cs_new_eq:NN \__char_generate_auxii:nnw \__char_generate_aux:nnw
18916     \cs_gset:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:

```

```

18917 {
18918 #3
18919 \if_int_compare:w #2 = 13 \exp_stop_f:
18920   \msg_expandable_error:nn { char } { active }
18921 \else:
18922   \_char_generate_auxii:nw {#1} {#2}
18923 \fi:
18924 \exp_end:
18925 }
18926 }
18927 \else:

```

For engines where `\Ucharcat` isn't available or emulated, we have to work in macros, and cover only the 8-bit range. The first stage is to build up a `tl` containing `^~@` with each category code that can be accessed in this way, with an error set up for the other cases. This is all done such that it can be quickly accessed using a `\if_case:w` low-level conditional. The list is done in reverse as this puts the case of an active token *first*: that's needed to cover the possibility that it is `\outer`. Getting the braces into the list is done using some standard `\if_false:` manipulation, while all of the `\exp_not:N` are required as there is an expansion in the setup.

```

18928 \char_set_catcode_active:N ^~L
18929 \cs_set:Npn ^~L { }
18930 \char_set_catcode_active:n { 0 }
18931 \tl_set:Nn \l__char_tmp_tl { \exp_not:N ^~@ \exp_not:N \or: }
18932 \char_set_catcode_other:n { 0 }
18933 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: }
18934 \char_set_catcode_letter:n { 0 }
18935 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: }
18936 % For making spaces, there needs to be an /o/-type expansion of a /\use:n/
18937 % (or some other tokenization) to avoid dropping the space.
18938 % \begin{macrocode}
18939 \tl_put_right:Nn \l__char_tmp_tl { \use:n { ~ } \exp_not:N \or: }
18940 \tl_put_right:Nn \l__char_tmp_tl { \exp_not:N \or: }
18941 \char_set_catcode_math_subscript:n { 0 }
18942 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: }
18943 \char_set_catcode_math_superscript:n { 0 }
18944 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: }
18945 \char_set_catcode_parameter:n { 0 }
18946 \tl_put_right:Nn \l__char_tmp_tl { ^~@^~@ \exp_not:N \or: }
18947 \tl_put_right:Nn \l__char_tmp_tl { { \if_false: } \fi: \exp_not:N \or: }
18948 \char_set_catcode_alignment:n { 0 }
18949 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: }
18950 \char_set_catcode_math_toggle:n { 0 }
18951 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: }
18952 \char_set_catcode_group_end:n { 0 }
18953 \tl_put_right:Nn \l__char_tmp_tl { \if_false: { \fi: ^~@ \exp_not:N \or: } % }
18954 \char_set_catcode_group_begin:n { 0 } % {
18955 \tl_put_right:Nn \l__char_tmp_tl { ^~@ \exp_not:N \or: } }

```

Convert the above temporary list into a series of constant token lists, one for each character code, using `\tex_lowercase:D` to convert `^~@` in each case. The x-type expansion ensures that `\tex_lowercase:D` receives the contents of the token list.

```

18956 \cs_set_protected:Npn \_char_tmp:n #1
18957   {

```

```

18958 \char_set_lccode:nn { 0 } {#1}
18959 \char_set_lccode:nn { 32 } {#1}
18960 \exp_args:Nx \tex_lowercase:D
18961 {
18962     \tl_const:Nx
18963         \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
18964         { \exp_not:o \l__char_tmp_tl }
18965     }
18966 }
18967 \int_step_function:nnN { 0 } { 255 } \__char_tmp:n

```

As TeX is very unhappy if it finds an alignment character inside a primitive `\halign` even when skipping false branches, some precautions are required. TeX is happy if the token is hidden between braces within `\if_false: ... \fi:`. The rather low-level approach here expands in one step to the `\langle target token \rangle (\or: ...)`, then `\exp_after:wN \langle target token \rangle (\or: ...)` expands in one step to `\langle target token \rangle`. This means that `\exp_not:N` is applied to a potentially-problematic active token.

```

18968 \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
18969 {
18970     #3
18971     \if_false: { \fi:
18972         \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
18973         \exp_after:wN \exp_after:wN
18974         \if_case:w \tex_numexpr:D 13 - #2
18975             \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
18976             \exp_after:wN \exp_after:wN \exp_after:wN \scan_stop:
18977             \exp_after:wN \exp_after:wN \exp_after:wN \exp_not:N
18978                 \cs:w c__char_ \__char_int_to_roman:w #1 _tl \cs_end:
18979         }
18980         \fi:
18981     }
18982     \fi:
18983 \group_end:

```

(End definition for `\char_generate:nn` and others. This function is documented on page 183.)

`\char_lowercase:N`  
`\char_uppercase:N`  
`\char_titlecase:N`  
`\char_foldcase:N`

```

\__char_change_case:nN
\__char_change_case:nnN
\__char_change_case_auxi:nN
\__char_change_case_auxii:nN
    \char_change_case_catcode:N
\char_str_lowercase:N
\char_str_uppercase:N
\char_str_titlecase:N
    \char_str_foldcase:N
\__char_str_change_case:nN
    \char_str_change_case_aux:nN
\__char_str_change_case:nnN
    \__char_str_change_case:n

```

```

18984 \cs_new:Npn \char_lowercase:N
18985     { \__char_change_case:nN { lowercase } }
18986 \cs_new:Npn \char_uppercase:N
18987     { \__char_change_case:nN { uppercase } }
18988 \cs_new:Npn \char_titlecase:N
18989     { \__char_change_case:nN { titlecase } }
18990 \cs_new:Npn \char_foldcase:N
18991     { \__char_change_case:nN { casefold } }
18992 \cs_new:Npn \__char_change_case:nN #1#2
18993 {
18994     \int_compare:nNnTF {'#2} = { '\ }
18995     { ~ }
18996     {
18997         \exp_args:Ne \__char_change_case_auxi:nN
18998             { \__kernel_codepoint_case:nn {#1} {'#2} } #2
18999     }

```

```

19000    }
19001 \cs_new:Npn \__char_change_case_auxi:nN #1#2
19002 { \use:e { \__char_change_case:nnnN #1 #2 } }
19003 \cs_new:Npn \__char_change_case:nnnN #1#2#3#4
19004 {
19005     \int_compare:nNnTF {#1} = {'#4}
19006     { \exp_not:n {#4} }
19007     {
19008         \__char_change_case_auxii:nN {#1} {#4}
19009         \tl_if_blank:nF {#2}
19010         {
19011             \__char_change_case_auxii:nN {#2} {#4}
19012             \tl_if_blank:nF {#3}
19013             { \__char_change_case_auxii:nN {#3} {#4} }
19014         }
19015     }
19016 }
19017 \cs_new:Npn \__char_change_case_auxii:nN #1#2
19018 {
19019     \char_generate:nn {#1}
19020     { \__char_change_case_catcode:N #2 }
19021 }
19022 \bool_lazy_or:nnF
19023 { \sys_if_engine_luatex_p: }
19024 { \sys_if_engine_xetex_p: }
19025 {
19026     \cs_gset:Npn \__char_change_case_auxii:nN #1#2
19027     {
19028         \int_compare:nNnTF {#1} < { "80 }
19029         {
19030             \char_generate:nn {#1}
19031             { \__char_change_case_catcode:N #2 }
19032         }
19033         { \exp_not:n {#2} }
19034     }
19035 }
19036 \cs_new:Npn \__char_change_case_catcode:N #1
19037 {
19038     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
19039     3
19040 \else:
19041     \if_catcode:w \exp_not:N #1 \c_alignment_token
19042     4
19043 \else:
19044     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
19045     7
19046 \else:
19047     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
19048     8
19049 \else:
19050     \if_catcode:w \exp_not:N #1 \c_space_token
19051     10
19052 \else:
19053     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token

```

```

19054      11
19055      \else:
19056          \if_catcode:w \exp_not:N #1 \c_catcode_other_token
19057              12
19058          \else:
19059              13
19060          \fi:
19061          \fi:
19062          \fi:
19063          \fi:
19064          \fi:
19065          \fi:
19066          \fi:
19067      }

```

Same story for the string version, except category code is easier to follow. This of course makes this version faster.

```

19068 \cs_new:Npn \char_str_lowercase:N
19069     { \__char_str_change_case:nN { lowercase } }
19070 \cs_new:Npn \char_str_uppercase:N
19071     { \__char_str_change_case:nN { uppercase } }
19072 \cs_new:Npn \char_str_titlecase:N
19073     { \__char_str_change_case:nN { titlecase } }
19074 \cs_new:Npn \char_str_foldcase:N
19075     { \__char_str_change_case:nN { casifold } }
19076 \cs_new:Npn \__char_str_change_case:nN #1#2
19077     {
19078         \int_compare:nNnTF {'#2} = { '\ }
19079             { ~ }
19080             {
19081                 \exp_args:N \__char_str_change_case_aux:nN
19082                     { \__kernel_codepoint_case:nn {#1} {'#2} } #2
19083             }
19084         }
19085 \cs_new:Npn \__char_str_change_case_aux:nN #1#2
19086     { \use:e { \__char_str_change_case:nnN #1 #2 } }
19087 \cs_new:Npn \__char_str_change_case:nnN #1#2#3#4
19088     {
19089         \int_compare:nNnTF {#1} = {'#4}
19090             { \tl_to_str:n {#4} }
19091             {
19092                 \__char_str_change_case:n {#1}
19093                 \tl_if_blank:nF {#2}
19094                 {
19095                     \__char_str_change_case:n {#2}
19096                     \tl_if_blank:nF {#3}
19097                         { \__char_str_change_case:n {#3} }
19098                 }
19099             }
19100         }
19101 \cs_new:Npn \__char_str_change_case:n #1
19102     { \char_generate:nn {#1} { 12 } }

```

(End definition for `\char_lowercase:N` and others. These functions are documented on page 198.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.  
<sup>19103</sup> `\tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }`  
*(End definition for `\c_catcode_other_space_tl`. This function is documented on page 183.)*

## 60.4 Generic tokens

```
19104 <@=token>  

\c_catcode_other_space_tl Create a space with category code 12: an “other” space.  

19105 \tl_const:Nx \c_catcode_other_space_tl { \char_generate:nn { '\ } { 12 } }  

19106 \scan_new:N \s__token_mark  

19107 \scan_new:N \s__token_stop  

(End definition for \s__token_mark and \s__token_stop.)
```

`\token_to_meaning:N`  
`\token_to_meaning:c`  
`\token_to_str:N`  
`\token_to_str:c`

These are all defined in l3basics, as they are needed “early”. This is just a reminder!  
*(End definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 187.)*

We define these useful tokens. For the brace and space tokens things have to be done by hand: the formal argument spec. for `\cs_new_eq:NN` does not cover them so we do things by hand. (As currently coded it would *work* with `\cs_new_eq:NN` but that’s not really a great idea to show off: we want people to stick to the defined interfaces and that includes us.) So that these few odd names go into the log when appropriate there is a need to hand-apply the `\_kernel_chk_if_free_cs:N` check.

```
19108 \group_begin:  

19109   \_kernel_chk_if_free_cs:N \c_group_begin_token  

19110   \tex_global:D \tex Let:D \c_group_begin_token {  

19111     \_kernel_chk_if_free_cs:N \c_group_end_token  

19112     \tex_global:D \tex Let:D \c_group_end_token }  

19113     \char_set_catcode_math_toggle:N /*  

19114     \cs_new_eq:NN \c_math_toggle_token *  

19115     \char_set_catcode_alignment:N /*  

19116     \cs_new_eq:NN \c_alignment_token *  

19117     \cs_new_eq:NN \c_parameter_token #  

19118     \cs_new_eq:NN \c_math_superscript_token ^  

19119     \char_set_catcode_math_subscript:N /*  

19120     \cs_new_eq:NN \c_math_subscript_token *  

19121     \_kernel_chk_if_free_cs:N \c_space_token  

19122     \use:n { \tex_global:D \tex Let:D \c_space_token = ~ } ~  

19123     \cs_new_eq:NN \c_catcode_letter_token a  

19124     \cs_new_eq:NN \c_catcode_other_token 1  

\group_end:
```

*(End definition for `\c_group_begin_token` and others. These functions are documented on page 187.)*

`\c_catcode_active_tl` Not an implicit token!

```
19125 \group_begin:  

19126   \char_set_catcode_active:N /*  

19127   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }  

\group_end:
```

*(End definition for `\c_catcode_active_tl`. This variable is documented on page 187.)*

## 60.5 Token conditionals

`\token_if_group_begin_p:N` Check if token is a begin group token. We use the constant `\c_group_begin_token` for this.

```

19129 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
19130   {
19131     \if_catcode:w \exp_not:N #1 \c_group_begin_token
19132       \prg_return_true: \else: \prg_return_false: \fi:
19133   }

```

(End definition for `\token_if_group_begin:NTF`. This function is documented on page 188.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

```

19134 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
19135   {
19136     \if_catcode:w \exp_not:N #1 \c_group_end_token
19137       \prg_return_true: \else: \prg_return_false: \fi:
19138   }

```

(End definition for `\token_if_group_end:NTF`. This function is documented on page 188.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

```

19139 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
19140   {
19141     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
19142       \prg_return_true: \else: \prg_return_false: \fi:
19143   }

```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 188.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

```

19144 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
19145   {
19146     \if_catcode:w \exp_not:N #1 \c_alignment_token
19147       \prg_return_true: \else: \prg_return_false: \fi:
19148   }

```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 188.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this. We have to trick TeX a bit to avoid an error message: within a group we prevent `\c_parameter_token` from behaving like a macro parameter character. The definitions of `\prg_new_conditional:Npnn` are global, so they remain after the group.

```

19149 \group_begin:
19150   \cs_set_eq:NN \c_parameter_token \scan_stop:
19151   \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
19152   {
19153     \if_catcode:w \exp_not:N #1 \c_parameter_token
19154       \prg_return_true: \else: \prg_return_false: \fi:
19155   }
19156 \group_end:

```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 188.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.

```
19157 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
19158   { p , T , F , TF }
19159   {
19160     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
19161     \prg_return_true: \else: \prg_return_false: \fi:
19162   }
```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 188.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.

```
19163 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
19164   {
19165     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
19166     \prg_return_true: \else: \prg_return_false: \fi:
19167   }
```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 188.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.

```
19168 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
19169   {
19170     \if_catcode:w \exp_not:N #1 \c_space_token
19171     \prg_return_true: \else: \prg_return_false: \fi:
19172   }
```

(End definition for `\token_if_space:NTF`. This function is documented on page 188.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.

```
19173 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
19174   {
19175     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
19176     \prg_return_true: \else: \prg_return_false: \fi:
19177   }
```

(End definition for `\token_if_letter:NTF`. This function is documented on page 189.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

```
19178 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
19179   {
19180     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
19181     \prg_return_true: \else: \prg_return_false: \fi:
19182   }
```

(End definition for `\token_if_other:NTF`. This function is documented on page 189.)

`\token_if_active_p:N` Check if token is an active char token. We use the constant `\c_catcode_active_tl` for this. A technical point is that `\c_catcode_active_tl` is in fact a macro expanding to `\exp_not:N *`, where \* is active.

```
19183 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
19184   {
19185     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
19186     \prg_return_true: \else: \prg_return_false: \fi:
19187   }
```

(End definition for `\token_if_active:NTF`. This function is documented on page 189.)

```
\token_if_eq_meaning_p:NN Check if the tokens #1 and #2 have same meaning.  
\token_if_eq_meaning:NNTF  
19188 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }  
19189 {  
19190 \if_meaning:w #1 #2  
19191 \prg_return_true: \else: \prg_return_false: \fi:  
19192 }
```

(End definition for `\token_if_eq_meaning:NNTF`. This function is documented on page 189.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

```
\token_if_eq_catcode:NNTF  
19193 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }  
19194 {  
19195 \if_catcode:w \exp_not:N #1 \exp_not:N #2  
19196 \prg_return_true: \else: \prg_return_false: \fi:  
19197 }
```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 189.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

```
\token_if_eq_charcode:NNTF  
19198 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }  
19199 {  
19200 \if_charcode:w \exp_not:N #1 \exp_not:N #2  
19201 \prg_return_true: \else: \prg_return_false: \fi:  
19202 }
```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 189.)

`\token_if_macro_p:N` When a token is a macro, `\token_to_meaning:N` always outputs something like `\long macro:#1->#1` so we could naively check to see if the meaning contains `->`.  
`\token_if_macro:NTF` However, this can fail the five `\...mark` primitives, whose meaning has the form `...mark:<user material>`. The problem is that the `<user material>` can contain `->`.

However, only characters, macros, and marks can contain the colon character. The idea is thus to grab until the first `:`, and analyse what is left. However, macros can have any combination of `\long`, `\protected` or `\outer` (not used in L<sup>A</sup>T<sub>E</sub>X3) before the string `macro::`. We thus only select the part of the meaning between the first `ma` and the first following `:`. If this string is `cro`, then we have a macro. If the string is `rk`, then we have a mark. The string can also be `cro parameter character` for a colon with a weird category code (namely the usual category code of `#`). Otherwise, it is empty.

This relies on the fact that `\long`, `\protected`, `\outer` cannot contain `ma`, regardless of the escape character, even if the escape character is `\m...`

Both `ma` and `:` must be of category code 12 (other), so are detokenized.

```
19203 \use:x  
19204 {  
19205 \prg_new_conditional:Npnn \exp_not:N \token_if_macro:N ##1  
19206 { p , T , F , TF }  
19207 {  
19208 \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w  
19209 \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }  
19210 \s__token_stop  
19211 }  
19212 \cs_new:Npn \exp_not:N \__token_if_macro_p:w
```

```

19213      ##1 \tl_to_str:n { ma } ##2 \c_colon_str ##3 \s__token_stop
19214  }
19215  {
19216      \str_if_eq:nnTF { #2 } { cro }
19217          { \prg_return_true: }
19218          { \prg_return_false: }
19219  }

```

(End definition for `\token_if_macro:NTF` and `\_token_if_macro_p:w`. This function is documented on page 189.)

**\token\_if\_cs\_p:N** Check if token has same catcode as a control sequence. This follows the same pattern as **\token\_if\_cs:NTF** for `\token_if_letter:N` etc. We use `\scan_stop:` for this.

```

19220 \prg_new_conditional:Npnn \token_if_cs:N #1 { p , T , F , TF }
19221  {
19222      \if_catcode:w \exp_not:N #1 \scan_stop:
19223          \prg_return_true: \else: \prg_return_false: \fi:
19224  }

```

(End definition for `\token_if_cs:NTF`. This function is documented on page 189.)

**\token\_if\_expandable\_p:N** **\token\_if\_expandable:NTF** Check if token is expandable. We use the fact that TeX temporarily converts `\exp_not:N` `\token` into `\scan_stop:` if `\token` is expandable. An undefined token is not considered as expandable. No problem nesting the conditionals, since the third #1 is only skipped if it is non-expandable (hence not part of TeX's conditional apparatus).

```

19225 \prg_new_conditional:Npnn \token_if_expandable:N #1 { p , T , F , TF }
19226  {
19227      \exp_after:wN \if_meaning:w \exp_not:N #1 #
19228          \prg_return_false:
19229      \else:
19230          \if_cs_exist:N #1
19231              \prg_return_true:
19232          \else:
19233              \prg_return_false:
19234          \fi:
19235      \fi:
19236  }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 189.)

`\_token_delimit_by_char":w` `\_token_delimit_by_count:w` `\_token_delimit_by_dimen:w` `\_token_delimit_by_\font:w` `\_token_delimit_by_macro:w` `\_token_delimit_by_muskip:w` `\_token_delimit_by_skip:w` `\_token_delimit_by_toks:w` These auxiliary functions are used below to define some conditionals which detect whether the `\meaning` of their argument begins with a particular string. Each auxiliary takes an argument delimited by a string, a second one delimited by `\s__token_stop`, and returns the first one and its delimiter. This result is eventually compared to another string. Note that the “font” auxiliary is delimited by a space followed by “font”. This avoids an unnecessary check for the `\font` primitive below.

```

19237 \group_begin:
19238 \cs_set_protected:Npn \_token_tmp:w #1
19239  {
19240      \use:x
19241      {
19242          \cs_new:Npn \exp_not:c { _token_delimit_by_ #1 :w }
19243              #####1 \tl_to_str:n {#1} #####2 \s__token_stop
19244              { #####1 \tl_to_str:n {#1} }

```

```

19245      }
19246  }
19247 \__token_tmp:w { char" }
19248 \__token_tmp:w { count }
19249 \__token_tmp:w { dimen }
19250 \__token_tmp:w { ~ font }
19251 \__token_tmp:w { macro }
19252 \__token_tmp:w { muskip }
19253 \__token_tmp:w { skip }
19254 \__token_tmp:w { toks }
19255 \group_end:

```

(End definition for `\__token_delimit_by_char":w` and others.)

```

\token_if_chardef_p:N
\token_if_chardef:NTF
\token_if_mathchardef_p:N
\token_if_mathchardef:NTF
\token_if_long_macro_p:N
\token_if_long_macro:NTF
  \token_if_protected_macro_p:N
\token_if_protected_macro:NTF
  \token_if_protected_long_macro_p:N
  \token_if_protected_long_macro:NTF
\token_if_font_selection_p:N
\token_if_font_selection:NTF
\token_if_dim_register_p:N
\token_if_dim_register:NTF
\token_if_int_register_p:N
\token_if_int_register:NTF
  \token_if_muskip_register_p:N
\token_if_muskip_register:NTF
  \token_if_skip_register_p:N
\token_if_skip_register:NTF
\token_if_toks_register_p:N
\token_if_toks_register:NTF

```

Each of these conditionals tests whether its argument's `\meaning` starts with a given string. This is essentially done by having an auxiliary grab an argument delimited by the string and testing whether the argument was empty. Of course, a copy of this string must first be added to the end of the `\meaning` to avoid a runaway argument in case it does not contain the string. Two complications arise. First, the escape character is not fixed, and cannot be included in the delimiter of the auxiliary function (this function cannot be defined on the fly because tests must remain expandable): instead the first argument of the auxiliary (plus the delimiter to avoid complications with trailing spaces) is compared using `\str_if_eq:eeTF` to the result of applying `\token_to_str:N` to a control sequence. Second, the `\meaning` of primitives such as `\dimen` or `\dimendef` starts in the same way as registers such as `\dimen123`, so they must be tested for.

Characters used as delimiters must have catcode 12 and are obtained through `\tl_to_str:n`. This requires doing all definitions within x-expansion. The temporary function `\__token_tmp:w` used to define each conditional receives three arguments: the name of the conditional, the auxiliary's delimiter (also used to name the auxiliary), and the string to which one compares the auxiliary's result. Note that the `\meaning` of a protected long macro starts with `\protected\long macro`, with no space after `\protected` but a space after `\long`, hence the mixture of `\token_to_str:N` and `\tl_to_str:n`.

For the first six conditionals, `\cs_if_exist:cT` turns out to be `false` (thanks to the leading space for `font`), and the code boils down to a string comparison between the result of the auxiliary on the `\meaning` of the conditional's argument `####1`, and `#3`. Both are evaluated at run-time, as this is important to get the correct escape character.

The other five conditionals have additional code that compares the argument `####1` to two TeX primitives which would wrongly be recognized as registers otherwise. Despite using TeX's primitive conditional construction, this does not break when `####1` is itself a conditional, because branches of the conditionals are only skipped if `####1` is one of the two primitives that are tested for (which are not TeX conditionals).

```

19256 \group_begin:
19257 \cs_set_protected:Npn \__token_tmp:w #1#2#3
19258 {
19259   \use:x
19260   {
19261     \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } ####1
19262       { p , T , F , TF }
19263       {
19264         \cs_if_exist:cT { tex_ #2 :D }
19265         {
19266           \exp_not:N \if_meaning:w ####1 \exp_not:c { tex_ #2 :D }

```

```

19267   \exp_not:N \prg_return_false:
19268   \exp_not:N \else:
19269   \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 def:D }
19270   \exp_not:N \prg_return_false:
19271   \exp_not:N \else:
19272   }
19273   \exp_not:N \str_if_eq:eeTF
19274   {
19275     \exp_not:N \exp_after:wN
19276     \exp_not:c { __token_delimit_by_ #2 :w }
19277     \exp_not:N \token_to_meaning:N #####1
19278     ? \tl_to_str:n {#2} \s__token_stop
19279   }
19280   { \exp_not:n {#3} }
19281   { \exp_not:N \prg_return_true: }
19282   { \exp_not:N \prg_return_false: }
19283   \cs_if_exist:cT { tex_ #2 :D }
19284   {
19285     \exp_not:N \fi:
19286     \exp_not:N \fi:
19287   }
19288 }
19289 }
19290 }
19291 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
19292 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
19293 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
19294 \__token_tmp:w { protected_macro } { macro }
19295   { \tl_to_str:n { \protected } macro }
19296 \__token_tmp:w { protected_long_macro } { macro }
19297   { \token_to_str:N \protected \tl_to_str:n { \long } macro }
19298 \__token_tmp:w { font_selection } { ~ font } { select ~ font }
19299 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
19300 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
19301 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
19302 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
19303 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
19304 \group_end:

```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 190.)

```

\token_if_primitive_p:N
\token_if_primitive:NTF
\__token_if_primitive:Nw
  \__token_if_primitive space:w
  \__token_if_primitive_nullfont:N
\__token_if_primitive_loop:N
  \__token_if_primitive:Nw
    \__token_if_primitive_undefined:N
\__token_if_primitive_lua:N

```

We filter out macros first, because they cause endless trouble later otherwise.

Primitives are almost distinguished by the fact that the result of `\token_to_meaning:N` is formed from letters only. Every other token has either a space (e.g., the letter A), a digit (e.g., `\count123`) or a double quote (e.g., `\char"A`).

Ten exceptions: on the one hand, `\tex_undefined:D` is not a primitive, but its meaning is `undefined`, only letters; on the other hand, `\space`, `\italiccorr`, `\hyphen`, `\firstmark`, `\topmark`, `\botmark`, `\splitfirstmark`, `\splitbotmark`, and `\nullfont` are primitives, but have non-letters in their meaning.

We start by removing the two first (non-space) characters from the meaning. This removes the escape character (which may be nonexistent depending on `\endlinechar`), and takes care of three of the exceptions: `\space`, `\italiccorr` and `\hyphen`, whose meaning is at most two characters. This leaves a string terminated by some `:`, and `\s__token_stop`.

The meaning of each one of the five `\...mark` primitives has the form  $\langle letters \rangle : \langle user material \rangle$ . In other words, the first non-letter is a colon. We remove everything after the first colon.

We are now left with a string, which we must analyze. For primitives, it contains only letters. For non-primitives, it contains either ", or a space, or a digit. Two exceptions remain: `\tex_undefined:D`, which is not a primitive, and `\nullfont`, which is a primitive.

Spaces cannot be grabbed in an undelimited way, so we check them separately. If there is a space, we test for `\nullfont`. Otherwise, we go through characters one by one, and stop at the first character less than 'A' (this is not quite a test for "only letters", but is close enough to work in this context). If this first character is : then we have a primitive, or `\tex_undefined:D`, and if it is " or a digit, then the token is not a primitive.

For LuaTeX we use a different implementation which just looks at the command code for the token and compares it to a list of non-primitives. Again, `\nullfont` is a special case because it is the only primitive with the normally non-primitive `set_font` command code.

In LuaMetaTeX some of the command names are different, so we check for both versions. The first one is always the LuaTeX version.

```

19305  \sys_if_engine_luatex:TF
19306    {
19307    </tex>
19308    <*lua>
19309    do
19310      local get_next = token.get_next
19311      local get_command = token.get_command
19312      local get_index = token.get_index
19313      local get_mode = token.get_mode or token.get_index
19314      local cmd = command_id
19315      local set_font = cmd'get_font'
19316      local biggest_char = token.biggest_char and token.biggest_char()
19317          or status.getconstants().max_character_code
19318
19319      local mode_below_biggest_char = {}
19320      local index_not_nil = {}
19321      local mode_not_null = {}
19322      local non_primitive = {
19323        [cmd'left_brace'] = true,
19324        [cmd'right_brace'] = true,
19325        [cmd'math_shift'] = true,
19326        [cmd'mac_param' or cmd'parameter'] = mode_below_biggest_char,
19327        [cmd'sup_mark' or cmd'superscript'] = true,
19328        [cmd'sub_mark' or cmd'subscript'] = true,
19329        [cmd'endv' or cmd'ignore'] = true,
19330        [cmd'spacer'] = true,
19331        [cmd'letter'] = true,
19332        [cmd'other_char'] = true,
19333        [cmd'tab_mark' or cmd'alignment_tab'] = mode_below_biggest_char,
19334        [cmd'char_given'] = true,
19335        [cmd'math_given' or 'math_char_given'] = true,
19336        [cmd'xmath_given' or 'math_char_xgiven'] = true,
19337        [cmd'set_font'] = mode_not_null,
19338        [cmd'undefined_cs'] = true,

```

```

19339   [cmd'call'] = true,
19340   [cmd'long_call' or cmd'protected_call'] = true,
19341   [cmd'outer_call' or cmd'tolerant_call'] = true,
19342   [cmd'long_outer_call' or cmd'tolerant_protected_call'] = true,
19343   [cmd'assign_glue' or cmd'register_glue'] = index_not_nil,
19344   [cmd'assign_mu_glue' or cmd'register_mu_glue'] = index_not_nil,
19345   [cmd'assign_toks' or cmd'register_toks'] = index_not_nil,
19346   [cmd'assign_int' or cmd'register_int'] = index_not_nil,
19347   [cmd'assign_attr' or cmd'register_attribute'] = true,
19348   [cmd'assign_dimen' or cmd'register_dimen'] = index_not_nil,
19349 }
19350
19351 luacmd("_token_if_primitive_lua:N", function()
19352   local tok = get_next()
19353   local is_non_primitive = non_primitive[get_command(tok)]
19354   return put_next(
19355     is_non_primitive == true
19356     and false_tok
19357     or is_non_primitive == nil
19358     and true_tok
19359     or is_non_primitive == mode_not_null
19360     and (get_mode(tok) == 0 and true_tok or false_tok)
19361     or is_non_primitive == index_not_nil
19362     and (get_index(tok) and false_tok or true_tok)
19363     or is_non_primitive == mode_below_biggest_char
19364     and (get_mode(tok) > biggest_char and true_tok or false_tok))
19365   end, "global")
19366 end
19367 
```

*(\*tex)*

```

19368
19369   \prg_new_conditional:Npnn \token_if_primitive:N #1 { p , T , F , TF }
19370   {
19371     \_token_if_primitive_lua:N #1
19372   }
19373 }
19374 {
19375   \tex_chardef:D \c__token_A_int = 'A ~ %
19376   \use:x
19377   {
19378     \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
19379       { p , T , F , TF }
19380     {
19381       \exp_not:N \token_if_macro:NTF ##
19382       \exp_not:N \prg_return_false:
19383     {
19384       \exp_not:N \exp_after:wN \exp_not:N \_token_if_primitive:NNw
19385       \exp_not:N \token_to_meaning:N ##1
19386         \tl_to_str:n { : : : } \s__token_stop ##1
19387     }
19388   }
19389   \cs_new:Npn \exp_not:N \_token_if_primitive:NNw
19390     ##1##2 ##3 \c_colon_str ##4 \s__token_stop
19391   {
19392     \exp_not:N \tl_if_empty:oTF

```

```

19393 { \exp_not:N \_token_if_primitive_space:w ##3 ~ }
19394 {
19395     \exp_not:N \_token_if_primitive_loop:N ##3
19396         \c_colon_str \s_token_stop
19397     }
19398     { \exp_not:N \_token_if_primitive_nullfont:N }
19399 }
19400 }
19401 \cs_new:Npn \_token_if_primitive_space:w #1 ~ { }
19402 \cs_new:Npn \_token_if_primitive_nullfont:N #1
19403 {
19404     \if_meaning:w \tex_nullfont:D #1
19405         \prg_return_true:
19406     \else:
19407         \prg_return_false:
19408     \fi:
19409 }
19410 \cs_new:Npn \_token_if_primitive_loop:N #1
19411 {
19412     \if_int_compare:w '#1 < \c_token_A_int %
19413         \exp_after:wN \_token_if_primitive:Nw
19414         \exp_after:wN #1
19415     \else:
19416         \exp_after:wN \_token_if_primitive_loop:N
19417     \fi:
19418 }
19419 \cs_new:Npn \_token_if_primitive:Nw #1 #2 \s_token_stop
19420 {
19421     \if:w : #1
19422         \exp_after:wN \_token_if_primitive_undefined:N
19423     \else:
19424         \prg_return_false:
19425         \exp_after:wN \use_none:n
19426     \fi:
19427 }
19428 \cs_new:Npn \_token_if_primitive_undefined:N #1
19429 {
19430     \if_cs_exist:N #1
19431         \prg_return_true:
19432     \else:
19433         \prg_return_false:
19434     \fi:
19435 }
19436 }

```

(End definition for `\token_if_primitive:NTF` and others. This function is documented on page 191.)

`\token_case_catcode:Nn`  
`\token_case_catcode:NnTF`  
`\token_case_charcode:Nn`  
`\token_case_charcode:NnTF`  
`\token_case_meaning:Nn`  
`\token_case_meaning:NnTF`

The aim here is to allow the case statement to be evaluated using a known number of expansion steps (two), and without needing to use an explicit “end of recursion” marker. That is achieved by using the test input as the final case, as this is always true. The trick is then to tidy up the output such that the appropriate case code plus either the `true` or `false` branch code is inserted.

```

19437 \cs_new:Npn \token_case_catcode:Nn #1#2
19438     { \exp:w \_token_case:NnTF \token_if_eq_catcode:NNTF #1 {#2} { } { } }
\_token_case:NnTF
    \_token_case>NNnTF
        \_token_case:Nnw
\_token_case_end:nw

```

```

19439 \cs_new:Npn \token_case_catcode:NnT #1#2#3
19440   { \exp:w \__token_case>NNnTF \token_if_eq_catcode:NNTF #1 {#2} {#3} { } }
19441 \cs_new:Npn \token_case_catcode:NnF #1#2
19442   { \exp:w \__token_case>NNnTF \token_if_eq_catcode:NNTF #1 {#2} { } }
19443 \cs_new:Npn \token_case_catcode:NnTF
19444   { \exp:w \__token_case>NNnTF \token_if_eq_catcode:NNTF }
19445 \cs_new:Npn \token_case_charcode:Nn #1#2
19446   { \exp:w \__token_case>NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } { } }
19447 \cs_new:Npn \token_case_charcode:NnT #1#2#3
19448   { \exp:w \__token_case>NNnTF \token_if_eq_charcode:NNTF #1 {#2} {#3} { } }
19449 \cs_new:Npn \token_case_charcode:Nnf #1#2
19450   { \exp:w \__token_case>NNnTF \token_if_eq_charcode:NNTF #1 {#2} { } }
19451 \cs_new:Npn \token_case_charcode:NnT
19452   { \exp:w \__token_case>NNnTF \token_if_eq_charcode:NNTF }
19453 \cs_new:Npn \token_case_meaning:Nn #1#2
19454   { \exp:w \__token_case>NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } { } }
19455 \cs_new:Npn \token_case_meaning:NnT #1#2#3
19456   { \exp:w \__token_case>NNnTF \token_if_eq_meaning:NNTF #1 {#2} {#3} { } }
19457 \cs_new:Npn \token_case_meaning:Nnf #1#2
19458   { \exp:w \__token_case>NNnTF \token_if_eq_meaning:NNTF #1 {#2} { } }
19459 \cs_new:Npn \token_case_meaning:NnT
19460   { \exp:w \__token_case>NNnTF \token_if_eq_meaning:NNTF }
19461 \cs_new:Npn \__token_case>NNnTF #1#2#3#4#5
19462   {
19463     \__token_case>NNw #1 #2 #3 #2 { }
19464     \s__token_mark {#4}
19465     \s__token_mark {#5}
19466     \s__token_stop
19467   }
19468 \cs_new:Npn \__token_case>NNw #1#2#3#4
19469   {
19470     #1 #2 #3
19471     { \__token_case_end:nw {#4} }
19472     { \__token_case>NNw #1 #2 }
19473   }

```

To tidy up the recursion, there are two outcomes. If there was a hit to one of the cases searched for, then #1 is the code to insert, #2 is the *next* case to check on and #3 is all of the rest of the cases code. That means that #4 is the **true** branch code, and #5 tidies up the spare \s\_\_token\_mark and the **false** branch. On the other hand, if none of the cases matched then we arrive here using the “termination” case of comparing the search with itself. That means that #1 is empty, #2 is the first \s\_\_token\_mark and so #4 is the **false** code (the **true** code is mopped up by #3).

```

19474 \cs_new:Npn \__token_case_end:nw #1#2#3 \s__token_mark #4#5 \s__token_stop
19475   { \exp_end: #1 #4 }

```

(End definition for \token\_case\_catcode:NnTF and others. These functions are documented on page 191.)

## 60.6 Peeking ahead at the next token

```

19476 <@=peek>

```

Peeking ahead is implemented using a two part mechanism. The outer level provides a defined interface to the lower level material. This allows a large amount of code to be shared. There are four cases:

1. peek at the next token;
2. peek at the next non-space token;
3. peek at the next token and remove it;
4. peek at the next non-space token and remove it.

**\l\_peek\_token** Storage tokens which are publicly documented: the token peeked.

```
19477 \cs_new_eq:NN \l_peek_token ?
19478 \cs_new_eq:NN \g_peek_token ?
```

(End definition for **\l\_peek\_token** and **\g\_peek\_token**. These variables are documented on page 192.)

**\l\_\_peek\_search\_token** The token to search for as an implicit token: cf. **\l\_\_peek\_search\_t1**.

```
19479 \cs_new_eq:NN \l__peek_search_token ?
```

(End definition for **\l\_\_peek\_search\_token**.)

**\l\_\_peek\_search\_t1** The token to search for as an explicit token: cf. **\l\_\_peek\_search\_token**.

```
19480 \tl_new:N \l__peek_search_t1
```

(End definition for **\l\_\_peek\_search\_t1**.)

**\\_\\_peek\_true:w** Functions used by the branching and space-stripping code.

```
19481 \cs_new:Npn \_\_peek_true:w { }
19482 \cs_new:Npn \_\_peek_true_aux:w { }
19483 \cs_new:Npn \_\_peek_false:w { }
19484 \cs_new:Npn \_\_peek_tmp:w { }
```

(End definition for **\\_\\_peek\_true:w** and others.)

**\s\_\_peek\_mark** Internal scan marks.

```
19485 \scan_new:N \s__peek_mark
19486 \scan_new:N \s__peek_stop
```

(End definition for **\s\_\_peek\_mark** and **\s\_\_peek\_stop**.)

**\\_\\_peek\_use\_none\_delimit\_by\_s\_stop:w** Functions to gobble up to a scan mark.

```
19487 \cs_new:Npn \_\_peek_use_none_delimit_by_s_stop:w #1 \s__peek_stop { }
```

(End definition for **\\_\\_peek\_use\_none\_delimit\_by\_s\_stop:w**.)

**\peek\_after:Nw** Simple wrappers for **\futurelet**: no arguments absorbed here.

```
19488 \cs_new_protected:Npn \peek_after:Nw
19489   { \tex_futurelet:D \l_peek_token }
19490 \cs_new_protected:Npn \peek_gafter:Nw
19491   { \tex_global:D \tex_futurelet:D \g_peek_token }
```

(End definition for **\peek\_after:Nw** and **\peek\_gafter:Nw**. These functions are documented on page 191.)

\\_\\_peek\\_true\\_remove:w A function to remove the next token and then regain control.

```

19492 \cs_new_protected:Npn \_\_peek_true_remove:w
19493 {
19494     \tex_afterassignment:D \_\_peek_true_aux:w
19495     \cs_set_eq:NN \_\_peek_tmp:w
19496 }

```

(End definition for \\_\\_peek\\_true\\_remove:w.)

\peek\_remove\_spaces:n Repeatedly use \\_\\_peek\\_true\\_remove:w to remove a space and call \\_\\_peek\\_true\\_aux:w.

```

19497 \cs_new_protected:Npn \peek_remove_spaces:n #1
19498 {
19499     \cs_set:Npx \_\_peek_false:w { \exp_not:n {#1} }
19500     \group_align_safe_begin:
19501     \cs_set:Npn \_\_peek_true_aux:w { \peek_after:Nw \_\_peek_remove_spaces: }
19502     \_\_peek_true_aux:w
19503 }
19504 \cs_new_protected:Npn \_\_peek_remove_spaces:
19505 {
19506     \if_meaning:w \l_peek_token \c_space_token
19507         \exp_after:wN \_\_peek_true_remove:w
19508     \else:
19509         \group_align_safe_end:
19510         \exp_after:wN \_\_peek_false:w
19511     \fi:
19512 }

```

(End definition for \peek\_remove\_spaces:n and \\_\\_peek\_remove\_spaces:. This function is documented on page 193.)

\peek\_remove\_filler:n Here we expand the input, removing spaces and \scan\_stop: tokens until we reach a non-expandable token. At that stage we re-insert the payload. To deal with the problem of & tokens, we have to put the align-safe group in the correct place.

```

19513 \cs_new_protected:Npn \peek_remove_filler:n #1
19514 {
19515     \cs_set:Npn \_\_peek_true_aux:w { \_\_peek_remove_filler:w }
19516     \cs_set:Npx \_\_peek_false:w
19517     {
19518         \exp_not:N \group_align_safe_end:
19519         \exp_not:n {#1}
19520     }
19521     \group_align_safe_begin:
19522     \_\_peek_remove_filler:w
19523 }
19524 \cs_new_protected:Npn \_\_peek_remove_filler:w
19525 {
19526     \exp_after:wN \peek_after:Nw \exp_after:wN \_\_peek_remove_filler:
19527     \exp:w \exp_end_continue_f:w
19528 }

```

Here we can nest conditionals as \l\_peek\_token is only skipped over in the nested one if it's a space: no problems with conditionals or outer tokens.

```
19529 \cs_new_protected:Npn \_\_peek_remove_filler:
```

```

19530   {
19531     \if_catcode:w \exp_not:N \l_peek_token \c_space_token
19532       \exp_after:wN \__peek_true_remove:w
19533     \else:
19534       \if_meaning:w \l_peek_token \scan_stop:
19535         \exp_after:wN \exp_after:wN \exp_after:wN
19536           \__peek_true_remove:w
19537       \else:
19538         \exp_after:wN \exp_after:wN \exp_after:wN
19539           \__peek_remove_filler_expand:w
19540       \fi:
19541     \fi:
19542   }

```

To deal with undefined control sequences in the same way TeX does, we need to check for expansion manually.

```

19543 \cs_new_protected:Npn \__peek_remove_filler_expand:w
19544   {
19545     \exp_after:wN \if_meaning:w \exp_not:N \l_peek_token \l_peek_token
19546       \exp_after:wN \__peek_false:w
19547     \else:
19548       \exp_after:wN \__peek_remove_filler:w
19549     \fi:
19550   }

```

(End definition for `\__peek_remove_filler:n` and others. This function is documented on page 193.)

#### `\__peek_token_generic_aux:NNNTF`

The generic functions store the test token in both implicit and explicit modes, and the `true` and `false` code as token lists, more or less. The two branches have to be absorbed here as the input stream needs to be cleared for the peek function itself. Here, #1 is `\__peek_true_remove:w` when removing the token and `\__peek_true_aux:w` otherwise.

```

19551 \cs_new_protected:Npn \__peek_token_generic_aux:NNNTF #1#2#3#4#5
19552   {
19553     \group_align_safe_begin:
19554     \cs_set_eq:NN \l_peek_search_token #3
19555     \tl_set:Nn \l_peek_search_tl {#3}
19556     \cs_set:Npx \__peek_true_aux:w
19557     {
19558       \exp_not:N \group_align_safe_end:
19559       \exp_not:n {#4}
19560     }
19561     \cs_set_eq:NN \__peek_true:w #1
19562     \cs_set:Npx \__peek_false:w
19563     {
19564       \exp_not:N \group_align_safe_end:
19565       \exp_not:n {#5}
19566     }
19567     \peek_after:Nw #2
19568   }

```

(End definition for `\__peek_token_generic_aux:NNNTF`.)

#### `\__peek_token_generic:NNTF`

For token removal there needs to be a call to the auxiliary function which does the work.

```

\__peek_token_remove_generic:NNTF
19569 \cs_new_protected:Npn \__peek_token_generic:NNTF

```

```

19570   { \_peek_token_generic_aux:NNNTF \_peek_true_aux:w }
19571 \cs_new_protected:Npn \_peek_token_generic:NNTF #1#2#3
19572   { \_peek_token_generic:NNTF #1 #2 {#3} { } }
19573 \cs_new_protected:Npn \_peek_token_generic:NNF #1#2#3
19574   { \_peek_token_generic:NNF #1 #2 { } {#3} }
19575 \cs_new_protected:Npn \_peek_token_remove_generic:NNTF
19576   { \_peek_token_generic_aux:NNNTF \_peek_true_remove:w }
19577 \cs_new_protected:Npn \_peek_token_remove_generic:NNTF #1#2#3
19578   { \_peek_token_remove_generic:NNTF #1 #2 {#3} { } }
19579 \cs_new_protected:Npn \_peek_token_remove_generic:NNF #1#2#3
19580   { \_peek_token_remove_generic:NNF #1 #2 { } {#3} }

(End definition for \_peek_token_generic:NNTF and \_peek_token_remove_generic:NNTF.)

```

\\_peek\_execute\_branches\_meaning:

```

19581 \cs_new:Npn \_peek_execute_branches_meaning:
19582   {
19583     \if_meaning:w \l_peek_token \l_peek_search_token
19584       \exp_after:wN \_peek_true:w
19585     \else:
19586       \exp_after:wN \_peek_false:w
19587     \fi:
19588   }

```

(End definition for \\_peek\_execute\_branches\_meaning..)

```

\_peek_execute_branches_catcode:
\_peek_execute_branches_charcode:
\_peek_execute_branches_catcode_aux:
\_peek_execute_branches_catcode_auxii:N
\_peek_execute_branches_catcode_auxiii:

```

The catcode and charcode tests are very similar, and in order to use the same auxiliaries we do something a little bit odd, firing \if\_catcode:w and \if\_charcode:w before finding the operands for those tests, which are only given in the auxii:N and auxiii: auxiliaries. For our purposes, three kinds of tokens may follow the peeking function:

- control sequences which are not equal to a non-active character token (*e.g.*, macro, primitive);
- active characters which are not equal to a non-active character token (*e.g.*, macro, primitive);
- explicit non-active character tokens, or control sequences or active characters set equal to a non-active character token.

The first two cases are not distinguishable simply using TeX's \futurelet, because we can only access the \meaning of tokens in that way. In those cases, detected thanks to a comparison with \scan\_stop:, we grab the following token, and compare it explicitly with the explicit search token stored in \l\_peek\_search\_t1. The \exp\_not:N prevents outer macros (coming from non- $\text{\LaTeX}3$  code) from blowing up. In the third case, \l\_peek\_token is good enough for the test, and we compare it again with the explicit search token. Just like the peek token, the search token may be of any of the three types above, hence the need to use the explicit token that was given to the peek function.

```

19589 \cs_new:Npn \_peek_execute_branches_catcode:
19590   { \if_catcode:w \_peek_execute_branches_catcode_aux: }
19591 \cs_new:Npn \_peek_execute_branches_charcode:
19592   { \if_charcode:w \_peek_execute_branches_catcode_aux: }
19593 \cs_new:Npn \_peek_execute_branches_catcode_aux:
19594   {

```

```

19595     \if_catcode:w \exp_not:N \l_peek_token \scan_stop:
19596         \exp_after:wN \exp_after:wN
19597             \exp_after:wN \__peek_execute_branches_catcode_auxii:N
19598                 \exp_after:wN \exp_not:N
19599             \else:
1960                 \exp_after:wN \__peek_execute_branches_catcode_auxiii:
19601             \fi:
19602         }
19603     \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
19604     {
19605         \exp_not:N #1
19606         \exp_after:wN \exp_not:N \l_peek_search_tl
19607             \exp_after:wN \__peek_true:w
19608             \else:
19609                 \exp_after:wN \__peek_false:w
19610             \fi:
19611             #1
19612         }
19613     \cs_new:Npn \__peek_execute_branches_catcode_auxiii:
19614     {
19615         \exp_not:N \l_peek_token
19616         \exp_after:wN \exp_not:N \l_peek_search_tl
19617             \exp_after:wN \__peek_true:w
19618             \else:
19619                 \exp_after:wN \__peek_false:w
19620             \fi:
19621         }

```

(End definition for `\__peek_execute_branches_catcode:` and others.)

**\peek\_catcode:NTF** The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`. Instead, the TF, T, F variants are defined in terms of corresponding variants of `\__peek_token_generic:NNTF` or `\__peek_token_remove_generic:NNTF`, with first argument one of `\__peek_execute_branches_catcode:`, `\__peek_execute_branches_charcode:`, or `\__peek_execute_branches_meaning:`.

```

19622     \tl_map_inline:nn { { catcode } { charcode } { meaning } }
19623     {
19624         \tl_map_inline:nn { { } { _remove } }
19625         {
19626             \tl_map_inline:nn { { TF } { T } { F } }
19627             {
19628                 \cs_new_protected:cp { peek_ #1 ##1 :N #####1 }
19629                 {
19630                     \exp_not:c { __peek_token ##1 _generic:NN #####1 }
19631                     \exp_not:c { __peek_execute_branches_ #1 : }
19632                 }
19633             }
19634         }
19635     }

```

(End definition for `\peek_catcode:NTF` and others. These functions are documented on page 192.)

**\peek\_N\_type:TF** All tokens are N-type tokens, except in four cases: begin-group tokens, end-group tokens, space tokens with character code 32, and outer tokens. Since `\l_peek_token` might be `\__peek_execute_branches_N_type:`

```

        \__peek_N_type:w
\__peek_N_type_aux:nw

```

outer, we cannot use the convenient `\bool_if:nTF` function, and must resort to the old trick of using `\ifodd` to expand a set of tests. The `false` branch of this test is taken if the token is one of the first three kinds of non-N-type tokens (explicit or implicit), thus we call `\_peek_false:w`. In the `true` branch, we must detect outer tokens, without impacting performance too much for non-outer tokens. The first filter is to search for `outer` in the `\meaning` of `\l_peek_token`. If that is absent, `\_peek_use_none_delimit_by_s_stop:w` cleans up, and we call `\_peek_true:w`. Otherwise, the token can be a non-outer macro or a primitive mark whose parameter or replacement text contains `outer`, it can be the primitive `\outer`, or it can be an outer token. Macros and marks would have `ma` in the part before the first occurrence of `outer`; the meaning of `\outer` has nothing after `outer`, contrarily to outer macros; and that covers all cases, calling `\_peek_true:w` or `\_peek_false:w` as appropriate. Here, there is no `(search token)`, so we feed a dummy `\scan_stop:` to the `\_peek_token_generic:NNTF` function.

```

19636 \group_begin:
19637   \cs_set_protected:Npn \_peek_tmp:w #1 \s_peek_stop
19638   {
19639     \cs_new_protected:Npn \_peek_execute_branches_N_type:
19640     {
19641       \if_int_odd:w
19642         \if_catcode:w \exp_not:N \l_peek_token { \c_zero_int \fi:
19643         \if_catcode:w \exp_not:N \l_peek_token } \c_zero_int \fi:
19644         \if_meaning:w \l_peek_token \c_space_token \c_zero_int \fi:
19645         \c_one_int
19646         \exp_after:wn \_peek_N_type:w
19647           \token_to_meaning:N \l_peek_token
19648           \s_peek_mark \_peek_N_type_aux:nnw
19649           #1 \s_peek_mark \_peek_use_none_delimit_by_s_stop:w
19650           \s_peek_stop
19651           \exp_after:wn \_peek_true:w
19652         \else:
19653           \exp_after:wn \_peek_false:w
19654         \fi:
19655       }
19656       \cs_new_protected:Npn \_peek_N_type:w ##1 #1 ##2 \s_peek_mark ##3
19657       { ##3 {##1} {##2} }
19658     }
19659     \exp_after:wn \_peek_tmp:w \tl_to_str:n { outer } \s_peek_stop
19660   \group_end:
19661   \cs_new_protected:Npn \_peek_N_type_aux:nnw #1 #2 #3 \fi:
19662   {
19663     \fi:
19664     \tl_if_in:notF {#1} { \tl_to_str:n {ma} }
19665     { \_peek_true:w }
19666     { \tl_if_empty:nTF {#2} { \_peek_true:w } { \_peek_false:w } }
19667   }
19668   \cs_new_protected:Npn \peek_N_type:TF
19669   {
19670     \_peek_token_generic:NNTF
19671       \_peek_execute_branches_N_type: \scan_stop:
19672   }
19673   \cs_new_protected:Npn \peek_N_type:T
19674     { \_peek_token_generic:NNT \_peek_execute_branches_N_type: \scan_stop: }
19675   \cs_new_protected:Npn \peek_N_type:F

```

19676 { \\_\\_peek\\_token\\_generic:NNF \\_\\_peek\\_execute\\_branches\\_N\\_type: \scan\_stop: }

*(End definition for \peek\_N\_type:TF and others. This function is documented on page 193.)*

19677 </tex>

19678 </package>

# Chapter 61

## I3prop implementation

The following test files are used for this code: *m3prop001*, *m3prop002*, *m3prop003*, *m3prop004*, *m3show001*.

19679 `<*package>`

19680 `<@@=prop>`

A property list is a macro whose top-level expansion is of the form

`\s__prop \__prop_pair:wn <key1> \s__prop {<value1>}`

`...`

`\__prop_pair:wn <keyn> \s__prop {<valuen>}`

where `\s__prop` is a scan mark (equal to `\scan_stop:`), and `\__prop_pair:wn` can be used to map through the property list.

`\s__prop` The internal token used at the beginning of property lists. This is also used after each `<key>` (see `\__prop_pair:wn`).

(End definition for `\s__prop`.)

`\__prop_pair:wn` `\__prop_pair:wn <key> \s__prop {<item>}`

The internal token used to begin each key–value pair in the property list. If expanded outside of a mapping or manipulation function, an error is raised. The definition should always be set globally.

(End definition for `\__prop_pair:wn`.)

`\l__prop_internal_tl` Token list used to store new key–value pairs to be inserted by functions of the `\prop_put:Nnn` family.

(End definition for `\l__prop_internal_tl`.)

<code>\__prop_split:NnTF</code>	<code>\__prop_split:NnTF &lt;property list&gt; {&lt;key&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
<small>Updated: 2013-01-08</small>	Splits the <i>&lt;property list&gt;</i> at the <i>&lt;key&gt;</i> , giving three token lists: the <i>&lt;extract&gt;</i> of <i>&lt;property list&gt;</i> before the <i>&lt;key&gt;</i> , the <i>&lt;value&gt;</i> associated with the <i>&lt;key&gt;</i> and the <i>&lt;extract&gt;</i> of the <i>&lt;property list&gt;</i> after the <i>&lt;value&gt;</i> . Both <i>&lt;extracts&gt;</i> retain the internal structure of a property list, and the concatenation of the two <i>&lt;extracts&gt;</i> is a property list. If the <i>&lt;key&gt;</i> is present in the <i>&lt;property list&gt;</i> then the <i>&lt;true code&gt;</i> is left in the input stream, with #1, #2, and #3 replaced by the first <i>&lt;extract&gt;</i> , the <i>&lt;value&gt;</i> , and the second extract. If the <i>&lt;key&gt;</i> is not present in the <i>&lt;property list&gt;</i> then the <i>&lt;false code&gt;</i> is left in the input stream, with no trailing material. Both <i>&lt;true code&gt;</i> and <i>&lt;false code&gt;</i> are used in the replacement text of a macro defined internally, hence macro parameter characters should be doubled, except #1, #2, and #3 which stand in the <i>&lt;true code&gt;</i> for the three extracts from the property list. The <i>&lt;key&gt;</i> comparison takes place as described for <code>\str_if_eq:nn</code> .
<code>\s__prop</code>	A private scan mark is used as a marker after each key, and at the very beginning of the property list.  19681 <code>\scan_new:N \s__prop</code>  (End definition for <code>\s__prop</code> .)
<code>\__prop_pair:wn</code>	The delimiter is always defined, but when misused simply triggers an error and removes its argument.  19682 <code>\cs_new:Npn \__prop_pair:wn #1 \s__prop #2</code> 19683   { <code>\msg_expandable_error:nn { prop } { misused }</code> }  (End definition for <code>\__prop_pair:wn</code> .)
<code>\l__prop_internal_tl</code>	Token list used to store the new key–value pair inserted by <code>\prop_put:Nnn</code> and friends.  19684 <code>\tl_new:N \l__prop_internal_tl</code>  (End definition for <code>\l__prop_internal_tl</code> .)
<code>\c_empty_prop</code>	An empty prop.  19685 <code>\tl_const:Nn \c_empty_prop { \s__prop }</code>  (End definition for <code>\c_empty_prop</code> . This variable is documented on page 206.)

## 61.1 Internal auxiliaries

<code>\s__prop_mark</code>	Internal scan marks.
<code>\s__prop_stop</code>	19686 <code>\scan_new:N \s__prop_mark</code> 19687 <code>\scan_new:N \s__prop_stop</code>  (End definition for <code>\s__prop_mark</code> and <code>\s__prop_stop</code> .)
<code>\q__prop_recursion_tail</code>	Internal recursion quarks.
<code>\q__prop_recursion_stop</code>	19688 <code>\quark_new:N \q__prop_recursion_tail</code> 19689 <code>\quark_new:N \q__prop_recursion_stop</code>  (End definition for <code>\q__prop_recursion_tail</code> and <code>\q__prop_recursion_stop</code> .)
<code>\__prop_if_recursion_tail_stop:n</code> <code>\__prop_if_recursion_tail_stop:o</code>	Functions to query recursion quarks.  19690 <code>\__kernel_quark_new_test:N \__prop_if_recursion_tail_stop:n</code> 19691 <code>\cs_generate_variant:Nn \__prop_if_recursion_tail_stop:n { o }</code>  (End definition for <code>\__prop_if_recursion_tail_stop:n</code> and <code>\__prop_if_recursion_tail_stop:o</code> .)

## 61.2 Allocation and initialisation

**\prop\_new:N** Property lists are initialized with the value \c\_empty\_prop.

```

19692 \cs_new_protected:Npn \prop_new:N #1
19693 {
19694     \__kernel_chk_if_free_cs:N #1
19695     \cs_gset_eq:NN #1 \c_empty_prop
19696 }
19697 \cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for \prop\_new:N. This function is documented on page 199.)

**\prop\_clear:N** The same idea for clearing.

```

19698 \cs_new_protected:Npn \prop_clear:N #1
19699 {
19700     \prop_set_eq:NN #1 \c_empty_prop
19701 \cs_generate_variant:Nn \prop_clear:N { c }
19702 \cs_new_protected:Npn \prop_gclear:N #1
19703 {
19704     \prop_gset_eq:NN #1 \c_empty_prop
19705 }
19706 \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for \prop\_clear:N and \prop\_gclear:N. These functions are documented on page 199.)

**\prop\_clear\_new:N** Once again a simple variation of the token list functions.

```

19704 \cs_new_protected:Npn \prop_clear_new:N #1
19705 {
19706     \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 }
19707 \cs_generate_variant:Nn \prop_clear_new:N { c }
19708 \cs_new_protected:Npn \prop_gclear_new:N #1
19709 {
19710     \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 }
19711 }
19712 \cs_generate_variant:Nn \prop_gclear_new:N { c }
```

(End definition for \prop\_clear\_new:N and \prop\_gclear\_new:N. These functions are documented on page 199.)

**\prop\_set\_eq:NN** These are simply copies from the token list functions.

```

19710 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
19711 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
19712 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
19713 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
19714 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
19715 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
19716 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
19717 \cs_new_eq:NN \prop_gset_eq:cc \tl_gset_eq:cc
```

(End definition for \prop\_set\_eq:NN and \prop\_gset\_eq:NN. These functions are documented on page 200.)

**\l\_tmpa\_prop** We can now initialize the scratch variables.

```

19718 \prop_new:N \l_tmpa_prop
19719 \prop_new:N \l_tmpb_prop
19720 \prop_new:N \g_tmpa_prop
19721 \prop_new:N \g_tmpb_prop
```

(End definition for \l\_tmpa\_prop and others. These variables are documented on page 206.)

**\l\_\_prop\_internal\_prop** Property list used by \prop\_concat:NNN, \prop\_set\_from\_keyval:Nn and others.

```
19722 \prop_new:N \l__prop_internal_prop
```

(End definition for `\l_prop_internal_prop`.)

```
\prop_concat:NNN
\prop_concat:ccc
\prop_gconcat:NNN
\prop_gconcat:ccc
\_prop_concat:NNNN
19723 \cs_new_protected:Npn \prop_concat:NNN
19724   { \__prop_concat:NNNN \prop_set_eq:NN }
19725 \cs_generate_variant:Nn \prop_concat:NNN { ccc }
19726 \cs_new_protected:Npn \prop_gconcat:NNN
19727   { \__prop_concat:NNNN \prop_gset_eq:NN }
19728 \cs_generate_variant:Nn \prop_gconcat:NNN { ccc }
19729 \cs_new_protected:Npn \__prop_concat:NNNN #1#2#3#4
19730   {
19731     \prop_set_eq:NN \l__prop_internal_prop #3
19732     \prop_map_inline:Nn #4 { \prop_put:Nnn \l__prop_internal_prop {##1} {##2} }
19733     #1 #2 \l__prop_internal_prop
19734 }
```

(End definition for `\prop_concat:NNN`, `\prop_gconcat:NNN`, and `\_prop_concat:NNNN`. These functions are documented on page 201.)

```
\prop_set_from_keyval:Nn
\prop_set_from_keyval:cn
\prop_gset_from_keyval:Nn
\prop_gset_from_keyval:cn
\prop_const_from_keyval:Nn
\prop_const_from_keyval:cn
  \prop_put_from_keyval:Nn
  \prop_put_from_keyval:cn
\prop_gput_from_keyval:Nn
\prop_gput_from_keyval:cn
  \__prop_missing_eq:n
```

To avoid tracking throughout the loop the variable name and whether the assignment is local/global, do everything in a scratch variable and empty it afterwards to avoid wasting memory. Loop through items separated by commas, with `\prg_do_nothing`: to avoid losing braces. After checking for termination, split the item at the first and then at the second = (which ought to be the first of the trailing = that we added). For both splits trim spaces and call a function (first `\__prop_from_keyval_key:w` then `\__prop_from_keyval_value:w`), followed by the trimmed material, `\s__prop_mark`, the subsequent part of the item, and the trailing =’s and `\s__prop_stop`. After finding the `\langle key` just store it after `\s__prop_stop`. After finding the `\langle value` ignore completely empty items (both trailing = were used as delimiters and all parts are empty); if the remaining part #2 consists exactly of the second trailing = (namely there was exactly one = in the item) then output one key–value pair for the property list; otherwise complain about a missing or extra =.

```
19735 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1
19736   {
19737     \prop_clear:N #1
19738     \prop_put_from_keyval:Nn #1
19739   }
19740 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
19741 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1
19742   {
19743     \prop_gclear:N #1
19744     \prop_gput_from_keyval:Nn #1
19745   }
19746 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
19747 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1#2
19748   {
19749     \prop_set_from_keyval:Nn \l__prop_internal_prop {#2}
19750     \tl_const:Nx #1 { \exp_not:o \l__prop_internal_prop }
19751     \prop_clear:N \l__prop_internal_prop
19752   }
19753 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
19754 \cs_new_protected:Npn \prop_put_from_keyval:Nn
```

```

19755   {
19756     \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
19757     { \__prop_keyval_parse:NNNn \c_true_bool }
19758     { \__prop_keyval_parse:NNNn \c_false_bool }
19759     \prop_put:Nnn
19760   }
19761 \cs_generate_variant:Nn \prop_put_from_keyval:Nn { c }
19762 \cs_new_protected:Npn \prop_gput_from_keyval:Nn
19763   {
19764     \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool
19765     { \__prop_keyval_parse:NNNn \c_true_bool }
19766     { \__prop_keyval_parse:NNNn \c_false_bool }
19767     \prop_gput:Nnn
19768   }
19769 \cs_generate_variant:Nn \prop_gput_from_keyval:Nn { c }
19770 \cs_new_protected:Npn \__prop_missing_eq:n
19771   { \msg_error:nnn { prop } { prop-keyval } }
19772 \cs_new_protected:Npn \__prop_keyval_parse:NNNn #1#2#3#4
19773   {
19774     \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool \c_true_bool
19775     \keyval_parse:nnn \__prop_missing_eq:n { #2 #3 } { #4 }
19776     \bool_set_eq:NN \l__kernel_keyval_allow_blank_keys_bool #1
19777   }

```

(End definition for `\prop_set_from_keyval:Nn` and others. These functions are documented on page 200.)

## 61.3 Accessing data in property lists

`\__prop_split:NnTF`  
`\__prop_split_aux:NnTF`  
`\__prop_split_aux:w`

This function is used by most of the module, and hence must be fast. It receives a *property list*, a *key*, a *true code* and a *false code*. The aim is to split the *property list* at the given *key* into the *extract<sub>1</sub>* before the key–value pair, the *value* associated with the *key* and the *extract<sub>2</sub>* after the key–value pair. This is done using a delimited function, whose definition is as follows, where the *key* is turned into a string.

```

\cs_set:Npn \__prop_split_aux:w #1
\__prop_pair:wn <key> \s__prop #2
#3 \s__prop_mark #4 #5 \s__prop_stop
{ #4 {<true code>} {<false code>} }

```

If the *key* is present in the property list, `\__prop_split_aux:w`'s #1 is the part before the *key*, #2 is the *value*, #3 is the part after the *key*, #4 is `\use_i:nn`, and #5 is additional tokens that we do not care about. The *true code* is left in the input stream, and can use the parameters #1, #2, #3 for the three parts of the property list as desired. Namely, the original property list is in this case #1 `\__prop_pair:wn <key> \s__prop {#2} #3`.

If the *key* is not there, then the *function* is `\use_ii:nn`, which keeps the *false code*.

```

19778 \cs_new_protected:Npn \__prop_split:NnTF #1#2
19779   { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
19780 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
19781   {
19782     \cs_set:Npn \__prop_split_aux:w ##1

```

```

19783     \_prop_pair:wn #2 \s__prop ##3 \s__prop_mark ##4 ##5 \s__prop_stop
19784     { ##4 {#3} {#4} }
19785     \exp_after:wN \_prop_split_aux:w #1 \s__prop_mark \use_i:nn
19786     \_prop_pair:wn #2 \s__prop { } \s__prop_mark \use_i:nn \s__prop_stop
19787   }
19788 \cs_new:Npn \_prop_split_aux:w { }

```

(End definition for `\_prop_split:NnTF`, `\_prop_split_aux:NnTF`, and `\_prop_split_aux:w`.)

**\prop\_remove:Nn** Deleting from a property starts by splitting the list. If the key is present in the property list, the returned value is ignored. If the key is missing, nothing happens.

```

19789 \cs_new_protected:Npn \prop_remove:Nn #1#2
19790   {
19791     \_prop_split:NnTF #1 {#2}
19792     { \tl_set:Nn #1 { ##1 ##3 } }
19793     { }
19794   }
19795 \cs_new_protected:Npn \prop_gremove:Nn #1#2
19796   {
19797     \_prop_split:NnTF #1 {#2}
19798     { \tl_gset:Nn #1 { ##1 ##3 } }
19799     { }
19800   }
19801 \cs_generate_variant:Nn \prop_remove:Nn { NV }
19802 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
19803 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
19804 \cs_generate_variant:Nn \prop_gremove:Nn { c , cV }

```

(End definition for `\prop_remove:Nn` and `\prop_gremove:Nn`. These functions are documented on page 202.)

**\prop\_get:NnN** Getting an item from a list is very easy: after splitting, if the key is in the property list, just set the token list variable to the return value, otherwise to `\q_no_value`.

```

19805 \cs_new_protected:Npn \prop_get:NnN #1#2#3
19806   {
19807     \_prop_split:NnTF #1 {#2}
19808     { \tl_set:Nn #3 {##2} }
19809     { \tl_set:Nn #3 { \q_no_value } }
19810   }
19811 \cs_generate_variant:Nn \prop_get:NnN { NV , Nv , No }
19812 \cs_generate_variant:Nn \prop_get:NnN { c , cV , cv , co }

```

(End definition for `\prop_get:NnN`. This function is documented on page 201.)

**\prop\_pop:NnN** Popping a value also starts by doing the split. If the key is present, save the value in the token list and update the property list as when deleting. If the key is missing, save `\q_no_value` in the token list.

```

19813 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
19814   {
19815     \_prop_split:NnTF #1 {#2}
19816     {
19817       \tl_set:Nn #3 {##2}
19818       \tl_set:Nn #1 { ##1 ##3 }
19819     }

```

```

19820      { \tl_set:Nn #3 { \q_no_value } }
19821    }
19822 \cs_new_protected:Npn \prop_gpop:NnN #1#2#3
19823  {
19824    \__prop_split:NnTF #1 {#2}
19825    {
19826      \tl_set:Nn #3 {##2}
19827      \tl_gset:Nn #1 { ##1 ##3 }
19828    }
19829    { \tl_set:Nn #3 { \q_no_value } }
19830  }
19831 \cs_generate_variant:Nn \prop_pop:NnN { No }
19832 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
19833 \cs_generate_variant:Nn \prop_gpop:NnN { No }
19834 \cs_generate_variant:Nn \prop_gpop:NnN { c , co }

```

(End definition for `\prop_pop:NnN` and `\prop_gpop:NnN`. These functions are documented on page 201.)

**\prop\_item:Nn**  
**\prop\_item:cN**  
**\\_\_prop\_item:nnn**

Getting the value corresponding to a key in a property list in an expandable fashion simply uses `\prop_map_tokens:Nn` to go through the property list. The auxiliary `\__prop_item:nnn` receives the search string `#1`, the key `#2` and the value `#3` and returns as appropriate.

```

19835 \cs_new:Npn \prop_item:Nn #1#2
19836  {
19837    \exp_args:NNo \prop_map_tokens:Nn #1
19838    { \exp_after:wN \__prop_item:nnn \exp_after:wN { \tl_to_str:n {#2} } }
19839  }
19840 \cs_new:Npn \__prop_item:nnn #1#2#3
19841  {
19842    \str_if_eq:eeT {#1} {#2}
19843    { \prop_map_break:n { \exp_not:n {#3} } }
19844  }
19845 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `\__prop_item:nnn`. This function is documented on page 202.)

**\prop\_count:N**  
**\prop\_count:cN**  
**\\_\_prop\_count:nn**

Counting the key–value pairs in a property list is done using the same approach as for other count functions: turn each entry into a `+1` then use integer evaluation to actually do the mathematics.

```

19846 \cs_new:Npn \prop_count:N #1
19847  {
19848    \int_eval:n
19849    {
19850      0
19851      \prop_map_function:NN #1 \__prop_count:nn
19852    }
19853  }
19854 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
19855 \cs_generate_variant:Nn \prop_count:N { c }

```

(End definition for `\prop_count:N` and `\__prop_count:nn`. This function is documented on page 202.)

**\prop\_to\_keyval:N**  
**\\_\_prop\_to\_keyval\_exp\_after:wN**  
**\\_\_prop\_to\_keyval:nn**  
**\\_\_prop\_to\_keyval:nw**

Each property name and value pair will be returned in the form `\{<name>\}=<value>`. As one of the main use cases for this macro is to pass the *property list* on to a key–value parser, we have to make sure that the behaviour is as good as possible. Using a

space before the opening brace we get the correct brace stripping behaviour for most of the key-value parsers available in L<sup>A</sup>T<sub>E</sub>X. If `\tex_expanded:D` is available this function makes use of it, so there are two different implementations here. They both start with `\_kernel_exp_not:w` to start the expansion context to expand in two steps. If the *property list* is empty they just leave an empty set of braces in the input stream for `\_kernel_exp_not:w`.

```
19856 \cs_if_exist:NTF \tex_expanded:D
19857   {
```

The variant using `\tex_expanded:D` can just iterate over the *property list* and remove the leading comma afterwards. Only the value has to be protected in `\_kernel_exp_not:w` as the property name is always a string. After the loop the leading comma is removed by `\use_none:n` and afterwards `\_kernel_exp_not:w` eventually finds the opening brace of its argument.

```
19858   \cs_new:Npn \prop_to_keyval:N #1
19859     {
19860       \_kernel_exp_not:w
19861         \prop_if_empty:NTF #1
19862           { {} }
19863           {
19864             \exp_after:wN \exp_after:wN \exp_after:wN
19865             {
19866               \tex_expanded:D
19867                 {
19868                   \_kernel_exp_not:w { \use_none:n }
19869                   \prop_map_function:NN #1 \_prop_to_keyval:nn
19870                 }
19871               }
19872             }
19873           }
19874     \cs_new:Npn \_prop_to_keyval:nn #1#2
19875       { , ~ {#1} =~ { \_kernel_exp_not:w {#2} } }
19876   }
```

The other variant will iterate over the *property list* and has to output the result in a group after the marker `\_prop_to_keyval_exp_after:wN`. As a result this is considerably slower than the `\tex_expanded:D` using variant as it has to read the entire contents of the *property list* for each item. Since the marker is just `\exp_after:wN` with another name, after the loop the leading comma is gobbled by `\use_none:n`, leaving the result as the argument to `\_kernel_exp_not:w`.

```
19877   {
19878     \cs_new:Npn \prop_to_keyval:N #1
19879       {
19880         \_kernel_exp_not:w
19881           \prop_if_empty:NTF #1
19882             { {} }
19883             {
19884               \prop_map_function:NN #1 \_prop_to_keyval:nnw
19885               \_prop_to_keyval_exp_after:wN { \use_none:n }
19886             }
19887           }
19888     \cs_new_eq:NN \_prop_to_keyval_exp_after:wN \exp_after:wN
19889     \cs_new:Npn \_prop_to_keyval:nnw #1#2#3 \_prop_to_keyval_exp_after:wN #4
```

```

19890     { #3 \__prop_to_keyval_exp_after:wN { #4 , ~ {#1} =~ {#2} } }
19891 }
```

(End definition for `\prop_to_keyval:N` and others. This function is documented on page 202.)

**\prop\_pop:NnNTF** Popping an item from a property list, keeping track of whether the key was present or not, is implemented as a conditional. If the key was missing, neither the property list, nor the token list are altered. Otherwise, `\prg_return_true:` is used after the assignments.

```

19892 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
19893 {
19894     \__prop_split:NnTF #1 {#2}
19895     {
19896         \tl_set:Nn #3 {##2}
19897         \tl_set:Nn #1 { ##1 ##3 }
19898         \prg_return_true:
19899     }
19900     { \prg_return_false: }
19901 }
```

```

19902 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
19903 {
19904     \__prop_split:NnTF #1 {#2}
19905     {
19906         \tl_set:Nn #3 {##2}
19907         \tl_gset:Nn #1 { ##1 ##3 }
19908         \prg_return_true:
19909     }
19910     { \prg_return_false: }
19911 }
```

```

19912 \prg_generate_conditional_variant:Nnn \prop_pop:NnN { c } { T , F , TF }
19913 \prg_generate_conditional_variant:Nnn \prop_gpop:NnN { c } { T , F , TF }
```

(End definition for `\prop_pop:NnNTF` and `\prop_gpop:NnNTF`. These functions are documented on page 203.)

**\prop\_put:Nnn** Since the branches of `\__prop_split:NnTF` are used as the replacement text of an internal macro, and since the `\langle key\rangle` and new `\langle value\rangle` may contain arbitrary tokens, it is not safe to include them in the argument of `\__prop_split:NnTF`. We thus start by storing in `\l__prop_internal_tl` tokens which (after x-expansion) encode the key–value pair. This variable can safely be used in `\__prop_split:NnTF`. If the `\langle key\rangle` was absent, append the new key–value to the list. Otherwise concatenate the extracts `##1` and `##3` with the new key–value pair `\l__prop_internal_tl`. The updated entry is placed at the same spot as the original `\langle key\rangle` in the property list, preserving the order of entries.

```

19914 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put>NNnn \__kernel_tl_set:Nx }
19915 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put>NNnn \__kernel_tl_gset:Nx }
19916 \cs_new_protected:Npn \__prop_put>NNnn #1#2#3#4
19917 {
19918     \tl_set:Nn \l__prop_internal_tl
19919     {
19920         \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
19921         \s_prop { \exp_not:n {#4} }
19922     }
19923     \__prop_split:NnTF #2 {#3}
19924     { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
19925     { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
```

**\prop\_gput:Nnn**

```

\prop_gput:NnV
\prop_gput:Nno
\prop_gput:Nnx
\prop_gput:NVn
\prop_gput:NVV
\prop_gput:Non
```

```

19926      }
19927  \cs_generate_variant:Nn \prop_put:Nnn
19928  { NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
19929  \cs_generate_variant:Nn \prop_put:Nnn
19930  { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }
19931  \cs_generate_variant:Nn \prop_gput:Nnn
19932  { NnV , Nno , Nnx , NV , NVV , NVx , Nvx , No , Noo , Nxx }
19933  \cs_generate_variant:Nn \prop_gput:Nnn
19934  { c , cnV , cno , cnx , cV , cVV , cVx , cvx , co , coo , cxx }

```

(End definition for `\prop_put:Nnn` and others. These functions are documented on page 200.)

`\prop_put_if_new:Nnn`  
`\prop_put_if_new:cnn`  
`\prop_gput_if_new:Nnn`  
`\prop_gput_if_new:cnn`  
`\__prop_put_if_new>NNnn`

```

19935 \cs_new_protected:Npn \prop_put_if_new:Nnn
19936  { \__prop_put_if_new:NNnn \__kernel_tl_set:Nx }
19937 \cs_new_protected:Npn \prop_gput_if_new:Nnn
19938  { \__prop_put_if_new:NNnn \__kernel_tl_gset:Nx }
19939 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
19940  {
19941    \tl_set:Nn \l__prop_internal_tl
19942    {
19943      \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
19944      \s__prop \exp_not:n {#4}
19945    }
19946    \__prop_split:NnTF #2 {#3}
19947    {
19948      { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
19949    }
19950 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
19951 \cs_generate_variant:Nn \prop_gput_if_new:Nnn { c }

```

(End definition for `\prop_put_if_new:Nnn`, `\prop_gput_if_new:Nnn`, and `\__prop_put_if_new>NNnn`. These functions are documented on page 201.)

## 61.4 Property list conditionals

`\prop_if_exist_p:N`  
`\prop_if_exist_p:c`  
`\prop_if_exist:NTF`  
`\prop_if_exist:cTF`

```

19952 \prg_new_eq_conditional>NNnn \prop_if_exist:N \cs_if_exist:N
19953 { TF , T , F , p }
19954 \prg_new_eq_conditional>NNnn \prop_if_exist:c \cs_if_exist:c
19955 { TF , T , F , p }

```

(End definition for `\prop_if_exist:NTF`. This function is documented on page 202.)

`\prop_if_empty_p:N`  
`\prop_if_empty_p:c`  
`\prop_if_empty:NTF`  
`\prop_if_empty:cTF`

```

19956 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
19957  {
19958    \tl_if_eq:NNTF #1 \c_empty_prop
19959    \prg_return_true: \prg_return_false:
19960  }
19961 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
19962 { c } { p , T , F , TF }

```

(End definition for `\prop_if_empty:N`. This function is documented on page 203.)

`\prop_if_in_p:Nn`  
`\prop_if_in_p:NV`  
`\prop_if_in_p:No`  
`\prop_if_in_p:cN`  
`\prop_if_in_p:cV`  
`\prop_if_in_p:co`  
`\prop_if_in:NnTF`  
`\prop_if_in:NVT`  
`\prop_if_in:NoT`  
`\prop_if_in:cNT`  
`\prop_if_in:cVT`  
`\prop_if_in:coT`  
`\_prop_if_in:nnn`

```
  \prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
    {
      \__split:NnTF #1 {#2}
      { \prg_return_true: }
      { \prg_return_false: }
    }
```

but `\_prop_split:N` is non-expandable. Instead, we use `\prop_map_tokens:N` to compare the search key to each key in turn using `\str_if_eq:ee`, which is expandable.

```
19963 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
19964   {
19965     \exp_args:NNo \prop_map_tokens:Nn #1
19966     { \exp_after:wN \_prop_if_in:nnn \exp_after:wN { \tl_to_str:n {#2} } }
19967     \prg_return_false:
19968   }
19969   \cs_new:Npn \_prop_if_in:nnn #1#2#3
19970   {
19971     \str_if_eq:eeT {#1} {#2}
19972     { \prop_map_break:n { \use_i:nn \prg_return_true: } }
19973   }
19974   \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
19975     { NV , No , c , cV , co } { p , T , F , TF }
```

(End definition for `\prop_if_in:NnTF` and `\_prop_if_in:nnn`. This function is documented on page 203.)

## 61.5 Recovering values from property lists with branching

`\prop_get:NnNTF`  
`\prop_get:NVNTF`  
`\prop_get:NoNTF` Getting the value corresponding to a key, keeping track of whether the key was present or not, is implemented as a conditional (with side effects). If the key was absent, the token list is not altered.

```
19976 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
19977   {
19978     \__prop_split:NnTF #1 {#2}
19979     {
19980       \tl_set:Nn #3 {##2}
19981       \prg_return_true:
19982     }
19983     { \prg_return_false: }
19984   }
19985   \prg_generate_conditional_variant:Nnn \prop_get:NnN
19986     { NV , Nv , No , c , cV , cv , co } { T , F , TF }
```

(End definition for `\prop_get:NnNTF`. This function is documented on page 203.)

## 61.6 Mapping over property lists

\prop\_map\_function:NN  
\prop\_map\_function:Nc  
\prop\_map\_function:cN  
\prop\_map\_function:cc

```

19987 \cs_new:Npn \prop_map_function:NN #1#2
19988 {
19989     \exp_after:wN \use_i_i:nnn
19990     \exp_after:wN \_prop_map_function:Nw
19991     \exp_after:wN #2
19992     #1
19993     \_prop_pair:wn \fi: \prop_map_break: \s_prop { }
19994     \_prop_pair:wn \fi: \prop_map_break: \s_prop { }
19995     \_prop_pair:wn \fi: \prop_map_break: \s_prop { }
19996     \_prop_pair:wn \fi: \prop_map_break: \s_prop { }
19997     \prg_break_point:Nn \prop_map_break: { }
19998 }
19999 \cs_new:Npn \_prop_map_function:Nw #1
20000     \_prop_pair:wn #2 \s_prop #3
20001     \_prop_pair:wn #4 \s_prop #5
20002     \_prop_pair:wn #6 \s_prop #7
20003     \_prop_pair:wn #8 \s_prop #9
20004 {
20005     \if_false: #2 \fi: #1 {#2} {#3}
20006     \if_false: #4 \fi: #1 {#4} {#5}
20007     \if_false: #6 \fi: #1 {#6} {#7}
20008     \if_false: #8 \fi: #1 {#8} {#9}
20009     \_prop_map_function:Nw #1
20010 }
20011 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }
```

(End definition for \prop\_map\_function:NN and \\_prop\_map\_function:Nw. This function is documented on page 204.)

\prop\_map\_inline:Nn  
\prop\_map\_inline:cN

Mapping in line requires a nesting level counter. Store the current definition of \\_prop\_pair:wn, and define it anew. At the end of the loop, revert to the earlier definition. Note that besides pairs of the form \\_prop\_pair:wn *key* \s\_prop {*value*}, there are a leading and a trailing tokens, but both are equal to \scan\_stop:, hence have no effect in such inline mapping. Such \scan\_stop: could have affected ligatures if they appeared during the mapping.

```

20012 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
20013 {
20014     \cs_gset_eq:cN
20015     { __prop_map_ \int_use:N \g_kernel_prg_map_int :wn } \_prop_pair:wn
20016     \int_gincr:N \g_kernel_prg_map_int
20017     \cs_gset_protected:Npn \_prop_pair:wn ##1 \s_prop ##2 {#2}
20018     #1
20019     \prg_break_point:Nn \prop_map_break:
20020     {
20021         \int_gdecr:N \g_kernel_prg_map_int
20022         \cs_gset_eq:Nc \_prop_pair:wn
20023             { __prop_map_ \int_use:N \g_kernel_prg_map_int :wn }
```

```

20024     }
20025   }
20026 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 204.)

`\prop_map_tokens:Nn`  
`\prop_map_tokens:cn`  
`\_prop_map_tokens:nw`

The mapping is very similar to `\prop_map_function:NN`. The `\use_i:nn` removes the leading `\s__prop`. The odd construction `\use:n {#1}` allows #1 to contain any token without interfering with `\prop_map_break:`. The loop stops when the `\langle key\rangle` between `\_prop_pair:wn` and `\s__prop` is `\fi: \prop_map_break:` instead of being a string.

```

20027 \cs_new:Npn \prop_map_tokens:Nn #1#2
20028   {
20029     \exp_last_unbraced:Nno
20030       \use_i:nn { \_prop_map_tokens:nw {#2} } #1
20031     \_prop_pair:wn \fi: \prop_map_break: \s__prop { }
20032     \_prop_pair:wn \fi: \prop_map_break: \s__prop { }
20033     \_prop_pair:wn \fi: \prop_map_break: \s__prop { }
20034     \_prop_pair:wn \fi: \prop_map_break: \s__prop { }
20035     \prg_break_point:Nn \prop_map_break: { }
20036   }
20037 \cs_new:Npn \_prop_map_tokens:nw #1
20038   \_prop_pair:wn #2 \s__prop #3
20039   \_prop_pair:wn #4 \s__prop #5
20040   \_prop_pair:wn #6 \s__prop #7
20041   \_prop_pair:wn #8 \s__prop #9
20042   {
20043     \if_false: #2 \fi: \use:n {#1} {#2} {#3}
20044     \if_false: #4 \fi: \use:n {#1} {#4} {#5}
20045     \if_false: #6 \fi: \use:n {#1} {#6} {#7}
20046     \if_false: #8 \fi: \use:n {#1} {#8} {#9}
20047     \_prop_map_tokens:nw {#1}
20048   }
20049 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }

```

(End definition for `\prop_map_tokens:Nn` and `\_prop_map_tokens:nw`. This function is documented on page 204.)

`\prop_map_break:`  
`\prop_map_break:n`

The break statements are based on the general `\prg_map_break:Nn`.

```

20050 \cs_new:Npn \prop_map_break:
20051   { \prg_map_break:Nn \prop_map_break: { } }
20052 \cs_new:Npn \prop_map_break:n
20053   { \prg_map_break:Nn \prop_map_break: }

```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 205.)

## 61.7 Viewing property lists

`\prop_show:N`  
`\prop_show:c`  
`\prop_log:N`  
`\prop_log:c`  
`\_prop_show>NN`  
`\_prop_show_validate:w`

Apply the general `\_kernel_chk_tl_type:NnnT`. Contrarily to sequences and comma lists, we use `\msg_show_item:nn` to format both the key and the value for each pair.

```

20054 \cs_new_protected:Npn \prop_show:N { \_prop_show:NN \msg_show:nnxxxxx }
20055 \cs_generate_variant:Nn \prop_show:N { c }
20056 \cs_new_protected:Npn \prop_log:N { \_prop_show:NN \msg_log:nnxxxxx }

```

```

20057 \cs_generate_variant:Nn \prop_log:N { c }
20058 \cs_new_protected:Npn \__prop_show:NN #1#2
20059   {
20060     \__kernel_chk_tl_type:NnnT #2 { prop }
20061     {
20062       \s__prop
20063       \exp_after:wN \use_i:nn \exp_after:wN \__prop_show_validate:w #2
20064       \__prop_pair:wn \q_recursion_tail \s__prop { } \q_recursion_stop
20065     }
20066     {
20067       #1 { prop } { show }
20068       { \token_to_str:N #2 }
20069       { \prop_map_function:NN #2 \msg_show_item:nn }
20070       { } { }
20071     }
20072   }
20073 \cs_new:Npn \__prop_show_validate:w #1 \__prop_pair:wn #2 \s__prop #3
20074   {
20075     \quark_if_recursion_tail_stop:n {#2}
20076     \exp_not:N \__prop_pair:wn \tl_to_str:n {#2} \s__prop \exp_not:n { {#3} }
20077     \__prop_show_validate:w
20078   }

```

(End definition for `\prop_show:N` and others. These functions are documented on page [205](#).)

```
20079 </package>
```

# Chapter 62

## I3skip implementation

```
20080  <*package>
20081  <@@=dim>
```

### 62.1 Length primitives renamed

```
\if_dim:w Primitives renamed.
\__dim_eval:w 20082 \cs_new_eq:NN \if_dim:w      \tex_ifdim:D
\__dim_eval_end: 20083 \cs_new_eq:NN \__dim_eval:w     \tex_dimexpr:D
20084 \cs_new_eq:NN \__dim_eval_end:   \tex_relax:D

(End definition for \if_dim:w, \__dim_eval:w, and \__dim_eval_end:. This function is documented on
page 221.)
```

### 62.2 Internal auxiliaries

```
\s_dim_mark Internal scan marks.
\s_dim_stop 20085 \scan_new:N \s_dim_mark
20086 \scan_new:N \s_dim_stop

(End definition for \s_dim_mark and \s_dim_stop.)
```

```
\__dim_use_none_delimit_by_s_stop:w Functions to gobble up to a scan mark.
20087 \cs_new:Npn \__dim_use_none_delimit_by_s_stop:w #1 \s_dim_stop { }

(End definition for \__dim_use_none_delimit_by_s_stop:w.)
```

### 62.3 Creating and initialising dim variables

```
\dim_new:N Allocating <dim> registers ...
\dim_new:c 20088 \cs_new_protected:Npn \dim_new:N #1
20089  {
20090      \__kernel_chk_if_free_cs:N #1
20091      \cs:w newdimen \cs_end: #1
20092  }
20093 \cs_generate_variant:Nn \dim_new:N { c }
```

(End definition for `\dim_new:N`. This function is documented on page 207.)

`\dim_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. We cannot use `\dim_gset:Nn` because debugging code would complain that the constant is not a global variable. Since `\dim_const:Nn` does not need to be fast, use `\dim_eval:n` to avoid needing a debugging patch that wraps the expression in checking code.

```
20094 \cs_new_protected:Npn \dim_const:Nn #1#2
20095   {
20096     \dim_new:N #1
20097     \tex_global:D #1 = \dim_eval:n {#2} \scan_stop:
20098   }
20099 \cs_generate_variant:Nn \dim_const:Nn { c }
```

(End definition for `\dim_const:Nn`. This function is documented on page 207.)

`\dim_zero:N` Reset the register to zero. Using `\c_zero_skip` deals with the case where the variable passed is incorrectly a skip (for example a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> length). Besides, these functions are then simply copied for `\skip_zero:N` and related functions.

`\dim_gzero:c`

```
20100 \cs_new_protected:Npn \dim_zero:N #1 { #1 = \c_zero_skip }
20101 \cs_new_protected:Npn \dim_gzero:N #1
20102   { \tex_global:D #1 = \c_zero_skip }
20103 \cs_generate_variant:Nn \dim_zero:N { c }
20104 \cs_generate_variant:Nn \dim_gzero:N { c }
```

(End definition for `\dim_zero:N` and `\dim_gzero:N`. These functions are documented on page 207.)

`\dim_zero_new:N` Create a register if needed, otherwise clear it.

`\dim_zero_new:c`

```
20105 \cs_new_protected:Npn \dim_zero_new:N #1
20106   { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
20107 \cs_new_protected:Npn \dim_gzero_new:N #1
20108   { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
20109 \cs_generate_variant:Nn \dim_zero_new:N { c }
20110 \cs_generate_variant:Nn \dim_gzero_new:N { c }
```

(End definition for `\dim_zero_new:N` and `\dim_gzero_new:N`. These functions are documented on page 207.)

`\dim_if_exist_p:N` Copies of the `cs` functions defined in l3basics.

`\dim_if_exist_p:c`

```
20111 \prg_new_eq_conditional:NNN \dim_if_exist:N \cs_if_exist:N
20112   { TF , T , F , p }
20113 \prg_new_eq_conditional:NNN \dim_if_exist:c \cs_if_exist:c
20114   { TF , T , F , p }
```

(End definition for `\dim_if_exist:NTF`. This function is documented on page 208.)

## 62.4 Setting dim variables

`\dim_set:Nn` Setting dimensions is easy enough but when debugging we want both to check that the variable is correctly local/global and to wrap the expression in some code. The `\scan_stop:` deals with the case where the variable passed is a skip (for example a L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> length).

```
20115 \cs_new_protected:Npn \dim_set:Nn #1#2
20116   { #1 = \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
```

```

20117 \cs_new_protected:Npn \dim_gset:Nn #1#2
20118   { \tex_global:D #1 = \_dim_eval:w #2 \_dim_eval_end: \scan_stop: }
20119 \cs_generate_variant:Nn \dim_set:Nn { c }
20120 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 208.)

`\dim_set_eq:NN` All straightforward, with a `\scan_stop:` to deal with the case where #1 is (incorrectly) a skip.

```

20121 \cs_new_protected:Npn \dim_set_eq:NN #1#2
20122   { #1 = #2 \scan_stop: }
20123 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
20124 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
20125   { \tex_global:D #1 = #2 \scan_stop: }
20126 \cs_generate_variant:Nn \dim_gset_eq:NN { c , Nc , cc }

```

(End definition for `\dim_set_eq:NN` and `\dim_gset_eq:NN`. These functions are documented on page 208.)

`\dim_add:Nn` Using `by` here would slow things down just to detect nonsensical cases such as passing `\dimen 123` as the first argument. Using `\scan_stop:` deals with skip variables. Since debugging checks that the variable is correctly local/global, the global versions cannot be defined as `\tex_global:D` followed by the local versions.

```

20127 \cs_new_protected:Npn \dim_add:Nn #1#2
20128   { \tex_advance:D #1 \_dim_eval:w #2 \_dim_eval_end: \scan_stop: }
20129 \cs_new_protected:Npn \dim_gadd:Nn #1#2
20130   {
20131     \tex_global:D \tex_advance:D #1
20132     \_dim_eval:w #2 \_dim_eval_end: \scan_stop:
20133   }
20134 \cs_generate_variant:Nn \dim_add:Nn { c }
20135 \cs_generate_variant:Nn \dim_gadd:Nn { c }
20136 \cs_new_protected:Npn \dim_sub:Nn #1#2
20137   { \tex_advance:D #1 - \_dim_eval:w #2 \_dim_eval_end: \scan_stop: }
20138 \cs_new_protected:Npn \dim_gsub:Nn #1#2
20139   {
20140     \tex_global:D \tex_advance:D #1
20141     -\_dim_eval:w #2 \_dim_eval_end: \scan_stop:
20142   }
20143 \cs_generate_variant:Nn \dim_sub:Nn { c }
20144 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 208.)

## 62.5 Utilities for dimension calculations

`\dim_abs:n` Functions for min, max, and absolute value with only one evaluation. The absolute value is evaluated by removing a leading `-` if present.

```

20145 \cs_new:Npn \dim_abs:n #1
20146   {
20147     \exp_after:wN \_dim_abs:N
20148     \dim_use:N \_dim_eval:w #1 \_dim_eval_end:
20149   }

```

```

20150 \cs_new:Npn \__dim_abs:N #1
20151   { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
20152 \cs_new:Npn \dim_max:nn #1#2
20153   {
20154     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
20155       \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
20156       \dim_use:N \__dim_eval:w #2 ;
20157     >
20158     \__dim_eval_end:
20159   }
20160 \cs_new:Npn \dim_min:nn #1#2
20161   {
20162     \dim_use:N \__dim_eval:w \exp_after:wN \__dim_maxmin:wwN
20163       \dim_use:N \__dim_eval:w #1 \exp_after:wN ;
20164       \dim_use:N \__dim_eval:w #2 ;
20165     <
20166     \__dim_eval_end:
20167   }
20168 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
20169   {
20170     \if_dim:w #1 #3 #2 ~
20171       #1
20172     \else:
20173       #2
20174     \fi:
20175   }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page [208](#).)

**\dim\_ratio:nn**  
**\\_\_dim\_ratio:n**

With dimension expressions, something like `10 pt * ( 5 pt / 10 pt )` does not work. Instead, the ratio part needs to be converted to an integer expression. Using `\int_value:w` forces everything into sp, avoiding any decimal parts.

```

20176 \cs_new:Npn \dim_ratio:nn #1#2
20177   { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
20178 \cs_new:Npn \__dim_ratio:n #1
20179   { \int_value:w \__dim_eval:w (#1) \__dim_eval_end: }

```

(End definition for `\dim_ratio:nn` and `\__dim_ratio:n`. This function is documented on page [209](#).)

## 62.6 Dimension expression conditionals

**\dim\_compare\_p:nNn** Simple comparison.

**\dim\_compare:nNnTF**

```

20180 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
20181   {
20182     \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
20183     \prg_return_true: \else: \prg_return_false: \fi:
20184   }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page [209](#).)

**\dim\_compare\_p:n**  
**\dim\_compare:nTF**

This code is adapted from the `\int_compare:nTF` function. First make sure that there is at least one relation operator, by evaluating a dimension expression with a trailing `\__dim_compare_error:`. Just like for integers, the looping auxiliary `\__dim_compare:wNN` closes a primitive conditional and opens a new one. It is actually easier to

`\__dim_compare:w`  
`\__dim_compare_=:w`  
`\__dim_compare_!=w`  
`\__dim_compare_<:w`  
`\__dim_compare_>:w`  
`\__dim_compare_error:`

grab a dimension operand than an integer one, because once evaluated, dimensions all end with `pt` (with category other). Thus we do not need specific auxiliaries for the three “simple” relations `<`, `=`, and `>`.

```

20185 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
20186 {
20187   \exp_after:wN \__dim_compare:w
20188   \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
20189 }
20190 \cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
20191 {
20192   \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
20193     \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \s_dim_stop
20194 }
20195 \exp_args:Nno \use:nn
20196 { \cs_new:Npn \__dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
20197 {
20198   \if_meaning:w = #3
20199     \use:c { __dim_compare_#2:w }
20200   \fi:
20201   #1 pt \exp_stop_f:
20202   \prg_return_false:
20203   \exp_after:wN \__dim_use_none_delimit_by_s_stop:w
20204   \fi:
20205   \reverse_if:N \if_dim:w #1 pt #2
20206     \exp_after:wN \__dim_compare:wNN
20207     \dim_use:N \__dim_eval:w #3
20208   }
20209   \cs_new:cpn { __dim_compare_ ! :w }
20210   #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
20211   \cs_new:cpn { __dim_compare_ = :w }
20212   #1 \__dim_eval:w = { #1 \__dim_eval:w }
20213   \cs_new:cpn { __dim_compare_ < :w }
20214   #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
20215   \cs_new:cpn { __dim_compare_ > :w }
20216   #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
20217   \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \s_dim_stop
20218   { #1 \prg_return_false: \else: \prg_return_true: \fi: }
20219   \cs_new_protected:Npn \__dim_compare_error:
20220   {
20221     \if_int_compare:w \c_zero_int \c_zero_int \fi:
20222     =
20223     \__dim_compare_error:
20224   }

```

(End definition for `\dim_compare:nTF` and others. This function is documented on page 210.)

### `\dim_case:nn`

### `\dim_case:nnTF`

```

\__dim_case:nntF
\__dim_case:nw
\__dim_case_end:nw
20225 \cs_new:Npn \dim_case:nnTF #1
20226 {
20227   \exp:w
20228   \exp_args:Nf \__dim_case:nntF { \dim_eval:n {#1} }
20229 }
2030 \cs_new:Npn \dim_case:nnT #1#2#3

```

For dimension cases, the first task to fully expand the check condition. The over all idea is then much the same as for `\str_case:nnTF` as described in l3basics.

```

20231   {
20232     \exp:w
20233     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
20234   }
20235 \cs_new:Npn \dim_case:nnF #1#2
20236   {
20237     \exp:w
20238     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }
20239   }
20240 \cs_new:Npn \dim_case:nn #1#2
20241   {
20242     \exp:w
20243     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
20244   }
20245 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
20246   { \__dim_case:nw {#1} #2 {#1} { } \s_dim_mark {#3} \s_dim_mark {#4} \s_dim_stop }
20247 \cs_new:Npn \__dim_case:nw #1#2#3
20248   {
20249     \dim_compare:nNnTF {#1} = {#2}
20250     { \__dim_case_end:nw {#3} }
20251     { \__dim_case:nw {#1} }
20252   }
20253 \cs_new:Npn \__dim_case_end:nw #1#2#3 \s_dim_mark #4#5 \s_dim_stop
20254   { \exp_end: #1 #4 }

```

(End definition for `\dim_case:nnTF` and others. This function is documented on page 211.)

## 62.7 Dimension expression loops

`\dim_while_do:nn` while\_do and do\_while functions for dimensions. Same as for the int type only the names have changed.

```

20255 \cs_new:Npn \dim_while_do:nn #1#2
20256   {
20257     \dim_compare:nT {#1}
20258     {
20259       #2
20260       \dim_while_do:nn {#1} {#2}
20261     }
20262   }
20263 \cs_new:Npn \dim_until_do:nn #1#2
20264   {
20265     \dim_compare:nF {#1}
20266     {
20267       #2
20268       \dim_until_do:nn {#1} {#2}
20269     }
20270   }
20271 \cs_new:Npn \dim_do_while:nn #1#2
20272   {
20273     #2
20274     \dim_compare:nT {#1}
20275     { \dim_do_while:nn {#1} {#2} }
20276   }

```

```

20277 \cs_new:Npn \dim_do_until:nn #1#2
20278 {
20279     #2
20280     \dim_compare:nF {#1}
20281         { \dim_do_until:nn {#1} {#2} }
20282 }
```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 212.)

`\dim_while_do:nNnn`  
`\dim_until_do:nNnn`

`\dim_do_while:nNnn`  
`\dim_do_until:nNnn`

```

20283 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
20284 {
20285     \dim_compare:nNnT {#1} #2 {#3}
20286     {
20287         #4
20288         \dim_while_do:nNnn {#1} #2 {#3} {#4}
20289     }
20290 }
20291 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
20292 {
20293     \dim_compare:nNnF {#1} #2 {#3}
20294     {
20295         #4
20296         \dim_until_do:nNnn {#1} #2 {#3} {#4}
20297     }
20298 }
20299 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
20300 {
20301     #4
20302     \dim_compare:nNnT {#1} #2 {#3}
20303         { \dim_while:nNnn {#1} #2 {#3} {#4} }
20304 }
20305 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
20306 {
20307     #4
20308     \dim_compare:nNnF {#1} #2 {#3}
20309         { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
20310 }
```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 212.)

## 62.8 Dimension step functions

`\dim_step_function:nnnN`  
`\_dim_step:wwwN`  
`\_dim_step:NnnnN`

Before all else, evaluate the initial value, step, and final value. Repeating a function by steps first needs a check on the direction of the steps. After that, do the function for the start value then step and loop around. It would be more symmetrical to test for a step size of zero before checking the sign, but we optimize for the most frequent case (positive step).

```

20311 \cs_new:Npn \dim_step_function:nnnN #1#2#3
20312 {
20313     \exp_after:wN \_dim_step:wwwN
20314     \tex_the:D \_dim_eval:w #1 \exp_after:wN ;
```

```

20315     \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
20316     \tex_the:D \__dim_eval:w #3 ;
20317 }
20318 \cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
20319 {
20320     \dim_compare:nNnTF {#2} > \c_zero_dim
20321     { \__dim_step:NnnnN > }
20322     {
20323         \dim_compare:nNnTF {#2} = \c_zero_dim
20324         {
20325             \msg_expandable_error:nnn { kernel } { zero-step } { #4 }
20326             \use_none:nnnn
20327         }
20328         { \__dim_step:NnnnN < }
20329     }
20330     {#1} {#2} {#3} #4
20331 }
20332 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
20333 {
20334     \dim_compare:nNnF {#2} #1 {#4}
20335     {
20336         #5 {#2}
20337         \exp_args:NNf \__dim_step:NnnnN
20338         #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
20339     }
20340 }

```

(End definition for `\dim_step_function:nnnN`, `\__dim_step:wwwN`, and `\__dim_step:NnnnN`. This function is documented on page 212.)

```

\dim_step_inline:nnn
\dim_step_variable:nnnNn
\__dim_step>NNnnnn

```

The approach here is to build a function, with a global integer required to make the nesting safe (as seen in other in line functions), and map that function using `\dim_step_function:nnnN`. We put a `\prg_break_point:Nn` so that `map_break` functions from other modules correctly decrement `\g_kernel_prg_map_int` before looking for their own break point. The first argument is `\scan_stop:`, so that no breaking function recognizes this break point as its own.

```

20341 \cs_new_protected:Npn \dim_step_inline:nnn
20342 {
20343     \int_gincr:N \g_kernel_prg_map_int
20344     \exp_args:NNc \__dim_step>NNnnnn
20345     \cs_gset_protected:Npn
20346     { \__dim_map_ \int_use:N \g_kernel_prg_map_int :w }
20347 }
20348 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
20349 {
20350     \int_gincr:N \g_kernel_prg_map_int
20351     \exp_args:NNc \__dim_step>NNnnnn
20352     \cs_gset_protected:Npx
20353     { \__dim_map_ \int_use:N \g_kernel_prg_map_int :w }
20354     {#1}{#2}{#3}
20355     {
20356         \tl_set:Nn \exp_not:N #4 {##1}
20357         \exp_not:n {#5}
20358     }

```

```

20359      }
20360  \cs_new_protected:Npn \__dim_step>NNnnnn #1#2#3#4#5#6
20361  {
20362      #1 #2 ##1 {#6}
20363      \dim_step_function:nnnN {#3} {#4} {#5} #2
20364      \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
20365  }

```

(End definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnNn`, and `\__dim_step>NNnnnn`. These functions are documented on page 212.)

## 62.9 Using dim expressions and variables

`\dim_eval:n` Evaluating a dimension expression expandably.

```

20366 \cs_new:Npn \dim_eval:n #1
20367  { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 213.)

`\dim_sign:n` See `\dim_abs:n`. Contrarily to `\int_sign:n` the case of a zero dimension cannot be distinguished from a positive dimension by looking only at the first character, since 0.2pt and Opt start the same way. We need explicit comparisons. We start by distinguishing the most common case of a positive dimension.

```

20368 \cs_new:Npn \dim_sign:n #1
20369  {
20370      \int_value:w \exp_after:wN \__dim_sign:Nw
20371      \dim_use:N \__dim_eval:w #1 \__dim_eval_end: ;
20372      \exp_stop_f:
20373  }
20374 \cs_new:Npn \__dim_sign:Nw #1#2 ;
20375  {
20376      \if_dim:w #1#2 > \c_zero_dim
20377          1
20378      \else:
20379          \if_meaning:w - #1
20380              -1
20381          \else:
20382              0
20383          \fi:
20384      \fi:
20385  }

```

(End definition for `\dim_sign:n` and `\__dim_sign:Nw`. This function is documented on page 213.)

`\dim_use:N` Accessing a  $\langle dim \rangle$ . We hand-code the c variant for some speed gain.

```

\dim_use:c 20386 \cs_new_eq:NN \dim_use:N \tex_the:D
20387 \cs_new:Npn \dim_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }

```

(End definition for `\dim_use:N`. This function is documented on page 213.)

`\dim_to_decimal:n` `\__dim_to_decimal:w` A function which comes up often enough to deserve a place in the kernel. Evaluate the dimension expression `#1` then remove the trailing pt. When debugging is enabled, the argument is put in parentheses as this prevents the dimension expression from terminating early and leaving extra tokens lying around. This is used a lot by low-level manipulations.

```

20388 \cs_new:Npn \dim_to_decimal:n #1
20389   {
20390     \exp_after:wN
20391       \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
20392   }
20393 \use:x
20394   {
20395     \cs_new:Npn \exp_not:N \__dim_to_decimal:w
20396       ##1 . ##2 \tl_to_str:n { pt }
20397   }
20398   {
20399     \int_compare:nNnTF {#2} > \c_zero_int
20400       { #1 . #2 }
20401       { #1 }
20402   }

```

(End definition for `\dim_to_decimal:n` and `\__dim_to_decimal:w`. This function is documented on page 213.)

**\dim\_to\_decimal\_in\_bp:n** Conversion to big points is done using a scaling inside `\__dim_eval:w` as  $\varepsilon$ -TeX does that using 64-bit precision. Here, 800/803 is the integer fraction for  $72/72.27$ . This is a common case so is hand-coded for accuracy (and speed).

```

20403 \cs_new:Npn \dim_to_decimal_in_bp:n #1
20404   { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }

```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 214.)

**\dim\_to\_decimal\_in\_sp:n** Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

20405 \cs_new:Npn \dim_to_decimal_in_sp:n #1
20406   { \int_value:w \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_to_decimal_in_sp:n`. This function is documented on page 214.)

**\dim\_to\_decimal\_in\_unit:nn** An analogue of `\dim_ratio:nn` that produces a decimal number as its result, rather than a rational fraction for use within dimension expressions.

```

20407 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
20408   {
20409     \dim_to_decimal:n
20410       {
20411         1pt *
20412         \dim_ratio:nn {#1} {#2}
20413       }
20414   }

```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 214.)

**\dim\_to\_fp:n** Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 214.)

## 62.10 Viewing dim variables

`\dim_show:N` Diagnostics.

`\dim_show:c` 20415 `\cs_new_eq:NN \dim_show:N \_kernel_register_show:N`  
20416 `\cs_generate_variant:Nn \dim_show:N { c }`

(End definition for `\dim_show:N`. This function is documented on page 214.)

`\dim_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show dimension expressions: this gives a more unified output.

20417 `\cs_new_protected:Npn \dim_show:n`  
20418 `{ \msg_show_eval:Nn \dim_eval:n }`

(End definition for `\dim_show:n`. This function is documented on page 215.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.

`\dim_log:c` 20419 `\cs_new_eq:NN \dim_log:N \_kernel_register_log:N`  
`\dim_log:n` 20420 `\cs_new_eq:NN \dim_log:c \_kernel_register_log:c`  
20421 `\cs_new_protected:Npn \dim_log:n`  
20422 `{ \msg_log_eval:Nn \dim_eval:n }`

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 215.)

## 62.11 Constant dimensions

`\c_zero_dim` Constant dimensions.

`\c_max_dim` 20423 `\dim_const:Nn \c_zero_dim { 0 pt }`  
20424 `\dim_const:Nn \c_max_dim { 16383.99999 pt }`

(End definition for `\c_zero_dim` and `\c_max_dim`. These variables are documented on page 215.)

## 62.12 Scratch dimensions

`\l_tmpa_dim` We provide two local and two global scratch registers, maybe we need more or less.

`\l_tmpb_dim` 20425 `\dim_new:N \l_tmpa_dim`  
`\g_tmpa_dim` 20426 `\dim_new:N \l_tmpb_dim`  
`\g_tmpb_dim` 20427 `\dim_new:N \g_tmpa_dim`  
20428 `\dim_new:N \g_tmpb_dim`

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 215.)

## 62.13 Creating and initialising skip variables

20429 `\@@=skip`

`\s__skip_stop` Internal scan marks.

20430 `\scan_new:N \s__skip_stop`

(End definition for `\s__skip_stop`.)

```

\skip_new:N Allocation of a new internal registers.
\skip_new:c 20431 \cs_new_protected:Npn \skip_new:N #1
20432 {
20433     \__kernel_chk_if_free_cs:N #1
20434     \cs:w newskip \cs_end: #1
20435 }
20436 \cs_generate_variant:Nn \skip_new:N { c }


```

(End definition for `\skip_new:N`. This function is documented on page 215.)

`\skip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. See `\dim_const:Nn` for why we cannot use `\skip_gset:Nn`.

```

20437 \cs_new_protected:Npn \skip_const:Nn #1#2
20438 {
20439     \skip_new:N #1
20440     \tex_global:D #1 = \skip_eval:n {#2} \scan_stop:
20441 }
20442 \cs_generate_variant:Nn \skip_const:Nn { c }


```

(End definition for `\skip_const:Nn`. This function is documented on page 216.)

`\skip_zero:N` Reset the register to zero.

```

\skip_zero:c 20443 \cs_new_eq:NN \skip_zero:N \dim_zero:N
\skip_gzero:N 20444 \cs_new_eq:NN \skip_gzero:N \dim_gzero:N
\skip_gzero:c 20445 \cs_generate_variant:Nn \skip_zero:N { c }
20446 \cs_generate_variant:Nn \skip_gzero:N { c }


```

(End definition for `\skip_zero:N` and `\skip_gzero:N`. These functions are documented on page 216.)

`\skip_zero_new:N` Create a register if needed, otherwise clear it.

```

\skip_zero_new:c 20447 \cs_new_protected:Npn \skip_zero_new:N #1
20448 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 20449 \cs_new_protected:Npn \skip_gzero_new:N #1
20450 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
20451 \cs_generate_variant:Nn \skip_zero_new:N { c }
20452 \cs_generate_variant:Nn \skip_gzero_new:N { c }


```

(End definition for `\skip_zero_new:N` and `\skip_gzero_new:N`. These functions are documented on page 216.)

`\skip_if_exist_p:N` Copies of the `cs` functions defined in l3basics.

```

\skip_if_exist_p:c 20453 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
20454 { TF , T , F , p }
\skip_if_exist:cTF 20455 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
20456 { TF , T , F , p }


```

(End definition for `\skip_if_exist:NTF`. This function is documented on page 216.)

## 62.14 Setting skip variables

`\skip_set:Nn`

`\skip_set:cn`

`\skip_gset:Nn`

`\skip_gset:cn`

Much the same as for dimensions.

```
20457 \cs_new_protected:Npn \skip_set:Nn #1#2
20458   { #1 = \tex_glueexpr:D #2 \scan_stop: }
20459   \cs_new_protected:Npn \skip_gset:Nn #1#2
20460   { \tex_global:D #1 = \tex_glueexpr:D #2 \scan_stop: }
20461   \cs_generate_variant:Nn \skip_set:Nn { c }
20462   \cs_generate_variant:Nn \skip_gset:Nn { c }
```

(End definition for `\skip_set:Nn` and `\skip_gset:Nn`. These functions are documented on page 216.)

`\skip_set_eq>NN`

`\skip_set_eq:cN`

`\skip_set_eq:Nc`

`\skip_gset_eq:NN`

`\skip_gset_eq:cN`

`\skip_gset_eq:Nc`

`\skip_gset_eq:cc`

All straightforward.

```
20463 \cs_new_protected:Npn \skip_set_eq:NN #1#2 { #1 = #2 }
20464   \cs_generate_variant:Nn \skip_set_eq:NN { c , Nc , cc }
20465 \cs_new_protected:Npn \skip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
20466 \cs_generate_variant:Nn \skip_gset_eq:NN { c , Nc , cc }
```

(End definition for `\skip_set_eq:NN` and `\skip_gset_eq:NN`. These functions are documented on page 216.)

Using by here deals with the (incorrect) case `\skip123`.

```
20467 \cs_new_protected:Npn \skip_add:Nn #1#2
20468   { \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
20469 \cs_new_protected:Npn \skip_gadd:Nn #1#2
20470   { \tex_global:D \tex_advance:D #1 \tex_glueexpr:D #2 \scan_stop: }
20471 \cs_generate_variant:Nn \skip_add:Nn { c }
20472 \cs_generate_variant:Nn \skip_gadd:Nn { c }
20473 \cs_new_protected:Npn \skip_sub:Nn #1#2
20474   { \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
20475 \cs_new_protected:Npn \skip_gsub:Nn #1#2
20476   { \tex_global:D \tex_advance:D #1 - \tex_glueexpr:D #2 \scan_stop: }
20477 \cs_generate_variant:Nn \skip_sub:Nn { c }
20478 \cs_generate_variant:Nn \skip_gsub:Nn { c }
```

(End definition for `\skip_add:Nn` and others. These functions are documented on page 216.)

## 62.15 Skip expression conditionals

`\skip_if_eq_p:nn`

`\skip_if_eq:nnTF`

Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```
20479 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
20480   {
20481     \str_if_eq:eeTF { \skip_eval:n {#1} } { \skip_eval:n {#2} }
20482     { \prg_return_true: }
20483     { \prg_return_false: }
20484   }
```

(End definition for `\skip_if_eq:nnTF`. This function is documented on page 217.)

```
\skip_if_finite_p:n  
\skip_if_finite:nTF  
\_skip_if_finite:wwNw
```

With  $\varepsilon$ -TeX, we have an easy access to the order of infinites of the stretch and shrink components of a skip. However, to access both, we either need to evaluate the expression twice, or evaluate it, then call an auxiliary to extract both pieces of information from the result. Since we are going to need an auxiliary anyways, it is quicker to make it search for the string `fil` which characterizes infinite glue.

```
20485 \cs_set_protected:Npn \_skip_tmp:w #1  
20486 {  
20487   \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }  
20488   {  
20489     \exp_after:wN \_skip_if_finite:wwNw  
20490     \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:  
20491     #1 ; \prg_return_true: \s__skip_stop  
20492   }  
20493   \cs_new:Npn \_skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \s__skip_stop {##3}  
20494 }  
20495 \exp_args:No \_skip_tmp:w { \tl_to_str:n { fil } }
```

(End definition for `\skip_if_finite:nTF` and `\_skip_if_finite:wwNw`. This function is documented on page 217.)

## 62.16 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```
20496 \cs_new:Npn \skip_eval:n #1  
20497 { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }
```

(End definition for `\skip_eval:n`. This function is documented on page 217.)

`\skip_use:N` Accessing a `<skip>`.

```
20498 \cs_new_eq:NN \skip_use:N \dim_use:N  
20499 \cs_new_eq:NN \skip_use:c \dim_use:c
```

(End definition for `\skip_use:N`. This function is documented on page 217.)

## 62.17 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```
20500 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D  
20501 \cs_new:Npn \skip_horizontal:n #1  
20502 { \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }  
20503 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D  
20504 \cs_new:Npn \skip_vertical:n #1  
20505 { \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }  
20506 \cs_generate_variant:Nn \skip_horizontal:N { c }  
20507 \cs_generate_variant:Nn \skip_vertical:N { c }
```

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 218.)

## 62.18 Viewing skip variables

**\skip\_show:N** Diagnostics.

```
20508 \cs_new_eq:NN \skip_show:N \_kernel_register_show:N  
20509 \cs_generate_variant:Nn \skip_show:N { c }
```

(End definition for `\skip_show:N`. This function is documented on page 217.)

**\skip\_show:c** Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```
20510 \cs_new_protected:Npn \skip_show:n  
20511 { \msg_show_eval:Nn \skip_eval:n }
```

(End definition for `\skip_show:n`. This function is documented on page 217.)

**\skip\_log:N** Diagnostics. Redirect output of `\skip_show:n` to the log.

```
20512 \cs_new_eq:NN \skip_log:N \_kernel_register_log:N  
20513 \cs_new_eq:NN \skip_log:c \_kernel_register_log:c  
20514 \cs_new_protected:Npn \skip_log:n  
20515 { \msg_log_eval:Nn \skip_eval:n }
```

(End definition for `\skip_log:N` and `\skip_log:n`. These functions are documented on page 218.)

## 62.19 Constant skips

**\c\_zero\_skip** Skips with no rubber component are just dimensions but need to terminate correctly.

```
20516 \skip_const:Nn \c_zero_skip { \c_zero_dim }  
20517 \skip_const:Nn \c_max_skip { \c_max_dim }
```

(End definition for `\c_zero_skip` and `\c_max_skip`. These functions are documented on page 218.)

## 62.20 Scratch skips

**\l\_tmpa\_skip** We provide two local and two global scratch registers, maybe we need more or less.

```
20518 \skip_new:N \l_tmpa_skip  
20519 \skip_new:N \l_tmpb_skip  
20520 \skip_new:N \g_tmpa_skip  
20521 \skip_new:N \g_tmpb_skip
```

(End definition for `\l_tmpa_skip` and others. These variables are documented on page 218.)

## 62.21 Creating and initialising `muskip` variables

**\muskip\_new:N** And then we add muskips.

```
20522 \cs_new_protected:Npn \muskip_new:N #1  
20523 {  
20524 \_kernel_chk_if_free_cs:N #1  
20525 \cs:w newmuskip \cs_end: #1  
20526 }  
20527 \cs_generate_variant:Nn \muskip_new:N { c }
```

(End definition for `\muskip_new:N`. This function is documented on page 219.)

\muskip\_const:Nn See \skip\_const:Nn.

```

20528 \cs_new_protected:Npn \muskip_const:Nn #1#2
20529   {
20530     \muskip_new:N #1
20531     \tex_global:D #1 = \muskip_eval:n {#2} \scan_stop:
20532   }
20533 \cs_generate_variant:Nn \muskip_const:Nn { c }

```

(End definition for \muskip\_const:Nn. This function is documented on page 219.)

\muskip\_zero:N Reset the register to zero.

```

20534 \cs_new_protected:Npn \muskip_zero:N #1
20535   { #1 = \c_zero_muskip }

```

\muskip\_gzero:N

\muskip\_gzero:c

```

20536 \cs_new_protected:Npn \muskip_gzero:N #1
20537   { \tex_global:D #1 = \c_zero_muskip }
20538 \cs_generate_variant:Nn \muskip_zero:N { c }
20539 \cs_generate_variant:Nn \muskip_gzero:N { c }

```

(End definition for \muskip\_zero:N and \muskip\_gzero:N. These functions are documented on page 219.)

\muskip\_zero\_new:N Create a register if needed, otherwise clear it.

```

20540 \cs_new_protected:Npn \muskip_zero_new:N #1
20541   { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
20542 \cs_new_protected:Npn \muskip_gzero_new:N #1
20543   { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
20544 \cs_generate_variant:Nn \muskip_zero_new:N { c }
20545 \cs_generate_variant:Nn \muskip_gzero_new:N { c }

```

(End definition for \muskip\_zero\_new:N and \muskip\_gzero\_new:N. These functions are documented on page 219.)

\muskip\_if\_exist\_p:N Copies of the cs functions defined in l3basics.

```

20546 \prg_new_eq_conditional:NNN \muskip_if_exist:N \cs_if_exist:N
20547   { TF , T , F , p }
20548 \prg_new_eq_conditional:NNN \muskip_if_exist:c \cs_if_exist:c
20549   { TF , T , F , p }

```

(End definition for \muskip\_if\_exist:NTF. This function is documented on page 219.)

## 62.22 Setting muskip variables

\muskip\_set:Nn

\muskip\_set:cn

\muskip\_gset:Nn

\muskip\_gset:cn

```

20550 \cs_new_protected:Npn \muskip_set:Nn #1#2
20551   { #1 = \tex_muexpr:D #2 \scan_stop: }
20552 \cs_new_protected:Npn \muskip_gset:Nn #1#2
20553   { \tex_global:D #1 = \tex_muexpr:D #2 \scan_stop: }
20554 \cs_generate_variant:Nn \muskip_set:Nn { c }
20555 \cs_generate_variant:Nn \muskip_gset:Nn { c }

```

(End definition for \muskip\_set:Nn and \muskip\_gset:Nn. These functions are documented on page 220.)

`\muskip_set_eq:NN`

All straightforward.

```
20556 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
20557 \cs_generate_variant:Nn \muskip_set_eq:NN { c , cc }
20558 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
20559 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , cc }
```

`\muskip_gset_eq:NN`

`\muskip_gset_eq:cN`

`\muskip_gset_eq:Nc`

`\muskip_gset_eq:cc`

`\muskip_add:Nn`

`\muskip_add:cn`

`\muskip_gadd:Nn`

`\muskip_gadd:cn`

`\muskip_sub:Nn`

`\muskip_sub:cn`

`\muskip_gsub:Nn`

`\muskip_gsub:cn`

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 220.)

Using by here deals with the (incorrect) case `\muskip123`.

```
20560 \cs_new_protected:Npn \muskip_add:Nn #1#2
20561   { \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
20562 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
20563   { \tex_global:D \tex_advance:D #1 \tex_muexpr:D #2 \scan_stop: }
20564 \cs_generate_variant:Nn \muskip_add:Nn { c }
20565 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
20566 \cs_new_protected:Npn \muskip_sub:Nn #1#2
20567   { \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
20568 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
20569   { \tex_global:D \tex_advance:D #1 - \tex_muexpr:D #2 \scan_stop: }
20570 \cs_generate_variant:Nn \muskip_sub:Nn { c }
20571 \cs_generate_variant:Nn \muskip_gsub:Nn { c }
```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 219.)

## 62.23 Using muskip expressions and variables

`\muskip_eval:n`

Evaluating a muskip expression expandably.

```
20572 \cs_new:Npn \muskip_eval:n #1
20573   { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }
```

(End definition for `\muskip_eval:n`. This function is documented on page 220.)

`\muskip_use:N`

Accessing a *<muskip>*.

`\muskip_use:c`

```
20574 \cs_new_eq:NN \muskip_use:N \dim_use:N
```

```
20575 \cs_new_eq:NN \muskip_use:c \dim_use:c
```

(End definition for `\muskip_use:N`. This function is documented on page 220.)

## 62.24 Viewing muskip variables

`\muskip_show:N`

Diagnostics.

`\muskip_show:c`

```
20576 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
20577 \cs_generate_variant:Nn \muskip_show:N { c }
```

(End definition for `\muskip_show:N`. This function is documented on page 220.)

`\muskip_show:n`

Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.

```
20578 \cs_new_protected:Npn \muskip_show:n
20579   { \msg_show_eval:Nn \muskip_eval:n }
```

(End definition for `\muskip_show:n`. This function is documented on page 221.)

```

\muskip_log:N  Diagnostics. Redirect output of \muskip_show:n to the log.
\muskip_log:c  20580 \cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
\muskip_log:n  20581 \cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
                \cs_new_protected:Npn \muskip_log:n
                { \msg_log_eval:Nn \muskip_eval:n }

```

(End definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 221.)

## 62.25 Constant muskips

`\c_zero_muskip` Constant muskips given by their value.

```

\c_max_muskip  20584 \muskip_const:Nn \c_zero_muskip { 0 mu }
                20585 \muskip_const:Nn \c_max_muskip { 16383.99999 mu }

```

(End definition for `\c_zero_muskip` and `\c_max_muskip`. These functions are documented on page 221.)

## 62.26 Scratch muskips

`\l_tmpa_muskip` We provide two local and two global scratch registers, maybe we need more or less.

```

\l_tmpb_muskip  20586 \muskip_new:N \l_tmpa_muskip
\g_tmpa_muskip  20587 \muskip_new:N \l_tmpb_muskip
\g_tmpb_muskip  20588 \muskip_new:N \g_tmpa_muskip
                20589 \muskip_new:N \g_tmpb_muskip

```

(End definition for `\l_tmpa_muskip` and others. These variables are documented on page 221.)

20590 ⟨/package⟩

# Chapter 63

## I3keys Implementation

20591 `(*package)`

### 63.1 Low-level interface

The low-level key parser's implementation is based heavily on `expkv`. Compared to `keyval` it adds a number of additional "safety" requirements and allows to process the parsed list of key-value pairs in a variety of ways. The net result is that this code needs around one and a half the amount of time as `keyval` to parse the same list of keys. To optimise speed as far as reasonably practical, a number of lower-level approaches are taken rather than using the higher-level `expl3` interfaces.

20592 `(@@=keyval)`

```
\s__keyval_nil
\s__keyval_mark
\s__keyval_stop
\s__keyval_tail
20593 \scan_new:N \s__keyval_nil
20594 \scan_new:N \s__keyval_mark
20595 \scan_new:N \s__keyval_stop
20596 \scan_new:N \s__keyval_tail
```

*(End definition for `\s__keyval_nil` and others.)*

`\l_kernel_keyval_allow_blank_keys_bool`

The general behavior of the `I3keys` module is to throw an error on blank key names. However to support the usage of `\keyval_parse:nnn` in the `I3prop` module we allow this error to be switched off temporarily and just ignore blank names.

20597 `\bool_new:N \l_kernel_keyval_allow_blank_keys_bool`

*(End definition for `\l_kernel_keyval_allow_blank_keys_bool`.)*

This temporary macro will be used since some of the definitions will need an active comma or equals sign. Inside of this macro #1 will be the active comma and #2 will be the active equals sign.

```
20598 \group_begin:
20599   \cs_set_protected:Npn \__keyval_tmp:w #1#2
20600   {
```

`\keyval_parse:nnn`  
`\keyval_parse:NNn`

The main function starts the first of two loops. The outer loop splits the key-value list at active commas, the inner loop will do so at other commas. The use of `\s__keyval_mark` here prevents loss of braces from the key argument.

```

20601 \cs_if_exist:NNTF \tex_expanded:D
20602 {
20603     \cs_new:Npn \keyval_parse:n {##1 ##2 ##3}
20604     {
20605         \__kernel_exp_not:w \tex_expanded:D
20606         {
20607             {
20608                 \__keyval_loop_active:nnw {##1} {##2}
20609                 \s_keyval_mark ##3 #1 \s_keyval_tail #1
20610             }
20611         }
20612     }
20613 }
20614 {
20615     \cs_new:Npn \keyval_parse:n {##1 ##2 ##3}
20616     {
20617         \group_align_safe_begin:
20618         \__keyval_loop_active:nnw {##1} {##2}
20619             \s_keyval_mark ##3 #1 \s_keyval_tail #1
20620             \group_align_safe_end:
20621         }
20622     }
20623 \cs_new_eq:NN \keyval_parse:NNn \keyval_parse:nnn

```

(End definition for `\keyval_parse:nnn` and `\keyval_parse:NNn`. These functions are documented on page 235.)

`\__keyval_loop_active:nnw` First a fast test for the end of the loop is done, it'll gobble everything up to a `\s_keyval_tail`. The loop ending macro will gobble everything to the last comma in this definition. If the end isn't reached yet, start the second loop splitting at other commas, the next iteration of this first loop will be inserted by the end of `\__keyval_loop_other:nnw`.

```

20624     \cs_new:Npn \__keyval_loop_active:nnw {##1 ##2 ##3 #1
20625     {
20626         \__keyval_if_recursion_tail:w ##3
20627             \__keyval_end_loop_active:w \s_keyval_tail
20628             \__keyval_loop_other:nnw {##1} {##2} ##3 , \s_keyval_tail ,
20629     }

```

(End definition for `\__keyval_loop_active:nnw`.)

`\__keyval_split_other:w` These two macros allow to split at the first equals sign of category 12 or 13. At the same time they also execute branching by inserting the first token following `\s_keyval_mark` that followed the equals sign. Hence they also test for the presence of such an equals sign simultaneously.

```

20630     \cs_new:Npn \__keyval_split_other:w {##1 = ##2 \s_keyval_mark ##3
20631         { ##3 ##1 \s_keyval_stop \s_keyval_mark ##2 }
20632     \cs_new:Npn \__keyval_split_active:w {##1 #2 ##2 \s_keyval_mark ##3
20633         { ##3 ##1 \s_keyval_stop \s_keyval_mark ##2 }

```

(End definition for `\__keyval_split_other:w` and `\__keyval_split_active:w`.)

`\__keyval_loop_other:nnw` The second loop uses the same test for its end as the first loop, next it splits at the first active equals sign using `\__keyval_split_active:w`. The `\s_keyval_nil` prevents

accidental brace stripping and acts as a delimiter in the next steps. First testing for an active equals sign will reduce the number of necessary expansion steps for the expected average use case of other equals signs and hence perform better on average.

```

20634      \cs_new:Npn \__keyval_loop_other:nnw ##1 ##2 ##3 ,
20635      {
20636          \__keyval_if_recursion_tail:w ##3
20637              \__keyval_end_loop_other:w \s__keyval_tail
20638          \__keyval_split_active:w ##3 \s__keyval_nil
20639              \s__keyval_mark \__keyval_split_active_auxi:w
20640                  #2 \s__keyval_mark \__keyval_clean_up_active:w
20641          {##1} {##2}
20642              \s__keyval_mark
20643      }

```

*(End definition for \\_\_keyval\_loop\_other:nnw.)*

\\_\_keyval\_split\_active\_auxi:w  
\\_\_keyval\_split\_active\_auxii:w  
\\_\_keyval\_split\_active\_auxiii:w  
\\_\_keyval\_split\_active\_auxiv:w  
\\_\_keyval\_split\_active\_auxv:w

After \\_\_keyval\_split\_active:w the following will only be called if there was at least one active equals sign in the current key-value pair. Therefore this is the execution branch for a key-value pair with an active equals sign. ##1 will be everything up to the first active equals sign. First it tests for other equals signs in the key name, which will eventually throw an error via \\_\_keyval\_misplaced\_equal\_after\_active\_error:w. If none was found we forward the key to \\_\_keyval\_split\_active\_auxii:w.

```

20644      \cs_new:Npn \__keyval_split_active_auxi:w ##1 \s__keyval_stop
20645      {
20646          \__keyval_split_other:w ##1 \s__keyval_nil
20647              \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
20648                  = \s__keyval_mark \__keyval_split_active_auxii:w
20649      }

```

\\_\_keyval\_split\_active\_auxii:w gets the correct key name with a leading \s\_\_keyval\_mark as ##1. It has to sanitise the remainder of the previous test and trims the key name which will be forwarded to \\_\_keyval\_split\_active\_auxiii:w.

```

20650      \cs_new:Npn \__keyval_split_active_auxii:w
20651          ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_after_active_error:w
20652              \s__keyval_stop \s__keyval_mark
20653                  ##2 \s__keyval_nil #2 \s__keyval_mark \__keyval_clean_up_active:w
20654          { \__keyval_trim:nN {##1} \__keyval_split_active_auxiii:w ##2 \s__keyval_nil }

```

Next we test for a misplaced active equals sign in the value, if none is found \\_\_keyval\_split\_active\_auxiv:w will be called.

```

20655      \cs_new:Npn \__keyval_split_active_auxiii:w ##1 ##2 \s__keyval_nil
20656      {
20657          \__keyval_split_active:w ##2 \s__keyval_nil
20658              \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20659                  #2 \s__keyval_mark \__keyval_split_active_auxiv:w
20660          {##1}
20661      }

```

This runs the last test after sanitising the remainder of the previous one. This time test for a misplaced equals sign of category 12 in the value. Finally the last auxiliary macro will be called.

```

20662      \cs_new:Npn \__keyval_split_active_auxiv:w
20663          ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20664              \s__keyval_stop \s__keyval_mark

```

```

20665    {
20666        \__keyval_split_other:w ##1 \s__keyval_nil
20667            \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20668                = \s__keyval_mark \__keyval_split_active_auxv:w
20669    }

```

This last macro in this execution branch sanitises the last test, trims the value and passes it to `\__keyval_pair:nnnn`.

```

20670    \cs_new:Npn \__keyval_split_active_auxv:w
20671        ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20672            \s__keyval_stop \s__keyval_mark
20673            { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }

```

*(End definition for `\__keyval_split_active_auxi:w` and others.)*

`\__keyval_clean_up_active:w` The following is the branch taken if the key-value pair doesn't contain an active equals sign. The remainder of that test will be cleaned up by `\__keyval_clean_up_active:w` which will then split at an equals sign of category other.

```

20674    \cs_new:Npn \__keyval_clean_up_active:w
20675        ##1 \s__keyval_nil \s__keyval_mark \__keyval_split_active_auxi:w \s__keyval_stop
20676    {
20677        \__keyval_split_other:w ##1 \s__keyval_nil
20678            \s__keyval_mark \__keyval_split_other_auxi:w
20679                = \s__keyval_mark \__keyval_clean_up_other:w
20680    }

```

*(End definition for `\__keyval_clean_up_active:w`.)*

`\__keyval_split_other_auxi:w` `\__keyval_split_other_auxii:w` `\__keyval_split_other_auxiii:w` This is executed if the key-value pair doesn't contain an active equals sign but at least one other. `##1` of `\__keyval_split_other_auxi:w` will contain the complete key name, which is trimmed and forwarded to the next auxiliary macro.

```

20681    \cs_new:Npn \__keyval_split_other_auxi:w ##1 \s__keyval_stop
20682        { \__keyval_trim:nN { ##1 } \__keyval_split_other_auxi:w }

```

We know that the value doesn't contain misplaced active equals signs but we have to test for others. Also we need to sanitise the previous test, which is done here and not earlier to avoid superfluous argument grabbing.

```

20683    \cs_new:Npn \__keyval_split_other_auxii:w
20684        ##1 ##2 \s__keyval_nil = \s__keyval_mark \__keyval_clean_up_other:w
20685    {
20686        \__keyval_split_other:w ##2 \s__keyval_nil
20687            \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20688                = \s__keyval_mark \__keyval_split_other_auxii:w
20689                { ##1 }
20690    }

```

`\__keyval_split_other_auxiii:w` sanitises the test for other equals signs, trims the value and forwards it to `\__keyval_pair:nnnn`.

```

20691    \cs_new:Npn \__keyval_split_other_auxiii:w
20692        ##1 \s__keyval_nil \s__keyval_mark \__keyval_misplaced_equal_in_split_error:w
20693            \s__keyval_stop \s__keyval_mark
20694            { \__keyval_trim:nN { ##1 } \__keyval_pair:nnnn }

```

*(End definition for `\__keyval_split_other_auxi:w`, `\__keyval_split_other_auxii:w`, and `\__keyval_split_other_auxiii:w`.)*

```

\_\_keyval_clean_up_other:w \_\_keyval_clean_up_other:w is the last branch that might exist. It is called if no equals sign was found, hence the only possibilities left are a blank list element, which is to be skipped, or a lonely key. If it's no empty list element this will trim the key name and forward it to \_\_keyval_key:nn.
20695   \cs_new:Npn \_\_keyval_clean_up_other:w
20696     ##1 \s__keyval_nil \s__keyval_mark \_\_keyval_split_other_auxi:w \s__keyval_stop \
20697     {
20698       \_\_keyval_if_blank:w ##1 \s__keyval_nil \s__keyval_stop \_\_keyval_blank_true:w
20699         \s__keyval_mark \s__keyval_stop
20700         \_\_keyval_trim:nN { ##1 } \_\_keyval_key:nn
20701     }

```

(End definition for \\_\\_keyval\_clean\_up\_other:w.)

All these two macros do is gobble the remainder of the current other loop execution and throw an error. Afterwards they have to insert the next loop iteration.

```

20702   \cs_new:Npn \_\_keyval_misplaced_equal_after_active_error:w
20703     \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
20704     = \s__keyval_mark \_\_keyval_split_active_auxii:w
20705     \s__keyval_mark ##3 \s__keyval_nil
20706     #2 \s__keyval_mark \_\_keyval_clean_up_active:w
20707   {
20708     \msg_expandable_error:nn
20709       { keyval } { misplaced-equals-sign }
20710     \_\_keyval_loop_other:nnw
20711   }
20712   \cs_new:Npn \_\_keyval_misplaced_equal_in_split_error:w
20713     \s__keyval_mark ##1 \s__keyval_stop \s__keyval_mark ##2 \s__keyval_nil
20714     ##3 \s__keyval_mark ##4 ##5
20715   {
20716     \msg_expandable_error:nn
20717       { keyval } { misplaced-equals-sign }
20718     \_\_keyval_loop_other:nnw
20719   }

```

(End definition for \\_\\_keyval\_misplaced\_equal\_after\_active\_error:w and \\_\\_keyval\_misplaced\_equal\_in\_split\_error:w.)

All that's left for the parsing loops are the macros which end the recursion. Both just gobble the remaining tokens of the respective loop including the next recursion call. \\_\\_keyval\_end\_loop\_other:w also has to insert the next iteration of the active loop.

```

20720   \cs_new:Npn \_\_keyval_end_loop_other:w
20721     \s__keyval_tail
20722     \_\_keyval_split_active:w
20723     \s__keyval_mark \s__keyval_tail
20724     \s__keyval_nil \s__keyval_mark
20725     \_\_keyval_split_active_auxi:w
20726     #2 \s__keyval_mark \_\_keyval_clean_up_active:w
20727   { \_\_keyval_loop_active:nnw }
20728   \cs_new:Npn \_\_keyval_end_loop_active:w
20729     \s__keyval_tail
20730     \_\_keyval_loop_other:nnw ##1 \s__keyval_mark \s__keyval_tail , \s__keyval_tail ,
20731   { }

```

(End definition for `\_keyval_end_loop_other:w` and `\_keyval_end_loop_active:w`.)

The parsing loops are done, so here ends the definition of `\_keyval_tmp:w`, which will finally set up the macros.

```
20732     }
20733     \char_set_catcode_active:n { '\` }
20734     \char_set_catcode_active:n { '\= }
20735     \_keyval_tmp:w , =
20736 \group_end:
```

`\_keyval_pair:nnnn`  
`\_keyval_key:nn`

These macros will be called on the parsed keys and values of the key–value list. All arguments are completely trimmed. They test for blank key names and call the functions passed to `\keyval_parse:nnn` inside of `\exp_not:n` with the correct arguments. Afterwards they insert the next iteration of the other loop.

```
20737 \group_begin:
20738   \cs_set_protected:Npn \_keyval_tmp:w #1#2
20739   {
20740     \cs_new:Npn \_keyval_pair:nnnn ##1 ##2 ##3 ##4
20741     {
20742       \_keyval_if_blank:w \s__keyval_mark ##2 \s__keyval_nil \s__keyval_stop \_keyval
20743         \s__keyval_mark \s__keyval_stop
20744       #1
20745       \exp_not:n { ##4 {##2} {##1} }
20746       #2
20747       \_keyval_loop_other:nnw {##3} {##4}
20748     }
20749   \cs_new:Npn \_keyval_key:nn ##1 ##2
20750   {
20751     \_keyval_if_blank:w \s__keyval_mark ##1 \s__keyval_nil \s__keyval_stop \_keyval
20752       \s__keyval_mark \s__keyval_stop
20753     #1
20754     \exp_not:n { ##2 {##1} }
20755     #2
20756     \_keyval_loop_other:nnw {##2}
20757   }
20758 }
20759 \cs_if_exist:NTF \tex_expanded:D
20760   { \_keyval_tmp:w { } { } }
20761   { \_keyval_tmp:w \group_align_safe_end: \group_align_safe_begin: }
20762 \group_end:
```

(End definition for `\_keyval_pair:nnnn` and `\_keyval_key:nn`.)

`\_keyval_if_empty:w`  
`\_keyval_if_blank:w`  
`\_keyval_if_recursion_tail:w`

All these tests work by gobbling tokens until a certain combination is met, which makes them pretty fast. The test for a blank argument should be called with an arbitrary token following the argument. Each of these utilize the fact that the argument will contain a leading `\s__keyval_mark`.

```
20763 \cs_new:Npn \_keyval_if_empty:w #1 \s__keyval_mark \s__keyval_stop { }
20764 \cs_new:Npn \_keyval_if_blank:w \s__keyval_mark #1 { \_keyval_if_empty:w \s__keyval_mark
20765 \cs_new:Npn \_keyval_if_recursion_tail:w \s__keyval_mark #1 \s__keyval_tail { }
```

(End definition for `\_keyval_if_empty:w`, `\_keyval_if_blank:w`, and `\_keyval_if_recursion_tail:w`.)

```
\_\_keyval\_blank\_true:w  
\_\_keyval\_blank\_key\_error:w
```

These macros will be called if the tests above didn't gobble them, they execute the branching.

```
20766 \cs_new:Npn \_\_keyval\_blank\_true:w \s__keyval_mark \s__keyval_stop \_\_keyval_trim:nN #1 \  
20767 { \_\_keyval\_loop\_other:nnw }  
20768 \cs_new:Npn \_\_keyval\_blank\_key\_error:w \s__keyval_mark \s__keyval_stop #1 \_\_keyval\_loop_o  
20769 {  
20770     \bool_if:NTF \l__kernel_keyval_allow_blank_keys_bool  
20771     { #1 }  
20772     { \msg_expandable_error:nn { keyval } { blank-key-name } }  
20773     \_\_keyval\_loop\_other:nnw  
20774 }
```

(End definition for \\_\\_keyval\\_blank\\_true:w and \\_\\_keyval\\_blank\\_key\\_error:w.)

Two messages for the low level parsing system.

```
20775 \msg_new:nnn { keyval } { misplaced-equals-sign }  
20776 { Misplaced~'='~in~key-value~input~\msg_line_context: }  
20777 \msg_new:nnn { keyval } { blank-key-name }  
20778 { Blank~key~name~in~key~value~input~\msg_line_context: }  
20779 \prop_gput:Nnn \g_msg_module_name_prop { keyval } { LaTeX }  
20780 \prop_gput:Nnn \g_msg_module_type_prop { keyval } { }
```

```
\_\_keyval\_trim:nN  
\_\_keyval\_trim\_auxi:w  
\_\_keyval\_trim\_auxii:w  
\_\_keyval\_trim\_auxiii:w  
\_\_keyval\_trim\_auxiv:w
```

And an adapted version of \\_\\_tl\\_trim\\_spaces:nn which is a bit faster for our use case, as it can strip the braces at the end. This is pretty much the same concept, so I won't comment on it here. The speed gain by using this instead of \tl\_trim\_spaces\_apply:nN is about 10 % of the total time for \keyval\_parse:NNn with one key and one key-value pair, so I think it's worth it.

```
20781 \group_begin:  
20782     \cs_set_protected:Npn \_\_keyval\_tmp:w #1  
20783     {  
20784         \cs_new:Npn \_\_keyval\_trim:nN ##1  
20785         {  
20786             \_\_keyval\_trim\_auxi:w  
20787             ##1  
20788             \s__keyval_nil  
20789             \s__keyval_mark #1 { }  
20790             \s__keyval_mark \_\_keyval\_trim\_auxii:w  
20791             \_\_keyval\_trim\_auxiii:w  
20792             #1 \s__keyval_nil  
20793             \_\_keyval\_trim\_auxiv:w  
20794         }  
20795         \cs_new:Npn \_\_keyval\_trim\_auxi:w ##1 \s__keyval_mark #1 ##2 \s__keyval_mark ##3  
20796         {  
20797             ##3  
20798             \_\_keyval\_trim\_auxi:w  
20799             \s__keyval_mark  
20800             ##2  
20801             \s__keyval_mark #1 {##1}  
20802         }  
20803         \cs_new:Npn \_\_keyval\_trim\_auxii:w \_\_keyval\_trim\_auxi:w \s__keyval_mark \s__keyval_m  
20804         {  
20805             \_\_keyval\_trim\_auxiii:w  
20806             ##1  
20807         }
```

```

20808     \cs_new:Npn \__keyval_trim_auxiii:w ##1 #1 \s__keyval_nil ##2
20809     {
20810         ##2
20811         ##1 \s__keyval_nil
20812         \__keyval_trim_auxiii:w
20813     }

```

This is the one macro which differs from the original definition.

```

20814     \cs_new:Npn \__keyval_trim_auxiv:w
20815         \s__keyval_mark ##1 \s__keyval_nil
20816         \__keyval_trim_auxiii:w \s__keyval_nil \__keyval_trim_auxiii:w
20817         ##2
20818         { ##2 { ##1 } }
20819     }
20820     \__keyval_tmp:w { ~ }
20821 \group_end:

```

(End definition for `\__keyval_trim:nN` and others.)

## 63.2 Constants and variables

```
20822 <@=keys>
```

Various storage areas for the different data which make up keys.

```

20823 \str_const:Nn \c__keys_code_root_str { key-code~>~ }
20824 \str_const:Nn \c__keys_check_root_str { key-check~>~ }
20825 \str_const:Nn \c__keys_default_root_str { key-default~>~ }
20826 \str_const:Nn \c__keys_groups_root_str { key-groups~>~ }
20827 \str_const:Nn \c__keys_inherit_root_str { key-inherit~>~ }
20828 \str_const:Nn \c__keys_type_root_str { key-type~>~ }

```

(End definition for `\c__keys_code_root_str` and others.)

`\c__keys_props_root_str`

The prefix for storing properties.

```
20829 \str_const:Nn \c__keys_props_root_str { key-prop~>~ }
```

(End definition for `\c__keys_props_root_str`.)

`\l_keys_choice_int`  
`\l_keys_choice_tl`

Publicly accessible data on which choice is being used when several are generated as a set.

```

20830 \int_new:N \l_keys_choice_int
20831 \tl_new:N \l_keys_choice_tl

```

(End definition for `\l_keys_choice_int` and `\l_keys_choice_tl`. These variables are documented on page 229.)

`\l__keys_groups_clist`

Used for storing and recovering the list of groups which apply to a key: set as a comma list but at one point we have to use this for a token list recovery.

```
20832 \clist_new:N \l__keys_groups_clist
```

(End definition for `\l__keys_groups_clist`.)

`\l_keys_key_str`  
`\l_keys_key_tl`

The name of a key itself: needed when setting keys. The `tl` version is deprecated but has to be handled manually.

```

20833 \str_new:N \l_keys_key_str
20834 \tl_new:N \l_keys_key_tl

```

*(End definition for \l\_keys\_key\_str and \l\_keys\_key\_tl. These variables are documented on page 231.)*

\l\_keys\_module\_str The module for an entire set of keys.

20835 `\str_new:N \l_keys_module_str`

*(End definition for \l\_keys\_module\_str.)*

\l\_keys\_no\_value\_bool A marker is needed internally to show if only a key or a key plus a value was seen: this is recorded here.

20836 `\bool_new:N \l_keys_no_value_bool`

*(End definition for \l\_keys\_no\_value\_bool.)*

\l\_keys\_only\_known\_bool Used to track if only “known” keys are being set.

20837 `\bool_new:N \l_keys_only_known_bool`

*(End definition for \l\_keys\_only\_known\_bool.)*

**\l\_keys\_path\_str** The “path” of the current key is stored here: this is available to the programmer and so is public. The older version is deprecated but has to be handled manually.

20838 `\str_new:N \l_keys_path_str`

20839 `\tl_new:N \l_keys_path_tl`

*(End definition for \l\_keys\_path\_str and \l\_keys\_path\_tl. These variables are documented on page 231.)*

\l\_keys\_inherit\_str

20840 `\str_new:N \l_keys_inherit_str`

*(End definition for \l\_keys\_inherit\_str.)*

\l\_keys\_relative\_tl The relative path for passing keys back to the user. As this can be explicitly no-value, it must be a token list.

20841 `\tl_new:N \l_keys_relative_tl`

20842 `\tl_set:Nn \l_keys_relative_tl { \q_keys_no_value }`

*(End definition for \l\_keys\_relative\_tl.)*

\l\_keys\_property\_str The “property” begin set for a key at definition time is stored here.

20843 `\str_new:N \l_keys_property_str`

*(End definition for \l\_keys\_property\_str.)*

\l\_keys\_selective\_bool \l\_keys\_filtered\_bool Two flags for using key groups: one to indicate that “selective” setting is active, a second to specify which type (“opt-in” or “opt-out”).

20844 `\bool_new:N \l_keys_selective_bool`

20845 `\bool_new:N \l_keys_filtered_bool`

*(End definition for \l\_keys\_selective\_bool and \l\_keys\_filtered\_bool.)*

\l\_keys\_selective\_seq The list of key groups being filtered in or out during selective setting.

20846 `\seq_new:N \l_keys_selective_seq`

*(End definition for \l\_keys\_selective\_seq.)*

\l_keys_unused_clist	Used when setting only some keys to store those left over. 20847 \tl_new:N \l_keys_unused_clist (End definition for \l_keys_unused_clist.)
\l_keys_value_tl	The value given for a key: may be empty if no value was given. 20848 \tl_new:N \l_keys_value_tl (End definition for \l_keys_value_tl. This variable is documented on page 231.)
\l_keys_tmp_bool	Scratch space. 20849 \bool_new:N \l_keys_tmp_bool 20850 \tl_new:N \l_keys_tmpa_tl 20851 \tl_new:N \l_keys_tmpb_tl (End definition for \l_keys_tmp_bool, \l_keys_tmpa_tl, and \l_keys_tmpb_tl.)
\l_keys_compile_bool	For digesting keys. 20852 \bool_new:N \l_keys_compile_bool 20853 \tl_new:N \l_keys_compile_tl (End definition for \l_keys_compile_bool and \l_keys_compile_tl.)
\l_keys_usage_load_prop \l_keys_usage_preamble_prop	Global data for document-level information. 20854 \prop_new:N \l_keys_usage_load_prop 20855 \prop_new:N \l_keys_usage_preamble_prop (End definition for \l_keys_usage_load_prop and \l_keys_usage_preamble_prop. These variables are documented on page 231.)

### 63.2.1 Internal auxiliaries

\s_keys_nil	Internal scan marks. 20856 \scan_new:N \s_keys_nil 20857 \scan_new:N \s_keys_mark 20858 \scan_new:N \s_keys_stop (End definition for \s_keys_nil, \s_keys_mark, and \s_keys_stop.)
\q_keys_no_value	Internal quarks. 20859 \quark_new:N \q_keys_no_value (End definition for \q_keys_no_value.)
\_keys_quark_if_no_value:p:N \_keys_quark_if_no_value:NTF	Branching quark conditional. 20860 \_kernel_quark_new_conditional:Nn \_keys_quark_if_no_value:N { TF } (End definition for \_keys_quark_if_no_value:NTF.)
\_keys_compile:n	An auxiliary to allow cleaner showing of code. 20861 \cs_new_protected:Npn \_keys_compile:n #1 20862 { 20863 \bool_if:NTF \l_keys_compile_bool 20864 { \tl_put_right:Nn \l_keys_compile_tl } 20865 { \use:n } 20866 {#1} 20867 } (End definition for \_keys_compile:n.)

### 63.3 The key defining mechanism

The public function for definitions is just a wrapper for the lower level mechanism, more or less. The outer function is designed to keep a track of the current module, to allow safe nesting. The module is set removing any leading / (which is not needed here).

```

20868 \cs_new_protected:Npn \keys_define:nn
20869   { \__keys_define:onn \l__keys_module_str }
20870 \cs_new_protected:Npn \__keys_define:nnn #1#2#3
20871   {
20872     \str_set:Nx \l__keys_module_str { \__keys_trim_spaces:n {#2} }
20873     \keyval_parse:NNn \__keys_define:n \__keys_define:nn {#3}
20874     \str_set:Nn \l__keys_module_str {#1}
20875   }
20876 \cs_generate_variant:Nn \__keys_define:nnn { o }

(End definition for \keys_define:nn and \__keys_define:nnn. This function is documented on page 223.)
```

\\_\_keys\_define:n  
\\_\_keys\_define:nn  
\\_\_keys\_define\_aux:nn

The outer functions here record whether a value was given and then converge on a common internal mechanism. There is first a search for a property in the current key name, then a check to make sure it is known before the code hands off to the next step.

```

20877 \cs_new_protected:Npn \__keys_define:n #1
20878   {
20879     \bool_set_true:N \l__keys_no_value_bool
20880     \__keys_define_aux:nn {#1} { }
20881   }
20882 \cs_new_protected:Npn \__keys_define:nn #1#2
20883   {
20884     \bool_set_false:N \l__keys_no_value_bool
20885     \__keys_define_aux:nn {#1} {#2}
20886   }
20887 \cs_new_protected:Npn \__keys_define_aux:nn #1#2
20888   {
20889     \__keys_property_find:n {#1}
20890     \cs_if_exist:cTF { \c__keys_props_root_str \l__keys_property_str }
20891       { \__keys_define_code:n {#2} }
20892       {
20893         \str_if_empty:NF \l__keys_property_str
20894           {
20895             \msg_error:nnxx { keys } { property-unknown }
20896             \l__keys_property_str \l__keys_path_str
20897           }
20898       }
20899   }
```

(End definition for \\_\_keys\_define:n, \\_\_keys\_define:nn, and \\_\_keys\_define\_aux:nn.)

\\_\_keys\_property\_find:n  
\\_\_keys\_property\_find\_auxi:w  
\\_\_keys\_property\_find\_auxii:w  
\\_\_keys\_property\_find\_auxiv:w

Searching for a property means finding the last . in the input, and storing the text before and after it. Everything is turned into strings, so there is no problem using an x-type expansion. Since \\_\_keys\_trim\_spaces:n will turn its argument into a string anyway, this function uses \cs\_set\_nopar:Npx instead of \tl\_set:Nx to gain some speed.

```

20900 \cs_new_protected:Npn \__keys_property_find:n #1
20901   {
20902     \cs_set_nopar:Npx \l__keys_property_str { \__keys_trim_spaces:n { #1 } }
```

```

20903     \exp_after:wN \__keys_property_find_auxi:w \l__keys_property_str
20904     \s__keys_nil \__keys_property_find_auxii:w
20905     . \s__keys_nil \__keys_property_find_err:w
20906   }
20907 \cs_new_protected:Npn \__keys_property_find_auxi:w #1 . #2 \s__keys_nil #3
20908   {
20909     #3 #1 \s__keys_mark #2 \s__keys_nil #3
20910   }
20911 \cs_new_protected:Npn \__keys_property_find_auxii:w
20912   #1 \s__keys_mark #2 \s__keys_nil \__keys_property_find_auxii:w . \s__keys_nil
20913   \__keys_property_find_err:w
20914   {
20915     \cs_set_nopar:Npx \l__keys_path_str
20916     { \str_if_empty:NF \l__keys_module_str { \l__keys_module_str / } #1 }
20917     \__keys_property_find_auxi:w #2 \s__keys_nil \__keys_property_find_auxiii:w . \s__keys_
20918     \__keys_property_find_auxiv:w
20919   }
20920 \cs_new_protected:Npn \__keys_property_find_auxiii:w #1 \s__keys_mark
20921   {
20922     \cs_set_nopar:Npx \l__keys_path_str { \l__keys_path_str . #1 }
20923     \__keys_property_find_auxi:w
20924   }
20925 \cs_new_protected:Npn \__keys_property_find_auxiv:w
20926   #1 \s__keys_nil \__keys_property_find_auxiii:w
20927   \s__keys_mark \s__keys_nil \__keys_property_find_auxiv:w
20928   {
20929     \cs_set_nopar:Npx \l__keys_property_str { . #1 }
20930     \cs_set_nopar:Npx \l__keys_path_str
20931     { \exp_after:wN \__keys_trim_spaces:n \exp_after:wN { \l__keys_path_str } }
20932     \tl_set_eq:NN \l__keys_path_tl \l__keys_path_str
20933   }
20934 \cs_new_protected:Npn \__keys_property_find_err:w
20935   #1 \s__keys_nil #2 \__keys_property_find_err:w
20936   {
20937     \str_clear:N \l__keys_property_str
20938     \msg_error:nnn { keys } { no-property } {#1}
20939   }

```

(End definition for `\__keys_property_find:n` and others.)

`\__keys_define_code:n`  
`\__keys_define_code:w`

Two possible cases. If there is a value for the key, then just use the function. If not, then a check to make sure there is no need for a value with the property. If there should be one then complain, otherwise execute it. There is no need to check for a : as if it was missing the earlier tests would have failed.

```

20940 \cs_new_protected:Npn \__keys_define_code:n #1
20941   {
20942     \bool_if:NTF \l__keys_no_value_bool
20943     {
20944       \exp_after:wN \__keys_define_code:w
20945       \l__keys_property_str \s__keys_stop
20946       { \use:c { \c__keys_props_root_str \l__keys_property_str } }
20947     }
20948       \msg_error:nnxx { keys } { property-requires-value }
20949       \l__keys_property_str \l__keys_path_str

```

```

20950         }
20951     }
20952     { \use:c { \c__keys_props_root_str \l__keys_property_str } {#1} }
20953   }
20954 \exp_last_unbraced:NNNNo
20955   \cs_new:Npn \__keys_define_code:w #1 \c_colon_str #2 \s__keys_stop
20956   { \tl_if_empty:nTF {#2} }

(End definition for \__keys_define_code:n and \__keys_define_code:w.)

```

## 63.4 Turning properties into actions

Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

20957 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
20958   { \__keys_bool_set:Nnnn #1 {#2} { true } { false } }
20959 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }
20960 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
20961   { \__keys_bool_set:Nnnn #1 {#2} { false } { true } }
20962 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }
20963 \cs_new_protected:Npn \__keys_bool_set:Nnnn #1#2#3#4
20964   {
20965     \bool_if_exist:NF #1 { \bool_new:N #1 }
20966     \__keys_choice_make:
20967     \__keys_cmd_set:nx { \l_keys_path_str / true }
20968       { \exp_not:c { bool_ #2 set_ #3 :N } \exp_not:N #1 }
20969     \__keys_cmd_set:nx { \l_keys_path_str / false }
20970       { \exp_not:c { bool_ #2 set_ #4 :N } \exp_not:N #1 }
20971     \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
20972       {
20973         \msg_error:nnx { keys } { boolean-values-only }
20974           \l_keys_key_str
20975       }
20976     \__keys_default_set:n { true }
20977   }
20978 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

(End definition for \__keys_bool_set:Nn, \__keys_bool_set_inverse:Nn, and \__keys_bool_set:Nnnn.)

```

To make a choice from a key, two steps: set the code, and set the unknown key. As multichoices and choices are essentially the same bar one function, the code is given together.

```

20979 \cs_new_protected:Npn \__keys_choice_make:
20980   { \__keys_choice_make:N \__keys_choice_find:n }
20981 \cs_new_protected:Npn \__keys_multichoice_make:
20982   { \__keys_choice_make:N \__keys_multichoice_find:n }
20983 \cs_new_protected:Npn \__keys_choice_make:N #1
20984   {
20985     \cs_if_exist:cTF
20986       { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }
20987       {
20988         \str_if_eq:vnTF
20989           { \c__keys_type_root_str \__keys_parent:o \l_keys_path_str }

```

```

20990      { choice }
20991      {
20992          \msg_error:nxxx { keys } { nested-choice-key }
20993              \l_keys_path_tl { \__keys_parent:o \l_keys_path_str }
20994          }
20995          { \__keys_choice_make_aux:N #1 }
20996      }
20997      { \__keys_choice_make_aux:N #1 }
20998  }
20999 \cs_new_protected:Npn \__keys_choice_make:N #1
21000  {
21001      \cs_set_nopar:cpx { \c__keys_type_root_str \l_keys_path_str }
21002          { choice }
21003          \__keys_cmd_set_direct:nn \l_keys_path_str { #1 {##1} }
21004          \__keys_cmd_set_direct:nn { \l_keys_path_str / unknown }
21005          {
21006              \msg_error:nxxx { keys } { choice-unknown }
21007                  \l_keys_path_str {##1}
21008          }
21009      }

```

(End definition for `\__keys_choice_make:` and others.)

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

21010 \cs_new_protected:Npn \__keys_choices_make:nn
21011     { \__keys_choices_make:Nnn \__keys_choice_make: }
21012 \cs_new_protected:Npn \__keys_multichoices_make:nn
21013     { \__keys_choices_make:Nnn \__keys_multichoices_make: }
21014 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
21015  {
21016      #1
21017      \int_zero:N \l_keys_choice_int
21018      \clist_map_inline:nn {#2}
21019      {
21020          \int_incr:N \l_keys_choice_int
21021          \__keys_cmd_set:nx
21022              { \l_keys_path_str / \__keys_trim_spaces:n {##1} }
21023              {
21024                  \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}
21025                  \int_set:Nn \exp_not:N \l_keys_choice_int
21026                      { \int_use:N \l_keys_choice_int }
21027                  \exp_not:n {#3}
21028              }
21029      }
21030  }

```

(End definition for `\__keys_choices_make:nn`, `\__keys_multichoices_make:nn`, and `\__keys_choices_make:Nnn`.)

Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```

21031 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
21032     { \__keys_cmd_set_direct:nn {#1} { \__keys_compile:n {#2} } }

```

```

21033 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
21034 \cs_new_protected:Npn \__keys_cmd_set_direct:nn #1#2
21035   { \cs_set_protected:cpx { \c__keys_code_root_str #1 } ##1 {#2} }

(End definition for \__keys_cmd_set:nn and \__keys_cmd_set_direct:nn.)

```

\\_\_keys\_cs\_set:NNpn  
\\_\_keys\_cs\_set:Ncpn

Creating control sequences is a bit more tricky than other cases as we need to pick up the **p** argument. To make the internals look clearer, the trailing **n** argument here is just for appearance.

```

21036 \cs_new_protected:Npn \__keys_cs_set:NNpn #1#2#3#
21037   {
21038     \cs_set_protected:cpx { \c__keys_code_root_str \l__keys_path_str } ##1
21039     {
21040       \__keys_compile:n
21041         { #1 \exp_not:N #2 \exp_not:n {#3} {##1} }
21042     }
21043     \use_none:n
21044   }
21045 \cs_generate_variant:Nn \__keys_cs_set:NNpn { Nc }

(End definition for \__keys_cs_set:NNpn.)

```

\\_\_keys\_default\_set:n

Setting a default value is easy. These are stored using `\cs_set_nopar:cpx` as this avoids any worries about whether a token list exists.

```

21046 \cs_new_protected:Npn \__keys_default_set:n #1
21047   {
21048     \tl_if_empty:nTF {#1}
21049     {
21050       \cs_set_eq:cN
21051         { \c__keys_default_root_str \l__keys_path_str }
21052       \tex_undefined:D
21053     }
21054   {
21055     \cs_set_nopar:cpx
21056       { \c__keys_default_root_str \l__keys_path_str }
21057       { \exp_not:n {#1} }
21058     \__keys_value_requirement:nn { required } { false }
21059   }
21060 }

```

(End definition for \\_\_keys\_default\_set:n.)

\\_\_keys\_groups\_set:n

Assigning a key to one or more groups uses comma lists. As the list of groups only exists if there is anything to do, the setting is done using a scratch list. For the usual grouping reasons we use the low-level approach to undefining a list. We also use the low-level approach for the other case to avoid tripping up the `check-declarations` code.

```

21061 \cs_new_protected:Npn \__keys_groups_set:n #1
21062   {
21063     \clist_set:Nn \l__keys_groups_clist {#1}
21064     \clist_if_empty:NTF \l__keys_groups_clist
21065     {
21066       \cs_set_eq:cN { \c__keys_groups_root_str \l__keys_path_str }
21067       \tex_undefined:D
21068     }

```

```

21069 {
21070   \cs_set_eq:cN { \c__keys_groups_root_str \l_keys_path_str }
21071     \l__keys_groups_clist
21072   }
21073 }
```

(End definition for `\__keys_groups_set:n`.)

`\__keys_inherit:n` Inheritance means ignoring anything already said about the key: zap the lot and set up.

```

21074 \cs_new_protected:Npn \__keys_inherit:n #1
21075   {
21076     \__keys_undefine:
21077     \cs_set_nopar:cpn { \c__keys_inherit_root_str \l_keys_path_str } {#1}
21078   }
```

(End definition for `\__keys_inherit:n`.)

`\__keys_initialise:n` A set up for initialisation: just run the code if it exists.

```

21079 \cs_new_protected:Npn \__keys_initialise:n #1
21080   {
21081     \cs_if_exist:cTF
21082       { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21083       { \__keys_execute_inherit: }
21084     {
21085       \str_clear:N \l__keys_inherit_str
21086       \cs_if_exist:cT { \c__keys_code_root_str \l_keys_path_str }
21087         { \__keys_execute:nn \l_keys_path_str {#1} }
21088     }
21089   }
```

(End definition for `\__keys_initialise:n`.)

`\__keys_legacy_if_set:nn` Much the same as `expl3` booleans, except we assume that the switch exists.

```

21090 \cs_new_protected:Npn \__keys_legacy_if_set:nn #1#2
21091   { \__keys_legacy_if_set:nnnn {#1} {#2} { true } { false } }
21092 \cs_new_protected:Npn \__keys_legacy_if_set_inverse:nn #1#2
21093   { \__keys_legacy_if_set:nnnn {#1} {#2} { false } { true } }
21094 \cs_new_protected:Npn \__keys_legacy_if_set:nnnn #1#2#3#4
21095   {
21096     \__keys_choice_make:
21097     \__keys_cmd_set:nx { \l_keys_path_str / true }
21098       { \exp_not:c { legacy_if_#2 set_ #3 :n } { \exp_not:n {#1} } }
21099     \__keys_cmd_set:nx { \l_keys_path_str / false }
21100       { \exp_not:c { legacy_if_#2 set_ #4 :n } { \exp_not:n {#1} } }
21101     \__keys_cmd_set:nn { \l_keys_path_str / unknown }
21102     {
21103       \msg_error:nnx { keys } { boolean-values-only }
21104         \l_keys_key_str
21105     }
21106     \__keys_default_set:n { true }
21107     \cs_if_exist:cF { if#1 }
21108     {
21109       \cs:w newif \exp_after:wN \cs_end:
21110         \cs:w if#1 \cs_end:
21111     }
21112 }
```

*(End definition for `\_keys_legacy_if_set:nn`, `\_keys_legacy_if_inverse:nn`, and `\_keys_legacy_if_inverse:nnnn`.)*

`\_keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```
21113 \cs_new_protected:Npn \_keys_meta_make:n #1
21114 {
21115   \exp_args:NVo \_keys_cmd_set_direct:nn \l_keys_path_str
21116   {
21117     \exp_after:wN \keys_set:nn \exp_after:wN
21118     { \l_keys_module_str } {#1}
21119   }
21120 }
21121 \cs_new_protected:Npn \_keys_meta_make:nn #1#2
21122 {
21123   \exp_args:NV \_keys_cmd_set_direct:nn
21124   { \l_keys_path_str { \keys_set:nn {#1} {#2} } }
21125 }
```

*(End definition for `\_keys_meta_make:n` and `\_keys_meta_make:nn`.)*

`\_keys_prop_put:Nn` Much the same as other variables, but needs a dedicated auxiliary.

```
21126 \cs_new_protected:Npn \_keys_prop_put:Nn #1#2
21127 {
21128   \prop_if_exist:NF #1 { \prop_new:N #1 }
21129   \exp_after:wN \_keys_find_key_module:wNN \l_keys_path_str \s_keys_stop
21130   \l_keys_tmpa_tl \l_keys_tmpb_tl
21131   \_keys_cmd_set:nx \l_keys_path_str
21132   {
21133     \exp_not:c { prop_ #2 put:Nnn }
21134     \exp_not:N #1
21135     { \l_keys_tmpb_tl }
21136     \exp_not:n { {##1} }
21137   }
21138 }
21139 \cs_generate_variant:Nn \_keys_prop_put:Nn { c }
```

*(End definition for `\_keys_prop_put:Nn`.)*

`\_keys_undefine:` Undefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```
21140 \cs_new_protected:Npn \_keys_undefine:
21141 {
21142   \clist_map_inline:nn
21143   { code , default , groups , inherit , type , check }
21144   {
21145     \cs_set_eq:cN
21146     { \tl_use:c { c_keys_ ##1 _root_str } \l_keys_path_str }
21147     \tex_undefined:D
21148   }
21149 }
```

*(End definition for `\_keys_undefine:..`)*

\\_\\_keys\\_value\\_requirement:nn  
\\_\\_keys\\_check\\_forbidden:  
\\_\\_keys\\_check\\_required:

Validating key input is done using a second function which runs before the main key code. Setting that up means setting it equal to a generic stub which does the check. This approach makes the lookup very fast at the cost of one additional csname per key that needs it. The cleanup here has to know the structure of the following code.

```

21150 \cs_new_protected:Npn \_\_keys_value_requirement:nn #1#2
21151 {
21152     \str_case:nnF {#2}
21153     {
21154         { true }
21155         {
21156             \cs_set_eq:cc
21157             { \c_\_keys_check_root_str \l_\_keys_path_str }
21158             { \_\_keys_check_ #1 : }
21159         }
21160     { false }
21161     {
21162         \cs_if_eq:ccT
21163             { \c_\_keys_check_root_str \l_\_keys_path_str }
21164             { \_\_keys_check_ #1 : }
21165             {
21166                 \cs_set_eq:cN
21167                     { \c_\_keys_check_root_str \l_\_keys_path_str }
21168                     \tex_undefined:D
21169             }
21170         }
21171     }
21172     {
21173         \msg_error:nnx { keys }
21174             { boolean-values-only }
21175             { .value_ #1 :n }
21176     }
21177 }
21178 \cs_new_protected:Npn \_\_keys_check_forbidden:
21179 {
21180     \bool_if:NF \l_\_keys_no_value_bool
21181     {
21182         \msg_error:nnxx { keys } { value-forbidden }
21183             \l_\_keys_path_str \l_\_keys_value_tl
21184             \use_none:nnn
21185     }
21186 }
21187 \cs_new_protected:Npn \_\_keys_check_required:
21188 {
21189     \bool_if:NT \l_\_keys_no_value_bool
21190     {
21191         \msg_error:nnx { keys } { value-required }
21192             \l_\_keys_path_str
21193             \use_none:nnn
21194     }
21195 }
```

(End definition for \\_\\_keys\\_value\\_requirement:nn, \\_\\_keys\\_check\\_forbidden:, and \\_\\_keys\\_check\\_required:.)

```

\__keys_usage:n Save the relevant data.
\__keys_usage:NN
\__keys_usage:w
21196 \cs_new_protected:Npn \__keys_usage:n #1
21197  {
21198      \str_case:nnF {#1}
21199      {
21200          { general }
21201          {
21202              \__keys_usage:NN \l__keys_usage_load_prop
21203                  \c_false_bool
21204              \__keys_usage:NN \l__keys_usage_preamble_prop
21205                  \c_false_bool
21206          }
21207          { load }
21208          {
21209              \__keys_usage:NN \l__keys_usage_load_prop
21210                  \c_true_bool
21211              \__keys_usage:NN \l__keys_usage_preamble_prop
21212                  \c_false_bool
21213          }
21214          { preamble }
21215          {
21216              \__keys_usage:NN \l__keys_usage_load_prop
21217                  \c_false_bool
21218              \__keys_usage:NN \l__keys_usage_preamble_prop
21219                  \c_true_bool
21220          }
21221      }
21222      {
21223          \msg_error:nnnn { keys }
21224          { choice-unknown }
21225          { .usage:n }
21226          {#1}
21227      }
21228  }
21229 \cs_new_protected:Npn \__keys_usage:NN #1#2
21230  {
21231      \prop_get:NVNF #1 \l__keys_module_str \l__keys_tmpa_tl
21232          { \tl_clear:N \l__keys_tmpa_tl }
21233      \tl_set:Nx \l__keys_tmpb_tl
21234          { \exp_after:wN \__keys_usage:w \l__keys_path_str \s__keys_stop }
21235      \bool_if:NTF #2
21236          { \clist_put_right:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
21237          { \clist_remove_all:NV \l__keys_tmpa_tl \l__keys_tmpb_tl }
21238      \prop_put:NVV #1 \l__keys_module_str
21239          \l__keys_tmpa_tl
21240  }
21241 \cs_new:Npn \__keys_usage:w #1 / #2 \s__keys_stop {#2}

```

(End definition for `\__keys_usage:n`, `\__keys_usage:NN`, and `\__keys_usage:w`.)

`\__keys_variable_set:NnnN`  
`\__keys_variable_set:cnnN`

```

\__keys_variable_set_required:NnnN
\__keys_variable_set_required:cnnN
21242 \cs_new_protected:Npn \__keys_variable_set:NnnN #1#2#3#4
21243  {

```

```

21244     \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
21245     \__keys_cmd_set:nx \l_keys_path_str
21246     {
21247         \exp_not:c { #2 _ #3 set:N #4 }
21248         \exp_not:N #1
21249         \exp_not:n { {##1} }
21250     }
21251 }
21252 \cs_generate_variant:Nn \__keys_variable_set:NnnN { c }
21253 \cs_new_protected:Npn \__keys_variable_set_required:NnnN #1#2#3#4
21254 {
21255     \__keys_variable_set:NnnN #1 {#2} {#3} #4
21256     \__keys_value_requirement:nn { required } { true }
21257 }
21258 \cs_generate_variant:Nn \__keys_variable_set_required:NnnN { c }

```

(End definition for `\__keys_variable_set:NnnN` and `\__keys_variable_set_required:NnnN`.)

## 63.5 Creating key properties

The key property functions are all wrappers for internal functions, meaning that things stay readable and can also be altered later on.

Importantly, while key properties have “normal” argument specs, the underlying code always supplies one braced argument to these. As such, argument expansion is handled by hand rather than using the standard tools. This shows up particularly for the two-argument properties, where things would otherwise go badly wrong.

**.bool\_set:N** One function for this.

```

21259 \cs_new_protected:cpx { \c__keys_props_root_str .bool_set:N } #1
21260 { \__keys_bool_set:Nn #1 { } }
21261 \cs_new_protected:cpx { \c__keys_props_root_str .bool_set:c } #1
21262 { \__keys_bool_set:cn {#1} { } }
21263 \cs_new_protected:cpx { \c__keys_props_root_str .bool_gset:N } #1
21264 { \__keys_bool_set:Nn #1 { g } }
21265 \cs_new_protected:cpx { \c__keys_props_root_str .bool_gset:c } #1
21266 { \__keys_bool_set:cn {#1} { g } }

```

(End definition for `.bool_set:N` and `.bool_gset:N`. These functions are documented on page [224](#).)

**.bool\_set\_inverse:N** One function for this.

```

21267 \cs_new_protected:cpx { \c__keys_props_root_str .bool_set_inverse:N } #1
21268 { \__keys_bool_set_inverse:Nn #1 { } }
21269 \cs_new_protected:cpx { \c__keys_props_root_str .bool_set_inverse:c } #1
21270 { \__keys_bool_set_inverse:cn {#1} { } }
21271 \cs_new_protected:cpx { \c__keys_props_root_str .bool_gset_inverse:N } #1
21272 { \__keys_bool_set_inverse:Nn #1 { g } }
21273 \cs_new_protected:cpx { \c__keys_props_root_str .bool_gset_inverse:c } #1
21274 { \__keys_bool_set_inverse:cn {#1} { g } }

```

(End definition for `.bool_set_inverse:N` and `.bool_gset_inverse:N`. These functions are documented on page [224](#).)

**.choice:** Making a choice is handled internally, as it is also needed by .generate\_choices:n.

```
21275 \cs_new_protected:cpn { \c_keys_props_root_str .choice: }  
21276   { \_keys_choice_make: }
```

(End definition for .choice:. This function is documented on page 224.)

**.choices:nn** For auto-generation of a series of mutually-exclusive choices. Here, #1 consists of two separate arguments, hence the slightly odd-looking implementation.

```
21277 \cs_new_protected:cpn { \c_keys_props_root_str .choices:nn } #1  
21278   { \_keys_choices_make:nn #1 }  
21279 \cs_new_protected:cpn { \c_keys_props_root_str .choices:Vn } #1  
21280   { \exp_args:NV \_keys_choices_make:nn #1 }  
21281 \cs_new_protected:cpn { \c_keys_props_root_str .choices:on } #1  
21282   { \exp_args:No \_keys_choices_make:nn #1 }  
21283 \cs_new_protected:cpn { \c_keys_props_root_str .choices:xn } #1  
21284   { \exp_args:Nx \_keys_choices_make:nn #1 }
```

(End definition for .choices:nn. This function is documented on page 224.)

**.code:n** Creating code is simply a case of passing through to the underlying set function.

```
21285 \cs_new_protected:cpn { \c_keys_props_root_str .code:n } #1  
21286   { \_keys_cmd_set:nn \l_keys_path_str {#1} }
```

(End definition for .code:n. This function is documented on page 225.)

```
.clist_set:N  
.clist_set:c 21287 \cs_new_protected:cpn { \c_keys_props_root_str .clist_set:N } #1  
.clist_gset:N 21288   { \_keys_variable_set:NnnN #1 { clist } { } n }  
.clist_gset:c 21289 \cs_new_protected:cpn { \c_keys_props_root_str .clist_set:c } #1  
21290   { \_keys_variable_set:cnnN {#1} { clist } { } n }  
21291 \cs_new_protected:cpn { \c_keys_props_root_str .clist_gset:N } #1  
21292   { \_keys_variable_set:NnnN #1 { clist } { g } n }  
21293 \cs_new_protected:cpn { \c_keys_props_root_str .clist_gset:c } #1  
21294   { \_keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for .clist\_set:N and .clist\_gset:N. These functions are documented on page 224.)

**.cs\_set:Np**

**.cs\_set:cp**

**.cs\_set\_protected:Np**

**.cs\_set\_protected:cp**

**.cs\_gset:Np**

**.cs\_gset:cp**

**.cs\_gset\_protected:Np**

**.cs\_gset\_protected:cp**

```
21295 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set:Np } #1  
21296   { \_keys_cs_set>NNpn \cs_set:Npn #1 { } }  
21297 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set:cp } #1  
21298   { \_keys_cs_set:Ncpn \cs_set:Npn #1 { } }  
21299 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set_protected:Np } #1  
21300   { \_keys_cs_set>NNpn \cs_set_protected:Npn #1 { } }  
21301 \cs_new_protected:cpn { \c_keys_props_root_str .cs_set_protected:cp } #1  
21302   { \_keys_cs_set:Ncpn \cs_set_protected:Npn #1 { } }  
21303 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset:Np } #1  
21304   { \_keys_cs_set>NNpn \cs_gset:Npn #1 { } }  
21305 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset:cp } #1  
21306   { \_keys_cs_set:Ncpn \cs_gset:Npn #1 { } }  
21307 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset_protected:Np } #1  
21308   { \_keys_cs_set>NNpn \cs_gset_protected:Npn #1 { } }  
21309 \cs_new_protected:cpn { \c_keys_props_root_str .cs_gset_protected:cp } #1  
21310   { \_keys_cs_set:Ncpn \cs_gset_protected:Npn #1 { } }
```

(End definition for `.cs_set:Np` and others. These functions are documented on page 225.)

**.default:n** Expansion is left to the internal functions.

```
21311 \cs_new_protected:cpn { \c_keys_props_root_str .default:n } #1
21312   { \_keys_default_set:n {#1} }
21313 \cs_new_protected:cpn { \c_keys_props_root_str .default:V } #1
21314   { \exp_args:NV \_keys_default_set:n #1 }
21315 \cs_new_protected:cpn { \c_keys_props_root_str .default:o } #1
21316   { \exp_args:No \_keys_default_set:n {#1} }
21317 \cs_new_protected:cpn { \c_keys_props_root_str .default:x } #1
21318   { \exp_args:Nx \_keys_default_set:n {#1} }
```

(End definition for `.default:n`. This function is documented on page 225.)

**.dim\_set:N** Setting a variable is very easy: just pass the data along.

```
21319 \cs_new_protected:cpn { \c_keys_props_root_str .dim_set:N } #1
21320   { \_keys_variable_set_required:NnnN #1 { dim } { } n }
21321 \cs_new_protected:cpn { \c_keys_props_root_str .dim_set:c } #1
21322   { \_keys_variable_set_required:cnnN {#1} { dim } { } n }
21323 \cs_new_protected:cpn { \c_keys_props_root_str .dim_gset:N } #1
21324   { \_keys_variable_set_required:NnnN #1 { dim } { g } n }
21325 \cs_new_protected:cpn { \c_keys_props_root_str .dim_gset:c } #1
21326   { \_keys_variable_set_required:cnnN {#1} { dim } { g } n }
```

(End definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 225.)

**.fp\_set:N** Setting a variable is very easy: just pass the data along.

```
21327 \cs_new_protected:cpn { \c_keys_props_root_str .fp_set:N } #1
21328   { \_keys_variable_set_required:NnnN #1 { fp } { } n }
21329 \cs_new_protected:cpn { \c_keys_props_root_str .fp_set:c } #1
21330   { \_keys_variable_set_required:cnnN {#1} { fp } { } n }
21331 \cs_new_protected:cpn { \c_keys_props_root_str .fp_gset:N } #1
21332   { \_keys_variable_set_required:NnnN #1 { fp } { g } n }
21333 \cs_new_protected:cpn { \c_keys_props_root_str .fp_gset:c } #1
21334   { \_keys_variable_set_required:cnnN {#1} { fp } { g } n }
```

(End definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 225.)

**.groups:n** A single property to create groups of keys.

```
21335 \cs_new_protected:cpn { \c_keys_props_root_str .groups:n } #1
21336   { \_keys_groups_set:n {#1} }
```

(End definition for `.groups:n`. This function is documented on page 225.)

**.inherit:n** Nothing complex: only one variant at the moment!

```
21337 \cs_new_protected:cpn { \c_keys_props_root_str .inherit:n } #1
21338   { \_keys_inherit:n {#1} }
```

(End definition for `.inherit:n`. This function is documented on page 226.)

**.initial:n** The standard hand-off approach.

```
21339 \cs_new_protected:cpn { \c_keys_props_root_str .initial:n } #1
21340   { \_keys_initialise:n {#1} }
21341 \cs_new_protected:cpn { \c_keys_props_root_str .initial:V } #1
21342   { \exp_args:NV \_keys_initialise:n #1 }
21343 \cs_new_protected:cpn { \c_keys_props_root_str .initial:o } #1
21344   { \exp_args:No \_keys_initialise:n {#1} }
21345 \cs_new_protected:cpn { \c_keys_props_root_str .initial:x } #1
21346   { \exp_args:Nx \_keys_initialise:n {#1} }
```

(End definition for `.initial:n`. This function is documented on page [226](#).)

**.int\_set:N** Setting a variable is very easy: just pass the data along.

```
21347 \cs_new_protected:cpn { \c_keys_props_root_str .int_set:N } #1
21348   { \_keys_variable_set_required:NnnN #1 { int } { } n }
21349 \cs_new_protected:cpn { \c_keys_props_root_str .int_set:c } #1
21350   { \_keys_variable_set_required:cnnN {#1} { int } { } n }
21351 \cs_new_protected:cpn { \c_keys_props_root_str .int_gset:N } #1
21352   { \_keys_variable_set_required:NnnN #1 { int } { g } n }
21353 \cs_new_protected:cpn { \c_keys_props_root_str .int_gset:c } #1
21354   { \_keys_variable_set_required:cnnN {#1} { int } { g } n }
```

(End definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page [226](#).)

**.legacy\_if\_set:n**

**.legacy\_if\_gset:n**

**.legacy\_if\_set\_inverse:n**

**.legacy\_if\_gset\_inverse:n**

```
21355 \cs_new_protected:cpn { \c_keys_props_root_str .legacy_if_set:n } #1
21356   { \_keys_legacy_if_set:nn {#1} { } }
21357 \cs_new_protected:cpn { \c_keys_props_root_str .legacy_if_gset:n } #1
21358   { \_keys_legacy_if_set:nn {#1} { g } }
21359 \cs_new_protected:cpn { \c_keys_props_root_str .legacy_if_set_inverse:n } #1
21360   { \_keys_legacy_if_set_inverse:nn {#1} { } }
21361 \cs_new_protected:cpn { \c_keys_props_root_str .legacy_if_gset_inverse:n } #1
21362   { \_keys_legacy_if_set_inverse:nn {#1} { g } }
```

(End definition for `.legacy_if_set:n` and others. These functions are documented on page [226](#).)

**.meta:n** Making a meta is handled internally.

```
21363 \cs_new_protected:cpn { \c_keys_props_root_str .meta:n } #1
21364   { \_keys_meta_make:n {#1} }
```

(End definition for `.meta:n`. This function is documented on page [226](#).)

**.meta:nn** Meta with path: potentially lots of variants, but for the moment no so many defined.

```
21365 \cs_new_protected:cpn { \c_keys_props_root_str .meta:nn } #1
21366   { \_keys_meta_make:nn #1 }
```

(End definition for `.meta:nn`. This function is documented on page [226](#).)

**.multichoice:** The same idea as `.choice:` and `.choices:nn`, but where more than one choice is allowed.

**.multichoices:nn**

**.multichoices:Vn**

**.multichoices:on**

**.multichoices:xn**

```
21367 \cs_new_protected:cpn { \c_keys_props_root_str .multichoice: }
21368   { \_keys_multichoice_make: }
21369 \cs_new_protected:cpn { \c_keys_props_root_str .multichoices:nn } #1
21370   { \_keys_multichoices_make:nn #1 }
21371 \cs_new_protected:cpn { \c_keys_props_root_str .multichoices:Vn } #1
21372   { \exp_args:NV \_keys_multichoices_make:nn #1 }
```

```

21373 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:on } #1
21374   { \exp_args:No \__keys_multichoices_make:nn #1 }
21375 \cs_new_protected:cpn { \c__keys_props_root_str .multichoices:xn } #1
21376   { \exp_args:Nx \__keys_multichoices_make:nn #1 }

```

(End definition for `.multichoice:` and `.multichoices:nn`. These functions are documented on page [227](#).)

**.muskip\_set:N** Setting a variable is very easy: just pass the data along.

```

21377 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:N } #1
21378   { \__keys_variable_set_required:NnnN #1 { muskip } { } n }
21379 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_set:c } #1
21380   { \__keys_variable_set_required:cnnN {#1} { muskip } { } n }
21381 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:N } #1
21382   { \__keys_variable_set_required:NnnN #1 { muskip } { g } n }
21383 \cs_new_protected:cpn { \c__keys_props_root_str .muskip_gset:c } #1
21384   { \__keys_variable_set_required:cnnN {#1} { muskip } { g } n }

```

(End definition for `.muskip_set:N` and `.muskip_gset:N`. These functions are documented on page [227](#).)

**.prop\_put:N** Setting a variable is very easy: just pass the data along.

```

21385 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:N } #1
21386   { \__keys_prop_put:Nn #1 { } }
21387 \cs_new_protected:cpn { \c__keys_props_root_str .prop_put:c } #1
21388   { \__keys_prop_put:cn {#1} { } }
21389 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:N } #1
21390   { \__keys_prop_put:Nn #1 { g } }
21391 \cs_new_protected:cpn { \c__keys_props_root_str .prop_gput:c } #1
21392   { \__keys_prop_put:cn {#1} { g } }

```

(End definition for `.prop_put:N` and `.prop_gput:N`. These functions are documented on page [227](#).)

**.skip\_set:N** Setting a variable is very easy: just pass the data along.

```

21393 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:N } #1
21394   { \__keys_variable_set_required:NnnN #1 { skip } { } n }
21395 \cs_new_protected:cpn { \c__keys_props_root_str .skip_set:c } #1
21396   { \__keys_variable_set_required:cnnN {#1} { skip } { } n }
21397 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:N } #1
21398   { \__keys_variable_set_required:NnnN #1 { skip } { g } n }
21399 \cs_new_protected:cpn { \c__keys_props_root_str .skip_gset:c } #1
21400   { \__keys_variable_set_required:cnnN {#1} { skip } { g } n }

```

(End definition for `.skip_set:N` and `.skip_gset:N`. These functions are documented on page [227](#).)

**.str\_set:N** Setting a variable is very easy: just pass the data along.

```

21401 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:N } #1
21402   { \__keys_variable_set:NnnN #1 { str } { } n }
21403 \cs_new_protected:cpn { \c__keys_props_root_str .str_set:c } #1
21404   { \__keys_variable_set:cnnN {#1} { str } { } n }
21405 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:N } #1
21406   { \__keys_variable_set:NnnN #1 { str } { } x }
21407 \cs_new_protected:cpn { \c__keys_props_root_str .str_set_x:c } #1
21408   { \__keys_variable_set:cnnN {#1} { str } { } x }
21409 \cs_new_protected:cpn { \c__keys_props_root_str .str_gset:N } #1
21410   { \__keys_variable_set:NnnN #1 { str } { g } n }

```

```

21411 \cs_new_protected:cpn { \c_keys_props_root_str .str_gset:c } #1
21412   { \_keys_variable_set:cnnN {#1} { str } { g } n }
21413 \cs_new_protected:cpn { \c_keys_props_root_str .str_gset_x:N } #1
21414   { \_keys_variable_set:NnnN #1 { str } { g } x }
21415 \cs_new_protected:cpn { \c_keys_props_root_str .str_gset_x:c } #1
21416   { \_keys_variable_set:cnnN {#1} { str } { g } x }

```

(End definition for `.str_set:N` and others. These functions are documented on page [227](#).)

**.tl\_set:N** Setting a variable is very easy: just pass the data along.

```

21417 \cs_new_protected:cpn { \c_keys_props_root_str .tl_set:N } #1
21418   { \_keys_variable_set:NnnN #1 { tl } { } n }
21419 \cs_new_protected:cpn { \c_keys_props_root_str .tl_set:c } #1
21420   { \_keys_variable_set:cnnN {#1} { tl } { } n }
21421 \cs_new_protected:cpn { \c_keys_props_root_str .tl_set_x:N } #1
21422   { \_keys_variable_set:NnnN #1 { tl } { } x }
21423 \cs_new_protected:cpn { \c_keys_props_root_str .tl_set_x:c } #1
21424   { \_keys_variable_set:cnnN {#1} { tl } { } x }
21425 \cs_new_protected:cpn { \c_keys_props_root_str .tl_gset:N } #1
21426   { \_keys_variable_set:NnnN #1 { tl } { g } n }
21427 \cs_new_protected:cpn { \c_keys_props_root_str .tl_gset:c } #1
21428   { \_keys_variable_set:cnnN {#1} { tl } { g } n }
21429 \cs_new_protected:cpn { \c_keys_props_root_str .tl_gset_x:N } #1
21430   { \_keys_variable_set:NnnN #1 { tl } { g } x }
21431 \cs_new_protected:cpn { \c_keys_props_root_str .tl_gset_x:c } #1
21432   { \_keys_variable_set:cnnN {#1} { tl } { g } x }

```

(End definition for `.tl_set:N` and others. These functions are documented on page [227](#).)

**.undefined:** Another simple wrapper.

```

21433 \cs_new_protected:cpn { \c_keys_props_root_str .undefined: }
21434   { \_keys_undefined: }

```

(End definition for `.undefined:`. This function is documented on page [228](#).)

**.usage:n**

```

21435 \cs_new_protected:cpn { \c_keys_props_root_str .usage:n } #1
21436   { \_keys_usage:n {#1} }

```

(End definition for `.usage:n`. This function is documented on page [231](#).)

**.value\_forbidden:n** These are very similar, so both call the same function.

```

21437 \cs_new_protected:cpn { \c_keys_props_root_str .value_forbidden:n } #1
21438   { \_keys_value_requirement:nn { forbidden } {#1} }
21439 \cs_new_protected:cpn { \c_keys_props_root_str .value_required:n } #1
21440   { \_keys_value_requirement:nn { required } {#1} }

```

(End definition for `.value_forbidden:n` and `.value_required:n`. These functions are documented on page [228](#).)

## 63.6 Setting keys

```
\keys_set:nn
\keys_set:nV
\keys_set:nv
\keys_set:no
\_keys_set:nn
\_keys_set:nnn
21441 \cs_new_protected:Npn \keys_set:nn #1#2
21442 {
21443     \use:x
21444     {
21445         \bool_set_false:N \exp_not:N \l__keys_only_known_bool
21446         \bool_set_false:N \exp_not:N \l__keys_filtered_bool
21447         \bool_set_false:N \exp_not:N \l__keys_selective_bool
21448         \tl_set:Nn \exp_not:N \l__keys_relative_tl
21449         { \exp_not:N \q__keys_no_value }
21450         \_keys_set:nn \exp_not:n { #1} {#2}
21451         \bool_if:NT \l__keys_only_known_bool
21452             { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
21453         \bool_if:NT \l__keys_filtered_bool
21454             { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
21455         \bool_if:NT \l__keys_selective_bool
21456             { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
21457         \tl_set:Nn \exp_not:N \l__keys_relative_tl
21458             { \exp_not:o \l__keys_relative_tl }
21459     }
21460 }
21461 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
21462 \cs_new_protected:Npn \_keys_set:nn #1#2
21463 { \exp_args:No \_keys_set:nnn \l__keys_module_str {#1} {#2} }
21464 \cs_new_protected:Npn \_keys_set:nnn #1#2#3
21465 {
21466     \str_set:Nx \l__keys_module_str { \_keys_trim_spaces:n {#2} }
21467     \keyval_parse:NNn \_keys_set_keyval:n \_keys_set_keyval:nn {#3}
21468     \str_set:Nn \l__keys_module_str {#1}
21469 }
```

(End definition for `\keys_set:nn`, `\_keys_set:nn`, and `\_keys_set:nnn`. This function is documented on page [231](#).)

```
\keys_set_known:nnN
\keys_set_known:nVN
\keys_set_known:nvN
\keys_set_known:nnN
\keys_set_known:nnnN
\keys_set_known:nnN
\keys_set_known:nV
\keys_set_known:nv
\keys_set_known:nn
\keys_set_known:nnn
\keys_set_known:nnnN
21470 \cs_new_protected:Npn \keys_set_known:nnN #1#2#3
21471 {
21472     \exp_args:No \_keys_set_known:nnnnN
21473         \l__keys_unused_clist \q__keys_no_value {#1} {#2} #3
21474 }
21475 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
21476 \cs_new_protected:Npn \keys_set_known:nnnN #1#2#3#4
21477 {
21478     \exp_args:No \_keys_set_known:nnnnN
21479         \l__keys_unused_clist {#3} {#1} {#2} #4
21480 }
21481 \cs_generate_variant:Nn \keys_set_known:nnnN { nV , nv , no }
21482 \cs_new_protected:Npn \_keys_set_known:nnnnN #1#2#3#4#5
21483 {
```

```

21484     \clist_clear:N \l__keys_unused_clist
21485     \__keys_set_known:n {#2} {#3} {#4}
21486     \__kernel_tl_set:Nx #5 { \exp_not:o \l__keys_unused_clist }
21487     \tl_set:Nn \l__keys_unused_clist {#1}
21488   }
21489   \cs_new_protected:Npn \keys_set_known:nn #1#2
21490   { \__keys_set_known:n {#1} \q__keys_no_value {#2} }
21491   \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
21492   \cs_new_protected:Npn \__keys_set_known:n {#1} {#2} {#3}
21493   {
21494     \use:x
21495     {
21496       \bool_set_true:N \exp_not:N \l__keys_only_known_bool
21497       \bool_set_false:N \exp_not:N \l__keys_filtered_bool
21498       \bool_set_false:N \exp_not:N \l__keys_selective_bool
21499       \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
21500       \__keys_set:nn \exp_not:n { {#2} {#3} }
21501       \bool_if:NF \l__keys_only_known_bool
21502         { \bool_set_false:N \exp_not:N \l__keys_only_known_bool }
21503       \bool_if:NT \l__keys_filtered_bool
21504         { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
21505       \bool_if:NT \l__keys_selective_bool
21506         { \bool_set_true:N \exp_not:N \l__keys_selective_bool }
21507       \tl_set:Nn \exp_not:N \l__keys_relative_tl
21508         { \exp_not:o \l__keys_relative_tl }
21509     }
21510   }

```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page 232.)

`\keys_set_filter:nnN`  
`\keys_set_filter:nnVN`  
`\keys_set_filter:nnvN`  
`\keys_set_filter:nnoN`  
`\keys_set_filter:nnnnN`  
`\keys_set_filter:nnVnN`  
`\keys_set_filter:nnvnN`  
`\keys_set_filter:nnonN`  
`\__keys_set_filter:nnnnnN`  
`\keys_set_filter:nnn`  
`\keys_set_filter:nnV`  
`\keys_set_filter:nnv`  
`\keys_set_filter:nnO`  
`\__keys_set_filter:nnnn`  
`\keys_set_groups:nnN`  
`\keys_set_groups:nnVN`  
`\keys_set_groups:nnvN`  
`\keys_set_groups:nnO`  
`\__keys_set_selective:nnN`  
`\__keys_set_selective:nnnnN`

The idea of setting keys in a selective manner again uses flags wrapped around the basic code. The comments on `\keys_set_known:nnN` also apply here. We have a bit more shuffling to do to keep everything nestable.

```

21511 \cs_new_protected:Npn \keys_set_filter:nnnnN #1#2#3#4
21512   {
21513     \exp_args:No \__keys_set_filter:nnnnnN
21514     \l__keys_unused_clist
21515       \q__keys_no_value {#1} {#2} {#3} #4
21516   }
21517 \cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
21518 \cs_new_protected:Npn \keys_set_filter:nnnnN #1#2#3#4#5
21519   {
21520     \exp_args:No \__keys_set_filter:nnnnnN
21521       \l__keys_unused_clist {#4} {#1} {#2} {#3} #5
21522   }
21523 \cs_generate_variant:Nn \keys_set_filter:nnnnN { nnV , nnv , nno }
21524 \cs_new_protected:Npn \__keys_set_filter:nnnnnN #1#2#3#4#5#6
21525   {
21526     \clist_clear:N \l__keys_unused_clist
21527     \__keys_set_filter:nnnn {#2} {#3} {#4} {#5}
21528     \__kernel_tl_set:Nx #6 { \exp_not:o \l__keys_unused_clist }
21529     \tl_set:Nn \l__keys_unused_clist {#1}
21530   }
21531 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3

```

```

21532   {\_\_keys\_set\_filter:nnnn \q\_keys\_no\_value {#1} {#2} {#3} }
21533   \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
21534   \cs_new_protected:Npn \_\_keys_set_filter:nnnn #1#2#3#4
21535   {
21536     \use:x
21537     {
21538       \bool_set_false:N \exp_not:N \l__keys_only_known_bool
21539       \bool_set_true:N \exp_not:N \l__keys_filtered_bool
21540       \bool_set_true:N \exp_not:N \l__keys_selective_bool
21541       \tl_set:Nn \exp_not:N \l__keys_relative_tl { \exp_not:n {#1} }
21542       \_\_keys_set_selective:nnn \exp_not:n { {#2} {#3} {#4} }
21543       \bool_if:NT \l__keys_only_known_bool
21544         { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
21545       \bool_if:NF \l__keys_filtered_bool
21546         { \bool_set_false:N \exp_not:N \l__keys_filtered_bool }
21547       \bool_if:NF \l__keys_selective_bool
21548         { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
21549       \tl_set:Nn \exp_not:N \l__keys_relative_tl
21550         { \exp_not:o \l__keys_relative_tl }
21551     }
21552   }
21553 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
21554   {
21555     \use:x
21556     {
21557       \bool_set_false:N \exp_not:N \l__keys_only_known_bool
21558       \bool_set_false:N \exp_not:N \l__keys_filtered_bool
21559       \bool_set_true:N \exp_not:N \l__keys_selective_bool
21560       \tl_set:Nn \exp_not:N \l__keys_relative_tl
21561         { \exp_not:N \q\_keys_no_value }
21562       \_\_keys_set_selective:nnn \exp_not:n { {#1} {#2} {#3} }
21563       \bool_if:NT \l__keys_only_known_bool
21564         { \bool_set_true:N \exp_not:N \l__keys_only_known_bool }
21565       \bool_if:NF \l__keys_filtered_bool
21566         { \bool_set_true:N \exp_not:N \l__keys_filtered_bool }
21567       \bool_if:NF \l__keys_selective_bool
21568         { \bool_set_false:N \exp_not:N \l__keys_selective_bool }
21569       \tl_set:Nn \exp_not:N \l__keys_relative_tl
21570         { \exp_not:o \l__keys_relative_tl }
21571     }
21572   }
21573 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
21574 \cs_new_protected:Npn \_\_keys_set_selective:nnn
21575   { \exp_args:No \_\_keys_set_selective:nnnn \l__keys_selective_seq }
21576 \cs_new_protected:Npn \_\_keys_set_selective:nnnn #1#2#3#4
21577   {
21578     \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
21579     \_\_keys_set:nn {#2} {#4}
21580     \tl_set:Nn \l__keys_selective_seq {#1}
21581   }

```

(End definition for `\keys_set_filter:nnnN` and others. These functions are documented on page 233.)

`\keys_compile:nnN` A simple wrapper.

```

21582 \cs_new_protected:Npn \keys_compile:nnN #1#2#3
21583 {
21584     \bool_set_true:N \l__keys_compile_bool
21585     \tl_clear:N \l__keys_compile_tl
21586     \keys_set:nn {#1} {#2}
21587     \bool_set_false:N \l__keys_compile_bool
21588     \tl_set_eq:NN #3 \l__keys_compile_tl
21589 }

```

(End definition for `\keys_compile:nnN`. This function is documented on page 233.)

`\__keys_set_keyval:n`  
`\__keys_set_keyval:nn`  
`\__keys_set_keyval:nnn`  
`\__keys_set_keyval:onn`  
`\__keys_find_key_module:wNN`  
`\__keys_find_key_module_auxi:Nw`  
`\__keys_find_key_module_auxii:Nw`  
`\__keys_find_key_module_auxiii:Nn`  
`\__keys_find_key_module_auxiv:Nw`  
`\__keys_set_selective:`

A shared system once again. First, set the current path and add a default if needed. There are then checks to see if the a value is required or forbidden. If everything passes, move on to execute the code.

```

21590 \cs_new_protected:Npn \__keys_set_keyval:n #1
21591 {
21592     \bool_set_true:N \l__keys_no_value_bool
21593     \__keys_set_keyval:onn \l__keys_module_str {#1} { }
21594 }
21595 \cs_new_protected:Npn \__keys_set_keyval:nn #1#2
21596 {
21597     \bool_set_false:N \l__keys_no_value_bool
21598     \__keys_set_keyval:onn \l__keys_module_str {#1} {#2}
21599 }

```

The key path here can be fully defined, after which there is a search for the key and module names: the user may have passed them with part of what is actually the module (for our purposes) in the key name. As that happens on a per-key basis, we use the stack approach to restore the module name without a group.

```

21600 \cs_new_protected:Npn \__keys_set_keyval:nnn #1#2#3
21601 {
21602     \__kernel_tl_set:Nx \l__keys_path_str
21603     {
21604         \tl_if_blank:nF {#1}
21605         { #1 / }
21606         \__keys_trim_spaces:n {#2}
21607     }
21608     \str_clear:N \l__keys_module_str
21609     \str_clear:N \l__keys_inherit_str
21610     \exp_after:wN \__keys_find_key_module:wNN \l__keys_path_str \s__keys_stop
21611         \l__keys_module_str \l__keys_key_str
21612     \tl_set_eq:NN \l__keys_key_tl \l__keys_key_str
21613     \__keys_value_or_default:n {#3}
21614     \bool_if:NTF \l__keys_selective_bool
21615         \__keys_set_selective:
21616         \__keys_execute:
21617         \str_set:Nn \l__keys_module_str {#1}
21618     }
21619 \cs_generate_variant:Nn \__keys_set_keyval:nnn { o }

```

This function uses `\cs_set_nopar:Npx` internally for performance reasons, the argument `#1` is already a string in every usage, so turning it into a string again seems unnecessary.

```

21620 \cs_new_protected:Npn \__keys_find_key_module:wNN #1 \s__keys_stop #2 #3
21621 {
21622     \__keys_find_key_module_auxi:Nw #2 #1 \s__keys_nil \__keys_find_key_module_auxii:Nw

```

```

21623      / \s__keys_nil \__keys_find_key_module_auxiv:Nw #3
21624    }
21625 \cs_new_protected:Npn \__keys_find_key_module_auxi:Nw #1 #2 / #3 \s__keys_nil #4
21626  {
21627    #4 #1 #2 \s__keys_mark #3 \s__keys_nil #4
21628  }
21629 \cs_new_protected:Npn \__keys_find_key_module_auxii:Nw
21630    #1 #2 \s__keys_mark #3 \s__keys_nil \__keys_find_key_module_auxii:Nw
21631  {
21632    \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
21633    \__keys_find_key_module_auxi:Nw #1 #3 \s__keys_nil \__keys_find_key_module_auxiii:Nw
21634  }
21635 \cs_new_protected:Npn \__keys_find_key_module_auxiii:Nw #1 #2 \s__keys_mark
21636  {
21637    \cs_set_nopar:Npx #1 { \tl_if_empty:NF #1 { #1 / } #2 }
21638    \__keys_find_key_module_auxi:Nw #1
21639  }
21640 \cs_new_protected:Npn \__keys_find_key_module_auxiv:Nw
21641    #1 #2 \s__keys_nil #3 \s__keys_mark
21642    \s__keys_nil \__keys_find_key_module_auxiv:Nw #4
21643  {
21644    \cs_set_nopar:Npn #4 { #2 }
21645  }

```

If selective setting is active, there are a number of possible sub-cases to consider. The key name may not be known at all or if it is, it may not have any groups assigned. There is then the question of whether the selection is opt-in or opt-out.

```

21646 \cs_new_protected:Npn \__keys_set_selective:
21647  {
21648    \cs_if_exist:cTF { \c__keys_groups_root_str \l_keys_path_str }
21649    {
21650      \clist_set_eq:Nc \l__keys_groups_clist
21651        { \c__keys_groups_root_str \l_keys_path_str }
21652      \__keys_check_groups:
21653    }
21654    {
21655      \bool_if:NTF \l__keys_filtered_bool
21656        \__keys_execute:
21657        \__keys_store_unused:
21658    }
21659  }

```

In the case where selective setting requires a comparison of the list of groups which apply to a key with the list of those which have been set active. That requires two mappings, and again a different outcome depending on whether opt-in or opt-out is set. We cannot replace the clist mapping by \clist\_if\_in:NnTF because catcodes may not be the same; they cannot be normalized easily in the clist because of the remote possibility that some items need braces if they involve commas or leading/trailing spaces.

```

21660 \cs_new_protected:Npn \__keys_check_groups:
21661  {
21662    \bool_set_false:N \l__keys_tmp_bool
21663    \seq_map_inline:Nn \l__keys_selective_seq
21664    {
21665      \clist_map_inline:Nn \l__keys_groups_clist

```

```

21666 {
21667   \str_if_eq:nnT {##1} {####1}
21668   {
21669     \bool_set_true:N \l__keys_tmp_bool
21670     \clist_map_break:n \seq_map_break:
21671   }
21672 }
21673 }
21674 \bool_if:NTF \l__keys_tmp_bool
21675 {
21676   \bool_if:NTF \l__keys_filtered_bool
21677   \__keys_store_unused:
21678   \__keys_execute:
21679 }
21680 {
21681   \bool_if:NTF \l__keys_filtered_bool
21682   \__keys_execute:
21683   \__keys_store_unused:
21684 }
21685 }

```

(End definition for `\__keys_set_keyval:n` and others.)

`\__keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```

\__keys_default_inherit:
21686 \cs_new_protected:Npn \__keys_value_or_default:n #1
21687 {
21688   \bool_if:NTF \l__keys_no_value_bool
21689   {
21690     \cs_if_exist:cTF { \c__keys_default_root_str \l_keys_path_str }
21691     {
21692       \tl_set_eq:Nc
21693       \l_keys_value_tl
21694       { \c__keys_default_root_str \l_keys_path_str }
21695     }
21696   {
21697     \tl_clear:N \l_keys_value_tl
21698     \cs_if_exist:cT
21699     { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21700     { \__keys_default_inherit: }
21701   }
21702 }
21703 { \tl_set:Nn \l_keys_value_tl {#1} }
21704 }
21705 \cs_new_protected:Npn \__keys_default_inherit:
21706 {
21707   \clist_map_inline:cn
21708   { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21709   {
21710     \cs_if_exist:cT
21711     { \c__keys_default_root_str ##1 / \l_keys_key_str }
21712     {
21713       \tl_set_eq:Nc
21714       \l_keys_value_tl
21715       { \c__keys_default_root_str ##1 / \l_keys_key_str }

```

```

21716           \clist_map_break:
21717       }
21718   }
21719 }

(End definition for \__keys_value_or_default:n and \__keys_default_inherit:.)
```

\\_\_keys\_execute:  
\\_\_keys\_execute\_inherit:  
\\_\_keys\_execute\_unknown:  
\\_\_keys\_execute:nn  
\\_\_keys\_execute:no  
\\_\_keys\_store\_unused:  
\\_\_keys\_store\_unused\_aux:

Actually executing a key is done in two parts. First, look for the key itself, then look for the `unknown` key with the same path. If both of these fail, complain. What exactly happens if a key is unknown depends on whether unknown keys are being skipped or if an error should be raised.

```

21720 \cs_new_protected:Npn \__keys_execute:
21721 {
21722     \cs_if_exist:cTF { \c__keys_code_root_str \l_keys_path_str }
21723     {
21724         \cs_if_exist_use:c { \c__keys_check_root_str \l_keys_path_str }
21725         \__keys_execute:no \l_keys_path_str \l_keys_value_tl
21726     }
21727     {
21728         \cs_if_exist:cTF
21729             { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21730             { \__keys_execute_inherit: }
21731             { \__keys_execute_unknown: }
21732     }
21733 }
```

To deal with the case where there is no hit, we leave `\__keys_execute_unknown:` in the input stream and clean it up using the `break` function: that avoids needing a boolean.

```

21734 \cs_new_protected:Npn \__keys_execute_inherit:
21735 {
21736     \clist_map_inline:cn
21737     { \c__keys_inherit_root_str \__keys_parent:o \l_keys_path_str }
21738     {
21739         \cs_if_exist:cT
21740             { \c__keys_code_root_str ##1 / \l_keys_key_str }
21741             {
21742                 \str_set:Nn \l__keys_inherit_str {##1}
21743                 \cs_if_exist_use:c { \c__keys_check_root_str ##1 / \l_keys_key_str }
21744                 \__keys_execute:no { ##1 / \l_keys_key_str } \l_keys_value_tl
21745                 \clist_map_break:n \use_none:n
21746             }
21747         }
21748     \__keys_execute_unknown:
21749 }
21750 \cs_new_protected:Npn \__keys_execute_unknown:
21751 {
21752     \bool_if:NTF \l__keys_only_known_bool
21753         { \__keys_store_unused: }
21754         {
21755             \cs_if_exist:cTF
21756                 { \c__keys_code_root_str \l__keys_module_str / unknown }
21757                 { \__keys_execute:no { \l__keys_module_str / unknown } \l_keys_value_tl }
21758             {
21759                 \msg_error:nnnx { keys } { unknown }
```

```

21760           \l_keys_path_str \l_keys_module_str
21761       }
21762   }
21763 }
```

A key's code is in the control sequence with csname `\c__keys_code_root_str #1`. We expand it once to get the replacement text (with argument #2) and call `\use:n` with this replacement as its argument. This ensures that any undefined control sequence error in the key's code will lead to an error message of the form `<argument>...<control sequence>` in which one can read the (undefined) `<control sequence>` in full, rather than an error message that starts with the potentially very long key name, which would make the (undefined) `<control sequence>` be truncated or sometimes completely hidden. See <https://github.com/latex3/latex2e/issues/351>.

```

21764 \cs_new:Npn \__keys_execute:nn #1#
21765   { \__keys_execute:no {#1} { \prg_do_nothing: #2 } }
21766 \cs_new:Npn \__keys_execute:no #1#2
21767   {
21768     \exp_args:NNo \exp_args:No \use:n
21769     {
21770       \cs:w \c__keys_code_root_str #1 \exp_after:wN \cs_end:
21771       \exp_after:wN {#2}
21772     }
21773 }
```

When there is no relative path, things here are easy: just save the key name and value. When we are working with a relative path, first we need to turn it into a string: that can't happen earlier as we need to store `\q__keys_no_value`. Then, use a standard delimited approach to fish out the partial path.

```

21774 \cs_new_protected:Npn \__keys_store_unused:
21775   {
21776     \__keys_quark_if_no_value:NTF \l__keys_relative_tl
21777     {
21778       \clist_put_right:Nx \l__keys_unused_clist
21779       {
21780         \l_keys_key_str
21781         \bool_if:NF \l__keys_no_value_bool
21782           { = { \exp_not:o \l_keys_value_tl } }
21783       }
21784     }
21785   {
21786     \tl_if_empty:NTF \l__keys_relative_tl
21787     {
21788       \clist_put_right:Nx \l__keys_unused_clist
21789       {
21790         \l_keys_path_str
21791         \bool_if:NF \l__keys_no_value_bool
21792           { = { \exp_not:o \l_keys_value_tl } }
21793       }
21794     }
21795   { \__keys_store_unused_aux: }
21796 }
21797 }
21798 \cs_new_protected:Npn \__keys_store_unused_aux:
21799   {
```

```

21800   \__kernel_tl_set:Nx \l__keys_relative_tl
21801     { \exp_args:No \__keys_trim_spaces:n \l__keys_relative_tl }
21802   \use:x
21803   {
21804     \cs_set_protected:Npn \__keys_store_unused:w
21805       #####1 \l__keys_relative_tl /
21806       #####2 \l__keys_relative_tl /
21807       #####3 \s__keys_stop
21808   }
21809   {
21810     \tl_if_blank:nF {##1}
21811     {
21812       \msg_error:nnnx { keys } { bad-relative-key-path }
21813         \l_keys_path_str
21814         \l__keys_relative_tl
21815     }
21816     \clist_put_right:Nx \l__keys_unused_clist
21817     {
21818       \exp_not:n {##2}
21819       \bool_if:NF \l__keys_no_value_bool
21820         { = { \exp_not:o \l_keys_value_tl } }
21821     }
21822   }
21823   \use:x
21824   {
21825     \__keys_store_unused:w \l_keys_path_str
21826       \l__keys_relative_tl / \l__keys_relative_tl /
21827       \s__keys_stop
21828   }
21829 }
21830 \cs_new_protected:Npn \__keys_store_unused:w { }

(End definition for \__keys_execute: and others.)

```

\\_\_keys\_choice\_find:n  
\\_\_keys\_choice\_find:nn  
\\_\_keys\_multichoice\_find:n

Executing a choice has two parts. First, try the choice given, then if that fails call the unknown key. That always exists, as it is created when a choice is first made. So there is no need for any escape code. For multiple choices, the same code ends up used in a mapping.

```

21831 \cs_new:Npn \__keys_choice_find:n #1
21832   {
21833     \str_if_empty:NTF \l__keys_inherit_str
21834       { \__keys_choice_find:nn \l_keys_path_str {#1} }
21835       {
21836         \__keys_choice_find:nn
21837           { \l__keys_inherit_str / \l_keys_key_str } {#1}
21838       }
21839   }
21840 \cs_new:Npn \__keys_choice_find:nn #1#2
21841   {
21842     \cs_if_exist:cTF { \c__keys_code_root_str #1 / \__keys_trim_spaces:n {#2} }
21843       { \__keys_execute:nn { #1 / \__keys_trim_spaces:n {#2} } {#2} }
21844       { \__keys_execute:nn { #1 / unknown } {#2} }
21845   }
21846 \cs_new:Npn \__keys_multichoice_find:n #1

```

```

21847 { \clist_map_function:nN {#1} \__keys_choice_find:n }

(End definition for \__keys_choice_find:n, \__keys_choice_find:nn, and \__keys_multichoice-
find:n.)

```

## 63.7 Utilities

\\_\_keys\_parent:o Used to strip off the ending part of the key path after the last /.

```

21848 \cs_new:Npn \__keys_parent:o #1
21849 {
21850     \exp_after:wN \__keys_parent_auxi:w #1 \q_nil \__keys_parent_auxii:w
21851     / \q_nil \__keys_parent_auxiv:w
21852 }
21853 \cs_new:Npn \__keys_parent_auxi:w #1 / #2 \q_nil #3
21854 {
21855     #3 { #1 } #2 \q_nil #3
21856 }
21857 \cs_new:Npn \__keys_parent_auxii:w #1 #2 \q_nil \__keys_parent_auxii:w
21858 {
21859     #1 \__keys_parent_auxi:w #2 \q_nil \__keys_parent_auxiii:n
21860 }
21861 \cs_new:Npn \__keys_parent_auxiii:n #1
21862 {
21863     / #1 \__keys_parent_auxi:w
21864 }
21865 \cs_new:Npn \__keys_parent_auxiv:w #1 \q_nil \__keys_parent_auxiv:w
21866 {
21867 }

```

*(End definition for \\_\_keys\_parent:o and others.)*

\\_\_keys\_trim\_spaces:n Space stripping has to allow for the fact that the key here might have several parts, and spaces need to be stripped from each part. Since the key name is turned into a string groups can't be stripped accidentally and the precautions of \tl\_trim\_spaces:n aren't necessary, in this case it is much faster to just directly strip spaces around /.

```

21868 \group_begin:
21869   \cs_set:Npn \__keys_tmp:w #1
21870   {
21871     \cs_new:Npn \__keys_trim_spaces:n ##1
21872     {
21873         \exp_after:wN \__keys_trim_spaces_auxi:w \tl_to_str:n { / ##1 } /
21874         \s__keys_nil \__keys_trim_spaces_auxi:w
21875         \s__keys_mark \__keys_trim_spaces_auxii:w
21876         #1 / #
21877         \s__keys_nil \__keys_trim_spaces_auxii:w
21878         \s__keys_mark \__keys_trim_spaces_auxiii:w
21879     }
21880   }
21881   \__keys_tmp:w { ~ }
21882 \group_end:
21883 \cs_new:Npn \__keys_trim_spaces_auxi:w #1 ~ / #2 \s__keys_nil #3
21884 {
21885     #3 #1 / #2 \s__keys_nil #3

```

```

21886      }
21887  \cs_new:Npn \__keys_trim_spaces_auxii:w #1 / ~ #2 \s__keys_mark #3
21888  {
21889      #3 #1 / #2 \s__keys_mark #3
21890  }
21891  \cs_new:Npn \__keys_trim_spaces_auxiii:w
21892      / #1 /
21893      \s__keys_nil  \__keys_trim_spaces_auxi:w
21894      \s__keys_mark \__keys_trim_spaces_auxii:w
21895      /
21896      \s__keys_nil  \__keys_trim_spaces_auxii:w
21897      \s__keys_mark \__keys_trim_spaces_auxiii:w
21898  {
21899      #1
21900  }

```

(End definition for `\__keys_trim_spaces:n` and others.)

### `\keys_if_exist:p:nn`

### `\keys_if_exist:nnTF`

A utility for others to see if a key exists.

```

21901  \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
21902  {
21903      \cs_if_exist:cTF
21904          { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 } }
21905          { \prg_return_true: }
21906          { \prg_return_false: }
21907  }
21908  \prg_generate_conditional_variant:Nnn \keys_if_exist:nn { ne } { T , F , TF }

```

(End definition for `\keys_if_exist:nnTF`. This function is documented on page 234.)

### `\keys_if_choice_exist:p:nnn`

### `\keys_if_choice_exist:nnnTF`

Just an alternative view on `\keys_if_exist:nnTF`.

```

21909  \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
21910  { p , T , F , TF }
21911  {
21912      \cs_if_exist:cTF
21913          { \c__keys_code_root_str \__keys_trim_spaces:n { #1 / #2 / #3 } }
21914          { \prg_return_true: }
21915          { \prg_return_false: }
21916  }

```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 234.)

### `\keys_show:nn`

### `\keys_log:nn`

To show a key, show its code using a message.

```

21917  \cs_new_protected:Npn \keys_show:nn
21918  { \__keys_show:Nnn \msg_show:nnxxxx }
21919  \cs_new_protected:Npn \keys_log:nn
21920  { \__keys_show:Nnn \msg_log:nnxxxx }
21921  \cs_new_protected:Npn \__keys_show:Nnn #1#2#3
21922  {
21923      #1 { keys } { show-key }
21924          { \__keys_trim_spaces:n { #2 / #3 } }
21925          {
21926              \keys_if_exist:nnT {#2} {#3}
21927                  {
21928                      \exp_args:Nnf \msg_show_item_unbraced:nn { code }

```

```

21929 {
21930   \exp_args:Nc \__keys_show:n
21931   {
21932     \exp_args:Nc \cs_replacement_spec:N
21933     {
21934       \c__keys_code_root_str
21935       \__keys_trim_spaces:n { #2 / #3 }
21936     }
21937   }
21938 }
21939 }
21940 { } { }
21941 }
21942 }
21943 \cs_new:Npx \__keys_show:n #1
21944 {
21945   \exp_not:N \__keys_show:w
21946   #1
21947   \tl_to_str:n { \__keys_compile:n }
21948   #1
21949   \tl_to_str:n { \__keys_compile:n }
21950   \exp_not:N \s__keys_stop
21951 }
21952 \use:x
21953 {
21954   \cs_new:Npn \exp_not:N \__keys_show:w
21955   ##1 \tl_to_str:n { \__keys_compile:n }
21956   ##2 \tl_to_str:n { \__keys_compile:n }
21957   ##3 \exp_not:N \s__keys_stop
21958 }
21959 {
21960   \tl_if_blank:nTF {#2}
21961   {#1}
21962   { \__keys_show:Nw #2 \s__keys_stop }
21963 }
21964 \use:x
21965 {
21966   \cs_new:Npn \exp_not:N \__keys_show:Nw ##1##2
21967   \c_right_brace_str \exp_not:N \s__keys_stop
21968 }
21969 {#2}

```

(End definition for `\keys_show:nn` and others. These functions are documented on page 234.)

## 63.8 Messages

For when there is a need to complain.

```

21970 \msg_new:nnnn { keys } { bad-relative-key-path }
21971   { The~key~'#1'~is~not~inside~the~'#2~path. }
21972   { The~key~'#1'~cannot~be~expressed~relative~to~path~'#2'. }
21973 \msg_new:nnnn { keys } { boolean-values-only }
21974   { Key~'#1'~accepts~boolean~values~only. }
21975   { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }

```

```

21976 \msg_new:nnnn { keys } { choice-unknown }
21977   { Key~'#1'~accepts~only-a~fixed~set~of~choices. }
21978   {
21979     The~key~'#1'~only~accepts~predefined~values,~
21980     and~'#2'~is~not~one~of~these.
21981   }
21982 \msg_new:nnnn { keys } { unknown }
21983   { The~key~'#1'~is~unknown~and~is~being~ignored. }
21984   {
21985     The~module~'#2'~does~not~have~a~key~called~'#1'.\\\
21986     Check~that~you~have~spelled~the~key~name~correctly.
21987   }
21988 \msg_new:nnnn { keys } { nested-choice-key }
21989   { Attempt~to~define~'#1'~as~a~nested~choice~key. }
21990   {
21991     The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
21992     itself~a~choice.
21993   }
21994 \msg_new:nnnn { keys } { value-forbidden }
21995   { The~key~'#1'~does~not~take~a~value. }
21996   {
21997     The~key~'#1'~should~be~given~without~a~value.\\\
21998     The~value~'#2'~was~present:~the~key~will~be~ignored.
21999   }
22000 \msg_new:nnnn { keys } { value-required }
22001   { The~key~'#1'~requires~a~value. }
22002   {
22003     The~key~'#1'~must~have~a~value.\\\
22004     No~value~was~present:~the~key~will~be~ignored.
22005   }
22006 \msg_new:nnn { keys } { show-key }
22007   {
22008     The~key~#1~
22009     \tl_if_empty:nTF {#2}
22010       { is~undefined. }
22011       { has~the~properties: #2 . }
22012   }
22013 \prop_gput:Nnn \g_msg_module_name_prop { keys } { LaTeX }
22014 \prop_gput:Nnn \g_msg_module_type_prop { keys } { }
22015 
```

# Chapter 64

## I3intarray implementation

```
22016  <*package>
22017  <@=intarray>
```

There are two implementations for this module: One `\fontdimen` based one for more traditional TeX engines and a Lua based one for engines with Lua support.

Both versions do not allow negative array sizes.

```
22018  <*tex>
22019  \msg_new:nnn { kernel } { negative-array-size }
22020    { Size-of~array-may-not-be-negative:#1 }
```

`\l__intarray_loop_int` A loop index.

```
22021  \int_new:N \l__intarray_loop_int
(End definition for \l__intarray_loop_int.)
```

### 64.1 Lua implementation

First, let's look at the Lua variant:

We select the Lua version if the Lua helpers were defined. This can be detected by the presence of `\__intarray_gset_count:Nw`.

```
22022  \cs_if_exist:NTF \__intarray_gset_count:Nw
22023    {
```

#### 64.1.1 Allocating arrays

Used to differentiate intarrays in Lua and to record an invalid index.

```
22024    \int_new:N \g__intarray_table_int
22025    \int_new:N \l__intarray_bad_index_int
22026  /tex>
```

(End definition for `\g__intarray_table_int` and `\l__intarray_bad_index_int`.)

`\__intarray:w` Used as marker for intarrays in Lua. Followed by an unbraced number identifying the array and a single space. This format is used to make it easy to scan from Lua.

```
22027  <*lua>
22028  luacmd('__intarray:w', function()
22029    scan_int()
```

```

22030   tex.error'LaTeX Error: Isolated intarray ignored'
22031 end, 'protected', 'global')
22032 
```

(End definition for `\_intarray:w`.)

`\intarray_new:Nn` Declare #1 as a tokenlist with the scanmark and a unique number. Pass the array's size to the Lua helper. Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```

22033 <tex>
22034   \cs_new_protected:Npn \_intarray_new:N #1
22035   {
22036     \_kernel_chk_if_free_cs:N #1
22037     \int_gincr:N \g__intarray_table_int
22038     \cs_gset_nopar:Npx #1 { \_intarray:w \int_use:N \g__intarray_table_int \c_space_tl
22039   }
22040   \cs_new_protected:Npn \intarray_new:Nn #1#2
22041   {
22042     \_intarray_new:N #1
22043     \_intarray_gset_count:Nw #1 \int_eval:n {#2} \scan_stop:
22044     \int_compare:nNnT { \intarray_count:N #1 } < 0
22045     {
22046       \msg_error:nnx { kernel } { negative-array-size }
22047       { \intarray_count:N #1 }
22048     }
22049   }
22050   \cs_generate_variant:Nn \intarray_new:Nn { c }
22051 </tex>
```

(End definition for `\intarray_new:Nn` and `\_intarray_new:N`. This function is documented on page 237.)

Before we get to the first command implemented in Lua, we first need some definitions. Since `token.create` only works correctly if TeX has seen the tokens before, we first run a short TeX sequence to ensure that all relevant control sequences are known.

```

22052 <*lua>
22053
22054 local scan_token = token.scan_token
22055 local put_next = token.put_next
22056 local intarray_marker = token_create_safe'\_intarray:w'
22057 local use_none = token_create_safe'use:none:n'
22058 local use_i = token_create_safe'use:n'
22059 local expand_after_scan_stop = {token_create_safe'exp_after:wN',
22060                           token_create_safe'scan_stop:'}
22061 local comma = token_create(string.byte,',')
```

`\_intarray_table` Internal helper to scan an intarray token, extract the associated Lua table and return an error if the input is invalid.

```

22062 local __intarray_table do
22063   local tables = get_luadata and get_luadata'\_intarray' or {[0] = {}}
22064   function __intarray_table()
22065     local t = scan_token()
22066     if t ~= intarray_marker then
22067       put_next(t)
22068       tex.error'LaTeX Error: intarray expected'
```

```

22069     return tables[0]
22070   end
22071   local i = scan_int()
22072   local current_table = tables[i]
22073   if current_table then return current_table end
22074   current_table = {}
22075   tables[i] = current_table
22076   return current_table
22077 end

```

Since in L<sup>A</sup>T<sub>E</sub>X this is loaded in the format, we want to preserve any intarrays which are created while format building for the actual run.

To do this, we use the `register_luadata` mechanism from l3luatex: Directly before the format get dumped, the following function gets invoked and serializes all existing tables into a string. This string gets compiled and dumped into the format and is made available at the beginning of regular runs as `get_luadata'@@'`.

```

22078   if register_luadata then
22079     register_luadata('__intarray', function()
22080       local t = "[{}]={},"
22081       for i=1, #tables do
22082         t = string.format("%s[%s],", t, table.concat(tables[i], ','))
22083       end
22084       return t .. "}"
22085     end)
22086   end
22087 end

```

(End definition for `__intarray_table`.)

`\intarray_count:N` Set and get the size of an array. “Setting the size” means in this context that we add zeros until we reach the desired size.  
`\intarray_count:c`

```

\intarray_gset_count:Nw
22088
22089 local sprint = tex.sprint
22090
22091 luacmd('__intarray_gset_count:Nw', function()
22092   local t = __intarray_table()
22093   local n = scan_int()
22094   for i=#t+1, n do t[i] = 0 end
22095 end, 'protected', 'global')
22096
22097 luacmd('intarray_count:N', function()
22098   sprint(-2, #__intarray_table())
22099 end, 'global')
22100 </lua>
22101 <*tex>
22102   \cs_generate_variant:Nn \intarray_count:N { c }
22103 </tex>

```

(End definition for `\intarray_count:N` and `\intarray_gset_count:Nw`. This function is documented on page 237.)

### 64.1.2 Array items

\\_\\_intarray\\_gset:wF  
\\_\\_intarray\\_gset:w

The setter provided by Lua. The argument order somewhat emulates the \fontdimen: First the array index, followed by the intarray and then the new value. This has been chosen over a more conventional order to provide a delimiter for the numbers.

```

22104  <*lua>
22105  luacmd('_\_intarray_gset:wF', function()
22106    local i = scan_int()
22107    local t = __intarray_table()
22108    if t[i] then
22109      t[i] = scan_int()
22110      put_next(use_none)
22111    else
22112      tex.count.l_\_intarray_bad_index_int = i
22113      scan_int()
22114      put_next(use_i)
22115    end
22116  end, 'protected', 'global')
22117
22118  luacmd('_\_intarray_gset:w', function()
22119    local i = scan_int()
22120    local t = __intarray_table()
22121    t[i] = scan_int()
22122  end, 'protected', 'global')
22123 </lua>
```

(End definition for \\_\\_intarray\\_gset:wF and \\_\\_intarray\\_gset:w.)

\intarray\_gset:Nnn  
\intarray\_gset:cnn  
\\_\\_kernel\\_intarray\\_gset:Nnn

The \\_\\_kernel\\_intarray\\_gset:Nnn function does not use \int\_eval:n, namely its arguments must be suitable for \int\_value:w. The user version checks the position and value are within bounds.

```

22124  <*tex>
22125    \cs_new_protected:Npn \_\_kernel_intarray_gset:Nnn #1#2#3
22126    {
22127      \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
22128      {
22129        \_\_intarray_gset:wF \int_eval:n {#2} #1 \int_eval:n{#3}
22130        {
22131          \msg_error:nnnn { kernel } { out-of-bounds }
22132          { \token_to_str:N #1 } { \int_use:N \l_\_intarray_bad_index_int } { \intarray_
22133        }
22134      }
22135      \cs_generate_variant:Nn \intarray_gset:Nnn { c }
22136    </tex>
```

(End definition for \intarray\_gset:Nnn and \\_\\_kernel\\_intarray\\_gset:Nnn. This function is documented on page 237.)

\intarray\_gzero:N Set the appropriate array entry to zero. No bound checking needed.

\intarray\_gzero:c

```

22137  <*lua>
22138  luacmd('intarray_gzero:N', function()
22139    local t = __intarray_table()
22140    for i=1, #t do
22141      t[i] = 0
```

```

22142     end
22143 end, 'global', 'protected')
22144 ⟨/lua⟩
22145 ⟨*tex⟩
22146   \cs_generate_variant:Nn \intarray_gzero:N { c }
22147 ⟨/tex⟩

```

(End definition for `\intarray_gzero:N`. This function is documented on page 238.)

`\intarray_item:Nn`  
`\intarray_item:cn`  
`\_kernel_intarray_item:Nn`  
`\_intarray_item:wF`  
`\_intarray_item:w`

Get the appropriate entry and perform bound checks. The `\_kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```

22148 ⟨*lua⟩
22149 luacmd('\_intarray_item:wF', function()
22150   local i = scan_int()
22151   local t = __intarray_table()
22152   local item = t[i]
22153   if item then
22154     put_next(use_none)
22155   else
22156     tex.l_intarray_bad_index_int = i
22157     put_next(use_i)
22158   end
22159   put_next(expand_after_scan_stop)
22160   scan_token()
22161   if item then
22162     sprint(-2, item)
22163   end
22164 end, 'global')
22165
22166 luacmd('\_intarray_item:w', function()
22167   local i = scan_int()
22168   local t = __intarray_table()
22169   sprint(-2, t[i])
22170 end, 'global')
22171 ⟨/lua⟩
22172 ⟨*tex⟩
22173   \cs_new:Npn \_kernel_intarray_item:Nn #1#2
22174   { \_intarray_item:w #2 #1 }
22175 \cs_new:Npn \intarray_item:Nn #1#2
22176 {
22177   \_intarray_item:wF \int_eval:n {#2} #1
22178   {
22179     \msg_expandable_error:nnffff { kernel } { out-of-bounds }
22180     { \token_to_str:N #1 } { \int_use:N \l_intarray_bad_index_int } { \intarray_
22181     0
22182   }
22183 }
22184 \cs_generate_variant:Nn \intarray_item:Nn { c }

```

(End definition for `\intarray_item:Nn` and others. This function is documented on page 238.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.  
`\intarray_rand_item:c`

```

22185   \cs_new:Npn \intarray_rand_item:N #1

```

```

22186   { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
22187   \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End definition for `\intarray_rand_item:N`. This function is documented on page 238.)

#### 64.1.3 Working with contents of integer arrays

We use the `\__kernel_intarray_gset:Nnn` which does not do bounds checking and instead automatically resizes the array. This is not implemented in Lua to ensure that theclist parsing is consistent with theclist module.

```

22188   \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
22189   {
22190     \__intarray_new:N #1
22191     \int_zero:N \l__intarray_loop_int
22192     \clist_map_inline:nn {#2}
22193     {
22194       \int_incr:N \l__intarray_loop_int
22195       \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int { \int_eval:n {##1} } }
22196     }
22197   \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }

```

(End definition for `\intarray_const_from_clist:Nn`. This function is documented on page 238.)

`\intarray_to_clist:N`

`\intarray_to_clist:c`

`\__intarray_to_clist:Nn`

`\__intarray_to_clist:w`

The `\__intarray_to_clist:Nn` auxiliary allows to choose the delimiter and is also used in `\intarray_show:N`. Here we just pass the information to Lua and let `table.concat` do the actual work. We discard the category codes of the passed delimiter but this is not an issue since the delimiter is always just a comma or a comma and a space. In both cases `sprint(2, ...)` provides the right catcodes.

```

22198   \cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
22199   \cs_generate_variant:Nn \intarray_to_clist:N { c }
22200   </tex>
22201   <*lua>
22202   local concat = table.concat
22203   luacmd(' \__intarray_to_clist:Nn', function()
22204     local t = __intarray_table()
22205     local sep = token.scan_string()
22206     sprint(-2, concat(t, sep))
22207   end, 'global')
22208   </lua>

```

(End definition for `\intarray_to_clist:N`, `\__intarray_to_clist:Nn`, and `\__intarray_to_clist:w`. This function is documented on page 307.)

Loop through part of the array.

```

22209   <*tex>
22210   \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
22211   {
22212     \__intarray_range_to_clist:w #1
22213     \int_eval:n {#2} ~ \int_eval:n {#3} ~
22214   }
22215   </tex>
22216   <*lua>
22217   luacmd(' \__intarray_range_to_clist:w', function()
22218     local t = __intarray_table()

```

```

22219 local from = scan_int()
22220 local to = scan_int()
22221 sprint(-2, concat(t, ',', from, to))
22222 end, 'global')
22223 
```

(End definition for `\_kernel_intarray_range_to_clist:Nnn` and `\_intarray_range_to_clist:w`.)

`\_kernel_intarray_gset_range_from clist:Nnn`  
`\_intarray_gset_range:nNw`

```

22224 (*tex)
22225   \cs_new_protected:Npn \_kernel_intarray_gset_range_from_clist:Nnn #1#2#3
22226   {
22227     \_intarray_gset_range:w \int_eval:w #2 #1 #3 , , \scan_stop:
22228   }
22229 
```

`\_intarray_gset_range:w`, function()

```

22230 (*lua)
22231 luacmd('\_intarray_gset_range:w', function()
22232   local from = scan_int()
22233   local t = \_intarray_table()
22234   while true do
22235     local tok = scan_token()
22236     if tok == comma then
22237       repeat
22238         tok = scan_token()
22239       until tok ~= comma
22240       break
22241     else
22242       put_next(tok)
22243     end
22244     t[from] = scan_int()
22245     scan_token()
22246     from = from + 1
22247   end
22248 end, 'global', 'protected')
22249 
```

(End definition for `\_kernel_intarray_gset_range_from_clist:Nnn` and `\_intarray_gset_range:nNw`.)

`\_intarray_gset_overflow_test:nw`

In order to allow some code sharing later we provide the `\_intarray_gset_overflow_test:nw` name here. It doesn't actually test anything since the Lua implementation accepts all integers which could be tested with `\tex_ifabsnum:D`.

```

22250 (*tex)
22251   \cs_new_protected:Npn \_intarray_gset_overflow_test:nw #1
22252   {
22253   }
```

(End definition for `\_intarray_gset_overflow_test:nw`.)

## 64.2 Font dimension based implementation

Go to the false branch of the conditional above.

```

22254   }
22255   {
```

### 64.2.1 Allocating arrays

<code>\__intarray_entry:w</code>	We use these primitives quite a lot in this module.
<code>\__intarray_count:w</code>	<pre>22256      \cs_new_eq:NN \__intarray_entry:w \tex_fontdimen:D 22257      \cs_new_eq:NN \__intarray_count:w \tex_hyphenchar:D</pre> <p>(End definition for <code>\__intarray_entry:w</code> and <code>\__intarray_count:w</code>.)</p>
<code>\c_intarray_sp_dim</code>	<p>Used to convert integers to dimensions fast.</p> <pre>22258      \dim_const:Nn \c_intarray_sp_dim { 1 sp }</pre> <p>(End definition for <code>\c_intarray_sp_dim</code>.)</p>
<code>\g_intarray_font_int</code>	<p>Used to assign one font per array.</p> <pre>22259      \int_new:N \g_intarray_font_int</pre> <p>(End definition for <code>\g_intarray_font_int</code>.)</p>
<code>\intarray_new:Nn</code> <code>\intarray_new:cn</code> <code>\__intarray_new:N</code>	<p>Declare #1 to be a font (arbitrarily <code>cmr10</code> at a never-used size). Store the array's size as the <code>\hyphenchar</code> of that font and make sure enough <code>\fontdimen</code> are allocated, by setting the last one. Then clear any <code>\fontdimen</code> that <code>cmr10</code> starts with. It seems LuaTeX's <code>cmr10</code> has an extra <code>\fontdimen</code> parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every <code>intarray</code> must be global; it's enough to run this check in <code>\intarray_new:Nn</code>.</p> <pre>22260      \cs_new_protected:Npn \__intarray_new:N #1 22261      { 22262          \__kernel_chk_if_free_cs:N #1 22263          \int_gincr:N \g_intarray_font_int 22264          \tex_global:D \tex_font:D #1 22265          = cmr10-at~ \g_intarray_font_int \c_intarray_sp_dim \scan_stop: 22266          \int_step_inline:nn { 8 } 22267          { \__kernel_intarray_gset:Nnn #1 {##1} \c_zero_int } 22268      } 22269      \cs_new_protected:Npn \intarray_new:Nn #1#2 22270      { 22271          \__intarray_new:N #1 22272          \__intarray_count:w #1 = \int_eval:n {#2} \scan_stop: 22273          \int_compare:nNnT { \intarray_count:N #1 } &lt; 0 22274          { 22275              \msg_error:nnx { kernel } { negative-array-size } 22276              { \intarray_count:N #1 } 22277          } 22278          \int_compare:nNnT { \intarray_count:N #1 } &gt; 0 22279          { \__kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } } 22280      } 22281      \cs_generate_variant:Nn \intarray_new:Nn { c }</pre> <p>(End definition for <code>\intarray_new:Nn</code> and <code>\__intarray_new:N</code>. This function is documented on page 237.)</p>
<code>\intarray_count:N</code>	Size of an array.
<code>\intarray_count:c</code>	<pre>22282      \cs_new:Npn \intarray_count:N #1 { \int_value:w \__intarray_count:w #1 } 22283      \cs_generate_variant:Nn \intarray_count:N { c }</pre> <p>(End definition for <code>\intarray_count:N</code>. This function is documented on page 237.)</p>

### 64.2.2 Array items

`\_\_intarray_signed\_max\_dim:n`

Used when an item to be stored is larger than `\c_max_dim` in absolute value; it is replaced by  $\pm \c_max_dim$ .

```
22284      \cs_new:Npn \_\_intarray_signed_max_dim:n #1
22285          { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(End definition for `\_\_intarray_signed_max_dim:n`.)

`\_\_intarray_bounds:NNnTF`  
`\_\_intarray_bounds_error>NNnw`

The functions `\intarray_gset:Nnn` and `\intarray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```
22286      \cs_new:Npn \_\_intarray_bounds:NNnTF #1#2#3
22287          {
22288              \if_int_compare:w 1 > #3 \exp_stop_f:
22289                  \_\_intarray_bounds_error>NNnw #1 #2 {#3}
22290              \else:
22291                  \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
22292                      \_\_intarray_bounds_error>NNnw #1 #2 {#3}
22293                  \fi:
22294                  \fi:
22295                  \use_i:nn
22296          }
22297      \cs_new:Npn \_\_intarray_bounds_error>NNnw #1#2#3#4 \use_i:nn #5#6
22298          {
22299              #4
22300              #1 { kernel } { out-of-bounds }
22301                  { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
22302              #6
22303          }
```

(End definition for `\_\_intarray_bounds:NNnTF` and `\_\_intarray_bounds_error>NNnw`.)

`\intarray_gset:Nnn`  
`\intarray_gset:cnn`  
`\_\_kernel_intarray_gset:Nnn`

Set the appropriate `\fontdimen`. The `\_\_kernel_intarray_gset:Nnn` function does not use `\int_eval:n`, namely its arguments must be suitable for `\int_value:w`. The user version checks the position and value are within bounds.

```
22304      \cs_new_protected:Npn \_\_kernel_intarray_gset:Nnn #1#2#3
22305          { \_\_intarray_entry:w #2 #1 #3 \c_intarray_sp_dim }
22306      \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
22307          {
22308              \exp_after:wN \_\_intarray_gset:Nww
22309              \exp_after:wN #1
22310              \int_value:w \int_eval:n {#2} \exp_after:wN ;
22311              \int_value:w \int_eval:n {#3} ;
22312          }
22313      \cs_generate_variant:Nn \intarray_gset:Nnn { c }
22314      \cs_new_protected:Npn \_\_intarray_gset:Nww #1#2 ; #3 ;
22315          {
22316              \_\_intarray_bounds:NNnTF \msg_error:nnnn #1 {#2}
22317                  {
22318                      \_\_intarray_gset_overflow_test:nw {#3}
22319                      \_\_kernel_intarray_gset:Nnn #1 {#2} {#3}
22320                  }
22321          }
22322      }
```

```

22323   \cs_if_exist:NTF \tex_ifabsnum:D
22324   {
22325     \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
22326     {
22327       \tex_ifabsnum:D #1 > \c_max_dim
22328         \exp_after:wN \__intarray_gset_overflow>NNnn
22329       \fi:
22330     }
22331   }
22332   {
22333     \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
22334     {
22335       \if_int_compare:w \int_abs:n {#1} > \c_max_dim
22336         \exp_after:wN \__intarray_gset_overflow>NNnn
22337       \fi:
22338     }
22339   }
22340   \cs_new_protected:Npn \__intarray_gset_overflow>NNnn #1#2#3#4
22341   {
22342     \msg_error:nnxxxx { kernel } { overflow }
22343     { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
22344     #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
22345   }

```

(End definition for `\intarray_gset:Nnn` and others. This function is documented on page 237.)

`\intarray_gzero:N` Set the appropriate `\fontdimen` to zero. No bound checking needed. The `\prg_replicate:nn` possibly uses quite a lot of memory, but this is somewhat comparable to the size of the array, and it is much faster than an `\int_step_inline:nn` loop.

```

22346   \cs_new_protected:Npn \intarray_gzero:N #1
22347   {
22348     \int_zero:N \l__intarray_loop_int
22349     \prg_replicate:nn { \intarray_count:N #1 }
22350     {
22351       \int_incr:N \l__intarray_loop_int
22352         \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
22353     }
22354   }
22355   \cs_generate_variant:Nn \intarray_gzero:N { c }

```

(End definition for `\intarray_gzero:N`. This function is documented on page 238.)

`\intarray_item:Nn`  
`\intarray_item:cn`  
`\__kernel_intarray_item:Nn`  
`\__intarray_item:Nn` Get the appropriate `\fontdimen` and perform bound checks. The `\__kernel_intarray_item:Nn` function omits bound checks and omits `\int_eval:n`, namely its argument must be a TeX integer suitable for `\int_value:w`.

```

22356   \cs_new:Npn \__kernel_intarray_item:Nn #1#2
22357   { \int_value:w \__intarray_entry:w #2 #1 }
22358   \cs_new:Npn \intarray_item:Nn #1#2
22359   {
22360     \exp_after:wN \__intarray_item:Nw
22361     \exp_after:wN #1
22362     \int_value:w \int_eval:n {#2} ;
22363   }
22364   \cs_generate_variant:Nn \intarray_item:Nn { c }

```

```

22365   \cs_new:Npn \__intarray_item:Nw #1#2 ;
22366   {
22367     \__intarray_bounds:NNnTF \msg_expandable_error:nffff #1 {#2}
22368     { \__kernel_intarray_item:Nn #1 {#2} }
22369     { 0 }
22370   }

```

(End definition for `\intarray_item:Nn`, `\__kernel_intarray_item:Nn`, and `\__intarray_item:Nn`. This function is documented on page 238.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```

\intarray_rand_item:c
22371   \cs_new:Npn \intarray_rand_item:N #1
22372     { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
22373   \cs_generate_variant:Nn \intarray_rand_item:N { c }

```

(End definition for `\intarray_rand_item:N`. This function is documented on page 238.)

#### 64.2.3 Working with contents of integer arrays

`\intarray_const_from_clist:Nn`  
`\intarray_const_from_clist:cn`  
`\__intarray_const_from_clist:nN`

Similar to `\intarray_new:Nn` (which we don't use because when debugging is enabled that function checks the variable name starts with `g_`). We make use of the fact that TeX allows allocation of successive `\fontdimen` as long as no other font has been declared: no need to count the comma list items first. We need the code in `\intarray_gset:Nnn` that checks the item value is not too big, namely `\__intarray_gset_overflow_test:nw`, but not the code that checks bounds. At the end, set the size of the intarray.

```

22374   \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
22375   {
22376     \__intarray_new:N #1
22377     \int_zero:N \l__intarray_loop_int
22378     \clist_map_inline:nn {#2}
22379       { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
22380     \__intarray_count:w #1 \l__intarray_loop_int
22381   }
22382   \cs_generate_variant:Nn \intarray_const_from_clist:Nn { c }
22383   \cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
22384   {
22385     \int_incr:N \l__intarray_loop_int
22386     \__intarray_gset_overflow_test:nw {#1}
22387     \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
22388   }

```

(End definition for `\intarray_const_from_clist:Nn` and `\__intarray_const_from_clist:nN`. This function is documented on page 238.)

`\intarray_to_clist:N`  
`\intarray_to_clist:c`  
`\__intarray_to_clist:Nn`  
`\__intarray_to_clist:w`

Loop through the array, putting a comma before each item. Remove the leading comma with f-expansion. We also use the auxiliary in `\intarray_show:N` with argument comma, space.

```

22389   \cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
22390   \cs_generate_variant:Nn \intarray_to_clist:N { c }
22391   \cs_new:Npn \__intarray_to_clist:Nn #1#2
22392   {
22393     \int_compare:nNnF { \intarray_count:N #1 } = \c_zero_int
22394     {
22395       \exp_last_unbraced:Nf \use_none:n

```

```

22396      { \__intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
22397    }
22398  }
22399 \cs_new:Npn \__intarray_to_clist:w #1 ; #2#3
22400  {
22401    \if_int_compare:w #1 > \__intarray_count:w #2
22402      \prg_break:n
22403    \fi:
22404    #3 \__kernel_intarray_item:Nn #2 {#1}
22405    \exp_after:wN \__intarray_to_clist:w
22406    \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
22407  }

```

(End definition for `\intarray_to_clist:N`, `\__intarray_to_clist:Nn`, and `\__intarray_to_clist:w`.  
This function is documented on page 307.)

Loop through part of the array.

```

22408 \cs_new:Npn \__kernel_intarray_range_to_clist:Nnn #1#2#3
22409  {
22410    \exp_last_unbraced:Nf \use_none:n
22411    {
22412      \exp_after:wN \__intarray_range_to_clist:ww
22413      \int_value:w \int_eval:w #2 \exp_after:wN ;
22414      \int_value:w \int_eval:w #3 ;
22415      #1 \prg_break_point:
22416    }
22417  }
22418 \cs_new:Npn \__intarray_range_to_clist:ww #1 ; #2 ; #3
22419  {
22420    \if_int_compare:w #1 > #2 \exp_stop_f:
22421      \prg_break:n
22422    \fi:
22423    , \__kernel_intarray_item:Nn #3 {#1}
22424    \exp_after:wN \__intarray_range_to_clist:ww
22425    \int_value:w \int_eval:w #1 + \c_one_int ; #2 ; #3
22426  }

```

(End definition for `\__kernel_intarray_range_to_clist:Nnn` and `\__intarray_range_to_clist:ww`.)

Loop through part of the array.

```

22427 \cs_new_protected:Npn \__kernel_intarray_gset_range_from_clist:Nnn #1#2#3
22428  {
22429    \int_set:Nn \l__intarray_loop_int {#2}
22430    \__intarray_gset_range:Nw #1 #3 , , \prg_break_point:
22431  }
22432 \cs_new_protected:Npn \__intarray_gset_range:Nw #1 #2 ,
22433  {
22434    \if_catcode:w \scan_stop: \tl_to_str:n {#2} \scan_stop:
22435      \prg_break:n
22436    \fi:
22437    \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
22438    \int_incr:N \l__intarray_loop_int
22439    \__intarray_gset_range:Nw #1
22440  }

```

```
(End definition for \_kernel_intarray_gset_range_from_clist:Nnn and \_intarray_gset_range:Nw.)  
22441 }
```

## 64.3 Common parts

```
\intarray_show:N Convert the list to a comma list (with spaces after each comma)
\intarray_show:c 22442 \cs_new_protected:Npn \intarray_show:N { \_intarray_show:NN \msg_show:nxxxxx }
\intarray_log:N 22443 \cs_generate_variant:Nn \intarray_show:N { c }
\intarray_log:c 22444 \cs_new_protected:Npn \intarray_log:N { \_intarray_show:NN \msg_log:nxxxxx }
22445 \cs_generate_variant:Nn \intarray_log:N { c }
22446 \cs_new_protected:Npn \_intarray_show:NN #1#2
22447 {
22448   \_kernel_chk_defined:NT #2
22449 {
22450   #1 { intarray } { show }
22451   { \token_to_str:N #2 }
22452   { \intarray_count:N #2 }
22453   { >~ \_intarray_to_clist:Nn #2 { , ~ } }
22454   { }
22455 }
22456 }
```

(End definition for `\intarray_show:N` and `\intarray_log:N`. These functions are documented on page 238.)

### 64.3.1 Random arrays

We only perform the bounds checks once. This is done by two `\_intarray_gset_-overflow_test:nw`, with an appropriate empty argument to avoid a spurious “at position #1” part in the error message. Then calculate the number of choices: this is at most  $(2^{30} - 1) - (-(2^{30} - 1)) + 1 = 2^{31} - 1$ , which just barely does not overflow. For small ranges use `\_kernel_randint:n` (making sure to subtract 1 *before* adding the random number to the `<min>`, to avoid overflow when `<min>` or `<max>` are  $\pm \c_max_int$ ), otherwise `\_kernel_randint:nn`. Finally, if there are no random numbers do not define any of the auxiliaries.

```
22457 \cs_new_protected:Npn \intarray_gset_rand:Nn #1
22458   { \intarray_gset_rand:Nnn #1 { 1 } }
22459 \cs_generate_variant:Nn \intarray_gset_rand:Nn { c }
22460 \sys_if_rand_exist:TF
22461 {
22462   \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
22463   {
22464     \_intarray_gset_rand:Nff #1
22465     { \int_eval:n {#2} } { \int_eval:n {#3} }
22466   }
22467 \cs_new_protected:Npn \_intarray_gset_rand:Nnn #1#2#3
22468   {
22469     \int_compare:nNnTF {#2} > {#3}
22470     {
22471       \msg_expandable_error:nnnn
22472       { kernel } { randint-backward-range } {#2} {#3}
22473     \_intarray_gset_rand:Nnn #1 {#3} {#2}
```

```

22474     }
22475     {
22476         \__intarray_gset_overflow_test:nw {#2}
22477         \__intarray_gset_rand_auxi:Nnnn #1 { } {#2} {#3}
22478     }
22479 }
22480 \cs_generate_variant:Nn \__intarray_gset_rand:Nnn { Nff }
22481 \cs_new_protected:Npn \__intarray_gset_rand_auxi:Nnnn #1#2#3#4
22482 {
22483     \__intarray_gset_overflow_test:nw {#4}
22484     \__intarray_gset_rand_auxii:Nnnn #1 { } {#4} {#3}
22485 }
22486 \cs_new_protected:Npn \__intarray_gset_rand_auxii:Nnnn #1#2#3#4
22487 {
22488     \exp_args:NNf \__intarray_gset_rand_auxiii:Nnnn #1
22489     { \int_eval:n { #3 - #4 + 1 } } {#4} {#3}
22490 }
22491 \cs_new_protected:Npn \__intarray_gset_rand_auxiii:Nnnn #1#2#3#4
22492 {
22493     \exp_args:NNf \__intarray_gset_all_same:Nn #1
22494     {
22495         \int_compare:nNnTF {#2} > \c_kernel_randint_max_int
22496         {
22497             \exp_stop_f:
22498             \int_eval:n { \__kernel_randint:nn {#3} {#4} }
22499         }
22500     {
22501         \exp_stop_f:
22502         \int_eval:n { \__kernel_randint:n {#2} - 1 + #3 }
22503     }
22504 }
22505 }
22506 \cs_new_protected:Npn \__intarray_gset_all_same:Nn #1#2
22507 {
22508     \int_zero:N \l__intarray_loop_int
22509     \prg_replicate:nn { \intarray_count:N #1 }
22510     {
22511         \int_incr:N \l__intarray_loop_int
22512         \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
22513     }
22514 }
22515 }
22516 {
22517     \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
22518     {
22519         \msg_error:nnn { kernel } { fp-no-random }
22520         { \intarray_gset_rand:Nnn #1 {#2} {#3} }
22521     }
22522 }
22523 \cs_generate_variant:Nn \intarray_gset_rand:Nnn { c }

(End definition for \intarray_gset_rand:Nn and others. These functions are documented on page 307.)
22524 </tex>
22525 </package>

```

## Chapter 65

# **l3fp implementation**

Nothing to see here: everything is in the subfiles!

# Chapter 66

## I3fp-aux implementation

```
22526  <*package>
22527  <@@=fp>
```

### 66.1 Access to primitives

`\_fp_int_eval:w` Largely for performance reasons, we need to directly access primitives rather than use `\int_eval:n`. This happens *a lot*, so we use private names. The same is true for `\romannumeral`, although it is used much less widely.  
`\_fp_int_eval_end:`  
`\_fp_int_to_roman:w`

```
22528 \cs_new_eq:NN \_fp_int_eval:w \tex_numexpr:D
22529 \cs_new_eq:NN \_fp_int_eval_end: \scan_stop:
22530 \cs_new_eq:NN \_fp_int_to_roman:w \tex_romannumeral:D
```

(End definition for `\_fp_int_eval:w`, `\_fp_int_eval_end:`, and `\_fp_int_to_roman:w`.)

### 66.2 Internal representation

Internally, a floating point number  $\langle X \rangle$  is a token list containing

```
\s_fp \_fp_chk:w <case> <sign> <body> ;
```

Let us explain each piece separately.

Internal floating point numbers are used in expressions, and in this context are subject to f-expansion. They must leave a recognizable mark after f-expansion, to prevent the floating point number from being re-parsed. Thus, `\s_fp` is simply another name for `\relax`.

When used directly without an accessor function, floating points should produce an error: this is the role of `\_fp_chk:w`. We could make floating point variables be protected to prevent them from expanding under x-expansion, but it seems more convenient to treat them as a subcase of token list variables.

The (decimal part of the) IEEE-754-2008 standard requires the format to be able to represent special floating point numbers besides the usual positive and negative cases. We distinguish the various possibilities by their  $\langle case \rangle$ , which is a single digit:

- 0 zeros: +0 and -0,
- 1 “normal” numbers (positive and negative),

Table 3: Internal representation of floating point numbers.

Representation	Meaning
0 0 \s_fp_... ;	Positive zero.
0 2 \s_fp_... ;	Negative zero.
1 0 {\langle exponent \rangle} {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} ;	Positive floating point.
1 2 {\langle exponent \rangle} {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} ;	Negative floating point.
2 0 \s_fp_... ;	Positive infinity.
2 2 \s_fp_... ;	Negative infinity.
3 1 \s_fp_... ;	Quiet nan.
3 1 \s_fp_... ;	Signalling nan.

2 infinities: `+inf` and `-inf`,

3 quiet and signalling `nan`.

The  $\langle sign \rangle$  is 0 (positive) or 2 (negative), except in the case of `nan`, which have  $\langle sign \rangle = 1$ . This ensures that changing the  $\langle sign \rangle$  digit to  $2 - \langle sign \rangle$  is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

`\s_fp \_fp_chk:w \langle case \rangle \langle sign \rangle \s_fp_... ;`

where `\s_fp_...` is a scan mark carrying information about how the number was formed (useful for debugging).

Normal floating point numbers ( $\langle case \rangle = 1$ ) have the form

`\s_fp \_fp_chk:w 1 \langle sign \rangle {\langle exponent \rangle} {\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle} ;`

Here, the  $\langle exponent \rangle$  is an integer, between  $-10000$  and  $10000$ . The body consists in four blocks of exactly 4 digits,  $0000 \leq \langle X_i \rangle \leq 9999$ , and the floating point is

$$(-1)^{\langle sign \rangle / 2} \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle \cdot 10^{\langle exponent \rangle - 16}$$

where we have concatenated the 16 digits. Currently, floating point numbers are normalized such that the  $\langle exponent \rangle$  is minimal, in other words,  $1000 \leq \langle X_1 \rangle \leq 9999$ .

Calculations are done in base 10000, *i.e.* one myriad.

### 66.3 Using arguments and semicolons

`\_fp_use_none_stop_f:n` This function removes an argument (typically a digit) and replaces it by `\exp_stop_f:`, a marker which stops f-type expansion.

22531 `\cs_new:Npn \_fp_use_none_stop_f:n #1 { \exp_stop_f: }`

(End definition for `\_fp_use_none_stop_f:n`.)

`\_fp_use_s:n` Those functions place a semicolon after one or two arguments (typically digits).

22532 `\cs_new:Npn \_fp_use_s:n #1 { #1; }`

22533 `\cs_new:Npn \_fp_use_s:nn #1#2 { #1#2; }`

(End definition for `\_fp_use_s:n` and `\_fp_use_s:nn`.)

`\__fp_use_none_until_s:w`  
`\__fp_use_i_until_s:nw`  
`\__fp_use_ii_until_s:nnw`

Those functions select specific arguments among a set of arguments delimited by a semi-colon.

```
22534 \cs_new:Npn \__fp_use_none_until_s:w #1; { }
22535 \cs_new:Npn \__fp_use_i_until_s:nw #1#2; {#1}
22536 \cs_new:Npn \__fp_use_ii_until_s:nnw #1#2#3; {#2}
```

(*End definition for \\_\_fp\_use\_none\_until\_s:w, \\_\_fp\_use\_i\_until\_s:nw, and \\_\_fp\_use\_ii\_until\_s:nnw.*)

`\__fp_reverse_args:Nww`

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.

```
22537 \cs_new:Npn \__fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }
```

(*End definition for \\_\_fp\_reverse\_args:Nww.*)

`\__fp_rrot:www`

Rotate three arguments delimited by semicolons. This is the inverse (or the square) of the Forth primitive ROT, hence the name.

```
22538 \cs_new:Npn \__fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(*End definition for \\_\_fp\_rrot:www.*)

`\__fp_use_i:ww`  
`\__fp_use_i:www`

Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
22539 \cs_new:Npn \__fp_use_i:ww #1; #2; { #1; }
22540 \cs_new:Npn \__fp_use_i:www #1; #2; #3; { #1; }
```

(*End definition for \\_\_fp\_use\_i:ww and \\_\_fp\_use\_i:www.*)

## 66.4 Constants, and structure of floating points

`\__fp_misused:n`

This receives a floating point object (floating point number or tuple) and generates an error stating that it was misused. This is called when for instance an `fp` variable is left in the input stream and its contents reach TeX's stomach.

```
22541 \cs_new_protected:Npn \__fp_misused:n #1
22542   { \msg_error:nnx { fp } { misused } { \fp_to_tl:n {#1} } }
```

(*End definition for \\_\_fp\_misused:n.*)

`\s_fp`  
`\__fp_chk:w`

Floating points numbers all start with `\s_fp \__fp_chk:w`, where `\s_fp` is equal to the TeX primitive `\relax`, and `\__fp_chk:w` is protected. The rest of the floating point number is made of characters (or `\relax`). This ensures that nothing expands under `f`-expansion, nor under `x`-expansion. However, when typeset, `\s_fp` does nothing, and `\__fp_chk:w` is expanded. We define `\__fp_chk:w` to produce an error.

```
22543 \scan_new:N \s_fp
22544 \cs_new_protected:Npn \__fp_chk:w #1 ;
22545   { \__fp_misused:n { \s_fp \__fp_chk:w #1 ; } }
```

(*End definition for \s\_fp and \\_\_fp\_chk:w.*)

`\s_fp_expr_mark`

Aliases of `\tex_relax:D`, used to terminate expressions.

```
22546 \scan_new:N \s_fp_expr_mark
22547 \scan_new:N \s_fp_expr_stop
```

(*End definition for \s\_fp\_expr\_mark and \s\_fp\_expr\_stop.*)

\s\_fp\_mark Generic scan marks used throughout the module.  
 \s\_fp\_stop  
 22548 \scan\_new:N \s\_fp\_mark  
 22549 \scan\_new:N \s\_fp\_stop

(End definition for \s\_fp\_mark and \s\_fp\_stop.)

\\_fp\_use\_i\_delimit\_by\_s\_stop:nw Functions to gobble up to a scan mark.  
 22550 \cs\_new:Npn \\_fp\_use\_i\_delimit\_by\_s\_stop:nw #1 #2 \s\_fp\_stop {#1}

(End definition for \\_fp\_use\_i\_delimit\_by\_s\_stop:nw.)

\s\_fp\_invalid \s\_fp\_underflow \s\_fp\_overflow \s\_fp\_division \s\_fp\_exact A couple of scan marks used to indicate where special floating point numbers come from.  
 22551 \scan\_new:N \s\_fp\_invalid  
 22552 \scan\_new:N \s\_fp\_underflow  
 22553 \scan\_new:N \s\_fp\_overflow  
 22554 \scan\_new:N \s\_fp\_division  
 22555 \scan\_new:N \s\_fp\_exact

(End definition for \s\_fp\_invalid and others.)

\c\_zero\_fp \c\_minus\_zero\_fp \c\_inf\_fp \c\_minus\_inf\_fp \c\_nan\_fp The special floating points. We define the floating points here as “exact”.  
 22556 \tl\_const:Nn \c\_zero\_fp { \s\_fp \\_fp\_chk:w 0 0 \s\_fp\_exact ; }  
 22557 \tl\_const:Nn \c\_minus\_zero\_fp { \s\_fp \\_fp\_chk:w 0 2 \s\_fp\_exact ; }  
 22558 \tl\_const:Nn \c\_inf\_fp { \s\_fp \\_fp\_chk:w 2 0 \s\_fp\_exact ; }  
 22559 \tl\_const:Nn \c\_minus\_inf\_fp { \s\_fp \\_fp\_chk:w 2 2 \s\_fp\_exact ; }  
 22560 \tl\_const:Nn \c\_nan\_fp { \s\_fp \\_fp\_chk:w 3 1 \s\_fp\_exact ; }

(End definition for \c\_zero\_fp and others. These variables are documented on page 247.)

\c\_fp\_prec\_int \c\_fp\_half\_prec\_int \c\_fp\_block\_int The number of digits of floating points.  
 22561 \int\_const:Nn \c\_fp\_prec\_int { 16 }  
 22562 \int\_const:Nn \c\_fp\_half\_prec\_int { 8 }  
 22563 \int\_const:Nn \c\_fp\_block\_int { 4 }

(End definition for \c\_fp\_prec\_int, \c\_fp\_half\_prec\_int, and \c\_fp\_block\_int.)

\c\_fp\_myriad\_int Blocks have 4 digits so this integer is useful.  
 22564 \int\_const:Nn \c\_fp\_myriad\_int { 10000 }

(End definition for \c\_fp\_myriad\_int.)

\c\_fp\_minus\_min\_exponent\_int \c\_fp\_max\_exponent\_int Normal floating point numbers have an exponent between \c\_fp\_minus\_min\_exponent and \c\_fp\_max\_exponent inclusive. Larger numbers are rounded to  $\pm\infty$ . Smaller numbers are rounded to  $\pm 0$ . It would be more natural to define a \c\_fp\_min\_exponent with the opposite sign but that would waste one TeX count.  
 22565 \int\_const:Nn \c\_fp\_minus\_min\_exponent\_int { 10000 }  
 22566 \int\_const:Nn \c\_fp\_max\_exponent\_int { 10000 }

(End definition for \c\_fp\_minus\_min\_exponent\_int and \c\_fp\_max\_exponent\_int.)

\c\_fp\_max\_exp\_exponent\_int If a number’s exponent is larger than that, its exponential overflows/underflows.  
 22567 \int\_const:Nn \c\_fp\_max\_exp\_exponent\_int { 5 }

(End definition for \c\_fp\_max\_exp\_exponent\_int.)

\c\_fp\_overflowing\_fp A floating point number that is bigger than all normal floating point numbers. This replaces infinities when converting to formats that do not support infinities.

```

22568 \tl_const:Nx \c_fp_overflowing_fp
22569 {
22570   \s_fp \fp_chk:w 1 0
22571   { \int_eval:n { \c_fp_max_exponent_int + 1 } }
22572   {1000} {0000} {0000} {0000} ;
22573 }
```

(End definition for \c\_fp\_overflowing\_fp.)

\fp\_zero\_fp:N In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```

22574 \cs_new:Npn \fp_zero_fp:N #1
22575   { \s_fp \fp_chk:w 0 #1 \s_fp_underflow ; }
22576 \cs_new:Npn \fp_inf_fp:N #1
22577   { \s_fp \fp_chk:w 2 #1 \s_fp_overflow ; }
```

(End definition for \fp\_zero\_fp:N and \fp\_inf\_fp:N.)

\fp\_exponent:w For normal numbers, the function expands to the exponent, otherwise to 0. This is used in l3str-format.

```

22578 \cs_new:Npn \fp_exponent:w \s_fp \fp_chk:w #1
22579 {
22580   \if_meaning:w 1 #1
22581     \exp_after:wN \fp_use_i_until_s:nnw
22582   \else:
22583     \exp_after:wN \fp_use_i_until_s:nnw
22584     \exp_after:wN 0
22585   \fi:
22586 }
```

(End definition for \fp\_exponent:w.)

\fp\_neg\_sign:N When appearing in an integer expression or after \int\_value:w, this expands to the sign opposite to #1, namely 0 (positive) is turned to 2 (negative), 1 (nan) to 1, and 2 to 0.

```

22587 \cs_new:Npn \fp_neg_sign:N #1
22588   { \fp_int_eval:w 2 - #1 \fp_int_eval_end: }
```

(End definition for \fp\_neg\_sign:N.)

\fp\_kind:w Expands to 0 for zeros, 1 for normal floating point numbers, 2 for infinities, 3 for nan, 4 for tuples.

```

22589 \cs_new:Npn \fp_kind:w #1
22590 {
22591   \fp_if_type_fp:NTwFw
22592   #1 \fp_use_i_until_s:nnw
22593   \s_fp { \fp_use_i_until_s:nnw 4 }
22594   \s_fp_stop
22595 }
```

(End definition for \fp\_kind:w.)

## 66.5 Overflow, underflow, and exact zero

`\_fp_sanitize:Nw`  
`\_fp_sanitize:wN`  
`\_fp_sanitize_zero:w`

Expects the sign and the exponent in some order, then the significand (which we don't touch). Outputs the corresponding floating point number, possibly underflowed to  $\pm 0$  or overflowed to  $\pm\infty$ . The functions `\_fp_underflow:w` and `\_fp_overflow:w` are defined in `l3fp-traps`.

```

22596 \cs_new:Npn \_fp_sanitize:Nw #1 #2;
22597   {
22598     \if_case:w
22599       \if_int_compare:w #2 > \c_fp_max_exponent_int 1 ~ \else:
22600         \if_int_compare:w #2 < - \c_fp_minus_min_exponent_int 2 ~ \else:
22601           \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
22602           \or: \exp_after:wN \_fp_overflow:w
22603           \or: \exp_after:wN \_fp_underflow:w
22604           \or: \exp_after:wN \_fp_sanitize_zero:w
22605           \fi:
22606           \s_fp \_fp_chk:w 1 #1 {#2}
22607   }
22608 \cs_new:Npn \_fp_sanitize:wN #1; #2 { \_fp_sanitize:Nw #2 #1; }
22609 \cs_new:Npn \_fp_sanitize_zero:w \s_fp \_fp_chk:w #1 #2 #3;
22610   { \c_zero_fp }
```

(End definition for `\_fp_sanitize:Nw`, `\_fp_sanitize:wN`, and `\_fp_sanitize_zero:w`.)

## 66.6 Expanding after a floating point number

`\_fp_exp_after_o:w`  
`\_fp_exp_after_f:nw`

`\_fp_exp_after_o:w` (*floating point*)  
`\_fp_exp_after_f:nw` {*tokens*} (*floating point*)

Places *tokens* (empty in the case of `\_fp_exp_after_o:w`) between the *floating point* and the following tokens, then hits those tokens with o or f-expansion, and leaves the floating point number unchanged.

We first distinguish normal floating points, which have a significand, from the much simpler special floating points.

```

22611 \cs_new:Npn \_fp_exp_after_o:w \s_fp \_fp_chk:w #1
22612   {
22613     \if_meaning:w 1 #1
22614       \exp_after:wN \_fp_exp_after_normal:nNNw
22615     \else:
22616       \exp_after:wN \_fp_exp_after_special:nNNw
22617     \fi:
22618     { }
22619     #1
22620   }
22621 \cs_new:Npn \_fp_exp_after_f:nw #1 \s_fp \_fp_chk:w #2
22622   {
22623     \if_meaning:w 1 #2
22624       \exp_after:wN \_fp_exp_after_normal:nNNw
22625     \else:
22626       \exp_after:wN \_fp_exp_after_special:nNNw
22627     \fi:
22628     { \exp:w \exp_end_continue_f:w #1 }
22629     #2
```

```
22630 }
```

(End definition for `\_fp_exp_after_o:w` and `\_fp_exp_after_f:nw`.)

```
\_fp_exp_after_special:nNnw  
22631 \_fp_exp_after_special:nNnw {\(after)} <case> <sign> <scan mark> ;  
Special floating point numbers are easy to jump over since they contain few tokens.  
22632 {  
22633   \exp_after:wN \s_fp  
22634   \exp_after:wN \_fp_chk:w  
22635   \exp_after:wN #2  
22636   \exp_after:wN #3  
22637   \exp_after:wN #4  
22638   \exp_after:wN ;  
22639   #1  
22640 }
```

(End definition for `\_fp_exp_after_special:nNnw`.)

`\_fp_exp_after_normal:nNnw` For normal floating point numbers, life is slightly harder, since we have many tokens to jump over. Here it would be slightly better if the digits were not braced but instead were delimited arguments (for instance delimited by ,). That may be changed some day.

```
22641 \cs_new:Npn \_fp_exp_after_normal:nNnw #1 1 #2 #3 #4#5#6#7;  
22642 {  
22643   \exp_after:wN \_fp_exp_after_normal:Nwwwww  
22644   \exp_after:wN #2  
22645   \int_value:w #3 \exp_after:wN ;  
22646   \int_value:w 1 #4 \exp_after:wN ;  
22647   \int_value:w 1 #5 \exp_after:wN ;  
22648   \int_value:w 1 #6 \exp_after:wN ;  
22649   \int_value:w 1 #7 \exp_after:wN ; #1  
22650 }  
22651 \cs_new:Npn \_fp_exp_after_normal:Nwwwww  
22652   #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;  
22653   { \s_fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }
```

(End definition for `\_fp_exp_after_normal:nNnw`.)

## 66.7 Other floating point types

```
\s_fp_tuple  
\_fp_tuple_chk:w  
\c_fp_empty_tuple_fp
```

Floating point tuples take the form `\s_fp_tuple \_fp_tuple_chk:w { <fp 1> <fp 2> ... }`; where each `<fp>` is a floating point number or tuple, hence ends with ; itself. When a tuple is typeset, `\_fp_tuple_chk:w` produces an error, just like usual floating point numbers. Tuples may have zero or one element.

```
22654 \scan_new:N \s_fp_tuple  
22655 \cs_new_protected:Npn \_fp_tuple_chk:w #1 ;  
22656   { \_fp_misused:n { \s_fp_tuple \_fp_tuple_chk:w #1 ; } }  
22657 \tl_const:Nn \c_fp_empty_tuple_fp  
22658   { \s_fp_tuple \_fp_tuple_chk:w { } ; }
```

(End definition for `\s_fp_tuple`, `\_fp_tuple_chk:w`, and `\c_fp_empty_tuple_fp`.)

`\_fp_tuple_count:w`  
`\_fp_array_count:n`  
`\_fp_tuple_count_loop:Nw`

Count the number of items in a tuple of floating points by counting semicolons. The technique is very similar to `\tl_count:n`, but with the loop built-in. Checking for the end of the loop is done with the `\use_none:n #1` construction.

```

22659 \cs_new:Npn \_fp_array_count:n #
22660   { \_fp_tuple_count:w \s_fp_tuple \_fp_tuple_chk:w {#1} ; }
22661 \cs_new:Npn \_fp_tuple_count:w \s_fp_tuple \_fp_tuple_chk:w #1 ;
22662   {
22663     \int_value:w \_fp_int_eval:w 0
22664       \_fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
22665       \prg_break_point:
22666       \_fp_int_eval_end:
22667   }
22668 \cs_new:Npn \_fp_tuple_count_loop:Nw #1#2;
22669   { \use_none:n #1 + 1 \_fp_tuple_count_loop:Nw }
```

(End definition for `\_fp_tuple_count:w`, `\_fp_array_count:n`, and `\_fp_tuple_count_loop:Nw`.)

`\_fp_if_type_fp:NTwFw`

Used as `\_fp_if_type_fp:NTwFw` *(marker)* {*true code*} `\s_fp` {*false code*} `\s_fp_stop`, this test whether the *(marker)* is `\s_fp` or not and runs the appropriate *(code)*. The very unusual syntax is for optimization purposes as that function is used for all floating point operations.

```
22670 \cs_new:Npn \_fp_if_type_fp:NTwFw #1 \s_fp #2 #3 \s_fp_stop {#2}
```

(End definition for `\_fp_if_type_fp:NTwFw`.)

`\_fp_array_if_all_fp:nTF`  
`\_fp_array_if_all_fp_loop:w`

True if all items are floating point numbers. Used for `min`.

```

22671 \cs_new:Npn \_fp_array_if_all_fp:nTF #1
22672   {
22673     \_fp_array_if_all_fp_loop:w #1 { \s_fp \prg_break: } ;
22674     \prg_break_point: \use_i:nn
22675   }
22676 \cs_new:Npn \_fp_array_if_all_fp_loop:w #1#2 ;
22677   {
22678     \_fp_if_type_fp:NTwFw
22679       #1 \_fp_array_if_all_fp_loop:w
22680       \s_fp { \prg_break:n \use_iii:nnn }
22681       \s_fp_stop
22682   }
```

(End definition for `\_fp_array_if_all_fp:nTF` and `\_fp_array_if_all_fp_loop:w`.)

`\_fp_type_from_scan:N`  
`\_fp_type_from_scan_other:N`  
`\_fp_type_from_scan:w`

Used as `\_fp_type_from_scan:N` *(token)*. Grabs the pieces of the stringified *(token)* which lies after the first `\s_fp`. If the *(token)* does not contain that string, the result is `_?`.

```

22683 \cs_new:Npn \_fp_type_from_scan:N #1
22684   {
22685     \_fp_if_type_fp:NTwFw
22686       #1 { }
22687       \s_fp { \_fp_type_from_scan_other:N #1 }
22688       \s_fp_stop
22689   }
22690 \cs_new:Npx \_fp_type_from_scan_other:N #1
22691   {
22692     \exp_not:N \exp_after:wN \exp_not:N \_fp_type_from_scan:w
```

```

22693     \exp_not:N \token_to_str:N #1 \s__fp_mark
22694         \tl_to_str:n { s__fp _? } \s__fp_mark \s__fp_stop
22695     }
22696 \exp_last_unbraced:NNNNo
22697     \cs_new:Npn \_fp_type_from_scan:w #1
22698     { \tl_to_str:n { s__fp } } #2 \s__fp_mark #3 \s__fp_stop {#2}

(End definition for \_fp_type_from_scan:N, \_fp_type_from_scan_other:N, and \_fp_type_from-
scan:w.)

```

`\_fp_change_func_type:NNN`  
`\_fp_change_func_type_aux:w`  
`\_fp_change_func_type_chk:NNN`

Arguments are *<type marker>* *<function>* *<recovery>*. This gives the function obtained by placing the type after `@@`. If the function is not defined then *<recovery>* *<function>* is used instead; however that test is not run when the *<type marker>* is `\s__fp`.

```

22699 \cs_new:Npn \_fp_change_func_type:NNN #1#2#3
22700     {
22701         \_fp_if_type_fp:NTwFw
22702             #1 #2
22703             \s__fp
22704             {
22705                 \exp_after:wN \_fp_change_func_type_chk:NNN
22706                 \cs:w
22707                     __fp \_fp_type_from_scan_other:N #1
22708                     \exp_after:wN \_fp_change_func_type_aux:w \token_to_str:N #2
22709                     \cs_end:
22710                     #2 #3
22711             }
22712             \s__fp_stop
22713         }
22714 \exp_last_unbraced:NNNNo
22715     \cs_new:Npn \_fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
22716 \cs_new:Npn \_fp_change_func_type_chk:NNN #1#2#3
22717     {
22718         \if_meaning:w \scan_stop: #1
22719             \exp_after:wN #3 \exp_after:wN #2
22720         \else:
22721             \exp_after:wN #1
22722         \fi:
22723     }

(End definition for \_fp_change_func_type:NNN, \_fp_change_func_type_aux:w, and \_fp_change-
func_type_chk:NNN.)

```

`\_fp_exp_after_any_f:Nnw`  
`\_fp_exp_after_any_f:nw`  
`\_fp_exp_after_expr_stop_f:nw`

The `Nnw` function simply dispatches to the appropriate `\_fp_exp_after..._f:nw` with “...” (either empty or *<type>*) extracted from #1, which should start with `\s__fp`. If it doesn’t start with `\s__fp` the function `\_fp_exp_after_?_f:nw` defined in `l3fp-parse` gives an error; another special *<type>* is `stop`, useful for loops, see below. The `nw` function has an important optimization for floating points numbers; it also fetches its type marker #2 from the floating point.

```

22724 \cs_new:Npn \_fp_exp_after_any_f:Nnw #1
22725     { \cs:w __fp_exp_after \_fp_type_from_scan_other:N #1 _f:nw \cs_end: }
22726 \cs_new:Npn \_fp_exp_after_any_f:nw #1#2
22727     {
22728         \_fp_if_type_fp:NTwFw
22729             #2 \_fp_exp_after_f:nw

```

```

22730     \s_fp { \_fp_exp_after_any_f:Nnw #2 }
22731     \s_fp_stop
22732     {#1} #2
22733   }
22734 \cs_new_eq:NN \_fp_exp_after_expr_stop_f:nw \use_none:nn

```

(End definition for `\_fp_exp_after_any_f:Nnw`, `\_fp_exp_after_any_f:nw`, and `\_fp_exp_after_expr_stop_f:nw`.)

`\_fp_exp_after_tuple_o:w`  
`\_fp_exp_after_tuple_f:nw`  
`\_fp_exp_after_array_f:w`

The loop works by using the `n` argument of `\_fp_exp_after_any_f:nw` to place the loop macro after the next item in the tuple and expand it.

```

\_fp_exp_after_array_f:w
⟨fp1⟩ ;
...
⟨fpn⟩ ;
\s_fp_expr_stop

22735 \cs_new:Npn \_fp_exp_after_tuple_o:w
22736   { \_fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
22737 \cs_new:Npn \_fp_exp_after_tuple_f:nw
22738   #1 \s_fp_tuple \_fp_tuple_chk:w #2 ;
22739   {
22740     \exp_after:wN \s_fp_tuple
22741     \exp_after:wN \_fp_tuple_chk:w
22742     \exp_after:wN {
22743       \exp:w \exp_end_continue_f:w
22744       \_fp_exp_after_array_f:w #2 \s_fp_expr_stop
22745     \exp_after:wN }
22746     \exp_after:wN ;
22747     \exp:w \exp_end_continue_f:w #1
22748   }
22749 \cs_new:Npn \_fp_exp_after_array_f:w
22750   { \_fp_exp_after_any_f:nw { \_fp_exp_after_array_f:w } }


```

(End definition for `\_fp_exp_after_tuple_o:w`, `\_fp_exp_after_tuple_f:nw`, and `\_fp_exp_after_array_f:w`.)

## 66.8 Packing digits

When a positive integer `#1` is known to be less than  $10^8$ , the following trick splits it into two blocks of 4 digits, padding with zeros on the left.

```

\cs_new:Npn \pack:NNNNw #1 #2#3#4#5 #6; { {#2#3#4#5} {#6} }
\exp_after:wN \pack:NNNNw
\fp_int_value:w \fp_int_eval:w 1 0000 0000 + #1 ;

```

The idea is that adding  $10^8$  to the number ensures that it has exactly 9 digits, and can then easily find which digits correspond to what position in the number. Of course, this can be modified for any number of digits less or equal to 9 (we are limited by TeX's integers). This method is very heavily relied upon in `13fp-basics`.

More specifically, the auxiliary inserts `+ #1#2#3#4#5 ; {#6}`, which allows us to compute several blocks of 4 digits in a nested manner, performing carries on the fly. Say we want to compute  $12345 \times 66778899$ . With simplified names, we would do

```

\exp_after:wN \post_processing:w
\__fp_int_value:w \__fp_int_eval:w - 5 0000
\exp_after:wN \pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 4 9995 0000
+ 12345 * 6677
\exp_after:wN \pack:NNNNNw
\__fp_int_value:w \__fp_int_eval:w 5 0000 0000
+ 12345 * 8899 ;

```

The `\exp_after:wN` triggers `\int_value:w \__fp_int_eval:w`, which starts a first computation, whose initial value is  $-5\ 0000$  (the “leading shift”). In that computation appears an `\exp_after:wN`, which triggers the nested computation `\int_value:w \__fp_int_eval:w` with starting value  $4\ 9995\ 0000$  (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation’s value is  $5\ 0000\ 0000 + 12345 \times 8899$ , which has 9 digits. Adding  $5 \cdot 10^8$  to the product allowed us to know how many digits to expect as long as the numbers to multiply are not too big; it also works to some extent with negative results. The `pack` function puts the last 4 of those 9 digits into a brace group, moves the semi-colon delimiter, and inserts a `+`, which combines the carry with the previous computation. The shifts nicely combine into  $5\ 0000\ 0000/10^4 + 4\ 9995\ 0000 = 5\ 0000\ 0000$ . As long as the operands are in some range, the result of this second computation has 9 digits. The corresponding `pack` function, expanded after the result is computed, braces the last 4 digits, and leaves `+ <5 digits>` for the initial computation. The “leading shift” cancels the combination of the other shifts, and the `\post_processing:w` takes care of packing the last few digits.

Admittedly, this is quite intricate. It is probably the key in making `l3fp` as fast as other pure `TEX` floating point units despite its increased precision. In fact, this is used so much that we provide different sets of packing functions and shifts, depending on ranges of input.

This set of shifts allows for computations involving results in the range  $[-4 \cdot 10^8, 5 \cdot 10^8 - 1]$ . Shifted values all have exactly 9 digits.

```

22751 \int_const:Nn \c_fp_leading_shift_int { - 5 0000 }
22752 \int_const:Nn \c_fp_middle_shift_int { 5 0000 * 9999 }
22753 \int_const:Nn \c_fp_trailing_shift_int { 5 0000 * 10000 }
22754 \cs_new:Npn \__fp_pack:NNNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }

```

(*End definition for `\__fp_pack:NNNNNw` and others.*)

This set of shifts allows for computations involving results in the range  $[-5 \cdot 10^8, 6 \cdot 10^8 - 1]$  (actually a bit more). Shifted values all have exactly 10 digits. Note that the upper bound is due to `TEX`’s limit of  $2^{31} - 1$  on integers. The shifts are chosen to be roughly the mid-point of  $10^9$  and  $2^{31}$ , the two bounds on 10-digit integers in `TEx`.

```

22755 \int_const:Nn \c_fp_big_leading_shift_int { - 15 2374 }
22756 \int_const:Nn \c_fp_big_middle_shift_int { 15 2374 * 9999 }
22757 \int_const:Nn \c_fp_big_trailing_shift_int { 15 2374 * 10000 }
22758 \cs_new:Npn \__fp_pack_big:NNNNNNw #1#2 #3#4#5#6 #7;
{ + #1#2#3#4#5#6 ; {#7} }

```

(*End definition for `\__fp_pack_big:NNNNNNw` and others.*)

```
\__fp_pack_Bigg:NNNNNNw
\c_fp_Bigg_trailing_shift_int
```

This set of shifts allows for computations with results in the range  $[-1 \cdot 10^9, 147483647]$ ; the end-point is  $2^{31} - 1 - 2 \cdot 10^9 \simeq 1.47 \cdot 10^8$ . Shifted values all have exactly 10 digits.

```
22760 \int_const:Nn \c_fp_Bigg_leading_shift_int { - 20 0000 }
22761 \int_const:Nn \c_fp_Bigg_middle_shift_int { 20 0000 * 9999 }
22762 \int_const:Nn \c_fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
22763 \cs_new:Npn \__fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
22764 { + #1#2#3#4#5#6 ; {#7} }
```

(End definition for `\__fp_pack_Bigg:NNNNNNw` and others.)

```
\__fp_pack_twice_four:wNNNNNNNN
```

```
\__fp_pack_twice_four:wNNNNNNNNN <tokens> ; <≥ 8 digits>
```

Grabs two sets of 4 digits and places them before the semi-colon delimiter. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
22765 \cs_new:Npn \__fp_pack_twice_four:wNNNNNNNNN #1; #2#3#4#5 #6#7#8#9
22766 { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(End definition for `\__fp_pack_twice_four:wNNNNNNNN`.)

```
\__fp_pack_eight:wNNNNNNNN
```

```
\__fp_pack_eight:wNNNNNNNNN <tokens> ; <≥ 8 digits>
```

Grabs one set of 8 digits and places them before the semi-colon delimiter as a single group. Putting several copies of this function before a semicolon packs more digits since each takes the digits packed by the others in its first argument.

```
22767 \cs_new:Npn \__fp_pack_eight:wNNNNNNNNN #1; #2#3#4#5 #6#7#8#9
22768 { #1 {#2#3#4#5#6#7#8#9} ; }
```

(End definition for `\__fp_pack_eight:wNNNNNNNN`.)

```
\__fp_basics_pack_low:NNNNw
```

```
\__fp_basics_pack_high:NNNNw
```

```
\__fp_basics_pack_high_carry:w
```

Addition and multiplication of significands are done in two steps: first compute a (more or less) exact result, then round and pack digits in the final (braced) form. These functions take care of the packing, with special attention given to the case where rounding has caused a carry. Since rounding can only shift the final digit by 1, a carry always produces an exact power of 10. Thus, `\__fp_basics_pack_high_carry:w` is always followed by four times {0000}.

This is used in l3fp-basics and l3fp-extended.

```
22769 \cs_new:Npn \__fp_basics_pack_low:NNNNw #1 #2#3#4#5 #6;
22770 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
22771 \cs_new:Npn \__fp_basics_pack_high:NNNNw #1 #2#3#4#5 #6;
22772 {
22773     \if_meaning:w 2 #
22774         \__fp_basics_pack_high_carry:w
22775     \fi:
22776     ; {#2#3#4#5} {#6}
22777 }
22778 \cs_new:Npn \__fp_basics_pack_high_carry:w \fi: ; #1
22779 { \fi: + 1 ; {1000} }
```

(End definition for `\__fp_basics_pack_low:NNNNw`, `\__fp_basics_pack_high:NNNNw`, and `\__fp_basics_pack_high_carry:w`.)

`\_fp_basics_pack_weird_low:NNNNw`  
`\_fp_basics_pack_weird_high:NNNNNNNNw`

This is used in `l3fp-basics` for additions and divisions. Their syntax is confusing, hence the name.

```

22780 \cs_new:Npn \_fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
22781   {
22782     \if_meaning:w 2 #1
22783       + 1
22784     \fi:
22785     \_fp_int_eval_end:
22786     #2#3#4; {#5} ;
22787   }
22788 \cs_new:Npn \_fp_basics_pack_weird_high:NNNNNNNNw
22789   1 #1#2#3#4 #5#6#7#8 #9; { ; {#1#2#3#4} {#5#6#7#8} {#9} }
```

(End definition for `\_fp_basics_pack_weird_low:NNNNw` and `\_fp_basics_pack_weird_high:NNNNNNNNw`.)

## 66.9 Decimate (dividing by a power of 10)

`\_fp_decimate:nNnnn`

`\_fp_decimate:nNnnn {\langle shift \rangle} {\langle f_1 \rangle}`  
`{\langle X_1 \rangle} {\langle X_2 \rangle} {\langle X_3 \rangle} {\langle X_4 \rangle}`

Each  $\langle X_i \rangle$  consists in 4 digits exactly, and  $1000 \leq \langle X_1 \rangle < 9999$ . The first argument determines by how much we shift the digits.  $\langle f_1 \rangle$  is called as follows:

$\langle f_1 \rangle \langle rounding \rangle {\langle X'_1 \rangle} {\langle X'_2 \rangle} \langle extra-digits \rangle ;$

where  $0 \leq \langle X'_i \rangle < 10^8 - 1$  are 8 digit integers, forming the truncation of our number. In other words,

$$\left( \sum_{i=1}^4 \langle X_i \rangle \cdot 10^{-4i} \cdot 10^{-\langle shift \rangle} \right) - (\langle X'_1 \rangle \cdot 10^{-8} + \langle X'_2 \rangle \cdot 10^{-16}) = 0. \langle extra-digits \rangle \cdot 10^{-16} \in [0, 10^{-16}).$$

To round properly later, we need to remember some information about the difference. The  $\langle rounding \rangle$  digit is 0 if and only if the difference is exactly 0, and 5 if and only if the difference is exactly  $0.5 \cdot 10^{-16}$ . Otherwise, it is the (non-0, non-5) digit closest to  $10^{17}$  times the difference. In particular, if the shift is 17 or more, all the digits are dropped,  $\langle rounding \rangle$  is 1 (not 0), and  $\langle X'_1 \rangle$  and  $\langle X'_2 \rangle$  are both zero.

If the shift is 1, the  $\langle rounding \rangle$  digit is simply the only digit that was pushed out of the brace groups (this is important for subtraction). It would be more natural for the  $\langle rounding \rangle$  digit to be placed after the  $\langle X'_i \rangle$ , but the choice we make involves less reshuffling.

Note that this function treats negative  $\langle shift \rangle$  as 0.

```

22790 \cs_new:Npn \_fp_decimate:nNnnn #1
22791   {
22792     \cs:w
22793       \_fp_decimate_
22794       \if_int_compare:w \_fp_int_eval:w #1 > \c_fp_prec_int
22795         tiny
22796       \else:
22797         \_fp_int_to_roman:w \_fp_int_eval:w #1
22798       \fi:
22799       :Nnnn
22800     \cs_end:
22801   }
```

Each of the auxiliaries see the function  $\langle f_1 \rangle$ , followed by 4 blocks of 4 digits.

(End definition for  $\_\_fp\_decimate:nNnnnn$ .)

If the  $\langle shift \rangle$  is zero, or too big, life is very easy.

```
22802 \cs_new:Npn \_\_fp_decimate_:Nnnnn #1 #2#3#4#5
22803   { #1 0 {#2#3} {#4#5} ; }
22804 \cs_new:Npn \_\_fp_decimate_tiny:Nnnnn #1 #2#3#4#5
22805   { #1 1 { 0000 0000 } { 0000 0000 } 0 #2#3#4#5 ; }
```

(End definition for  $\_\_fp\_decimate_:Nnnnn$  and  $\_\_fp\_decimate_tiny:Nnnnn$ .)

$\_\_fp\_decimate\_auxi:Nnnnn$

Shifting happens in two steps: compute the  $\langle rounding \rangle$  digit, and repack digits into two blocks of 8. The sixteen functions are very similar, and defined through  $\_\_fp\_tmp:w$ . The arguments are as follows: #1 indicates which function is being defined; after one step of expansion, #2 yields the “extra digits” which are then converted by  $\_\_fp\_round\_digit:Nw$  to the  $\langle rounding \rangle$  digit (note the + separating blocks of digits to avoid overflowing TeX’s integers). This triggers the f-expansion of  $\_\_fp\_decimate\_pack:nnnnnnnnnw$ ,<sup>10</sup> responsible for building two blocks of 8 digits, and removing the rest. For this to work, #3 alternates between braced and unbraced blocks of 4 digits, in such a way that the 5 first and 5 next token groups yield the correct blocks of 8 digits.

```
22806 \cs_new:Npn \_\_fp_tmp:w #1 #2 #3
22807   {
22808     \cs_new:cpn { \_\_fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
22809     {
22810       \exp_after:wN ##1
22811       \int_value:w
22812       \exp_after:wN \_\_fp_round_digit:Nw #2 ;
22813       \_\_fp_decimate_pack:nnnnnnnnnw #3 ;
22814     }
22815   }
22816 \_\_fp_tmp:w {i}  {\use_none:nnn      #50}{ 0{#2}#3{#4}#5      }
22817 \_\_fp_tmp:w {ii} {\use_none:nn      #5 }{ 00{#2}#3{#4}#5      }
22818 \_\_fp_tmp:w {iii} {\use_none:n      #5 }{ 000{#2}#3{#4}#5      }
22819 \_\_fp_tmp:w {iv}  {          #5 }{ {0000}#2{#3}#4 #5      }
22820 \_\_fp_tmp:w {v}   {\use_none:nnn    #4#5}{ 0{0000}#2{#3}#4 #5      }
22821 \_\_fp_tmp:w {vi} {\use_none:nn      #4#5}{ 00{0000}#2{#3}#4 #5      }
22822 \_\_fp_tmp:w {vii} {\use_none:n      #4#5}{ 000{0000}#2{#3}#4 #5      }
22823 \_\_fp_tmp:w {viii}{          #4#5}{ {0000}0000{#2}#3 #4 #5      }
22824 \_\_fp_tmp:w {ix}  {\use_none:nnn   #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5      }
22825 \_\_fp_tmp:w {x}   {\use_none:nn      #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5      }
22826 \_\_fp_tmp:w {xi}  {\use_none:n      #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5      }
22827 \_\_fp_tmp:w {xii} {\use_none:nnn   #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5      }
22828 \_\_fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5      }
22829 \_\_fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5      }
22830 \_\_fp_tmp:w {xv}  {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5      }
22831 \_\_fp_tmp:w {xvi} {\use_none:nnn#2#3+#4#5}{ {{0000}0000{0000}0000 #2 #3 #4 #5      }
```

(End definition for  $\_\_fp\_decimate\_auxi:Nnnnn$  and others.)

---

<sup>10</sup>No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```
\_fp_decimate_pack:nnnnnnnnnw
```

The computation of the *rounding* digit leaves an unfinished `\int_value:w`, which expands the following functions. This allows us to repack nicely the digits we keep. Those digits come as an alternation of unbraced and braced blocks of 4 digits, such that the first 5 groups of token consist in 4 single digits, and one brace group (in some order), and the next 5 have the same structure. This is followed by some digits and a semicolon.

```
22832 \cs_new:Npn \_fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
22833   { \_fp_decimate_pack:nnnnnnw { #1#2#3#4#5 } }
22834 \cs_new:Npn \_fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
22835   { {#1} {#2#3#4#5#6} }
```

(End definition for `\_fp_decimate_pack:nnnnnnnnnw`.)

## 66.10 Functions for use within primitive conditional branches

The functions described in this section are not pretty and can easily be misused. When correctly used, each of them removes one `\fi:` as part of its parameter text, and puts one back as part of its replacement text.

Many computation functions in l3fp must perform tests on the type of floating points that they receive. This is often done in an `\if_case:w` statement or another conditional statement, and only a few cases lead to actual computations: most of the special cases are treated using a few standard functions which we define now. A typical use context for those functions would be

```
\if_case:w <integer> \exp_stop_f:
  \_fp_case_return_o:Nw <fp var>
\or: \_fp_case_use:nw {<some computation>}
\or: \_fp_case_return_same_o:w
\or: \_fp_case_return:nw {<something>}
\fi:
<junk>
<floating point>
```

In this example, the case 0 returns the floating point `<fp var>`, expanding once after that floating point. Case 1 does `<some computation>` using the `<floating point>` (presumably compute the operation requested by the user in that non-trivial case). Case 2 returns the `<floating point>` without modifying it, removing the `<junk>` and expanding once after. Case 3 closes the conditional, removes the `<junk>` and the `<floating point>`, and expands `<something>` next. In other cases, the “`<junk>`” is expanded, performing some other operation on the `<floating point>`. We provide similar functions with two trailing `<floating points>`.

```
\_fp_case_use:nw
```

This function ends a TeX conditional, removes junk until the next floating point, and places its first argument before that floating point, to perform some operation on the floating point.

```
22836 \cs_new:Npn \_fp_case_use:nw #1#2 \fi: #3 \s_fp { \fi: #1 \s_fp }
```

(End definition for `\_fp_case_use:nw`.)

\\_\\_fp\\_case\\_return:nw This function ends a TeX conditional, removes junk and a floating point, and places its first argument in the input stream. A quirk is that we don't define this function requiring a floating point to follow, simply anything ending in a semicolon. This, in turn, means that the *junk* may not contain semicolons.

```
22837 \cs_new:Npn \_\_fp\_case\_return:nw #1#2 \fi: #3 ; { \fi: #1 }
```

(End definition for \\_\\_fp\\_case\\_return:nw.)

\\_\\_fp\\_case\\_return\\_o:Nw This function ends a TeX conditional, removes junk and a floating point, and returns its first argument (an *fp var*) then expands once after it.

```
22838 \cs_new:Npn \_\_fp\_case\_return\_o:Nw #1#2 \fi: #3 \s\_fp #4 ;
22839 { \fi: \exp_after:wN #1 }
```

(End definition for \\_\\_fp\\_case\\_return\\_o:Nw.)

\\_\\_fp\\_case\\_return\\_same\\_o:w This function ends a TeX conditional, removes junk, and returns the following floating point, expanding once after it.

```
22840 \cs_new:Npn \_\_fp\_case\_return\_same\_o:w #1 \fi: #2 \s\_fp
22841 { \fi: \_\_fp\_exp\_after\_o:w \s\_fp }
```

(End definition for \\_\\_fp\\_case\\_return\\_same\\_o:w.)

\\_\\_fp\\_case\\_return\\_o:Nww Same as \\_\\_fp\\_case\\_return\\_o:Nw but with two trailing floating points.

```
22842 \cs_new:Npn \_\_fp\_case\_return\_o:Nww #1#2 \fi: #3 \s\_fp #4 ; #5 ;
22843 { \fi: \exp_after:wN #1 }
```

(End definition for \\_\\_fp\\_case\\_return\\_o:Nww.)

\\_\\_fp\\_case\\_return\\_i\\_o:ww \\_\\_fp\\_case\\_return\\_ii\\_o:ww Similar to \\_\\_fp\\_case\\_return\\_same\\_o:w, but this returns the first or second of two trailing floating point numbers, expanding once after the result.

```
22844 \cs_new:Npn \_\_fp\_case\_return\_i\_o:ww #1 \fi: #2 \s\_fp #3 ; \s\_fp #4 ;
22845 { \fi: \_\_fp\_exp\_after\_o:w \s\_fp #3 ; }
22846 \cs_new:Npn \_\_fp\_case\_return\_ii\_o:ww #1 \fi: #2 \s\_fp #3 ;
22847 { \fi: \_\_fp\_exp\_after\_o:w }
```

(End definition for \\_\\_fp\\_case\\_return\\_i\\_o:ww and \\_\\_fp\\_case\\_return\\_ii\\_o:ww.)

## 66.11 Integer floating points

\\_\\_fp\\_int\\_p:w \\_\\_fp\\_int:wTF Tests if the floating point argument is an integer. For normal floating point numbers, this holds if the rounding digit resulting from \\_\\_fp\\_decimate:nNnnnn is 0.

```
22848 \prg_new_conditional:Npnn \_\_fp\_int:w \s\_fp \_\_fp_chk:w #1 #2 #3 #4;
22849 { TF , T , F , p }
22850 {
22851 \if_case:w #1 \exp_stop_f:
22852 \prg_return_true:
22853 \or:
22854 \if_charcode:w 0
22855 \_\_fp_decimate:nNnnnn { \c\_fp_prec_int - #3 }
22856 \_\_fp_use_i_until_s:nw #4
22857 \prg_return_true:
22858 \else:
22859 \prg_return_false:
```

```

22860      \fi:
22861      \else: \prg_return_false:
22862      \fi:
22863  }

```

(End definition for `\_fp_int:wTF`.)

## 66.12 Small integer floating points

```

\fp_small_int:wTF
\fp_small_int_true:wTF
\fp_small_int_normal:NnwTF
\fp_small_int_test:NnnwNTF

```

Tests if the floating point argument is an integer or  $\pm\infty$ . If so, it is clipped to an integer in the range  $[-10^8, 10^8]$  and fed as a braced argument to the *true code*. Otherwise, the *false code* is performed.

First filter special cases: zeros and infinities are integers, `nan` is not. For normal numbers, decimate. If the rounding digit is not 0 run the *false code*. If it is, then the integer is #2 #3; use #3 if #2 vanishes and otherwise  $10^8$ .

```

22864 \cs_new:Npn \fp_small_int:wTF { \fp \fp_chk:w #1#2
22865  {
22866      \if_case:w #1 \exp_stop_f:
22867          \fp_case_return:nw { \fp_small_int_true:wTF 0 } ;
22868      \or: \exp_after:wN \fp_small_int_normal:NnwTF
22869      \or:
22870          \fp_case_return:nw
22871          {
22872              \exp_after:wN \fp_small_int_true:wTF \int_value:w
22873                  \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
22874          }
22875      \else: \fp_case_return:nw \use_i:nn
22876      \fi:
22877      #2
22878  }
22879 \cs_new:Npn \fp_small_int_true:wTF { #1; #2#3 { #2 {#1} } }
22880 \cs_new:Npn \fp_small_int_normal:NnwTF { #1#2#3;
22881  {
22882      \fp_decimate:nNnnnn { \fp_prec_int - #2 }
22883      \fp_small_int_test:NnnwNw
22884      #3 #1
22885  }
22886 \cs_new:Npn \fp_small_int_test:NnnwNw { #1#2#3#4; #5
22887  {
22888      \if_meaning:w 0 #1
22889          \exp_after:wN \fp_small_int_true:wTF
22890          \int_value:w \if_meaning:w 2 #5 - \fi:
22891              \if_int_compare:w #2 > \c_zero_int
22892                  1 0000 0000
22893              \else:
22894                  #3
22895              \fi:
22896          \exp_after:wN ;
22897      \else:
22898          \exp_after:wN \use_i:nn
22899      \fi:
22900  }

```

(End definition for `\fp_small_int:wTF` and others.)

## 66.13 Fast string comparison

\\_fp\\_str\\_if\\_eq:nn A private version of the low-level string comparison function.

22901 \cs\_new\_eq:NN \\_fp\_str\_if\_eq:nn \tex\_strcmp:D

(End definition for \\_fp\\_str\\_if\\_eq:nn.)

## 66.14 Name of a function from its l3fp-parse name

\\_fp\_func\_to\_name:N The goal is to convert for instance \\_fp\_sin\_o:w to sin. This is used in error messages  
\\_fp\_func\_to\_name\_aux:w hence does not need to be fast.

22902 \cs\_new:Npn \\_fp\_func\_to\_name:N #1  
22903 {  
22904 \exp\_last\_unbraced:Nf  
22905 \\_fp\_func\_to\_name\_aux:w { \cs\_to\_str:N #1 } X  
22906 }  
22907 \cs\_set\_protected:Npn \\_fp\_tmp:w #1 #2  
22908 { \cs\_new:Npn \\_fp\_func\_to\_name\_aux:w ##1 #1 ##2 #2 ##3 X {##2} }  
22909 \exp\_args:Nff \\_fp\_tmp:w { \tl\_to\_str:n { \_fp\_ } }  
22910 { \tl\_to\_str:n { \_o: } }

(End definition for \\_fp\_func\_to\_name:N and \\_fp\_func\_to\_name\_aux:w.)

## 66.15 Messages

Using a floating point directly is an error.

22911 \msg\_new:nnnn { fp } { misused }  
22912 { A-floating-point-with-value-'#1'-was-misused. }  
22913 {  
22914 To-obtain-the-value-of-a-floating-point-variable,-use-  
22915 '\token\_to\_str:N \fp\_to\_decimal:N',~  
22916 '\token\_to\_str:N \fp\_to\_tl:N',~or-other-  
22917 conversion-functions.  
22918 }  
22919 \prop\_gput:Nnn \g\_msg\_module\_name\_prop { fp } { LaTeX }  
22920 \prop\_gput:Nnn \g\_msg\_module\_type\_prop { fp } { }  
22921 </package>

# Chapter 67

## I3fp-traps Implementation

22922 `<*package>`

22923 `<@=fp>`

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`
- `underflow`
- `inexact` (actually never used).

### 67.1 Flags

`flag_fp_invalid_operation`

`flag_fp_division_by_zero`

`flag_fp_overflow`

`flag_fp_underflow`

Flags to denote exceptions.

22924 `\flag_new:n { fp_invalid_operation }`  
22925 `\flag_new:n { fp_division_by_zero }`  
22926 `\flag_new:n { fp_overflow }`  
22927 `\flag_new:n { fp_underflow }`

(End definition for flag `fp_invalid_operation` and others. These variables are documented on page 249.)

### 67.2 Traps

Exceptions can be trapped to obtain custom behaviour. When an invalid operation or a division by zero is trapped, the trap receives as arguments the result as an N-type floating point number, the function name (multiple letters for prefix operations, or a single symbol for infix operations), and the operand(s). When an overflow or underflow is trapped, the trap receives the resulting overly large or small floating point number if it is not too big, otherwise it receives  $+\infty$ . Currently, the inexact exception is entirely ignored.

The behaviour when an exception occurs is controlled by the definitions of the functions

- `\__fp_invalid_operation:nnw,`

- `\_fp_invalid_operation_o:Nww,`
- `\_fp_invalid_operation_t1_o:ff,`
- `\_fp_division_by_zero_o:Nnw,`
- `\_fp_division_by_zero_o>NNww,`
- `\_fp_overflow:w,`
- `\_fp_underflow:w.`

Rather than changing them directly, we provide a user interface as `\fp_trap:nn` `{⟨exception⟩} {⟨way of trapping⟩}`, where the *⟨way of trapping⟩* is one of `error`, `flag`, or `none`.

We also provide `\_fp_invalid_operation_o:nw`, defined in terms of `\_fp_invalid_operation:nnw`.

### `\fp_trap:nn`

```

22928 \cs_new_protected:Npn \fp_trap:nn #1#2
22929   {
22930     \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
22931     {
22932       \clist_if_in:mnTF
22933         { invalid_operation , division_by_zero , overflow , underflow }
22934         {#1}
22935         {
22936           \msg_error:nnxx { fp }
22937             { unknown-fpu-trap-type } {#1} {#2}
22938         }
22939         {
22940           \msg_error:nnx
22941             { fp } { unknown-fpu-exception } {#1}
22942         }
22943     }
22944   }

```

(End definition for `\fp_trap:nn`. This function is documented on page 249.)

We provide three types of trapping for invalid operations: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In most cases, the function produces as a result its first argument, possibly with post-expansion.

```

22945 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_error:
22946   { \_fp_trap_invalid_operation_set:N \prg_do_nothing: }
22947 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_flag:
22948   { \_fp_trap_invalid_operation_set:N \use_none:nnnnn }
22949 \cs_new_protected:Npn \_fp_trap_invalid_operation_set_none:
22950   { \_fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
22951 \cs_new_protected:Npn \_fp_trap_invalid_operation_set:N #1
22952   {
22953     \exp_args:Nno \use:n
22954       { \cs_set:Npn \_fp_invalid_operation:nw ##1##2##3; }
22955       {
22956         #1
22957         \_fp_error:nnfn { invalid } {##2} { \fp_to_t1:n { ##3; } } { }

```

```

22958     \flag_raise_if_clear:n { fp_invalid_operation }
22959     ##1
22960   }
22961 \exp_args:Nno \use:n
22962   { \cs_set:Npn \_fp_invalid_operation_o:Nww ##1##2; ##3; }
22963   {
22964     #1
22965     \_fp_error:nffn { invalid-ii }
22966       { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
22967     \flag_raise_if_clear:n { fp_invalid_operation }
22968     \exp_after:wN \c_nan_fp
22969   }
22970 \exp_args:Nno \use:n
22971   { \cs_set:Npn \_fp_invalid_operation_tl_o:ff ##1##2 }
22972   {
22973     #1
22974     \_fp_error:nffn { invalid } {##1} {##2} { }
22975     \flag_raise_if_clear:n { fp_invalid_operation }
22976     \exp_after:wN \c_nan_fp
22977   }
22978 }

```

(End definition for `\_fp_trap_invalid_operation_set_error:` and others.)

We provide three types of trapping for invalid operations and division by zero: either produce an error and raise the relevant flag; or only raise the flag; or don't even raise the flag. In all cases, the function must produce a result, namely its first argument,  $\pm\infty$  or `nan`.

```

22979 \cs_new_protected:Npn \_fp_trap_division_by_zero_set_error:
22980   { \_fp_trap_division_by_zero_set:N \prg_do_nothing: }
22981 \cs_new_protected:Npn \_fp_trap_division_by_zero_set_flag:
22982   { \_fp_trap_division_by_zero_set:N \use_none:nnnn }
22983 \cs_new_protected:Npn \_fp_trap_division_by_zero_set_none:
22984   { \_fp_trap_division_by_zero_set:N \use_none:nnnnnn }
22985 \cs_new_protected:Npn \_fp_trap_division_by_zero_set:N #1
22986   {
22987     \exp_args:Nno \use:n
22988       { \cs_set:Npn \_fp_division_by_zero_o:Nnw ##1##2##3; }
22989     {
22990       #1
22991       \_fp_error:nfn { zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
22992       \flag_raise_if_clear:n { fp_division_by_zero }
22993       \exp_after:wN ##1
22994     }
22995   \exp_args:Nno \use:n
22996     { \cs_set:Npn \_fp_division_by_zero_o:NNww ##1##2##3; ##4; }
22997     {
22998       #1
22999       \_fp_error:nffn { zero-div-ii }
23000         { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
23001       \flag_raise_if_clear:n { fp_division_by_zero }
23002       \exp_after:wN ##1
23003     }
23004 }

```

(End definition for `\_fp_trap_division_by_zero_set_error`: and others.)

```
\_fp_trap_overflow_set_error:
\fp_trap_overflow_set_flag:
\fp_trap_overflow_set_none:
\fp_trap_overflow_set:N
\fp_trap_underflow_set_error:
\fp_trap_underflow_set_flag:
\fp_trap_underflow_set_none:
\fp_trap_underflow_set:N
\fp_trap_overflow_set:NnNn
```

Just as for invalid operations and division by zero, the three different behaviours are obtained by feeding `\prg_do_nothing`, `\use_none:nnnnn` or `\use_none:nnnnnnn` to an auxiliary, with a further auxiliary common to overflow and underflow functions. In most cases, the argument of the `\fp_overflow:w` and `\fp_underflow:w` functions will be an (almost) normal number (with an exponent outside the allowed range), and the error message thus displays that number together with the result to which it overflowed or underflowed. For extreme cases such as  $10^{**} 1e9999$ , the exponent would be too large for TeX, and `\fp_overflow:w` receives  $\pm\infty$  (`\fp_underflow:w` would receive  $\pm 0$ ); then we cannot do better than simply say an overflow or underflow occurred.

```
23005 \cs_new_protected:Npn \fp_trap_overflow_set_error:
23006   { \fp_trap_overflow_set:N \prg_do_nothing: }
23007 \cs_new_protected:Npn \fp_trap_overflow_set_flag:
23008   { \fp_trap_overflow_set:N \use_none:nnnnn }
23009 \cs_new_protected:Npn \fp_trap_overflow_set_none:
23010   { \fp_trap_overflow_set:N \use_none:nnnnnnn }
23011 \cs_new_protected:Npn \fp_trap_overflow_set:N #1
23012   { \fp_trap_overflow_set:NnNn #1 { overflow } \fp_inf_fp:N { inf } }
23013 \cs_new_protected:Npn \fp_trap_underflow_set_error:
23014   { \fp_trap_underflow_set:N \prg_do_nothing: }
23015 \cs_new_protected:Npn \fp_trap_underflow_set_flag:
23016   { \fp_trap_underflow_set:N \use_none:nnnnn }
23017 \cs_new_protected:Npn \fp_trap_underflow_set_none:
23018   { \fp_trap_underflow_set:N \use_none:nnnnnnn }
23019 \cs_new_protected:Npn \fp_trap_underflow_set:N #1
23020   { \fp_trap_overflow_set:NnNn #1 { underflow } \fp_zero_fp:N { 0 } }
23021 \cs_new_protected:Npn \fp_trap_overflow_set:NnNn #1#2#3#4
23022   {
23023     \exp_args:Nno \use:n
23024       { \cs_set:cpn { fp_#2 :w } \s_fp \fp_chk:w ##1##2##3; }
23025     {
23026       #1
23027       \fp_error:nffn
23028         { flow \if_meaning:w 1 ##1 -to \fi: }
23029         { \fp_to_tl:n { \s_fp \fp_chk:w ##1##2##3; } }
23030         { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
23031         {##2}
23032         \flag_raise_if_clear:n { fp_#2 }
23033         #3 ##2
23034     }
23035   }
```

(End definition for `\fp_trap_overflow_set_error`: and others.)

```
\fp_invalid_operation:nw
\fp_invalid_operation_o:Nww
\fp_invalid_operation_tl_o:ff
\fp_division_by_zero_o:Nnw
\fp_division_by_zero_o:NNww
\fp_overflow:w
\fp_underflow:w
```

Initialize the control sequences (to log properly their existence). Then set invalid operations to trigger an error, and division by zero, overflow, and underflow to act silently on their flag.

```
23036 \cs_new:Npn \fp_invalid_operation:nw #1#2#3; { }
23037 \cs_new:Npn \fp_invalid_operation_o:Nww #1#2; #3; { }
23038 \cs_new:Npn \fp_invalid_operation_tl_o:ff #1 #2 { }
23039 \cs_new:Npn \fp_division_by_zero_o:Nnw #1#2#3; { }
23040 \cs_new:Npn \fp_division_by_zero_o:NNww #1#2#3; #4; { }
```

```

23041 \cs_new:Npn \__fp_overflow:w { }
23042 \cs_new:Npn \__fp_underflow:w { }
23043 \fp_trap:nn { invalid_operation } { error }
23044 \fp_trap:nn { division_by_zero } { flag }
23045 \fp_trap:nn { overflow } { flag }
23046 \fp_trap:nn { underflow } { flag }

```

(End definition for `\__fp_invalid_operation:nnw` and others.)

`\__fp_invalid_operation_o:nw` Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

```

23047 \cs_new:Npn \__fp_invalid_operation_o:nw
23048   { \__fp_invalid_operation:nnw { \exp_after:wN \c_nan_fp } }
23049 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for `\__fp_invalid_operation_o:nw`.)

## 67.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn
\__fp_error:nfnf
\__fp_error:nfff

```

(End definition for `\__fp_error:nnnn`.)

## 67.4 Messages

Some messages.

```

23053 \msg_new:nnnn { fp } { unknown-fpu-exception }
23054   {
23055     The~FPU~exception~'#1'~is~not~known:~
23056     that~trap~will~never~be~triggered.
23057   }
23058   {
23059     The~only~exceptions~to~which~traps~can~be~attached~are \\
23060     \iow_indent:n
23061       {
23062         * ~ invalid_operation \\
23063         * ~ division_by_zero \\
23064         * ~ overflow \\
23065         * ~ underflow
23066       }
23067     }
23068 \msg_new:nnnn { fp } { unknown-fpu-trap-type }
23069   { The~FPU~trap~type~'#2'~is~not~known. }
23070   {
23071     The~trap~type~must~be~one~of \\
23072     \iow_indent:n
23073       {
23074         * ~ error \\
23075         * ~ flag \\

```

```
23076      * ~ none
23077      }
23078  }
23079 \msg_new:nnn { fp } { flow }
23080   { An ~ #3 ~ occurred. }
23081 \msg_new:nnn { fp } { flow-to }
23082   { #1 ~ #3 ed ~ to ~ #2 . }
23083 \msg_new:nnn { fp } { zero-div }
23084   { Division~by-zero~in~ #1 (#2) }
23085 \msg_new:nnn { fp } { zero-div-ii }
23086   { Division~by-zero~in~ (#1) #3 (#2) }
23087 \msg_new:nnn { fp } { invalid }
23088   { Invalid~operation~ #1 (#2) }
23089 \msg_new:nnn { fp } { invalid-ii }
23090   { Invalid~operation~ (#1) #3 (#2) }
23091 \msg_new:nnn { fp } { unknown-type }
23092   { Unknown~type~for~'#1' }
23093 </package>
```

# Chapter 68

## I3fp-round implementation

```
23094 <*package>
23095 <@@=fp>

\_\_fp\_parse\_word\_trunc:N
\_\_fp\_parse\_word\_floor:N
\_\_fp\_parse\_word\_ceil:N
23096 \cs_new:Npn \_\_fp\_parse\_word\_trunc:N
23097   { \_\_fp\_parse\_function>NNN \_\_fp\_round_o:Nw \_\_fp\_round\_to\_zero>NNN }
23098 \cs_new:Npn \_\_fp\_parse\_word\_floor:N
23099   { \_\_fp\_parse\_function>NNN \_\_fp\_round_o:Nw \_\_fp\_round\_to\_ninf>NNN }
23100 \cs_new:Npn \_\_fp\_parse\_word\_ceil:N
23101   { \_\_fp\_parse\_function>NNN \_\_fp\_round_o:Nw \_\_fp\_round\_to\_pinf>NNN }

(End definition for \_\_fp\_parse\_word\_trunc:N, \_\_fp\_parse\_word\_floor:N, and \_\_fp\_parse\_word\_ceil:N.)
```

```
\_\_fp\_parse\_word\_round:N
\_\_fp\_parse\_round:Nw
23102 \cs_new:Npn \_\_fp\_parse\_word\_round:N #1#2
23103   {
23104     \_\_fp\_parse\_function>NNN
23105     \_\_fp\_round_o:Nw \_\_fp\_round\_to\_nearest>NNN #1
23106     #2
23107   }
23108 \cs_new:Npn \_\_fp\_parse\_round:Nw #1 #2 \_\_fp\_round\_to\_nearest>NNN #3#4
23109   { #2 #1 #3 }

(End definition for \_\_fp\_parse\_word\_round:N and \_\_fp\_parse\_round:Nw.)
```

### 68.1 Rounding tools

\c\\_fp\\_five\\_int This is used as the half-point for which numbers are rounded up/down.

```
23111 \int_const:Nn \c\_fp\_five\_int { 5 }
```

(End definition for \c\\_fp\\_five\\_int.)

Floating point operations often yield a result that cannot be exactly represented in a significand with 16 digits. In that case, we need to round the exact result to a representable number. The IEEE standard defines four rounding modes:

- Round to nearest: round to the representable floating point number whose absolute difference with the exact result is the smallest. If the exact result lies exactly at the mid-point between two consecutive representable floating point numbers, round to the floating point number whose last digit is even.
- Round towards negative infinity: round to the greatest floating point number not larger than the exact result.
- Round towards zero: round to a floating point number with the same sign as the exact result, with the largest absolute value not larger than the absolute value of the exact result.
- Round towards positive infinity: round to the least floating point number not smaller than the exact result.

This is not fully implemented in l3fp yet, and transcendental functions fall back on the “round to nearest” mode. All rounding for basic algebra is done through the functions defined in this module, which can be redefined to change their rounding behaviour (but there is not interface for that yet).

The rounding tools available in this module are many variations on a base function `\__fp_round:NNN`, which expands to `0\exp_stop_f:` or `1\exp_stop_f:` depending on whether the final result should be rounded up or down.

- `\__fp_round:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f::`.
- `\__fp_round_s:NNNw <sign> <digit1> <digit2> <more digits>`; can expand to `0\exp_stop_f::`; or `1\exp_stop_f::`.
- `\__fp_round_neg:NNN <sign> <digit1> <digit2>` can expand to `0\exp_stop_f:` or `1\exp_stop_f::`.

See implementation comments for details on the syntax.

```
\__fp_round:NNN
\__fp_round_to_nearest:NNN
  \__fp_round_to_nearest_ninf:NNN
  \__fp_round_to_nearest_zero:NNN
  \__fp_round_to_nearest_pinf:NNN
\__fp_round_to_ninf:NNN
\__fp_round_to_zero:NNN
\__fp_round_to_pinf:NNN
```

If rounding the number `<final sign><digit1>.<digit2>` to an integer rounds it towards zero (truncates it), this function expands to `0\exp_stop_f::`, and otherwise to `1\exp_stop_f::`. Typically used within the scope of an `\__fp_int_eval:w`, to add 1 if needed, and thereby round correctly. The result depends on the rounding mode.

It is very important that `<final sign>` be the final sign of the result. Otherwise, the result would be incorrect in the case of rounding towards  $-\infty$  or towards  $+\infty$ . Also recall that `<final sign>` is 0 for positive, and 2 for negative.

By default, the functions below return `0\exp_stop_f::`, but this is superseded by `\__fp_round_return_one::`, which instead returns `1\exp_stop_f::`, expanding everything and removing `0\exp_stop_f::` in the process. In the case of rounding towards  $\pm\infty$  or towards 0, this is not really useful, but it prepares us for the “round to nearest, ties to even” mode.

The “round to nearest” mode is the default. If the `<digit2>` is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that `<digit1>` plus the result is even. In other words, round up if `<digit1>` is odd.

The “round to nearest” mode has three variants, which differ in how ties are rounded: down towards  $-\infty$ , truncated towards 0, or up towards  $+\infty$ .

```
23112 \cs_new:Npn \__fp_round_return_one:
23113   { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
```

```

23114 \cs_new:Npn \__fp_round_to_ninf:NNN #1 #2 #3
23115 {
23116     \if_meaning:w 2 #1
23117         \if_int_compare:w #3 > \c_zero_int
23118             \__fp_round_return_one:
23119         \fi:
23120     \fi:
23121     \c_zero_int
23122 }
23123 \cs_new:Npn \__fp_round_to_zero:NNN #1 #2 #3 { \c_zero_int }
23124 \cs_new:Npn \__fp_round_to_pinf:NNN #1 #2 #3
23125 {
23126     \if_meaning:w 0 #1
23127         \if_int_compare:w #3 > \c_zero_int
23128             \__fp_round_return_one:
23129         \fi:
23130     \fi:
23131     \c_zero_int
23132 }
23133 \cs_new:Npn \__fp_round_to_nearest:NNN #1 #2 #3
23134 {
23135     \if_int_compare:w #3 > \c_fp_five_int
23136         \__fp_round_return_one:
23137     \else:
23138         \if_meaning:w 5 #3
23139             \if_int_odd:w #2 \exp_stop_f:
23140                 \__fp_round_return_one:
23141             \fi:
23142         \fi:
23143     \fi:
23144     \c_zero_int
23145 }
23146 \cs_new:Npn \__fp_round_to_nearest_ninf:NNN #1 #2 #3
23147 {
23148     \if_int_compare:w #3 > \c_fp_five_int
23149         \__fp_round_return_one:
23150     \else:
23151         \if_meaning:w 5 #3
23152             \if_meaning:w 2 #1
23153                 \__fp_round_return_one:
23154             \fi:
23155         \fi:
23156     \fi:
23157     \c_zero_int
23158 }
23159 \cs_new:Npn \__fp_round_to_nearest_zero:NNN #1 #2 #3
23160 {
23161     \if_int_compare:w #3 > \c_fp_five_int
23162         \__fp_round_return_one:
23163     \fi:
23164     \c_zero_int
23165 }
23166 \cs_new:Npn \__fp_round_to_nearest_pinf:NNN #1 #2 #3
23167 {

```

```

23168     \if_int_compare:w #3 > \c_fp_five_int
23169         \_fp_round_return_one:
23170     \else:
23171         \if_meaning:w 5 #3
23172             \if_meaning:w 0 #1
23173                 \_fp_round_return_one:
23174             \fi:
23175         \fi:
23176     \fi:
23177     \c_zero_int
23178 }
23179 \cs_new_eq:NN \_fp_round:NNN \_fp_round_to_nearest:NNN

```

(End definition for `\_fp_round:NNN` and others.)

`\_fp_round_s:NNNw`

```

\_fp_round_s:NNNw <final sign> <digit> <more digits>;

```

Similar to `\_fp_round:NNN`, but with an extra semicolon, this function expands to `0\exp_stop_f:`; if rounding `<final sign><digit>.<more digits>` to an integer truncates, and to `1\exp_stop_f:`; otherwise. The `<more digits>` part must be a digit, followed by something that does not overflow a `\int_use:N \_fp_int_eval:w` construction. The only relevant information about this piece is whether it is zero or not.

```

23180 \cs_new:Npn \_fp_round_s:NNNw #1 #2 #3 #4;
23181 {
23182     \exp_after:wN \_fp_round:NNN
23183     \exp_after:wN #1
23184     \exp_after:wN #2
23185     \int_value:w \_fp_int_eval:w
23186     \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
23187         \if_meaning:w 5 #3 1 \fi:
23188         \exp_stop_f:
23189         \if_int_compare:w \_fp_int_eval:w #4 > \c_zero_int
23190             1 +
23191         \fi:
23192     \fi:
23193     #3
23194 ;
23195 }

```

(End definition for `\_fp_round_s:NNNw`.)

`\_fp_round_digit:Nw`

```

\int_value:w \_fp_round_digit:Nw <digit> <intexpr>;

```

This function should always be called within an `\int_value:w` or `\_fp_int_eval:w` expansion; it may add an extra `\_fp_int_eval:w`, which means that the integer or integer expression should not be ended with a synonym of `\relax`, but with a semi-colon for instance.

```

23196 \cs_new:Npn \_fp_round_digit:Nw #1 #2;
23197 {
23198     \if_int_odd:w \if_meaning:w 0 #1 1 \else:
23199         \if_meaning:w 5 #1 1 \else:
23200             0 \fi: \fi: \exp_stop_f:
23201         \if_int_compare:w \_fp_int_eval:w #2 > \c_zero_int
23202             \_fp_int_eval:w 1 +
23203         \fi:
23204     \fi:

```

```

23205      #1
23206    }

```

(End definition for `\_fp_round_digit:Nw.`)

```

\_\_fp\_round\_neg:NNN
\_\_fp\_round\_to\_nearest\_neg:NNN
\_\_fp\_round\_to\_nearest\_ninf\_neg:NNN
\_\_fp\_round\_to\_nearest\_zero\_neg:NNN
\_\_fp\_round\_to\_nearest\_pinf\_neg:NNN
\_\_fp\_round\_to\_ninf\_neg:NNN
\_\_fp\_round\_to\_zero\_neg:NNN
\_\_fp\_round\_to\_pinf\_neg:NNN

```

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test. Starting from a number of the form  $\langle\text{final sign}\rangle 0.(15 \text{ digits}) \langle\text{digit}_1\rangle$  with exactly 15 (non-all-zero) digits before  $\langle\text{digit}_1\rangle$ , subtract from it  $\langle\text{final sign}\rangle 0.0\dots 0 \langle\text{digit}_2\rangle$ , where there are 16 zeros. If in the current rounding mode the result should be rounded down, then this function returns `1\exp_stop_f:`. Otherwise, *i.e.*, if the result is rounded back to the first operand, then this function returns `0\exp_stop_f:`.

It turns out that this negative “round to nearest” is identical to the positive one. And this is the default mode.

```

23207 \cs_new_eq:NN \_\_fp_round_to_ninf_neg:NNN \_\_fp_round_to_pinf:NNN
23208 \cs_new:Npn \_\_fp_round_to_zero_neg:NNN #1 #2 #3
23209  {
23210    \if_int_compare:w #3 > \c_zero_int
23211      \_\_fp_round_return_one:
23212    \fi:
23213    \c_zero_int
23214  }
23215 \cs_new_eq:NN \_\_fp_round_to_pinf_neg:NNN \_\_fp_round_to_ninf:NNN
23216 \cs_new_eq:NN \_\_fp_round_to_nearest_neg:NNN \_\_fp_round_to_nearest:NNN
23217 \cs_new_eq:NN \_\_fp_round_to_nearest_ninf_neg:NNN
23218   \_\_fp_round_to_nearest_pinf:NNN
23219 \cs_new:Npn \_\_fp_round_to_nearest_zero_neg:NNN #1 #2 #3
23220  {
23221    \if_int_compare:w #3 < \c_fp_five_int \else:
23222      \_\_fp_round_return_one:
23223    \fi:
23224    \c_zero_int
23225  }
23226 \cs_new_eq:NN \_\_fp_round_to_nearest_pinf_neg:NNN
23227   \_\_fp_round_to_nearest_ninf:NNN
23228 \cs_new_eq:NN \_\_fp_round_neg:NNN \_\_fp_round_to_nearest_neg:NNN

```

(End definition for `\_fp_round_neg:NNN` and others.)

## 68.2 The round function

```

\_\_fp_round_o:Nw
\_\_fp_round_aux_o:Nw

```

First check that all arguments are floating point numbers. The `trunc`, `ceil` and `floor` functions expect one or two arguments (the second is 0 by default), and the `round` function also accepts a third argument (`nan` by default), which changes #1 from `\_fp_round_to_nearest:NNN` to one of its analogues.

```

23229 \cs_new:Npn \_\_fp_round_o:Nw #1
23230  {
23231    \_\_fp_parse_function_all_fp_o:fnw
23232    { \_\_fp_round_name_from_cs:N #1 }
23233    { \_\_fp_round_aux_o:Nw #1 }
23234  }
23235 \cs_new:Npn \_\_fp_round_aux_o:Nw #1#2 @

```

```

23236   {
23237     \if_case:w
23238       \_\_fp\_int\_eval:w \_\_fp\_array\_count:n {#2} \_\_fp\_int\_eval\_end:
23239         \_\_fp\_round\_no\_arg_o:Nw #1 \exp:w
23240       \or: \_\_fp\_round:Nwn #1 #2 {0} \exp:w
23241       \or: \_\_fp\_round:Nww #1 #2 \exp:w
23242       \else: \_\_fp\_round:Nwww #1 #2 @ \exp:w
23243     \fi:
23244     \exp_after:wN \exp_end:
23245   }

```

(End definition for `\_\_fp\_round_o:Nw` and `\_\_fp\_round\_aux_o:Nw`.)

`\_\_fp\_round\_no\_arg_o:Nw`

```

23246   \cs_new:Npn \_\_fp\_round\_no\_arg_o:Nw #1
23247   {
23248     \cs_if_eq:NNTF #1 \_\_fp\_round_to_nearest:NNN
23249       { \_\_fp_error:nnnn { num-args } { round () } { 1 } { 3 } }
23250       {
23251         \_\_fp_error:nffn { num-args }
23252           { \_\_fp\_round\_name\_from\_cs:N #1 () } { 1 } { 2 }
23253       }
23254     \exp_after:wN \c_nan_fp
23255   }

```

(End definition for `\_\_fp\_round_no_arg_o:Nw`.)

`\_\_fp\_round:Nwww`

Having three arguments is only allowed for `round`, not `trunc`, `ceil`, `floor`, so check for that case. If all is well, construct one of `\_\_fp\_round_to_nearest:NNN`, `\_\_fp\_round_to_nearest_zero:NNN`, `\_\_fp\_round_to_nearest_ninf:NNN`, `\_\_fp\_round_to_nearest_pinf:NNN` and act accordingly.

```

23256   \cs_new:Npn \_\_fp\_round:Nwww #1#2 ; #3 ; \s_fp \_\_fp_chk:w #4#5#6 ; #7 @
23257   {
23258     \cs_if_eq:NNTF #1 \_\_fp\_round_to_nearest:NNN
23259     {
23260       \tl_if_empty:nTF {#7}
23261       {
23262         \exp_args:Nc \_\_fp\_round:Nww
23263         {
23264           \_\_fp\_round_to_nearest
23265           \if_meaning:w 0 #4 _zero \else:
23266             \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
23267             :NNN
23268           }
23269         #2 ; #3 ;
23270       }
23271     {
23272       \_\_fp_error:nnnn { num-args } { round () } { 1 } { 3 }
23273       \exp_after:wN \c_nan_fp
23274     }
23275   }
23276   {
23277     \_\_fp_error:nffn { num-args }
23278       { \_\_fp\_round\_name\_from\_cs:N #1 () } { 1 } { 2 }

```

```

23279           \exp_after:wN \c_nan_fp
23280       }
23281   }

```

(End definition for `\_fp_round:Nwww`.)

```

\fp_round_name_from_cs:N
23282 \cs_new:Npn \fp_round_name_from_cs:N #1
23283 {
23284     \cs_if_eq:NNTF #1 \fp_round_to_zero:NNN { trunc }
23285     {
23286         \cs_if_eq:NNTF #1 \fp_round_to_ninf:NNN { floor }
23287         {
23288             \cs_if_eq:NNTF #1 \fp_round_to_pinf:NNN { ceil }
23289             { round }
23290         }
23291     }
23292 }

```

(End definition for `\_fp_round_name_from_cs:N`.)

`\fp_round:Nww`  
`\fp_round:Nwn`  
`\fp_round_normal:NwNNnw`  
`\fp_round_normal:NnnwNNnn`  
`\fp_round_normal:Nw`  
`\fp_round_normal>NNwNnn`  
`\fp_round_normal_end:wwNnn`  
`\fp_round_special:NwwNnn`  
`\fp_round_special_aux:Nw`

If the number of digits to round to is an integer or infinity all is good; if it is `nan` then just produce a `nan`; otherwise invalid as we have something like `round(1,3.14)` where the number of digits is not an integer.

```

23293 \cs_new:Npn \fp_round:Nww #1#2 ; #3 ;
23294 {
23295     \fp_small_int:wTF #3; { \fp_round:Nwn #1#2; }
23296     {
23297         \if:w 3 \fp_kind:w #3 ;
23298             \exp_after:wN \use_i:nn
23299         \else:
23300             \exp_after:wN \use_ii:nn
23301         \fi:
23302             { \exp_after:wN \c_nan_fp }
23303         {
23304             \fp_invalid_operation_tl_o:ff
23305                 { \fp_round_name_from_cs:N #1 }
23306                 { \fp_array_to_clist:n { #2; #3; } }
23307         }
23308     }
23309 }
23310 \cs_new:Npn \fp_round:Nwn #1 \s_fp \fp_chk:w #2#3#4; #5
23311 {
23312     \if_meaning:w 1 #2
23313         \exp_after:wN \fp_round_normal:NwNNnw
23314         \exp_after:wN #1
23315         \int_value:w #5
23316     \else:
23317         \exp_after:wN \fp_exp_after_o:w
23318     \fi:
23319         \s_fp \fp_chk:w #2#3#4;
23320     }
23321 \cs_new:Npn \fp_round_normal:NwNNnw #1#2 \s_fp \fp_chk:w 1#3#4#5;
23322 {

```

```

23323     \__fp_decimate:nNnnnn { \c_fp_prec_int - #4 - #2 }
23324         \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
23325     }
23326 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
23327     {
23328         \exp_after:wN \__fp_round_normal>NNwNnn
23329         \int_value:w \__fp_int_eval:w
23330         \if_int_compare:w #2 > \c_zero_int
23331             1 \int_value:w #2
23332             \exp_after:wN \__fp_round_pack:Nw
23333             \int_value:w \__fp_int_eval:w 1#3 +
23334         \else:
23335             \if_int_compare:w #3 > \c_zero_int
23336                 1 \int_value:w #3 +
23337             \fi:
23338         \fi:
23339         \exp_after:wN #5
23340         \exp_after:wN #6
23341         \use_none:nnnnnnnn #3
23342         #1
23343         \__fp_int_eval_end:
23344             0000 0000 0000 0000 ; #6
23345     }
23346 \cs_new:Npn \__fp_round_pack:Nw #1
23347     { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
23348 \cs_new:Npn \__fp_round_normal>NNwNnn #1 #2
23349     {
23350         \if_meaning:w 0 #2
23351             \exp_after:wN \__fp_round_special:NwwNnn
23352             \exp_after:wN #1
23353         \fi:
23354         \__fp_pack_twice_four:wNNNNNNNN
23355         \__fp_pack_twice_four:wNNNNNNNN
23356         \__fp_round_normal_end:wwNnn
23357         ; #2
23358     }
23359 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
23360     {
23361         \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
23362         \__fp_SANITIZE:Nw #3 #4 ; #1 ;
23363     }
23364 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
23365     {
23366         \if_meaning:w 0 #1
23367             \__fp_case_return:nw
23368             { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
23369         \else:
23370             \exp_after:wN \__fp_round_special_aux:Nw
23371             \exp_after:wN #4
23372             \int_value:w \__fp_int_eval:w 1
23373             \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
23374         \fi:
23375         ;
23376     }

```

```
23377 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
23378   {
23379     \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
23380     \__fp_SANITIZE:Nw #1#2; {1000}{0000}{0000}{0000};
23381   }
23382 
```

(End definition for `\__fp_round:Nww` and others.)

```
23382 </package>
```

# Chapter 69

## l3fp-parse implementation

23383 `<*package>`

23384 `<@@=fp>`

### 69.1 Work plan

The task at hand is non-trivial, and some previous failed attempts show that the code leads to unreadable logs, so we had better get it (almost) right the first time. Let us first describe our goal, then discuss the design precisely before writing any code.

In this file at least, a `<floating point object>` is a floating point number or tuple. This can be extended to anything that starts with `\s_fp` or `\s_fp_{type}` and ends with ; with some internal structure that depends on the `<type>`.

`\_fp_parse:n`      `\_fp_parse:n {<fpexpr>}`

Evaluates the `<floating point expression>` and leaves the result in the input stream as a floating point object. This function forms the basis of almost all public l3fp functions. During evaluation, each token is fully f-expanded.

`\_fp_parse_o:n` does the same but expands once after its result.

**TeXhackers note:** Registers (integers, toks, etc.) are automatically unpacked, without requiring a function such as `\int_use:N`. Invalid tokens remaining after f-expansion lead to unrecoverable low-level TeX errors.

(End definition for `\_fp_parse:n`.)

`\c_fp_prec_func_int`  
`\c_fp_prec_hatii_int`  
`\c_fp_prec_hat_int`  
`\c_fp_prec_not_int`  
`\c_fp_prec_juxtap_int`  
`\c_fp_prec_times_int`  
`\c_fp_prec_plus_int`  
`\c_fp_prec_comp_int`  
`\c_fp_prec_and_int`  
`\c_fp_prec_or_int`

`\c_fp_prec_quest_int`  
`\c_fp_prec_colon_int`  
`\c_fp_prec_comma_int`  
`\c_fp_prec_tuple_int`  
`\c_fp_prec_end_int`

16 Function calls.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

11 Juxtaposition (implicit `*`) with no parenthesis.

- 10 Binary `*` and `/`.
- 9 Binary `+` and `-`.
- 7 Comparisons.
- 6 Logical `and`, denoted by `&&`.
- 5 Logical `or`, denoted by `||`.
- 4 Ternary operator `?:`, piece `?`.
- 3 Ternary operator `??:`, piece `:`.
- 2 Commas.
- 1 Place where a comma is allowed and generates a tuple.
- 0 Start and end of the expression.

```

23385 \int_const:Nn \c_fp_prec_func_int { 16 }
23386 \int_const:Nn \c_fp_prec_hatii_int { 14 }
23387 \int_const:Nn \c_fp_prec_hat_int { 13 }
23388 \int_const:Nn \c_fp_prec_not_int { 12 }
23389 \int_const:Nn \c_fp_prec_juxtap_int { 11 }
23390 \int_const:Nn \c_fp_prec_times_int { 10 }
23391 \int_const:Nn \c_fp_prec_plus_int { 9 }
23392 \int_const:Nn \c_fp_prec_comp_int { 7 }
23393 \int_const:Nn \c_fp_prec_and_int { 6 }
23394 \int_const:Nn \c_fp_prec_or_int { 5 }
23395 \int_const:Nn \c_fp_prec_quest_int { 4 }
23396 \int_const:Nn \c_fp_prec_colon_int { 3 }
23397 \int_const:Nn \c_fp_prec_comma_int { 2 }
23398 \int_const:Nn \c_fp_prec_tuple_int { 1 }
23399 \int_const:Nn \c_fp_prec_end_int { 0 }

```

*(End definition for `\c_fp_prec_func_int` and others.)*

### 69.1.1 Storing results

The main question in parsing expressions expandably is to decide where to put the intermediate results computed for various subexpressions.

One option is to store the values at the start of the expression, and carry them together as the first argument of each macro. However, we want to f-expand tokens one by one in the expression (as `\int_eval:n` does), and with this approach, expanding the next unread token forces us to jump with `\exp_after:wN` over every value computed earlier in the expression. With this approach, the run-time grows at least quadratically in the length of the expression, if not as its cube (inserting the `\exp_after:wN` is tricky and slow).

A second option is to place those values at the end of the expression. Then expanding the next unread token is straightforward, but this still hits a performance issue: for long expressions we would be reaching all the way to the end of the expression at every step of the calculation. The run-time is again quadratic.

A variation of the above attempts to place the intermediate results which appear when computing a parenthesized expression near the closing parenthesis. This still lets

us expand tokens as we go, and avoids performance problems as long as there are enough parentheses. However, it would be better to avoid requiring the closing parenthesis to be present as soon as the corresponding opening parenthesis is read: the closing parenthesis may still be hidden in a macro yet to be expanded.

Hence, we need to go for some fine expansion control: the result is stored *before* the start!

Let us illustrate this idea in a simple model: adding positive integers which may be resulting from the expansion of macros, or may be values of registers. Assume that one number, say, 12345, has already been found, and that we want to parse the next number. The current status of the code may look as follows.

```
\exp_after:wN \add:ww \int_value:w 12345 \exp_after:wN ;
\exp:w \operand:w <stuff>
```

One step of expansion expands `\exp_after:wN`, which triggers the primitive `\int_value:w`, which reads the five digits we have already found, 12345. This integer is unfinished, causing the second `\exp_after:wN` to expand, and to trigger the construction `\exp:w`, which expands `\operand:w`, defined to read what follows and make a number out of it, then leave `\exp_end:`, the number, and a semicolon in the input stream. Once `\operand:w` is done expanding, we obtain essentially

```
\exp_after:wN \add:ww \int_value:w 12345 ;
\exp:w \exp_end: 333444 ;
```

where in fact `\exp_after:wN` has already been expanded, `\int_value:w` has already seen 12345, and `\exp:w` is still looking for a number. It finds `\exp_end:`, hence expands to nothing. Now, `\int_value:w` sees the `;`, which cannot be part of a number. The expansion stops, and we are left with

```
\add:ww 12345 ; 333444 ;
```

which can safely perform the addition by grabbing two arguments delimited by `;`.

If we were to continue parsing the expression, then the following number should also be cleaned up before the next use of a binary operation such as `\add:ww`. Just like `\int_value:w 12345 \exp_after:wN ;` expanded what follows once, we need `\add:ww` to do the calculation, and in the process to expand the following once. This is also true in our real application: all the functions of the form `\_fp_\dots_o:ww` expand what follows once. This comes at the cost of leaving tokens in the input stack, and we need to be careful not to waste this memory. All of our discussion above is nice but simplistic, as operations should not simply be performed in the order they appear.

### 69.1.2 Precedence and infix operators

The various operators we will encounter have different precedences, which influence the order of calculations:  $1 + 2 \times 3 = 1 + (2 \times 3)$  because  $\times$  has a higher precedence than  $+$ . The true analog of our macro `\operand:w` must thus take care of that. When looking for an operand, it needs to perform calculations until reaching an operator which has lower precedence than the one which called `\operand:w`. This means that `\operand:w` must know what the previous binary operator is, or rather, its precedence: we thus rename it `\operand:Nw`. Let us describe as an example how we plan to do the calculation  $41 - 2^3 * 4 + 5$ . More precisely we describe how to perform the first operation in this expression. Here, we abuse notations: the first argument of `\operand:Nw` should be an integer

constant (`\c_fp_prec_plus_int`, ...) equal to the precedence of the given operator, not directly the operator itself.

- Clean up 41 and find `-`. We call `\operand:Nw -` to find the second operand.
- Clean up 2 and find `^`.
- Compare the precedences of `-` and `^`. Since the latter is higher, we need to compute the exponentiation. For this, find the second operand with a nested call to `\operand:Nw ^`.
- Clean up 3 and find `*`.
- Compare the precedences of `^` and `*`. Since the former is higher, `\operand:Nw ^` has found the second operand of the exponentiation, which is computed:  $2^3 = 8$ .
- We now have  $41 - 8 * 4 + 5$ , and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 8?
- Compare the precedences of `-` and `*`. Since the latter is higher, we are not done with 8. Call `\operand:Nw *` to find the second operand of the multiplication.
- Clean up 4, and find `+`.
- Compare the precedences of `*` and `+`. Since the former is higher, `\operand:Nw *` has found the second operand of the multiplication, which is computed:  $8 * 4 = 32$ .
- We now have  $41 - 32 + 5$ , and `\operand:Nw -` is still looking for a second operand for the subtraction. Is it 32?
- Compare the precedences of `-` and `+`. Since they are equal, `\operand:Nw -` has found the second operand for the subtraction, which is computed:  $41 - 32 = 9$ .
- We now have  $9 + 5$ .

The procedure above stops short of performing all computations, but adding a surrounding call to `\operand:Nw` with a very low precedence ensures that all computations are performed before `\operand:Nw` is done. Adding a trailing marker with the same very low precedence prevents the surrounding `\operand:Nw` from going beyond the marker.

The pattern above to find an operand for a given operator, is to find one number and the next operator, then compare precedences to know if the next computation should be done. If it should, then perform it after finding its second operand, and look at the next operator, then compare precedences to know if the next computation should be done. This continues until we find that the next computation should not be done. Then, we stop.

We are now ready to get a bit more technical and describe which of the `l3fp-parse` functions correspond to each step above.

First, `\_fp_parse_operand:Nw` is the `\operand:Nw` function above, with small modifications due to expansion issues discussed later. We denote by `<precedence>` the argument of `\_fp_parse_operand:Nw`, that is, the precedence of the binary operator whose operand we are trying to find. The basic action is to read numbers from the input stream. This is done by `\_fp_parse_one:Nw`. A first approximation of this function is that it reads one `<number>`, performing no computation, and finds the following binary `<operator>`. Then it expands to

```

⟨number⟩
\__fp_parse_infix_⟨operator⟩:N ⟨precedence⟩

```

expanding the `infix` auxiliary before leaving the above in the input stream.

We now explain the `infix` auxiliaries. We need some flexibility in how we treat the case of equal precedences: most often, the first operation encountered should be performed, such as  $1-2-3$  being computed as  $(1-2)-3$ , but  $2^3^4$  should be evaluated as  $2^{(3^4)}$  instead. For this reason, and to support the equivalence between `**` and `^` more easily, each binary operator is converted to a control sequence `\__fp_parse_infix_⟨operator⟩:N` when it is encountered for the first time. Instead of passing both precedences to a test function to do the comparison steps above, we pass the `⟨precedence⟩` (of the earlier operator) to the `infix` auxiliary for the following `⟨operator⟩`, to know whether to perform the computation of the `⟨operator⟩`. If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \__fp_parse_infix_⟨operator⟩:N
```

and otherwise it calls `\__fp_parse_operand:Nw` with the precedence of the `⟨operator⟩` to find its second operand `⟨number₂⟩` and the next `⟨operator₂⟩`, and expands to

```

@ \__fp_parse_apply_binary:NwNwN
  ⟨operator⟩ ⟨number₂⟩
@ \__fp_parse_infix_⟨operator₂⟩:N

```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand `⟨number⟩` is before the `infix` function in the input stream. This is why we stop the expansion here and give control to another function to close the loop.

A definition of `\__fp_parse_operand:Nw ⟨precedence⟩` with some of the expansion control removed is

```

\exp_after:wN \__fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
\__fp_parse_one:Nw ⟨precedence⟩

```

This expands `\__fp_parse_one:Nw ⟨precedence⟩` completely, which finds a number, wraps the next `⟨operator⟩` into an `infix` function, feeds this function the `⟨precedence⟩`, and expands it, yielding either

```

\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\use_none:n \__fp_parse_infix_⟨operator⟩:N

```

or

```

\__fp_parse_continue:NwN ⟨precedence⟩
⟨number⟩ @
\__fp_parse_apply_binary:NwNwN
  ⟨operator⟩ ⟨number₂⟩
@ \__fp_parse_infix_⟨operator₂⟩:N

```

The definition of `\__fp_parse_continue:NwN` is then very simple:

```
\cs_new:Npn \__fp_parse_continue:NwN #1#2#3 { #3 #1 #2 @ }
```

In the first case, #3 is `\use_none:n`, yielding

```
\use_none:n <precedence> <number> @
\__fp_parse_infix_<operator>:N
```

then  $\langle \text{number} \rangle @ \_\_\text{fp\_parse\_infix\_}\langle \text{operator} \rangle : N$ . In the second case, #3 is `\__fp_parse_apply_binary:NwNwN`, whose role is to compute  $\langle \text{number} \rangle \langle \text{operator} \rangle \langle \text{number}_2 \rangle$  and to prepare for the next comparison of precedences: first we get

```
\_\_\text{fp\_parse\_apply\_binary}:NwNwN
  <precedence> <number> @
  <operator> <number>
@ \_\_\text{fp\_parse\_infix\_}\langle \text{operator}_2 \rangle : N
```

then

```
\exp_after:wN \_\_\text{fp\_parse\_continue}:NwN
\exp_after:wN <precedence>
\exp:w \exp_end_continue_f:w
\_\_\text{fp\_}\langle \text{operator} \rangle _o:ww <number> <number>
\exp:w \exp_end_continue_f:w
\_\_\text{fp\_parse\_infix\_}\langle \text{operator}_2 \rangle : N <precedence>
```

where `\_\_\text{fp\_}\langle \text{operator} \rangle _o:ww` computes  $\langle \text{number} \rangle \langle \text{operator} \rangle \langle \text{number}_2 \rangle$  and expands after the result, thus triggers the comparison of the precedence of the  $\langle \text{operator}_2 \rangle$  and the  $\langle \text{precedence} \rangle$ , continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

### 69.1.3 Prefix operators, parentheses, and functions

Prefix operators (unary `-`, `+`, `!`) and parentheses are taken care of by the same mechanism, and functions (`sin`, `exp`, etc.) as well. Finding the argument of the unary `-`, for instance, is very similar to grabbing the second operand of a binary infix operator, with a subtle precedence explained below. Once that operand is found, the operator can be applied to it (for the unary `-`, this simply flips the sign). A left parenthesis is just a prefix operator with a very low precedence equal to that of the closing parenthesis (which is treated as an infix operator, since it normally appears just after numbers), so that all computations are performed until the closing parenthesis. The prefix operator associated to the left parenthesis does not alter its argument, but it removes the closing parenthesis (with some checks).

Prefix operators are the reason why we only summarily described the function `\_\_\text{fp\_parse\_one}:Nw` earlier. This function is responsible for reading in the input stream the first possible  $\langle \text{number} \rangle$  and the next infix  $\langle \text{operator} \rangle$ . If what follows `\_\_\text{fp\_parse\_one}:Nw`  $\langle \text{precedence} \rangle$  is a prefix operator, then we must find the operand of this prefix operator through a nested call to `\_\_\text{fp\_parse\_operand}:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `\_\_\text{fp\_parse\_one}:Nw`. So far, all is simple.

The unary operators `+`, `-`, `!` complicate things a little bit:  $-3^{**2}$  should be  $-(3^2) = -9$ , and not  $(-3)^2 = 9$ . This would easily be done by giving `-` a lower precedence, equal to that of the infix `+` and `-`. Unfortunately, this fails in cases such as  $3^{**-2*4}$ , yielding  $3^{-2\times 4}$  instead of the correct  $3^{-2} \times 4$ . A second attempt would be to call `\_\_\text{fp\_parse\_operand}:Nw` with the  $\langle \text{precedence} \rangle$  of the previous operator, but `0>-2+3` is then

parsed as  $0 > -(2+3)$ : the addition is performed because it binds more tightly than the comparision which precedes  $-$ . The correct approach is for a unary  $-$  to perform operations whose precedence is greater than both that of the previous operation, and that of the unary  $-$  itself. The unary  $-$  is given a precedence higher than multiplication and division. This does not lead to any surprising result, since  $-(x/y) = (-x)/y$  and similarly for multiplication, and it reduces the number of nested calls to `\_fp_parse_operand:Nw`.

Functions are implemented as prefix operators with very high precedence, so that their argument is the first number that can possibly be built.

Note that contrarily to the `infix` functions discussed earlier, the `prefix` functions do perform tests on the previous *(precedence)* to decide whether to find an argument or not, since we know that we need a number, and must never stop there.

#### 69.1.4 Numbers and reading tokens one by one

So far, we have glossed over one important point: what is a “number”? A number is typically given in the form  $\langle\text{significand}\rangle e \langle\text{exponent}\rangle$ , where the  $\langle\text{significand}\rangle$  is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “ $e \langle\text{exponent}\rangle$ ” is optional and is composed of an exponent mark `e` followed by a possibly empty string of signs `+` or `-` and a non-empty string of decimal digits. The  $\langle\text{significand}\rangle$  can also be an integer, dimension, skip, or muskip variable, in which case dimensions are converted from points (or mu units) to floating points, and the  $\langle\text{exponent}\rangle$  can also be an integer variable. Numbers can also be given as floating point variables, or as named constants such as `nan`, `inf` or `pi`. We may add more types in the future.

When `\_fp_parse_one:Nw` is looking for a “number”, here is what happens.

- If the next token is a control sequence with the meaning of `\scan_stop:`, it can be: `\s_fp`, in which case our job is done, as what follows is an internal floating point number, or `\s_fp_expr_mark`, in which case the expression has come to an early end, as we are still looking for a number here, or something else, in which case we consider the control sequence to be a bad variable resulting from c-expansion.
- If the next token is a control sequence with a different meaning, we assume that it is a register, unpack it with `\tex_the:D`, and use its value (in `pt` for dimensions and skips, `mu` for muskips) as the  $\langle\text{significand}\rangle$  of a number: we look for an exponent.
- If the next token is a digit, we remove any leading zeros, then read a significand larger than 1 if the next character is a digit, read a significand smaller than 1 if the next character is a period, or we have found a significand equal to 0 otherwise, and look for an exponent.
- If the next token is a letter, we collect more letters until the first non-letter: the resulting word may denote a function such as `asin`, a constant such as `pi` or be unknown. In the first case, we call `\_fp_parse_operand:Nw` to find the argument of the function, then apply the function, before declaring that we are done. Otherwise, we are done, either with the value of the constant, or with the value `nan` for unknown words.
- If the next token is anything else, we check whether it is a known prefix operator, in which case `\_fp_parse_operand:Nw` finds its operand. If it is not known, then either a number is missing (if the token is a known infix operator) or the token is simply invalid in floating point expressions.

Once a number is found, `\__fp_parse_one:Nw` also finds an infix operator. This goes as follows.

- If the next token is a control sequence, it could be the special marker `\s_--fp_expr_mark`, and otherwise it is a case of juxtaposing numbers, such as `2\c_zero_int`, with an implied multiplication.
- If the next token is a letter, it is also a case of juxtaposition, as letters cannot be proper infix operators.
- Otherwise (including in the case of digits), if the token is a known infix operator, the appropriate `\__fp_infix_{operator}:N` function is built, and if it does not exist, we complain. In particular, the juxtaposition `\c_zero_int 2` is disallowed.

In the above, we need to test whether a character token `#1` is a digit:

```
\if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
  is a digit
\else:
  not a digit
\fi:
```

To exclude 0, replace 9 by 10. The use of `\token_to_str:N` ensures that a digit with any catcode is detected. To test if a character token is a letter, we need to work with its character code, testing if ‘#1 lies in [65, 90] (uppercase letters) or [97, 112] (lowercase letters)

```
\if_int_compare:w \__fp_int_eval:w
  ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26 = 3 \exp_stop_f:
  is a letter
\else:
  not a letter
\fi:
```

At all steps, we try to accept all category codes: when `#1` is kept to be used later, it is almost always converted to category code other through `\token_to_str:N`. More precisely, catcodes {3, 6, 7, 8, 11, 12} should work without trouble, but not {1, 2, 4, 10, 13}, and of course {0, 5, 9} cannot become tokens.

Floating point expressions should behave as much as possible like  $\varepsilon$ - $\text{\TeX}$ -based integer expressions and dimension expressions. In particular, f-expansion should be performed as the expression is read, token by token, forcing the expansion of protected macros, and ignoring spaces. One advantage of expanding at every step is that restricted expandable functions can then be used in floating point expressions just as they can be in other kinds of expressions. Problematically, spaces stop f-expansion: for instance, the macro `\X` below would not be expanded if we simply performed f-expansion.

```
\DeclareDocumentCommand {\test} {m} { \fp_eval:n {#1} }
\ExplSyntaxOff
\test { 1 + \X }
```

Of course, spaces typically do not appear in a code setting, but may very easily come in document-level input, from which some expressions may come. To avoid this problem, at every step, we do essentially what `\use:f` would do: take an argument, put it back

in the input stream, then **f**-expand it. This is not a complete solution, since a macro's expansion could contain leading spaces which would stop the **f**-expansion before further macro calls are performed. However, in practice it should be enough: in particular, floating point numbers are correctly expanded to the underlying `\s_fp ...` structure. The **f**-expansion is performed by `\_fp_parse_expand:w`.

## 69.2 Main auxiliary functions

`\_fp_parse_operand:Nw`

`\exp:w \_fp_parse_operand:Nw <precedence> \_fp_parse_expand:w`

Reads the “...”, performing every computation with a precedence higher than `<precedence>`, then expands to

`<result> @ \_fp_parse_infix_<operation>:N ...`

where the `<operation>` is the first operation with a lower precedence, possibly `end`, and the “...” start just after the `<operation>`.

(End definition for `\_fp_parse_operand:Nw`.)

`\_fp_parse_infix_+:N`

`\_fp_parse_infix_+:N <precedence> ...`

If `+` has a precedence higher than the `<precedence>`, cleans up a second `<operand>` and finds the `<operation>2` which follows, and expands to

`@ \_fp_parse_apply_binary:NwNwN + <operand> @ \_fp_parse_infix_<operation>2:N ...`

Otherwise expands to

`@ \use_none:n \_fp_parse_infix_+:N ...`

A similar function exists for each infix operator.

(End definition for `\_fp_parse_infix_+:N`.)

`\_fp_parse_one:Nw`

`\_fp_parse_one:Nw <precedence> ...`

Cleans up one or two operands depending on how the precedence of the next operation compares to the `<precedence>`. If the following `<operation>` has a precedence higher than `<precedence>`, expands to

`<operand>1 @ \_fp_parse_apply_binary:NwNwN <operation> <operand>2 @ \_fp_parse_infix_<operation>2:N ...`

and otherwise expands to

`<operand> @ \use_none:n \_fp_parse_infix_<operation>:N ...`

(End definition for `\_fp_parse_one:Nw`.)

## 69.3 Helpers

\\_fp\_parse\_expand:w \exp:w \\_fp\_parse\_expand:w ⟨tokens⟩

This function must always come within a \exp:w expansion. The ⟨tokens⟩ should be the part of the expression that we have not yet read. This requires in particular closing all conditionals properly before expanding.

23400 \cs\_new:Npn \\_fp\_parse\_expand:w #1 { \exp\_end\_continue\_f:w #1 }

(End definition for \\_fp\_parse\_expand:w.)

\\_fp\_parse\_return\_semicolon:w This very odd function swaps its position with the following \fi: and removes \\_fp\_parse\_expand:w normally responsible for expansion. That turns out to be useful.

23401 \cs\_new:Npn \\_fp\_parse\_return\_semicolon:w  
23402 #1 \fi: \\_fp\_parse\_expand:w { \fi: ; #1 }

(End definition for \\_fp\_parse\_return\_semicolon:w.)

\\_fp\_parse\_digits\_vii:N \\_fp\_parse\_digits\_vi:N \\_fp\_parse\_digits\_v:N \\_fp\_parse\_digits\_iv:N \\_fp\_parse\_digits\_iii:N \\_fp\_parse\_digits\_ii:N \\_fp\_parse\_digits\_i:N \\_fp\_parse\_digits\_:N These functions must be called within an \int\_value:w or \\_fp\_int\_eval:w construction. The first token which follows must be f-expanded prior to calling those functions. The functions read tokens one by one, and output digits into the input stream, until meeting a non-digit, or up to a number of digits equal to their index. The full expansion is

*(digits)* ; *(filling 0)* ; *(length)*

where *(filling 0)* is a string of zeros such that *(digits)* *(filling 0)* has the length given by the index of the function, and *(length)* is the number of zeros in the *(filling 0)* string. Each function puts a digit into the input stream and calls the next function, until we find a non-digit. We are careful to pass the tested tokens through \token\_to\_str:N to normalize their category code.

23403 \cs\_set\_protected:Npn \\_fp\_tmp:w #1 #2 #3  
23404 {  
23405 \cs\_new:cpn { \_fp\_parse\_digits\_ #1 :N } ##1  
23406 {  
23407 \if\_int\_compare:w 9 < 1 \token\_to\_str:N ##1 \exp\_stop\_f:  
23408 \token\_to\_str:N ##1 \exp\_after:wN #2 \exp:w  
23409 \else:  
23410 \\_fp\_parse\_return\_semicolon:w #3 ##1  
23411 \fi:  
23412 \\_fp\_parse\_expand:w  
23413 }  
23414 }  
23415 \\_fp\_tmp:w {vii} \\_fp\_parse\_digits\_vi:N { 0000000 ; 7 }  
23416 \\_fp\_tmp:w {vi} \\_fp\_parse\_digits\_v:N { 000000 ; 6 }  
23417 \\_fp\_tmp:w {v} \\_fp\_parse\_digits\_iv:N { 00000 ; 5 }  
23418 \\_fp\_tmp:w {iv} \\_fp\_parse\_digits\_iii:N { 0000 ; 4 }  
23419 \\_fp\_tmp:w {iii} \\_fp\_parse\_digits\_ii:N { 000 ; 3 }  
23420 \\_fp\_tmp:w {ii} \\_fp\_parse\_digits\_i:N { 00 ; 2 }  
23421 \\_fp\_tmp:w {i} \\_fp\_parse\_digits\_:N { 0 ; 1 }  
23422 \cs\_new:Npn \\_fp\_parse\_digits\_:N { ; 0 }

(End definition for \\_fp\_parse\_digits\_vii:N and others.)

## 69.4 Parsing one number

\\_fp\_parse\_one:Nw

This function finds one number, and packs the symbol which follows in an \\_fp\_parse\_infix\_\\_... csname. #1 is the previous *(precedence)*, and #2 the first token of the operand. We distinguish four cases: #2 is equal to \scan\_stop: in meaning, #2 is a different control sequence, #2 is a digit, and #2 is something else (this last case is split further later). Despite the earlier f-expansion, #2 may still be expandable if it was protected by \exp\_not:N, as may happen with the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  command \protect. Using a well placed \reverse\_if:N, this case is sent to \\_fp\_parse\_one\_fp:NN which deals with it robustly.

```

23423 \cs_new:Npn \_fp_parse_one:Nw #1 #2
23424 {
23425   \if_catcode:w \scan_stop: \exp_not:N #2
23426     \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
23427       \exp_after:wN \reverse_if:N
23428     \fi:
23429     \if_meaning:w \scan_stop: #2
23430       \exp_after:wN \exp_after:wN
23431       \exp_after:wN \_fp_parse_one_fp:NN
23432     \else:
23433       \exp_after:wN \exp_after:wN
23434       \exp_after:wN \_fp_parse_one_register:NN
23435     \fi:
23436   \else:
23437     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
23438       \exp_after:wN \exp_after:wN
23439       \exp_after:wN \_fp_parse_one_digit:NN
23440     \else:
23441       \exp_after:wN \exp_after:wN
23442       \exp_after:wN \_fp_parse_one_other:NN
23443     \fi:
23444   \fi:
23445   #1 #2
23446 }
```

(End definition for \\_fp\_parse\_one:Nw.)

\\_fp\_parse\_one\_fp:NN  
\fp\_exp\_after\_expr\_mark\_f:nw  
\\_fp\_exp\_after\_?\_f:nw

This function receives a *(precedence)* and a control sequence equal to \scan\_stop: in meaning. There are three cases.

- \sfp starts a floating point number, and we call \fp\_exp\_after\_f:nw, which f-expands after the floating point.
- \sfp\_expr\_mark is a premature end, we call \fp\_exp\_after\_expr\_mark\_f:nw, which triggers an fp-early-end error.
- For a control sequence not containing \sfp, we call \fp\_exp\_after\_?\_f:nw, causing a bad-variable error.

This scheme is extensible: additional types can be added by starting the variables with a scan mark of the form \sfp\_{type} and defining \fp\_exp\_after\_{type}\_f:nw. In all cases, we make sure that the second argument of \fp\_parse\_infix:NN is correctly expanded. A special case only enabled in L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  is that if \protect is encountered then

the error message mentions the control sequence which follows it rather than `\protect` itself. The test for L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\epsilon$</sub>  uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop`: hence “does not exist”.

```

23447 \cs_new:Npn \__fp_parse_one_fp:NN #1
23448 {
23449   \__fp_exp_after_any_f:nw
23450   {
23451     \exp_after:wN \__fp_parse_infix:NN
23452     \exp_after:wN #1 \exp:w \__fp_parse_expand:w
23453   }
23454 }
23455 \cs_new:Npn \__fp_exp_after_expr_mark_f:nw #1
23456 {
23457   \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
23458   {
23459     \c__fp_prec_comma_int { }
23460     \c__fp_prec_tuple_int { }
23461     \c__fp_prec_end_int
23462     {
23463       \exp_after:wN \c__fp_empty_tuple_fp
23464       \exp:w \exp_end_continue_f:w
23465     }
23466   }
23467   {
23468     \msg_expandable_error:nn { fp } { early-end }
23469     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
23470   }
23471   #1
23472 }
23473 \cs_new:cpn { __fp_exp_after_?_f:nw } #1#2
23474 {
23475   \msg_expandable_error:nnn { kernel } { bad-variable }
23476   {##2}
23477   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
23478 }
23479 \cs_set_protected:Npn \__fp_tmp:w #1
23480 {
23481   \cs_if_exist:NT #1
23482   {
23483     \cs_gset:cpn { __fp_exp_after_?_f:nw } ##1##2
23484     {
23485       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1
23486       \str_if_eq:nnTF {##2} { \protect }
23487       {
23488         \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
23489         {
23490           \msg_expandable_error:nnn { fp }
23491             { robust-cmd }
23492         }
23493       }
23494       {
23495         \msg_expandable_error:nnn { kernel }
23496           { bad-variable } {##2}
23497       }
23498     }
23499   }

```

```

23498         }
23499     }
23500   }
23501 \exp_args:Nc \__fp_tmp:w { @unexpandable@protect }

(End definition for \__fp_parse_one_fp:NN, \__fp_exp_after_expr_mark_f:nw, and \__fp_exp_after_-
?_f:nw.)

```

\\_\_fp\_parse\_one\_register:NN  
  \\_\_fp\_parse\_one\_register\_aux:Nw  
  \\_\_fp\_parse\_one\_register\_auxii:wwwNw  
  \\_\_fp\_parse\_one\_register\_int:www  
  \\_\_fp\_parse\_one\_register\_mu:www  
  \\_\_fp\_parse\_one\_register\_dim:ww

This is called whenever #2 is a control sequence other than `\scan_stop:` in meaning. We special-case `\wd`, `\ht`, `\dp` (see later) and otherwise assume that it is a register, but carefully unpack it with `\tex_the:D` within braces. First, we find the exponent following #2. Then we unpack #2 with `\tex_the:D`, and the `auxii` auxiliary distinguishes integer registers from dimensions/skips from muskips, according to the presence of a period and/or of `pt`. For integers, simply convert `\langle value \rangle e \langle exponent \rangle` to a floating point number with `\__fp_parse:n` (this is somewhat wasteful). For other registers, the decimal rounding provided by TeX does not accurately represent the binary value that it manipulates, so we extract this binary value as a number of scaled points with `\int_value:w \dim_to_decimal_in_sp:n { \langle decimal value \rangle pt }`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by  $2^{-16}$ , correctly rounded.

```

23502 \cs_new:Npn \__fp_parse_one_register:NN #1#2
23503   {
23504     \exp_after:wN \__fp_parse_infix_after_operand:NwN
23505     \exp_after:wN #1
23506     \exp:w \exp_end_continue_f:w
23507       \__fp_parse_one_register_special:N #2
23508       \exp_after:wN \__fp_parse_one_register_aux:Nw
23509       \exp_after:wN #2
23510       \int_value:w
23511         \exp_after:wN \__fp_parse_exponent:N
23512         \exp:w \__fp_parse_expand:w
23513   }
23514 \cs_new:Npx \__fp_parse_one_register_aux:Nw #1
23515   {
23516     \exp_not:n
23517   {
23518     \exp_after:wN \use:nn
23519     \exp_after:wN \__fp_parse_one_register_auxii:wwwNw
23520   }
23521   \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
23522     ; \exp_not:N \__fp_parse_one_register_dim:ww
23523     \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_mu:www
23524     . \tl_to_str:n { pt } ; \exp_not:N \__fp_parse_one_register_int:www
23525     \s__fp_stop
23526   }
23527 \exp_args:Nno \use:nn
23528   { \cs_new:Npn \__fp_parse_one_register_auxii:wwwNw #1 . #2 }
23529     { \tl_to_str:n { pt } #3 ; #4#5 \s__fp_stop }
23530     { #4 #1.#2; }
23531 \exp_args:Nno \use:nn
23532   { \cs_new:Npn \__fp_parse_one_register_mu:www #1 }
23533     { \tl_to_str:n { mu } ; #2 ; }
23534     { \__fp_parse_one_register_dim:ww #1 ; }
23535 \cs_new:Npn \__fp_parse_one_register_int:www #1; #2.; #3;
23536   { \__fp_parse:n { #1 e #3 } }

```

```

23537 \cs_new:Npn \__fp_parse_one_register_dim:ww #1; #2;
23538 {
23539     \exp_after:wN \__fp_from_dim_test:ww
23540     \int_value:w #2 \exp_after:wN ,
23541     \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
23542 }

```

(End definition for `\__fp_parse_one_register>NN` and others.)

```

\__fp_parse_one_register_special:N
\__fp_parse_one_register_math:NNw
    \__fp_parse_one_register_wd:w
\__fp_parse_one_register_wd:Nw

```

The `\wd`, `\dp`, `\ht` primitives expect an integer argument. We abuse the exponent parser to find the integer argument: simply include the exponent marker `e`. Once that “exponent” is found, use `\tex_the:D` to find the box dimension and then copy what we did for dimensions.

```

23543 \cs_new:Npn \__fp_parse_one_register_special:N #1
23544 {
23545     \if_meaning:w \box_wd:N #1 \__fp_parse_one_register_wd:w \fi:
23546     \if_meaning:w \box_ht:N #1 \__fp_parse_one_register_wd:w \fi:
23547     \if_meaning:w \box_dp:N #1 \__fp_parse_one_register_wd:w \fi:
23548     \if_meaning:w \infty #1
23549         \__fp_parse_one_register_math:NNw \infty #1
23550     \fi:
23551     \if_meaning:w \pi #1
23552         \__fp_parse_one_register_math:NNw \pi #1
23553     \fi:
23554 }
23555 \cs_new:Npn \__fp_parse_one_register_math:NNw
23556     #1#2#3#4 \__fp_parse_expand:w
23557 {
23558     #3
23559     \str_if_eq:nnTF {#1} {#2}
23560     {
23561         \msg_expandable_error:nnn
23562             { fp } { infinity-pi } {#1}
23563         \c_nan_fp
23564     }
23565     { #4 \__fp_parse_expand:w }
23566 }
23567 \cs_new:Npn \__fp_parse_one_register_wd:w
23568     #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
23569 {
23570     #1
23571     \exp_after:wN \__fp_parse_one_register_wd:Nw
23572     #4 \__fp_parse_expand:w e
23573 }
23574 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
23575 {
23576     \exp_after:wN \__fp_from_dim_test:ww
23577     \exp_after:wN 0 \exp_after:wN ,
23578     \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
23579 }

```

(End definition for `\__fp_parse_one_register_special:N` and others.)

```
\__fp_parse_one_digit:NN
```

A digit marks the beginning of an explicit floating point number. Once the number is found, we catch the case of overflow and underflow with `\__fp_sanitize:wN`,

then `\__fp_parse_infix_after_operand:NwN` expands `\__fp_parse_infix>NN` after the number we find, to wrap the following infix operator as required. Finding the number itself begins by removing leading zeros: further steps are described later.

```

23580 \cs_new:Npn \__fp_parse_one_digit:NN #1
23581 {
23582     \exp_after:wN \__fp_parse_infix_after_operand:NwN
23583     \exp_after:wN #1
23584     \exp:w \exp_end_continue_f:w
23585     \exp_after:wN \__fp_SANITIZE:wN
23586     \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
23587 }
```

*(End definition for `\__fp_parse_one_digit:NN`.)*

`\__fp_parse_one_other:NN` For this function, #2 is a character token which is not a digit. If it is an ASCII letter, `\__fp_parse_letters:N` beyond this one and give the result to `\__fp_parse_word:Nw`. Otherwise, the character is assumed to be a prefix operator, and we build `\__fp_parse_prefix_<operator>:Nw`.

```

23588 \cs_new:Npn \__fp_parse_one_other:NN #1 #2
23589 {
23590     \if_int_compare:w
23591         \__fp_int_eval:w
23592             ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
23593             = 3 \exp_stop_f:
23594             \exp_after:wN \__fp_parse_word:Nw
23595             \exp_after:wN #1
23596             \exp_after:wN #2
23597             \exp:w \exp_after:wN \__fp_parse_letters:N
23598             \exp:w
23599     \else:
23600         \exp_after:wN \__fp_parse_prefix:NNN
23601         \exp_after:wN #1
23602         \exp_after:wN #2
23603         \cs:w
23604             _fp_parse_prefix_ \token_to_str:N #2 :Nw
23605             \exp_after:wN
23606             \cs_end:
23607             \exp:w
23608             \fi:
23609             \__fp_parse_expand:w
23610 }
```

*(End definition for `\__fp_parse_one_other:NN`.)*

`\__fp_parse_word:Nw`  
`\__fp_parse_letters:N` Finding letters is a simple recursion. Once `\__fp_parse_letters:N` has done its job, we try to build a control sequence from the word #2. If it is a known word, then the corresponding action is taken, and otherwise, we complain about an unknown word, yield `\c_nan_fp`, and look for the following infix operator. Note that the unknown word could be a mistyped function as well as a mistyped constant, so there is no way to tell whether to look for arguments; we do not. The standard requires “inf” and “infinity” and “nan” to be recognized regardless of case, but we probably don’t want to allow every l3fp word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

23611 \cs_new:Npn \__fp_parse_word:Nw #1#2;
23612 {
23613     \cs_if_exist_use:cF { __fp_parse_word_#2:N }
23614     {
23615         \cs_if_exist_use:cF
23616             { __fp_parse_caseless_ \str_casefold:n {#2} :N }
23617             {
23618                 \msg_expandable_error:nnn
23619                     { fp } { unknown-fp-word } {#2}
23620                     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
23621                     \__fp_parse_infix:NN
23622             }
23623     }
23624     #1
23625 }
23626 \cs_new:Npn \__fp_parse_letters:N #1
23627 {
23628     \exp_end_continue_f:w
23629     \if_int_compare:w
23630         \if_catcode:w \scan_stop: \exp_not:N #1
23631         0
23632     \else:
23633         \__fp_int_eval:w
23634             ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26
23635             \fi:
23636             = 3 \exp_stop_f:
23637             \exp_after:wN #1
23638             \exp:w \exp_after:wN \__fp_parse_letters:N
23639             \exp:w
23640     \else:
23641         \__fp_parse_return_semicolon:w #1
23642         \fi:
23643         \__fp_parse_expand:w
23644 }

```

(End definition for `\__fp_parse_word:Nw` and `\__fp_parse_letters:N`.)

`\__fp_parse_prefix:NNN`  
`\__fp_parse_prefix_unknown:NNN`

For this function, #1 is the previous *precedence*, #2 is the operator just seen, and #3 is a control sequence which implements the operator if it is a known operator. If this control sequence is `\scan_stop:`, then the operator is in fact unknown. Either the expression is missing a number there (if the operator is valid as an infix operator), and we put `nan`, wrapping the infix operator in a csname as appropriate, or the character is simply invalid in floating point expressions, and we continue looking for a number, starting again from `\__fp_parse_one:Nw`.

```

23645 \cs_new:Npn \__fp_parse_prefix:NNN #1#2#3
23646 {
23647     \if_meaning:w \scan_stop: #3
23648         \exp_after:wN \__fp_parse_prefix_unknown:NNN
23649         \exp_after:wN #2
23650         \fi:
23651         #3 #1
23652     }
23653 \cs_new:Npn \__fp_parse_prefix_unknown:NNN #1#2#3
23654 {

```

```

23655   \cs_if_exist:cTF { __fp_parse_infix_ \token_to_str:N #1 :N }
23656   {
23657     \msg_expandable_error:nnn
23658       { fp } { missing-number } {#1}
23659     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
23660     \__fp_parse_infix:NN #3 #1
23661   }
23662   {
23663     \msg_expandable_error:nnn
23664       { fp } { unknown-symbol } {#1}
23665     \__fp_parse_one:Nw #3
23666   }
23667 }

```

(End definition for `\__fp_parse_prefix:NNN` and `\__fp_parse_prefix_unknown:NNN`.)

#### 69.4.1 Numbers: trimming leading zeros

Numbers are parsed as follows: first we trim leading zeros, then if the next character is a digit, start reading a significand  $\geq 1$  with the set of functions `\__fp_parse_large...`; if it is a period, the significand is  $< 1$ ; and otherwise it is zero. In the second case, trim additional zeros after the period, counting them for an exponent shift  $\langle exp_1 \rangle < 0$ , then read the significand with the set of functions `\__fp_parse_small...`. Once the significand is read, read the exponent if `e` is present.

`\__fp_parse_trim_zeros:N` This function expects an already expanded token. It removes any leading zero, then distinguishes three cases: if the first non-zero token is a digit, then call `\__fp_parse_large:N` (the significand is  $\geq 1$ ); if it is `.`, then continue trimming zeros with `\__fp_parse_strim_zeros:N`; otherwise, our number is exactly zero, and we call `\__fp_parse_zero:` to take care of that case.

```

23668 \cs_new:Npn \__fp_parse_trim_zeros:N #1
23669   {
23670     \if:w 0 \exp_not:N #1
23671       \exp_after:wN \__fp_parse_trim_zeros:N
23672       \exp:w
23673     \else:
23674       \if:w . \exp_not:N #1
23675         \exp_after:wN \__fp_parse_strim_zeros:N
23676         \exp:w
23677       \else:
23678         \__fp_parse_trim_end:w #1
23679       \fi:
23680     \fi:
23681     \__fp_parse_expand:w
23682   }
23683 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
23684   {
23685     \fi:
23686   \fi:
23687   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23688     \exp_after:wN \__fp_parse_large:N
23689   \else:
23690     \exp_after:wN \__fp_parse_zero:

```

```

23691      \fi:
23692      #1
23693  }

```

(End definition for `\_fp_parse_trim_zeros:N` and `\_fp_parse_trim_end:w`.)

`\_fp_parse_strim_zeros:N` If we have removed all digits until a period (or if the body started with a period), then enter the “`small_trim`” loop which outputs  $-1$  for each removed 0. Those  $-1$  are added to an integer expression waiting for the exponent. If the first non-zero token is a digit, call `\_fp_parse_small:N` (our significand is smaller than 1), and otherwise, the number is an exact zero. The name `strim` stands for “small trim”.

```

23694 \cs_new:Npn \_fp_parse_strim_zeros:N #1
23695  {
23696      \if:w 0 \exp_not:N #1
23697          - 1
23698          \exp_after:wN \_fp_parse_strim_zeros:N \exp:w
23699      \else:
23700          \_fp_parse_strim_end:w #1
23701      \fi:
23702          \_fp_parse_expand:w
23703  }
23704 \cs_new:Npn \_fp_parse_strim_end:w #1 \fi: \_fp_parse_expand:w
23705  {
23706      \fi:
23707      \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23708          \exp_after:wN \_fp_parse_small:N
23709      \else:
23710          \exp_after:wN \_fp_parse_zero:
23711      \fi:
23712          #1
23713  }

```

(End definition for `\_fp_parse_strim_zeros:N` and `\_fp_parse_strim_end:w`.)

`\_fp_parse_zero:` After reading a significand of 0, find any exponent, then put a sign of 1 for `\_fp-sanitize:wN`, which removes everything and leaves an exact zero.

```

23714 \cs_new:Npn \_fp_parse_zero:
23715  {
23716      \exp_after:wN ; \exp_after:wN 1
23717      \int_value:w \_fp_parse_exponent:N
23718  }

```

(End definition for `\_fp_parse_zero:..`)

#### 69.4.2 Number: small significand

`\_fp_parse_small:N` This function is called after we have passed the decimal separator and removed all leading zeros from the significand. It is followed by a non-zero digit (with any catcode). The goal is to read up to 16 digits. But we can’t do that all at once, because `\int_value:w` (which allows us to collect digits and continue expanding) can only go up to 9 digits. Hence we grab digits in two steps of 8 digits. Since `#1` is a digit, read seven more digits using `\_fp_parse_digits_vii:N`. The `small_leading` auxiliary leaves those digits in the `\int_value:w`, and grabs some more, or stops if there are no more digits. Then the

`pack_leading` auxiliary puts the various parts in the appropriate order for the processing further up.

```

23719 \cs_new:Npn \__fp_parse_small:N #1
23720 {
23721   \exp_after:wN \__fp_parse_pack_leading>NNNNNww
23722   \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
23723   \exp_after:wN \__fp_parse_small_leading:wwNN
23724   \int_value:w 1
23725   \exp_after:wN \__fp_parse_digits_vii:N
23726   \exp:w \__fp_parse_expand:w
23727 }
```

(End definition for `\__fp_parse_small:N`.)

`\__fp_parse_small_leading:wwNN`

We leave  $\langle digits \rangle$   $\langle zeros \rangle$  in the input stream: the functions used to grab digits are such that this constitutes digits 1 through 8 of the significand. Then prepare to pack 8 more digits, with an exponent shift of zero (this shift is used in the case of a large significand). If #4 is a digit, leave it behind for the packing function, and read 6 more digits to reach a total of 15 digits: further digits are involved in the rounding. Otherwise put 8 zeros in to complete the significand, then look for an exponent.

```

23728 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
23729 {
23730   #1 #2
23731   \exp_after:wN \__fp_parse_pack_trailing>NNNNNNNww
23732   \exp_after:wN 0
23733   \int_value:w \__fp_int_eval:w 1
23734   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23735     \token_to_str:N #4
23736     \exp_after:wN \__fp_parse_small_trailing:wwNN
23737     \int_value:w 1
23738     \exp_after:wN \__fp_parse_digits_vi:N
23739     \exp:w
23740   \else:
23741     0000 0000 \__fp_parse_exponent:Nw #4
23742   \fi:
23743   \__fp_parse_expand:w
23744 }
```

(End definition for `\__fp_parse_small_leading:wwNN`.)

`\__fp_parse_small_trailing:wwNN`

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the  $\langle next\ token \rangle$  is a digit, it is the 16th digit, we keep it, then the `small_round` auxiliary considers this digit and all further digits to perform the rounding: the function expands to nothing, to +0 or to +1. Otherwise, there is no 16-th digit, so we put a 0, and look for an exponent.

```

23745 \cs_new:Npn \__fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
23746 {
23747   #1 #2
23748   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23749     \token_to_str:N #4
23750     \exp_after:wN \__fp_parse_small_round:NN
```

```

23751     \exp_after:wN #4
23752     \exp:w
23753     \else:
23754     0 \__fp_parse_exponent:Nw #4
23755     \fi:
23756     \__fp_parse_expand:w
23757 }

```

(End definition for `\__fp_parse_small_trailing:wwNN.`)

```

\__fp_parse_pack_trailing:NNNNNNww
\__fp_parse_pack_leading:NNNNNNww
\__fp_parse_pack_carry:w

```

Those functions are expanded after all the digits are found, we took care of the rounding, as well as the exponent. The last argument is the exponent. The previous five arguments are 8 digits which we pack in groups of 4, and the argument before that is 1, except in the rare case where rounding lead to a carry, in which case the argument is 2. The `trailing` function has an exponent shift as its first argument, which we add to the exponent found in the `e...` syntax. If the trailing digits cause a carry, the integer expression for the leading digits is incremented (+1 in the code below). If the leading digits propagate this carry all the way up, the function `\__fp_parse_pack_carry:w` increments the exponent, and changes the significand from `0000...` to `1000...`: this is simple because such a carry can only occur to give rise to a power of 10.

```

23758 \cs_new:Npn \__fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
23759 {
23760     \if_meaning:w 2 #2 + 1 \fi:
23761     ; #8 + #1 ; {#3#4#5#6} {#7};
23762 }
23763 \cs_new:Npn \__fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
23764 {
23765     + #7
23766     \if_meaning:w 2 #1 \__fp_parse_pack_carry:w \fi:
23767     ; 0 {#2#3#4#5} {#6}
23768 }
23769 \cs_new:Npn \__fp_parse_pack_carry:w \fi: ; 0 #1
23770     { \fi: + 1 ; 0 {1000} }

```

(End definition for `\__fp_parse_pack_trailing:NNNNNNww`, `\__fp_parse_pack_leading:NNNNNNww`, and `\__fp_parse_pack_carry:w`.)

#### 69.4.3 Number: large significand

Parsing a significand larger than 1 is a little bit more difficult than parsing small significands. We need to count the number of digits before the decimal separator, and add that to the final exponent. We also need to test for the presence of a dot each time we run out of digits, and branch to the appropriate `parse_small` function in those cases.

```
\__fp_parse_large:N
```

This function is followed by the first non-zero digit of a “large” significand ( $\geq 1$ ). It is called within an integer expression for the exponent. Grab up to 7 more digits, for a total of 8 digits.

```

23771 \cs_new:Npn \__fp_parse_large:N #1
23772 {
23773     \exp_after:wN \__fp_parse_large_leading:wwNN
23774     \int_value:w 1 \token_to_str:N #1
23775     \exp_after:wN \__fp_parse_digits_vii:N
23776     \exp:w \__fp_parse_expand:w
23777 }

```

(End definition for `\_fp_parse_large:N.`)

```
\_fp_parse_large_leading:wwNN
  \_fp_parse_large_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>
    <next token>
```

We shift the exponent by the number of digits in #1, namely the target number, 8, minus the  $\langle \text{number of zeros} \rangle$  (number of digits missing). Then prepare to pack the 8 first digits. If the  $\langle \text{next token} \rangle$  is a digit, read up to 6 more digits (digits 10 to 15). If it is a period, try to grab the end of our 8 first digits, branching to the `small` functions since the number of digit does not affect the exponent anymore. Finally, if this is the end of the significand, insert the  $\langle \text{zeros} \rangle$  to complete the 8 first digits, insert 8 more, and look for an exponent.

```
23778 \cs_new:Npn \_fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
23779 {
23780   + \c_fp_half_prec_int - #3
23781   \exp_after:wN \_fp_parse_pack_leading:NNNNNww
23782   \int_value:w \_fp_int_eval:w 1 #1
23783   \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23784     \exp_after:wN \_fp_parse_large_trailing:wwNN
23785     \int_value:w 1 \token_to_str:N #4
23786     \exp_after:wN \_fp_parse_digits_vi:N
23787     \exp:w
23788   \else:
23789     \if:w . \exp_not:N #4
23790       \exp_after:wN \_fp_parse_small_leading:wwNN
23791       \int_value:w 1
23792       \cs:w
23793         \_fp_parse_digits_
23794         \_fp_int_to_roman:w #3
23795         :N \exp_after:wN
23796         \cs_end:
23797         \exp:w
23798   \else:
23799     #2
23800     \exp_after:wN \_fp_parse_pack_trailing:NNNNNNww
23801     \exp_after:wN 0
23802     \int_value:w 1 0000 0000
23803     \_fp_parse_exponent:Nw #4
23804     \fi:
23805   \fi:
23806   \_fp_parse_expand:w
23807 }
```

(End definition for `\_fp_parse_large_leading:wwNN.`)

```
\_fp_parse_large_trailing:wwNN
  \_fp_parse_large_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
    <next token>
```

We have just read 15 digits. If the  $\langle \text{next token} \rangle$  is a digit, then the exponent shift caused by this block of 8 digits is 8, first argument to the `pack_trailing` function. We keep the  $\langle \text{digits} \rangle$  and this 16-th digit, and find how this should be rounded using `\_fp_parse_large_round>NN`. Otherwise, the exponent shift is the number of  $\langle \text{digits} \rangle$ , 7 minus the  $\langle \text{number of zeros} \rangle$ , and we test for a decimal point. This case happens in 123451234512345.67 with exactly 15 digits before the decimal separator. Then branch to the appropriate `small` auxiliary, grabbing a few more digits to complement the digits

we already grabbed. Finally, if this is truly the end of the significand, look for an exponent after using the `(zeros)` and providing a 16-th digit of 0.

```

23808 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
23809 {
23810     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
23811         \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
23812         \exp_after:wN \c__fp_half_prec_int
23813         \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
23814             \exp_after:wN \__fp_parse_large_round:NN
23815             \exp_after:wN #4
23816             \exp:w
23817     \else:
23818         \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
23819         \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
23820         \int_value:w \__fp_int_eval:w 1 #1
23821             \if:w . \exp_not:N #4
23822                 \exp_after:wN \__fp_parse_small_trailing:wwNN
23823                 \int_value:w 1
23824                     \cs:w
23825                         __fp_parse_digits_
23826                         \__fp_int_to_roman:w #3
23827                         :N \exp_after:wN
23828                         \cs_end:
23829                         \exp:w
23830     \else:
23831         #2 0 \__fp_parse_exponent:Nw #4
23832     \fi:
23833     \fi:
23834     \__fp_parse_expand:w
23835 }
```

*(End definition for `\__fp_parse_large_trailing:wwNN`.)*

#### 69.4.4 Number: beyond 16 digits, rounding

`\__fp_parse_round_loop:N` This loop is called when rounding a number (whether the mantissa is small or large).  
`\__fp_parse_round_up:N` It should appear in an integer expression. This function reads digits one by one, until reaching a non-digit, and adds 1 to the integer expression for each digit. If all digits found are 0, the function ends the expression by ;0, otherwise by ;1. This is done by switching the loop to `round_up` at the first non-zero digit, thus we avoid to test whether digits are 0 or not once we see a first non-zero digit.

```

23836 \cs_new:Npn \__fp_parse_round_loop:N #1
23837 {
23838     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23839         + 1
23840         \if:w 0 \token_to_str:N #1
23841             \exp_after:wN \__fp_parse_round_loop:N
23842             \exp:w
23843         \else:
23844             \exp_after:wN \__fp_parse_round_up:N
23845             \exp:w
23846         \fi:
23847     \else:
```

```

23848      \_fp_parse_return_semicolon:w 0 #1
23849      \fi:
23850      \_fp_parse_expand:w
23851  }
23852 \cs_new:Npn \_fp_parse_round_up:N #1
23853 {
23854     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23855     + 1
23856     \exp_after:wN \_fp_parse_round_up:N
23857     \exp:w
23858 \else:
23859     \_fp_parse_return_semicolon:w 1 #1
23860 \fi:
23861     \_fp_parse_expand:w
23862 }

```

(End definition for `\_fp_parse_round_loop:N` and `\_fp_parse_round_up:N`.)

`\_fp_parse_round_after:wN`

After the loop `\_fp_parse_round_loop:N`, this function fetches an exponent with `\_fp_parse_exponent:N`, and combines it with the number of digits counted by `\_fp_parse_round_loop:N`. At the same time, the result 0 or 1 is added to the surrounding integer expression.

```

23863 \cs_new:Npn \_fp_parse_round_after:wN #1; #2
23864 {
23865     + #2 \exp_after:wN ;
23866     \int_value:w \_fp_int_eval:w #1 + \_fp_parse_exponent:N
23867 }

```

(End definition for `\_fp_parse_round_after:wN`.)

`\_fp_parse_small_round:NN`  
`\_fp_parse_round_after:wN`

Here, `#1` is the digit that we are currently rounding (we only care whether it is even or odd). If `#2` is not a digit, then fetch an exponent and expand to `;⟨exponent⟩` only. Otherwise, we expand to `+0` or `+1`, then `;⟨exponent⟩`. To decide which, call `\_fp_round_s:NNNw` to know whether to round up, giving it as arguments a sign 0 (all explicit numbers are positive), the digit `#1` to round, the first following digit `#2`, and either `+0` or `+1` depending on whether the following digits are all zero or not. This last argument is obtained by `\_fp_parse_round_loop:N`, whose number of digits we discard by multiplying it by 0. The exponent which follows the number is also fetched by `\_fp_parse_round_after:wN`.

```

23868 \cs_new:Npn \_fp_parse_small_round:NN #1#2
23869 {
23870     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
23871     +
23872     \exp_after:wN \_fp_round_s:NNNw
23873     \exp_after:wN 0
23874     \exp_after:wN #1
23875     \exp_after:wN #2
23876     \int_value:w \_fp_int_eval:w
23877     \exp_after:wN \_fp_parse_round_after:wN
23878     \int_value:w \_fp_int_eval:w 0 * \_fp_int_eval:w 0
23879     \exp_after:wN \_fp_parse_round_loop:N
23880     \exp:w
23881 \else:
23882     \_fp_parse_exponent:Nw #2

```

```

23883     \fi:
23884     \_\_fp\_parse\_expand:w
23885 }

```

(End definition for `\_\_fp\_parse\_small\_round:NN` and `\_\_fp\_parse\_round\_after:wN`.)

```
\_\_fp_parse_large_round:NN
\_\_fp_parse_large_round_test:NN
\_\_fp_parse_large_round_aux:wNN
```

Large numbers are harder to round, as there may be a period in the way. Again, #1 is the digit that we are currently rounding (we only care whether it is even or odd). If there are no more digits (#2 is not a digit), then we must test for a period: if there is one, then switch to the rounding function for small significands, otherwise fetch an exponent. If there are more digits (#2 is a digit), then round, checking with `\_\_fp\_parse\_round\_loop:N` if all further digits vanish, or some are non-zero. This loop is not enough, as it is stopped by a period. After the loop, the `aux` function tests for a period: if it is present, then we must continue looking for digits, this time discarding the number of digits we find.

```

23886 \cs_new:Npn \_\_fp_parse_large_round:NN #1#2
23887 {
23888     \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
23889     +
2390     \exp_after:wN \_\_fp_round_s:NNNw
2391     \exp_after:wN 0
2392     \exp_after:wN #1
2393     \exp_after:wN #2
2394     \int_value:w \_\_fp_int_eval:w
2395     \exp_after:wN \_\_fp_parse_large_round_aux:wNN
2396     \int_value:w \_\_fp_int_eval:w 1
2397     \exp_after:wN \_\_fp_parse_round_loop:N
2398     \else: %^A could be dot, or e, or other
2399     \exp_after:wN \_\_fp_parse_large_round_test:NN
2400     \exp_after:wN #1
2401     \exp_after:wN #2
2402     \fi:
2403 }
2404 \cs_new:Npn \_\_fp_parse_large_round_test:NN #1#2
2405 {
2406     \if:w . \exp_not:N #2
2407     \exp_after:wN \_\_fp_parse_small_round:NN
2408     \exp_after:wN #1
2409     \exp:w
2410     \else:
2411     \_\_fp_parse_exponent:Nw #2
2412     \fi:
2413     \_\_fp_parse_expand:w
2414 }
2415 \cs_new:Npn \_\_fp_parse_large_round_aux:wNN #1 ; #2 #3
2416 {
2417     + #2
2418     \exp_after:wN \_\_fp_parse_round_after:wN
2419     \int_value:w \_\_fp_int_eval:w #1
2420     \if:w . \exp_not:N #3
2421     + 0 * \_\_fp_int_eval:w 0
2422     \exp_after:wN \_\_fp_parse_round_loop:N
2423     \exp:w \exp_after:wN \_\_fp_parse_expand:w
2424     \else:

```

```

23925     \exp_after:wN ;
23926     \exp_after:wN 0
23927     \exp_after:wN #3
23928     \fi:
23929 }

```

*(End definition for \\_\_fp\_parse\_large\_round:NN, \\_\_fp\_parse\_large\_round\_test:NN, and \\_\_fp\_parse\_large\_round\_aux:wNN.)*

#### 69.4.5 Number: finding the exponent

Expansion is a little bit tricky here, in part because we accept input where multiplication is implicit.

```

\__fp_parse:n { 3.2 erf(0.1) }
\__fp_parse:n { 3.2 e\l_my_int }
\__fp_parse:n { 3.2 \c_pi_fp }

```

The first case indicates that just looking one character ahead for an “e” is not enough, since we would mistake the function `erf` for an exponent of “rf”. An alternative would be to look two tokens ahead and check if what follows is a sign or a digit, considering in that case that we must be finding an exponent. But taking care of the second case requires that we unpack registers after `e`. However, blindly expanding the two tokens ahead completely would break the third example (unpacking is even worse). Indeed, in the course of reading 3.2, `\c_pi_fp` is expanded to `\s_fp \__fp_chk:w 1 0 {-1} {3141} ... ;` and `\s_fp` stops the expansion. Expanding two tokens ahead would then force the expansion of `\__fp_chk:w` (despite it being protected), and that function tries to produce an error.

What can we do? Really, the reason why this last case breaks is that just as TeX does, we should read ahead as little as possible. Here, the only case where there may be an exponent is if the first token ahead is `e`. Then we expand (and possibly unpack) the second token.

`\__fp_parse_exponent:Nw` This auxiliary is convenient to smuggle some material through `\fi:` ending conditional processing. We place those `\fi:` (argument #2) at a very odd place because this allows us to insert `\__fp_int_eval:w ...` there if needed.

```

23930 \cs_new:Npn \__fp_parse_exponent:Nw #1 #2 \__fp_parse_expand:w
23931 {
23932     \exp_after:wN ;
23933     \int_value:w #2 \__fp_parse_exponent:N #1
23934 }

```

*(End definition for \\_\_fp\_parse\_exponent:Nw.)*

`\__fp_parse_exponent:N` This function should be called within an `\int_value:w` expansion (or within an integer expression). It leaves digits of the exponent behind it in the input stream, and terminates the expansion with a semicolon. If there is no `e` (or `E`), leave an exponent of 0. If there is an `e` or `E`, expand the next token to run some tests on it. The first rough test is that if the character code of #1 is greater than that of 9 (largest code valid for an exponent, less than any code valid for an identifier), there was in fact no exponent; otherwise, we search for the sign of the exponent.

```

23935 \cs_new:Npn \__fp_parse_exponent:N #1
23936 {

```

```

23937      \if:w e \if:w E \exp_not:N #1 e \else: \exp_not:N #1 \fi:
23938          \exp_after:wN \_fp_parse_exponent_aux:NN
23939          \exp_after:wN #1
23940          \exp:w
23941      \else:
23942          0 \_fp_parse_return_semicolon:w #1
23943      \fi:
23944      \_fp_parse_expand:w
23945  }
23946 \cs_new:Npn \_fp_parse_exponent_aux:NN #1#2
23947  {
23948      \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #2
23949          0 \else: '#2 \fi: > '9 \exp_stop_f:
23950          0 \exp_after:wN ; \exp_after:wN #1
23951      \else:
23952          \exp_after:wN \_fp_parse_exponent_sign:N
23953      \fi:
23954  #2
23955 }

```

(End definition for `\_fp_parse_exponent:N` and `\_fp_parse_exponent_aux:NN`.)

`\_fp_parse_exponent_sign:N` Read signs one by one (if there is any).

```

23956 \cs_new:Npn \_fp_parse_exponent_sign:N #1
23957  {
23958      \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
23959          \exp_after:wN \_fp_parse_exponent_sign:N
23960          \exp:w \exp_after:wN \_fp_parse_expand:w
23961      \else:
23962          \exp_after:wN \_fp_parse_exponent_body:N
23963          \exp_after:wN #1
23964      \fi:
23965 }

```

(End definition for `\_fp_parse_exponent_sign:N`.)

`\_fp_parse_exponent_body:N` An exponent can be an explicit integer (most common case), or various other things (most of which are invalid).

```

23966 \cs_new:Npn \_fp_parse_exponent_body:N #1
23967  {
23968      \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23969          \token_to_str:N #1
23970          \exp_after:wN \_fp_parse_exponent_digits:N
23971          \exp:w
23972      \else:
23973          \_fp_parse_exponent_keep:NTF #1
23974          { \_fp_parse_return_semicolon:w #1 }
23975          {
23976              \exp_after:wN ;
23977              \exp:w
23978          }
23979      \fi:
23980      \_fp_parse_expand:w
23981 }

```

(End definition for `\_fp_parse_exponent_body:N`.)

`\_fp_parse_exponent_digits:N`

Read digits one by one, and leave them behind in the input stream. When finding a non-digit, stop, and insert a semicolon. Note that we do not check for overflow of the exponent, hence there can be a TeX error. It is mostly harmless, except when parsing 0e9876543210, which should be a valid representation of 0, but is not.

```
23982 \cs_new:Npn \_fp_parse_exponent_digits:N #1
23983 {
23984     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
23985         \token_to_str:N #1
23986         \exp_after:wN \_fp_parse_exponent_digits:N
23987             \exp:w
23988     \else:
23989         \_fp_parse_return_semicolon:w #1
23990     \fi:
23991     \_fp_parse_expand:w
23992 }
```

(End definition for `\_fp_parse_exponent_digits:N`.)

`\_fp_parse_exponent_keep:NTF`

This is the last building block for parsing exponents. The argument `#1` is already fully expanded, and neither `+` nor `-` nor a digit. It can be:

- `\s_fp`, marking the start of an internal floating point, invalid here;
- another control sequence equal to `\relax`, probably a bad variable;
- a register: in this case we make sure that it is an integer register, not a dimension;
- a character other than `+`, `-` or digits, again, an error.

```
23993 \prg_new_conditional:Npnn \_fp_parse_exponent_keep:N #1 { TF }
23994 {
23995     \if_catcode:w \scan_stop: \exp_not:N #1
23996         \if_meaning:w \scan_stop: #1
23997             \if:w 0 \_fp_str_if_eq:nn { \s_fp } { \exp_not:N #1 }
23998                 0
23999                 \msg_expandable_error:nnn
24000                     { fp } { after-e } { floating-point~ }
24001                     \prg_return_true:
24002             \else:
24003                 0
24004                 \msg_expandable_error:nnn
24005                     { kernel } { bad-variable } { #1 }
24006                     \prg_return_false:
24007             \fi:
24008         \else:
24009             \if:w 0 \_fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
24010                 \int_value:w #1
24011             \else:
24012                 0
24013                 \msg_expandable_error:nnn
24014                     { fp } { after-e } { dimension~#1 }
24015             \fi:
24016             \prg_return_false:
```

```

24017      \fi:
24018  \else:
24019    0
24020    \msg_expandable_error:nnn
24021      { fp } { missing } { exponent }
24022    \prg_return_true:
24023  \fi:
24024 }

```

(End definition for `\_fp_parse_exponent_keep:NTF.`)

## 69.5 Constants, functions and prefix operators

### 69.5.1 Prefix operators

`\_fp_parse_prefix_+:`Nw A unary + does nothing: we should continue looking for a number.

```
24025 \cs_new_eq:cN { \_fp_parse_prefix_+:Nw } \_fp_parse_one:Nw
```

(End definition for `\_fp_parse_prefix_+:Nw.`)

`\_fp_parse_apply_function:NNNwN` Here, #1 is a precedence, #2 is some extra data used by some functions, #3 is e.g., `\_fp_sin_o:w`, and expands once after the calculation, #4 is the operand, and #5 is a `\_fp_parse_infix_...:N` function. We feed the data #2, and the argument #4, to the function #3, which expands `\exp:w` thus the `infix` function #5.

```

24026 \cs_new:Npn \_fp_parse_apply_function:NNNwN #1#2#3#4@#5
24027 {
24028   #3 #2 #4 @
24029   \exp:w \exp_end_continue_f:w #5 #1
24030 }
```

(End definition for `\_fp_parse_apply_function:NNNwN.`)

`\_fp_parse_apply_unary:NNNwN` In contrast to `\_fp_parse_apply_function:NNNwN`, this checks that the operand #4 is a single argument (namely there is a single ;). We use the fact that any floating point starts with a “safe” token like `\s_fp`. If there is no argument produce the `fp-no-arg` error; if there are at least two produce `fp-multi-arg`. For the error message extract the mathematical function name (such as `sin`) from the `expl3` function that computes it, such as `\_fp_sin_o:w`.

In addition, since there is a single argument we can dispatch on type and check that the resulting function exists. This catches things like `sin((1,2))` where it does not make sense to take the sine of a tuple.

```

24031 \cs_new:Npn \_fp_parse_apply_unary:NNNwN #1#2#3#4@#5
24032 {
24033   \_fp_parse_apply_unary_chk:NwNw #4 @ . \s_fp_stop
24034   \_fp_parse_apply_unary_type:NNN
24035   #3 #2 #4 @
24036   \exp:w \exp_end_continue_f:w #5 #1
24037 }
24038 \cs_new:Npn \_fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \s_fp_stop
24039 {
24040   \if_meaning:w @ #3 \else:
24041     \token_if_eq_meaning:NNTF . #3
24042       { \_fp_parse_apply_unary_chk:nNNNNw { no } }
```

```

24043     { \__fp_parse_apply_unary_chk:nNNNNw { multi } }
24044     \fi:
24045   }
24046 \cs_new:Npn \__fp_parse_apply_unary_chk:nNNNNw #1#2#3#4#5#6 @
24047   {
24048     #2
24049     \__fp_error:nffn { #1-arg } { \__fp_func_to_name:N #4 } { } { }
24050     \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
24051   }
24052 \cs_new:Npn \__fp_parse_apply_unary_type:NNN #1#2#3
24053   {
24054     \__fp_change_func_type:NNN #3 #1 \__fp_parse_apply_unary_error>NNw
24055     #2 #3
24056   }
24057 \cs_new:Npn \__fp_parse_apply_unary_error>NNw #1#2#3 @
24058   { \__fp_invalid_operation_o:fw { \__fp_func_to_name:N #1 } #3 }

(End definition for \__fp_parse_apply_unary>NNNwN and others.)

```

\\_\_fp\_parse\_prefix\_:-:Nw  
\\_\_fp\_parse\_prefix\_!:Nw

The unary - and boolean not are harder: we parse the operand using a precedence equal to the maximum of the previous precedence ##1 and the precedence \c\_\_fp\_prec\_not\_-int of the unary operator, then call the appropriate \\_\_fp\_{operation}\_o:w function, where the *(operation)* is set\_sign or not.

```

24059 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
24060   {
24061     \cs_new:cpn { __fp_parse_prefix_- :Nw } ##1
24062     {
24063       \exp_after:wN \__fp_parse_apply_unary>NNNwN
24064       \exp_after:wN ##1
24065       \exp_after:wN #4
24066       \exp_after:wN #3
24067       \exp:w
24068       \if_int_compare:w #2 < ##1
24069         \__fp_parse_operand:Nw ##1
24070       \else:
24071         \__fp_parse_operand:Nw #2
24072       \fi:
24073       \__fp_parse_expand:w
24074     }
24075   }
24076 \__fp_tmp:w - \c__fp_prec_not_int \__fp_set_sign_o:w 2
24077 \__fp_tmp:w ! \c__fp_prec_not_int \__fp_not_o:w ?

(End definition for \__fp_parse_prefix_:-:Nw and \__fp_parse_prefix_!:Nw.)

```

\\_\_fp\_parse\_prefix\_.:Nw

Numbers which start with a decimal separator (a period) end up here. Of course, we do not look for an operand, but for the rest of the number. This function is very similar to \\_\_fp\_parse\_one\_digit:NN but calls \\_\_fp\_parse\_strim\_zeros:N to trim zeros after the decimal point, rather than the trim\_zeros function for zeros before the decimal point.

```

24078 \cs_new:cpn { __fp_parse_prefix_.:Nw } #1
24079   {
24080     \exp_after:wN \__fp_parse_infix_after_operand:NwN
24081     \exp_after:wN #1

```

```

24082     \exp:w \exp_end_continue_f:w
24083     \exp_after:wN \__fp_SANITIZE:wN
24084     \int_value:w \__fp_int_eval:w 0 \__fp_PARSE_STRIM_ZEROS:N
24085   }

```

(End definition for `\__fp_PARSE_PREFIX_`.)

`\__fp_PARSE_PREFIX_(:Nw`

The left parenthesis is treated as a unary prefix operator because it appears in exactly the same settings. If the previous precedence is `\c__fp_prec_func_int` we are parsing arguments of a function and commas should not build tuples; otherwise commas should build tuples. We distinguish these cases by precedence: `\c__fp_prec_comma_int` for the case of arguments, `\c__fp_prec_tuple_int` for the case of tuples. Once the operand is found, the `1paren_after` auxiliary makes sure that there was a closing parenthesis (otherwise it complains), and leaves in the input stream an operand, fetching the following infix operator.

```

24086 \cs_new:cpn { \__fp_PARSE_PREFIX_(:Nw } #1
24087   {
24088     \exp_after:wN \__fp_PARSE_Lparen_after:NwN
24089     \exp_after:wN #1
24090     \exp:w
24091     \if_int_compare:w #1 = \c__fp_prec_func_int
24092       \__fp_PARSE_OPERAND:Nw \c__fp_prec_comma_int
24093     \else:
24094       \__fp_PARSE_OPERAND:Nw \c__fp_prec_tuple_int
24095     \fi:
24096     \__fp_PARSE_EXPAND:w
24097   }
24098 \cs_new:Npx \__fp_PARSE_Lparen_after:NwN #1#2 @ #3
24099   {
24100     \exp_not:N \token_if_eq_meaning:NNTF #3
24101     \exp_not:c { \__fp_PARSE_INFIX_ }:N }
24102   {
24103     \exp_not:N \__fp_EXP_AFTER_ARRAY_F:w #2 \s__fp_EXPR_STOP
24104     \exp_not:N \exp_after:wN
24105     \exp_not:N \__fp_PARSE_INFIX_AFTER_PAREN:NN
24106     \exp_not:N \exp_after:wN #1
24107     \exp_not:N \exp:w
24108     \exp_not:N \__fp_PARSE_EXPAND:w
24109   }
24110   {
24111     \exp_not:N \msg_expandable_error:nnn
24112     { fp } { missing } { ) }
24113     \exp_not:N \tl_if_empty:nT { #2 } \exp_not:N \c__fp_EMPTY_TUPLE_fp
24114     #2 @
24115     \exp_not:N \use_none:n #3
24116   }
24117 }

```

(End definition for `\__fp_PARSE_PREFIX_` and `\__fp_PARSE_Lparen_after`.)

`\__fp_PARSE_PREFIX_):Nw`

The right parenthesis can appear as a prefix in two similar cases: in an empty tuple or tuple ending with a comma, or in an empty argument list or argument list ending with a comma, such as in `max(1,2,)` or in `rand()`.

```
24118 \cs_new:cpn { \__fp_PARSE_PREFIX_ ):Nw } #1
```

```

24119  {
24120  \if_int_compare:w #1 = \c_fp_prec_comma_int
24121  \else:
24122  \if_int_compare:w #1 = \c_fp_prec_tuple_int
24123  \exp_after:wN \c_empty_tuple_fp \exp:w
24124  \else:
24125  \msg_expandable_error:nnn
24126  { fp } { missing-number } { }
24127  \exp_after:wN \c_nan_fp \exp:w
24128  \fi:
24129  \exp_end_continue_f:w
24130  \fi:
24131  \__fp_parse_infix_after_paren:NN #1 )
24132  }

```

(End definition for `\__fp_parse_prefix_`):Nw.)

### 69.5.2 Constants

Some words correspond to constant floating points. The floating point constant is left as a result of `\__fp_parse_one:Nw` after expanding `\__fp_parse_infix:NN`.

```

24133 \cs_set_protected:Npn \__fp_tmp:w #1 #2
24134  {
24135  \cs_new:cpn { __fp_parse_word_#1:N }
24136  { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
24137  }
24138 \__fp_tmp:w { inf } \c_inf_fp
24139 \__fp_tmp:w { nan } \c_nan_fp
24140 \__fp_tmp:w { pi } \c_pi_fp
24141 \__fp_tmp:w { deg } \c_one_degree_fp
24142 \__fp_tmp:w { true } \c_one_fp
24143 \__fp_tmp:w { false } \c_zero_fp

```

(End definition for `\__fp_parse_word_inf:N` and others.)

Copies of `\__fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

```

24144 \cs_new_eq:NN \__fp_parse_caseless_inf:N \__fp_parse_word_inf:N
24145 \cs_new_eq:NN \__fp_parse_caseless_infinity:N \__fp_parse_word_inf:N
24146 \cs_new_eq:NN \__fp_parse_caseless_nan:N \__fp_parse_word_nan:N

```

(End definition for `\__fp_parse_caseless_inf:N`, `\__fp_parse_caseless_infinity:N`, and `\__fp_parse_caseless_nan:N`.)

`\__fp_parse_word_pt:N` Dimension units are also floating point constants but their value is not stored as a floating point constant. We give the values explicitly here.

```

24147 \cs_set_protected:Npn \__fp_tmp:w #1 #2
24148  {
24149  \cs_new:cpn { __fp_parse_word_#1:N }
24150  {
24151  \__fp_exp_after_f:nw { \__fp_parse_infix:NN }
24152  \s_fp \__fp_chk:w 10 #2 ;
24153  }
24154  }

```

```

24155 \__fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
24156 \__fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
24157 \__fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
24158 \__fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
24159 \__fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
24160 \__fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
24161 \__fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
24162 \__fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
24163 \__fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
24164 \__fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
24165 \__fp_tmp:w {sp} { {-4} {1525} {8789} {0625} {0000} }

```

(End definition for `\__fp_parse_word_pt:N` and others.)

`\__fp_parse_word_em:N` The font-dependent units `em` and `ex` must be evaluated on the fly. We reuse an auxiliary of `\dim_to_fp:n`.

```

24166 \tl_map_inline:nn { {em} {ex} }
24167 {
24168   \cs_new:cpn { __fp_parse_word_#1:N }
24169   {
24170     \exp_after:wN \__fp_from_dim_test:ww
24171     \exp_after:wN 0 \exp_after:wN ,
24172     \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
24173     \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN
24174   }
24175 }

```

(End definition for `\__fp_parse_word_em:N` and `\__fp_parse_word_ex:N`.)

### 69.5.3 Functions

```

\__fp_parse_unary_function:NNN
\__fp_parse_function:NNN
24176 \cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
24177 {
24178   \exp_after:wN \__fp_parse_apply_unary:NNNwN
24179   \exp_after:wN #3
24180   \exp_after:wN #2
24181   \exp_after:wN #1
24182   \exp:w
24183   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
24184 }
24185 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
24186 {
24187   \exp_after:wN \__fp_parse_apply_function:NNNwN
24188   \exp_after:wN #3
24189   \exp_after:wN #2
24190   \exp_after:wN #1
24191   \exp:w
24192   \__fp_parse_operand:Nw \c__fp_prec_func_int \__fp_parse_expand:w
24193 }

```

(End definition for `\__fp_parse_unary_function:NNN` and `\__fp_parse_function:NNN`.)

## 69.6 Main functions

`\_fp_parse:n` Start an `\exp:w` expansion so that `\_fp_parse:n` expands in two steps. The `\_fp_parse_operand:Nw` function performs computations until reaching an operation with precedence `\c_fp_prec_end_int` or less, namely, the end of the expression. The marker `\s_fp_expr_mark` indicates that the next token is an already parsed version of an infix operator, and `\_fp_parse_infix_end:N` has infinitely negative precedence. Finally, clean up a (well-defined) set of extra tokens and stop the initial expansion with `\exp_end:`.

```

24194 \cs_new:Npn \_fp_parse:n #1
24195 {
24196     \exp:w
24197         \exp_after:wN \_fp_parse_after:ww
24198         \exp:w
24199             \_fp_parse_operand:Nw \c_fp_prec_end_int
24200             \_fp_parse_expand:w #1
24201             \s_fp_expr_mark \_fp_parse_infix_end:N
24202             \s_fp_expr_stop
24203             \exp_end:
24204     }
24205 \cs_new:Npn \_fp_parse_after:ww
24206     #1@ \_fp_parse_infix_end:N \s_fp_expr_stop #2 { #2 #1 }
24207 \cs_new:Npn \_fp_parse_o:n #1
24208 {
24209     \exp:w
24210         \exp_after:wN \_fp_parse_after:ww
24211         \exp:w
24212             \_fp_parse_operand:Nw \c_fp_prec_end_int
24213             \_fp_parse_expand:w #1
24214             \s_fp_expr_mark \_fp_parse_infix_end:N
24215             \s_fp_expr_stop
24216 {
24217     \exp_end_continue_f:w
24218     \_fp_exp_after_any_f:nw { \exp_after:wN \exp_stop_f: }
24219 }
24220 }
```

(End definition for `\_fp_parse:n`, `\_fp_parse_o:n`, and `\_fp_parse_after:ww`.)

`\_fp_parse_operand:Nw` This is just a shorthand which sets up both `\_fp_parse_continue:NwN` and `\_fp_parse_one:Nw` with the same precedence. Note the trailing `\exp:w`.

```

24221 \cs_new:Npn \_fp_parse_operand:Nw #1
24222 {
24223     \exp_end_continue_f:w
24224     \exp_after:wN \_fp_parse_continue:NwN
24225     \exp_after:wN #1
24226     \exp:w \exp_end_continue_f:w
24227     \exp_after:wN \_fp_parse_one:Nw
24228     \exp_after:wN #1
24229     \exp:w
24230 }
24231 \cs_new:Npn \_fp_parse_continue:NwN #1 #2 @ #3 { #3 #1 #2 @ }
```

(End definition for `\_fp_parse_operand:Nw` and `\_fp_parse_continue:NwN`.)

```
\_\_fp_parse_apply_binary:NwNwN
\_\_fp_parse_apply_binary_chk:NN
\_\_fp_parse_apply_binary_error:NNN
```

Receives  $\langle precedence \rangle \langle operand_1 \rangle @ \langle operation \rangle \langle operand_2 \rangle @ \langle infix command \rangle$ . Builds the appropriate call to the  $\langle operation \rangle$  #3, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```
24232 \cs_new:Npn \_\_fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
24233 {
24234     \exp_after:wN \_\_fp_parse_continue:NwN
24235     \exp_after:wN #1
24236     \exp:w \exp_end_continue_f:w
24237     \exp_after:wN \_\_fp_parse_apply_binary_chk:NN
24238     \cs:w
24239         __fp
24240         \_\_fp_type_from_scan:N #2
24241         _#4
24242         \_\_fp_type_from_scan:N #5
24243         _o:ww
24244         \cs_end:
24245         #4
24246         #2#3 #5#6
24247         \exp:w \exp_end_continue_f:w #7 #1
24248     }
24249 \cs_new:Npn \_\_fp_parse_apply_binary_chk:NN #1#2
24250 {
24251     \if_meaning:w \scan_stop: #1
24252     \_\_fp_parse_apply_binary_error:NNN #2
24253     \fi:
24254     #1
24255 }
24256 \cs_new:Npn \_\_fp_parse_apply_binary_error:NNN #1#2#3
24257 {
24258     #2
24259     \_\_fp_invalid_operation_o:Nww #1
24260 }
```

(End definition for  $\_\_fp_parse_apply_binary:NwNwN$ ,  $\_\_fp_parse_apply_binary_chk:NN$ , and  $\_\_fp_parse_apply_binary_error:NNN$ .)

```
\_\_fp_binary_type_o:Nww
\_\_fp_binary_rev_type_o:Nww
```

Applies the operator #1 to its two arguments, dispatching according to their types, and expands once after the result. The `rev` version swaps its arguments before doing this.

```
24261 \cs_new:Npn \_\_fp_binary_type_o:Nww #1 #2#3 ; #4
24262 {
24263     \exp_after:wN \_\_fp_parse_apply_binary_chk:NN
24264     \cs:w
24265         __fp
24266         \_\_fp_type_from_scan:N #2
24267         _#1
24268         \_\_fp_type_from_scan:N #4
24269         _o:ww
24270         \cs_end:
24271         #1
24272         #2 #3 ; #4
24273     }
24274 \cs_new:Npn \_\_fp_binary_rev_type_o:Nww #1 #2#3 ; #4#5 ;
24275 {
```

```

24276 \exp_after:wN \_fp_parse_apply_binary_chk:NN
24277   \cs:w
24278     _fp
24279     \_fp_type_from_scan:N #4
24280     _#1
24281     \_fp_type_from_scan:N #2
24282     _o:ww
24283   \cs_end:
24284   #1
24285   #4 #5 ; #2 #3 ;
24286 }

```

(End definition for \\_fp\_binary\_type\_o:Nww and \\_fp\_binary\_rev\_type\_o:Nww.)

## 69.7 Infix operators

```

\_fp_parse_infix_after_operand:NwN
24287 \cs_new:Npn \_fp_parse_infix_after_operand:NwN #1 #2;
24288 {
24289   \_fp_exp_after_f:nw { \_fp_parse_infix:NN #1 }
24290   #2;
24291 }
24292 \cs_new:Npn \_fp_parse_infix:NN #1 #2
24293 {
24294   \if_catcode:w \scan_stop: \exp_not:N #2
24295   \if:w 0 \_fp_str_if_eq:nn { \s_fp_expr_mark } { \exp_not:N #2 }
24296   \exp_after:wN \exp_after:wN
24297   \exp_after:wN \_fp_parse_infix_mark:NNN
24298   \else:
24299   \exp_after:wN \exp_after:wN
24300   \exp_after:wN \_fp_parse_infix_juxt:N
24301   \fi:
24302   \else:
24303   \if_int_compare:w
24304     \_fp_int_eval:w
24305     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
24306     = 3 \exp_stop_f:
24307     \exp_after:wN \exp_after:wN
24308     \exp_after:wN \_fp_parse_infix_juxt:N
24309   \else:
24310     \exp_after:wN \_fp_parse_infix_check:NNN
24311     \cs:w
24312       \_fp_parse_infix_ \token_to_str:N #2 :N
24313       \exp_after:wN \exp_after:wN \exp_after:wN
24314     \cs_end:
24315     \fi:
24316   \fi:
24317   #1
24318   #2
24319 }
24320 \cs_new:Npn \_fp_parse_infix_check:NNN #1#2#3
24321 {
24322   \if_meaning:w \scan_stop: #1

```

```

24323   \msg_expandable_error:nnn
24324     { fp } { missing } { * }
24325   \exp_after:wN \__fp_parse_infix_mul:N
24326   \exp_after:wN #2
24327   \exp_after:wN #3
24328 \else:
24329   \exp_after:wN #1
24330   \exp_after:wN #2
24331   \exp:w \exp_after:wN \__fp_parse_expand:w
24332 \fi:
24333 }

```

(End definition for `\__fp_parse_infix_after_operand:NwN.`)

`\__fp_parse_infix_after_paren:NN` Variant of `\__fp_parse_infix:NN` for use after a closing parenthesis. The only difference is that `\__fp_parse_infix_juxt:N` is replaced by `\__fp_parse_infix_mul:N`.

```

24334 \cs_new:Npn \__fp_parse_infix_after_paren:NN #1 #2
24335   {
24336     \if_catcode:w \scan_stop: \exp_not:N #2
24337     \if:w 0 \__fp_str_if_eq:nn { \s_fp_expr_mark } { \exp_not:N #2 }
24338       \exp_after:wN \exp_after:wN
24339       \exp_after:wN \__fp_parse_infix_mark:NNN
24340     \else:
24341       \exp_after:wN \exp_after:wN
24342       \exp_after:wN \__fp_parse_infix_mul:N
24343     \fi:
24344   \else:
24345     \if_int_compare:w
24346       \__fp_int_eval:w
24347         ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
24348         = 3 \exp_stop_f:
24349         \exp_after:wN \exp_after:wN
24350         \exp_after:wN \__fp_parse_infix_mul:N
24351     \else:
24352       \exp_after:wN \__fp_parse_infix_check:NNN
24353       \cs:w
24354         \__fp_parse_infix_ \token_to_str:N #2 :N
24355         \exp_after:wN \exp_after:wN \exp_after:wN
24356         \cs_end:
24357       \fi:
24358     \fi:
24359     #1
24360     #2
24361 }

```

(End definition for `\__fp_parse_infix_after_paren:NN.`)

### 69.7.1 Closing parentheses and commas

As an infix operator, `\s_fp_expr_mark` means that the next token (#3) has already gone through `\__fp_parse_infix:NN` and should be provided the precedence #1. The scan mark #2 is discarded.

```
24362 \cs_new:Npn \__fp_parse_infix_mark:NNN #1#2#3 { #3 #1 }
```

(End definition for `\_fp_parse_infix_mark:NNN`.)

`\_fp_parse_infix_end:N` This one is a little bit odd: force every previous operator to end, regardless of the precedence.

```
24363 \cs_new:Npn \_fp_parse_infix_end:N #1
24364   { @ \use_none:n \_fp_parse_infix_end:N }
```

(End definition for `\_fp_parse_infix_end:N`.)

`\_fp_parse_infix_):N` This is very similar to `\_fp_parse_infix_end:N`, complaining about an extra closing parenthesis if the previous operator was the beginning of the expression, with precedence `\c_fp_prec_end_int`.

```
24365 \cs_set_protected:Npn \_fp_tmp:w #1
24366   {
24367     \cs_new:Npn #1 ##1
24368     {
24369       \if_int_compare:w ##1 > \c_fp_prec_end_int
24370         \exp_after:wN @
24371         \exp_after:wN \use_none:n
24372         \exp_after:wN #1
24373       \else:
24374         \msg_expandable_error:nnn { fp } { extra } { )
24375         \exp_after:wN \_fp_parse_infix:NN
24376         \exp_after:wN ##1
24377         \exp:w \exp_after:wN \_fp_parse_expand:w
24378       \fi:
24379     }
24380   }
24381 \exp_args:Nc \_fp_tmp:w { \_fp_parse_infix_):N }
```

(End definition for `\_fp_parse_infix_):N`.)

`\_fp_parse_infix_,:N` As for other infix operations, if the previous operations has higher precedence the comma waits. Otherwise we call `\_fp_parse_operand:Nw` to read more comma-delimited arguments that `\_fp_parse_infix_comma:w` simply concatenates into a `@`-delimited array. The first comma in a tuple that is not a function argument is distinguished: in that case call `\_fp_parse_apply_comma:NwNwN` whose job is to convert the first item of the tuple and an array of the remaining items into a tuple. In contrast to `\_fp_parse_apply_binary:NwNwN` this function's operands are not single-object arrays.

```
24382 \cs_set_protected:Npn \_fp_tmp:w #1
24383   {
24384     \cs_new:Npn #1 ##1
24385     {
24386       \if_int_compare:w ##1 > \c_fp_prec_comma_int
24387         \exp_after:wN @
24388         \exp_after:wN \use_none:n
24389         \exp_after:wN #1
24390       \else:
24391         \if_int_compare:w ##1 < \c_fp_prec_comma_int
24392           \exp_after:wN @
24393           \exp_after:wN \_fp_parse_apply_comma:NwNwN
24394           \exp_after:wN ,
24395           \exp:w
24396       \else:
```

```

24397           \exp_after:wN \_fp_parse_infix_comma:w
24398           \exp:w
24399       \fi:
24400           \_fp_parse_operand:Nw \c_fp_prec_comma_int
24401           \exp_after:wN \_fp_parse_expand:w
24402       \fi:
24403   }
24404 }
24405 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_,:N }
24406 \cs_new:Npn \_fp_parse_infix_comma:w #1 @
24407   { #1 @ \use_none:n }
24408 \cs_new:Npn \_fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
24409   {
24410     \exp_after:wN \_fp_parse_continue:NwN
24411     \exp_after:wN #1
24412     \exp:w \exp_end_continue_f:w
24413     \_fp_exp_after_tuple_f:nw { }
24414     \s_fp_tuple \_fp_tuple_chk:w { #2 #4 } ;
24415     #5 #1
24416   }

```

(End definition for `\_fp_parse_infix_,:N`, `\_fp_parse_infix_comma:w`, and `\_fp_parse_apply_comma:NwNwN`.)

### 69.7.2 Usual infix operators

`\_fp_parse_infix_+:N`  
`\_fp_parse_infix_-:N`  
`\_fp_parse_infix_juxt:N`  
`\_fp_parse_infix/_:N`  
`\_fp_parse_infix_mul:N`  
`\_fp_parse_infix_and:N`  
`\_fp_parse_infix_or:N`  
`\_fp_parse_infix_^:N`

As described in the “work plan”, each infix operator has an associated `\..._infix_...` function, a computing function, and precedence, given as arguments to `\_fp_tmp:w`. Using the general mechanism for arithmetic operations. The power operation must be associative in the opposite order from all others. For this, we use two distinct precedences.

```

24417 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
24418   {
24419     \cs_new:Npn #1 ##1
24420     {
24421       \if_int_compare:w ##1 < #3
24422         \exp_after:wN @
24423         \exp_after:wN \_fp_parse_apply_binary:NwNwN
24424         \exp_after:wN #2
24425         \exp:w
24426         \_fp_parse_operand:Nw #4
24427         \exp_after:wN \_fp_parse_expand:w
24428       \else:
24429         \exp_after:wN @
24430         \exp_after:wN \use_none:n
24431         \exp_after:wN #1
24432       \fi:
24433     }
24434   }
24435 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_^:N } ^ ~
24436   \c_fp_prec_hatii_int \c_fp_prec_hat_int
24437 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_juxt:N } *
24438   \c_fp_prec_juxt_int \c_fp_prec_juxt_int
24439 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix/_:N } /
24440   \c_fp_prec_times_int \c_fp_prec_times_int

```

```

24441 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_mul:N } *
24442   \c_fp_prec_times_int \c_fp_prec_times_int
24443 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_-:N } -
24444   \c_fp_prec_plus_int \c_fp_prec_plus_int
24445 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_+:N } +
24446   \c_fp_prec_plus_int \c_fp_prec_plus_int
24447 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_and:N } &
24448   \c_fp_prec_and_int \c_fp_prec_and_int
24449 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_or:N } |
24450   \c_fp_prec_or_int \c_fp_prec_or_int

```

(End definition for `\_fp_parse_infix_+:N` and others.)

### 69.7.3 Juxtaposition

- `\_fp_parse_infix_(:N)` When an opening parenthesis appears where we expect an infix operator, we compute the product of the previous operand and the contents of the parentheses using `\_fp_parse_infix_mul:N`.

```

24451 \cs_new:cpn { __fp_parse_infix_(:N } #1
24452   { \_fp_parse_infix_mul:N #1 ( }

```

(End definition for `\_fp_parse_infix_(:N)`.)

### 69.7.4 Multi-character cases

```

\__fp_parse_infix_*:N
24453 \cs_set_protected:Npn \_fp_tmp:w #1
24454   {
24455     \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
24456     {
24457       \if:w * \exp_not:N ##2
24458         \exp_after:wN #1
24459         \exp_after:wN ##1
24460       \else:
24461         \exp_after:wN \_fp_parse_infix_mul:N
24462         \exp_after:wN ##1
24463         \exp_after:wN ##2
24464       \fi:
24465     }
24466   }
24467 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_~:N }

(End definition for \_fp_parse_infix_*:N.)
```

```

\__fp_parse_infix_|:Nw
\__fp_parse_infix_&:Nw
24468 \cs_set_protected:Npn \_fp_tmp:w #1#2#3
24469   {
24470     \cs_new:Npn #1 ##1##2
24471     {
24472       \if:w #2 \exp_not:N ##2
24473         \exp_after:wN #1
24474         \exp_after:wN ##1
24475         \exp:w \exp_after:wN \_fp_parse_expand:w
24476       \else:
```

```

24477           \exp_after:wN #3
24478           \exp_after:wN ##1
24479           \exp_after:wN ##2
24480       \fi:
24481   }
24482 }
24483 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_|:N } | \_fp_parse_infix_or:N
24484 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_&:N } & \_fp_parse_infix_and:N

```

(End definition for `\_fp_parse_infix_|:Nw` and `\_fp_parse_infix_&:Nw`.)

### 69.7.5 Ternary operator

```

\_fp_parse_infix_?:N
\_fp_parse_infix_::N
24485 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
24486 {
24487     \cs_new:Npn #1 ##1
24488     {
24489         \if_int_compare:w ##1 < \c_fp_prec_quest_int
24490             #4
24491             \exp_after:wN @
24492             \exp_after:wN #2
24493             \exp:w
24494             \_fp_parse_operand:Nw #3
24495             \exp_after:wN \_fp_parse_expand:w
24496         \else:
24497             \exp_after:wN @
24498             \exp_after:wN \use_none:n
24499             \exp_after:wN #1
24500         \fi:
24501     }
24502 }
24503 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_?:N }
24504     \_fp_ternary:NwwN \c_fp_prec_quest_int { }
24505 \exp_args:Nc \_fp_tmp:w { _fp_parse_infix_::N }
24506     \_fp_ternary_auxii:NwwN \c_fp_prec_colon_int
24507 {
24508     \msg_expandable_error:nnnn
24509     { fp } { missing } { ? } { ~for~?: }
24510 }

```

(End definition for `\_fp_parse_infix_?:N` and `\_fp_parse_infix_::N`.)

### 69.7.6 Comparisons

```

\_fp_parse_infix_<:N
\_fp_parse_infix_=:N
\_fp_parse_infix_>:N
\_fp_parse_infix_!/:N
\_fp_parse_excl_error:
\_fp_parse_compare:NNNNNNNN
\_fp_parse_compare_auxi:NNNNNNNN
\_fp_parse_compare_auxii:NNNNNN
\_fp_parse_compare_end:NNNNw
\_fp_compare:wNNNNw
24511 \cs_new:cpn { _fp_parse_infix_<:N } #1
24512     { \_fp_parse_compare>NNNNNNNN #1 1 0 0 0 0 < }
24513 \cs_new:cpn { _fp_parse_infix_=:N } #1
24514     { \_fp_parse_compare>NNNNNNNN #1 1 0 0 0 0 = }
24515 \cs_new:cpn { _fp_parse_infix_>:N } #1
24516     { \_fp_parse_compare>NNNNNNNN #1 1 0 0 0 0 > }
24517 \cs_new:cpn { _fp_parse_infix_!/:N } #1
24518     {

```

```

24519      \exp_after:wN \_fp_parse_compare:NNNNNNN
24520      \exp_after:wN #1
24521      \exp_after:wN 0
24522      \exp_after:wN 1
24523      \exp_after:wN 1
24524      \exp_after:wN 1
24525      \exp_after:wN 1
24526  }
24527 \cs_new:Npn \_fp_parse_excl_error:
24528 {
24529     \msg_expandable_error:nnn
24530     { fp } { missing } { = } { ~after~!. }
24531 }
24532 \cs_new:Npn \_fp_parse_compare:NNNNNNN #1
24533 {
24534     \if_int_compare:w #1 < \c_fp_prec_comp_int
24535         \exp_after:wN \_fp_parse_compare_auxi:NNNNNNN
24536         \exp_after:wN \_fp_parse_excl_error:
24537     \else:
24538         \exp_after:wN @
24539         \exp_after:wN \use_none:n
24540         \exp_after:wN \_fp_parse_compare:NNNNNNN
24541     \fi:
24542 }
24543 \cs_new:Npn \_fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
24544 {
24545     \if_case:w
24546         \_fp_int_eval:w \exp_after:wN ` \token_to_str:N #7 - `<
24547             \_fp_int_eval_end:
24548             \_fp_parse_compare_auxii:NNNNN #2#2#4#5#6
24549         \or: \_fp_parse_compare_auxii:NNNNN #2#3#2#5#6
24550         \or: \_fp_parse_compare_auxii:NNNNN #2#3#4#2#6
24551         \or: \_fp_parse_compare_auxii:NNNNN #2#3#4#5#2
24552         \else: #1 \_fp_parse_compare_end:NNNNw #3#4#5#6#7
24553     \fi:
24554 }
24555 \cs_new:Npn \_fp_parse_compare_auxii:NNNNN #1#2#3#4#5
24556 {
24557     \exp_after:wN \_fp_parse_compare_auxi:NNNNNNN
24558     \exp_after:wN \prg_do_nothing:
24559     \exp_after:wN #1
24560     \exp_after:wN #2
24561     \exp_after:wN #3
24562     \exp_after:wN #4
24563     \exp_after:wN #5
24564     \exp:w \exp_after:wN \_fp_parse_expand:w
24565 }
24566 \cs_new:Npn \_fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
24567 {
24568     \fi:
24569     \exp_after:wN @
24570     \exp_after:wN \_fp_parse_apply_compare:NwNNNNNwN
24571     \exp_after:wN \c_one_fp
24572     \exp_after:wN #1

```

```

24573      \exp_after:wN #2
24574      \exp_after:wN #3
24575      \exp_after:wN #4
24576      \exp:w
24577      \_fp_parse_operand:Nw \c_fp_prec_comp_int \_fp_parse_expand:w #5
24578  }
24579 \cs_new:Npn \_fp_parse_apply_compare:NwNNNNwN
24580   #1 #2@ #3 #4#5#6#7 #8@ #9
24581  {
24582    \if_int_odd:w
24583      \if_meaning:w \c_zero_fp #3
24584        0
24585      \else:
24586        \if_case:w \_fp_compare_back_any:ww #8 #2 \exp_stop_f:
24587          #5 \or: #6 \or: #7 \else: #4
24588        \fi:
24589      \fi:
24590    \exp_stop_f:
24591    \exp_after:wN \_fp_parse_apply_compare_aux>NNwN
24592    \exp_after:wN \c_one_fp
24593  \else:
24594    \exp_after:wN \_fp_parse_apply_compare_aux>NNwN
24595    \exp_after:wN \c_zero_fp
24596  \fi:
24597  #1 #8 #9
24598  }
24599 \cs_new:Npn \_fp_parse_apply_compare_aux>NNwN #1 #2 #3; #4
24600  {
24601    \if_meaning:w \_fp_parse_compare>NNNNNN #4
24602      \exp_after:wN \_fp_parse_continue_compare>NNwNN
24603      \exp_after:wN #1
24604      \exp_after:wN #2
24605      \exp:w \exp_end_continue_f:w
24606      \_fp_exp_after_o:w #3;
24607      \exp:w \exp_end_continue_f:w
24608  \else:
24609    \exp_after:wN \_fp_parse_continue:NwN
24610    \exp_after:wN #2
24611    \exp:w \exp_end_continue_f:w
24612    \exp_after:wN #1
24613    \exp:w \exp_end_continue_f:w
24614  \fi:
24615  #4 #2
24616  }
24617 \cs_new:Npn \_fp_parse_continue_compare>NNwNN #1#2 #3@ #4#5
24618  { #4 #2 #3@ #1 }

```

(End definition for `\_fp_parse_infix_<:N` and others.)

## 69.8 Tools for functions

`\_fp_parse_function_all_fp_o:fnw` Followed by `{<function name>}` `{<code>}` `<float array>` @ this checks all floats are floating point numbers (no tuples).

```

24619 \cs_new:Npn \__fp_parse_function_all_fp_o:fnw #1#2#3 @
24620   {
24621     \__fp_array_if_all_fp:nTF {#3}
24622       { #2 #3 @ }
24623       {
24624         \__fp_error:nffn { bad-args }
24625           {#1}
24626           { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#3} ; } }
24627           { }
24628         \exp_after:wN \c_nan_fp
24629       }
24630   }

```

(End definition for `\__fp_parse_function_all_fp_o:fnw`.)

```

\fp_parse_function_one_two:nnw
\fp_parse_function_one_two_error_o:w
\fp_parse_function_one_two_aux:nnw
\fp_parse_function_one_two_auxii:nnw

```

This is followed by `{<function name>} {<code>} <float array> @`. It checks that the `<float array>` consists of one or two floating point numbers (not tuples), then leaves the `<code>` (if there is one float) or its tail (if there are two floats) followed by the `<float array>`. The `<code>` should start with a single token such as `\__fp_atan_default:w` that deals with the single-float case.

The first `\__fp_if_type_fp:NTwFw` test catches the case of no argument and the case of a tuple argument. The next one distinguishes the case of a single argument (no error, just add `\c_one_fp`) from a tuple second argument. Finally check there is no further argument.

```

24631 \cs_new:Npn \__fp_parse_function_one_two:nnw #1#2#3
24632   {
24633     \__fp_if_type_fp:NTwFw
24634       #3 { } \s__fp \__fp_parse_function_one_two_error_o:w \s__fp_stop
24635       \__fp_parse_function_one_two_aux:nnw {#1} {#2} #3
24636   }
24637 \cs_new:Npn \__fp_parse_function_one_two_error_o:w #1#2#3#4 @
24638   {
24639     \__fp_error:nffn { bad-args }
24640       {#2}
24641       { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#4} ; } }
24642       { }
24643     \exp_after:wN \c_nan_fp
24644   }
24645 \cs_new:Npn \__fp_parse_function_one_two_aux:nnw #1#2 #3; #4
24646   {
24647     \__fp_if_type_fp:NTwFw
24648       #4 { }
24649       \s__fp
24650       {
24651         \if_meaning:w @ #4
24652           \exp_after:wN \use_iv:nnnn
24653           \fi:
24654           \__fp_parse_function_one_two_error_o:w
24655       }
24656       \s__fp_stop
24657       \__fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
24658   }
24659 \cs_new:Npn \__fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
24660   {

```

```

24661     \if_meaning:w @ #5 \else:
24662         \exp_after:wN \__fp_parse_function_one_two_error_o:w
24663     \fi:
24664     \use_i:nn {#1} { \use_none:n #2 } #3; #4; #5
24665 }

```

(End definition for `\__fp_parse_function_one_two:nw` and others.)

Apply #1 to all items in the following tuple and expand once afterwards. The code #1 should itself expand once after its result.

```

24666 \cs_new:Npn \__fp_tuple_map_o:nw #1 \s__fp_tuple \__fp_tuple_chk:w #2 ;
24667 {
24668     \exp_after:wN \s__fp_tuple
24669     \exp_after:wN \__fp_tuple_chk:w
24670     \exp_after:wN {
24671         \exp:w \exp_end_continue_f:w
24672         \__fp_tuple_map_loop_o:nw {#1} #2
24673         { \s__fp \prg_break: } ;
24674         \prg_break_point:
24675         \exp_after:wN } \exp_after:wN ;
24676     }
24677 \cs_new:Npn \__fp_tuple_map_loop_o:nw #1#2#3 ;
24678 {
24679     \use_none:n #2
24680     #1 #2 #3 ;
24681     \exp:w \exp_end_continue_f:w
24682     \__fp_tuple_map_loop_o:nw {#1}
24683 }

```

(End definition for `\__fp_tuple_map_o:nw` and `\__fp_tuple_map_loop_o:nw`.)

Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

24684 \cs_new:Npn \__fp_tuple_mapthread_o:nww #1
24685     \s__fp_tuple \__fp_tuple_chk:w #2 ;
24686     \s__fp_tuple \__fp_tuple_chk:w #3 ;
24687 {
24688     \exp_after:wN \s__fp_tuple
24689     \exp_after:wN \__fp_tuple_chk:w
24690     \exp_after:wN {
24691         \exp:w \exp_end_continue_f:w
24692         \__fp_tuple_mapthread_loop_o:nw {#1}
24693         #2 { \s__fp \prg_break: } ; @
24694         #3 { \s__fp \prg_break: } ;
24695         \prg_break_point:
24696         \exp_after:wN } \exp_after:wN ;
24697     }
24698 \cs_new:Npn \__fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
24699 {
24700     \use_none:n #2
24701     \use_none:n #5
24702     #1 #2 #3 ; #5 #6 ;
24703     \exp:w \exp_end_continue_f:w
24704     \__fp_tuple_mapthread_loop_o:nw {#1} #4 @
24705 }

```

(End definition for `\__fp_tuple_mapthread_o:nww` and `\__fp_tuple_mapthread_loop_o:nw`.)

## 69.9 Messages

```
24706 \msg_new:nnn { fp } { deprecated }
24707   { '#1'~deprecated;~use~'#2' }
24708 \msg_new:nnn { fp } { unknown-fp-word }
24709   { Unknown~fp~word~#1. }
24710 \msg_new:nnn { fp } { missing }
24711   { Missing~#1~inserted #2. }
24712 \msg_new:nnn { fp } { extra }
24713   { Extra~#1~ignored. }
24714 \msg_new:nnn { fp } { early-end }
24715   { Premature~end~in~fp~expression. }
24716 \msg_new:nnn { fp } { after-e }
24717   { Cannot~use~#1 after~'e'. }
24718 \msg_new:nnn { fp } { missing-number }
24719   { Missing~number~before~'#1'. }
24720 \msg_new:nnn { fp } { unknown-symbol }
24721   { Unknown~symbol~#1~ignored. }
24722 \msg_new:nnn { fp } { extra-comma }
24723   { Unexpected~comma~turned~to~nan~result. }
24724 \msg_new:nnn { fp } { no-arg }
24725   { #1~got~no~argument;~used~nan. }
24726 \msg_new:nnn { fp } { multi-arg }
24727   { #1~got~more~than~one~argument;~used~nan. }
24728 \msg_new:nnn { fp } { num-args }
24729   { #1~expects~between~#2~and~#3~arguments. }
24730 \msg_new:nnn { fp } { bad-args }
24731   { Arguments~in~#1#2~are~invalid. }
24732 \msg_new:nnn { fp } { infty-pi }
24733   { Math~command~#1 is~not~an~fp }
24734 \cs_if_exist:cT { @unexpandable@protect }
24735   {
24736     \msg_new:nnn { fp } { robust-cmd }
24737     { Robust~command~#1 invalid~in~fp~expression! }
24738   }
24739 </package>
```

# Chapter 70

## I3fp-assign implementation

24740 ⟨\*package⟩

24741 ⟨@=fp⟩

### 70.1 Assigning values

**\fp\_new:N** Floating point variables are initialized to be +0.

```
24742 \cs_new_protected:Npn \fp_new:N #1
24743   { \cs_new_eq:NN #1 \c_zero_fp }
24744 \cs_generate_variant:Nn \fp_new:N {c}
```

(End definition for `\fp_new:N`. This function is documented on page 241.)

**\fp\_set:Nn** Simply use `\_fp_parse:n` within various f-expanding assignments.

```
24745 \cs_new_protected:Npn \fp_set:Nn #1#2
24746   { \_kernel_tl_set:Nx #1 { \exp_not:f { \_fp_parse:n {#2} } } }
24747 \cs_new_protected:Npn \fp_gset:Nn #1#2
24748   { \_kernel_tl_gset:Nx #1 { \exp_not:f { \_fp_parse:n {#2} } } }
24749 \cs_new_protected:Npn \fp_const:Nn #1#2
24750   { \tl_const:Nx #1 { \exp_not:f { \_fp_parse:n {#2} } } }
24751 \cs_generate_variant:Nn \fp_set:Nn {c}
24752 \cs_generate_variant:Nn \fp_gset:Nn {c}
24753 \cs_generate_variant:Nn \fp_const:Nn {c}
```

(End definition for `\fp_set:Nn`, `\fp_gset:Nn`, and `\fp_const:Nn`. These functions are documented on page 241.)

**\fp\_set\_eq:NN** Copying a floating point is the same as copying the underlying token list.

```
24754 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
24755 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
24756 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
24757 \cs_generate_variant:Nn \fp_gset_eq:NN { c , Nc , cc }
```

(End definition for `\fp_set_eq:NN` and `\fp_gset_eq:NN`. These functions are documented on page 241.)

**\fp\_gset\_eq:cN** Setting a floating point to zero: copy `\c_zero_fp`.

```
24758 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
24759 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
24760 \cs_generate_variant:Nn \fp_zero:N { c }
24761 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 241.)

`\fp_zero_new:N` Set the floating point to zero, or define it if needed.

```
24762 \cs_new_protected:Npn \fp_zero_new:N #1
24763   { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
24764 \cs_new_protected:Npn \fp_gzero_new:N #1
24765   { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
24766 \cs_generate_variant:Nn \fp_zero_new:N { c }
24767 \cs_generate_variant:Nn \fp_gzero_new:N { c }
```

(End definition for `\fp_zero_new:N` and `\fp_gzero_new:N`. These functions are documented on page 241.)

## 70.2 Updating values

These match the equivalent functions in `\int` and `\skip`.

`\fp_add:Nn` For the sake of error recovery we should not simply set #1 to  $\#1 \pm (\#2)$ : for instance, if #2 is 0)+2, the parsing error would be raised at the last closing parenthesis rather than at the closing parenthesis in the user argument. Thus we evaluate #2 instead of just putting parentheses. As an optimization we use `\_fp_parse:n` rather than `\fp_eval:n`, which would convert the result away from the internal representation and back.

```
24768 \cs_new_protected:Npn \fp_add:Nn { \_fp_add:NNNn \fp_set:Nn + }
24769 \cs_new_protected:Npn \fp_gadd:Nn { \_fp_add:NNNn \fp_gset:Nn + }
24770 \cs_new_protected:Npn \fp_sub:Nn { \_fp_add:NNNn \fp_set:Nn - }
24771 \cs_new_protected:Npn \fp_gsub:Nn { \_fp_add:NNNn \fp_gset:Nn - }
24772 \cs_new_protected:Npn \_fp_add:NNNn #1#2#3#4
24773   { #1 #3 { #3 #2 \_fp_parse:n {#4} } }
24774 \cs_generate_variant:Nn \fp_add:Nn { c }
24775 \cs_generate_variant:Nn \fp_gadd:Nn { c }
24776 \cs_generate_variant:Nn \fp_sub:Nn { c }
24777 \cs_generate_variant:Nn \fp_gsub:Nn { c }
```

(End definition for `\fp_add:Nn` and others. These functions are documented on page 241.)

## 70.3 Showing values

`\fp_show:N` This shows the result of computing its argument by passing the right data to `\tl_show:n` or `\tl_log:n`.

```
24778 \cs_new_protected:Npn \fp_show:N { \_fp_show:NN \tl_show:n }
24779 \cs_generate_variant:Nn \fp_show:N { c }
24780 \cs_new_protected:Npn \fp_log:N { \_fp_show:NN \tl_log:n }
24781 \cs_generate_variant:Nn \fp_log:N { c }
24782 \cs_new_protected:Npn \_fp_show:NN #1#2
24783   {
24784     \_kernel_chk_tl_type:NnnT #2 { fp }
24785     {
24786       \str_if_eq:eeTF { \tl_head:N #2 } { \s__fp_tuple } { \exp_not:o #2 }
24787       {
24788         \exp_after:wN \_fp_show_validate:w #2
24789         \s__fp \_fp_chk:w ??? ; \s__fp_stop
24790       }
24791     }
24792   }
```

```

24791      }
24792      { \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
24793  }
24794 \cs_new:Npn \__fp_show_validate:w
24795     #1 \s_fp \__fp_chk:w #2#3#4#5 ; #6 \s_fp_stop
24796  {
24797      \token_if_eq_meaning:NNTF #2 1
24798      { \s_fp \__fp_chk:w #2 #3 {#4} #5 ; }
24799      { \s_fp \__fp_chk:w #2 #3 #4 #5 ; }
24800  }

```

(End definition for `\fp_show:N` and others. These functions are documented on page 249.)

`\fp_show:n` Use general tools.

```

\fp_log:n 24801 \cs_new_protected:Npn \fp_show:n
            { \msg_show_eval:Nn \fp_to_tl:n }
24803 \cs_new_protected:Npn \fp_log:n
            { \msg_log_eval:Nn \fp_to_tl:n }
24804

```

(End definition for `\fp_show:n` and `\fp_log:n`. These functions are documented on page 249.)

## 70.4 Some useful constants and scratch variables

`\c_one_fp` Some constants.

```

\c_e_fp 24805 \fp_const:Nn \c_e_fp           { 2.718 2818 2845 9045 }
24806 \fp_const:Nn \c_one_fp          { 1 }

```

(End definition for `\c_one_fp` and `\c_e_fp`. These variables are documented on page 247.)

`\c_pi_fp` We simply round  $\pi$  to and  $\pi/180$  to 16 significant digits.

```

\c_one_degree_fp 24807 \fp_const:Nn \c_pi_fp        { 3.141 5926 5358 9793 }
24808 \fp_const:Nn \c_one_degree_fp { 0.0 1745 3292 5199 4330 }

```

(End definition for `\c_pi_fp` and `\c_one_degree_fp`. These variables are documented on page 248.)

`\l_tmpa_fp` Scratch variables are simply initialized there.

```

\l_tmpb_fp 24809 \fp_new:N \l_tmpa_fp
\g_tmpa_fp 24810 \fp_new:N \l_tmpb_fp
\g_tmpb_fp 24811 \fp_new:N \g_tmpa_fp
24812 \fp_new:N \g_tmpb_fp

```

(End definition for `\l_tmpa_fp` and others. These variables are documented on page 248.)

24813 ⟨/package⟩

# Chapter 71

## I3fp-logic Implementation

```
24814  <*package>
24815  <@@=fp>
\__fp_parse_word_max:N Those functions may receive a variable number of arguments.
\__fp_parse_word_min:N
24816 \cs_new:Npn \__fp_parse_word_max:N
24817  { \__fp_parse_function>NNN \__fp_minmax_o:Nw 2 }
24818 \cs_new:Npn \__fp_parse_word_min:N
24819  { \__fp_parse_function>NNN \__fp_minmax_o:Nw 0 }

(End definition for \__fp_parse_word_max:N and \__fp_parse_word_min:N.)
```

### 71.1 Syntax of internal functions

- `\__fp_compare_npos:nwnw` {`<exp01>`} `<body1>` ; {`<exp02>`} `<body2>` ;
- `\__fp_minmax_o:Nw` `<sign>` `<floating point array>`
- `\__fp_not_o:w` ? `<floating point array>` (with one floating point number only)
- `\__fp_&_o:ww` `<floating point>` `<floating point>`
- `\__fp_|_o:ww` `<floating point>` `<floating point>`
- `\__fp_ternary:NwwN`, `\__fp_ternary_auxi:NwwN`, `\__fp_ternary_auxii:NwwN`  
have to be understood.

### 71.2 Tests

```
\fp_if_exist_p:N Copies of the cs functions defined in I3basics.
\fp_if_exist_p:c
24820 \prg_new_eq_conditional>NNN \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
24821 \prg_new_eq_conditional>NNN \fp_if_exist:c \cs_if_exist:c { TF , T , F , p }

\fp_if_exist:cTF
(End definition for \fp_if_exist:NTF. This function is documented on page 243.)
```

```

\fp_if_nan_p:n Evaluate and check if the result is a floating point of the same kind as nan.
\fp_if_nan:nTF 24822 \prg_new_conditional:Npnn \fp_if_nan:n #1 { TF , T , F , p }
24823 {
24824     \if:w 3 \exp_last_unbraced:Nf \__fp_kind:w { \__fp_parse:n {#1} }
24825         \prg_return_true:
24826     \else:
24827         \prg_return_false:
24828     \fi:
24829 }

```

(End definition for `\fp_if_nan:nTF`. This function is documented on page [306](#).)

## 71.3 Comparison

`\fp_compare_p:n` Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with  $\pm 0$ . Tuples are true.

```

\fp_compare:nTF 24830 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
24831 {
24832     \exp_after:wN \__fp_compare_return:w
24833     \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
24834 }
24835 \cs_new:Npn \__fp_compare_return:w #1#2#3;
24836 {
24837     \if_charcode:w 0
24838         \__fp_if_type_fp:NTwFw
24839             #1 { \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
24840             \s__fp 1 \s__fp_stop
24841         \prg_return_false:
24842     \else:
24843         \prg_return_true:
24844     \fi:
24845 }

```

(End definition for `\fp_compare:nTF` and `\__fp_compare_return:w`. This function is documented on page [245](#).)

`\fp_compare_p:nNn` Evaluate #1 and #3, using an auxiliary to expand both, and feed the two floating point numbers swapped to `\__fp_compare_back_any:ww`, defined below. Compare the result with '#2-'=, which is -1 for <, 0 for =, 1 for > and 2 for ?.

```

\fp_compare:nNn 24846 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
24847 {
24848     \if_int_compare:w
24849         \exp_after:wN \__fp_compare_aux:wn
24850         \exp:w \exp_end_continue_f:w \__fp_parse:n {#1} {#3}
24851         = \__fp_int_eval:w '#2 - '=' \__fp_int_eval_end:
24852         \prg_return_true:
24853     \else:
24854         \prg_return_false:
24855     \fi:
24856 }
24857 \cs_new:Npn \__fp_compare_aux:wn #1; #2
24858 {
24859     \exp_after:wN \__fp_compare_back_any:ww

```

```

24860     \exp:w \exp_end_continue_f:w \_fp_parse:n {#2} #1;
24861 }

```

(End definition for `\fp_compare:nNnTF` and `\_fp_compare_aux:wn`. This function is documented on page 244.)

`\_fp_compare_back_any:ww`  
`\_fp_compare_back:ww`  
`\_fp_compare_nan:ww`

`\_fp_compare_back_any:ww ⟨y⟩ ; ⟨x⟩ ;`

Expands (in the same way as `\int_eval:n`) to  $-1$  if  $x < y$ ,  $0$  if  $x = y$ ,  $1$  if  $x > y$ , and  $2$  otherwise (denoted as  $x?y$ ). If either operand is `nan`, stop the comparison with `\_fp_compare_nan:w` returning  $2$ . If  $x$  is negative, swap the outputs  $1$  and  $-1$  (*i.e.*,  $>$  and  $<$ ); we can henceforth assume that  $x \geq 0$ . If  $y \geq 0$ , and they have the same type, either they are normal and we compare them with `\_fp_compare_npos:nwnw`, or they are equal. If  $y \geq 0$ , but of a different type, the highest type is a larger number. Finally, if  $y \leq 0$ , then  $x > y$ , unless both are zero.

```

24862 \cs_new:Npn \_fp_compare_back_any:ww #1#2; #3
24863 {
24864     \_fp_if_type_fp:NTwFw
24865         #1 { \_fp_if_type_fp:NTwFw #3 \use_i:nn \s_fp \use_i:nn \s_fp_stop }
24866         \s_fp \use_i:nn \s_fp_stop
24867     \_fp_compare_back:ww
24868 {
24869     \cs:w
24870         _fp
24871         \_fp_type_from_scan:N #1
24872         _compare_back
24873         \_fp_type_from_scan:N #3
24874         :ww
24875         \cs_end:
24876     }
24877     #1#2 ; #3
24878 }
24879 \cs_new:Npn \_fp_compare_back:ww
24880     \s_fp \_fp_chk:w #1 #2 #3;
24881     \s_fp \_fp_chk:w #4 #5 #6;
24882 {
24883     \int_value:w
24884         \if_meaning:w 3 #1 \exp_after:wN \_fp_compare_nan:w \fi:
24885         \if_meaning:w 3 #4 \exp_after:wN \_fp_compare_nan:w \fi:
24886         \if_meaning:w 2 #5 - \fi:
24887         \if_meaning:w #2 #5
24888             \if_meaning:w #1 #4
24889                 \if_meaning:w 1 #1
24890                     \_fp_compare_npos:nwnw #6; #3;
24891                 \else:
24892                     0
24893                 \fi:
24894             \else:
24895                 \if_int_compare:w #4 < #1 - \fi: 1
24896                 \fi:
24897             \else:
24898                 \if_int_compare:w #1#4 = \c_zero_int
24899                     0
24900                 \else:
24901                     1

```

```

24902         \fi:
24903     \fi:
24904   \exp_stop_f:
24905 }
24906 \cs_new:Npn \__fp_compare_nan:w #1 \fi: \exp_stop_f: { 2 \exp_stop_f: }

(End definition for \__fp_compare_back_any:ww, \__fp_compare_back:ww, and \__fp_compare_nan:w.)

```

\\_\_fp\_compare\_back\_tuple:ww  
\\_\_fp\_tuple\_compare\_back:ww  
\\_\_fp\_tuple\_compare\_back\_tuple:ww  
\\_\_fp\_tuple\_compare\_back\_loop:ww

Tuple and floating point numbers are not comparable so return 2 in mixed cases or when tuples have a different number of items. Otherwise compare pairs of items with \\_\_fp\_compare\_back\_any:ww and if any don't match return 2 (as \int\_value:w 02 \exp\_stop\_f:).

```

24907 \cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
24908 \cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
24909 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
24910   \s_fp_tuple \__fp_tuple_chk:w #1;
24911   \s_fp_tuple \__fp_tuple_chk:w #2;
24912 {
24913   \int_compare:nNnTF { \__fp_array_count:n {#1} } =
24914     { \__fp_array_count:n {#2} }
24915   {
24916     \int_value:w 0
24917     \__fp_tuple_compare_back_loop:w
24918       #1 { \s_fp \prg_break: } ; @
24919       #2 { \s_fp \prg_break: } ;
24920     \prg_break_point:
24921     \exp_stop_f:
24922   }
24923   { 2 }
24924 }
24925 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 @ #4#5 ;
24926 {
24927   \use_none:n #1
24928   \use_none:n #4
24929   \if_int_compare:w
24930     \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = \c_zero_int
24931   \else:
24932     2 \exp_after:wN \prg_break:
24933   \fi:
24934   \__fp_tuple_compare_back_loop:w #3 @
24935 }

(End definition for \__fp_compare_back_tuple:ww and others.)

```

\\_\_fp\_compare\_npos:nwnw  
\\_\_fp\_compare\_significand:nnnnnnnn

$\_\_fp\_compare\_npos:nwnw \{ \langle expo_1 \rangle \} \langle body_1 \rangle ; \{ \langle expo_2 \rangle \} \langle body_2 \rangle ;$

Within an \int\_value:w ... \exp\_stop\_f: construction, this expands to 0 if the two numbers are equal, -1 if the first is smaller, and 1 if the first is bigger. First compare the exponents: the larger one denotes the larger number. If they are equal, we must compare significands. If both the first 8 digits and the next 8 digits coincide, the numbers are equal. If only the first 8 digits coincide, the next 8 decide. Otherwise, the first 8 digits are compared.

```

24936 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
24937 {
24938   \if_int_compare:w #1 = #3 \exp_stop_f:

```

```

24939     \_fp_compare_significand:nnnnnnnn #2 #4
24940     \else:
24941         \if_int_compare:w #1 < #3 - \fi: 1
24942         \fi:
24943     }
24944 \cs_new:Npn \_fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
24945 {
24946     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
24947         \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
24948             0
24949         \else:
24950             \if_int_compare:w #3#4 < #7#8 - \fi: 1
24951             \fi:
24952         \else:
24953             \if_int_compare:w #1#2 < #5#6 - \fi: 1
24954             \fi:
24955     }

```

(End definition for `\_fp_compare_npos:nwnw` and `\_fp_compare_significand:nnnnnnnn`.)

## 71.4 Floating point expression loops

`\fp_do_until:nn` These are quite easy given the above functions. The `do_until` and `do_while` versions execute the body, then test. The `until_do` and `while_do` do it the other way round.

```

24956 \cs_new:Npn \fp_do_until:nn #1#2
24957 {
24958     #2
24959     \fp_compare:nF {#1}
24960         { \fp_do_until:nn {#1} {#2} }
24961     }
24962 \cs_new:Npn \fp_do_while:nn #1#2
24963 {
24964     #2
24965     \fp_compare:nT {#1}
24966         { \fp_do_while:nn {#1} {#2} }
24967     }
24968 \cs_new:Npn \fp_until_do:nn #1#2
24969 {
24970     \fp_compare:nF {#1}
24971         {
24972             #2
24973             \fp_until_do:nn {#1} {#2}
24974         }
24975     }
24976 \cs_new:Npn \fp_while_do:nn #1#2
24977 {
24978     \fp_compare:nT {#1}
24979         {
24980             #2
24981             \fp_while_do:nn {#1} {#2}
24982         }
24983     }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 246.)

\fp\_do\_until:nNnn As above but not using the nNn syntax.

```

24984 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
24985 {
24986     #4
24987     \fp_compare:nNnF {#1} #2 {#3}
24988     { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
24989 }
24990 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
24991 {
24992     #4
24993     \fp_compare:nNnT {#1} #2 {#3}
24994     { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
24995 }
24996 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
24997 {
24998     \fp_compare:nNnF {#1} #2 {#3}
24999     {
25000         #4
25001         \fp_until_do:nNnn {#1} #2 {#3} {#4}
25002     }
25003 }
25004 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
25005 {
25006     \fp_compare:nNnT {#1} #2 {#3}
25007     {
25008         #4
25009         \fp_while_do:nNnn {#1} #2 {#3} {#4}
25010     }
25011 }
```

(End definition for \fp\_do\_until:nNnn and others. These functions are documented on page 245.)

\fp\_step\_function:nnnN The approach here is somewhat similar to \int\_step\_function:nnnN. There are two subtleties: we use the internal parser \fp\_parse:n to avoid converting back and forth from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

25012 \cs_new:Npn \fp_step_function:nnnN #1#2#3
25013 {
25014     \exp_after:wN \fp_step:wwwN
25015     \exp:w \exp_end_continue_f:w \fp_parse_o:n {#1}
25016     \exp:w \exp_end_continue_f:w \fp_parse_o:n {#2}
25017     \exp:w \exp_end_continue_f:w \fp_parse:n {#3}
25018 }
25019 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
25020 %     \end{macrocode}
25021 % Only floating point numbers (not tuples) are allowed arguments.
25022 % Only \enquote{normal} floating points (not $\pm 0$,
25023 % $\pm\text{inf}$, $\text{nan}$) can be used as step; if positive,
25024 % call \cs{\fp_step:NnnnnN} with argument $|>|$ otherwise $\sim|<|$. This
25025 % function has one more argument than its integer counterpart, namely
25026 % the previous value, to catch the case where the loop has made no
25027 % progress. Conversion to decimal is done just before calling the
25028 % user's function.
25029 %     \begin{macrocode}
```

```

25030 \cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
25031   {
25032     \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \s__fp_stop
25033     \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \s__fp_stop
25034     \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \s__fp_stop
25035     \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
25036     \prg_break_point:
25037     \use:n
25038       {
25039         \__fp_error:nfff { step-tuple } { \fp_to_tl:n { #1#2 ; } }
25040           { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }
25041       }
25042     }
25043 \cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
25044   {
25045     \token_if_eq_meaning:NNTF #2 1
25046       {
25047         \token_if_eq_meaning:NNTF #3 0
25048           { \__fp_step:NnnnnN > }
25049           { \__fp_step:NnnnnN < }
25050       }
25051     {
25052       \token_if_eq_meaning:NNTF #2 0
25053         {
25054           \msg_expandable_error:nnn { kernel }
25055             { zero-step } {#6}
25056         }
25057         {
25058           \__fp_error:nnfn { bad-step } { }
25059             { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
25060         }
25061       \use_none:nnnn
25062     }
25063     { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6
25064   }
25065 \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
25066   {
25067     \fp_compare:nNnTF {#2} = {#3}
25068       {
25069         \__fp_error:nffn { tiny-step }
25070           { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
25071       }
25072       {
25073         \fp_compare:nNnF {#2} #1 {#5}
25074           {
25075             \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
25076             \__fp_step:NfnnnN
25077               #1 { \__fp_parse:n { #2 + #4 } } {#2} {#4} {#5} #6
25078           }
25079       }
25080     }
25081 \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }

```

(End definition for `\fp_step_function:nnnN` and others. This function is documented on page 247.)

```
\fp_step_inline:nnn
\fp_step_variable:nnNn
```

```
\_fp_step>NNnnnn
```

As for \int\_step\_inline:nnnn, create a global function and apply it, following up with a break point.

```
25082 \cs_new_protected:Npn \fp_step_inline:nnnn
25083 {
25084     \int_gincr:N \g_kernel_prg_map_int
25085     \exp_args:NNc \_fp_step>NNnnnn
25086     \cs_gset_protected:Npn
25087     { _fp_map_ \int_use:N \g_kernel_prg_map_int :w }
25088 }
25089 \cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5
25090 {
25091     \int_gincr:N \g_kernel_prg_map_int
25092     \exp_args:NNc \_fp_step>NNnnnn
25093     \cs_gset_protected:Npx
25094     { _fp_map_ \int_use:N \g_kernel_prg_map_int :w }
25095     {#1} {#2} {#3}
25096     {
25097         \tl_set:Nn \exp_not:N #4 {##1}
25098         \exp_not:n {#5}
25099     }
25100 }
25101 \cs_new_protected:Npn \_fp_step>NNnnnn #1#2#3#4#5#6
25102 {
25103     #1 #2 ##1 {#6}
25104     \fp_step_function:nnnN {#3} {#4} {#5} #2
25105     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g_kernel_prg_map_int }
25106 }
```

(End definition for \fp\_step\_inline:nnnn, \fp\_step\_variable:nnnNn, and \\_fp\_step>NNnnnn. These functions are documented on page 247.)

```
25107 \msg_new:nnn { fp } { step-tuple }
25108     { Tuple~argument~in~fp_step_...~{#1}{#2}{#3}. }
25109 \msg_new:nnn { fp } { bad-step }
25110     { Invalid~step-size~#2~for~function~#3. }
25111 \msg_new:nnn { fp } { tiny-step }
25112     { Tiny~step~size~(#1+#2=#1)~for~function~#3. }
```

## 71.5 Extrema

```
\_fp_minmax_o:Nw
\fp_minmax_aux_o:Nw
```

First check all operands are floating point numbers. The argument #1 is 2 to find the maximum of an array #2 of floating point numbers, and 0 to find the minimum. We read numbers sequentially, keeping track of the largest (smallest) number found so far. If numbers are equal (for instance  $\pm 0$ ), the first is kept. We append  $-\infty$  ( $\infty$ ), for the case of an empty array. Since no number is smaller (larger) than that, this additional item only affects the maximum (minimum) in the case of `max()` and `min()` with no argument. The weird fp-like trailing marker breaks the loop correctly: see the precise definition of \\_fp\_minmax\_loop:Nww.

```
25113 \cs_new:Npn \_fp_minmax_o:Nw #1
25114 {
25115     \_fp_parse_function_all_fp_o:fnw
25116     { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
25117     { \_fp_minmax_aux_o:Nw #1 }
```

```

25118     }
25119   \cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
25120   {
25121     \if_meaning:w 0 #1
25122       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
25123     \else:
25124       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
25125     \fi:
25126     #2
25127     \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
25128     \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
25129   }

```

(End definition for `\__fp_minmax_o:Nw` and `\__fp_minmax_aux_o:Nw`.)

`\__fp_minmax_loop:Nww` The first argument is `-` or `+` to denote the case where the currently largest (smallest) number found (first floating point argument) should be replaced by the new number (second floating point argument). If the new number is `nan`, keep that as the extremum, unless that extremum is already a `nan`. Otherwise, compare the two numbers. If the new number is larger (in the case of `max`) or smaller (in the case of `min`), the test yields `true`, and we keep the second number as a new maximum; otherwise we keep the first number. Then loop.

```

25130 \cs_new:Npn \__fp_minmax_loop:Nww
25131   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
25132   {
25133     \if_meaning:w 3 #4
25134       \if_meaning:w 3 #2
25135         \__fp_minmax_auxi:ww
25136       \else:
25137         \__fp_minmax_auxii:ww
25138       \fi:
25139     \else:
25140       \if_int_compare:w
25141         \__fp_compare_back:ww
25142           \s__fp \__fp_chk:w #4#5;
25143           \s__fp \__fp_chk:w #2#3;
25144           = #1 1 \exp_stop_f:
25145           \__fp_minmax_auxii:ww
25146         \else:
25147           \__fp_minmax_auxi:ww
25148         \fi:
25149     \fi:
25150     \__fp_minmax_loop:Nww #1
25151       \s__fp \__fp_chk:w #2#3;
25152       \s__fp \__fp_chk:w #4#5;
25153   }

```

(End definition for `\__fp_minmax_loop:Nww`.)

`\__fp_minmax_auxi:ww` Keep the first/second number, and remove the other.

```

25154 \cs_new:Npn \__fp_minmax_auxi:ww #1 \fi: \fi: #2 \s__fp #3 ; \s__fp #4;
25155   { \fi: \fi: #2 \s__fp #3 ; }
25156 \cs_new:Npn \__fp_minmax_auxii:ww #1 \fi: \fi: #2 \s__fp #3 ;
25157   { \fi: \fi: #2 }

```

(End definition for `\_fp_minmax_auxi:ww` and `\_fp_minmax_auxii:ww`.)

`\_fp_minmax_break_o:w`

This function is called from within an `\if_meaning:w` test. Skip to the end of the tests, close the current test with `\fi:`, clean up, and return the appropriate number with one post-expansion.

```
25158 \cs_new:Npn \_fp_minmax_break_o:w #1 \fi: \fi: #2 \s_fp #3; #4;
25159   { \fi: \_fp_exp_after_o:w \s_fp #3; }
```

(End definition for `\_fp_minmax_break_o:w`.)

## 71.6 Boolean operations

`\_fp_not_o:w`  
`\_fp_tuple_not_o:w`

Return `true` or `false`, with two expansions, one to exit the conditional, and one to please `\l3fp-parse`. The first argument is provided by `\l3fp-parse` and is ignored.

```
25160 \cs_new:Npn \_fp_not_o:w #1 \s_fp \_fp_chk:w #2#3; @
25161   {
25162     \if_meaning:w 0 #2
25163       \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
25164     \else:
25165       \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
25166     \fi:
25167   }
25168 \cs_new:Npn \_fp_tuple_not_o:w #1 @ { \exp_after:wN \c_zero_fp }
```

(End definition for `\_fp_not_o:w` and `\_fp_tuple_not_o:w`.)

`\_fp_&_o:ww`  
`\_fp_tuple_&_o:ww`  
`\_fp_&_tuple_o:ww`  
`\_fp_tuple_&_tuple_o:ww`  
  `\_fp_|_o:ww`

For `and`, if the first number is zero, return it (with the same sign). Otherwise, return the second one. For `or`, the logic is reversed: if the first number is non-zero, return it, otherwise return the second number: we achieve that by hi-jacking `\_fp_&_o:ww`, inserting an extra argument, `\else:`, before `\s_fp`. In all cases, expand after the floating point number.

```
25169 \group_begin:
25170   \char_set_catcode_letter:N &
25171   \char_set_catcode_letter:N |
25172   \cs_new:Npn \_fp_&_o:ww #1 \s_fp \_fp_chk:w #2#3;
25173   {
25174     \if_meaning:w 0 #2 #1
25175       \_fp_and_return:wNw \s_fp \_fp_chk:w #2#3;
25176     \fi:
25177       \_fp_exp_after_o:w
25178   }
25179 \cs_new:Npn \_fp_&_tuple_o:ww #1 \s_fp \_fp_chk:w #2#3;
25180   {
25181     \if_meaning:w 0 #2 #1
25182       \_fp_and_return:wNw \s_fp \_fp_chk:w #2#3;
25183     \fi:
25184       \_fp_exp_after_tuple_o:w
25185   }
25186 \cs_new:Npn \_fp_tuple_&_o:ww #1; { \_fp_exp_after_o:w }
25187 \cs_new:Npn \_fp_tuple_&_tuple_o:ww #1; { \_fp_exp_after_tuple_o:w }
25188 \cs_new:Npn \_fp_|_o:ww { \_fp_&_o:ww \else: }
25189 \cs_new:Npn \_fp_|_tuple_o:ww { \_fp_&_tuple_o:ww \else: }
25190 \cs_new:Npn \_fp_tuple_|_o:ww #1; #2; { \_fp_exp_after_tuple_o:w #1; }
```

```

25191   \cs_new:Npn \__fp_tuple_1_tuple_o:ww #1; #2;
25192     { \__fp_exp_after_tuple_o:w #1; }
25193 \group_end:
25194 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
25195   { \fi: \__fp_exp_after_o:w #1; }

```

(End definition for `\__fp_&_o:ww` and others.)

## 71.7 Ternary operator

```

\__fp_ternary:NwwN
\__fp_ternary_auxi:NwwN
\__fp_ternary_auxii:NwwN

```

The first function receives the test and the true branch of the ?: ternary operator. It calls `\__fp_ternary_auxii:NwwN` if the test branch is a floating point number  $\pm 0$ , and otherwise calls `\__fp_ternary_auxi:NwwN`. These functions select one of their two arguments.

```

25196 \cs_new:Npn \__fp_ternary:NwwN #1 #2#3@ #4@ #5
25197   {
25198     \if_meaning:w \__fp_parse_infix_::N #5
25199       \if_charcode:w 0
25200         \__fp_if_type_fp:NTwFw
25201           #2 { \use_i:nN \__fp_use_i_delimit_by_s_stop:nw #3 \s__fp_stop }
25202             \s__fp 1 \s__fp_stop
25203           \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwwN
25204         \else:
25205           \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwwN
25206         \fi:
25207         \exp_after:wN #1
25208         \exp:w \exp_end_continue_f:w
25209         \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
25210         \exp_after:wN @
25211         \exp:w
25212           \__fp_parse_operand:Nw \c__fp_prec_colon_int
25213             \__fp_parse_expand:w
25214         \else:
25215           \msg_expandable_error:nnnn
25216             { fp } { missing } { : } { ~for~?: }
25217           \exp_after:wN \__fp_parse_continue:NwN
25218           \exp_after:wN #1
25219           \exp:w \exp_end_continue_f:w
25220           \__fp_exp_after_array_f:w #4 \s__fp_expr_stop
25221           \exp_after:wN #5
25222           \exp_after:wN #1
25223         \fi:
25224       }
25225 \cs_new:Npn \__fp_ternary_auxi:NwwN #1#2@#3@#4
25226   {
25227     \exp_after:wN \__fp_parse_continue:NwN
25228     \exp_after:wN #1
25229     \exp:w \exp_end_continue_f:w
25230     \__fp_exp_after_array_f:w #2 \s__fp_expr_stop
25231     #4 #1
25232   }
25233 \cs_new:Npn \__fp_ternary_auxii:NwwN #1#2@#3@#4
25234   {

```

```
25235     \exp_after:wN \__fp_parse_continue:NwN
25236     \exp_after:wN #1
25237     \exp:w \exp_end_continue_f:w
25238     \__fp_exp_after_array_f:w #3 \s__fp_expr_stop
25239     #4 #1
25240 }
```

(End definition for `\__fp_ternary:NwwN`, `\__fp_ternary_auxi:NwwN`, and `\__fp_ternary_auxii:NwwN`.)  
25241 ⟨/package⟩

# Chapter 72

## I3fp-basics Implementation

```
25242  <*package>
25243  <@@=fp>
```

The I3fp-basics module implements addition, subtraction, multiplication, and division of two floating points, and the absolute value and sign-changing operations on one floating point. All operations implemented in this module yield the outcome of rounding the infinitely precise result of the operation to the nearest floating point.

Some algorithms used below end up being quite similar to some described in “What Every Computer Scientist Should Know About Floating Point Arithmetic”, by David Goldberg, which can be found at <http://cr.yp.to/2005-590/goldberg.pdf>.

Unary functions.

```
25244 \cs_new:Npn \__fp_parse_word_abs:N
25245   { \__fp_parse_unary_function:NNN \__fp_set_sign_o:w 0 }
25246 \cs_new:Npn \__fp_parse_word_logb:N
25247   { \__fp_parse_unary_function:NNN \__fp_logb_o:w ? }
25248 \cs_new:Npn \__fp_parse_word_sign:N
25249   { \__fp_parse_unary_function:NNN \__fp_sign_o:w ? }
25250 \cs_new:Npn \__fp_parse_word_sqrt:N
25251   { \__fp_parse_unary_function:NNN \__fp_sqrt_o:w ? }
```

(End definition for `\__fp_parse_word_abs:N` and others.)

### 72.1 Addition and subtraction

We define here two functions, `\__fp_-_o:ww` and `\__fp+_o:ww`, which perform the subtraction and addition of their two floating point operands, and expand the tokens following the result once.

A more obscure function, `\__fp_add_big_i_o:wNww`, is used in I3fp-expo.

The logic goes as follows:

- `\__fp_-_o:ww` calls `\__fp+_o:ww` to do the work, with the sign of the second operand flipped;
- `\__fp+_o:ww` dispatches depending on the type of floating point, calling specialized auxiliaries;

- in all cases except summing two normal floating point numbers, we return one or the other operands depending on the signs, or detect an invalid operation in the case of  $\infty - \infty$ ;
- for normal floating point numbers, compare the signs;
- to add two floating point numbers of the same sign or of opposite signs, shift the significand of the smaller one to match the bigger one, perform the addition or subtraction of significands, check for a carry, round, and pack using the `\_fp-basics_pack_...` functions.

The trickiest part is to round correctly when adding or subtracting normal floating point numbers.

### 72.1.1 Sign, exponent, and special numbers

`\_fp-_o:ww` The `\_fp+_o:ww` auxiliary has a hook: it takes one argument between the first `\s_fp` and `\_fp_chk:w`, which is applied to the sign of the second operand. Positioning the hook there means that `\_fp+_o:ww` can still perform the sanity check that it was followed by `\s_fp`.

```
25252 \cs_new:cpx { __fp-_o:ww } \s_fp
25253 {
25254     \exp_not:c { __fp+_o:ww }
25255     \exp_not:n { \s_fp \_fp_neg_sign:N }
25256 }
```

(End definition for `\_fp-_o:ww`.)

`\_fp+_o:ww` This function is either called directly with an empty #1 to compute an addition, or it is called by `\_fp-_o:ww` with `\_fp_neg_sign:N` as #1 to compute a subtraction, in which case the second operand's sign should be changed. If the `<types>` #2 and #4 are the same, dispatch to case #2 (0, 1, 2, or 3), where we call specialized functions: thanks to `\int_value:w`, those receive the tweaked `<sign2>` (expansion of #1#5) as an argument. If the `<types>` are distinct, the result is simply the floating point number with the highest `<type>`. Since case 3 (used for two `nan`) also picks the first operand, we can also use it when `<type1>` is greater than `<type2>`. Also note that we don't need to worry about `<sign2>` in that case since the second operand is discarded.

```
25257 \cs_new:cpn { __fp+_o:ww }
25258     \s_fp #1 \_fp_chk:w #2 #3 ; \s_fp \_fp_chk:w #4 #5
25259 {
25260     \if_case:w
25261         \if_meaning:w #2 #4
25262             #2
25263         \else:
25264             \if_int_compare:w #2 > #4 \exp_stop_f:
25265                 3
25266             \else:
25267                 4
25268             \fi:
25269         \fi:
25270         \exp_stop_f:
25271             \exp_after:wN \_fp_add_zeros_o:Nww \int_value:w
25272     \or: \exp_after:wN \_fp_add_normal_o:Nww \int_value:w
```

```

25273     \or:   \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
25274     \or:   \__fp_case_return_i_o:ww
25275     \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
25276     \fi:
25277     #1 #5
25278     \s__fp \__fp_chk:w #2 #3 ;
25279     \s__fp \__fp_chk:w #4 #5
25280 }

```

(End definition for `\__fp_+_o:ww.`)

`\__fp_add_return_ii_o:Nww` Ignore the first operand, and return the second, but using the sign #1 rather than #4. As usual, expand after the floating point.

```

25281 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
25282 { \__fp_exp_after_o:w \s__fp \__fp_chk:w #3 #1 }

```

(End definition for `\__fp_add_return_ii_o:Nww.`)

`\__fp_add_zeros_o:Nww` Adding two zeros yields `\c_zero_fp`, except if both zeros were  $-0$ .

```

25283 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
25284 {
25285     \if_int_compare:w #2 #1 = 20 \exp_stop_f:
25286     \exp_after:wN \__fp_add_return_ii_o:Nww
25287     \else:
25288         \__fp_case_return_i_o:ww
25289     \fi:
25290     #1
25291     \s__fp \__fp_chk:w 0 #2
25292 }

```

(End definition for `\__fp_add_zeros_o:Nww.`)

`\__fp_add_inf_o:Nww` If both infinities have the same sign, just return that infinity, otherwise, it is an invalid operation. We find out if that invalid operation is an addition or a subtraction by testing whether the tweaked  $\langle sign_2 \rangle$  (#1) and the  $\langle sign_2 \rangle$  (#4) are identical.

```

25293 \cs_new:Npn \__fp_add_inf_o:Nww
25294     #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
25295 {
25296     \if_meaning:w #1 #2
25297         \__fp_case_return_i_o:ww
25298     \else:
25299         \__fp_case_use:nw
25300         {
25301             \exp_last_unbraced:Nf \__fp_invalid_operation_o:Nww
25302             { \token_if_eq_meaning:NNTF #1 #4 + - }
25303         }
25304         \fi:
25305         \s__fp \__fp_chk:w 2 #2 #3;
25306         \s__fp \__fp_chk:w 2 #4
25307 }

```

(End definition for `\__fp_add_inf_o:Nww.`)

```
\_\_fp\_add\_normal\_o:Nww
\_\_fp\_add\_normal\_o:Nww <sign2> \s\_fp \_\_fp\_chk:w 1 <sign1> <exp1>
<body1> ; \s\_fp \_\_fp\_chk:w 1 <initial sign2> <exp2> <body2> ;
```

We now have two normal numbers to add, and we have to check signs and exponents more carefully before performing the addition.

```
25308 \cs_new:Npn \_\_fp\_add\_normal\_o:Nww #1 \s\_fp \_\_fp\_chk:w 1 #2
25309 {
25310     \if_meaning:w #1#2
25311         \exp_after:wN \_\_fp\_add\_npos\_o:NnwNnw
25312     \else:
25313         \exp_after:wN \_\_fp\_sub\_npos\_o:NnwNnw
25314     \fi:
25315     #2
25316 }
```

(End definition for `\_\_fp\_add\_normal\_o:Nww`.)

### 72.1.2 Absolute addition

In this subsection, we perform the addition of two positive normal numbers.

```
\_\_fp\_add\_npos\_o:NnwNnw
\_\_fp\_add\_npos\_o:NnwNnw <sign1> <exp1> <body1> ; \s\_fp \_\_fp\_chk:w 1
<initial sign2> <exp2> <body2> ;
```

Since we are doing an addition, the final sign is `<sign1>`. Start an `\_\_fp\_int\_eval:w`, responsible for computing the exponent: the result, and the `<final sign>` are then given to `\_\_fp\_sanitize:Nw` which checks for overflow. The exponent is computed as the largest exponent `#2` or `#5`, incremented if there is a carry. To add the significands, we decimate the smaller number by the difference between the exponents. This is done by `\_\_fp\_add\_big\_i:wNww` or `\_\_fp\_add\_big\_ii:wNww`. We need to bring the final sign with us in the midst of the calculation to round properly at the end.

```
25317 \cs_new:Npn \_\_fp\_add\_npos\_o:NnwNnw #1#2#3 ; \s\_fp \_\_fp\_chk:w 1 #4 #5
25318 {
25319     \exp_after:wN \_\_fp\_sanitize:Nw
25320     \exp_after:wN #1
25321     \int_value:w \_\_fp\_int\_eval:w
25322     \if_int_compare:w #2 > #5 \exp_stop_f:
25323         #2
25324         \exp_after:wN \_\_fp\_add\_big\_i_o:wNww \int_value:w -
25325     \else:
25326         #5
25327         \exp_after:wN \_\_fp\_add\_big\_ii_o:wNww \int_value:w
25328     \fi:
25329     \_\_fp\_int\_eval:w #5 - #2 ; #1 #3;
25330 }
```

(End definition for `\_\_fp\_add\_npos\_o:NnwNnw`.)

```
\_\_fp\_add\_big\_i\_o:wNww <shift> ; <final sign> <body1> ; <body2> ;
```

Used in `\l3fp-expo`. Shift the significand of the small number, then add with `\_\_fp\_add\_significand\_o:NnnwnnnnnN`.

```
25331 \cs_new:Npn \_\_fp\_add\_big\_i\_o:wNww #1; #2 #3; #4;
25332 {
25333     \_\_fp\_decimate:nNnnnn {#1}
25334     \_\_fp\_add\_significand\_o:NnnwnnnnnN
```

```

25335      #4
25336      #3
25337      #2
25338  }
25339 \cs_new:Npn \__fp_add_big_ii_o:wNww #1; #2 #3; #4;
25340  {
25341      \__fp_decimate:nNnnnn {#1}
25342          \__fp_add_significand_o:NnnwnnnnN
25343          #3
25344          #4
25345          #2
25346  }

```

(End definition for `\__fp_add_big_i_o:wNww` and `\__fp_add_big_ii_o:wNww`.)

```

\__fp_add_significand_o:NnnwnnnnN
\__fp_add_significand_pack:NNNNNNN
\__fp_add_significand_test_o:N

```

To round properly, we must know at which digit the rounding should occur. This requires to know whether the addition produces an overall carry or not. Thus, we do the computation now and check for a carry, then go back and do the rounding. The rounding may cause a carry in very rare cases such as  $0.99\cdots 95 \rightarrow 1.00\cdots 0$ , but this situation always give an exact power of 10, for which it is easy to correct the result at the end.

```

25347 \cs_new:Npn \__fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
25348  {
25349      \exp_after:wN \__fp_add_significand_test_o:N
25350      \int_value:w \__fp_int_eval:w 1#5#6 + #2
25351          \exp_after:wN \__fp_add_significand_pack:NNNNNNN
25352          \int_value:w \__fp_int_eval:w 1#7#8 + #3 ; #1
25353  }
25354 \cs_new:Npn \__fp_add_significand_pack:NNNNNNN #1 #2#3#4#5#6#7
25355  {
25356      \if_meaning:w 2 #1
25357          + 1
25358      \fi:
25359      ; #2 #3 #4 #5 #6 #7 ;
25360  }
25361 \cs_new:Npn \__fp_add_significand_test_o:N #1
25362  {
25363      \if_meaning:w 2 #1
25364          \exp_after:wN \__fp_add_significand_carry_o:wwwNN
25365      \else:
25366          \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
25367      \fi:
25368  }

```

(End definition for `\__fp_add_significand_o:NnnwnnnnN`, `\__fp_add_significand_pack:NNNNNNN`, and `\__fp_add_significand_test_o:N`.)

`\__fp_add_significand_no_carry_o:wwwNN`

`\__fp_add_significand_no_carry_o:wwwNN`  $\langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle \text{rounding digit} \rangle \langle \text{sign} \rangle$

If there's no carry, grab all the digits again and round. The packing function `\__fp Basics_pack_high:NNNNNw` takes care of the case where rounding brings a carry.

```

25369 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
25370      #1; #2; #3#4 ; #5#6

```

```

25371   {
25372     \exp_after:wN \__fp_basics_pack_high:NNNNNw
25373     \int_value:w \__fp_int_eval:w 1 #1
25374       \exp_after:wN \__fp_basics_pack_low:NNNNNw
25375       \int_value:w \__fp_int_eval:w 1 #2 #3#4
25376         + \__fp_round:NNN #6 #4 #5
25377       \exp_after:wN ;
25378   }

```

(End definition for `\__fp_add_significand_no_carry_o:wwwNN`.)

`\__fp add significand carry o:wwwNN`  $\langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle \text{rounding digit} \rangle \langle \text{sign} \rangle$

The case where there is a carry is very similar. Rounding can even raise the first digit from 1 to 2, but we don't care.

```

25379 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
25380   #1; #2; #3#4; #5#6
25381   {
25382     +
25383     \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNNw
25384     \int_value:w \__fp_int_eval:w 1 1 #1
25385       \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
25386       \int_value:w \__fp_int_eval:w 1 #2#3 +
25387         \exp_after:wN \__fp_round:NNN
25388         \exp_after:wN #6
25389         \exp_after:wN #3
25390         \int_value:w \__fp_round_digit:Nw #4 #5 ;
25391       \exp_after:wN ;
25392   }

```

(End definition for `\__fp_add_significand_carry_o:wwwNN`.)

### 72.1.3 Absolute subtraction

`\__fp_sub_npos_o:NnwNnw`  $\langle \text{sign}_1 \rangle \langle \text{exp}_1 \rangle \langle \text{body}_1 \rangle ; \text{s\_fp } \__fp_chk:w 1$   
`\__fp_sub_eq_o:NnwNnw`  $\langle \text{initial sign}_2 \rangle \langle \text{exp}_2 \rangle \langle \text{body}_2 \rangle ;$

Rounding properly in some modes requires to know what the sign of the result will be. Thus, we start by comparing the exponents and significands. If the numbers coincide, return zero. If the second number is larger, swap the numbers and call `\__fp_sub_npos_i_o:NnwNnw` with the opposite of  $\langle \text{sign}_1 \rangle$ .

```

25393 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s_fp \__fp_chk:w 1 #4#5#6;
25394   {
25395     \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
25396       \exp_after:wN \__fp_sub_eq_o:NnwNnw
25397     \or:
25398       \exp_after:wN \__fp_sub_npos_i_o:NnwNnw
25399     \else:
25400       \exp_after:wN \__fp_sub_npos_ii_o:NnwNnw
25401     \fi:
25402     #1 {#2} #3; {#5} #6;
25403   }
25404 \cs_new:Npn \__fp_sub_eq_o:NnwNnw #1#2; #3; { \exp_after:wN \c_zero_fp }
25405 \cs_new:Npn \__fp_sub_npos_ii_o:NnwNnw #1 #2; #3;

```

```

25406      {
25407          \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
25408          \int_value:w \__fp_neg_sign:N #1
25409          #3; #2;
25410      }

```

(End definition for `\__fp_sub_npos_o:Nnwnw`, `\__fp_sub_eq_o:Nnwnw`, and `\__fp_sub_npos_i_o:Nnwnw`.)

`\__fp_sub_npos_i_o:Nnwnw`

After the computation is done, `\__fp_sanitize:Nw` checks for overflow/underflow. It expects the `<final sign>` and the `<exponent>` (delimited by ;). Start an integer expression for the exponent, which starts with the exponent of the largest number, and may be decreased if the two numbers are very close. If the two numbers have the same exponent, call the `near` auxiliary. Otherwise, decimate  $y$ , then call the `far` auxiliary to evaluate the difference between the two significands. Note that we decimate by 1 less than one could expect.

```

25411 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
25412 {
25413     \exp_after:wN \__fp_sanitize:Nw
25414     \exp_after:wN #1
25415     \int_value:w \__fp_int_eval:w
25416     #2
25417     \if_int_compare:w #2 = #4 \exp_stop_f:
25418         \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
25419     \else:
25420         \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
25421         { \int_value:w \__fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
25422         \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnN
25423     \fi:
25424     #5
25425     #3
25426     #1
25427 }

```

(End definition for `\__fp_sub_npos_i_o:Nnwnw`.)

`\__fp_sub_back_near_o:nnnnnnnnN`  
`\__fp_sub_back_near_pack:NNNNNNw`  
`\__fp_sub_back_near_after:wNNNNw`

In this case, the subtraction is exact, so we discard the `<final sign> #9`. The very large shifts of  $10^9$  and  $1.1 \cdot 10^9$  are unnecessary here, but allow the auxiliaries to be reused later. Each integer expression produces a 10 digit result. If the resulting 16 digits start with a 0, then we need to shift the group, padding with trailing zeros.

```

25428 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnN #1#2#3#4 #5#6#7#8 #9
25429 {
25430     \exp_after:wN \__fp_sub_back_near_after:wNNNNw
25431     \int_value:w \__fp_int_eval:w 10#5#6 - #1#2 - 11
25432     \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
25433     \int_value:w \__fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
25434 }
25435 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
25436     { + #1#2 ; {#3#4#5#6} {#7} ; }
25437 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
25438 {
25439     \if_meaning:w 0 #1
25440     \exp_after:wN \__fp_sub_back_shift:wnnnn

```

```

25441     \fi:
25442     ; {#1#2#3#4} {#5}
25443   }

```

(End definition for  $\_\_fp\_sub\_back\_near\_o:nnnnnnnnN$ ,  $\_\_fp\_sub\_back\_near\_pack:NNNNNNw$ , and  $\_\_fp\_sub\_back\_near\_after:wNNNNw$ .)

```

\_\_fp\_sub\_back\_shift:wnnnn
\_\_fp\_sub\_back\_shift\_ii:ww
  \_\_fp\_sub\_back\_shift\_iii:NNNNNNNNw
    \_\_fp\_sub\_back\_shift\_iv:nnnnw

```

$\_\_fp\_sub\_back\_shift:wnnnn$  ;  $\{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\}$  ;

This function is called with  $\langle Z_1 \rangle \leq 999$ . Act with `\number` to trim leading zeros from  $\langle Z_1 \rangle \langle Z_2 \rangle$  (we don't do all four blocks at once, since non-zero blocks would then overflow TEX's integers). If the first two blocks are zero, the auxiliary receives an empty #1 and trims #2#30 from leading zeros, yielding a total shift between 7 and 16 to the exponent. Otherwise we get the shift from #1 alone, yielding a result between 1 and 6. Once the exponent is taken care of, trim leading zeros from #1#2#3 (when #1 is empty, the space before #2#3 is ignored), get four blocks of 4 digits and finally clean up. Trailing zeros are added so that digits can be grabbed safely.

```

25444 \cs_new:Npn \_\_fp\_sub\_back\_shift:wnnnn ; #1#2
25445   {
25446     \exp_after:wN \_\_fp\_sub\_back\_shift\_ii:ww
25447     \int_value:w #1 #2 0 ;
25448   }
25449 \cs_new:Npn \_\_fp\_sub\_back\_shift\_ii:ww #1 0 ; #2#3 ;
25450   {
25451     \if_meaning:w @ #1 @
25452       - 7
25453       - \exp_after:wN \use_i:nnn
25454         \exp_after:wN \_\_fp\_sub\_back\_shift\_iii:NNNNNNNNw
25455           \int_value:w #2#3 0 ~ 123456789;
25456     \else:
25457       - \_\_fp\_sub\_back\_shift\_iii:NNNNNNNNw #1 123456789;
25458   \fi:
25459   \exp_after:wN \_\_fp\_pack\_twice\_four:wNNNNNNNN
25460   \exp_after:wN \_\_fp\_pack\_twice\_four:wNNNNNNNN
25461   \exp_after:wN \_\_fp\_sub\_back\_shift\_iv:nnnnw
25462   \exp_after:wN ;
25463   \int_value:w
25464     #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
25465   }
25466 \cs_new:Npn \_\_fp\_sub\_back\_shift\_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
25467 \cs_new:Npn \_\_fp\_sub\_back\_shift\_iv:nnnnw #1 ; #2 ; { ; #1 ; }


```

(End definition for  $\_\_fp\_sub\_back\_shift:wnnnn$  and others.)

$\_\_fp\_sub\_back\_far\_o:NnnnnnnnN$

$\_\_fp\_sub\_back\_far\_o:NnnnnnnnN \langle rounding \rangle \{\langle Y'_1 \rangle\} \{\langle Y'_2 \rangle\}$   
 $\langle extra-digits \rangle ; \{\langle X_1 \rangle\} \{\langle X_2 \rangle\} \{\langle X_3 \rangle\} \{\langle X_4 \rangle\} \langle final sign \rangle$

If the difference is greater than  $10^{\langle expox \rangle}$ , call the `very_far` auxiliary. If the result is less than  $10^{\langle expox \rangle}$ , call the `not_far` auxiliary. If it is too close a call to know yet, namely if  $1\langle Y'_1 \rangle \langle Y'_2 \rangle = \langle X_1 \rangle \langle X_2 \rangle \langle X_3 \rangle \langle X_4 \rangle 0$ , then call the `quite_far` auxiliary. We use the odd combination of space and semi-colon delimiters to allow the `not_far` auxiliary to grab each piece individually, the `very_far` auxiliary to use  $\_\_fp\_pack\_eight:wNNNNNNNN$ , and the `quite_far` to ignore the significands easily (using the ; delimiter).

```

25468 \cs_new:Npn \_\_fp\_sub\_back\_far\_o:NnnnnnnnN #1 #2#3 #4; #5#6#7#8
25469   {

```

```

25470   \if_case:w
25471     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
25472       \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
25473         0
25474       \else:
25475         \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
25476     \fi:
25477   \else:
25478     \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
25479   \fi:
25480   \exp_stop_f:
25481     \exp_after:wN \__fp_sub_back_quite_far_o:wwNN
25482   \or: \exp_after:wN \__fp_sub_back_very_far_o:wwwNN
25483   \else: \exp_after:wN \__fp_sub_back_not_far_o:wwwNN
25484   \fi:
25485   #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
25486 }

```

(End definition for `\__fp_sub_back_far_o:NnnnnnnnN.`)

`\__fp_sub_back_quite_far_o:wwNN` The easiest case is when  $x - y$  is extremely close to a power of 10, namely the first digit of  $x$  is 1, and all others vanish when subtracting  $y$ . Then the *rounding* #3 and the *final sign* #4 control whether we get 1 or 0.999999999999999. In the usual round-to-nearest mode, we get 1 whenever the *rounding* digit is less than or equal to 5 (remember that the *rounding* digit is only equal to 5 if there was no further non-zero digit).

```

25487 \cs_new:Npn \__fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
25488   {
25489     \exp_after:wN \__fp_sub_back_quite_far_ii>NN
25490     \exp_after:wN #3
25491     \exp_after:wN #4
25492   }
25493 \cs_new:Npn \__fp_sub_back_quite_far_ii>NN #1#2
25494   {
25495     \if_case:w \__fp_round_neg:NNN #2 0 #1
25496       \exp_after:wN \use_i:nn
25497     \else:
25498       \exp_after:wN \use_ii:nn
25499     \fi:
25500     { ; {1000} {0000} {0000} {0000} ; }
25501     { - 1 ; {9999} {9999} {9999} {9999} ; }
25502   }

```

(End definition for `\__fp_sub_back_quite_far_o:wwNN` and `\__fp_sub_back_quite_far_ii>NN`.)

`\__fp_sub_back_not_far_o:wwwNN` In the present case,  $x$  and  $y$  have different exponents, but  $y$  is large enough that  $x - y$  has a smaller exponent than  $x$ . Decrement the exponent (with  $-1$ ). Then proceed in a way similar to the `near` auxiliaries seen earlier, but multiplying  $x$  by 10 (#30 and #40 below), and with the added quirk that the *rounding* digit has to be taken into account. Namely, we may have to decrease the result by one unit if `\__fp_round_neg:NNN` returns 1. This function expects the *final sign* #6, the last digit of 1100000000+#40-#2, and the *rounding* digit. Instead of redoing the computation for the second argument, we note that `\__fp_round_neg:NNN` only cares about its parity, which is identical to that of the last digit of #2.

```

25503 \cs_new:Npn \__fp_sub_back_not_far_o:wwwNNN #1 ~ #2; #3 ~ #4; #5#6
25504 {
25505     - 1
25506     \exp_after:wN \__fp_sub_back_near_after:wNNNNw
25507     \int_value:w \__fp_int_eval:w 1#30 - #1 - 11
25508         \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
25509         \int_value:w \__fp_int_eval:w 11 0000 0000 + #40 - #2
25510             - \exp_after:wN \__fp_round_neg:NNN
25511                 \exp_after:wN #6
25512                     \use_none:nnnnnnn #2 #5
25513             \exp_after:wN ;
25514 }

```

(End definition for `\__fp_sub_back_not_far_o:wwwNNN`.)

The case where  $x - y$  and  $x$  have the same exponent is a bit more tricky, mostly because it cannot reuse the same auxiliaries. Shift the  $y$  significand by adding a leading 0. Then the logic is similar to the `not_far` functions above. Rounding is a bit more complicated: we have two *(rounding)* digits #3 and #6 (from the decimation, and from the new shift) to take into account, and getting the parity of the main result requires a computation. The first `\int_value:w` triggers the second one because the number is unfinished; we can thus not use 0 in place of 2 there.

```

25515 \cs_new:Npn \__fp_sub_back_very_far_o:wwwNNN #1#2#3#4#5#6#7
25516 {
25517     \__fp_pack_eight:wNNNNNNNNN
25518     \__fp_sub_back_very_far_i_o:nnNwwNN
25519     { 0 #1#2#3 #4#5#6#7 }
25520 ;
25521 }
25522 \cs_new:Npn \__fp_sub_back_very_far_i_o:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
25523 {
25524     \exp_after:wN \__fp_basics_pack_high:NNNNNw
25525     \int_value:w \__fp_int_eval:w 1#4 - #1 - 1
25526         \exp_after:wN \__fp_basics_pack_low:NNNNNw
25527         \int_value:w \__fp_int_eval:w 2#5 - #2
25528             - \exp_after:wN \__fp_round_neg:NNN
25529                 \exp_after:wN #7
25530                     \int_value:w
25531                         \if_int_odd:w \__fp_int_eval:w #5 - #2 \__fp_int_eval_end:
25532                             1 \else: 2 \fi:
25533                         \int_value:w \__fp_round_digit:Nw #3 #6 ;
25534                 \exp_after:wN ;
25535 }

```

(End definition for `\__fp_sub_back_very_far_o:wwwNNN` and `\__fp_sub_back_very_far_i_o:nnNwwNN`.)

## 72.2 Multiplication

### 72.2.1 Signs, and special numbers

We go through an auxiliary, which is common with `\__fp_/_o:ww`. The first argument is the operation, used for the invalid operation exception. The second is inserted in a formula to dispatch cases slightly differently between multiplication and division. The

third is the operation for normal floating points. The fourth is there for extra cases needed in `\_fp/_o:ww`.

```

25536 \cs_new:cpn { __fp_*_o:ww }
25537 {
25538     __fp_mul_cases_o:NnNnww
25539     *
25540     { - 2 + }
25541     __fp_mul_npos_o:Nww
25542     { }
25543 }
```

(End definition for `\_fp_*_o:ww`.)

`\_fp_mul_cases_o:nNnnww`

Split into 10 cases (12 for division). If both numbers are normal, go to case 0 (same sign) or case 1 (opposite signs): in both cases, call `\_fp_mul_npos_o:Nww` to do the work. If the first operand is `nan`, go to case 2, in which the second operand is discarded; if the second operand is `nan`, go to case 3, in which the first operand is discarded (note the weird interaction with the final test on signs). Then we separate the case where the first number is normal and the second is zero: this goes to cases 4 and 5 for multiplication, 10 and 11 for division. Otherwise, we do a computation which dispatches the products  $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$  to case 4 or 5 depending on the combined sign, the products  $0 \times \infty$  and  $\infty \times 0$  to case 6 or 7 (invalid operation), and the products  $1 \times \infty = \infty \times 1 = \infty \times \infty = \infty$  to cases 8 and 9. Note that the code for these two cases (which return  $\pm\infty$ ) is inserted as argument #4, because it differs in the case of divisions.

```

25544 \cs_new:Npn __fp_mul_cases_o:NnNnww
25545     #1#2#3#4 \s_fp __fp_chk:w #5#6#7; \s_fp __fp_chk:w #8#9
25546 {
25547     \if_case:w __fp_int_eval:w
25548         \if_int_compare:w #5 #8 = 11 ~
25549             1
25550         \else:
25551             \if_meaning:w 3 #8
25552                 3
25553             \else:
25554                 \if_meaning:w 3 #5
25555                     2
25556             \else:
25557                 \if_int_compare:w #5 #8 = 10 ~
25558                     9 #2 - 2
25559             \else:
25560                 (#5 #2 #8) / 2 * 2 + 7
25561             \fi:
25562         \fi:
25563     \fi:
25564     \if_meaning:w #6 #9 - 1 \fi:
25565         __fp_int_eval_end:
25566         __fp_case_use:nw { #3 0 }
25567     \or: __fp_case_use:nw { #3 2 }
25568     \or: __fp_case_return_i_o:ww
25569     \or: __fp_case_return_ii_o:ww
25570     \or: __fp_case_return_o:Nww \c_zero_fp
25571     \or: __fp_case_return_o:Nww \c_minus_zero_fp
```

```

25573     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
25574     \or: \__fp_case_use:nw { \__fp_invalid_operation_o:Nww #1 }
25575     \or: \__fp_case_return_o:Nww \c_inf_fp
25576     \or: \__fp_case_return_o:Nww \c_minus_inf_fp
25577     #4
25578     \fi:
25579     \s_fp \__fp_chk:w #5 #6 #7;
25580     \s_fp \__fp_chk:w #8 #9
25581   }

```

(End definition for `\__fp_mul_cases_o:nNnnww.`)

### 72.2.2 Absolute multiplication

In this subsection, we perform the multiplication of two positive normal numbers.

```
\__fp_mul_npos_o:Nww \__fp_mul_npos_o:Nww <final sign> \s_fp \__fp_chk:w 1 <sign1> {<exp1>}
    <body1> ; \s_fp \__fp_chk:w 1 <sign2> {<exp2>} <body2>
```

After the computation, `\__fp_sanitize:Nw` checks for overflow or underflow. As we did for addition, `\__fp_int_eval:w` computes the exponent, catching any shift coming from the computation in the significand. The `<final sign>` is needed to do the rounding properly in the significand computation. We setup the post-expansion here, triggered by `\__fp_mul_significand_o:nnnnNnnnn`.

This is also used in l3fp-convert.

```

25582 \cs_new:Npn \__fp_mul_npos_o:Nww
25583   #1 \s_fp \__fp_chk:w #2 #3 #4 #5 ; \s_fp \__fp_chk:w #6 #7 #8 #9 ;
25584   {
25585     \exp_after:wN \__fp_sanitize:Nw
25586     \exp_after:wN #1
25587     \int_value:w \__fp_int_eval:
25588     #4 + #8
25589     \__fp_mul_significand_o:nnnnNnnnn #5 #1 #9
25590   }

```

(End definition for `\__fp_mul_npos_o:Nww.`)

```
\__fp_mul_significand_o:nnnnNnnnn \__fp_mul_significand_drop:NNNNNw
\__fp_mul_significand_keep:NNNNNw
```

Note the three semicolons at the end of the definition. One is for the last `\__fp_mul_significand_drop:NNNNNw`; one is for `\__fp_round_digit:Nw` later on; and one, preceded by `\exp_after:wN`, which is correctly expanded (within an `\__fp_int_eval:w`), is used by `\__fp_basics_pack_low:NNNNNw`.

The product of two 16 digit integers has 31 or 32 digits, but it is impossible to know which one before computing. The place where we round depends on that number of digits, and may depend on all digits until the last in some rare cases. The approach is thus to compute the 5 first blocks of 4 digits (the first one is between 100 and 9999 inclusive), and a compact version of the remaining 3 blocks. Afterwards, the number of digits is known, and we can do the rounding within yet another set of `\__fp_int_eval:w`.

```

25591 \cs_new:Npn \__fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
25592   {
25593     \exp_after:wN \__fp_mul_significand_test_f:NNN
25594     \exp_after:wN #5
25595     \int_value:w \__fp_int_eval:w 99990000 + #1*#6 +

```

```

25596 \exp_after:wN \_fp_mul_significand_keep:NNNNNw
25597 \int_value:w \_fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
25598   \exp_after:wN \_fp_mul_significand_keep:NNNNNw
25599   \int_value:w \_fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
25600   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25601   \int_value:w \_fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
25602     #3*#7 + #4*#6 +
25603   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25604   \int_value:w \_fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
25605     #4*#7 +
25606   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25607   \int_value:w \_fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
25608   \exp_after:wN \_fp_mul_significand_drop:NNNNNw
25609   \int_value:w \_fp_int_eval:w 100000000 + #4*#9 ;
25610 ; \exp_after:wN ;
25611 }
25612 \cs_new:Npn \_fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
25613 { #1#2#3#4#5 ; + #6 }
25614 \cs_new:Npn \_fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
25615 { #1#2#3#4#5 ; #6 ; }

```

(End definition for `\_fp_mul_significand_o:nnnnNnnnn`, `\_fp_mul_significand_drop:NNNNNw`, and `\_fp_mul_significand_keep:NNNNNw`.)

```
\_fp_mul_significand_test_f:NNN
\__fp_mul_significand_test_f:NNN <sign> 1 <digits 1-8> ; <digits 9-12> ;
<digits 13-16> ; + <digits 17-20> + <digits 21-24> + <digits 25-28> + <digits
29-32> ; \exp_after:wN ;
```

If the  $\langle digit 1 \rangle$  is non-zero, then for rounding we only care about the digits 16 and 17, and whether further digits are zero or not (check for exact ties). On the other hand, if  $\langle digit 1 \rangle$  is zero, we care about digits 17 and 18, and whether further digits are zero.

```

25616 \cs_new:Npn \_fp_mul_significand_test_f:NNN #1 #2 #3
25617 {
25618   \if_meaning:w 0 #3
25619     \exp_after:wN \_fp_mul_significand_small_f:NNwwNw
25620   \else:
25621     \exp_after:wN \_fp_mul_significand_large_f:NwwNNNN
25622   \fi:
25623   #1 #3
25624 }
```

(End definition for `\_fp_mul_significand_test_f:NNN`.)

In this branch,  $\langle digit 1 \rangle$  is non-zero. The result is thus  $\langle digits 1-16 \rangle$ , plus some rounding which depends on the digits 16, 17, and whether all subsequent digits are zero or not. Here, `\_fp_round_digit:Nw` takes digits 17 and further (as an integer expression), and replaces it by a  $\langle rounding digit \rangle$ , suitable for `\_fp_round:NNN`.

```

25625 \cs_new:Npn \_fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
25626 {
25627   \exp_after:wN \_fp_basics_pack_high:NNNNNw
25628   \int_value:w \_fp_int_eval:w 1#2
25629     \exp_after:wN \_fp_basics_pack_low:NNNNNw
25630     \int_value:w \_fp_int_eval:w 1#3#4#5#6#7
25631     + \exp_after:wN \_fp_round:NNN
25632     \exp_after:wN #1
```

```

25633     \exp_after:wN #
25634     \int_value:w \_fp_round_digit:Nw
25635 }

```

(End definition for `\_fp_mul_significand_large_f:NwwNNNN.`)

`\_fp_mul_significand_small_f:NNwwwN`

In this branch,  $\langle digit\ 1\rangle$  is zero. Our result is thus  $\langle digits\ 2-17\rangle$ , plus some rounding which depends on the digits 17, 18, and whether all subsequent digits are zero or not. The 8 digits  $1\#3$  are followed, after expansion of the `small_pack` auxiliary, by the next digit, to form a 9 digit number.

```

25636 \cs_new:Npn \_fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
25637 {
25638     - 1
25639     \exp_after:wN \_fp_basics_pack_high>NNNNNw
25640     \int_value:w \_fp_int_eval:w 1#3#4
25641     \exp_after:wN \_fp_basics_pack_low>NNNNNw
25642     \int_value:w \_fp_int_eval:w 1#5#6#7
25643     + \exp_after:wN \_fp_round:NNN
25644     \exp_after:wN #1
25645     \exp_after:wN #7
25646     \int_value:w \_fp_round_digit:Nw
25647 }

```

(End definition for `\_fp_mul_significand_small_f:NNwwwN.`)

## 72.3 Division

### 72.3.1 Signs, and special numbers

Time is now ripe to tackle the hardest of the four elementary operations: division.

`\_fp/_o:ww`

Filtering special floating point is very similar to what we did for multiplications, with a few variations. Invalid operation exceptions display `/` rather than `*`. In the formula for dispatch, we replace `- 2 +` by `-`. The case of normal numbers is treated using `\_fp_div_npos_o:Nww` rather than `\_fp_mul_npos_o:Nww`. There are two additional cases: if the first operand is normal and the second is a zero, then the division by zero exception is raised: cases 10 and 11 of the `\if_case:w` construction in `\_fp_mul_cases_o:NnNnww` are provided as the fourth argument here.

```

25648 \cs_new:cpxn { _fp/_o:ww }
25649 {
25650     \_fp_mul_cases_o:NnNnww
25651     /
25652     { - }
25653     \_fp_div_npos_o:Nww
25654 {
25655     \or:
25656         \_fp_case_use:nw
25657             { \_fp_division_by_zero_o:NNww \c_inf_fp / }
25658     \or:
25659         \_fp_case_use:nw
25660             { \_fp_division_by_zero_o:NNww \c_minus_inf_fp / }
25661 }
25662 }

```

(End definition for `\_fp\_npo:ww`.)

```
\_fp_div_npos_o:Nww \_fp_div_npos_o:Nww <final sign> \s_fp \_fp_chk:w 1 <sign_A> {\exp A}
{\langle A_1 \rangle} {\langle A_2 \rangle} {\langle A_3 \rangle} {\langle A_4 \rangle} ; \s_fp \_fp_chk:w 1 <sign_Z> {\exp Z}
{\langle Z_1 \rangle} {\langle Z_2 \rangle} {\langle Z_3 \rangle} {\langle Z_4 \rangle} ;
```

We want to compute  $A/Z$ . As for multiplication, `\_fp_sanitize:Nw` checks for overflow or underflow; we provide it with the `<final sign>`, and an integer expression in which we compute the exponent. We set up the arguments of `\_fp_div_significand_i_o:wnnw`, namely an integer  $y$  obtained by adding 1 to the first 5 digits of  $Z$  (explanation given soon below), then the four  $\{A_i\}$ , then the four  $\{Z_i\}$ , a semi-colon, and the `<final sign>`, used for rounding at the end.

```
25663 \cs_new:Npn \_fp_div_npos_o:Nww
25664     #1 \s_fp \_fp_chk:w 1 #2 #3 #4 ; \s_fp \_fp_chk:w 1 #5 #6 #7#8#9;
25665     {
25666         \exp_after:wN \_fp_sanitize:Nw
25667         \exp_after:wN #1
25668         \int_value:w \_fp_int_eval:w
25669             #3 - #6
25670             \exp_after:wN \_fp_div_significand_i_o:wnnw
25671                 \int_value:w \_fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
25672                 #4
25673                 {#7}{#8}#9 ;
25674                 #1
25675 }
```

(End definition for `\_fp_div_npos_o:Nww`.)

### 72.3.2 Work plan

In this subsection, we explain how to avoid overflowing TeX's integers when performing the division of two positive normal numbers.

We are given two numbers,  $A = 0.A_1A_2A_3A_4$  and  $Z = 0.Z_1Z_2Z_3Z_4$ , in blocks of 4 digits, and we know that the first digits of  $A_1$  and of  $Z_1$  are non-zero. To compute  $A/Z$ , we proceed as follows.

- Find an integer  $Q_A \simeq 10^4 A/Z$ .
- Replace  $A$  by  $B = 10^4 A - Q_A Z$ .
- Find an integer  $Q_B \simeq 10^4 B/Z$ .
- Replace  $B$  by  $C = 10^4 B - Q_B Z$ .
- Find an integer  $Q_C \simeq 10^4 C/Z$ .
- Replace  $C$  by  $D = 10^4 C - Q_C Z$ .
- Find an integer  $Q_D \simeq 10^4 D/Z$ .
- Consider  $E = 10^4 D - Q_D Z$ , and ensure correct rounding.

The result is then  $Q = 10^{-4}Q_A + 10^{-8}Q_B + 10^{-12}Q_C + 10^{-16}Q_D + \text{rounding}$ . Since the  $Q_i$  are integers,  $B$ ,  $C$ ,  $D$ , and  $E$  are all exact multiples of  $10^{-16}$ , in other words, computing with 16 digits after the decimal separator yields exact results. The problem is the risk of overflow: in general  $B$ ,  $C$ ,  $D$ , and  $E$  may be greater than 1.

Unfortunately, things are not as easy as they seem. In particular, we want all intermediate steps to be positive, since negative results would require extra calculations at the end. This requires that  $Q_A \leq 10^4 A/Z$  etc. A reasonable attempt would be to define  $Q_A$  as

$$\text{\textbackslash int\_eval:n}\left\{\frac{A_1 A_2}{Z_1 + 1} - 1\right\} \leq 10^4 \frac{A}{Z}$$

Subtracting 1 at the end takes care of the fact that  $\varepsilon$ -TeX's `\_fp_int_eval:w` rounds divisions instead of truncating (really,  $1/2$  would be sufficient, but we work with integers). We add 1 to  $Z_1$  because  $Z_1 \leq 10^4 Z < Z_1 + 1$  and we need  $Q_A$  to be an underestimate. However, we are now underestimating  $Q_A$  too much: it can be wrong by up to 100, for instance when  $Z = 0.1$  and  $A \simeq 1$ . Then  $B$  could take values up to 10 (maybe more), and a few steps down the line, we would run into arithmetic overflow, since TeX can only handle integers less than roughly  $2 \cdot 10^9$ .

A better formula is to take

$$Q_A = \text{\textbackslash int\_eval:n}\left\{\frac{10 \cdot A_1 A_2}{[10^{-3} \cdot Z_1 Z_2] + 1} - 1\right\}.$$

This is always less than  $10^9 A / (10^5 Z)$ , as we wanted. In words, we take the 5 first digits of  $Z$  into account, and the 8 first digits of  $A$ , using 0 as a 9-th digit rather than the true digit for efficiency reasons. We shall prove that using this formula to define all the  $Q_i$  avoids any overflow. For convenience, let us denote

$$y = \lfloor 10^{-3} \cdot Z_1 Z_2 \rfloor + 1,$$

so that, taking into account the fact that  $\varepsilon$ -TeX rounds ties away from zero,

$$\begin{aligned} Q_A &= \left\lfloor \frac{A_1 A_2 0}{y} - \frac{1}{2} \right\rfloor \\ &> \frac{A_1 A_2 0}{y} - \frac{3}{2}. \end{aligned}$$

Note that  $10^4 < y \leq 10^5$ , and  $999 \leq Q_A \leq 99989$ . Also note that this formula does not cause an overflow as long as  $A < (2^{31} - 1)/10^9 \simeq 2.147 \dots$ , since the numerator involves an integer slightly smaller than  $10^9 A$ .

Let us bound  $B$ :

$$\begin{aligned} 10^5 B &= A_1 A_2 0 + 10 \cdot 0.A_3 A_4 - 10 \cdot Z_1.Z_2 Z_3 Z_4 \cdot Q_A \\ &< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1.Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1.Z_2 Z_3 Z_4 + 10 \\ &\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1.Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\ &\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y. \end{aligned}$$

At the last step, we hide 10 into the second term for later convenience. The same reasoning yields

$$\begin{aligned} 10^5 B &< 10^9 A/y + 1.6y, \\ 10^5 C &< 10^9 B/y + 1.6y, \\ 10^5 D &< 10^9 C/y + 1.6y, \\ 10^5 E &< 10^9 D/y + 1.6y. \end{aligned}$$

The goal is now to prove that none of  $B$ ,  $C$ ,  $D$ , and  $E$  can go beyond  $(2^{31} - 1)/10^9 = 2.147\ldots$ .

Combining the various inequalities together with  $A < 1$ , we get

$$\begin{aligned} 10^5 B &< 10^9/y + 1.6y, \\ 10^5 C &< 10^{13}/y^2 + 1.6(y + 10^4), \\ 10^5 D &< 10^{17}/y^3 + 1.6(y + 10^4 + 10^8/y), \\ 10^5 E &< 10^{21}/y^4 + 1.6(y + 10^4 + 10^8/y + 10^{12}/y^2). \end{aligned}$$

All of those bounds are convex functions of  $y$  (since every power of  $y$  involved is convex, and the coefficients are positive), and thus maximal at one of the end-points of the allowed range  $10^4 < y \leq 10^5$ . Thus,

$$\begin{aligned} 10^5 B &< \max(1.16 \cdot 10^5, 1.7 \cdot 10^5), \\ 10^5 C &< \max(1.32 \cdot 10^5, 1.77 \cdot 10^5), \\ 10^5 D &< \max(1.48 \cdot 10^5, 1.777 \cdot 10^5), \\ 10^5 E &< \max(1.64 \cdot 10^5, 1.7777 \cdot 10^5). \end{aligned}$$

All of those bounds are less than  $2.147 \cdot 10^5$ , and we are thus within TeX's bounds in all cases!

We later need to have a bound on the  $Q_i$ . Their definitions imply that  $Q_A < 10^9 A/y - 1/2 < 10^5 A$  and similarly for the other  $Q_i$ . Thus, all of them are less than 177770.

The last step is to ensure correct rounding. We have

$$A/Z = \sum_{i=1}^4 (10^{-4i} Q_i) + 10^{-16} E/Z$$

exactly. Furthermore, we know that the result is in  $[0.1, 10)$ , hence will be rounded to a multiple of  $10^{-16}$  or of  $10^{-15}$ , so we only need to know the integer part of  $E/Z$ , and a "rounding" digit encoding the rest. Equivalently, we need to find the integer part of  $2E/Z$ , and determine whether it was an exact integer or not (this serves to detect ties). Since

$$\frac{2E}{Z} = 2 \frac{10^5 E}{10^5 Z} \leq 2 \frac{10^5 E}{10^4} < 36,$$

this integer part is between 0 and 35 inclusive. We let  $\varepsilon$ -TeX round

$$P = \text{\int\_eval:n} \left\{ \frac{2 \cdot E_1 E_2}{Z_1 Z_2} \right\},$$

which differs from  $2E/Z$  by at most

$$\frac{1}{2} + 2 \left| \frac{E}{Z} - \frac{E}{10^{-8} Z_1 Z_2} \right| + 2 \left| \frac{10^8 E - E_1 E_2}{Z_1 Z_2} \right| < 1,$$

( $1/2$  comes from  $\varepsilon$ -TeX's rounding) because each absolute value is less than  $10^{-7}$ . Thus  $P$  is either the correct integer part, or is off by 1; furthermore, if  $2E/Z$  is an integer,  $P = 2E/Z$ . We will check the sign of  $2E - PZ$ . If it is negative, then  $E/Z \in ((P-1)/2, P/2)$ . If it is zero, then  $E/Z = P/2$ . If it is positive, then  $E/Z \in (P/2, (P+1)/2)$ . In each case, we know how to round to an integer, depending on the parity of  $P$ , and the rounding mode.

### 72.3.3 Implementing the significand division

```
\_\_fp\_div\_significand\_i\_o:wnnw
  \_\_fp\_div\_significand\_i\_o:wnnw <y> ; {<A1>} {<A2>} {<A3>} {<A4>}
    {<Z1>} {<Z2>} {<Z3>} {<Z4>} ; <sign>
  Compute  $10^6 + Q_A$  (a 7 digit number thanks to the shift), unbrace  $\langle A_1 \rangle$  and  $\langle A_2 \rangle$ , and prepare the  $\langle \text{continuation} \rangle$  arguments for 4 consecutive calls to  $\_\_fp\_div\_significand\_calc:wnnnnnnnn$ . Each of these calls needs  $\langle y \rangle$  (#1), and it turns out that we need post-expansion there, hence the  $\text{\int\_value:w}$ . Here, #4 is six brace groups, which give the six first n-type arguments of the calc function.
  25676 \cs_new:Npn \_\_fp\_div\_significand\_i\_o:wnnw #1 ; #2#3 #4 ;
  25677 {
  25678   \exp_after:wN \_\_fp\_div\_significand\_test_o:w
  25679   \int_value:w \_\_fp\_int\_eval:w
  25680   \exp_after:wN \_\_fp\_div\_significand\_calc:wnnnnnnnn
  25681   \int_value:w \_\_fp\_int\_eval:w 999999 + #2 #3 0 / #1 ;
  25682   #2 #3 ;
  25683   #4
  25684   { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \int_value:w #1 }
  25685   { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \int_value:w #1 }
  25686   { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \int_value:w #1 }
  25687   { \exp_after:wN \_\_fp\_div\_significand\_iii:wwnnnnn \int_value:w #1 }
  25688 }
```

(End definition for  $\_\_fp\_div\_significand\_i\_o:wnnw$ .)

```
\_\_fp\_div\_significand\_calc:wnnnnnnnn <math>10^6 + Q_A</math> ; <math>\langle A_1 \rangle \langle A_2 \rangle ; \{ \langle A_3 \rangle \} \{ \langle A_4 \rangle \} \{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \{ \langle \text{continuation} \rangle \}
```

expands to

$$\langle 10^6 + Q_A \rangle \langle \text{continuation} \rangle ; \langle B_1 \rangle \langle B_2 \rangle ; \{ \langle B_3 \rangle \} \{ \langle B_4 \rangle \} \{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \}$$

where  $B = 10^4 A - Q_A \cdot Z$ . This function is also used to compute  $C$ ,  $D$ ,  $E$  (with the input shifted accordingly), and is used in  $\text{l3fp-expo}$ .

We know that  $0 < Q_A < 1.8 \cdot 10^5$ , so the product of  $Q_A$  with each  $Z_i$  is within TeX's bounds. However, it is a little bit too large for our purposes: we would not be able to

use the usual trick of adding a large power of 10 to ensure that the number of digits is fixed.

The bound on  $Q_A$ , implies that  $10^6 + Q_A$  starts with the digit 1, followed by 0 or 1. We test, and call different auxiliaries for the two cases. An earlier implementation did the tests within the computation, but since we added a *(continuation)*, this is not possible because the macro has 9 parameters.

The result we want is then (the overall power of 10 is arbitrary):

$$10^{-4}(\#2 - \#1 \cdot \#5 - 10 \cdot \langle i \rangle \cdot \#5\#6) + 10^{-8}(\#3 - \#1 \cdot \#6 - 10 \cdot \langle i \rangle \cdot \#7) \\ + 10^{-12}(\#4 - \#1 \cdot \#7 - 10 \cdot \langle i \rangle \cdot \#8) + 10^{-16}(-\#1 \cdot \#8),$$

where  $\langle i \rangle$  stands for the  $10^5$  digit of  $Q_A$ , which is 0 or 1, and  $\#1$ ,  $\#2$ , etc. are the parameters of either auxiliary. The factors of 10 come from the fact that  $Q_A = 10 \cdot 10^4 \cdot \langle i \rangle + \#1$ . As usual, to combine all the terms, we need to choose some shifts which must ensure that the number of digits of the second, third, and fourth terms are each fixed. Here, the positive contributions are at most  $10^8$  and the negative contributions can go up to  $10^9$ . Indeed, for the auxiliary with  $\langle i \rangle = 1$ ,  $\#1$  is at most 80000, leading to contributions of at worse  $-8 \cdot 10^8$ , while the other negative term is very small  $< 10^6$  (except in the first expression, where we don't care about the number of digits); for the auxiliary with  $\langle i \rangle = 0$ ,  $\#1$  can go up to 99999, but there is no other negative term. Hence, a good choice is  $2 \cdot 10^9$ , which produces totals in the range  $[10^9, 2.1 \cdot 10^9]$ . We are flirting with TeX's limits once more.

```

25689 \cs_new:Npn \__fp_div_significand_calc:wnnnnnnnn 1#1
25690  {
25691    \if_meaning:w 1 #1
25692      \exp_after:wN \__fp_div_significand_calc_i:wnnnnnnnn
25693    \else:
25694      \exp_after:wN \__fp_div_significand_calc_ii:wnnnnnnnn
25695    \fi:
25696  }
25697 \cs_new:Npn \__fp_div_significand_calc_i:wnnnnnnnn
25698  #1; #2;#3#4 #5#6#7#8 #9
25699  {
25700    1 1 #1
25701    #9 \exp_after:wN ;
25702    \int_value:w \__fp_int_eval:w \c__fp_Bigg_leading_shift_int
25703      + #2 - #1 * #5 - #5#60
25704    \exp_after:wN \__fp_pack_Bigg:NNNNNNw
25705    \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
25706      + #3 - #1 * #6 - #70
25707    \exp_after:wN \__fp_pack_Bigg:NNNNNNw
25708    \int_value:w \__fp_int_eval:w \c__fp_Bigg_middle_shift_int
25709      + #4 - #1 * #7 - #80
25710    \exp_after:wN \__fp_pack_Bigg:NNNNNNw
25711    \int_value:w \__fp_int_eval:w \c__fp_Bigg_trailing_shift_int
25712      - #1 * #8 ;
25713    {#5}{#6}{#7}{#8}
25714  }
25715 \cs_new:Npn \__fp_div_significand_calc_ii:wnnnnnnnn
25716  #1; #2;#3#4 #5#6#7#8 #9
25717  {
25718    1 0 #1

```

```

25719      #9 \exp_after:wN ;
25720      \int_value:w \_fp_int_eval:w \c_fp_Bigg_leading_shift_int
25721          + #2 - #1 * #5
25722          \exp_after:wN \_fp_pack_Bigg:NNNNNNw
25723          \int_value:w \_fp_int_eval:w \c_fp_Bigg_middle_shift_int
25724              + #3 - #1 * #6
25725              \exp_after:wN \_fp_pack_Bigg:NNNNNNw
25726              \int_value:w \_fp_int_eval:w \c_fp_Bigg_middle_shift_int
25727                  + #4 - #1 * #7
25728                  \exp_after:wN \_fp_pack_Bigg:NNNNNNw
25729                  \int_value:w \_fp_int_eval:w \c_fp_Bigg_trailing_shift_int
25730                      - #1 * #8 ;
25731          {#5}{#6}{#7}{#8}
25732      }

```

(End definition for  $\_fp\_div\_significand\_calc:wwnnnnnn$ ,  $\_fp\_div\_significand\_calc\_i:wwnnnnnn$ , and  $\_fp\_div\_significand\_calc\_ii:wwnnnnnn$ .)

$\_fp\_div\_significand\_ii:wwn$   $\langle y \rangle ; \langle B_1 \rangle ; \{\langle B_2 \rangle\} \{\langle B_3 \rangle\} \{\langle B_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\}$   $\langle \text{continuations} \rangle \langle \text{sign} \rangle$

Compute  $Q_B$  by evaluating  $\langle B_1 \rangle \langle B_2 \rangle / 0/y - 1$ . The result is output to the left, in an  $\_fp\_int\_eval:w$  which we start now. Once that is evaluated (and the other  $Q_i$  also, since later expansions are triggered by this one), a packing auxiliary takes care of placing the digits of  $Q_B$  in an appropriate way for the final addition to obtain  $Q$ . This auxiliary is also used to compute  $Q_C$  and  $Q_D$  with the inputs  $C$  and  $D$  instead of  $B$ .

```

25733 \cs_new:Npn \_fp_div_significand_ii:wwn #1; #2;#3
25734 {
25735     \exp_after:wN \_fp_div_significand_pack:NNN
25736     \int_value:w \_fp_int_eval:w
25737         \exp_after:wN \_fp_div_significand_calc:wwnnnnnn
25738         \int_value:w \_fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
25739     }

```

(End definition for  $\_fp\_div\_significand\_ii:wwn$ .)

$\_fp\_div\_significand\_iii:wwnnnn$   $\langle y \rangle ; \langle E_1 \rangle ; \{\langle E_2 \rangle\} \{\langle E_3 \rangle\} \{\langle E_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\}$   $\langle \text{sign} \rangle$

We compute  $P \simeq 2E/Z$  by rounding  $2E_1 E_2 / Z_1 Z_2$ . Note the first 0, which multiplies  $Q_D$  by 10: we later add (roughly)  $5 \cdot P$ , which amounts to adding  $P/2 \simeq E/Z$  to  $Q_D$ , the appropriate correction from a hypothetical  $Q_E$ .

```

25740 \cs_new:Npn \_fp_div_significand_iii:wwnnnn #1; #2;#3#4#5 #6#7
25741 {
25742     0
25743     \exp_after:wN \_fp_div_significand_iv:wwnnnnnnn
25744     \int_value:w \_fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 ; % <- P
25745         #2 ; {#3} {#4} {#5}
25746         {#6} {#7}
25747     }

```

(End definition for  $\_fp\_div\_significand\_iii:wwnnnn$ .)

$\_fp\_div\_significand\_iv:wwnnnnnn$   $\langle P \rangle ; \langle E_1 \rangle ; \{\langle E_2 \rangle\} \{\langle E_3 \rangle\} \{\langle E_4 \rangle\} \{\langle Z_1 \rangle\} \{\langle Z_2 \rangle\} \{\langle Z_3 \rangle\} \{\langle Z_4 \rangle\}$   $\langle \text{sign} \rangle$

$\_fp\_div\_significand\_iv:wwnnnnnn$   
 $\_fp\_div\_significand\_v:NNw$   
 $\_fp\_div\_significand\_vi:Nw$

This adds to the current expression  $(10^7 + 10 \cdot Q_D)$  a contribution of  $5 \cdot P + \text{sign}(T)$  with  $T = 2E - PZ$ . This amounts to adding  $P/2$  to  $Q_D$ , with an extra  $\langle \text{rounding} \rangle$  digit. This  $\langle \text{rounding} \rangle$  digit is 0 or 5 if  $T$  does not contribute, *i.e.*, if  $0 = T = 2E - PZ$ , in other words if  $10^{16} A/Z$  is an integer or half-integer. Otherwise it is in the appropriate range, [1, 4] or [6, 9]. This is precise enough for rounding purposes (in any mode).

It seems an overkill to compute  $T$  exactly as I do here, but I see no faster way right now.

Once more, we need to be careful and show that the calculation  $\#1 \cdot \#6\#7$  below does not cause an overflow: naively,  $P$  can be up to 35, and  $\#6\#7$  up to  $10^8$ , but both cannot happen simultaneously. To show that things are fine, we split in two (non-disjoint) cases.

- For  $P < 10$ , the product obeys  $P \cdot \#6\#7 < 10^8 \cdot P < 10^9$ .
- For large  $P \geq 3$ , the rounding error on  $P$ , which is at most 1, is less than a factor of 2, hence  $P \leq 4E/Z$ . Also,  $\#6\#7 \leq 10^8 \cdot Z$ , hence  $P \cdot \#6\#7 \leq 4E \cdot 10^8 < 10^9$ .

Both inequalities could be made tighter if needed.

Note however that  $P \cdot \#8\#9$  may overflow, since the two factors are now independent, and the result may reach  $3.5 \cdot 10^9$ . Thus we compute the two lower levels separately. The rest is standard, except that we use  $+$  as a separator (ending integer expressions explicitly).  $T$  is negative if the first character is  $-$ , it is positive if the first character is neither 0 nor  $-$ . It is also positive if the first character is 0 and second argument of  $\_\_fp\_div\_significand\_vi:Nw$ , a sum of several terms, is also zero. Otherwise, there was an exact agreement:  $T = 0$ .

```

25748 \cs_new:Npn \_fp_div_significand_iv:wwnnnnnn #1; #2;#3#4#5 #6#7#8#9
25749 {
25750     + 5 * #1
25751     \exp_after:wN \_fp_div_significand_vi:Nw
25752     \int_value:w \_fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
25753         \exp_after:wN \_fp_div_significand_v:NN
25754         \int_value:w \_fp_int_eval:w 199980 + 2*#4 - #1*#8 +
25755             \exp_after:wN \_fp_div_significand_v:NN
25756             \int_value:w \_fp_int_eval:w 200000 + 2*#5 - #1*#9 ;
25757 }
25758 \cs_new:Npn \_fp_div_significand_v:NN #1#2 { #1#2 \_fp_int_eval_end: + }
25759 \cs_new:Npn \_fp_div_significand_vi:Nw #1#2;
25760 {
25761     \if_meaning:w 0 #1
25762         \if_int_compare:w \_fp_int_eval:w #2 > 0 + 1 \fi:
25763     \else:
25764         \if_meaning:w - #1 - \else: + \fi: 1
25765     \fi:
25766     ;
25767 }
```

(End definition for  $\_\_fp\_div\_significand\_iv:wwnnnnnn$ ,  $\_\_fp\_div\_significand\_v:NNw$ , and  $\_\_fp\_div\_significand\_vi:Nw$ .)

$\_\_fp\_div\_significand\_pack:NNN$  At this stage, we are in the following situation:  $\text{\TeX}$  is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```
\_\_fp\_div\_significand\_test\_o:w  $10^6 + Q_A$  \_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_B$  \_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_C$  \_\_fp\_div\_significand\_pack:NNN  $10^7 + 10 \cdot Q_D + 5 \cdot P + \varepsilon$  ;  $\langle sign \rangle$ 
```

Here,  $\varepsilon = \text{sign}(T)$  is 0 in case  $2E = PZ$ , 1 in case  $2E > PZ$ , which means that  $P$  was the correct value, but not with an exact quotient, and  $-1$  if  $2E < PZ$ , i.e.,  $P$  was an overestimate. The packing function we define now does nothing special: it removes the  $10^6$  and carries two digits (for the  $10^5$ 's and the  $10^4$ 's).

```
25768 \cs_new:Npn \_\_fp\_div\_significand\_pack:NNN 1 #1 #2 { + #1 #2 ; }
```

(End definition for \\_\\_fp\\_div\\_significand\\_pack:NNN.)

```
\_\_fp\_div\_significand\_test\_o:w \_\_fp\_div\_significand\_test\_o:w 1 0 ⟨5d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ; ⟨sign⟩
```

The reason we know that the first two digits are 1 and 0 is that the final result is known to be between 0.1 (inclusive) and 10, hence  $\tilde{Q}_A$  (the tilde denoting the contribution from the other  $Q_i$ ) is at most 99999, and  $10^6 + \tilde{Q}_A = 10 \dots$ .

It is now time to round. This depends on how many digits the final result will have.

```
25769 \cs_new:Npn \_\_fp\_div\_significand\_test\_o:w 10 #1
25770 {
25771     \if_meaning:w 0 #1
25772         \exp_after:wN \_\_fp\_div\_significand\_small_o:wwwNNNNwN
25773     \else:
25774         \exp_after:wN \_\_fp\_div\_significand\_large_o:wwwNNNNwN
25775     \fi:
25776     #1
25777 }
```

(End definition for \\_\\_fp\\_div\\_significand\\_test\\_o:w.)

```
\_\_fp\_div\_significand\_small_o:wwwNNNNwN 0 ⟨4d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ;
; ⟨final sign⟩
```

Standard use of the functions \\_\\_fp\\_basis\\_pack\\_low:NNNNNw and \\_\\_fp\\_basis\\_pack\\_high:NNNNNw. We finally get to use the ⟨final sign⟩ which has been sitting there for a while.

```
25778 \cs_new:Npn \_\_fp\_div\_significand\_small_o:wwwNNNNwN
25779     0 #1; #2; #3; #4#5#6#7#8; #9
25780 {
25781     \exp_after:wN \_\_fp\_basis\_pack\_high:NNNNNw
25782     \int_value:w \_\_fp\_int\_eval:w 1 #1#2
25783     \exp_after:wN \_\_fp\_basis\_pack\_low:NNNNNw
25784     \int_value:w \_\_fp\_int\_eval:w 1 #3#4#5#6#7
25785     + \_\_fp\_round:NNN #9 #7 #8
25786     \exp_after:wN ;
25787 }
```

(End definition for \\_\\_fp\\_div\\_significand\\_small\_o:wwwNNNNwN.)

```
\_\_fp\_div\_significand\_large_o:wwwNNNNwN ⟨5d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ;
⟨sign⟩
```

We know that the final result cannot reach 10, hence  $1\#1\#2$ , together with contributions from the level below, cannot reach  $2 \cdot 10^9$ . For rounding, we build the ⟨rounding digit⟩ from the last two of our 18 digits.

```
25788 \cs_new:Npn \_\_fp\_div\_significand\_large_o:wwwNNNNwN
```

```

25789      #1; #2; #3; #4#5#6#7#8; #9
25790  {
25791      + 1
25792      \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
25793      \int_value:w \__fp_int_eval:w 1 #1 #2
25794          \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
25795          \int_value:w \__fp_int_eval:w 1 #3 #4 #5 #6 +
25796              \exp_after:wN \__fp_round:NNN
25797              \exp_after:wN #9
25798              \exp_after:wN #6
25799                  \int_value:w \__fp_round_digit:Nw #7 #8 ;
25800          \exp_after:wN ;
25801      }

```

(End definition for `\__fp_div_significand_large_o:wwwNNNNwN.`)

## 72.4 Square root

`\__fp_sqrt_o:w` Zeros are unchanged:  $\sqrt{-0} = -0$  and  $\sqrt{+0} = +0$ . Negative numbers (other than  $-0$ ) have no real square root. Positive infinity, and `nan`, are unchanged. Finally, for normal positive numbers, there is some work to do.

```

25802 \cs_new:Npn \__fp_sqrt_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
25803  {
25804      \if_meaning:w 0 #2 \__fp_case_return_same_o:w \fi:
25805      \if_meaning:w 2 #3
25806          \__fp_case_use:nw { \__fp_invalid_operation_o:nw { sqrt } }
25807      \fi:
25808      \if_meaning:w 1 #2 \else: \__fp_case_return_same_o:w \fi:
25809          \__fp_sqrt_npos_o:w
25810          \s_fp \__fp_chk:w #2 #3 #4;
25811      }

```

(End definition for `\__fp_sqrt_o:w.`)

`\__fp_sqrt_npos_o:w` Prepare `\__fp_sanitize:Nw` to receive the final sign 0 (the result is always positive) and the exponent, equal to half of the exponent #1 of the argument. If the exponent #1 is even, find a first approximation of the square root of the significand  $10^8 a_1 + a_2 = 10^8 \#2\#3+\#4\#5$  through Newton's method, starting at  $x = 57234133 \simeq 10^{7.75}$ . Otherwise, first shift the significand of the argument by one digit, getting  $a'_1 \in [10^6, 10^7)$  instead of  $[10^7, 10^8)$ , then use Newton's method starting at  $17782794 \simeq 10^{7.25}$ .

```

25812 \cs_new:Npn \__fp_sqrt_npos_o:w \s_fp \__fp_chk:w 1 0 #1#2#3#4#5;
25813  {
25814      \exp_after:wN \__fp_sanitize:Nw
25815      \exp_after:wN 0
25816      \int_value:w \__fp_int_eval:w
25817          \if_int_odd:w #1 \exp_stop_f:
25818              \exp_after:wN \__fp_sqrt_npos_auxi_o:wwnnN
25819          \fi:
25820          #1 / 2
25821          \__fp_sqrt_Newton_o:wwn 56234133; 0; {\#2\#3} {\#4\#5} 0
25822      }
25823 \cs_new:Npn \__fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
25824  {

```

```

25825      ( #1 + 1 ) / 2
25826      \_\_fp\_pack\_eight:wNNNNNNNNN
25827      \_\_fp\_sqrt\_npos\_auxii\_o:wNNNNNNNNN
25828      ;
25829      0 #3 #4
25830  }
25831 \cs_new:Npn \_\_fp\_sqrt\_npos\_auxii\_o:wNNNNNNNNN #1; #2#3#4#5#6#7#8#9
25832   { \_\_fp\_sqrt\_Newton_o:wnn 17782794; 0; {#1} {#2#3#4#5#6#7#8#9} }

(End definition for \_\_fp\_sqrt\_npos\_o:w, \_\_fp\_sqrt\_npos\_auxii\_o:wwnnN, and \_\_fp\_sqrt\_npos\_auxii\_o:wNNNNNNNNN.)

```

\\_\\_fp\\_sqrt\\_Newton\\_o:wnn

Newton's method maps  $x \mapsto [(x + [10^8 a_1/x])/2]$  in each iteration, where  $[b/c]$  denotes  $\varepsilon$ -TeX's division. This division rounds the real number  $b/c$  to the closest integer, rounding ties away from zero, hence when  $c$  is even,  $b/c - 1/2 + 1/c \leq [b/c] \leq b/c + 1/2$  and when  $c$  is odd,  $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2 - 1/(2c)$ . For all  $c$ ,  $b/c - 1/2 + 1/(2c) \leq [b/c] \leq b/c + 1/2$ .

Let us prove that the method converges when implemented with  $\varepsilon$ -TeX integer division, for any  $10^6 \leq a_1 < 10^8$  and starting value  $10^6 \leq x < 10^8$ . Using the inequalities above and the arithmetic-geometric inequality  $(x + t)/2 \geq \sqrt{xt}$  for  $t = 10^8 a_1/x$ , we find

$$x' = \left[ \frac{x + [10^8 a_1/x]}{2} \right] \geq \frac{x + 10^8 a_1/x - 1/2 + 1/(2x)}{2} \geq \sqrt{10^8 a_1} - \frac{1}{4} + \frac{1}{4x}.$$

After any step of iteration, we thus have  $\delta = x - \sqrt{10^8 a_1} \geq -0.25 + 0.25 \cdot 10^{-8}$ . The new difference  $\delta' = x' - \sqrt{10^8 a_1}$  after one step is bounded above as

$$x' - \sqrt{10^8 a_1} \leq \frac{x + 10^8 a_1/x + 1/2}{2} + \frac{1}{2} - \sqrt{10^8 a_1} \leq \frac{\delta}{2} \frac{\delta}{\sqrt{10^8 a_1} + \delta} + \frac{3}{4}.$$

For  $\delta > 3/2$ , this last expression is  $\leq \delta/2 + 3/4 < \delta$ , hence  $\delta$  decreases at each step: since all  $x$  are integers,  $\delta$  must reach a value  $-1/4 < \delta \leq 3/2$ . In this range of values, we get  $\delta' \leq \frac{3}{4} \frac{3}{2\sqrt{10^8 a_1}} + \frac{3}{4} \leq 0.75 + 1.125 \cdot 10^{-7}$ . We deduce that the difference  $\delta = x - \sqrt{10^8 a_1}$  eventually reaches a value in the interval  $[-0.25 + 0.25 \cdot 10^{-8}, 0.75 + 11.25 \cdot 10^{-8}]$ , whose width is  $1 + 11 \cdot 10^{-8}$ . The corresponding interval for  $x$  may contain two integers, hence  $x$  might oscillate between those two values.

However, the fact that  $x \mapsto x - 1$  and  $x - 1 \mapsto x$  puts stronger constraints, which are not compatible: the first implies

$$x + [10^8 a_1/x] \leq 2x - 2$$

hence  $10^8 a_1/x \leq x - 3/2$ , while the second implies

$$x - 1 + [10^8 a_1/(x - 1)] \geq 2x - 1$$

hence  $10^8 a_1/(x - 1) \geq x - 1/2$ . Combining the two inequalities yields  $x^2 - 3x/2 \geq 10^8 a_1 \geq x - 3x/2 + 1/2$ , which cannot hold. Therefore, the iteration always converges to a single integer  $x$ . To stop the iteration when two consecutive results are equal, the function \\_\\_fp\\_sqrt\\_Newton\\_o:wnn receives the newly computed result as #1, the previous result as #2, and  $a_1$  as #3. Note that  $\varepsilon$ -TeX combines the computation of a multiplication and a following division, thus avoiding overflow in #3 \* 100000000 / #1. In any case, the result is within  $[10^7, 10^8]$ .

```

25833 \cs_new:Npn \__fp_sqrt_Newton_o:wwn #1; #2; #3
25834 {
25835     \if_int_compare:w #1 = #2 \exp_stop_f:
25836         \exp_after:wN \__fp_sqrt_auxi_o:NNNNwnnN
25837         \int_value:w \__fp_int_eval:w 9999 9999 +
25838             \exp_after:wN \__fp_use_none_until_s:w
25839     \fi:
25840     \exp_after:wN \__fp_sqrt_Newton_o:wwn
25841     \int_value:w \__fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
25842     #1; {#3}
25843 }

```

(End definition for `\__fp_sqrt_Newton_o:wwn`.)

`\__fp_sqrt_auxi_o:NNNNwnnN`

This function is followed by  $10^8 + x - 1$ , which has 9 digits starting with 1, then ;  $\{a_1\} \{a_2\} \langle a \rangle$ . Here,  $x \simeq \sqrt{10^8 a_1}$  and we want to estimate the square root of  $a = 10^{-8} a_1 + 10^{-16} a_2 + 10^{-17} a'$ . We set up an initial underestimate

$$y = (x - 1)10^{-8} + 0.2499998875 \cdot 10^{-8} \lesssim \sqrt{a}.$$

From the inequalities shown earlier, we know that  $y \leq \sqrt{10^{-8} a_1} \leq \sqrt{a}$  and that  $\sqrt{10^{-8} a_1} \leq y + 10^{-8} + 11 \cdot 10^{-16}$  hence (using  $0.1 \leq y \leq \sqrt{a} \leq 1$ )

$$a - y^2 \leq 10^{-8} a_1 + 10^{-8} - y^2 \leq (y + 10^{-8} + 11 \cdot 10^{-16})^2 - y^2 + 10^{-8} < 3.2 \cdot 10^{-8},$$

and  $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y) \leq 16 \cdot 10^{-8}$ . Next, `\__fp_sqrt_auxii_o:NnnnnnnnnN` is called several times to get closer and closer underestimates of  $\sqrt{a}$ . By construction, the underestimates  $y$  are always increasing,  $a - y^2 < 3.2 \cdot 10^{-8}$  for all. Also,  $y < 1$ .

```

25844 \cs_new:Npn \__fp_sqrt_auxii_o:NNNNwnnN 1 #1#2#3#4#5;
25845 {
25846     \__fp_sqrt_auxii_o:NnnnnnnnnN
25847     \__fp_sqrt_auxiii_o:wnnnnnnnnN
25848     {#1#2#3#4} {#5} {2499} {9988} {7500}
25849 }

```

(End definition for `\__fp_sqrt_auxi_o:NNNNwnnN`.)

`\__fp_sqrt_auxii_o:NnnnnnnnnN`

This receives a continuation function #1, then five blocks of 4 digits for  $y$ , then two 8-digit blocks and a single digit for  $a$ . A common estimate of  $\sqrt{a} - y = (a - y^2)/(\sqrt{a} + y)$  is  $(a - y^2)/(2y)$ , which leads to alternating overestimates and underestimates. We tweak this, to only work with underestimates (no need then to worry about signs in the computation). Each step finds the largest integer  $j \leq 6$  such that  $10^{4j}(a - y^2) < 2 \cdot 10^8$ , then computes the integer (with  $\varepsilon$ -TEX's rounding division)

$$10^{4j}z = \left[ \left( \lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8 y + 1 \rfloor \right].$$

The choice of  $j$  ensures that  $10^{4j}z < 2 \cdot 10^8 \cdot 0.5 \cdot 10^8 / 10^7 = 10^9$ , thus  $10^9 + 10^{4j}z$  has exactly 10 digits, does not overflow T<sub>E</sub>X's integer range, and starts with 1. Incidentally, since all  $a - y^2 \leq 3.2 \cdot 10^{-8}$ , we know that  $j \geq 3$ .

Let us show that  $z$  is an underestimate of  $\sqrt{a} - y$ . On the one hand,  $\sqrt{a} - y \leq 16 \cdot 10^{-8}$  because this holds for the initial  $y$  and values of  $y$  can only increase. On the other hand, the choice of  $j$  implies that  $\sqrt{a} - y \leq 5(\sqrt{a} + y)(\sqrt{a} - y) = 5(a - y^2) < 10^{9-4j}$ . For  $j = 3$ ,

the first bound is better, while for larger  $j$ , the second bound is better. For all  $j \in [3, 6]$ , we find  $\sqrt{a} - y < 16 \cdot 10^{-2j}$ . From this, we deduce that

$$10^{4j}(\sqrt{a} - y) = \frac{10^{4j}(a - y^2 - (\sqrt{a} - y)^2)}{2y} \geq \frac{\lfloor 10^{4j}(a - y^2) \rfloor - 257}{2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor} + \frac{1}{2}$$

where we have replaced the bound  $10^{4j}(16 \cdot 10^{-2j}) = 256$  by 257 and extracted the corresponding term  $1/(2 \cdot 10^{-8} \lfloor 10^8 y + 1 \rfloor) \geq 1/2$ . Given that  $\varepsilon$ -TEX's integer division obeys  $[b/c] \leq b/c + 1/2$ , we deduce that  $10^{4j}z \leq 10^{4j}(\sqrt{a} - y)$ , hence  $y + z \leq \sqrt{a}$  is an underestimate of  $\sqrt{a}$ , as claimed. One implementation detail: because the computation involves  $-4*#4 - 2*#3*#5 - 2*#2*#6$  which may be as low as  $-5 \cdot 10^8$ , we need to use the `pack_big` functions, and the `big` shifts.

```

25850 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnnN #1 #2#3#4#5#6 #7#8#9
25851 {
25852   \exp_after:wN #1
25853   \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
25854     + #7 - #2 * #2
25855   \exp_after:wN \__fp_pack_big:NNNNNNw
25856   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25857     - 2 * #2 * #3
25858   \exp_after:wN \__fp_pack_big:NNNNNNw
25859   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25860     + #8 - #3 * #3 - 2 * #2 * #4
25861   \exp_after:wN \__fp_pack_big:NNNNNNw
25862   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25863     - 2 * #3 * #4 - 2 * #2 * #5
25864   \exp_after:wN \__fp_pack_big:NNNNNNw
25865   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25866     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
25867   \exp_after:wN \__fp_pack_big:NNNNNNw
25868   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25869     - 2 * #4 * #5 - 2 * #3 * #6
25870   \exp_after:wN \__fp_pack_big:NNNNNNw
25871   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
25872     - #5 * #5 - 2 * #4 * #6
25873   \exp_after:wN \__fp_pack_big:NNNNNNw
25874   \int_value:w \__fp_int_eval:w
25875     \c__fp_big_middle_shift_int
25876     - 2 * #5 * #6
25877   \exp_after:wN \__fp_pack_big:NNNNNNw
25878   \int_value:w \__fp_int_eval:w
25879     \c__fp_big_trailing_shift_int
25880     - #6 * #6 ;
25881 %
25882 - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
25883 {#2}{#3}{#4}{#5}{#6} {#7}{#8}{#9}
25884 }
```

(End definition for `\__fp_sqrt_auxii_o:NnnnnnnnN`.)

We receive here the difference  $a - y^2 = d = \sum_i d_i \cdot 10^{-4i}$ , as  $\langle d_2 \rangle ; \{ \langle d_3 \rangle \} \dots \{ \langle d_{10} \rangle \}$ , where each block has 4 digits, except  $\langle d_2 \rangle$ . This function finds the largest  $j \leq 6$  such that  $10^{4j}(a - y^2) < 2 \cdot 10^8$ , then leaves an open parenthesis and the integer  $\lfloor 10^{4j}(a - y^2) \rfloor$

```

\__fp_sqrt_auxii_o:wnnnnnnn
\__fp_sqrt_auxiv_o:NNNNNw
\__fp_sqrt_auxv_o:NNNNNw
\__fp_sqrt_auxvi_o:NNNNNw
\__fp_sqrt_auxvii_o:NNNNNw
```

in an integer expression. The closing parenthesis is provided by the caller `\_fp_sqrt_auxii_o:NnnnnnnnnN`, which completes the expression

$$10^{4j}z = \left[ ( \lfloor 10^{4j}(a - y^2) \rfloor - 257 ) \cdot (0.5 \cdot 10^8) \Big/ \lfloor 10^8y + 1 \rfloor \right]$$

for an estimate of  $10^{4j}(\sqrt{a} - y)$ . If  $d_2 \geq 2$ ,  $j = 3$  and the `auxiv` auxiliary receives  $10^{12}z$ . If  $d_2 \leq 1$  but  $10^4d_2 + d_3 \geq 2$ ,  $j = 4$  and the `auxv` auxiliary is called, and receives  $10^{16}z$ , and so on. In all those cases, the `auxviii` auxiliary is set up to add  $z$  to  $y$ , then go back to the `auxii` step with continuation `auxiii` (the function we are currently describing). The maximum value of  $j$  is 6, regardless of whether  $10^{12}d_2 + 10^8d_3 + 10^4d_4 + d_5 \geq 1$ . In this last case, we detect when  $10^{24}z < 10^7$ , which essentially means  $\sqrt{a} - y \lesssim 10^{-17}$ : once this threshold is reached, there is enough information to find the correctly rounded  $\sqrt{a}$  with only one more call to `\_fp_sqrt_auxii_o:NnnnnnnnnN`. Note that the iteration cannot be stuck before reaching  $j = 6$ , because for  $j < 6$ , one has  $2 \cdot 10^8 \leq 10^{4(j+1)}(a - y^2)$ , hence

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

25885 \cs_new:Npn \_fp_sqrt_auxii_o:wnnnnnnnnn
25886   #1; #2#3#4#5#6#7#8#9
25887   {
25888     \if_int_compare:w #1 > \c_one_int
25889       \exp_after:wN \_fp_sqrt_auxiv_o:NNNNNw
25890       \int_value:w \_fp_int_eval:w (#1#2 %)
25891     \else:
25892       \if_int_compare:w #1#2 > \c_one_int
25893         \exp_after:wN \_fp_sqrt_auxv_o:NNNNNw
25894         \int_value:w \_fp_int_eval:w (#1#2#3 %)
25895       \else:
25896         \if_int_compare:w #1#2#3 > \c_one_int
25897           \exp_after:wN \_fp_sqrt_auxvi_o:NNNNNw
25898           \int_value:w \_fp_int_eval:w (#1#2#3#4 %)
25899       \else:
25900         \exp_after:wN \_fp_sqrt_auxvii_o:NNNNNw
25901         \int_value:w \_fp_int_eval:w (#1#2#3#4#5 %)
25902       \fi:
25903     \fi:
25904   \fi:
25905 }
25906 \cs_new:Npn \_fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
25907   { \_fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
25908 \cs_new:Npn \_fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
25909   { \_fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
25910 \cs_new:Npn \_fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
25911   { \_fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
25912 \cs_new:Npn \_fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
25913   {
25914     \if_int_compare:w #1#2 = \c_zero_int
25915       \exp_after:wN \_fp_sqrt_auxxx_o:Nnnnnnnn
25916     \fi:
25917     \_fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
25918 }
```

(End definition for `\_fp_sqrt_auxii_o:wnnnnnnnn` and others.)

```
\_fp_sqrt_auxviii_o:nnnnnnnn
\fp_sqrt_auxix_o:wnwnw
```

Simply add the two 8-digit blocks of  $z$ , aligned to the last four of the five 4-digit blocks of  $y$ , then call the `auxii` auxiliary to evaluate  $y^2 = (y + z)^2$ .

```
25919 \cs_new:Npn \fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
25920 {
25921   \exp_after:wN \fp_sqrt_auxix_o:wnwnw
25922   \int_value:w \fp_int_eval:w #3
25923   \exp_after:wN \fp_basics_pack_low>NNNNw
25924   \int_value:w \fp_int_eval:w #1 + 1#4#5
25925   \exp_after:wN \fp_basics_pack_low>NNNNw
25926   \int_value:w \fp_int_eval:w #2 + 1#6#7 ;
25927 }
25928 \cs_new:Npn \fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
25929 {
25930   \fp_sqrt_auxii_o:NnnnnnnnN
25931   \fp_sqrt_auxii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
25932 }
```

(End definition for `\fp_sqrt_auxviii_o:nnnnnnnn` and `\fp_sqrt_auxix_o:wnwnw`.)

At this stage,  $j = 6$  and  $10^{24}z < 10^7$ , hence

$$10^7 + 1/2 > 10^{24}z + 1/2 \geq (10^{24}(a - y^2) - 258) \cdot (0.5 \cdot 10^8) / (10^8y + 1),$$

then  $10^{24}(a - y^2) - 258 < 2(10^7 + 1/2)(y + 10^{-8})$ , and

$$10^{24}(a - y^2) < (10^7 + 1290.5)(1 + 10^{-8}/y)(2y) < (10^7 + 1290.5)(1 + 10^{-7})(y + \sqrt{a}),$$

which finally implies  $0 \leq \sqrt{a} - y < 0.2 \cdot 10^{-16}$ . In particular,  $y$  is an underestimate of  $\sqrt{a}$  and  $y + 0.5 \cdot 10^{-16}$  is a (strict) overestimate. There is at exactly one multiple  $m$  of  $0.5 \cdot 10^{-16}$  in the interval  $[y, y + 0.5 \cdot 10^{-16}]$ . If  $m^2 > a$ , then the square root is inexact and is obtained by rounding  $m - \epsilon$  to a multiple of  $10^{-16}$  (the precise shift  $0 < \epsilon < 0.5 \cdot 10^{-16}$  is irrelevant for rounding). If  $m^2 = a$  then the square root is exactly  $m$ , and there is no rounding. If  $m^2 < a$  then we round  $m + \epsilon$ . For now, discard a few irrelevant arguments `#1`, `#2`, `#3`, and find the multiple of  $0.5 \cdot 10^{-16}$  within  $[y, y + 0.5 \cdot 10^{-16}]$ ; rather, only the last 4 digits `#8` of  $y$  are considered, and we do not perform any carry yet. The `auxxi` auxiliary sets up `auxii` with a continuation function `auxxii` instead of `auxiii` as before. To prevent `auxii` from giving a negative results  $a - m^2$ , we compute  $a + 10^{-16} - m^2$  instead, always positive since  $m < \sqrt{a} + 0.5 \cdot 10^{-16}$  and  $a \leq 1 - 10^{-16}$ .

```
25933 \cs_new:Npn \fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
25934 {
25935   \exp_after:wN \fp_sqrt_auxxi_o:wnnnN
25936   \int_value:w \fp_int_eval:w
25937   (#8 + 2499) / 5000 * 5000 ;
25938   {#4} {#5} {#6} {#7} ;
25939 }
25940 \cs_new:Npn \fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
25941 {
25942   \fp_sqrt_auxii_o:NnnnnnnnN
25943   \fp_sqrt_auxxii_o:nnnnnnnnw
25944   #2 {#1}
25945   {#3} {#4 + 1} #5
25946 }
```

(End definition for `\_fp_sqrt_auxxx_o:Nnnnnnnn` and `\_fp_sqrt_auxxi_o:wwnnN`.)

`\_fp_sqrt_auxxii_o:nmmmmmmw`  
`\_fp_sqrt_auxxiii_o:w`

The difference  $0 \leq a + 10^{-16} - m^2 \leq 10^{-16} + (\sqrt{a} - m)(\sqrt{a} + m) \leq 2 \cdot 10^{-16}$  was just computed: its first 8 digits vanish, as do the next four, #1, and most of the following four, #2. The guess  $m$  is an overestimate if  $a + 10^{-16} - m^2 < 10^{-16}$ , that is, #1#2 vanishes. Otherwise it is an underestimate, unless  $a + 10^{-16} - m^2 = 10^{-16}$  exactly. For an underestimate, call the `auxxiv` function with argument 9998. For an exact result call it with 9999, and for an overestimate call it with 10000.

```

25947 \cs_new:Npn \_fp_sqrt_auxxii_o:nmmmmmmw #1#2#3#4#5#6#7#8 #9;
25948 {
25949     \if_int_compare:w #1#2 > \c_zero_int
25950         \if_int_compare:w #1#2 = \c_one_int
25951             \if_int_compare:w #3#4 = \c_zero_int
25952                 \if_int_compare:w #5#6 = \c_zero_int
25953                     \if_int_compare:w #7#8 = \c_zero_int
25954                         \_fp_sqrt_auxxiii_o:w
25955                     \fi:
25956                 \fi:
25957             \fi:
25958         \fi:
25959         \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
25960         \int_value:w 9998
25961     \else:
25962         \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
25963         \int_value:w 10000
25964     \fi:
25965 ;
25966 }
25967 \cs_new:Npn \_fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: #1 \fi: ;
25968 {
25969     \fi: \fi: \fi: \fi:
25970     \_fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
25971 }
```

(End definition for `\_fp_sqrt_auxxii_o:nmmmmmmw` and `\_fp_sqrt_auxxiii_o:w`.)

`\_fp_sqrt_auxxiv_o:wnnnnnnnN`

This receives 9998, 9999 or 10000 as #1 when  $m$  is an underestimate, exact, or an overestimate, respectively. Then comes  $m$  as five blocks of 4 digits, but where the last block #6 may be 0, 5000, or 10000. In the latter case, we need to add a carry, unless  $m$  is an overestimate (#1 is then 10000). Then comes  $a$  as three arguments. Rounding is done by `\_fp_round:NNN`, whose first argument is the final sign 0 (square roots are positive). We fake its second argument. It should be the last digit kept, but this is only used when ties are “rounded to even”, and only when the result is exactly half-way between two representable numbers rational square roots of numbers with 16 significant digits have: this situation never arises for the square root, as any exact square root of a 16 digit number has at most 8 significant digits. Finally, the last argument is the next digit, possibly shifted by 1 when there are further nonzero digits. This is achieved by `\_fp_round_digit:Nw`, which receives (after removal of the 10000’s digit) one of 0000, 0001, 4999, 5000, 5001, or 9999, which it converts to 0, 1, 4, 5, 6, and 9, respectively.

```

25972 \cs_new:Npn \_fp_sqrt_auxxiv_o:wnnnnnnnN #1; #2#3#4#5#6 #7#8#9
25973 {
25974     \exp_after:wN \_fp_basics_pack_high:NNNNNw
25975     \int_value:w \_fp_int_eval:w 1 0000 0000 + #2#3
```

```

25976   \exp_after:wN \_fp_basics_pack_low>NNNNNw
25977   \int_value:w \_fp_int_eval:w 1 0000 0000
25978     + #4#5
25979     \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
25980     + \exp_after:wN \_fp_round:NNN
25981       \exp_after:wN 0
25982       \exp_after:wN 0
25983       \int_value:w
25984         \exp_after:wN \use_i:nn
25985         \exp_after:wN \_fp_round_digit:Nw
25986         \int_value:w \_fp_int_eval:w #6 + 19999 - #1 ;
25987       \exp_after:wN ;
25988   }

```

(End definition for `\_fp_sqrt_auxxiv_o:wnnnnnnnN.`)

## 72.5 About the sign and exponent

`\_fp_logb_o:w` The exponent of a normal number is its *exponent* minus one.

```

\_fp_logb_aux_o:w 25989 \cs_new:Npn \_fp_logb_o:w ? \s_fp \_fp_chk:w #1#2; @
25990   {
25991     \if_case:w #1 \exp_stop_f:
25992       \_fp_case_use:nw
25993         { \_fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
25994     \or: \exp_after:wN \_fp_logb_aux_o:w
25995     \or: \_fp_case_return_o:Nw \c_inf_fp
25996     \else: \_fp_case_return_same_o:w
25997     \fi:
25998     \s_fp \_fp_chk:w #1 #2;
25999   }
26000 \cs_new:Npn \_fp_logb_aux_o:w \s_fp \_fp_chk:w #1 #2 #3 #4 ;
26001   {
26002     \exp_after:wN \_fp_parse:n \exp_after:wN
26003       { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
26004   }

```

(End definition for `\_fp_logb_o:w` and `\_fp_logb_aux_o:w.`)

`\_fp_sign_o:w` Find the sign of the floating point: nan, +0, -0, +1 or -1.

```

\_fp_sign_aux_o:w 26005 \cs_new:Npn \_fp_sign_o:w ? \s_fp \_fp_chk:w #1#2; @
26006   {
26007     \if_case:w #1 \exp_stop_f:
26008       \_fp_case_return_same_o:w
26009     \or: \exp_after:wN \_fp_sign_aux_o:w
26010     \or: \exp_after:wN \_fp_sign_aux_o:w
26011     \else: \_fp_case_return_same_o:w
26012     \fi:
26013     \s_fp \_fp_chk:w #1 #2;
26014   }
26015 \cs_new:Npn \_fp_sign_aux_o:w \s_fp \_fp_chk:w #1 #2 #3 ;
26016   { \exp_after:wN \_fp_set_sign_o:w \exp_after:wN #2 \c_one_fp @ }

```

(End definition for `\_fp_sign_o:w` and `\_fp_sign_aux_o:w.`)

```
\__fp_set_sign_o:ww
```

This function is used for the unary minus and for `abs`. It leaves the sign of `nan` invariant, turns negative numbers (sign 2) to positive numbers (sign 0) and positive numbers (sign 0) to positive or negative numbers depending on #1. It also expands after itself in the input stream, just like `\__fp_+_o:ww`.

```
26017 \cs_new:Npn \__fp_set_sign_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
26018 {
26019     \exp_after:wN \__fp_exp_after_o:w
26020     \exp_after:wN \s__fp
26021     \exp_after:wN \__fp_chk:w
26022     \exp_after:wN #2
26023     \int_value:w
26024     \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
26025     #4;
26026 }
```

(End definition for `\__fp_set_sign_o:ww`.)

## 72.6 Operations on tuples

```
\__fp_tuple_set_sign_o:ww
```

Two cases: `abs(<tuple>)` for which #1 is 0 (invalid for tuples) and `-<tuple>` for which #1 is 2. In that case, map over all items in the tuple an auxiliary that dispatches to the type-appropriate sign-flipping function.

```
26027 \cs_new:Npn \__fp_tuple_set_sign_o:w #1
26028 {
26029     \if_meaning:w 2 #1
26030         \exp_after:wN \__fp_tuple_set_sign_aux_o:Nnw
26031         \fi:
26032         \__fp_invalid_operation_o:nw { abs }
26033     }
26034 \cs_new:Npn \__fp_tuple_set_sign_aux_o:Nnw #1#2#3 @
26035     { \__fp_tuple_map_o:nw \__fp_tuple_set_sign_aux_o:w #3 }
26036 \cs_new:Npn \__fp_tuple_set_sign_aux_o:w #1#2 ;
26037     {
26038         \__fp_change_func_type>NNN #1 \__fp_set_sign_o:w
26039         \__fp_parse_apply_unary_error>NNw
26040         2 #1 #2 ; @
26041 }
```

(End definition for `\__fp_tuple_set_sign_o:ww`, `\__fp_tuple_set_sign_aux_o:Nnw`, and `\__fp_tuple_set_sign_aux_o:ww`.)

```
\__fp_*_tuple_o:ww  
\__fp_tuple_*_o:ww  
\__fp_tuple/_o:ww
```

For `<number>*<tuple>` and `<tuple>*<number>` and `<tuple>/<number>`, loop through the `<tuple>` some code that multiplies or divides by the appropriate `<number>`. Importantly we need to dispatch according to the type, and we make sure to apply the operator in the correct order.

```
26042 \cs_new:cpn { __fp_*_tuple_o:ww } #1 ;
26043     { \__fp_tuple_map_o:nw { \__fp_binary_type_o:Nww * #1 ; } }
26044 \cs_new:cpn { __fp_tuple_*_o:ww } #1 ; #2 ;
26045     { \__fp_tuple_map_o:nw { \__fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
26046 \cs_new:cpn { __fp_tuple/_o:ww } #1 ; #2 ;
26047     { \__fp_tuple_map_o:nw { \__fp_binary_rev_type_o:Nww / #2 ; } #1 ; }
```

(End definition for `\__fp_*_tuple_o:ww`, `\__fp_tuple_*_o:ww`, and `\__fp_tuple/_o:ww`.)

```
\__fp_tuple_+_tuple_o:ww  
\__fp_tuple_-_tuple_o:ww
```

Check the two tuples have the same number of items and map through these a helper that dispatches appropriately depending on the types. This means  $(1,2)+((1,1),2)$  gives  $(\text{nan}, 4)$ .

```
26048 \cs_set_protected:Npn \__fp_tmp:w #1
26049  {
26050    \cs_new:cpn { __fp_tuple_#1_tuple_o:ww }
26051    \s__fp_tuple \__fp_tuple_chk:w ##1 ;
26052    \s__fp_tuple \__fp_tuple_chk:w ##2 ;
26053  {
26054    \int_compare:nNnTF
26055      { \__fp_array_count:n {##1} } = { \__fp_array_count:n {##2} }
26056      { \__fp_tuple_mapthread_o:nww { \__fp_binary_type_o:Nww #1 } }
26057      { \__fp_invalid_operation_o:nww #1 }
26058      \s__fp_tuple \__fp_tuple_chk:w {##1} ;
26059      \s__fp_tuple \__fp_tuple_chk:w {##2} ;
26060    }
26061  }
26062 \__fp_tmp:w +
26063 \__fp_tmp:w -  
  
(End definition for \__fp_tuple_+_tuple_o:ww and \__fp_tuple_-_tuple_o:ww.)  
26064 </package>
```

# Chapter 73

## I3fp-extended implementation

```
26065  {*package}
```

```
26066  (@@=fp)
```

### 73.1 Description of fixed point numbers

This module provides a few functions to manipulate positive floating point numbers with extended precision (24 digits), but mostly provides functions for fixed-point numbers with this precision (24 digits). Those are used in the computation of Taylor series for the logarithm, exponential, and trigonometric functions. Since we eventually only care about the 16 first digits of the final result, some of the calculations are not performed with the full 24-digit precision. In other words, the last two blocks of each fixed point number may be wrong as long as the error is small enough to be rounded away when converting back to a floating point number. The fixed point numbers are expressed as

```
{⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;
```

where each ⟨a<sub>i</sub>⟩ is exactly 4 digits (ranging from 0000 to 9999), except ⟨a<sub>1</sub>⟩, which may be any “not-too-large” non-negative integer, with or without leading zeros. Here, “not-too-large” depends on the specific function (see the corresponding comments for details). Checking for overflow is the responsibility of the code calling those functions. The fixed point number  $a$  corresponding to the representation above is  $a = \sum_{i=1}^6 \langle a_i \rangle \cdot 10^{-4i}$ .

Most functions we define here have the form

```
\_fp_fixed_{⟨calculation⟩}:wwn ⟨operand1⟩ ; ⟨operand2⟩ ; {⟨continuation⟩}
```

They perform the ⟨calculation⟩ on the two ⟨operands⟩, then feed the result (6 brace groups followed by a semicolon) to the ⟨continuation⟩, responsible for the next step of the calculation. Some functions only accept an N-type ⟨continuation⟩. This allows constructions such as

```
\_fp_fixed_add:wwn ⟨X1⟩ ; ⟨X2⟩ ;
\_fp_fixed_mul:wwn ⟨X3⟩ ;
\_fp_fixed_add:wwn ⟨X4⟩ ;
```

to compute  $(X_1 + X_2) \cdot X_3 + X_4$ . This turns out to be very appropriate for computing continued fractions and Taylor series.

At the end of the calculation, the result is turned back to a floating point number using `\_\_fp\_fixed_to_float_o:wN`. This function has to change the exponent of the floating point number: it must be used after starting an integer expression for the overall exponent of the result.

## 73.2 Helpers for numbers with extended precision

`\c\_\_fp\_\_one\_\_fixed\_\_t1` The fixed-point number 1, used in l3fp-expo.

```
26067 \t1_const:Nn \c\_\_fp\_\_one\_\_fixed\_\_t1
26068 { {10000} {0000} {0000} {0000} {0000} ; }
```

(End definition for `\c\_\_fp\_\_one\_\_fixed\_\_t1`.)

`\_\_fp\_\_fixed\_\_continue:wn` This function simply calls the next function.

```
26069 \cs_new:Npn \_\_fp\_\_fixed\_\_continue:wn #1; #2 { #2 #1; }
```

(End definition for `\_\_fp\_\_fixed\_\_continue:wn`.)

`\_\_fp\_\_fixed\_\_add\_\_one:wn` `\_\_fp\_\_fixed\_\_add\_\_one:wN <a> ; <continuation>`

This function adds 1 to the fixed point  $\langle a \rangle$ , by changing  $a_1$  to  $10000 + a_1$ , then calls the  $\langle \text{continuation} \rangle$ . This requires  $a_1 + 10000 < 2^{31}$ .

```
26070 \cs_new:Npn \_\_fp\_\_fixed\_\_add\_\_one:wN #1#2; #3
26071 {
26072     \exp_after:wN #3 \exp_after:wN
26073         { \int_value:w \_\_fp\_\_int\_\_eval:w \c\_\_fp\_\_myriad\_\_int + #1 } #2 ;
26074 }
```

(End definition for `\_\_fp\_\_fixed\_\_add\_\_one:wn`.)

`\_\_fp\_\_fixed\_\_div\_\_myriad:wn` Divide a fixed point number by 10000. This is a little bit more subtle than just removing the last group and adding a leading group of zeros: the first group  $\#1$  may have any number of digits, and we must split  $\#1$  into the new first group and a second group of exactly 4 digits. The choice of shifts allows  $\#1$  to be in the range  $[0, 5 \cdot 10^8 - 1]$ .

```
26075 \cs_new:Npn \_\_fp\_\_fixed\_\_div\_\_myriad:wn #1#2#3#4#5#6;
26076 {
26077     \exp_after:wN \_\_fp\_\_fixed\_\_mul\_\_after:wwn
26078     \int_value:w \_\_fp\_\_int\_\_eval:w \c\_\_fp\_\_leading\_\_shift\_\_int
26079         \exp_after:wN \_\_fp\_\_pack:NNNNw
26080         \int_value:w \_\_fp\_\_int\_\_eval:w \c\_\_fp\_\_trailing\_\_shift\_\_int
26081             + #1 ; {#2}{#3}{#4}{#5};
26082 }
```

(End definition for `\_\_fp\_\_fixed\_\_div\_\_myriad:wn`.)

`\_\_fp\_\_fixed\_\_mul\_\_after:wwn` The fixed point operations which involve multiplication end by calling this auxiliary. It braces the last block of digits, and places the  $\langle \text{continuation} \rangle$   $\#3$  in front.

```
26083 \cs_new:Npn \_\_fp\_\_fixed\_\_mul\_\_after:wwn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for `\_\_fp\_\_fixed\_\_mul\_\_after:wwn`.)

### 73.3 Multiplying a fixed point number by a short one

```
\__fp_fixed_mul_short:wwn
  {⟨a1⟩} {⟨a2⟩} {⟨a3⟩} {⟨a4⟩} {⟨a5⟩} {⟨a6⟩} ;
  {⟨b0⟩} {⟨b1⟩} {⟨b2⟩} ; {⟨continuation⟩}

  Computes the product  $c = ab$  of  $a = \sum_i \langle a_i \rangle 10^{-4i}$  and  $b = \sum_i \langle b_i \rangle 10^{-4i}$ , rounds it to the closest multiple of  $10^{-24}$ , and leaves ⟨continuation⟩ {⟨c1⟩} ... {⟨c6⟩} ; in the input stream, where each of the ⟨ci⟩ are blocks of 4 digits, except ⟨c1⟩, which is any TeX integer. Note that indices for ⟨b⟩ start at 0: for instance a second operand of {0001}{0000}{0000} leaves the first operand unchanged (rather than dividing it by  $10^4$ , as \__fp_fixed_mul:wwn would).

26084 \cs_new:Npn \__fp_fixed_mul_short:wwn #1#2#3#4#5#6; #7#8#9;
26085   {
26086     \exp_after:wN \__fp_fixed_mul_after:wwn
26087     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
26088       + #1*#7
26089     \exp_after:wN \__fp_pack:NNNNw
26090     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26091       + #1*#8 + #2*#7
26092     \exp_after:wN \__fp_pack:NNNNNw
26093     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26094       + #1*#9 + #2*#8 + #3*#7
26095     \exp_after:wN \__fp_pack:NNNNNw
26096     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26097       + #2*#9 + #3*#8 + #4*#7
26098     \exp_after:wN \__fp_pack:NNNNNw
26099     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
26100       + #3*#9 + #4*#8 + #5*#7
26101     \exp_after:wN \__fp_pack:NNNNNw
26102     \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
26103       + #4*#9 + #5*#8 + #6*#7
26104       + ( #5*#9 + #6*#8 + #6*#9 / \c__fp_myriad_int )
26105       / \c__fp_myriad_int ; ;
26106   }
```

(End definition for \\_\_fp\_fixed\_mul\_short:wwn.)

### 73.4 Dividing a fixed point number by a small integer

```
\__fp_fixed_div_int:wwN
\__fp_fixed_div_int:wnN
\__fp_fixed_div_int_auxi:wnN
  \fp_fixed_div_int_auxii:wnN
\__fp_fixed_div_int_pack:Nw
\__fp_fixed_div_int_after:Nw

  \__fp_fixed_div_int:wwN ⟨a⟩ ; ⟨n⟩ ; ⟨continuation⟩

  Divides the fixed point number ⟨a⟩ by the (small) integer  $0 < \langle n \rangle < 10^4$  and feeds the result to the ⟨continuation⟩. There is no bound on a1.
```

The arguments of the i auxiliary are 1: one of the a<sub>i</sub>, 2: n, 3: the ii or the iii auxiliary. It computes a (somewhat tight) lower bound Q<sub>i</sub> for the ratio a<sub>i</sub>/n.

The ii auxiliary receives Q<sub>i</sub>, n, and a<sub>i</sub> as arguments. It adds Q<sub>i</sub> to a surrounding integer expression, and starts a new one with the initial value 9999, which ensures that the result of this expression has 5 digits. The auxiliary also computes a<sub>i</sub> - n · Q<sub>i</sub>, placing the result in front of the 4 digits of a<sub>i+1</sub>. The resulting a'<sub>i+1</sub> = 10<sup>4</sup>(a<sub>i</sub> - n · Q<sub>i</sub>) + a<sub>i+1</sub> serves as the first argument for a new call to the i auxiliary.

When the iii auxiliary is called, the situation looks like this:

```

\__fp_fixed_div_int_after:Nw <continuation>
-1 + Q1
\__fp_fixed_div_int_pack:Nw 9999 + Q2
\__fp_fixed_div_int_pack:Nw 9999 + Q3
\__fp_fixed_div_int_pack:Nw 9999 + Q4
\__fp_fixed_div_int_pack:Nw 9999 + Q5
\__fp_fixed_div_int_pack:Nw 9999
\__fp_fixed_div_int_auxii:wnn Q6 ; {⟨n⟩} {⟨a6⟩}

```

where expansion is happening from the last line up. The `iii` auxiliary adds  $Q_6 + 2 \simeq a_6/n + 1$  to the last 9999, giving the integer closest to  $10000 + a_6/n$ .

Each pack auxiliary receives 5 digits followed by a semicolon. The first digit is added as a carry to the integer expression above, and the 4 other digits are braced. Each call to the pack auxiliary thus produces one brace group. The last brace group is produced by the `after` auxiliary, which places the *<continuation>* as appropriate.

```

26107 \cs_new:Npn \__fp_fixed_div_int:wwN #1#2#3#4#5#6 ; #7 ; #8
26108 {
26109   \exp_after:wN \__fp_fixed_div_int_after:Nw
26110   \exp_after:wN #8
26111   \int_value:w \__fp_int_eval:w - 1
26112   \__fp_fixed_div_int:wnN
26113   #1; {#7} \__fp_fixed_div_int_auxi:wnn
26114   #2; {#7} \__fp_fixed_div_int_auxi:wnn
26115   #3; {#7} \__fp_fixed_div_int_auxi:wnn
26116   #4; {#7} \__fp_fixed_div_int_auxi:wnn
26117   #5; {#7} \__fp_fixed_div_int_auxi:wnn
26118   #6; {#7} \__fp_fixed_div_int_auxi:wnn ;
26119 }
26120 \cs_new:Npn \__fp_fixed_div_int:wnN #1; #2 #3
26121 {
26122   \exp_after:wN #3
26123   \int_value:w \__fp_int_eval:w #1 / #2 - 1 ;
26124   {#2}
26125   {#1}
26126 }
26127 \cs_new:Npn \__fp_fixed_div_int_auxi:wnn #1; #2 #3
26128 {
26129   + #1
26130   \exp_after:wN \__fp_fixed_div_int_pack:Nw
26131   \int_value:w \__fp_int_eval:w 9999
26132   \exp_after:wN \__fp_fixed_div_int:wnN
26133   \int_value:w \__fp_int_eval:w #3 - #1*#2 \__fp_int_eval_end:
26134 }
26135 \cs_new:Npn \__fp_fixed_div_int_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
26136 \cs_new:Npn \__fp_fixed_div_int_pack:Nw #1 #2; { + #1; {#2} }
26137 \cs_new:Npn \__fp_fixed_div_int_after:Nw #1 #2; { #1 {#2} }

```

(End definition for `\__fp_fixed_div_int:wwN` and others.)

### 73.5 Adding and subtracting fixed points

```

\__fp_fixed_add:wnn
\__fp_fixed_sub:wnn
\__fp_fixed_add:Nnnnnwnn
\__fp_fixed_add:nnNnnnnw
\__fp_fixed_add_pack:NNNNNwn
\__fp_fixed_add_after:NNNNNwn
\__fp_fixed_add:wnn ⟨a⟩ ; ⟨b⟩ ; {⟨continuation⟩}

```

Computes  $a + b$  (resp.  $a - b$ ) and feeds the result to the  $\langle\text{continuation}\rangle$ . This function requires  $0 \leq a_1, b_1 \leq 114748$ , its result must be positive (this happens automatically for addition) and its first group must have at most 5 digits:  $(a \pm b)_1 < 100000$ . The two functions only differ by a sign, hence use a common auxiliary. It would be nice to grab the 12 brace groups in one go; only 9 parameters are allowed. Start by grabbing the sign,  $a_1, \dots, a_4$ , the rest of  $a$ , and  $b_1$  and  $b_2$ . The second auxiliary receives the rest of  $a$ , the sign multiplying  $b$ , the rest of  $b$ , and the  $\langle\text{continuation}\rangle$  as arguments. After going down through the various level, we go back up, packing digits and bringing the  $\langle\text{continuation}\rangle$  (#8, then #7) from the end of the argument list to its start.

```

26138 \cs_new:Npn \__fp_fixed_add:wwn { \__fp_fixed_add:Nnnnnwn + }
26139 \cs_new:Npn \__fp_fixed_sub:wwn { \__fp_fixed_add:Nnnnnwn - }
26140 \cs_new:Npn \__fp_fixed_add:Nnnnnwn #1 #2#3#4#5 #6; #7#8
26141 {
26142     \exp_after:wN \__fp_fixed_add_after:NNNNNwn
26143     \int_value:w \__fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
26144         \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
26145         \int_value:w \__fp_int_eval:w 1 9999 9998 + #4#5
26146             \__fp_fixed_add:nnNnnwn #6 #1
26147     }
26148 \cs_new:Npn \__fp_fixed_add:nnNnnwn #1#2 #3 #4#5 #6#7 ; #8
26149 {
26150     #3 #4#5
26151     \exp_after:wN \__fp_fixed_add_pack:NNNNNwn
26152     \int_value:w \__fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
26153 }
26154 \cs_new:Npn \__fp_fixed_add_pack:NNNNNwn #1 #2#3#4#5 #6; #7
26155 { + #1 ; {#7} {#2#3#4#5} {#6} }
26156 \cs_new:Npn \__fp_fixed_add_after:NNNNNwn 1 #1 #2#3#4#5 #6; #7
26157 { #7 {#1#2#3#4#5} {#6} }
```

(End definition for  $\_\_fp\_fixed\_add:wwn$  and others.)

## 73.6 Multiplying fixed points

```

\__fp_fixed_mul:wwn
\__fp_fixed_mul:nnnnnnnw
```

$\_\_fp\_fixed\_mul:wwn \langle a \rangle ; \langle b \rangle ; \{\langle\text{continuation}\rangle\}$

Computes  $a \times b$  and feeds the result to  $\langle\text{continuation}\rangle$ . This function requires  $0 \leq a_1, b_1 < 10000$ . Once more, we need to play around the limit of 9 arguments for T<sub>E</sub>X macros. Note that we don't need to obtain an exact rounding, contrarily to the \* operator, so things could be harder. We wish to perform carries in

$$\begin{aligned}
a \times b = & a_1 \cdot b_1 \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left( a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where the  $O(10^{-24})$  stands for terms which are at most  $5 \cdot 10^{-24}$ ; ignoring those leads to an error of at most 5 ulp. Note how the first 15 terms only depend on  $a_1, \dots, a_4$  and  $b_1, \dots, b_4$ , while the last 6 terms only depend on  $a_1, a_2, a_5, a_6$ , and the corresponding parts of  $b$ . Hence, the first function grabs  $a_1, \dots, a_4$ , the rest of  $a$ , and  $b_1, \dots, b_4$ , and writes the 15 first terms of the expression, including a left parenthesis for the fraction. The `i` auxiliary receives  $a_5, a_6, b_1, b_2, a_1, a_2, b_5, b_6$  and finally the `{continuation}` as arguments. It writes the end of the expression, including the right parenthesis and the denominator of the fraction. The `{continuation}` is finally placed in front of the 6 brace groups by `\_\_fp\_fixed\_mul\_after:wwn`.

```

26158 \cs_new:Npn \_\_fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
26159   {
26160     \exp_after:wN \_\_fp_fixed_mul_after:wwn
26161     \int_value:w \_\_fp_int_eval:w \c__fp_leading_shift_int
26162       \exp_after:wN \_\_fp_pack:NNNNNw
26163       \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
26164         + #1*#6
26165       \exp_after:wN \_\_fp_pack:NNNNNw
26166       \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
26167         + #1*#7 + #2*#6
26168       \exp_after:wN \_\_fp_pack:NNNNNw
26169       \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
26170         + #1*#8 + #2*#7 + #3*#6
26171       \exp_after:wN \_\_fp_pack:NNNNNw
26172       \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
26173         + #1*#9 + #2*#8 + #3*#7 + #4*#6
26174       \exp_after:wN \_\_fp_pack:NNNNNw
26175       \int_value:w \_\_fp_int_eval:w \c__fp_trailing_shift_int
26176         + #2*#9 + #3*#8 + #4*#7
26177         + ( #3*#9 + #4*#8
26178           + \_\_fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
26179     }
26180 \cs_new:Npn \_\_fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
26181   {
26182     #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
26183     + #1*#3 + #5*#7 ; ;
26184   }

```

(End definition for `\_\_fp_fixed_mul:wwn` and `\_\_fp_fixed_mul:nnnnnnnw`.)

## 73.7 Combining product and sum of fixed points

```

\_\_fp_fixed_mul_add:wwnn
\_\_fp_fixed_mul_sub_back:wwnn
\_\_fp_fixed_mul_one_minus_mul:wwn

```

Sometimes called FMA (fused multiply-add), these functions compute  $a \times b + c$ ,  $c - a \times b$ , and  $1 - a \times b$  and feed the result to the `{continuation}`. Those functions require  $0 \leq a_1, b_1, c_1 \leq 10000$ . Since those functions are at the heart of the computation of Taylor expansions, we over-optimize them a bit, and in particular we do not factor out the common parts of the three functions.

For definiteness, consider the task of computing  $a \times b + c$ . We perform carries in

$$\begin{aligned}
a \times b + c = & (a_1 \cdot b_1 + c_1 c_2) \cdot 10^{-8} \\
& + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{-12} \\
& + (a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4) \cdot 10^{-16} \\
& + (a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1) \cdot 10^{-20} \\
& + \left( a_2 \cdot b_4 + a_3 \cdot b_3 + a_4 \cdot b_2 \right. \\
& \quad \left. + \frac{a_3 \cdot b_4 + a_4 \cdot b_3 + a_1 \cdot b_6 + a_2 \cdot b_5 + a_5 \cdot b_2 + a_6 \cdot b_1}{10^4} \right. \\
& \quad \left. + a_1 \cdot b_5 + a_5 \cdot b_1 + c_5 c_6 \right) \cdot 10^{-24} + O(10^{-24}),
\end{aligned}$$

where  $c_1 c_2$ ,  $c_3 c_4$ ,  $c_5 c_6$  denote the 8-digit number obtained by juxtaposing the two blocks of digits of  $c$ , and  $\cdot$  denotes multiplication. The task is obviously tough because we have 18 brace groups in front of us.

Each of the three function starts the first two levels (the first, corresponding to  $10^{-4}$ , is empty), with  $c_1 c_2$  in the first level, calls the `i` auxiliary with arguments described later, and adds a trailing `+ c5c6 ; {⟨continuation⟩} ;`. The `+ c5c6` piece, which is omitted for `__fp_fixed_one_minus_mul:wwn`, is taken in the integer expression for the  $10^{-24}$  level.

```

26185 \cs_new:Npn __fp_fixed_mul:wwwn #1; #2; #3#4#5#6#7#8;
26186 {
26187   \exp_after:wN __fp_fixed_mul_after:wwwn
26188   \int_value:w __fp_int_eval:w \c_fp_big_leading_shift_int
26189     \exp_after:wN __fp_pack_big:NNNNNNw
26190   \int_value:w __fp_int_eval:w \c_fp_big_middle_shift_int + #3 #4
26191     __fp_fixed_mul_add:Nwnnnnnnn +
26192       + #5 #6 ; #2 ; #1 ; #2 ; +
26193       + #7 #8 ; ;
26194 }
26195 \cs_new:Npn __fp_fixed_mul_sub_back:wwwn #1; #2; #3#4#5#6#7#8;
26196 {
26197   \exp_after:wN __fp_fixed_mul_after:wwwn
26198   \int_value:w __fp_int_eval:w \c_fp_big_leading_shift_int
26199     \exp_after:wN __fp_pack_big:NNNNNNw
26200   \int_value:w __fp_int_eval:w \c_fp_big_middle_shift_int + #3 #4
26201     __fp_fixed_mul_add:Nwnnnnnnn -
26202       + #5 #6 ; #2 ; #1 ; #2 ; -
26203       + #7 #8 ; ;
26204 }
26205 \cs_new:Npn __fp_fixed_one_minus_mul:wwwn #1; #2;
26206 {
26207   \exp_after:wN __fp_fixed_mul_after:wwwn
26208   \int_value:w __fp_int_eval:w \c_fp_big_leading_shift_int
26209     \exp_after:wN __fp_pack_big:NNNNNNw
26210   \int_value:w __fp_int_eval:w \c_fp_big_middle_shift_int +
26211     1 0000 0000
26212     __fp_fixed_mul_add:Nwnnnnnnn -
26213       ; #2 ; #1 ; #2 ; -
26214       ; ;
26215 }

```

(End definition for `\_fp_fixed_mul_add:wwnn`, `\_fp_fixed_mul_sub_back:wwnn`, and `\_fp_fixed_mul_one_minus_mul:wwn.`)

```
\_fp_fixed_mul_add:Nwnnnwnnn \_fp_fixed_mul_add:Nwnnnwnnn \op + \c3 \c4 ;
\langle b \rangle ; \langle a \rangle ; \langle b \rangle ; \langle op \rangle
+ \langle c5 \rangle \langle c6 \rangle ;
```

Here,  $\langle op \rangle$  is either + or -. Arguments #3, #4, #5 are  $\langle b_1 \rangle, \langle b_2 \rangle, \langle b_3 \rangle$ ; arguments #7, #8, #9 are  $\langle a_1 \rangle, \langle a_2 \rangle, \langle a_3 \rangle$ . We can build three levels:  $a_1 \cdot b_1$  for  $10^{-8}$ ,  $(a_1 \cdot b_2 + a_2 \cdot b_1)$  for  $10^{-12}$ , and  $(a_1 \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_1 + c_3 c_4)$  for  $10^{-16}$ . The  $a \cdot b$  products use the sign #1. Note that #2 is empty for `\_fp_fixed_one_minus_mul:wwn.` We call the `ii` auxiliary for levels  $10^{-20}$  and  $10^{-24}$ , keeping the pieces of  $\langle a \rangle$  we've read, but not  $\langle b \rangle$ , since there is another copy later in the input stream.

```
26216 \cs_new:Npn \_fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
26217 {
26218 #1 #7*#3
26219 \exp_after:wN \_fp_pack_big>NNNNNNw
26220 \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int
26221 #1 #7*#4 #1 #8*#3
26222 \exp_after:wN \_fp_pack_big>NNNNNNw
26223 \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int
26224 #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
26225 \exp_after:wN \_fp_pack_big>NNNNNNw
26226 \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int
26227 #1 \_fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
26228 }
```

(End definition for `\_fp_fixed_mul_add:Nwnnnwnnn.`)

```
\_fp_fixed_mul_add:nnnnwnnnn \a ; \b ; \op
+ \c5 \c6 ;
```

Level  $10^{-20}$  is  $(a_1 \cdot b_4 + a_2 \cdot b_3 + a_3 \cdot b_2 + a_4 \cdot b_1)$ , multiplied by the sign, which was inserted by the `i` auxiliary. Then we prepare level  $10^{-24}$ . We don't have access to all parts of  $\langle a \rangle$  and  $\langle b \rangle$  needed to make all products. Instead, we prepare the partial expressions

$$\begin{aligned} & b_1 + a_4 \cdot b_2 + a_3 \cdot b_3 + a_2 \cdot b_4 + a_1 \\ & b_2 + a_4 \cdot b_3 + a_3 \cdot b_4 + a_2. \end{aligned}$$

Obviously, those expressions make no mathematical sense: we complete them with  $a_5 \cdot$  and  $\cdot b_5$ , and with  $a_6 \cdot b_1 + a_5 \cdot$  and  $\cdot b_5 + a_1 \cdot b_6$ , and of course with the trailing  $+ c_5 c_6$ . To do all this, we keep  $a_1, a_5, a_6$ , and the corresponding pieces of  $\langle b \rangle$ .

```
26229 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
26230 {
26231 ( #1*#9 + #2*#8 + #3*#7 + #4*#6 )
26232 \exp_after:wN \_fp_pack_big>NNNNNNw
26233 \int_value:w \_fp_int_eval:w \c_fp_big_trailing_shift_int
26234 \_fp_fixed_mul_add:nnnnwnnnN
26235 { #6 + #4*#7 + #3*#8 + #2*#9 + #1 }
26236 { #7 + #4*#8 + #3*#9 + #2 }
26237 {#1} #5;
26238 {#6}
26239 }
```

(End definition for `\_fp_fixed_mul_add:nnnnwnnnn.`)

```
\_fp_fixed_mul_add:nnnnwnnnN {{partial1}} {{partial2}}
  {{a1}} {{a5}} {{a6}} ; {{b1}} {{b5}} {{b6}} ;
  ⟨op⟩ + ⟨c5⟩ ⟨c6⟩ ;
```

Complete the  $\langle \text{partial}_1 \rangle$  and  $\langle \text{partial}_2 \rangle$  expressions as explained for the `ii` auxiliary. The second one is divided by 10000: this is the carry from level  $10^{-28}$ . The trailing  $+ c_5 c_6$  is taken into the expression for level  $10^{-24}$ . Note that the total of level  $10^{-24}$  is in the interval  $[-5 \cdot 10^8, 6 \cdot 10^8]$  (give or take a couple of 10000), hence adding it to the shift gives a 10-digit number, as expected by the packing auxiliaries. See `l3fp-aux` for the definition of the shifts and packing auxiliaries.

```
26240 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnnN #1#2 #3#4#5; #6#7#8; #9
26241   {
26242     #9 (#4* #1 *#7)
26243     #9 (#5*#6+#4* #2 *#7+#3*#8) / \c_fp_myriad_int
26244 }
```

(End definition for `\_fp_fixed_mul_add:nnnnwnnnN.`)

## 73.8 Extended-precision floating point numbers

In this section we manipulate floating point numbers with roughly 24 significant figures (“extended-precision” numbers, in short, “ep”), which take the form of an integer exponent, followed by a comma, then six groups of digits, ending with a semicolon. The first group of digit may be any non-negative integer, while other groups of digits have 4 digits. In other words, an extended-precision number is an exponent ending in a comma, then a fixed point number. The corresponding value is  $0.\langle \text{digits} \rangle \cdot 10^{\langle \text{exponent} \rangle}$ . This convention differs from floating points.

Converts an extended-precision number with an exponent at most 4 and a first block less than  $10^8$  to a fixed point number whose first block has 12 digits, hopefully starting with many zeros.

```
26245 \cs_new:Npn \_fp_ep_to_fixed:wwn #1,#2
26246   {
26247     \exp_after:wN \_fp_ep_to_fixed_auxi:www
26248     \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
26249     \exp:w \exp_end_continue_f:w
26250     \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
26251   }
26252 \cs_new:Npn \_fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
26253   {
26254     \_fp_pack_eight:wNNNNNNNN
26255     \_fp_pack_twice_four:wNNNNNNNN
26256     \_fp_pack_twice_four:wNNNNNNNN
26257     \_fp_pack_twice_four:wNNNNNNNN
26258     \_fp_ep_to_fixed_auxii:nnnnnnwn ;
26259     #2 #1#3#4#5#6#7 0000 !
26260   }
26261 \cs_new:Npn \_fp_ep_to_fixed_auxii:nnnnnnwn #1#2#3#4#5#6#7; #8! #9
26262   { #9 {#1#2}{#3}{#4}{#5}{#6}{#7}; }
```

(End definition for `\_fp_ep_to_fixed:wwn`, `\_fp_ep_to_fixed_auxi:www`, and `\_fp_ep_to_fixed_auxii:nnnnnnwn`.)

```
\_\_fp\_ep\_to\_ep:wwN
\_\_fp\_ep\_to\_ep\_loop:N
\_\_fp\_ep\_to\_ep\_end:www
\_\_fp\_ep\_to\_ep\_zero:ww
```

Normalize an extended-precision number. More precisely, leading zeros are removed from the mantissa of the argument, decreasing its exponent as appropriate. Then the digits are packed into 6 groups of 4 (discarding any remaining digit, not rounding). Finally, the continuation #8 is placed before the resulting exponent-mantissa pair. The input exponent may in fact be given as an integer expression. The `loop` auxiliary grabs a digit: if it is 0, decrement the exponent and continue looping, and otherwise call the `end` auxiliary, which places all digits in the right order (the digit that was not 0, and any remaining digits), followed by some 0, then packs them up neatly in  $3 \times 2 = 6$  blocks of four. At the end of the day, remove with `\_\_fp\_use\_i:ww` any digit that did not make it in the final mantissa (typically only zeros, unless the original first block has more than 4 digits).

```
26263 \cs_new:Npn \_\_fp\_ep\_to\_ep:wwN #1,#2#3#4#5#6#7; #8
26264 {
26265   \exp_after:wN #8
26266   \int_value:w \_\_fp\_int\_eval:w #1 + 4
26267   \exp_after:wN \use_i:nn
26268   \exp_after:wN \_\_fp\_ep\_to\_ep\_loop:N
26269   \int_value:w \_\_fp\_int\_eval:w 1 0000 0000 + #2 \_\_fp\_int\_eval_end:
26270   #3#4#5#6#7 ; ;
26271 }
26272 \cs_new:Npn \_\_fp\_ep\_to\_ep\_loop:N #1
26273 {
26274   \if_meaning:w 0 #1
26275     - 1
26276   \else:
26277     \_\_fp\_ep\_to\_ep\_end:www #1
26278   \fi:
26279   \_\_fp\_ep\_to\_ep\_loop:N
26280 }
26281 \cs_new:Npn \_\_fp\_ep\_to\_ep\_end:www
26282   #1 \fi: \_\_fp\_ep\_to\_ep\_loop:N #2; #3!
26283 {
26284   \fi:
26285   \if_meaning:w ; #1
26286     - 2 * \c_fp_max_exponent_int
26287     \_\_fp\_ep\_to\_ep\_zero:ww
26288   \fi:
26289   \_\_fp\_pack\_twice\_four:wNNNNNNNN
26290   \_\_fp\_pack\_twice\_four:wNNNNNNNN
26291   \_\_fp\_pack\_twice\_four:wNNNNNNNN
26292   \_\_fp\_use\_i:ww , ;
26293   #1 #2 0000 0000 0000 0000 0000 ;
26294 }
26295 \cs_new:Npn \_\_fp\_ep\_to\_ep\_zero:ww \fi: #1; #2; #3;
26296 { \fi: , {1000}{0000}{0000}{0000}{0000}{0000} ; }
```

(End definition for `\_\_fp\_ep\_to\_ep:wwN` and others.)

In `I3fp-trig` we need to compare two extended-precision numbers. This is based on the same function for positive floating point numbers, with an extra test if comparing only 16 decimals is not enough to distinguish the numbers. Note that this function only works if the numbers are normalized so that their first block is in [1000, 9999].

```
26297 \cs_new:Npn \_\_fp\_ep\_compare:www #1,#2#3#4#5#6#7;
```

```

26298 { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
26299 \cs_new:Npn \__fp_ep_compare_aux:www #1:#2:#3,#4#5#6#7#8#9;
26300 {
26301     \if_case:w
26302         \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
26303             \if_int_compare:w #2 = #8#9 \exp_stop_f:
26304                 0
26305             \else:
26306                 \if_int_compare:w #2 < #8#9 - \fi: 1
26307             \fi:
26308     \or: 1
26309     \else: -1
26310     \fi:
26311 }
```

(End definition for `\__fp_ep_compare:www` and `\__fp_ep_compare_aux:www`.)

`\__fp_ep_mul:wwwn`  
`\__fp_ep_mul_raw:wwwnN`

Multiply two extended-precision numbers: first normalize them to avoid losing too much precision, then multiply the mantissas #2 and #4 as fixed point numbers, and sum the exponents #1 and #3. The result's first block is in [100, 9999].

```

26312 \cs_new:Npn \__fp_ep_mul:wwwn #1,#2; #3,#4;
26313 {
26314     \__fp_ep_to_ep:wnN #3,#4;
26315     \__fp_fixed_continue:wn
26316 {
26317     \__fp_ep_to_ep:wnN #1,#2;
26318     \__fp_ep_mul_raw:wwwn
26319 }
26320     \__fp_fixed_continue:wn
26321 }
26322 \cs_new:Npn \__fp_ep_mul_raw:wwwnN #1,#2; #3,#4; #5
26323 {
26324     \__fp_fixed_mul:wnN #2; #4;
26325     { \exp_after:wnN #5 \int_value:w \__fp_int_eval:w #1 + #3 , }
26326 }
```

(End definition for `\__fp_ep_mul:wwwn` and `\__fp_ep_mul_raw:wwwnN`.)

### 73.9 Dividing extended-precision numbers

Divisions of extended-precision numbers are difficult to perform with exact rounding: the technique used in `l3fp-basics` for 16-digit floating point numbers does not generalize easily to 24-digit numbers. Thankfully, there is no need for exact rounding.

Let us call  $\langle n \rangle$  the numerator and  $\langle d \rangle$  the denominator. After a simple normalization step, we can assume that  $\langle n \rangle \in [0.1, 1)$  and  $\langle d \rangle \in [0.1, 1)$ , and compute  $\langle n \rangle / (10\langle d \rangle) \in (0.01, 1)$ . In terms of the 6 blocks of digits  $\langle n_1 \rangle \cdots \langle n_6 \rangle$  and the 6 blocks  $\langle d_1 \rangle \cdots \langle d_6 \rangle$ , the condition translates to  $\langle n_1 \rangle, \langle d_1 \rangle \in [1000, 9999]$ .

We first find an integer estimate  $a \simeq 10^8/\langle d \rangle$  by computing

$$\begin{aligned}\alpha &= \left[ \frac{10^9}{\langle d_1 \rangle + 1} \right] \\ \beta &= \left[ \frac{10^9}{\langle d_1 \rangle} \right] \\ a &= 10^3\alpha + (\beta - \alpha) \cdot \left( 10^3 - \left[ \frac{\langle d_2 \rangle}{10} \right] \right) - 1250,\end{aligned}$$

where  $\left[ \frac{\bullet}{\bullet} \right]$  denotes  $\varepsilon$ -TEX's rounding division, which rounds ties away from zero. The idea is to interpolate between  $10^3\alpha$  and  $10^3\beta$  with a parameter  $\langle d_2 \rangle/10^4$ , so that when  $\langle d_2 \rangle = 0$  one gets  $a = 10^3\beta - 1250 \simeq 10^{12}/\langle d_1 \rangle \simeq 10^8/\langle d \rangle$ , while when  $\langle d_2 \rangle = 9999$  one gets  $a = 10^3\alpha - 1250 \simeq 10^{12}/(\langle d_1 \rangle + 1) \simeq 10^8/\langle d \rangle$ . The shift by 1250 helps to ensure that  $a$  is an underestimate of the correct value. We shall prove that

$$1 - 1.755 \cdot 10^{-5} < \frac{\langle d \rangle a}{10^8} < 1.$$

We can then compute the inverse of  $\langle d \rangle a/10^8 = 1 - \epsilon$  using the relation  $1/(1 - \epsilon) \simeq (1 + \epsilon)(1 + \epsilon^2) + \epsilon^4$ , which is correct up to a relative error of  $\epsilon^5 < 1.6 \cdot 10^{-24}$ . This allows us to find the desired ratio as

$$\frac{\langle n \rangle}{\langle d \rangle} = \frac{\langle n \rangle a}{10^8} ((1 + \epsilon)(1 + \epsilon^2) + \epsilon^4).$$

Let us prove the upper bound first (multiplied by  $10^{15}$ ). Note that  $10^7\langle d \rangle < 10^3\langle d_1 \rangle + 10^{-1}(\langle d_2 \rangle + 1)$ , and that  $\varepsilon$ -TEX's division  $\left[ \frac{\langle d_2 \rangle}{10} \right]$  underestimates  $10^{-1}(\langle d_2 \rangle + 1)$  by 0.5 at most, as can be checked for each possible last digit of  $\langle d_2 \rangle$ . Then,

$$10^7\langle d \rangle a < \left( 10^3\langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left( \left( 10^3 - \left[ \frac{\langle d_2 \rangle}{10} \right] \right) \beta + \left[ \frac{\langle d_2 \rangle}{10} \right] \alpha - 1250 \right) \quad (1)$$

$$< \left( 10^3\langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \quad (2)$$

$$\left( \left( 10^3 - \left[ \frac{\langle d_2 \rangle}{10} \right] \right) \left( \frac{10^9}{\langle d_1 \rangle} + \frac{1}{2} \right) + \left[ \frac{\langle d_2 \rangle}{10} \right] \left( \frac{10^9}{\langle d_1 \rangle + 1} + \frac{1}{2} \right) - 1250 \right) \quad (3)$$

$$< \left( 10^3\langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] + \frac{1}{2} \right) \left( \frac{10^{12}}{\langle d_1 \rangle} - \left[ \frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} - 750 \right) \quad (4)$$

We recognize a quadratic polynomial in  $\left[ \langle d_2 \rangle/10 \right]$  with a negative leading coefficient: this polynomial is bounded above, according to  $(\left[ \langle d_2 \rangle/10 \right] + a)(b - c[\langle d_2 \rangle/10]) \leq (b + ca)^2/(4c)$ . Hence,

$$10^7\langle d \rangle a < \frac{10^{15}}{\langle d_1 \rangle(\langle d_1 \rangle + 1)} \left( \langle d_1 \rangle + \frac{1}{2} + \frac{1}{4}10^{-3} - \frac{3}{8} \cdot 10^{-9}\langle d_1 \rangle(\langle d_1 \rangle + 1) \right)^2$$

Since  $\langle d_1 \rangle$  takes integer values within  $[1000, 9999]$ , it is a simple programming exercise to check that the squared expression is always less than  $\langle d_1 \rangle(\langle d_1 \rangle + 1)$ , hence  $10^7\langle d \rangle a < 10^{15}$ . The upper bound is proven. We also find that  $\frac{3}{8}$  can be replaced by slightly smaller numbers, but nothing less than 0.374563..., and going back through the derivation of

the upper bound, we find that 1250 is as small a shift as we can obtain without breaking the bound.

Now, the lower bound. The same computation as for the upper bound implies

$$10^7 \langle d \rangle a > \left( 10^3 \langle d_1 \rangle + \left[ \frac{\langle d_2 \rangle}{10} \right] - \frac{1}{2} \right) \left( \frac{10^{12}}{\langle d_1 \rangle} - \left[ \frac{\langle d_2 \rangle}{10} \right] \frac{10^9}{\langle d_1 \rangle (\langle d_1 \rangle + 1)} - 1750 \right)$$

This time, we want to find the minimum of this quadratic polynomial. Since the leading coefficient is still negative, the minimum is reached for one of the extreme values  $[y/10] = 0$  or  $[y/10] = 100$ , and we easily check the bound for those values.

We have proven that the algorithm gives us a precise enough answer. Incidentally, the upper bound that we derived tells us that  $a < 10^8 / \langle d \rangle \leq 10^9$ , hence we can compute  $a$  safely as a TeX integer, and even add  $10^9$  to it to ease grabbing of all the digits. The lower bound implies  $10^8 - 1755 < a$ , which we do not care about.

`\__fp_ep_div:wwwn` Compute the ratio of two extended-precision numbers. The result is an extended-precision number whose first block lies in the range [100, 9999], and is placed after the `\continuation` once we are done. First normalize the inputs so that both first block lie in [1000, 9999], then call `\__fp_ep_div_esti:wwwn` *(denominator)* *(numerator)*, responsible for estimating the inverse of the denominator.

```

26327 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
26328   {
26329     \__fp_ep_to_ep:wwN #1,#2;
26330     \__fp_fixed_continue:wn
26331   {
26332     \__fp_ep_to_ep:wwN #3,#4;
26333     \__fp_ep_div_esti:wwwn
26334   }
26335 }
```

(End definition for `\__fp_ep_div:wwwn`.)

`\__fp_ep_div_esti:wwwn`  
`\__fp_ep_div_estii:wwnnwwn`  
`\__fp_ep_div_estiii:NNNNNwwn`

The `esti` function evaluates  $\alpha = 10^9 / (\langle d_1 \rangle + 1)$ , which is used twice in the expression for  $a$ , and combines the exponents `#1` and `#4` (with a shift by 1 because we later compute  $\langle n \rangle / (10 \langle d \rangle)$ ). Then the `estii` function evaluates  $10^9 + a$ , and puts the exponent `#2` after the continuation `#7`: from there on we can forget exponents and focus on the mantissa. The `estiii` function multiplies the denominator `#7` by  $10^{-8}a$  (obtained as  $a$  split into the single digit `#1` and two blocks of 4 digits, `#2#3#4#5` and `#6`). The result  $10^{-8}a \langle d \rangle = (1 - \epsilon)$ , and a partially packed  $10^{-9}a$  (as a block of four digits, and five individual digits, not packed by lack of available macro parameters here) are passed to `\__fp_ep_div_epsi:vn>NNNNn`, which computes  $10^{-9}a / (1 - \epsilon)$ , that is,  $1 / (10 \langle d \rangle)$  and we finally multiply this by the numerator `#8`.

```

26336 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
26337   {
26338     \exp_after:wN \__fp_ep_div_estii:wwnnwwn
26339     \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
26340       \exp_after:wN ;
26341     \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
26342     {#2} #3;
26343   }
26344 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
26345   {
```

```

26346   \exp_after:wN \_fp_ep_div_estiii:NNNNNwwn
26347   \int_value:w \_fp_int_eval:w 10 0000 0000 - 1750
26348   + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
26349   {#3}{#4}#5; #6; { #7 #2, }
26350   }
26351 \cs_new:Npn \_fp_ep_div_estiii:NNNNNwwn 1#1#2#3#4#5#6; #7;
26352   {
26353     \_fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
26354     \_fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
26355     \_fp_fixed_mul:wwn
26356   }

```

(End definition for `\_fp_ep_div_esti:wwwn`, `\_fp_ep_div_estii:wwnnwwn`, and `\_fp_ep_div-estiii:NNNNNwwn`.)

`\_fp_ep_div_epsi:wnNNNNNn`  
`\_fp_ep_div_eps_pack:NNNNNw`  
`\_fp_ep_div_epsii:wwnNNNNNn`

The bounds shown above imply that the `epsi` function's first operand is  $(1 - \epsilon)$  with  $\epsilon \in [0, 1.755 \cdot 10^{-5}]$ . The `epsi` function computes  $\epsilon$  as  $1 - (1 - \epsilon)$ . Since  $\epsilon < 10^{-4}$ , its first block vanishes and there is no need to explicitly use `#1` (which is 9999). Then `epsii` evaluates  $10^{-9}a/(1 - \epsilon)$  as  $(1 + \epsilon^2)(1 + \epsilon)(10^{-9}a\epsilon) + 10^{-9}a$ . Importantly, we compute  $10^{-9}a\epsilon$  before multiplying it with the rest, rather than multiplying by  $\epsilon$  and then  $10^{-9}a$ , as this second option loses more precision. Also, the combination of `short_mul` and `div_myriad` is both faster and more precise than a simple `mul`.

```

26357 \cs_new:Npn \_fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
26358   {
26359     \exp_after:wN \_fp_ep_div_epsii:wwnNNNNNn
26360     \int_value:w \_fp_int_eval:w 1 9998 - #2
26361     \exp_after:wN \_fp_ep_div_eps_pack:NNNNNw
26362     \int_value:w \_fp_int_eval:w 1 9999 9998 - #3#4
26363     \exp_after:wN \_fp_ep_div_eps_pack:NNNNNw
26364     \int_value:w \_fp_int_eval:w 2 0000 0000 - #5#6 ; ;
26365   }
26366 \cs_new:Npn \_fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
26367   { + #1 ; {#2#3#4#5} {#6} }
26368 \cs_new:Npn \_fp_ep_div_epsii:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
26369   {
26370     \_fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
26371     \_fp_fixed_add_one:wn
26372     \_fp_fixed_mul:wwn {10000} {#1} #2 ;
26373   {
26374     \_fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
26375     \_fp_fixed_div_myriad:wn
26376     \_fp_fixed_mul:wwn
26377   }
26378   \_fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
26379 }

```

(End definition for `\_fp_ep_div_epsi:wnNNNNNn`, `\_fp_ep_div_eps_pack:NNNNNw`, and `\_fp_ep-  
div_epsii:wwnNNNNNn`.)

### 73.10 Inverse square root of extended precision numbers

The idea here is similar to division. Normalize the input, multiplying by powers of 100 until we have  $x \in [0.01, 1)$ . Then find an integer approximation  $r \in [101, 1003]$  of  $10^2/\sqrt{x}$ , as the fixed point of iterations of the Newton method: essentially  $r \mapsto (r + 10^8/(x_1 r))/2$ , starting from a guess that optimizes the number of steps before convergence. In fact, just as there is a slight shift when computing divisions to ensure that some inequalities hold, we replace  $10^8$  by a slightly larger number which ensures that  $r^2 x \geq 10^4$ . This also causes  $r \in [101, 1003]$ . Another correction to the above is that the input is actually normalized to  $[0.1, 1)$ , and we use either  $10^8$  or  $10^9$  in the Newton method, depending on the parity of the exponent. Skipping those technical hurdles, once we have the approximation  $r$ , we set  $y = 10^{-4}r^2x$  (or rather, the correct power of 10 to get  $y \simeq 1$ ) and compute  $y^{-1/2}$  through another application of Newton's method. This time, the starting value is  $z = 1$ , each step maps  $z \mapsto z(1.5 - 0.5yz^2)$ , and we perform a fixed number of steps. Our final result combines  $r$  with  $y^{-1/2}$  as  $x^{-1/2} = 10^{-2}ry^{-1/2}$ .

```
\__fp_ep_isqrt:wwn
\__fp_ep_isqrt_aux:wwn
\__fp_ep_isqrt_auxii:wwnnnwn
```

First normalize the input, then check the parity of the exponent #1. If it is even, the result's exponent will be  $-\#1/2$ , otherwise it will be  $(\#1 - 1)/2$  (except in the case where the input was an exact power of 100). The auxii function receives as #1 the result's exponent just computed, as #2 the starting value for the iteration giving  $r$  (the values 168 and 535 lead to the least number of iterations before convergence, on average), as #3 and #4 one empty argument and one 0, depending on the parity of the original exponent, as #5 and #6 the normalized mantissa (#5  $\in [1000, 9999]$ ), and as #7 the continuation. It sets up the iteration giving  $r$ : the esti function thus receives the initial two guesses #2 and 0, an approximation #5 of  $10^4x$  (its first block of digits), and the empty/zero arguments #3 and #4, followed by the mantissa and an altered continuation where we have stored the result's exponent.

```
26380 \cs_new:Npn \__fp_ep_isqrt:wwn #1,#2;
26381 {
26382     \__fp_ep_to_ep:wwN #1,#2;
26383     \__fp_ep_isqrt_auxi:wwn
26384 }
26385 \cs_new:Npn \__fp_ep_isqrt_auxi:wwn #1,
26386 {
26387     \exp_after:wn \__fp_ep_isqrt_auxii:wwnnnwn
26388     \int_value:w \__fp_int_eval:w
26389     \int_if_odd:nTF {#1}
26390     { (1 - #1) / 2 , 535 , { 0 } { } }
26391     { 1 - #1 / 2 , 168 , { } { 0 } }
26392 }
26393 \cs_new:Npn \__fp_ep_isqrt_auxii:wwnnnwn #1, #2, #3#4 #5#6; #7
26394 {
26395     \__fp_ep_isqrt_esti:wwnnnwn #2, 0, #5, {#3} {#4}
26396     {#5} #6 ; { #7 #1 , }
26397 }
```

(End definition for `\__fp_ep_isqrt:wwn`, `\__fp_ep_isqrt_aux:wwn`, and `\__fp_ep_isqrt_auxii:wwnnnwn`.)

```
\__fp_ep_isqrt_esti:wwnnnwn
\__fp_ep_isqrt_estii:wwnnnwn
\__fp_ep_isqrt_estiii:NNNNNwwn
```

If the last two approximations gave the same result, we are done: call the estii function to clean up. Otherwise, evaluate  $(\langle prev \rangle + 1.005 \cdot 10^8 \text{ or } 10^9 / (\langle prev \rangle \cdot x)) / 2$ , as the next approximation: omitting the 1.005 factor, this would be Newton's method. We can

check by brute force that if #4 is empty (the original exponent was even), the process computes an integer slightly larger than  $100/\sqrt{x}$ , while if #4 is 0 (the original exponent was odd), the result is an integer slightly larger than  $100/\sqrt{x}/10$ . Once we are done, we evaluate  $100r^2/2$  or  $10r^2/2$  (when the exponent is even or odd, respectively) and feed that to `estiii`. This third auxiliary finds  $y_{\text{even}}/2 = 10^{-4}r^2x/2$  or  $y_{\text{odd}}/2 = 10^{-5}r^2x/2$  (again, depending on earlier parity). A simple program shows that  $y \in [1, 1.0201]$ . The number  $y/2$  is fed to `\_fp_ep_isqrt_epsi:wN`, which computes  $1/\sqrt{y}$ , and we finally multiply the result by  $r$ .

```

26398 \cs_new:Npn \_fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
26399 {
26400     \if_int_compare:w #1 = #2 \exp_stop_f:
26401         \exp_after:wN \_fp_ep_isqrt_estii:wwnnwn
26402     \fi:
26403     \exp_after:wN \_fp_ep_isqrt_esti:wwnnwn
26404     \int_value:w \_fp_int_eval:w
26405         (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
26406         #1, #3, {#4}
26407     }
26408 \cs_new:Npn \_fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
26409 {
26410     \exp_after:wN \_fp_ep_isqrt_estiii:NNNNNwwnn
26411     \int_value:w \_fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
26412     \exp_after:wN , \int_value:w \_fp_int_eval:w 10000 + #2 ;
26413 }
26414 \cs_new:Npn \_fp_ep_isqrt_estiii:NNNNNwwnn 1#1#2#3#4#5#6, 1#7#8; #9;
26415 {
26416     \_fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
26417     \_fp_ep_isqrt_epsi:wN
26418     \_fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
26419 }

```

(End definition for `\_fp_ep_isqrt_esti:wwnnwn`, `\_fp_ep_isqrt_estii:wwnnwn`, and `\_fp_ep_isqrt_estiii:NNNNNwwnn`.)

`\_fp_ep_isqrt_epsi:wN` Here, we receive a fixed point number  $y/2$  with  $y \in [1, 1.0201]$ . Starting from  $z = 1$  we iterate  $z \mapsto z(3/2 - z^2y/2)$ . In fact, we start from the first iteration  $z = 3/2 - y/2$  to avoid useless multiplications. The `epsii` auxiliary receives  $z$  as #1 and  $y$  as #2.

```

26420 \cs_new:Npn \_fp_ep_isqrt_epsi:wN #1;
26421 {
26422     \_fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}; #1;
26423     \_fp_ep_isqrt_epsii:wwN #1;
26424     \_fp_ep_isqrt_epsii:wwN #1;
26425     \_fp_ep_isqrt_epsii:wwN #1;
26426 }
26427 \cs_new:Npn \_fp_ep_isqrt_epsii:wwN #1; #2;
26428 {
26429     \_fp_fixed_mul:wwn #1; #1;
26430     \_fp_fixed_mul_sub_back:wwnn #2;
26431     {15000}{0000}{0000}{0000}{0000};
26432     \_fp_fixed_mul:wwn #1;
26433 }

```

(End definition for `\_fp_ep_isqrt_epsi:wN` and `\_fp_ep_isqrt_epsii:wwN`.)

### 73.11 Converting from fixed point to floating point

After computing Taylor series, we wish to convert the result from extended precision (with or without an exponent) to the public floating point format. The functions here should be called within an integer expression for the overall exponent of the floating point.

```
\_\_fp\_ep\_to\_float\_o:wwN
\_\_fp\_inv\_to\_float\_o:wwN
```

An extended-precision number is simply a comma-delimited exponent followed by a fixed point number. Leave the exponent in the current integer expression then convert the fixed point number.

```
26434 \cs_new:Npn \_\_fp\_ep\_to\_float\_o:wwN #1,
26435   { + \_\_fp\_int\_eval:w #1 \_\_fp\_fixed\_to\_float\_o:wN }
26436 \cs_new:Npn \_\_fp\_inv\_to\_float\_o:wwN #1,#2;
26437   {
26438     \_\_fp\_ep\_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}; #1,#2;
26439     \_\_fp\_ep\_to\_float\_o:wwN
26440   }
```

(End definition for `\_\_fp\_ep\_to\_float\_o:wwN` and `\_\_fp\_inv\_to\_float\_o:wwN`.)

```
\_\_fp\_fixed\_inv\_to\_float\_o:wN
```

Another function which reduces to converting an extended precision number to a float.

```
26441 \cs_new:Npn \_\_fp\_fixed\_inv\_to\_float\_o:wN
26442   { \_\_fp\_inv\_to\_float\_o:wwN 0, }
```

(End definition for `\_\_fp\_fixed\_inv\_to\_float\_o:wN`.)

```
\_\_fp\_fixed\_to\_float\_rad\_o:wN
```

Converts the fixed point number #1 from degrees to radians then to a floating point number. This could perhaps remain in `l3fp-trig`.

```
26443 \cs_new:Npn \_\_fp\_fixed\_to\_float\_rad\_o:wN #1;
26444   {
26445     \_\_fp\_fixed\_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
26446     { \_\_fp\_ep\_to\_float\_o:wwN 2, }
26447   }
```

(End definition for `\_\_fp\_fixed\_to\_float\_rad\_o:wN`.)

```
\_\_fp\_fixed\_to\_float\_o:wN
\_\_fp\_fixed\_to\_float\_o:Nw
```

```
... \_\_fp\_int\_eval:w <exponent> \_\_fp\_fixed\_to\_float\_o:wN {<a1>} {<a2>} {<a3>}
{<a4>} {<a5>} {<a6>} ; <sign>
yields
<exponent'> ; {<a'1>} {<a'2>} {<a'3>} {<a'4>} ;
```

And the `to_fixed` version gives six brace groups instead of 4, ensuring that  $1000 \leq \langle a'_1 \rangle \leq 9999$ . At this stage, we know that  $\langle a_1 \rangle$  is positive (otherwise, it is sign of an error before), and we assume that it is less than  $10^8$ .<sup>11</sup>

```
26448 \cs_new:Npn \_\_fp\_fixed\_to\_float\_o:Nw #1#2;
26449   { \_\_fp\_fixed\_to\_float\_o:wN #2; #1 }
26450 \cs_new:Npn \_\_fp\_fixed\_to\_float\_o:wN #1#2#3#4#5#6; #7
26451   { % for the 8-digit-at-the-start thing
26452     + \_\_fp\_int\_eval:w \c\_fp\_block\_int
26453     \exp\_after:wN \exp\_after:wN
26454     \exp\_after:wN \_\_fp\_fixed\_to\_loop:N
```

---

<sup>11</sup>Bruno: I must double check this assumption.

```

26455   \exp_after:wN \use_none:n
26456   \int_value:w \_fp_int_eval:w
26457     1 0000 0000 + #1 \exp_after:wN \_fp_use_none_stop_f:n
26458     \int_value:w 1#2 \exp_after:wN \_fp_use_none_stop_f:n
26459     \int_value:w 1#3#4 \exp_after:wN \_fp_use_none_stop_f:n
26460     \int_value:w 1#5#6
26461   \exp_after:wN ;
26462   \exp_after:wN ;
26463 }
26464 \cs_new:Npn \_fp_fixed_to_loop:N #1
26465 {
26466   \if_meaning:w 0 #1
26467     - 1
26468   \exp_after:wN \_fp_fixed_to_loop:N
26469   \else:
26470     \exp_after:wN \_fp_fixed_to_loop_end:w
26471     \exp_after:wN #1
26472   \fi:
26473 }
26474 \cs_new:Npn \_fp_fixed_to_loop_end:w #1 #2 ;
26475 {
26476   \if_meaning:w ; #1
26477   \exp_after:wN \_fp_fixed_to_float_zero:w
26478   \else:
26479     \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
26480     \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
26481     \exp_after:wN \_fp_fixed_to_float_pack:ww
26482     \exp_after:wN ;
26483   \fi:
26484   #1 #2 0000 0000 0000 0000 ;
26485 }
26486 \cs_new:Npn \_fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
26487 {
26488   - 2 * \c_fp_max_exponent_int ;
26489   {0000} {0000} {0000} {0000} ;
26490 }
26491 \cs_new:Npn \_fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
26492 {
26493   \if_int_compare:w #2 > 4 \exp_stop_f:
26494     \exp_after:wN \_fp_fixed_to_float_round_up:wnnnnw
26495   \fi:
26496   ; #1 ;
26497 }
26498 \cs_new:Npn \_fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
26499 {
26500   \exp_after:wN \_fp_basics_pack_high>NNNNNw
26501   \int_value:w \_fp_int_eval:w 1 #1#2
26502   \exp_after:wN \_fp_basics_pack_low>NNNNNw
26503   \int_value:w \_fp_int_eval:w 1 #3#4 + 1 ;
26504 }

```

(End definition for `\_fp_fixed_to_float_o:wN` and `\_fp_fixed_to_float_o:Nw`.)

26505 `</package>`

# Chapter 74

## I3fp-expo implementation

```
26506  <*package>
26507  <@@=fp>
\__fp_parse_word_exp:N
\__fp_parse_word_ln:N
\__fp_parse_word_fact:N
Unary functions.
26508  \cs_new:Npn \__fp_parse_word_exp:N
26509  { \__fp_parse_unary_function:NNN \__fp_exp_o:w ? }
26510  \cs_new:Npn \__fp_parse_word_ln:N
26511  { \__fp_parse_unary_function:NNN \__fp_ln_o:w ? }
26512  \cs_new:Npn \__fp_parse_word_fact:N
26513  { \__fp_parse_unary_function:NNN \__fp_fact_o:w ? }
(End definition for \__fp_parse_word_exp:N, \__fp_parse_word_ln:N, and \__fp_parse_word_fact:N.)
```

### 74.1 Logarithm

#### 74.1.1 Work plan

As for many other functions, we filter out special cases in `\__fp_ln_o:w`. Then `\__fp_ln_npos_o:w` receives a positive normal number, which we write in the form  $a \cdot 10^b$  with  $a \in [0.1, 1]$ .

*The rest of this section is actually not in sync with the code. Or is the code not in sync with the section? In the current code,  $c \in [1, 10]$  is such that  $0.7 \leq ac < 1.4$ .*

We are given a positive normal number, of the form  $a \cdot 10^b$  with  $a \in [0.1, 1)$ . To compute its logarithm, we find a small integer  $5 \leq c < 50$  such that  $0.91 \leq ac/5 < 1.1$ , and use the relation

$$\ln(a \cdot 10^b) = b \cdot \ln(10) - \ln(c/5) + \ln(ac/5).$$

The logarithms  $\ln(10)$  and  $\ln(c/5)$  are looked up in a table. The last term is computed using the following Taylor series of  $\ln$  near 1:

$$\ln\left(\frac{ac}{5}\right) = \ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + t^2 \left(\frac{1}{3} + t^2 \left(\frac{1}{5} + t^2 \left(\frac{1}{7} + t^2 \left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

where  $t = 1 - 10/(ac + 5)$ . We can now see one reason for the choice of  $ac \sim 5$ : then  $ac + 5 = 10(1 - \epsilon)$  with  $-0.05 < \epsilon \leq 0.045$ , hence

$$t = \frac{\epsilon}{1 - \epsilon} = \epsilon(1 + \epsilon)(1 + \epsilon^2)(1 + \epsilon^4) \dots,$$

is not too difficult to compute.

### 74.1.2 Some constants

```
\c_fp_ln_i_fixed_tl
\c_fp_ln_ii_fixed_tl
\c_fp_ln_iii_fixed_tl
\c_fp_ln_iv_fixed_tl
\c_fp_ln_vi_fixed_tl
\c_fp_ln_vii_fixed_tl
\c_fp_ln_viii_fixed_tl
\c_fp_ln_ix_fixed_tl
\c_fp_ln_x_fixed_tl
```

A few values of the logarithm as extended fixed point numbers. Those are needed in the implementation. It turns out that we don't need the value of  $\ln(5)$ .

```
26514 \tl_const:Nn \c_fp_ln_i_fixed_tl { {0000}{0000}{0000}{0000}{0000}{0000}; }
26515 \tl_const:Nn \c_fp_ln_ii_fixed_tl { {6931}{4718}{0559}{9453}{0941}{7232}; }
26516 \tl_const:Nn \c_fp_ln_iii_fixed_tl {{10986}{1228}{8668}{1096}{9139}{5245}; }
26517 \tl_const:Nn \c_fp_ln_iv_fixed_tl {{13862}{9436}{1119}{8906}{1883}{4464}; }
26518 \tl_const:Nn \c_fp_ln_vi_fixed_tl {{17917}{5946}{9228}{0550}{0081}{2477}; }
26519 \tl_const:Nn \c_fp_ln_vii_fixed_tl {{19459}{1014}{9055}{3133}{0510}{5353}; }
26520 \tl_const:Nn \c_fp_ln_viii_fixed_tl {{20794}{4154}{1679}{8359}{2825}{1696}; }
26521 \tl_const:Nn \c_fp_ln_ix_fixed_tl {{21972}{2457}{7336}{2193}{8279}{0490}; }
26522 \tl_const:Nn \c_fp_ln_x_fixed_tl {{23025}{8509}{2994}{0456}{8401}{7991}; }
```

(End definition for  $\c_fp_ln_i_fixed_tl$  and others.)

### 74.1.3 Sign, exponent, and special numbers

```
\_fp_ln_o:w
```

The logarithm of negative numbers (including  $-\infty$  and  $-0$ ) raises the “invalid” exception. The logarithm of  $+0$  is  $-\infty$ , raising a division by zero exception. The logarithm of  $+\infty$  or a  $\text{nan}$  is itself. Positive normal numbers call  $\_fp_ln_npos_o:w$ .

```
26523 \cs_new:Npn \_fp_ln_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
26524 {
26525   \if_meaning:w 2 #3
26526     \_fp_case_use:nw { \_fp_invalid_operation_o:nw { \ln } }
26527   \fi:
26528   \if_case:w #2 \exp_stop_f:
26529     \_fp_case_use:nw
26530       { \_fp_division_by_zero_o:Nnw \c_minus_inf_fp { \ln } }
26531   \or:
26532   \else:
26533     \_fp_case_return_same_o:w
26534   \fi:
26535   \_fp_ln_npos_o:w \s_fp \_fp_chk:w #2#3#4;
26536 }
```

(End definition for  $\_fp_ln_o:w$ .)

### 74.1.4 Absolute ln

```
\_fp_ln_npos_o:w
```

We catch the case of a significand very close to 0.1 or to 1. In all other cases, the final result is at least  $10^{-4}$ , and then an error of  $0.5 \cdot 10^{-20}$  is acceptable.

```
26537 \cs_new:Npn \_fp_ln_npos_o:w \s_fp \_fp_chk:w 10#1#2#3;
26538   { %^^A todo: \ln(1) should be "exact zero", not "underflow"
26539     \exp_after:wN \_fp_sanitize:Nw
26540     \int_value:w % for the overall sign
26541       \if_int_compare:w #1 < \c_one_int
26542         2
26543       \else:
26544         0
26545     \fi:
```

```

26546 \exp_after:wN \exp_stop_f:
26547 \int_value:w \__fp_int_eval:w % for the exponent
26548   \__fp_ln_significand:NNNNnnnN #2#3
26549   \__fp_ln_exponent:wn {#1}
26550 }

```

(End definition for `\__fp_ln_npos_o:w.`)

`\__fp_ln_significand:NNNNnnnN`  $\langle X_1 \rangle \{ \langle X_2 \rangle \} \{ \langle X_3 \rangle \} \{ \langle X_4 \rangle \}$  (continuation)  
This function expands to

$\langle \text{continuation} \rangle \{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \} \{ \langle Y_5 \rangle \} \{ \langle Y_6 \rangle \} ;$

where  $Y = -\ln(X)$  as an extended fixed point.

```

26551 \cs_new:Npn \__fp_ln_significand:NNNNnnnN #1#2#3#4
26552 {
26553   \exp_after:wN \__fp_ln_x_ii:wnnnn
26554   \int_value:w
26555     \if_case:w #1 \exp_stop_f:
26556     \or:
26557       \if_int_compare:w #2 < 4 \exp_stop_f:
26558         \__fp_int_eval:w 10 - #2
26559       \else:
26560         6
26561       \fi:
26562       \or: 4
26563       \or: 3
26564       \or: 2
26565       \or: 2
26566       \or: 2
26567       \else: 1
26568       \fi:
26569     ; { #1 #2 #3 #4 }
26570 }

```

(End definition for `\__fp_ln_significand:NNNNnnnN.`)

`\__fp_ln_x_ii:wnnnn` We have thus found  $c \in [1, 10]$  such that  $0.7 \leq ac < 1.4$  in all cases. Compute  $1 + x = 1 + ac \in [1.7, 2.4)$ .

```

26571 \cs_new:Npn \__fp_ln_x_ii:wnnnn #1; #2#3#4#5
26572 {
26573   \exp_after:wN \__fp_ln_div_after:Nw
26574   \cs:w c__fp_ln_ \__fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
26575   \int_value:w
26576     \exp_after:wN \__fp_ln_x_iv:wnnnnnnnn
26577     \int_value:w \__fp_int_eval:w
26578       \exp_after:wN \__fp_ln_x_iii_var:NNNNNw
26579       \int_value:w \__fp_int_eval:w 9999 9990 + #1*#2#3 +
26580       \exp_after:wN \__fp_ln_x_iii:NNNNNw
26581       \int_value:w \__fp_int_eval:w 10 0000 0000 + #1*#4#5 ;
26582     {20000} {0000} {0000} {0000}
26583   } %^A todo: reoptimize (a generalization attempt failed).
26584 \cs_new:Npn \__fp_ln_x_iii:NNNNNw #1#2 #3#4#5#6 #7;
26585   { #1#2; {#3#4#5#6} {#7} }
26586 \cs_new:Npn \__fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;

```

```

26587      {
26588      #1#2#3#4#5 + 1 ;
26589      {#1#2#3#4#5} {#6}
26590  }

```

The Taylor series to be used is expressed in terms of  $t = (x - 1)/(x + 1) = 1 - 2/(x + 1)$ . We now compute the quotient with extended precision, reusing some code from `\_fp_-/_o:ww`. Note that  $1 + x$  is known exactly.

To reuse notations from `l3fp-basics`, we want to compute  $A/Z$  with  $A = 2$  and  $Z = x + 1$ . In `l3fp-basics`, we considered the case where both  $A$  and  $Z$  are arbitrary, in the range  $[0.1, 1)$ , and we had to monitor the growth of the sequence of remainders  $A, B, C$ , etc. to ensure that no overflow occurred during the computation of the next quotient. The main source of risk was our choice to define the quotient as roughly  $10^9 \cdot A/10^5 \cdot Z$ : then  $A$  was bound to be below  $2.147 \dots$ , and this limit was never far.

In our case, we can simply work with  $10^8 \cdot A$  and  $10^4 \cdot Z$ , because our reason to work with higher powers has gone: we needed the integer  $y \simeq 10^5 \cdot Z$  to be at least  $10^4$ , and now, the definition  $y \simeq 10^4 \cdot Z$  suffices.

Let us thus define  $y = \lfloor 10^4 \cdot Z \rfloor + 1 \in (1.7 \cdot 10^4, 2.4 \cdot 10^4]$ , and

$$Q_1 = \left\lfloor \frac{\lfloor 10^8 \cdot A \rfloor}{y} - \frac{1}{2} \right\rfloor.$$

(The  $1/2$  comes from how  $\varepsilon$ -TeX rounds.) As for division, it is easy to see that  $Q_1 \leq 10^4 A/Z$ , i.e.,  $Q_1$  is an underestimate.

Exactly as we did for division, we set  $B = 10^4 A - Q_1 Z$ . Then

$$\begin{aligned} 10^4 B &\leq A_1 A_2 \cdot A_3 A_4 - \left( \frac{A_1 A_2}{y} - \frac{3}{2} \right) 10^4 Z \\ &\leq A_1 A_2 \left( 1 - \frac{10^4 Z}{y} \right) + 1 + \frac{3}{2} y \\ &\leq 10^8 \frac{A}{y} + 1 + \frac{3}{2} y \end{aligned}$$

In the same way, and using  $1.7 \cdot 10^4 \leq y \leq 2.4 \cdot 10^4$ , and convexity, we get

$$\begin{aligned} 10^4 A &= 2 \cdot 10^4 \\ 10^4 B &\leq 10^8 \frac{A}{y} + 1.6y \leq 4.7 \cdot 10^4 \\ 10^4 C &\leq 10^8 \frac{B}{y} + 1.6y \leq 5.8 \cdot 10^4 \\ 10^4 D &\leq 10^8 \frac{C}{y} + 1.6y \leq 6.3 \cdot 10^4 \\ 10^4 E &\leq 10^8 \frac{D}{y} + 1.6y \leq 6.5 \cdot 10^4 \\ 10^4 F &\leq 10^8 \frac{E}{y} + 1.6y \leq 6.6 \cdot 10^4 \end{aligned}$$

Note that we compute more steps than for division: since  $t$  is not the end result, we need to know it with more accuracy (on the other hand, the ending is much simpler, as we don't need an exact rounding for transcendental functions, but just a faithful rounding).

```
\_\_fp\_ln\_x\_iv:wnnnnnnnn \langle 1 or 2 \rangle \langle 8d \rangle ; \{\langle 4d \rangle\} \{\langle 4d \rangle\} \langle fixed-tl \rangle
```

The number is  $x$ . Compute  $y$  by adding 1 to the five first digits.

```

26591 \cs_new:Npn \_\_fp\_ln\_x\_iv:wnnnnnnnn #1; #2#3#4#5 #6#7#8#9
26592 {
26593     \exp_after:wN \_\_fp\_div\_significand\_pack:NNN
26594     \int_value:w \_\_fp\_int\_eval:w
26595     \_\_fp\_ln\_div\_i:w #1 ;
26596     #6 #7 ; {#8} {#9}
26597     {#2} {#3} {#4} {#5}
26598     { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \int_value:w #1 }
26599     { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \int_value:w #1 }
26600     { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \int_value:w #1 }
26601     { \exp_after:wN \_\_fp\_ln\_div\_ii:wwn \int_value:w #1 }
26602     { \exp_after:wN \_\_fp\_ln\_div\_vi:wwn \int_value:w #1 }
26603 }
26604 \cs_new:Npn \_\_fp\_ln\_div\_i:w #1;
26605 {
26606     \exp_after:wN \_\_fp\_div\_significand\_calc:wwnnnnnnn
26607     \int_value:w \_\_fp\_int\_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
26608 }
26609 \cs_new:Npn \_\_fp\_ln\_div\_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
26610 {
26611     \exp_after:wN \_\_fp\_div\_significand\_pack:NNN
26612     \int_value:w \_\_fp\_int\_eval:w
26613     \exp_after:wN \_\_fp\_div\_significand\_calc:wwnnnnnnn
26614     \int_value:w \_\_fp\_int\_eval:w 999999 + #2 #3 / #1 ; % Q2
26615     #2 #3 ;
26616 }
26617 \cs_new:Npn \_\_fp\_ln\_div\_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
26618 {
26619     \exp_after:wN \_\_fp\_div\_significand\_pack:NNN
26620     \int_value:w \_\_fp\_int\_eval:w 1000000 + #2 #3 / #1 ; % Q6
26621 }
```

We now have essentially

```
\_\_fp\_ln\_div\_after:Nw \langle fixed tl \rangle
\_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_1$ 
\_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_2$ 
\_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_3$ 
\_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_4$ 
\_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_5$ 
\_\_fp\_div\_significand\_pack:NNN  $10^6 + Q_6$  ;
\langle exponent \rangle ; \langle continuation \rangle
```

where  $\langle fixed tl \rangle$  holds the logarithm of a number in  $[1, 10]$ , and  $\langle exponent \rangle$  is the exponent. Also, the expansion is done backwards. Then  $\_\_fp\_div\_significand\_pack:NNN$  puts things in the correct order to add the  $Q_i$  together and put semicolons between each piece. Once those have been expanded, we get

```
\_\_fp\_ln\_div\_after:Nw \langle fixed-tl \rangle \langle 1d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ;
\langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle 4d \rangle ; \langle exponent \rangle ;
```

Just as with division, we know that the first two digits are 1 and 0 because of bounds on the final result of the division  $2/(x+1)$ , which is between roughly 0.8 and 1.2. We then compute  $1 - 2/(x+1)$ , after testing whether  $2/(x+1)$  is greater than or smaller than 1.

```

26622 \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
26623 {
26624     \if_meaning:w 0 #2
26625         \exp_after:wN \__fp_ln_t_small:Nw
26626     \else:
26627         \exp_after:wN \__fp_ln_t_large>NNw
26628         \exp_after:wN -
26629     \fi:
26630     #1
26631 }
26632 \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
26633 {
26634     \exp_after:wN \__fp_ln_t_large>NNw
26635     \exp_after:wN + % <sign>
26636     \exp_after:wN #1
26637     \int_value:w \__fp_int_eval:w 9999 - #2 \exp_after:wN ;
26638     \int_value:w \__fp_int_eval:w 9999 - #3 \exp_after:wN ;
26639     \int_value:w \__fp_int_eval:w 9999 - #4 \exp_after:wN ;
26640     \int_value:w \__fp_int_eval:w 9999 - #5 \exp_after:wN ;
26641     \int_value:w \__fp_int_eval:w 9999 - #6 \exp_after:wN ;
26642     \int_value:w \__fp_int_eval:w 1 0000 - #7 ;
26643 }
26644 \__fp_ln_t_large>NNw <sign> <fixed t1>
    <t1>; <t2> ; <t3>; <t4>; <t5> ; <t6>;
    <exponent> ; <continuation>

```

Compute the square  $t^2$ , and keep  $t$  at the end with its sign. We know that  $t < 0.1765$ , so every piece has at most 4 digits. However, since we were not careful in  $\__fp_ln_t_small:w$ , they can have less than 4 digits.

```

26644 \cs_new:Npn \__fp_ln_t_large>NNw #1 #2 #3; #4; #5; #6; #7; #8;
26645 {
26646     \exp_after:wN \__fp_ln_square_t_after:w
26647     \int_value:w \__fp_int_eval:w 9999 0000 + #3##3
26648     \exp_after:wN \__fp_ln_square_t_pack:NNNNw
26649     \int_value:w \__fp_int_eval:w 9999 0000 + 2##3##4
26650     \exp_after:wN \__fp_ln_square_t_pack:NNNNw
26651     \int_value:w \__fp_int_eval:w 9999 0000 + 2##3##5 + #4##4
26652     \exp_after:wN \__fp_ln_square_t_pack:NNNNw
26653     \int_value:w \__fp_int_eval:w 9999 0000 + 2##3##6 + 2##4##5
26654     \exp_after:wN \__fp_ln_square_t_pack:NNNNw
26655     \int_value:w \__fp_int_eval:w
26656         1 0000 0000 + 2##3##7 + 2##4##6 + #5##5
26657         + (2##3##8 + 2##4##7 + 2##5##6) / 1 0000
26658         % ; ;
26659     \exp_after:wN \__fp_ln_twice_t_after:w
26660     \int_value:w \__fp_int_eval:w -1 + 2##3
26661     \exp_after:wN \__fp_ln_twice_t_pack:Nw
26662     \int_value:w \__fp_int_eval:w 9999 + 2##4
26663     \exp_after:wN \__fp_ln_twice_t_pack:Nw

```

```

26664   \int_value:w \_fp_int_eval:w 9999 + 2*#5
26665   \exp_after:wN \_fp_ln_twice_t_pack:Nw
26666   \int_value:w \_fp_int_eval:w 9999 + 2*#6
26667   \exp_after:wN \_fp_ln_twice_t_pack:Nw
26668   \int_value:w \_fp_int_eval:w 9999 + 2*#7
26669   \exp_after:wN \_fp_ln_twice_t_pack:Nw
26670   \int_value:w \_fp_int_eval:w 10000 + 2*#8 ; ;
26671   { \_fp_ln_c:NwNw #1 }
26672   #2
26673   }
26674 \cs_new:Npn \_fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
26675 \cs_new:Npn \_fp_ln_twice_t_after:w #1; { ; ; {#1} }
26676 \cs_new:Npn \_fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
26677 { + #1#2#3#4#5 ; {#6} }
26678 \cs_new:Npn \_fp_ln_square_t_after:w 1 0 #1#2#3 #4;
26679 { \_fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

(End definition for \_fp_ln_x_i:wnnnn.)

```

\\_fp\_ln\_Taylor:wwNw Denoting  $T = t^2$ , we get

```

\_fp_ln_Taylor:wwNw
{⟨T1⟩} {⟨T2⟩} {⟨T3⟩} {⟨T4⟩} {⟨T5⟩} {⟨T6⟩} ; ;
{⟨(2t)1⟩} {⟨(2t)2⟩} {⟨(2t)3⟩} {⟨(2t)4⟩} {⟨(2t)5⟩} {⟨(2t)6⟩} ;
{ \_fp_ln_c:NwNw ⟨sign⟩ }
⟨fixed tl⟩ ⟨exponent⟩ ; ⟨continuation⟩

```

And we want to compute

$$\ln\left(\frac{1+t}{1-t}\right) = 2t \left(1 + T\left(\frac{1}{3} + T\left(\frac{1}{5} + T\left(\frac{1}{7} + T\left(\frac{1}{9} + \dots\right)\right)\right)\right)\right)$$

The process looks as follows

```

\loop 5; A;
\div_int 5; 1.0; \add A; \mul T; {\loop \eval 5-2;}
\add 0.2; A; \mul T; {\loop \eval 5-2;}
\mul B; T; {\loop 3;}
\loop 3; C;

```

This uses the routine for dividing a number by a small integer ( $< 10^4$ ).

```

26680 \cs_new:Npn \_fp_ln_Taylor:wwNw
26681 { \_fp_ln_Taylor_loop:www 21 ; {0000}{0000}{0000}{0000}{0000} ; }
26682 \cs_new:Npn \_fp_ln_Taylor_loop:www #1; #2; #3;
26683 {
26684   \if_int_compare:w #1 = \c_one_int
26685     \_fp_ln_Taylor_break:w
26686   \fi:
26687   \exp_after:wN \_fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
26688   \_fp_fixed_add:wwN #2;
26689   \_fp_fixed_mul:wwN #3;
26690   {
26691     \exp_after:wN \_fp_ln_Taylor_loop:www
26692     \int_value:w \_fp_int_eval:w #1 - 2 ;
26693   }

```

```

26694      #3;
26695  }
26696 \cs_new:Npn \__fp_ln_Taylor_break:w \fi: #1 \__fp_fixed_add:wwn #2#3; #4 ;
26697  {
26698  \fi:
26699  \exp_after:wN \__fp_fixed_mul:wwn
26700  \exp_after:wN { \int_value:w \__fp_int_eval:w 10000 + #2 } #3;
26701 }

```

(End definition for `\__fp_ln_Taylor:wwNw.`)

```

\__fp_ln_c:NwNw <sign>
  {⟨r1⟩} {⟨r2⟩} {⟨r3⟩} {⟨r4⟩} {⟨r5⟩} {⟨r6⟩} ;
  ⟨fixed t1⟩ ⟨exponent⟩ ; ⟨continuation⟩

```

We are now reduced to finding  $\ln(c)$  and  $\langle \text{exponent} \rangle \ln(10)$  in a table, and adding it to the mixture. The first step is to get  $\ln(c) - \ln(x) = -\ln(a)$ , then we get  $b \ln(10)$  and add or subtract.

For now,  $\ln(x)$  is given as  $\cdot 10^0$ . Unless both the exponent is 1 and  $c = 1$ , we shift to working in units of  $\cdot 10^4$ , since the final result is at least  $\ln(10/7) \simeq 0.35$ .

```

26702 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
26703  {
26704  \if_meaning:w + #1
26705  \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
26706  \else:
26707  \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
26708  \fi:
26709  #3 #2 ;
26710 }

```

(End definition for `\__fp_ln_c:NwNw.`)

```

\__fp_ln_exponent:wn
  {⟨s1⟩} {⟨s2⟩} {⟨s3⟩} {⟨s4⟩} {⟨s5⟩} {⟨s6⟩} ;
  ⟨exponent⟩

```

Compute  $\langle \text{exponent} \rangle$  times  $\ln(10)$ . Apart from the cases where  $\langle \text{exponent} \rangle$  is 0 or 1, the result is necessarily at least  $\ln(10) \simeq 2.3$  in magnitude. We can thus drop the least significant 4 digits. In the case of a very large (positive or negative) exponent, we can (and we need to) drop 4 additional digits, since the result is of order  $10^4$ . Naively, one would think that in both cases we can drop 4 more digits than we do, but that would be slightly too tight for rounding to happen correctly. Besides, we already have addition and subtraction for 24 digits fixed point numbers.

```

26711 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
26712  {
26713  \if_case:w #2 \exp_stop_f:
26714  0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
26715  \or:
26716  \exp_after:wN \__fp_ln_exponent_one:ww \int_value:w
26717  \else:
26718  \if_int_compare:w #2 > \c_zero_int
26719  \exp_after:wN \__fp_ln_exponent_small:NNww
26720  \exp_after:wN 0
26721  \exp_after:wN \__fp_fixed_sub:wwn \int_value:w
26722  \else:

```

```

26723     \exp_after:wN \__fp_ln_exponent_small:NNww
26724     \exp_after:wN 2
26725     \exp_after:wN \__fp_fixed_add:wwn \int_value:w -
26726     \fi:
26727     \fi:
26728     #2; #1;
26729 }

```

Now we painfully write all the cases.<sup>12</sup> No overflow nor underflow can happen, except when computing  $\ln(1)$ .

```

26730 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
26731 {
26732     0
26733     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
26734     \__fp_fixed_to_float_o:wn 0
26735 }

```

For small exponents, we just drop one block of digits, and set the exponent of the log to 4 (minus any shift coming from leading zeros in the conversion from fixed point to floating point). Note that here the exponent has been made positive.

```

26736 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
26737 {
26738     4
26739     \exp_after:wN \__fp_fixed_mul:wwn
26740         \c__fp_ln_x_fixed_tl
26741         {#3}{0000}{0000}{0000}{0000} ;
26742     #2
26743     {0000}{#4}{#5}{#6}{#7}{#8};
26744     \__fp_fixed_to_float_o:wn #1
26745 }

```

(End definition for  $\__fp_ln_exponent:wn$ .)

## 74.2 Exponential

### 74.2.1 Sign, exponent, and special numbers

```

\__fp_exp_o:w
26746 \cs_new:Npn \__fp_exp_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
26747 {
26748     \if_case:w #2 \exp_stop_f:
26749         \__fp_case_return_o:Nw \c_one_fp
26750     \or:
26751         \exp_after:wN \__fp_exp_normal_o:w
26752     \or:
26753         \if_meaning:w 0 #3
26754             \exp_after:wN \__fp_case_return_o:Nw
26755             \exp_after:wN \c_inf_fp
26756         \else:
26757             \exp_after:wN \__fp_case_return_o:Nw
26758             \exp_after:wN \c_zero_fp
26759     \fi:

```

---

<sup>12</sup>Bruno: do rounding.

```

26760     \or:
26761         \_fp_case_return_same_o:w
26762     \fi:
26763     \s_fp \_fp_chk:w #2#3#4;
26764 }

```

(End definition for `\_fp_exp_o:w`.)

```

\_\_fp\_exp\_normal_o:w
\_\_fp\_exp\_pos_o:NnwNw
\_\_fp\_exp\_overflow>NN
26765 \cs_new:Npn \_fp_exp_normal_o:w \s_fp \_fp_chk:w 1#1
26766 {
26767     \if_meaning:w 0 #1
26768         \_fp_exp_pos_o>NNnw + \_fp_fixed_to_float_o:wN
26769     \else:
26770         \_fp_exp_pos_o>NNnw - \_fp_fixed_inv_to_float_o:wN
26771     \fi:
26772 }
26773 \cs_new:Npn \_fp_exp_pos_o>NNnw #1#2#3 \fi: #4#5;
26774 {
26775     \fi:
26776     \if_int_compare:w #4 > \c_fp_max_exp_exponent_int
26777         \token_if_eq_charcode:NNTF + #1
26778             { \_fp_exp_overflow>NN \_fp_overflow:w \c_inf_fp }
26779             { \_fp_exp_overflow>NN \_fp_underflow:w \c_zero_fp }
26780         \exp:w
26781     \else:
26782         \exp_after:wN \_fp_sanitize:Nw
26783         \exp_after:wN 0
26784         \int_value:w #1 \_fp_int_eval:w
26785             \if_int_compare:w #4 < \c_zero_int
26786                 \exp_after:wN \use_i:nn
26787             \else:
26788                 \exp_after:wN \use_ii:nn
26789             \fi:
26790         {
26791             0
26792             \_fp_decimate:nNnnn { - #4 }
26793                 \_fp_exp_Taylor:Nnnwn
26794         }
26795         {
26796             \_fp_decimate:nNnnn { \c_fp_prec_int - #4 }
26797                 \_fp_exp_pos_large:NnnNwn
26798         }
26799         #5
26800         {#4}
26801         #1 #2 0
26802         \exp:w
26803     \fi:
26804         \exp_after:wN \exp_end:
26805 }
26806 \cs_new:Npn \_fp_exp_overflow>NN #1#2
26807 {
26808     \exp_after:wN \exp_after:wN
26809     \exp_after:wN #1

```

```

26810      \exp_after:wN #2
26811  }

```

(End definition for `\_fp_exp_normal_o:w`, `\_fp_exp_pos_o:Nnwnw`, and `\_fp_exp_overflow>NN`.)

```

\fp_exp_Taylor:Nnnwn
\fp_exp_Taylor_loop:www
\_fp_exp_Taylor_break:Nww

```

This function is called for numbers in the range  $[10^{-9}, 10^{-1}]$ . We compute 10 terms of the Taylor series. The first argument is irrelevant (rounding digit used by some other functions). The next three arguments, at least 16 digits, delimited by a semicolon, form a fixed point number, so we pack it in blocks of 4 digits.

```

26812 \cs_new:Npn \fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
26813  {
26814  #6
26815  \fp_pack_twice_four:wNNNNNNNN
26816  \fp_pack_twice_four:wNNNNNNNN
26817  \fp_pack_twice_four:wNNNNNNNN
26818  \fp_exp_Taylor_ii:ww
26819  ; #2#3#4 0000 0000 ;
26820  }
26821 \cs_new:Npn \fp_exp_Taylor_ii:ww #1; #2;
26822  { \fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_fp_stop }
26823 \cs_new:Npn \fp_exp_Taylor_loop:www #1; #2; #3;
26824  {
26825  \if_int_compare:w #1 = \c_one_int
26826  \exp_after:wN \fp_exp_Taylor_break:Nww
26827  \fi:
26828  \fp_fixed_div_int:wwN #3 ; #1 ;
26829  \fp_fixed_add_one:wN
26830  \fp_fixed_mul:wwn #2 ;
26831  {
26832  \exp_after:wN \fp_exp_Taylor_loop:www
26833  \int_value:w \fp_int_eval:w #1 - 1 ;
26834  #2 ;
26835  }
26836  }
26837 \cs_new:Npn \fp_exp_Taylor_break:Nww #1 #2; #3 \s_fp_stop
26838  { \fp_fixed_add_one:wN #2 ; }

```

(End definition for `\_fp_exp_Taylor:Nnnwn`, `\_fp_exp_Taylor_loop:www`, and `\_fp_exp_Taylor_break:Nww`.)

```
\c_fp_exp_intarray
```

The integer array has  $6 \times 9 \times 4 = 216$  items encoding the values of  $\exp(j \times 10^i)$  for  $j = 1, \dots, 9$  and  $i = -1, \dots, 4$ . Each value is expressed as  $\simeq 10^p \times 0.m_1m_2m_3$  with three 8-digit blocks  $m_1, m_2, m_3$  and an integer exponent  $p$  (one more than the scientific exponent), and these are stored in the integer array as four items:  $p, 10^8 + m_1, 10^8 + m_2, 10^8 + m_3$ . The various exponentials are stored in increasing order of  $j \times 10^i$ .

Storing this data in an integer array makes it slightly harder to access (slower, too), but uses 16 bytes of memory per exponential stored, while storing as tokens used around 40 tokens; tokens have an especially large footprint in Unicode-aware engines.

```

26839 \intarray_const_from_clist:Nn \c_fp_exp_intarray
26840  {
26841  1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
26842  1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
26843  1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
26844  1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,

```

```

26845      1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
26846      1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
26847      1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
26848      1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
26849      1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
26850      1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
26851      1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
26852      2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
26853      2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
26854      3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
26855      3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
26856      4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
26857      4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
26858      4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
26859      5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
26860      9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
26861     14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
26862     18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
26863     22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
26864     27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
26865     31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
26866     35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
26867     40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
26868     44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
26869     87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
26870    131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
26871    174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
26872    218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
26873    261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
26874    305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
26875    348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
26876    391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
26877    435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
26878    869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
26879   1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
26880   1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
26881   2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
26882   2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
26883   3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
26884   3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
26885   3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
26886   4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
26887   8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
26888  13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
26889  17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
26890  21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
26891  26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
26892  30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
26893  34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
26894  39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
26895      }

```

(End definition for `\c_fp_exp_intarray`.)

`\_fp_exp_pos_large:NnnNwn` The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros).

`\_fp_exp_large_after:wnw`

`\_fp_exp_large:NwN`

`\_fp_exp_intarray:w`

`\_fp_exp_intarray_aux:w`

The third argument is the integer part of our number, then we have the decimal part delimited by a semicolon, and finally the exponent, in the range [0, 5]. Remove leading zeros from the integer part: putting #4 in there too ensures that an integer part of 0 is also removed. Then read digits one by one, looking up  $\exp(\langle digit \rangle \cdot 10^{\langle exponent \rangle})$  in a table, and multiplying that to the current total. The loop is done by `\_\_fp\_exp\_large:NwN`, whose #1 is the  $\langle exponent \rangle$ , #2 is the current mantissa, and #3 is the  $\langle digit \rangle$ . At the end, `\_\_fp\_exp\_large\_after:wwn` moves on to the Taylor series, eventually multiplied with the mantissa that we have just computed.

```

26896 \cs_new:Npn \_\_fp\_exp\_pos\_large:NnnNwn #1#2#3 #4#5; #6
26897 {
26898     \exp_after:wN \exp_after:wN \exp_after:wN \_\_fp\_exp\_large:NwN
26899     \exp_after:wN \exp_after:wN \exp_after:wN #6
26900     \exp_after:wN \c\_fp\_one\_fixed\_tl
26901     \int_value:w #3 #4 \exp_stop_f:
26902     #5 00000 ;
26903 }
26904 \cs_new:Npn \_\_fp\_exp\_large:NwN #1#2; #3
26905 {
26906     \if_case:w #3 ~
26907         \exp_after:wN \_\_fp\_fixed\_continue:wn
26908     \else:
26909         \exp_after:wN \_\_fp\_exp\_intarray:w
26910         \int_value:w \_\_fp\_int\_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
26911     \fi:
26912     #2;
26913 {
26914     \if_meaning:w 0 #1
26915         \exp_after:wN \_\_fp\_exp\_large\_after:wwn
26916     \else:
26917         \exp_after:wN \_\_fp\_exp\_large:NwN
26918         \int_value:w \_\_fp\_int\_eval:w #1 - 1 \exp_after:wN \scan_stop:
26919     \fi:
26920 }
26921 }
26922 \cs_new:Npn \_\_fp\_exp\_intarray:w #1 ;
26923 {
26924     +
26925     \_\_kernel\_intarray\_item:Nn \c\_fp\_exp\_intarray
26926     { \_\_fp\_int\_eval:w #1 - 3 \scan_stop: }
26927     \exp_after:wN \use_i:nnn
26928     \exp_after:wN \_\_fp\_fixed\_mul:wwn
26929     \int_value:w 0
26930     \exp_after:wN \_\_fp\_exp\_intarray\_aux:w
26931     \int_value:w \_\_kernel\_intarray\_item:Nn
26932         \c\_fp\_exp\_intarray { \_\_fp\_int\_eval:w #1 - 2 }
26933     \exp_after:wN \_\_fp\_exp\_intarray\_aux:w
26934     \int_value:w \_\_kernel\_intarray\_item:Nn
26935         \c\_fp\_exp\_intarray { \_\_fp\_int\_eval:w #1 - 1 }
26936     \exp_after:wN \_\_fp\_exp\_intarray\_aux:w
26937     \int_value:w \_\_kernel\_intarray\_item:Nn \c\_fp\_exp\_intarray {#1} ; ;
26938 }
26939 \cs_new:Npn \_\_fp\_exp\_intarray\_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
26940 \cs_new:Npn \_\_fp\_exp\_large\_after:wwn #1; #2; #3

```

```

26941   {
26942     \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
26943     \__fp_fixed_mul:wwn #1;
26944   }

```

(End definition for `\__fp_exp_pos_large:NnnNwn` and others.)

## 74.3 Power

Raising a number  $a$  to a power  $b$  leads to many distinct situations.

$a^b$	$-\infty$	$(-\infty, -0)$	-integer	$\pm 0$	+integer	$(0, \infty)$	$+\infty$	nan
$+\infty$	+0	+0		+1	$+\infty$		$+\infty$	nan
$(1, \infty)$	+0	$+ a ^b$		+1	$+ a ^b$		$+\infty$	nan
+1	+1	+1		+1	+1		+1	+1
$(0, 1)$	$+\infty$	$+ a ^b$		+1	$+ a ^b$		+0	nan
+0	$+\infty$	$+\infty$		+1	$+\infty$		+0	nan
-0	$+\infty$	nan	$(-1)^b \infty$	+1	$(-1)^b 0$	+0	+0	nan
$(-1, 0)$	$+\infty$	nan	$(-1)^b  a ^b$	+1	$(-1)^b  a ^b$	nan	+0	nan
-1	+1	nan	$(-1)^b$	+1	$(-1)^b$	nan	+1	nan
$(-\infty, -1)$	+0	nan	$(-1)^b  a ^b$	+1	$(-1)^b  a ^b$	nan	$+\infty$	nan
$-\infty$	+0	+0	$(-1)^b 0$	+1	$(-1)^b \infty$	nan	$+\infty$	nan
nan	nan	nan	nan	+1	nan	nan	nan	nan

We distinguished in this table the cases of finite (positive or negative) integer exponents, as  $(-1)^b$  is defined in that case. One peculiarity of this operation is that  $\text{nan}^0 = 1^{\text{nan}} = 1$ , because this relation is obeyed for any number, even  $\pm\infty$ .

`\__fp_<^>_o:ww` We cram most of the tests into a single function to save csnames. First treat the case  $b = 0$ :  $a^0 = 1$  for any  $a$ , even `nan`. Then test the sign of  $a$ .

- If it is positive, and  $a$  is a normal number, call `\__fp_pow_normal_o:ww` followed by the two `fp`  $a$  and  $b$ . For  $a = +0$  or  $+\infty$ , call `\__fp_pow_zero_or_inf:ww` instead, to return either  $+0$  or  $+\infty$  as appropriate.
- If  $a$  is a `nan`, then skip to the next semicolon (which happens to be conveniently the end of  $b$ ) and return `nan`.
- Finally, if  $a$  is negative, compute  $a^b$  (`\__fp_pow_normal_o:ww` which ignores the sign of its first operand), and keep an extra copy of  $a$  and  $b$  (the second brace group, containing  $\{ b a \}$ , is inserted between  $a$  and  $b$ ). Then do some tests to find the final sign of the result if it exists.

```

26945 \cs_new:cpn { __fp_<^>_o:ww }
26946   \s__fp \__fp_chk:w #1#2#3; \s__fp \__fp_chk:w #4#5#6;
26947   {
26948     \if_meaning:w 0 #4
26949       \__fp_case_return_o:Nw \c_one_fp
26950     \fi:
26951     \if_case:w #2 \exp_stop_f:
26952       \exp_after:wN \use_i:nn
26953     \or:
26954       \__fp_case_return_o:Nw \c_nan_fp

```

```

26955     \else:
26956         \exp_after:wN \__fp_pow_neg:www
26957         \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
26958     \fi:
26959     {
26960         \if_meaning:w 1 #1
26961             \exp_after:wN \__fp_pow_normal_o:ww
26962         \else:
26963             \exp_after:wN \__fp_pow_zero_or_inf:ww
26964         \fi:
26965         \s_fp \__fp_chk:w #1#2#3;
26966     }
26967     { \s_fp \__fp_chk:w #4#5#6; \s_fp \__fp_chk:w #1#2#3; }
26968     \s_fp \__fp_chk:w #4#5#6;
26969 }

```

(End definition for  $\_\_fp\_\wedge_o:ww$ .)

$\_\_fp\_pow\_zero\_or\_inf:ww$  Raising  $-0$  or  $-\infty$  to `nan` yields `nan`. For other powers, the result is  $+0$  if  $0$  is raised to a positive power or  $\infty$  to a negative power, and  $+\infty$  otherwise. Thus, if the type of  $a$  and the sign of  $b$  coincide, the result is  $0$ , since those conveniently take the same possible values,  $0$  and  $2$ . Otherwise, either  $a = \pm\infty$  and  $b > 0$  and the result is  $+\infty$ , or  $a = \pm 0$  with  $b < 0$  and we have a division by zero unless  $b = -\infty$ .

```

26970 \cs_new:Npn \__fp_pow_zero_or_inf:ww
26971     \s_fp \__fp_chk:w #1#2; \s_fp \__fp_chk:w #3#4
26972 {
26973     \if_meaning:w 1 #4
26974         \__fp_case_return_same_o:w
26975     \fi:
26976     \if_meaning:w #1 #4
26977         \__fp_case_return_o:Nw \c_zero_fp
26978     \fi:
26979     \if_meaning:w 2 #1
26980         \__fp_case_return_o:Nw \c_inf_fp
26981     \fi:
26982     \if_meaning:w 2 #3
26983         \__fp_case_return_o:Nw \c_inf_fp
26984     \else:
26985         \__fp_case_use:nw
26986         {
26987             \__fp_division_by_zero_o>NNww \c_inf_fp ^
26988             \s_fp \__fp_chk:w #1 #2 ;
26989         }
26990     \fi:
26991     \s_fp \__fp_chk:w #3#4
26992 }

```

(End definition for  $\_\_fp\_pow\_zero\_or\_inf:ww$ .)

$\_\_fp\_normal_o:ww$  We have in front of us  $a$ , and  $b \neq 0$ , we know that  $a$  is a normal number, and we wish to compute  $|a|^b$ . If  $|a| = 1$ , we return  $1$ , unless  $a = -1$  and  $b$  is `nan`. Indeed, returning  $1$  at this point would wrongly raise “invalid” when the sign is considered. If  $|a| \neq 1$ , test the type of  $b$ :

- 0 Impossible, we already filtered  $b = \pm 0$ .

- 1 Call `\_fp_pow_npos_o:Nww`.
- 2 Return  $+\infty$  or  $+0$  depending on the sign of  $b$  and whether the exponent of  $a$  is positive or not.
- 3 Return  $b$ .

```

26993 \cs_new:Npn \_fp_pow_normal_o:ww
26994   \s_fp \_fp_chk:w 1 #1#2#3; \s_fp \_fp_chk:w #4#5
26995 {
26996   \if:w 0 \_fp_str_if_eq:nn { #2 #3 } { 1 {1000} {0000} {0000} {0000} }
26997     \if_int_compare:w #4 #1 = 32 \exp_stop_f:
26998       \exp_after:wN \_fp_case_return_ii_o:ww
26999     \fi:
27000     \_fp_case_return_o:Nww \c_one_fp
27001   \fi:
27002   \if_case:w #4 \exp_stop_f:
27003     \or:
27004       \exp_after:wN \_fp_pow_npos_o:Nww
27005       \exp_after:wN #5
27006     \or:
27007       \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
27008       \if_int_compare:w #2 > \c_zero_int
27009         \exp_after:wN \_fp_case_return_o:Nww
27010         \exp_after:wN \c_inf_fp
27011       \else:
27012         \exp_after:wN \_fp_case_return_o:Nww
27013         \exp_after:wN \c_zero_fp
27014       \fi:
27015     \or:
27016       \_fp_case_return_ii_o:ww
27017     \fi:
27018     \s_fp \_fp_chk:w 1 #1 {#2} #3 ;
27019     \s_fp \_fp_chk:w #4 #5
27020 }

```

(End definition for `\_fp_pow_normal_o:ww`.)

`\_fp_pow_npos_o:Nww` We now know that  $a \neq \pm 1$  is a normal number, and  $b$  is a normal number too. We want to compute  $|a|^b = (|x| \cdot 10^n)^{y \cdot 10^p} = \exp((\ln|x| + n \ln(10)) \cdot y \cdot 10^p) = \exp(z)$ . To compute the exponential accurately, we need to know the digits of  $z$  up to the 16-th position. Since the exponential of  $10^5$  is infinite, we only need at most 21 digits, hence the fixed point result of `\_fp_ln_o:w` is precise enough for our needs. Start an integer expression for the decimal exponent of  $e^{|z|}$ . If  $z$  is negative, negate that decimal exponent, and prepare to take the inverse when converting from the fixed point to the floating point result.

```

27021 \cs_new:Npn \_fp_pow_npos_o:Nww #1 \s_fp \_fp_chk:w 1#2#3
27022 {
27023   \exp_after:wN \_fp_sanitize:Nw
27024   \exp_after:wN 0
27025   \int_value:w
27026   \if:w #1 \if_int_compare:w #3 > \c_zero_int 0 \else: 2 \fi:
27027     \exp_after:wN \_fp_pow_npos_aux:NNnww
27028     \exp_after:wN +
27029     \exp_after:wN \_fp_fixed_to_float_o:wN

```

```

27030     \else:
27031         \exp_after:wN \__fp_pow_npos_aux:NNnww
27032         \exp_after:wN -
27033         \exp_after:wN \__fp_fixed_inv_to_float_o:wN
27034     \fi:
27035     {#3}
27036 }

```

(End definition for `\__fp_pow_npos_o:Nww.`)

`\__fp_pow_npos_aux:NNnww` The first argument is the conversion function from fixed point to float. Then comes an exponent and the 4 brace groups of  $x$ , followed by  $b$ . Compute  $-\ln(x)$ .

```

27037 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s_fp \__fp_chk:w 1#6#7#8;
27038 {
27039     #1
27040     \__fp_int_eval:w
27041         \__fp_ln_significand>NNNNnnnN #4#5
27042         \__fp_pow_exponent:wnN {#3}
27043         \__fp_fixed_mul:wwn #8 {0000}{0000} ;
27044         \__fp_pow_B:wwN #7;
27045         #1 #2 0 % fixed_to_float_o:wN
27046     }
27047 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
27048 {
27049     \if_int_compare:w #2 > \c_zero_int
27050         \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
27051         \exp_after:wN +
27052     \else:
27053         \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
27054         \exp_after:wN -
27055     \fi:
27056     #2; #1;
27057 }
27058 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
27059 { %^A todo: use that in ln.
27060     \exp_after:wN \__fp_fixed_mul_after:wwn
27061     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
27062         \exp_after:wN \__fp_pack>NNNNNw
27063         \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27064             #1#2*23025 - #1 #3
27065         \exp_after:wN \__fp_pack>NNNNNw
27066         \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27067             #1 #2*8509 - #1 #4
27068         \exp_after:wN \__fp_pack>NNNNNw
27069         \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27070             #1 #2*2994 - #1 #5
27071         \exp_after:wN \__fp_pack>NNNNNw
27072         \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
27073             #1 #2*0456 - #1 #6
27074         \exp_after:wN \__fp_pack>NNNNNw
27075         \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
27076             #1 #2*8401 - #1 #7
27077             #1 ( #2*7991 - #8 ) / 1 0000 ; ;
27078 }

```

```

27079 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
27080 {
27081     \if_int_compare:w #7 < \c_zero_int
27082         \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
27083     \else:
27084         \if_int_compare:w #7 < 22 \exp_stop_f:
27085             \exp_after:wN \__fp_pow_C_pos:w \int_value:w
27086         \else:
27087             \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
27088         \fi:
27089     \fi:
27090     #7 \exp_after:wN ;
27091     \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
27092     #2#3#4#5#6 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
27093 }
27094 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
27095 {
27096     + 2 * \c__fp_max_exponent_int
27097     \exp_after:wN \__fp_fixed_continue:wn \c__fp_one_fixed_tl
27098 }
27099 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
27100 {
27101     \exp_after:wN \exp_after:wN \exp_after:wn \__fp_pow_C_pack:w
27102     \prg_replicate:nn {#1} {0}
27103 }
27104 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
27105 { \__fp_pow_C_pos_loop:wN #1; }
27106 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
27107 {
27108     \if_meaning:w 0 #1
27109         \exp_after:wN \__fp_pow_C_pack:w
27110         \exp_after:wN #2
27111     \else:
27112         \if_meaning:w 0 #2
27113             \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
27114         \else:
27115             \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
27116         \fi:
27117             \__fp_int_eval:w #1 - 1 \exp_after:wN ;
27118         \fi:
27119     }
27120 \cs_new:Npn \__fp_pow_C_pack:w
27121 {
27122     \exp_after:wN \__fp_exp_large:NwN
27123     \exp_after:wN 5
27124     \c__fp_one_fixed_tl
27125 }

```

(End definition for `\__fp_pow_npos_aux:NNnww`.)

`\__fp_pow_neg:www`  
`\__fp_pow_neg_aux:wNN`

This function is followed by three floating point numbers:  $a^b$ ,  $a \in [-\infty, -0]$ , and  $b$ . If  $b$  is an even integer (case  $-1$ ),  $a^b = a^b$ . If  $b$  is an odd integer (case  $0$ ),  $a^b = -a^b$ , obtained by a call to `\__fp_pow_neg_aux:wNN`. Otherwise, the sign is undefined. This is invalid, unless  $a^b$  turns out to be  $+0$  or `nan`, in which case we return that as  $a^b$ . In particular, since the

underflow detection occurs before `\_fp_pow_neg:www` is called,  $(-0.1)^{**}(12345.67)$  gives +0 rather than complaining that the sign is not defined.

```

27126 \cs_new:Npn \_fp_pow_neg:www \s_fp \_fp_chk:w #1#2; #3; #4;
27127 {
27128     \if_case:w \_fp_pow_neg_case:w #4 ;
27129         \exp_after:wN \_fp_pow_neg_aux:wNN
27130     \or:
27131         \if_int_compare:w \_fp_int_eval:w #1 / 2 = \c_one_int
27132             \_fp_invalid_operation_o:Nww ^ #3; #4;
27133             \exp:w \exp_end_continue_f:w
27134             \exp_after:wN \exp_after:wN
27135                 \exp_after:wN \_fp_use_none_until_s:w
27136             \fi:
27137         \fi:
27138         \_fp_exp_after_o:w
27139         \s_fp \_fp_chk:w #1#2;
27140     }
27141 \cs_new:Npn \_fp_pow_neg_aux:wNN #1 \s_fp \_fp_chk:w #2#3
27142 {
27143     \exp_after:wN \_fp_exp_after_o:w
27144     \exp_after:wN \s_fp
27145     \exp_after:wN \_fp_chk:w
27146     \exp_after:wN #2
27147     \int_value:w \_fp_int_eval:w 2 - #3 \_fp_int_eval_end:
27148 }
```

(End definition for `\_fp_pow_neg:www` and `\_fp_pow_neg_aux:wNN`.)

`\_fp_pow_neg_case:w`  
`\_fp_pow_neg_case_aux:nnnnn`  
`\_fp_pow_neg_case_aux:Nnnw`

This function expects a floating point number, and determines its “parity”. It should be used after `\if_case:w` or in an integer expression. It gives -1 if the number is an even integer, 0 if the number is an odd integer, and 1 otherwise. Zeros and  $\pm\infty$  are even (because very large finite floating points are even), while `nan` is a non-integer. The sign of normal numbers is irrelevant to parity. After `\_fp_decimate:nNnnnn` the argument #1 of `\_fp_pow_neg_case_aux:Nnnw` is a rounding digit, 0 if and only if the number was an integer, and #3 is the 8 least significant digits of that integer.

```

27149 \cs_new:Npn \_fp_pow_neg_case:w \s_fp \_fp_chk:w #1#2#3;
27150 {
27151     \if_case:w #1 \exp_stop_f:
27152         -1
27153     \or: \_fp_pow_neg_case_aux:nnnnn #3
27154     \or: -1
27155     \else: 1
27156     \fi:
27157     \exp_stop_f:
27158 }
27159 \cs_new:Npn \_fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
27160 {
27161     \if_int_compare:w #1 > \c_fp_prec_int
27162         -1
27163     \else:
27164         \_fp_decimate:nNnnnn { \c_fp_prec_int - #1 }
27165         \_fp_pow_neg_case_aux:Nnnw
27166             {#2} {#3} {#4} {#5}
```

```

27167      \fi:
27168  }
27169 \cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
27170  {
27171      \if_meaning:w 0 #1
27172          \if_int_odd:w #3 \exp_stop_f:
27173              0
27174          \else:
27175              -1
27176          \fi:
27177      \else:
27178          1
27179      \fi:
27180  }

(End definition for \__fp_pow_neg_case:w, \__fp_pow_neg_case_aux:nnnn, and \__fp_pow_neg_case_aux:Nnnw.)

```

## 74.4 Factorial

\c\_fp\_fact\_max\_arg\_int The maximum integer whose factorial fits in the exponent range is 3248, as  $3249! \sim 10^{10000.8}$

```
27181 \int_const:Nn \c_fp_fact_max_arg_int { 3248 }
```

(End definition for \c\_fp\_fact\_max\_arg\_int.)

\\_\_fp\_fact\_o:w First detect  $\pm 0$  and  $+\infty$  and `nan`. Then note that factorial of anything with a negative sign (except  $-0$ ) is undefined. Then call \\_\_fp\_small\_int:wTF to get an integer as the argument, and start a loop. This is not the most efficient way of computing the factorial, but it works all right. Of course we work with 24 digits instead of 16. It is easy to check that computing factorials with this precision is enough.

```

27182 \cs_new:Npn \__fp_fact_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
27183  {
27184      \if_case:w #2 \exp_stop_f:
27185          \__fp_case_return_o:Nw \c_one_fp
27186      \or:
27187      \or:
27188          \if_meaning:w 0 #3
27189              \exp_after:wN \__fp_case_return_same_o:w
27190          \fi:
27191      \or:
27192          \__fp_case_return_same_o:w
27193      \fi:
27194      \if_meaning:w 2 #3
27195          \__fp_case_use:nw { \__fp_invalid_operation_o:fw { fact } }
27196      \fi:
27197      \__fp_fact_pos_o:w
27198      \s_fp \__fp_chk:w #2 #3 #4 ;
27199  }

(End definition for \__fp_fact_o:w.)

```

```
\_fp_fact_pos_o:w  
\_fp_fact_int_o:w
```

Then check the input is an integer, and call `\_fp_factorial_int_o:n` with that int as an argument. If it's too big the factorial overflows. Otherwise call `\_fp_sanitize:Nw` with a positive sign marker 0 and an integer expression that will mop up any exponent in the calculation.

```
27200 \cs_new:Npn \_fp_fact_pos_o:w #1;  
27201 {  
27202     \_fp_small_int:wTF #1;  
27203     { \_fp_fact_int_o:n }  
27204     { \_fp_invalid_operation_o:fw { fact } #1; }  
27205 }  
27206 \cs_new:Npn \_fp_fact_int_o:n #1  
27207 {  
27208     \if_int_compare:w #1 > \c_fp_fact_max_arg_int  
27209         \_fp_case_return:nw  
27210         {  
27211             \exp_after:wN \exp_after:wN \exp_after:wN \_fp_overflow:w  
27212             \exp_after:wN \c_inf_fp  
27213         }  
27214     \fi:  
27215     \exp_after:wN \_fp_sanitize:Nw  
27216     \exp_after:wN 0  
27217     \int_value:w \_fp_int_eval:w  
27218     \_fp_fact_loop_o:w #1 . 4 , { 1 } { } { } { } { } { } ;  
27219 }
```

(End definition for `\_fp_fact_pos_o:w` and `\_fp_fact_int_o:w`.)

```
\_fp_fact_loop_o:w
```

The loop receives an integer `#1` whose factorial we want to compute, which we progressively decrement, and the result so far as an extended-precision number `#2` in the form  $\langle\text{exponent}\rangle,\langle\text{mantissa}\rangle$ . The loop goes in steps of two because we compute `#1*#1-1` as an integer expression (it must fit since `#1` is at most 3248), then multiply with the result so far. We don't need to fill in most of the mantissa with zeros because `\_fp_ep_mul:wwwn` first normalizes the extended precision number to avoid loss of precision. When reaching a small enough number simply use a table of factorials less than  $10^8$ . This limit is chosen because the normalization step cannot deal with larger integers.

```
27220 \cs_new:Npn \_fp_fact_loop_o:w #1 . #2 ;  
27221 {  
27222     \if_int_compare:w #1 < 12 \exp_stop_f:  
27223         \_fp_fact_small_o:w #1  
27224     \fi:  
27225     \exp_after:wN \_fp_ep_mul:wwwn  
27226     \exp_after:wN 4 \exp_after:wN ,  
27227     \exp_after:wN { \int_value:w \_fp_int_eval:w #1 * (#1 - 1) }  
27228     { } { } { } { } { } ;  
27229     #2 ;  
27230     {  
27231         \exp_after:wN \_fp_fact_loop_o:w  
27232         \int_value:w \_fp_int_eval:w #1 - 2 .  
27233     }  
27234 }  
27235 \cs_new:Npn \_fp_fact_small_o:w #1 \fi: #2 ; #3 ; #4  
27236 {  
27237     \fi:
```

```

27238   \exp_after:wN \_fp_ep_mul:wwwn
27239   \exp_after:wN 4 \exp_after:wN ,
27240   \exp_after:wN
27241   {
27242     \int_value:w
27243     \if_case:w #1 \exp_stop_f:
27244       1 \or: 1 \or: 2 \or: 6 \or: 24 \or: 120 \or: 720 \or: 5040
27245       \or: 40320 \or: 362880 \or: 3628800 \or: 39916800
27246       \fi:
27247     } { } { } { } { } { } ;
27248   #3 ;
27249   \_fp_ep_to_float_o:wwN 0
27250 }

```

(End definition for `\_fp_fact_loop_o:w.`)

```
27251 </package>
```

# Chapter 75

## I3fp-trig Implementation

```
27252  <*package>
27253  @@@=fp

\__fp_parse_word_acos:N
\__fp_parse_word_acosd:N
\__fp_parse_word_acsc:N
\__fp_parse_word_acscd:N
\__fp_parse_word_asec:N
\__fp_parse_word_aecd:N
\__fp_parse_word_asin:N
\__fp_parse_word_asind:N
\__fp_parse_word_cos:N
\__fp_parse_word_cosd:N
\__fp_parse_word_cot:N
\__fp_parse_word_cotd:N
\__fp_parse_word_csc:N
\__fp_parse_word_cscd:N
\__fp_parse_word_sec:N
\__fp_parse_word_secd:N
\__fp_parse_word_sin:N
\__fp_parse_word_sind:N
\__fp_parse_word_tan:N
\__fp_parse_word_tand:N

Unary functions.

27254 \tl_map_inline:nn
27255  {
27256    {acos} {acsc} {asec} {asin}
27257    {cos} {cot} {csc} {sec} {sin} {tan}
27258  }
27259  {
27260    \cs_new:cpx { __fp_parse_word_#1:N }
27261    {
27262      \exp_not:N \__fp_parse_unary_function:NNN
27263      \exp_not:c { __fp_#1_o:w }
27264      \exp_not:N \use_i:nn
27265    }
27266    \cs_new:cpx { __fp_parse_word_#1d:N }
27267    {
27268      \exp_not:N \__fp_parse_unary_function:NNN
27269      \exp_not:c { __fp_#1_o:w }
27270      \exp_not:N \use_ii:nn
27271    }
27272  }

(End definition for \__fp_parse_word_acos:N and others.)
```

Those functions may receive a variable number of arguments.

```
27273 \cs_new:Npn \__fp_parse_word_acot:N
27274  { \__fp_parse_function>NNN \__fp_acot_o:Nw \use_i:nn }
27275 \cs_new:Npn \__fp_parse_word_atand:N
27276  { \__fp_parse_function>NNN \__fp_acot_o:Nw \use_ii:nn }
27277 \cs_new:Npn \__fp_parse_word_atan:N
27278  { \__fp_parse_function>NNN \__fp_atan_o:Nw \use_i:nn }
27279 \cs_new:Npn \__fp_parse_word_atand:N
27280  { \__fp_parse_function>NNN \__fp_atan_o:Nw \use_ii:nn }

(End definition for \__fp_parse_word_acot:N and others.)
```

## 75.1 Direct trigonometric functions

The approach for all trigonometric functions (sine, cosine, tangent, cotangent, cosecant, and secant), with arguments given in radians or in degrees, is the same.

- Filter out special cases ( $\pm 0$ ,  $\pm \infty$  and `nan`).
- Keep the sign for later, and work with the absolute value  $|x|$  of the argument.
- Small numbers ( $|x| < 1$  in radians,  $|x| < 10$  in degrees) are converted to fixed point numbers (and to radians if  $|x|$  is in degrees).
- For larger numbers, we need argument reduction. Subtract a multiple of  $\pi/2$  (in degrees, 90) to bring the number to the range to  $[0, \pi/2)$  (in degrees,  $[0, 90)$ ).
- Reduce further to  $[0, \pi/4]$  (in degrees,  $[0, 45]$ ) using  $\sin x = \cos(\pi/2 - x)$ , and when working in degrees, convert to radians.
- Use the appropriate power series depending on the octant  $\lfloor \frac{x}{\pi/4} \rfloor \bmod 8$  (in degrees, the same formula with  $\pi/4 \rightarrow 45$ ), the sign, and the function to compute.

### 75.1.1 Filtering special cases

`\_fp_sin_o:w`

This function, and its analogs for `cos`, `csc`, `sec`, `tan`, and `cot` instead of `sin`, are followed either by `\use_i:nn` and a float in radians or by `\use_ii:nn` and a float in degrees. The sine of  $\pm 0$  or `nan` is the same float. The sine of  $\pm \infty$  raises an invalid operation exception with the appropriate function name. Otherwise, call the `trig` function to perform argument reduction and if necessary convert the reduced argument to radians. Then, `\_fp_sin_series_o>NNwww` is called to compute the Taylor series: this function receives a sign #3, an initial octant of 0, and the function `\_fp_ep_to_float_o:wwN` which converts the result of the series to a floating point directly rather than taking its inverse, since  $\sin(x) = \#3 \sin|x|$ .

```

27281 \cs_new:Npn \_fp_sin_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
27282   {
27283     \if_case:w #2 \exp_stop_f:
27284       \_fp_case_return_same_o:w
27285     \or: \_fp_case_use:nw
27286       {
27287         \_fp_trig:NNNNNwn #1 \_fp_sin_series_o:NNwww
27288           \_fp_ep_to_float_o:wwN #3 0
27289       }
27290     \or: \_fp_case_use:nw
27291       { \_fp_invalid_operation_o:fw { #1 { sin } { sind } } }
27292   \else: \_fp_case_return_same_o:w
27293   \fi:
27294   \s_fp \_fp_chk:w #2 #3 #4;
27295 }
```

(End definition for `\_fp_sin_o:w`.)

`\_fp_cos_o:w`

The cosine of  $\pm 0$  is 1. The cosine of  $\pm \infty$  raises an invalid operation exception. The cosine of `nan` is itself. Otherwise, the `trig` function reduces the argument to at most half a right-angle and converts if necessary to radians. We then call the same series as

for sine, but using a positive sign 0 regardless of the sign of  $x$ , and with an initial octant of 2, because  $\cos(x) = +\sin(\pi/2 + |x|)$ .

```

27296 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
27297   {
27298     \if_case:w #2 \exp_stop_f:
27299       \__fp_case_return_o:Nw \c_one_fp
27300     \or: \__fp_case_use:nw
27301       {
27302         \__fp_trig>NNNNwn #1 \__fp_sin_series_o>NNwwww
27303           \__fp_ep_to_float_o:wwN 0 2
27304       }
27305     \or: \__fp_case_use:nw
27306       { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
27307   \else: \__fp_case_return_same_o:w
27308   \fi:
27309   \s__fp \__fp_chk:w #2 #3;
27310 }
```

(End definition for `\__fp_cos_o:w`.)

`\__fp_csc_o:w` The cosecant of  $\pm 0$  is  $\pm\infty$  with the same sign, with a division by zero exception (see `\__fp_cot_zero_o:Nfw` defined below), which requires the function name. The cosecant of  $\pm\infty$  raises an invalid operation exception. The cosecant of `nan` is itself. Otherwise, the `trig` function performs the argument reduction, and converts if necessary to radians before calling the same series as for sine, using the sign `#3`, a starting octant of 0, and inverting during the conversion from the fixed point sine to the floating point result, because  $\csc(x) = \#3(\sin|x|)^{-1}$ .

```

27311 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
27312   {
27313     \if_case:w #2 \exp_stop_f:
27314       \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
27315     \or: \__fp_case_use:nw
27316       {
27317         \__fp_trig>NNNNwn #1 \__fp_sin_series_o>NNwwww
27318           \__fp_inv_to_float_o:wwN #3 0
27319       }
27320     \or: \__fp_case_use:nw
27321       { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
27322   \else: \__fp_case_return_same_o:w
27323   \fi:
27324   \s__fp \__fp_chk:w #2 #3 #4;
27325 }
```

(End definition for `\__fp_csc_o:w`.)

`\__fp_sec_o:w` The secant of  $\pm 0$  is 1. The secant of  $\pm\infty$  raises an invalid operation exception. The secant of `nan` is itself. Otherwise, the `trig` function reduces the argument and turns it to radians before calling the same series as for sine, using a positive sign 0, a starting octant of 2, and inverting upon conversion, because  $\sec(x) = +1/\sin(\pi/2 + |x|)$ .

```

27326 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
27327   {
27328     \if_case:w #2 \exp_stop_f:
27329       \__fp_case_return_o:Nw \c_one_fp
```

```

27330     \or:   \__fp_case_use:nw
27331     {
27332         \__fp_trig>NNNNNwn #1 \__fp_sin_series_o>NNwwww
27333             \__fp_ep_inv_to_float_o:wwN 0 2
27334         }
27335     \or:   \__fp_case_use:nw
27336         { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
27337     \else: \__fp_case_return_same_o:w
27338     \fi:
27339     \s_fp \__fp_chk:w #2 #3;
27340 }

```

(End definition for  $\_\_fp\_sec\_o:w$ .)

$\_\_fp\_tan\_o:w$  The tangent of  $\pm 0$  or  $\text{nan}$  is the same floating point number. The tangent of  $\pm\infty$  raises an invalid operation exception. Once more, the `trig` function does the argument reduction step and conversion to radians before calling  $\_\_fp\_tan\_series_o:NNwwww$ , with a sign #3 and an initial octant of 1 (this shift is somewhat arbitrary). See  $\_\_fp\_cot\_o:w$  for an explanation of the 0 argument.

```

27341 \cs_new:Npn \__fp_tan_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
27342 {
27343     \if_case:w #2 \exp_stop_f:
27344         \__fp_case_return_same_o:w
27345     \or:   \__fp_case_use:nw
27346         {
27347             \__fp_trig>NNNNNwn #1
27348                 \__fp_tan_series_o>NNwwww 0 #3 1
27349         }
27350     \or:   \__fp_case_use:nw
27351         { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
27352     \else: \__fp_case_return_same_o:w
27353     \fi:
27354     \s_fp \__fp_chk:w #2 #3 #4;
27355 }

```

(End definition for  $\_\_fp\_tan\_o:w$ .)

$\_\_fp\_cot\_o:w$  The cotangent of  $\pm 0$  is  $\pm\infty$  with the same sign, with a division by zero exception (see  $\_\_fp\_cot\_zero_o:Nfw$ ). The cotangent of  $\pm\infty$  raises an invalid operation exception. The cotangent of  $\text{nan}$  is itself. We use  $\cot x = -\tan(\pi/2 + x)$ , and the initial octant for the tangent was chosen to be 1, so the octant here starts at 3. The change in sign is obtained by feeding  $\_\_fp\_tan\_series_o:NNwwww$  two signs rather than just the sign of the argument: the first of those indicates whether we compute tangent or cotangent. Those signs are eventually combined.

```

27356 \cs_new:Npn \__fp_cot_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
27357 {
27358     \if_case:w #2 \exp_stop_f:
27359         \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
27360     \or:   \__fp_case_use:nw
27361         {
27362             \__fp_trig>NNNNNwn #1
27363                 \__fp_tan_series_o>NNwwww 2 #3 3
27364         }
27365     \or:   \__fp_case_use:nw

```

```

27366           { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
27367     \else: \__fp_case_return_same_o:w
27368     \fi:
27369     \s__fp \__fp_chk:w #2 #3 #4;
27370   }
27371 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
27372   {
27373     \fi:
27374     \token_if_eq_meaning:NNTF 0 #1
27375       { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
27376       { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
27377     {#2}
27378   }

```

(End definition for `\__fp_cot_o:w` and `\__fp_cot_zero_o:Nfw`.)

### 75.1.2 Distinguishing small and large arguments

`\__fp_trig:NNNNNwn` The first argument is `\use_i:nn` if the operand is in radians and `\use_ii:nn` if it is in degrees. Arguments #2 to #5 control what trigonometric function we compute, and #6 to #8 are pieces of a normal floating point number. Call the `_series` function #2, with arguments #3, either a conversion function (`\__fp_ep_to_float_o:wN` or `\__fp_ep_inv_to_float_o:wN`) or a sign 0 or 2 when computing tangent or cotangent; #4, a sign 0 or 2; the octant, computed in an integer expression starting with #5 and stopped by a period; and a fixed point number obtained from the floating point number by argument reduction (if necessary) and conversion to radians (if necessary). Any argument reduction adjusts the octant accordingly by leaving a (positive) shift into its integer expression. Let us explain the integer comparison. Two of the four `\exp_after:wN` are expanded, the expansion hits the test, which is true if the float is at least 1 when working in radians, and at least 10 when working in degrees. Then one of the remaining `\exp_after:wN` hits #1, which picks the `trig` or `trigd` function in whichever branch of the conditional was taken. The final `\exp_after:wN` closes the conditional. At the end of the day, a number is `large` if it is  $\geq 1$  in radians or  $\geq 10$  in degrees, and `small` otherwise. All four `trig/trigd` auxiliaries receive the operand as an extended-precision number.

```

27379 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
27380   {
27381     \exp_after:wN #2
27382     \exp_after:wN #3
27383     \exp_after:wN #4
27384     \int_value:w \__fp_int_eval:w #5
27385     \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
27386     \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
27387       #1 \__fp_trig_large:ww \__fp_trigd_large:ww
27388     \else:
27389       #1 \__fp_trig_small:ww \__fp_trigd_small:ww
27390     \fi:
27391     #7,#8{0000}{0000};
27392   }

```

(End definition for `\__fp_trig:NNNNNwn`.)

### 75.1.3 Small arguments

```
\_\_fp\_trig\_small:ww
```

This receives a small extended-precision number in radians and converts it to a fixed point number. Some trailing digits may be lost in the conversion, so we keep the original floating point number around: when computing sine or tangent (or their inverses), the last step is to multiply by the floating point number (as an extended-precision number) rather than the fixed point number. The period serves to end the integer expression for the octant.

```
27393 \cs_new:Npn \_\_fp\_trig\_small:ww #1,#2;
27394   { \_\_fp\_ep\_to\_fixed:wwn #1,#2; . #1,#2; }
```

(End definition for `\_\_fp\_trig\_small:ww`.)

```
\_\_fp\_trigd\_small:ww
```

Convert the extended-precision number to radians, then call `\_\_fp\_trig\_small:ww` to massage it in the form appropriate for the `_series` auxiliary.

```
27395 \cs_new:Npn \_\_fp\_trigd\_small:ww #1,#2;
27396   {
27397     \_\_fp\_ep\_mul\_raw:wwwnN
27398       -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
27399     \_\_fp\_trig\_small:ww
27400 }
```

(End definition for `\_\_fp\_trigd\_small:ww`.)

### 75.1.4 Argument reduction in degrees

Note that  $25 \times 360 = 9000$ , so  $10^{k+1} \equiv 10^k \pmod{360}$  for  $k \geq 3$ . When the exponent `#1` is very large, we can thus safely replace it by 22 (or even 19). We turn the floating point number into a fixed point number with two blocks of 8 digits followed by five blocks of 4 digits. The original float is  $100 \times \langle block_1 \rangle \cdots \langle block_3 \rangle \langle block_4 \rangle \cdots \langle block_7 \rangle$ , or is equal to it modulo 360 if the exponent `#1` is very large. The first auxiliary finds  $\langle block_1 \rangle + \langle block_2 \rangle \pmod{9}$ , a single digit, and prepends it to the 4 digits of  $\langle block_3 \rangle$ . It also unpacks  $\langle block_4 \rangle$  and grabs the 4 digits of  $\langle block_7 \rangle$ . The second auxiliary grabs the  $\langle block_3 \rangle$  plus any contribution from the first two blocks as `#1`, the first digit of  $\langle block_4 \rangle$  (just after the decimal point in hundreds of degrees) as `#2`, and the three other digits as `#3`. It finds the quotient and remainder of `#1#2` modulo 9, adds twice the quotient to the integer expression for the octant, and places the remainder (between 0 and 8) before `#3` to form a new  $\langle block_4 \rangle$ . The resulting fixed point number is  $x \in [0, 0.9]$ . If  $x \geq 0.45$ , we add 1 to the octant and feed  $0.9 - x$  with an exponent of 2 (to compensate the fact that we are working in units of hundreds of degrees rather than degrees) to `\_\_fp\_trigd\_small:ww`. Otherwise, we feed it  $x$  with an exponent of 2. The third auxiliary also discards digits which were not packed into the various  $\langle blocks \rangle$ . Since the original exponent `#1` is at least 2, those are all 0 and no precision is lost (`#6` and `#7` are four 0 each).

```
27401 \cs_new:Npn \_\_fp\_trigd\_large:ww #1, #2#3#4#5#6#7;
27402   {
27403     \exp_after:wN \_\_fp\_pack\_eight:wNNNNNNNN
27404     \exp_after:wN \_\_fp\_pack\_eight:wNNNNNNNN
27405     \exp_after:wN \_\_fp\_pack\_twice\_four:wNNNNNNNN
27406     \exp_after:wN \_\_fp\_pack\_twice\_four:wNNNNNNNN
27407     \exp_after:wN \_\_fp\_trigd\_large\_auxi:nnnwNNNN
27408     \exp_after:wN ;
27409     \exp:w \exp_end_continue_f:w
```

```

27410      \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
27411      #2#3#4#5#6#7 0000 0000 0000 !
27412  }
27413 \cs_new:Npn \_fp_trigd_large_auxi:nnnnwNNNN #1#2#3#4#5; #6#7#8#9
27414  {
27415      \exp_after:wN \_fp_trigd_large_auxii:wNw
27416      \int_value:w \_fp_int_eval:w #1 + #2
27417      - (#1 + #2 - 4) / 9 * 9 \_fp_int_eval_end:
27418      #3;
27419      #4; #5{#6#7#8#9};
27420  }
27421 \cs_new:Npn \_fp_trigd_large_auxii:wNw #1; #2#3;
27422  {
27423      + (#1#2 - 4) / 9 * 2
27424      \exp_after:wN \_fp_trigd_large_auxiii:www
27425      \int_value:w \_fp_int_eval:w #1#2
27426      - (#1#2 - 4) / 9 * 9 \_fp_int_eval_end: #3 ;
27427  }
27428 \cs_new:Npn \_fp_trigd_large_auxiii:www #1; #2; #3!
27429  {
27430      \if_int_compare:w #1 < 4500 \exp_stop_f:
27431      \exp_after:wN \_fp_use_i_until_s:nw
27432      \exp_after:wN \_fp_fixed_continue:wn
27433      \else:
27434      + 1
27435      \fi:
27436      \_fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
27437      {#1}#2{0000}{0000};
27438      { \_fp_trigd_small:ww 2, }
27439  }

```

(End definition for `\_fp_trigd_large:ww` and others.)

### 75.1.5 Argument reduction in radians

Arguments greater or equal to 1 need to be reduced to a range where we only need a few terms of the Taylor series. We reduce to the range  $[0, 2\pi]$  by subtracting multiples of  $2\pi$ , then to the smaller range  $[0, \pi/2]$  by subtracting multiples of  $\pi/2$  (keeping track of how many times  $\pi/2$  is subtracted), then to  $[0, \pi/4]$  by mapping  $x \rightarrow \pi/2 - x$  if appropriate. When the argument is very large, say,  $10^{100}$ , an equally large multiple of  $2\pi$  must be subtracted, hence we must work with a very good approximation of  $2\pi$  in order to get a sensible remainder modulo  $2\pi$ .

Specifically, we multiply the argument by an approximation of  $1/(2\pi)$  with 10048 digits, then discard the integer part of the result, keeping 52 digits of the fractional part. From the fractional part of  $x/(2\pi)$  we deduce the octant (quotient of the first three digits by 125). We then multiply by 8 or  $-8$  (the latter when the octant is odd), ignore any integer part (related to the octant), and convert the fractional part to an extended precision number, before multiplying by  $\pi/4$  to convert back to a value in radians in  $[0, \pi/4]$ .

It is possible to prove that given the precision of floating points and their range of exponents, the 52 digits may start at most with 24 zeros. The 5 last digits are affected by carries from computations which are not done, hence we are left with at least  $52 - 24 - 5 = 23$  significant digits, enough to round correctly up to  $0.6 \cdot \text{ulp}$  in all cases.

\c\_fp\_trig\_intarray

This integer array stores blocks of 8 decimals of  $10^{-16}/(2\pi)$ . Each entry is  $10^8$  plus an 8 digit number storing 8 decimals. In total we store 10112 decimals of  $10^{-16}/(2\pi)$ . The number of decimals we really need is the maximum exponent plus the number of digits we later need, 52, plus 12 (4 – 1 groups of 4 digits). The memory footprint (1/2 byte per digit) is the same as an earlier method of storing the data as a control sequence name, but the major advantage is that we can unpack specific subsets of the digits without unpacking the 10112 decimals.

```
27440 \intarray_const_from_clist:Nn \c_fp_trig_intarray
27441 {
27442     100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
27443     143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
27444     150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
27445     161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
27446     130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
27447     117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
27448     139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
27449     123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
27450     186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
27451     180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
27452     175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
27453     171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
27454     106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
27455     100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
27456     147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
27457     145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
27458     109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
27459     154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
27460     162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
27461     190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
27462     166895116, 162545705, 194332763, 112686500, 126122717, 197115321,
27463     112599504, 138667945, 103762556, 108363171, 116952597, 158128224,
27464     194162333, 143145106, 112353687, 185631136, 136692167, 114206974,
27465     169601292, 150578336, 105311960, 185945098, 139556718, 170995474,
27466     165104316, 123815517, 158083944, 129799709, 199505254, 138756612,
27467     194458833, 106846050, 178529151, 151410404, 189298850, 163881607,
27468     176196993, 107341038, 199957869, 118905980, 193737772, 106187543,
27469     122271893, 101366255, 126123878, 103875388, 181106814, 106765434,
27470     108282785, 126933426, 179955607, 107903860, 160352738, 199624512,
27471     159957492, 176297023, 159409558, 143011648, 129641185, 157771240,
27472     157544494, 157021789, 176979240, 194903272, 194770216, 164960356,
27473     153181535, 144003840, 168987471, 176915887, 163190966, 150696440,
27474     147769706, 187683656, 177810477, 197954503, 153395758, 130188183,
27475     186879377, 166124814, 195305996, 155802190, 183598751, 103512712,
27476     190432315, 180498719, 168687775, 194656634, 162210342, 104440855,
27477     149785037, 192738694, 129353661, 193778292, 187359378, 143470323,
27478     102371458, 137923557, 111863634, 119294601, 183182291, 196416500,
27479     187830793, 131353497, 179099745, 186492902, 167450609, 189368909,
27480     145883050, 133703053, 180547312, 132158094, 131976760, 132283131,
27481     141898097, 149822438, 133517435, 169898475, 101039500, 168388003,
27482     197867235, 199608024, 100273901, 108749548, 154787923, 156826113,
27483     199489032, 168997427, 108349611, 149208289, 103776784, 174303550,
27484     145684560, 183671479, 130845672, 133270354, 185392556, 120208683,
27485     193240995, 162211753, 131839402, 109707935, 170774965, 149880868,
```

27486	160663609, 168661967, 103747454, 121028312, 119251846, 122483499,
27487	111611495, 166556037, 196967613, 199312829, 196077608, 127799010,
27488	107830360, 102338272, 198790854, 102387615, 157445430, 192601191,
27489	100543379, 198389046, 154921248, 129516070, 172853005, 122721023,
27490	160175233, 113173179, 175931105, 103281551, 109373913, 163964530,
27491	157926071, 180083617, 195487672, 146459804, 173977292, 144810920,
27492	109371257, 186918332, 189588628, 139904358, 168666639, 175673445,
27493	114095036, 137327191, 174311388, 106638307, 125923027, 159734506,
27494	105482127, 178037065, 133778303, 121709877, 134966568, 149080032,
27495	169885067, 141791464, 168350828, 116168533, 114336160, 173099514,
27496	198531198, 119733758, 144420984, 116559541, 152250643, 139431286,
27497	144403838, 183561508, 179771645, 101706470, 167518774, 156059160,
27498	187168578, 157939226, 123475633, 117111329, 198655941, 159689071,
27499	198506887, 144230057, 151919770, 156900382, 118392562, 120338742,
27500	135362568, 108354156, 151729710, 188117217, 195936832, 156488518,
27501	174997487, 108553116, 159830610, 113921445, 144601614, 188452770,
27502	125114110, 170248521, 173974510, 138667364, 103872860, 109967489,
27503	131735618, 112071174, 104788993, 168886556, 192307848, 150230570,
27504	157144063, 163863202, 136852010, 174100574, 185922811, 115721968,
27505	100397824, 175953001, 166958522, 112303464, 118773650, 143546764,
27506	164565659, 171901123, 108476709, 193097085, 191283646, 166919177,
27507	169387914, 133315566, 150669813, 121641521, 100895711, 172862384,
27508	126070678, 145176011, 113450800, 169947684, 122356989, 162488051,
27509	157759809, 153397080, 185475059, 175362656, 149034394, 145420581,
27510	178864356, 183042000, 131509559, 147434392, 152544850, 167491429,
27511	108647514, 142303321, 133245695, 111634945, 167753939, 142403609,
27512	105438335, 152829243, 142203494, 184366151, 146632286, 102477666,
27513	166049531, 140657343, 157553014, 109082798, 180914786, 169343492,
27514	127376026, 134997829, 195701816, 119643212, 133140475, 176289748,
27515	140828911, 174097478, 126378991, 181699939, 148749771, 151989818,
27516	172666294, 160183053, 195832752, 109236350, 168538892, 128468247,
27517	125997252, 183007668, 156937583, 165972291, 198244297, 147406163,
27518	181831139, 158306744, 134851692, 185973832, 137392662, 140243450,
27519	119978099, 140402189, 161348342, 173613676, 144991382, 171541660,
27520	163424829, 136374185, 106122610, 186132119, 198633462, 184709941,
27521	183994274, 129559156, 128333990, 148038211, 175011612, 111667205,
27522	119125793, 103552929, 124113440, 131161341, 112495318, 138592695,
27523	184904438, 146807849, 109739828, 108855297, 104515305, 139914009,
27524	188698840, 188365483, 166522246, 168624087, 125401404, 100911787,
27525	142122045, 123075334, 173972538, 114940388, 141905868, 142311594,
27526	163227443, 139066125, 116239310, 162831953, 123883392, 113153455,
27527	163815117, 152035108, 174595582, 101123754, 135976815, 153401874,
27528	107394340, 136339780, 138817210, 104531691, 182951948, 179591767,
27529	139541778, 179243527, 161740724, 160593916, 102732282, 187946819,
27530	136491289, 149714953, 143255272, 135916592, 198072479, 198580612,
27531	169007332, 118844526, 179433504, 155801952, 149256630, 162048766,
27532	116134365, 133992028, 175452085, 155344144, 109905129, 182727454,
27533	165911813, 122232840, 151166615, 165070983, 175574337, 129548631,
27534	120411217, 116380915, 160616116, 157320000, 183306114, 160618128,
27535	103262586, 195951602, 146321661, 138576614, 180471993, 127077713,
27536	116441201, 159496011, 106328305, 120759583, 148503050, 179095584,
27537	198298218, 167402898, 138551383, 123957020, 180763975, 150429225,
27538	198476470, 171016426, 197438450, 143091658, 164528360, 132493360,
27539	143546572, 137557916, 113663241, 120457809, 196971566, 134022158,

27540	180545794, 131328278, 100552461, 132088901, 187421210, 192448910,
27541	141005215, 149680971, 113720754, 100571096, 134066431, 135745439,
27542	191597694, 135788920, 179342561, 177830222, 137011486, 142492523,
27543	192487287, 113132021, 176673607, 156645598, 127260957, 141566023,
27544	143787436, 129132109, 174858971, 150713073, 191040726, 143541417,
27545	197057222, 165479803, 181512759, 157912400, 125344680, 148220261,
27546	173422990, 101020483, 106246303, 137964746, 178190501, 181183037,
27547	151538028, 179523433, 141955021, 135689770, 191290561, 143178787,
27548	192086205, 174499925, 178975690, 118492103, 124206471, 138519113,
27549	188147564, 102097605, 154895793, 178514140, 141453051, 151583964,
27550	128232654, 106020603, 131189158, 165702720, 186250269, 191639375,
27551	115278873, 160608114, 155694842, 110322407, 177272742, 116513642,
27552	134366992, 171634030, 194053074, 180652685, 109301658, 192136921,
27553	141431293, 171341061, 157153714, 106203978, 147618426, 150297807,
27554	186062669, 169960809, 118422347, 163350477, 146719017, 145045144,
27555	161663828, 146208240, 186735951, 102371302, 190444377, 194085350,
27556	134454426, 133413062, 163074595, 113830310, 122931469, 134466832,
27557	185176632, 182415152, 110179422, 164439571, 181217170, 121756492,
27558	119644493, 196532222, 118765848, 182445119, 109401340, 150443213,
27559	198586286, 121083179, 139396084, 143898019, 114787389, 177233102,
27560	186310131, 148695521, 126205182, 178063494, 157118662, 177825659,
27561	188310053, 151552316, 165984394, 109022180, 163144545, 121212978,
27562	197344714, 188741258, 126822386, 102360271, 109981191, 152056882,
27563	134723983, 158013366, 106837863, 128867928, 161973236, 172536066,
27564	185216856, 132011948, 197807339, 158419190, 166595838, 167852941,
27565	124187182, 117279875, 106103946, 106481958, 157456200, 160892122,
27566	184163943, 173846549, 158993202, 184812364, 133466119, 170732430,
27567	195458590, 173361878, 162906318, 150165106, 126757685, 112163575,
27568	188696307, 145199922, 100107766, 176830946, 198149756, 122682434,
27569	179367131, 108412102, 119520899, 148191244, 140487511, 171059184,
27570	141399078, 189455775, 118462161, 190415309, 134543802, 180893862,
27571	180732375, 178615267, 179711433, 123241969, 185780563, 176301808,
27572	184386640, 160717536, 183213626, 129671224, 126094285, 140110963,
27573	121826276, 151201170, 122552929, 128965559, 146082049, 138409069,
27574	107606920, 103954646, 119164002, 115673360, 117909631, 187289199,
27575	186343410, 186903200, 157966371, 103128612, 135698881, 176403642,
27576	152540837, 109810814, 183519031, 121318624, 172281810, 150845123,
27577	169019064, 166322359, 138872454, 163073727, 128087898, 130041018,
27578	194859136, 173742589, 141812405, 167291912, 138003306, 134499821,
27579	196315803, 186381054, 124578934, 150084553, 128031351, 118843410,
27580	107373060, 159565443, 173624887, 171292628, 198074235, 139074061,
27581	178690578, 144431052, 174262641, 176783005, 182214864, 162289361,
27582	192966929, 192033046, 169332843, 181580535, 164864073, 118444059,
27583	195496893, 153773183, 167266131, 130108623, 158802128, 180432893,
27584	144562140, 147978945, 142337360, 158506327, 104399819, 132635916,
27585	168734194, 136567839, 101281912, 120281622, 195003330, 112236091,
27586	185875592, 101959081, 122415367, 194990954, 148881099, 175891989,
27587	108115811, 163538891, 163394029, 123722049, 184837522, 142362091,
27588	100834097, 156679171, 100841679, 157022331, 178971071, 102928884,
27589	189701309, 195339954, 124415335, 106062584, 139214524, 133864640,
27590	134324406, 157317477, 155340540, 144810061, 177612569, 108474646,
27591	114329765, 143900008, 138265211, 145210162, 136643111, 197987319,
27592	102751191, 144121361, 169620456, 193602633, 161023559, 162140467,
27593	102901215, 167964187, 135746835, 187317233, 110047459, 163339773,

27594	124770449, 118885134, 141536376, 100915375, 164267438, 145016622,
27595	113937193, 106748706, 128815954, 164819775, 119220771, 102367432,
27596	189062690, 170911791, 194127762, 112245117, 123546771, 115640433,
27597	135772061, 166615646, 174474627, 130562291, 133320309, 153340551,
27598	138417181, 194605321, 150142632, 180008795, 151813296, 175497284,
27599	167018836, 157425342, 150169942, 131069156, 134310662, 160434122,
27600	105213831, 158797111, 150754540, 163290657, 102484886, 148697402,
27601	187203725, 198692811, 149360627, 140384233, 128749423, 132178578,
27602	177507355, 171857043, 178737969, 134023369, 102911446, 196144864,
27603	197697194, 134527467, 144296030, 189437192, 154052665, 188907106,
27604	162062575, 150993037, 199766583, 167936112, 181374511, 104971506,
27605	115378374, 135795558, 167972129, 135876446, 130937572, 103221320,
27606	124605656, 161129971, 131027586, 191128460, 143251843, 143269155,
27607	129284585, 173495971, 150425653, 199302112, 118494723, 121323805,
27608	116549802, 190991967, 168151180, 122483192, 151273721, 199792134,
27609	133106764, 121874844, 126215985, 112167639, 167793529, 182985195,
27610	185453921, 106957880, 158685312, 132775454, 133229161, 198905318,
27611	190537253, 191582222, 192325972, 178133427, 181825606, 148823337,
27612	160719681, 101448145, 131983362, 137910767, 112550175, 128826351,
27613	183649210, 135725874, 110356573, 189469487, 154446940, 118175923,
27614	106093708, 128146501, 185742532, 149692127, 164624247, 183221076,
27615	154737505, 168198834, 156410354, 158027261, 125228550, 131543250,
27616	139591848, 191898263, 104987591, 115406321, 103542638, 190012837,
27617	142615518, 178773183, 175862355, 117537850, 169565995, 170028011,
27618	158412588, 170150030, 117025916, 174630208, 142412449, 112839238,
27619	105257725, 114737141, 123102301, 172563968, 130555358, 132628403,
27620	183638157, 168682846, 143304568, 105994018, 170010719, 152092970,
27621	117799058, 132164175, 179868116, 158654714, 177489647, 116547948,
27622	183121404, 131836079, 184431405, 157311793, 149677763, 173989893,
27623	102277656, 107058530, 140837477, 152640947, 143507039, 152145247,
27624	101683884, 107090870, 161471944, 137225650, 128231458, 172995869,
27625	173831689, 171268519, 139042297, 111072135, 107569780, 137262545,
27626	181410950, 138270388, 198736451, 162848201, 180468288, 120582913,
27627	153390138, 135649144, 130040157, 106509887, 192671541, 174507066,
27628	186888783, 143805558, 135011967, 145862340, 180595327, 124727843,
27629	182925939, 157715840, 136885940, 198993925, 152416883, 178793572,
27630	179679516, 154076673, 192703125, 164187609, 162190243, 104699348,
27631	159891990, 160012977, 174692145, 132970421, 167781726, 115178506,
27632	153008552, 155999794, 102099694, 155431545, 127458567, 104403686,
27633	168042864, 184045128, 181182309, 179349696, 127218364, 192935516,
27634	120298724, 169583299, 148193297, 183358034, 159023227, 105261254,
27635	121144370, 184359584, 194433836, 138388317, 175184116, 108817112,
27636	151279233, 137457721, 193398208, 119005406, 132929377, 175306906,
27637	160741530, 149976826, 147124407, 176881724, 186734216, 185881509,
27638	191334220, 175930947, 117385515, 193408089, 157124410, 163472089,
27639	131949128, 180783576, 131158294, 100549708, 191802336, 165960770,
27640	170927599, 101052702, 181508688, 197828549, 143403726, 142729262,
27641	110348701, 139928688, 153550062, 106151434, 130786653, 196085995,
27642	100587149, 139141652, 106530207, 100852656, 124074703, 166073660,
27643	153338052, 163766757, 120188394, 197277047, 122215363, 138511354,
27644	183463624, 161985542, 159938719, 133367482, 104220974, 149956672,
27645	170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
27646	134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
27647	168243130, 184279693, 171132070, 121049823, 123819574, 171759855,

```

27648    119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
27649    132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
27650    127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
27651    159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
27652    100064922, 112650013, 132686230, 121550837,
27653    }

```

(End definition for \c\_fp\_trig\_intarray.)

The exponent #1 is between 1 and 10000. We wish to look up decimals  $10^{\#1-16}/(2\pi)$  starting from the digit  $\#1 + 1$ . Since they are stored in batches of 8, compute  $\lceil \#1/8 \rceil$  and fetch blocks of 8 digits starting there. The numbering of items in \c\_fp\_trig\_intarray starts at 1, so the block  $\lceil \#1/8 \rceil + 1$  contains the digit we want, at one of the eight positions. Each call to \int\_value:w \kernel\_intarray\_item:Nn expands the next, until being stopped by \fp\_trig\_large\_auxiii:w using \exp\_stop\_f:. Once all these blocks are unpacked, the \exp\_stop\_f: and 0 to 7 digits are removed by \use\_none:n...n. Finally, \fp\_trig\_large\_auxii:w packs 64 digits (there are between 65 and 72 at this point) into groups of 4 and the auxv auxiliary is called.

```

27654 \cs_new:Npn \fp_trig_large:ww #1, #2#3#4#5#6;
27655 {
27656     \exp_after:wN \fp_trig_large_auxi:w
27657     \int_value:w \fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
27658     \int_value:w #1 , ;
27659     {#2}{#3}{#4}{#5} ;
27660 }
27661 \cs_new:Npn \fp_trig_large_auxi:w #1, #2,
27662 {
27663     \exp_after:wN \exp_after:wN
27664     \exp_after:wN \fp_trig_large_auxii:w
27665     \cs:w
27666         use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
27667         \exp_after:wN
27668     \cs_end:
27669     \int_value:w
27670     \kernel_intarray_item:Nn \c_fp_trig_intarray
27671         { \fp_int_eval:w #1 + 1 \scan_stop: }
27672     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
27673     \kernel_intarray_item:Nn \c_fp_trig_intarray
27674         { \fp_int_eval:w #1 + 2 \scan_stop: }
27675     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
27676     \kernel_intarray_item:Nn \c_fp_trig_intarray
27677         { \fp_int_eval:w #1 + 3 \scan_stop: }
27678     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
27679     \kernel_intarray_item:Nn \c_fp_trig_intarray
27680         { \fp_int_eval:w #1 + 4 \scan_stop: }
27681     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
27682     \kernel_intarray_item:Nn \c_fp_trig_intarray
27683         { \fp_int_eval:w #1 + 5 \scan_stop: }
27684     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
27685     \kernel_intarray_item:Nn \c_fp_trig_intarray
27686         { \fp_int_eval:w #1 + 6 \scan_stop: }
27687     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
27688     \kernel_intarray_item:Nn \c_fp_trig_intarray
27689         { \fp_int_eval:w #1 + 7 \scan_stop: }

```

```

27690     \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
27691     \_kernel_intarray_item:Nn \c_fp_trig_intarray
27692     { \_fp_int_eval:w #1 + 8 \scan_stop: }
27693     \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
27694     \_kernel_intarray_item:Nn \c_fp_trig_intarray
27695     { \_fp_int_eval:w #1 + 9 \scan_stop: }
27696     \exp_stop_f:
27697   }
27698 \cs_new:Npn \_fp_trig_large_auxii:w
27699   {
27700     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27701     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27702     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27703     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
27704     \_fp_trig_large_auxv:www ;
27705   }
27706 \cs_new:Npn \_fp_trig_large_auxiii:w 1 { \exp_stop_f: }

```

(End definition for `\_fp_trig_large:ww` and others.)

`\_fp_trig_large_auxv:www`  
`\_fp_trig_large_auxvi:wnnnnnnn`  
`\_fp_trig_large_pack:NNNNNw`

First come the first 64 digits of the fractional part of  $10^{#1-16}/(2\pi)$ , arranged in 16 blocks of 4, and ending with a semicolon. Then a few more digits of the same fractional part, ending with a semicolon, then 4 blocks of 4 digits holding the significand of the original argument. Multiply the 16-digit significand with the 64-digit fractional part: the `auxvi` auxiliary receives the significand as `#2#3#4#5` and 16 digits of the fractional part as `#6#7#8#9`, and computes one step of the usual ladder of `pack` functions we use for multiplication (see *e.g.*, `\_fp_fixed_mul:wwn`), then discards one block of the fractional part to set things up for the next step of the ladder. We perform 13 such steps, replacing the last `middle` shift by the appropriate `trailing` shift, then discard the significand and remaining 3 blocks from the fractional part, as there are not enough digits to compute any more step in the ladder. The last semicolon closes the ladder, and we return control to the `auxvii` auxiliary.

```

27707 \cs_new:Npn \_fp_trig_large_auxv:www #1; #2; #3;
27708   {
27709     \exp_after:wN \_fp_use_i_until_s:nw
27710     \exp_after:wN \_fp_trig_large_auxvii:w
27711     \int_value:w \_fp_int_eval:w \c_fp_leading_shift_int
27712     \prg_replicate:nn { 13 }
27713     { \_fp_trig_large_auxvi:wnnnnnnn }
27714     + \c_fp_trailing_shift_int - \c_fp_middle_shift_int
27715     \_fp_use_i_until_s:nw
27716     ; #3 #1 ; ;
27717   }
27718 \cs_new:Npn \_fp_trig_large_auxvi:wnnnnnnnn #1; #2#3#4#5#6#7#8#9
27719   {
27720     \exp_after:wN \_fp_trig_large_pack:NNNNNw
27721     \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
27722     + #2*#9 + #3*#8 + #4*#7 + #5*#6
27723     #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
27724   }
27725 \cs_new:Npn \_fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
27726   { + #1#2#3#4#5 ; #6 }

```

(End definition for `\_fp_trig_large_auxv:www`, `\_fp_trig_large_auxvi:wnnnnnnnn`, and `\_fp_trig_large_pack:NNNNNw`.)

```
\_fp_trig_large_auxvii:w
\_fp_trig_large_auxviii:w
\_fp_trig_large_auxix:Nw
\_fp_trig_large_auxxx:wNNNNN
\_fp_trig_large_auxxi:w
```

The `auxvii` auxiliary is followed by 52 digits and a semicolon. We find the octant as the integer part of 8 times what follows, or equivalently as the integer part of `#1#2#3/125`, and add it to the surrounding integer expression for the octant. We then compute 8 times the 52-digit number, with a minus sign if the octant is odd. Again, the last `middle` shift is converted to a `trailing` shift. Any integer part (including negative values which come up when the octant is odd) is discarded by `\_fp_use_i_until_s:nw`. The resulting fractional part should then be converted to radians by multiplying by  $2\pi/8$ , but first, build an extended precision number by abusing `\_fp_ep_to_ep_loop:N` with the appropriate trailing markers. Finally, `\_fp_trig_small:ww` sets up the argument for the functions which compute the Taylor series.

```
27727 \cs_new:Npn \_fp_trig_large_auxvii:w #1#2#3
27728 {
27729     \exp_after:wN \_fp_trig_large_auxviii:ww
27730     \int_value:w \_fp_int_eval:w (#1#2#3 - 62) / 125 ;
27731     #1#2#3
27732 }
27733 \cs_new:Npn \_fp_trig_large_auxviii:ww #1;
27734 {
27735     + #1
27736     \if_int_odd:w #1 \exp_stop_f:
27737         \exp_after:wN \_fp_trig_large_auxix:Nw
27738         \exp_after:wN -
27739     \else:
27740         \exp_after:wN \_fp_trig_large_auxix:Nw
27741         \exp_after:wN +
27742     \fi:
27743 }
27744 \cs_new:Npn \_fp_trig_large_auxix:Nw
27745 {
27746     \exp_after:wN \_fp_use_i_until_s:nw
27747     \exp_after:wN \_fp_trig_large_auxxi:w
27748     \int_value:w \_fp_int_eval:w \c_fp_leading_shift_int
27749     \prg_replicate:nn { 13 }
27750         { \_fp_trig_large_auxx:wNNNNN }
27751         + \c_fp_trailing_shift_int - \c_fp_middle_shift_int
27752     ;
27753 }
27754 \cs_new:Npn \_fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
27755 {
27756     \exp_after:wN \_fp_trig_large_pack:NNNNNw
27757     \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
27758     #2 8 * #3#4#5#6
27759     #1; #2
27760 }
27761 \cs_new:Npn \_fp_trig_large_auxxi:w #1;
27762 {
27763     \exp_after:wN \_fp_ep_mul_raw:wwwwN
27764     \int_value:w \_fp_int_eval:w 0 \_fp_ep_to_ep_loop:N #1 ; ;
27765     0,{7853}{9816}{3397}{4483}{0961}{5661};
27766     \_fp_trig_small:ww
```

27767 }

(End definition for `\_fp_trig_large_auxvii:w` and others.)

### 75.1.6 Computing the power series

Here we receive a conversion function `\_fp_ep_to_float_o:wwN` or `\_fp_ep_inv_to_float_o:wwN`, a *(sign)* (0 or 2), a (non-negative) *(octant)* delimited by a dot, a *(fixed point)* number delimited by a semicolon, and an extended-precision number. The auxiliary receives:

- the conversion function #1;
- the final sign, which depends on the octant #3 and the sign #2;
- the octant #3, which controls the series we use;
- the square #4 \* #4 of the argument as a fixed point number, computed with `\_fp_fixed_mul:wwn`;
- the number itself as an extended-precision number.

If the octant is in  $\{1, 2, 5, 6, \dots\}$ , we are near an extremum of the function and we use the series

$$\cos(x) = 1 - x^2 \left( \frac{1}{2!} - x^2 \left( \frac{1}{4!} - x^2 \left( \dots \right) \right) \right).$$

Otherwise, the series

$$\sin(x) = x \left( 1 - x^2 \left( \frac{1}{3!} - x^2 \left( \frac{1}{5!} - x^2 \left( \dots \right) \right) \right) \right)$$

is used. Finally, the extended-precision number is converted to a floating point number with the given sign, and `\_fp_sanitize:Nw` checks for overflow and underflow.

```
27768 \cs_new:Npn \_fp_sin_series_o:NNwww #1#2#3. #4;
27769 {
27770     \_fp_fixed_mul:wwn #4; #4;
27771     {
27772         \exp_after:wN \_fp_sin_series_aux_o>NNwww
27773         \exp_after:wN #1
27774         \int_value:w
27775             \if_int_odd:w \_fp_int_eval:w (#3 + 2) / 4 \_fp_int_eval_end:
27776                 #2
27777             \else:
27778                 \if_meaning:w #2 0 2 \else: 0 \fi:
27779             \fi:
2780             {#3}
2781     }
2782 }
2783 \cs_new:Npn \_fp_sin_series_aux_o:NNwww #1#2#3 #4; #5,#6;
2784 {
2785     \if_int_odd:w \_fp_int_eval:w #3 / 2 \_fp_int_eval_end:
2786         \exp_after:wN \use_i:nn
2787     \else:
2788         \exp_after:wN \use_ii:nn
```

```

27789 \fi:
27790 { % 1/18!
27791   \_\_fp\_fixed\_mul\_sub\_back:wwn    {0000}{0000}{0000}{0001}{5619}{2070};
27792                           #4;{0000}{0000}{0000}{0477}{9477}{3324};
27793   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
27794   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
27795   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
27796   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
27797   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0013}{8888}{8888}{8888}{8889};
27798   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
27799   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
27800   \_\_fp\_fixed\_mul\_sub\_back:wwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
27801   { \_\_fp\_fixed\_continue:wn 0, }
27802 }
27803 { % 1/17!
27804   \_\_fp\_fixed\_mul\_sub\_back:wwn    {0000}{0000}{0000}{0028}{1145}{7254};
27805                           #4;{0000}{0000}{0000}{7647}{1637}{3182};
27806   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
27807   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
27808   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
27809   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
27810   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
27811   \_\_fp\_fixed\_mul\_sub\_back:wwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
27812   \_\_fp\_fixed\_mul\_sub\_back:wwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
27813   { \_\_fp\_ep\_mul:wwnn 0, } #5,#6;
27814 }
27815 {
27816   \exp\_after:wN \_\_fp\_sanitize:Nw
27817   \exp\_after:wN #2
27818   \int\_value:w \_\_fp\_int\_eval:w #1
27819 }
27820 #2
27821 }

```

(End definition for `\_\_fp\_sin\_series_o:NNwww` and `\_\_fp\_sin\_series\_aux_o:NNnwww`.)

`\_\_fp\_tan\_series_o:NNwww`  
`\_\_fp\_tan\_series\_aux_o:Nnwww`

Contrarily to `\_\_fp\_sin\_series_o:NNwww` which received a conversion auxiliary as #1, here, #1 is 0 for tangent and 2 for cotangent. Consider first the case of the tangent. The octant #3 starts at 1, which means that it is 1 or 2 for  $|x| \in [0, \pi/2]$ , it is 3 or 4 for  $|x| \in [\pi/2, \pi]$ , and so on: the intervals on which  $\tan|x| \geq 0$  coincide with those for which  $\lfloor (\#3 + 1)/2 \rfloor$  is odd. We also have to take into account the original sign of  $x$  to get the sign of the final result; it is straightforward to check that the first `\int\_value:w` expansion produces 0 for a positive final result, and 2 otherwise. A similar story holds for  $\cot(x)$ .

The auxiliary receives the sign, the octant, the square of the (reduced) input, and the (reduced) input (an extended-precision number) as arguments. It then computes the numerator and denominator of

$$\tan(x) \simeq \frac{x(1 - x^2(a_1 - x^2(a_2 - x^2(a_3 - x^2(a_4 - x^2a_5)))))}{1 - x^2(b_1 - x^2(b_2 - x^2(b_3 - x^2(b_4 - x^2b_5))))}.$$

The ratio is computed by `\_\_fp\_ep\_div:wwnn`, then converted to a floating point number. For octants #3 (really, quadrants) next to a pole of the functions, the fixed point

numerator and denominator are exchanged before computing the ratio. Note that this `\if_int_odd:w` test relies on the fact that the octant is at least 1.

```

27822 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
27823 {
27824   \__fp_fixed_mul:wwn #4; #4;
27825 {
27826   \exp_after:wN \__fp_tan_series_aux_o:Nnwww
27827   \int_value:w
27828   \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
27829   \exp_after:wN \reverse_if:N
27830   \fi:
27831   \if_meaning:w #1#2 2 \else: 0 \fi:
27832 {#3}
27833 }
27834 }
27835 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
27836 {
27837   \__fp_fixed_mul_sub_back:wwnn      {0000}{0000}{1527}{3493}{0856}{7059};
27838                               #3; {0000}{0159}{6080}{0274}{5257}{6472};
27839   \__fp_fixed_mul_sub_back:wwnn #3; {0002}{4571}{2320}{0157}{2558}{8481};
27840   \__fp_fixed_mul_sub_back:wwnn #3; {0115}{5830}{7533}{5397}{3168}{2147};
27841   \__fp_fixed_mul_sub_back:wwnn #3; {1929}{8245}{6140}{3508}{7719}{2982};
27842   \__fp_fixed_mul_sub_back:wwnn #3; {10000}{0000}{0000}{0000}{0000}{0000};
27843 { \__fp_ep_mul:wwwn 0, } #4,#5;
27844 {
27845   \__fp_fixed_mul_sub_back:wwnn      {0000}{0007}{0258}{0681}{9408}{4706};
27846                               #3,{0000}{2343}{7175}{1399}{6151}{7670};
27847   \__fp_fixed_mul_sub_back:wwnn #3; {0019}{2638}{4588}{9232}{8861}{3691};
27848   \__fp_fixed_mul_sub_back:wwnn #3; {0536}{6357}{0691}{4344}{6852}{4252};
27849   \__fp_fixed_mul_sub_back:wwnn #3; {5263}{1578}{9473}{6842}{1052}{6315};
27850   \__fp_fixed_mul_sub_back:wwnn#3; {10000}{0000}{0000}{0000}{0000}{0000};
27851 {
27852   \reverse_if:N \if_int_odd:w
27853   \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
27854   \exp_after:wN \__fp_reverse_args:Nww
27855   \fi:
27856   \__fp_ep_div:wwwn 0,
27857 }
27858 }
27859 {
27860   \exp_after:wN \__fp_sanitize:Nw
27861   \exp_after:wN #1
27862   \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
27863 }
27864 #1
27865 }

```

(End definition for `\__fp_tan_series_o:NNwww` and `\__fp_tan_series_aux_o:Nnwww`.)

## 75.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcosecant, and arcsecant) are based on a function often denoted `atan2`. This func-

tion is accessed directly by feeding two arguments to arctangent, and is defined by  $\text{atan}(y, x) = \text{atan}(y/x)$  for generic  $y$  and  $x$ . Its advantages over the conventional arctangent is that it takes values in  $[-\pi, \pi]$  rather than  $[-\pi/2, \pi/2]$ , and that it is better behaved in boundary cases. Other inverse trigonometric functions are expressed in terms of  $\text{atan}$  as

$$\text{acos } x = \text{atan}(\sqrt{1 - x^2}, x) \quad (5)$$

$$\text{asin } x = \text{atan}(x, \sqrt{1 - x^2}) \quad (6)$$

$$\text{asec } x = \text{atan}(\sqrt{x^2 - 1}, 1) \quad (7)$$

$$\text{acsc } x = \text{atan}(1, \sqrt{x^2 - 1}) \quad (8)$$

$$\text{atan } x = \text{atan}(x, 1) \quad (9)$$

$$\text{acot } x = \text{atan}(1, x). \quad (10)$$

Rather than introducing a new function, `atan2`, the arctangent function `atan` is overloaded: it can take one or two arguments. In the comments below, following many texts, we call the first argument  $y$  and the second  $x$ , because  $\text{atan}(y, x) = \text{atan}(y/x)$  is the angular coordinate of the point  $(x, y)$ .

As for direct trigonometric functions, the first step in computing  $\text{atan}(y, x)$  is argument reduction. The sign of  $y$  gives that of the result. We distinguish eight regions where the point  $(x, |y|)$  can lie, of angular size roughly  $\pi/8$ , characterized by their “octant”, between 0 and 7 included. In each region, we compute an arctangent as a Taylor series, then shift this arctangent by the appropriate multiple of  $\pi/4$  and sign to get the result. Here is a list of octants, and how we compute the arctangent (we assume  $y > 0$ : otherwise replace  $y$  by  $-y$  below):

0  $0 < |y| < 0.41421x$ , then  $\text{atan} \frac{|y|}{x}$  is given by a nicely convergent Taylor series;

1  $0 < 0.41421x < |y| < x$ , then  $\text{atan} \frac{|y|}{x} = \frac{\pi}{4} - \text{atan} \frac{x-|y|}{x+|y|}$ ;

2  $0 < 0.41421|y| < x < |y|$ , then  $\text{atan} \frac{|y|}{x} = \frac{\pi}{4} + \text{atan} \frac{-x+|y|}{x+|y|}$ ;

3  $0 < x < 0.41421|y|$ , then  $\text{atan} \frac{|y|}{x} = \frac{\pi}{2} - \text{atan} \frac{x}{|y|}$ ;

4  $0 < -x < 0.41421|y|$ , then  $\text{atan} \frac{|y|}{x} = \frac{\pi}{2} + \text{atan} \frac{-x}{|y|}$ ;

5  $0 < 0.41421|y| < -x < |y|$ , then  $\text{atan} \frac{|y|}{x} = \frac{3\pi}{4} - \text{atan} \frac{x+|y|}{-x+|y|}$ ;

6  $0 < -0.41421x < |y| < -x$ , then  $\text{atan} \frac{|y|}{x} = \frac{3\pi}{4} + \text{atan} \frac{-x-|y|}{-x+|y|}$ ;

7  $0 < |y| < -0.41421x$ , then  $\text{atan} \frac{|y|}{x} = \pi - \text{atan} \frac{|y|}{-x}$ .

In the following, we denote by  $z$  the ratio among  $|\frac{y}{x}|$ ,  $|\frac{x}{y}|$ ,  $|\frac{x+y}{x-y}|$ ,  $|\frac{x-y}{x+y}|$  which appears in the right-hand side above.

### 75.2.1 Arctangent and arccotangent

```
\_\_fp_atan_o:Nw      The parsing step manipulates atan and acot like min and max, reading in an array of
\_\_fp_acot_o:Nw      operands, but also leaves \use_i:nn or \use_ii:nn depending on whether the result
\_\_fp_atan_default:w
```

should be given in radians or in degrees. The helper `\_fp_parse_function_one_two:nw` checks that the operand is one or two floating point numbers (not tuples) and leaves its second argument or its tail accordingly (its first argument is used for error messages). More precisely if we are given a single floating point number `\_fp_atan_default:w` places `\c_one_fp` (expanded) after it; otherwise `\_fp_atan_default:w` is omitted by `\_fp_parse_function_one_two:nw`.

```

27866 \cs_new:Npn \_fp_atan_o:Nw #1
27867 {
27868     \_fp_parse_function_one_two:nw
27869     { #1 { atan } { atand } }
27870     { \_fp_atan_default:w \_fp_atanii_o:Nww #1 }
27871 }
27872 \cs_new:Npn \_fp_acot_o:Nw #1
27873 {
27874     \_fp_parse_function_one_two:nw
27875     { #1 { acot } { acotd } }
27876     { \_fp_atan_default:w \_fp_acotii_o:Nww #1 }
27877 }
27878 \cs_new:Npx \_fp_atan_default:w #1#2#3 @ { #1 #2 #3 \c_one_fp @ }

(End definition for \_fp_atan_o:Nw, \_fp_acot_o:Nw, and \_fp_atan_default:w.)
```

`\_fp_atanii_o:Nww` If either operand is `nan`, we return it. If both are normal, we call `\_fp_atan_normal_o>NNnwNnw`. If both are zero or both infinity, we call `\_fp_atan_inf_o>NNNw` with argument 2, leading to a result among  $\{\pm\pi/4, \pm 3\pi/4\}$  (in degrees,  $\{\pm 45, \pm 135\}$ ). Otherwise, one is much bigger than the other, and we call `\_fp_atan_inf_o>NNNw` with either an argument of 4, leading to the values  $\pm\pi/2$  (in degrees,  $\pm 90$ ), or 0, leading to  $\{\pm 0, \pm\pi\}$  (in degrees,  $\{\pm 0, \pm 180\}$ ). Since  $\text{acot}(x, y) = \text{atan}(y, x)$ , `\_fp_acotii_o:ww` simply reverses its two arguments.

```

27879 \cs_new:Npn \_fp_atanii_o:Nww
27880     #1 \s_fp \_fp_chk:w #2#3#4; \s_fp \_fp_chk:w #5 #6 @
27881 {
27882     \if_meaning:w 3 #2 \_fp_case_return_i_o:ww \fi:
27883     \if_meaning:w 3 #5 \_fp_case_return_ii_o:ww \fi:
27884     \if_case:w
27885         \if_meaning:w #2 #5
27886             \if_meaning:w 1 #2 10 \else: 0 \fi:
27887         \else:
27888             \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
27889         \fi:
27890         \exp_stop_f:
27891         \_fp_case_return:nw { \_fp_atan_inf_o>NNNw #1 #3 2 }
27892         \or: \_fp_case_return:nw { \_fp_atan_inf_o>NNNw #1 #3 4 }
27893         \or: \_fp_case_return:nw { \_fp_atan_inf_o>NNNw #1 #3 0 }
27894         \fi:
27895         \_fp_atan_normal_o>NNnwNnw #1
27896         \s_fp \_fp_chk:w #2#3#4;
27897         \s_fp \_fp_chk:w #5 #6
27898 }
27899 \cs_new:Npn \_fp_acotii_o:Nww #1#2; #3;
27900     { \_fp_atanii_o:Nww #1#3; #2; }
```

(End definition for \\_fp\_atanii\_o:Nww and \\_fp\_acotii\_o:Nww.)

\\_\\_fp\\_atan\\_inf\\_o:NNNw

This auxiliary is called whenever one number is  $\pm 0$  or  $\pm\infty$  (and neither is `nan`). Then the result only depends on the signs, and its value is a multiple of  $\pi/4$ . We use the same auxiliary as for normal numbers, `\_\_fp\_atan\_combine_o:NwwwN`, with arguments the final sign #2; the octant #3;  $\text{atan } z/z = 1$  as a fixed point number;  $z = 0$  as a fixed point number; and  $z = 0$  as an extended-precision number. Given the values we provide,  $\text{atan } z$  is computed to be 0, and the result is  $[\#3/2] \cdot \pi/4$  if the sign #5 of  $x$  is positive, and  $[(7 - \#3)/2] \cdot \pi/4$  for negative  $x$ , where the divisions are rounded up.

```
27901 \cs_new:Npn \_\_fp\_atan\_inf\_o:NNNw #1#2#3 \s\_fp \_\_fp\_chk:w #4#5#6;
27902 {
27903     \exp_after:wN \_\_fp\_atan\_combine_o:NwwwN
27904     \exp_after:wN #2
27905     \int_value:w \_\_fp\_int\_eval:w
27906         \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
27907         \c\_fp\_one\_fixed_tl
27908         {0000}{0000}{0000}{0000}{0000}{0000};
27909         0,{0000}{0000}{0000}{0000}{0000}{0000}; #1
27910 }
```

(End definition for `\_\_fp\_atan\_inf\_o:NNNw`.)

\\_\\_fp\\_atan\\_normal\\_o:NNnwNnw

Here we simply reorder the floating point data into a pair of signed extended-precision numbers, that is, a sign, an exponent ending with a comma, and a six-block mantissa ending with a semi-colon. This extended precision is required by other inverse trigonometric functions, to compute things like  $\text{atan}(x, \sqrt{1 - x^2})$  without intermediate rounding errors.

```
27911 \cs_new_protected:Npn \_\_fp\_atan\_normal\_o:NNnwNnw
27912     #1 \s\_fp \_\_fp\_chk:w 1#2#3#4; \s\_fp \_\_fp\_chk:w 1#5#6#7;
27913 {
27914     \_\_fp\_atan\_test_o:NwwNwwN
27915     #2 #3, #4{0000}{0000};
27916     #5 #6, #7{0000}{0000}; #1
27917 }
```

(End definition for `\_\_fp\_atan\_normal\_o:NNnwNnw`.)

\\_\\_fp\\_atan\\_test\\_o:NwwNwwN

This receives: the sign #1 of  $y$ , its exponent #2, its 24 digits #3 in groups of 4, and similarly for  $x$ . We prepare to call `\_\_fp\_atan\_combine_o:NwwwN` which expects the sign #1, the octant, the ratio  $(\text{atan } z)/z = 1 - \dots$ , and the value of  $z$ , both as a fixed point number and as an extended-precision floating point number with a mantissa in  $[0.01, 1)$ . For now, we place #1 as a first argument, and start an integer expression for the octant. The sign of  $x$  does not affect  $z$ , so we simply leave a contribution to the octant:  $\langle \text{octant} \rangle \rightarrow 7 - \langle \text{octant} \rangle$  for negative  $x$ . Then we order  $|y|$  and  $|x|$  in a non-decreasing order: if  $|y| > |x|$ , insert 3- in the expression for the octant, and swap the two numbers. The finer test with 0.41421 is done by `\_\_fp\_atan\_div:wnwwnw` after the operands have been ordered.

```
27918 \cs_new:Npn \_\_fp\_atan\_test_o:NwwNwwN #1#2,#3; #4#5,#6;
27919 {
27920     \exp_after:wN \_\_fp\_atan\_combine_o:NwwwN
27921     \exp_after:wN #1
27922     \int_value:w \_\_fp\_int\_eval:w
27923         \if_meaning:w 2 #4
27924             7 - \_\_fp\_int\_eval:w
27925             \fi:
```

```

27926   \if_int_compare:w
27927     \__fp_ep_compare:www #2,#3; #5,#6; > \c_zero_int
27928     3 -
27929     \exp_after:wN \__fp_reverse_args:Nww
27930   \fi:
27931   \__fp_atan_div:wnwwnw #2,#3; #5,#6;
27932 }

```

(End definition for `\__fp_atan_test_o:NwwNwwN.`)

```

\__fp_atan_div:wnwwnw
\__fp_atan_near:wwwn
\__fp_atan_near_aux:wwn

```

This receives two positive numbers  $a$  and  $b$  (equal to  $|x|$  and  $|y|$  in some order), each as an exponent and 6 blocks of 4 digits, such that  $0 < a < b$ . If  $0.41421b < a$ , the two numbers are “near”, hence the point  $(y, x)$  that we started with is closer to the diagonals  $\{|y| = |x|\}$  than to the axes  $\{xy = 0\}$ . In that case, the octant is 1 (possibly combined with the 7– and 3– inserted earlier) and we wish to compute  $\text{atan} \frac{b-a}{a+b}$ . Otherwise, the octant is 0 (again, combined with earlier terms) and we wish to compute  $\text{atan} \frac{a}{b}$ . In any case, call `\__fp_atan_auxi:ww` followed by  $z$ , as a comma-delimited exponent and a fixed point number.

```

27933 \cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
27934 {
27935   \if_int_compare:w
27936     \__fp_int_eval:w 41421 * #5 < #2 000
27937     \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
27938       00 \or: 0 \fi:
27939     \exp_stop_f:
27940     \exp_after:wN \__fp_atan_near:wwwn
27941   \fi:
27942   0
27943   \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
27944   \__fp_atan_auxi:ww
27945 }
27946 \cs_new:Npn \__fp_atan_near:wwwn
27947   0 \__fp_ep_div:wwwn #1,#2; #3,
27948 {
27949   1
27950   \__fp_ep_to_fixed:wwn #1 - #3, #2;
27951   \__fp_atan_near_aux:wwn
27952 }
27953 \cs_new:Npn \__fp_atan_near_aux:wwn #1; #2;
27954 {
27955   \__fp_fixed_add:wwn #1; #2;
27956   { \__fp_fixed_sub:wwn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
27957 }

```

(End definition for `\__fp_atan_div:wnwwnw`, `\__fp_atan_near:wwwn`, and `\__fp_atan_near_aux:wwn`.)

```

\__fp_atan_auxi:ww
\__fp_atan_auxii:ww

```

Convert  $z$  from a representation as an exponent and a fixed point number in  $[0.01, 1)$  to a fixed point number only, then set up the call to `\__fp_atan_Taylor_loop:www`, followed by the fixed point representation of  $z$  and the old representation.

```

27958 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
27959   { \__fp_ep_to_fixed:wwn #1,#2; \__fp_atan_auxii:w #1,#2; }
27960 \cs_new:Npn \__fp_atan_auxii:w #1;
27961 {
27962   \__fp_fixed_mul:wwn #1; #1;

```

```

27963      {
27964          \_fp_atan_Taylor_loop:www 39 ;
27965          {0000}{0000}{0000}{0000}{0000} ;
27966      }
27967      ! #1;
27968  }

```

(End definition for `\_fp_atan_auxi:ww` and `\_fp_atan_auxi:w`)

We compute the series of  $(\tan z)/z$ . A typical intermediate stage has  $\#1 = 2k - 1$ ,  $\#2 = \frac{1}{2k+1} - z^2(\frac{1}{2k+3} - z^2(\dots - z^2 \frac{1}{39}))$ , and  $\#3 = z^2$ . To go to the next step  $k \rightarrow k - 1$ , we compute  $\frac{1}{2k-1}$ , then subtract from it  $z^2$  times  $\#2$ . The loop stops when  $k = 0$ : then  $\#2$  is  $(\tan z)/z$ , and there is a need to clean up all the unnecessary data, end the integer expression computing the octant with a semicolon, and leave the result  $\#2$  afterwards.

```

27969 \cs_new:Npn \_fp_atan_Taylor_loop:www #1; #2; #3;
27970  {
27971      \if_int_compare:w #1 = - \c_one_int
27972          \_fp_atan_Taylor_break:w
27973      \fi:
27974      \exp_after:wN \_fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
27975      \_fp_rrrot:www \_fp_fixed_mul_sub_back:wwn #2; #3;
27976      {
27977          \exp_after:wN \_fp_atan_Taylor_loop:www
27978          \int_value:w \_fp_int_eval:w #1 - 2 ;
27979      }
27980      #3;
27981  }
27982 \cs_new:Npn \_fp_atan_Taylor_break:w
27983  \fi: #1 \_fp_fixed_mul_sub_back:wwn #2; #3 !
27984  { \fi: ; #2 ; }

```

(End definition for `\_fp_atan_Taylor_loop:www` and `\_fp_atan_Taylor_break:w`)

`\_fp_atan_combine_o:NwwwwN`  
`\_fp_atan_combine_aux:ww`

This receives a  $\langle sign \rangle$ , an  $\langle octant \rangle$ , a fixed point value of  $(\tan z)/z$ , a fixed point number  $z$ , and another representation of  $z$ , as an  $\langle exponent \rangle$  and the fixed point number  $10^{-\langle exponent \rangle} z$ , followed by either `\use_i:nn` (when working in radians) or `\use_ii:nn` (when working in degrees). The function computes the floating point result

$$\langle sign \rangle \left( \left\lceil \frac{\langle octant \rangle}{2} \right\rceil \frac{\pi}{4} + (-1)^{\langle octant \rangle} \frac{\tan z}{z} \cdot z \right), \quad (11)$$

multiplied by  $180/\pi$  if working in degrees, and using in any case the most appropriate representation of  $z$ . The floating point result is passed to `\_fp_sanitize:Nw`, which checks for overflow or underflow. If the octant is 0, leave the exponent  $\#5$  for `\_fp_sanitize:Nw`, and multiply  $\#3 = \frac{\tan z}{z}$  with  $\#6$ , the adjusted  $z$ . Otherwise, multiply  $\#3 = \frac{\tan z}{z}$  with  $\#4 = z$ , then compute the appropriate multiple of  $\frac{\pi}{4}$  and add or subtract the product  $\#3 \cdot \#4$ . In both cases, convert to a floating point with `\_fp_fixed_to_float_o:wN`.

```

27985 \cs_new:Npn \_fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5, #6; #7
27986  {
27987      \exp_after:wN \_fp_sanitize:Nw
27988      \exp_after:wN #1
27989      \int_value:w \_fp_int_eval:w

```

```

27990   \if_meaning:w 0 #2
27991     \exp_after:wN \use_i:nn
27992   \else:
27993     \exp_after:wN \use_ii:nn
27994   \fi:
27995   { #5 \__fp_fixed_mul:wwn #3; #6; }
27996   {
27997     \__fp_fixed_mul:wwn #3; #4;
27998   {
27999     \exp_after:wN \__fp_atan_combine_aux:ww
28000     \int_value:w \__fp_int_eval:w #2 / 2 ; #2;
28001   }
28002   }
28003   { #7 \__fp_fixed_to_float_o:wn \__fp_fixed_to_float_rad_o:wn }
28004   #1
28005   }
28006 \cs_new:Npn \__fp_atan_combine_aux:ww #1; #2;
28007   {
28008     \__fp_fixed_short:wwn
28009     {7853}{9816}{3397}{4483}{0961}{5661};
28010     {#1}{0000}{0000};
28011   {
28012     \if_int_odd:w #2 \exp_stop_f:
28013       \exp_after:wN \__fp_fixed_sub:wwn
28014     \else:
28015       \exp_after:wN \__fp_fixed_add:wwn
28016     \fi:
28017   }
28018 }

```

(End definition for `\__fp_atan_combine_o:NwwwwN` and `\__fp_atan_combine_aux:ww`.)

### 75.2.2 Arcsine and arccosine

`\__fp_asin_o:w` Again, the first argument provided by l3fp-parse is `\use_i:nn` if we are to work in radians and `\use_ii:nn` for degrees. Then comes a floating point number. The arcsine of  $\pm 0$  or `nan` is the same floating point number. The arcsine of  $\pm\infty$  raises an invalid operation exception. Otherwise, call an auxiliary common with `\__fp_acos_o:w`, feeding it information about what function is being performed (for “invalid operation” exceptions).

```

28019 \cs_new:Npn \__fp_asin_o:w #1 \s__fp \__fp_chk:w #2#3; @
28020   {
28021     \if_case:w #2 \exp_stop_f:
28022       \__fp_case_return_same_o:w
28023     \or:
28024       \__fp_case_use:nw
28025         { \__fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
28026     \or:
28027       \__fp_case_use:nw
28028         { \__fp_invalid_operation_o:fw { #1 { asin } { asind } } }
28029     \else:
28030       \__fp_case_return_same_o:w
28031     \fi:
28032     \s__fp \__fp_chk:w #2 #3;
28033   }

```

(End definition for `\_fp_asin_o:w`.)

`\_fp_acos_o:w` The arccosine of  $\pm 0$  is  $\pi/2$  (in degrees, 90). The arccosine of  $\pm\infty$  raises an invalid operation exception. The arccosine of `nan` is itself. Otherwise, call an auxiliary common with `\_fp_sin_o:w`, informing it that it was called by `acos` or `acosd`, and preparing to swap some arguments down the line.

```

28034 \cs_new:Npn \_fp_acos_o:w #1 \s_fp \_fp_chk:w #2#3; @
28035   {
28036     \if_case:w #2 \exp_stop_f:
28037       \_fp_case_use:nw { \_fp_atan_inf_o:NNNw #1 0 4 }
28038     \or:
28039       \_fp_case_use:nw
28040         {
28041           \_fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
28042             \_fp_reverse_args:Nww
28043         }
28044     \or:
28045       \_fp_case_use:nw
28046         { \_fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
28047     \else:
28048       \_fp_case_return_same_o:w
28049     \fi:
28050     \s_fp \_fp_chk:w #2 #3;
28051   }

```

(End definition for `\_fp_acos_o:w`.)

`\_fp_asin_normal_o:NfwNnnnnw` If the exponent #5 is at most 0, the operand lies within  $(-1, 1)$  and the operation is permitted: call `\_fp_asin_auxi_o:NnNww` with the appropriate arguments. If the number is exactly  $\pm 1$  (the test works because we know that  $\#5 \geq 1$ ,  $\#6\#7 \geq 10000000$ ,  $\#8\#9 \geq 0$ , with equality only for  $\pm 1$ ), we also call `\_fp_asin_auxi_o:NnNww`. Otherwise, `\_fp_use_i:ww` gets rid of the `asin` auxiliary, and raises instead an invalid operation, because the operand is outside the domain of arcsine or arccosine.

```

28052 \cs_new:Npn \_fp_asin_normal_o:NfwNnnnnw
28053   #1#2#3 \s_fp \_fp_chk:w 1#4#5#6#7#8#9;
28054   {
28055     \if_int_compare:w #5 < \c_one_int
28056       \exp_after:wN \_fp_use_none_until_s:w
28057     \fi:
28058     \if_int_compare:w \_fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
28059       \exp_after:wN \_fp_use_none_until_s:w
28060     \fi:
28061     \_fp_use_i:ww
28062     \_fp_invalid_operation_o:fw {#2}
28063       \s_fp \_fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
28064     \_fp_asin_auxi_o:NnNww
28065       #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
28066   }

```

(End definition for `\_fp_asin_normal_o:NfwNnnnnw`.)

`\_fp_asin_auxi_o:NnNww` We compute  $x/\sqrt{1-x^2}$ . This function is used by `asin` and `acos`, but also by `acsc` and `asec` after inverting the operand, thus it must manipulate extended-precision numbers. First evaluate  $1-x^2$  as  $(1+x)(1-x)$ : this behaves better near  $x=1$ . We do the

addition/subtraction with fixed point numbers (they are not implemented for extended-precision floats), but go back to extended-precision floats to multiply and compute the inverse square root  $1/\sqrt{1 - x^2}$ . Finally, multiply by the (positive) extended-precision float  $|x|$ , and feed the (signed) result, and the number +1, as arguments to the arctangent function. When computing the arccosine, the arguments  $x/\sqrt{1 - x^2}$  and +1 are swapped by #2 (`\_fp_reverse_args:Nww` in that case) before `\_fp_atan_test_o:NwwNwwN` is evaluated. Note that the arctangent function requires normalized arguments, hence the need for `ep_to_ep` and `continue` after `ep_mul`.

```

28067 \cs_new:Npn \_fp_asin_auxi_o:NnNww #1#2#3#4,#5;
28068 {
28069     \_fp_ep_to_fixed:wwn #4,#5;
28070     \_fp_asin_isqrt:wn
28071     \_fp_ep_mul:wwwn #4,#5;
28072     \_fp_ep_to_ep:wwN
28073     \_fp_fixed_continue:wn
28074     { #2 \_fp_atan_test_o:NwwNwwN #3 }
28075     0 1,{1000}{0000}{0000}{0000}{0000}; #1
28076 }
28077 \cs_new:Npn \_fp_asin_isqrt:wn #1;
28078 {
28079     \exp_after:wN \_fp_fixed_sub:wwn \c_fp_one_fixed_tl #1;
28080     {
28081         \_fp_fixed_add_one:wN #1;
28082         \_fp_fixed_continue:wn { \_fp_ep_mul:wwwn 0, } 0,
28083     }
28084     \_fp_ep_isqrt:wwn
28085 }
```

(End definition for `\_fp_asin_auxi_o:NnNww` and `\_fp_asin_isqrt:wn`.)

### 75.2.3 Arcosecant and arcsecant

`\_fp_acsc_o:w`

Cases are mostly labelled by #2, except when #2 is 2: then we use #3#2, which is 02 = 2 when the number is  $+\infty$  and 22 when the number is  $-\infty$ . The arcosecant of  $\pm 0$  raises an invalid operation exception. The arcosecant of  $\pm\infty$  is  $\pm 0$  with the same sign. The arcosecant of `nan` is itself. Otherwise, `\_fp_acsc_normal_o:NfwNnw` does some more tests, keeping the function name (`acsc` or `acscl`) as an argument for invalid operation exceptions.

```

28086 \cs_new:Npn \_fp_acsc_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
28087 {
28088     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
28089         \_fp_case_use:nw
28090         { \_fp_invalid_operation_o:fw { #1 { acsc } { acscd } } }
28091     \or: \_fp_case_use:nw
28092         { \_fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscd } } }
28093     \or: \_fp_case_return_o:Nw \c_zero_fp
28094     \or: \_fp_case_return_same_o:w
28095     \else: \_fp_case_return_o:Nw \c_minus_zero_fp
28096     \fi:
28097     \s_fp \_fp_chk:w #2 #3 #4;
28098 }
```

(End definition for `\_fp_acsc_o:w`.)

`\_fp_asec_o:w` The arcsecant of  $\pm 0$  raises an invalid operation exception. The arcsecant of  $\pm\infty$  is  $\pi/2$  (in degrees, 90). The arcosecant of `nan` is itself. Otherwise, do some more tests, keeping the function name `asec` (or `asecd`) as an argument for invalid operation exceptions, and a `\_fp_reverse_args:Nww` following precisely that appearing in `\_fp_acos_o:w`.

```

28099 \cs_new:Npn \_fp_asec_o:w #1 \s_fp \_fp_chk:w #2#3; @
28100   {
28101     \if_case:w #2 \exp_stop_f:
28102       \_fp_case_use:nw
28103         { \_fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
28104     \or:
28105       \_fp_case_use:nw
28106         {
28107           \_fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
28108             \_fp_reverse_args:Nww
28109         }
28110     \or: \_fp_case_use:nw { \_fp_atan_inf_o:NNNw #1 0 4 }
28111     \else: \_fp_case_return_same_o:w
28112     \fi:
28113     \s_fp \_fp_chk:w #2 #3;
28114   }

```

(End definition for `\_fp_asec_o:w`.)

`\_fp_acsc_normal_o:NfwNnw` If the exponent is non-positive, the operand is less than 1 in absolute value, which is always an invalid operation: complain. Otherwise, compute the inverse of the operand, and feed it to `\_fp_asin_auxi_o:NnNww` (with all the appropriate arguments). This computes what we want thanks to  $\text{acsc}(x) = \text{asin}(1/x)$  and  $\text{asec}(x) = \text{acos}(1/x)$ .

```

28115 \cs_new:Npn \_fp_acsc_normal_o:NfwNnw #1#2#3 \s_fp \_fp_chk:w 1#4#5#6;
28116   {
28117     \int_compare:nNnTF {#5} < 1
28118       {
28119         \_fp_invalid_operation_o:fw {#2}
28120           \s_fp \_fp_chk:w 1#4{#5}#6;
28121       }
28122       {
28123         \_fp_ep_div:wwwwn
28124           1,{1000}{0000}{0000}{0000}{0000}{0000};
28125           #5,#6{0000}{0000};
28126           { \_fp_asin_auxi_o:NnNww #1 {#3} #4 }
28127       }
28128   }

```

(End definition for `\_fp_acsc_normal_o:NfwNnw`.)

28129 `</package>`

# Chapter 76

## 13fp-convert implementation

```
28130  (*package)
28131  (@@=fp)
```

### 76.1 Dealing with tuples

The first argument is for instance `\_fp_to_t1_dispatch:w`, which converts any floating point object to the appropriate representation. We loop through all items, putting `,`  
`~` between all of them and making sure to remove the leading `,`  
`~`.

```
28132  \cs_new:Npn \_fp_tuple_convert:Nw #1 \s_fp_tuple \_fp_tuple_chk:w #2 ;
28133  {
28134      \int_case:nnF { \_fp_array_count:n {#2} }
28135      {
28136          { 0 } { () }
28137          { 1 } { \_fp_tuple_convert_end:w @ { #1 #2 , } }
28138      }
28139      {
28140          \_fp_tuple_convert_loop:nNw { } #1
28141          #2 { ? \_fp_tuple_convert_end:w } ;
28142          @ { \use_none:nn }
28143      }
28144  }
28145  \cs_new:Npn \_fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
28146  {
28147      \use_none:n #3
28148      \exp_args:Nf \_fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
28149      @ { #6 , ~ #1 }
28150  }
28151  \cs_new:Npn \_fp_tuple_convert_end:w #1 @ #2
28152  { \exp_after:wN ( \exp:w \exp_end_continue_f:w #2 ) }
```

(End definition for `\_fp_tuple_convert:Nw`, `\_fp_tuple_convert_loop:nNw`, and `\_fp_tuple_convert_end:w`.)

### 76.2 Trimming trailing zeros

`\_fp_trim_zeros:w` If `#1` ends with a 0, the `loop` auxiliary takes that zero as an end-delimiter for its first  
`\_fp_trim_zeros_loop:w` argument, and the second argument is the same `loop` auxiliary. Once the last trailing  
`\_fp_trim_zeros_dot:w`  
`\_fp_trim_zeros_end:w`

zero is reached, the second argument is the `dot` auxiliary, which removes a trailing dot if any. We then clean-up with the `end` auxiliary, keeping only the number.

```

28153 \cs_new:Npn \__fp_trim_zeros:w #1 ;
28154 {
28155     \__fp_trim_zeros_loop:w #1
28156     ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s__fp_stop
28157 }
28158 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
28159 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
28160 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s__fp_stop { #1 }

(End definition for \__fp_trim_zeros:w and others.)
```

### 76.3 Scientific notation

`\fp_to_scientific:N` `\fp_to_scientific:c` The three public functions evaluate their argument, then pass it to `\__fp_to_scientific_dispatch:w`.

```

\fp_to_scientific:n
28161 \cs_new:Npn \fp_to_scientific:N #1
28162 { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
28163 \cs_generate_variant:Nn \fp_to_scientific:N { c }
28164 \cs_new:Npn \fp_to_scientific:n
28165 {
28166     \exp_after:wN \__fp_to_scientific_dispatch:w
28167     \exp:w \exp_end_continue_f:w \__fp_parse:n
28168 }
```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 243.)

`\__fp_to_scientific_dispatch:w`  
`\__fp_to_scientific_recover:w`  
`\__fp_tuple_to_scientific:w` We allow tuples.

```

28169 \cs_new:Npn \__fp_to_scientific_dispatch:w #1
28170 {
28171     \__fp_change_func_type:NNN
28172     #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
28173     #1
28174 }
28175 \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
28176 {
28177     \__fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
28178     nan
28179 }
28180 \cs_new:Npn \__fp_tuple_to_scientific:w
28181 { \__fp_tuple_convert:Nw \__fp_to_scientific_dispatch:w }
```

(End definition for `\__fp_to_scientific_dispatch:w`, `\__fp_to_scientific_recover:w`, and `\__fp_tuple_to_scientific:w`)

`\__fp_to_scientific:w`  
`\__fp_to_scientific_normal:wnnnnn`  
`\__fp_to_scientific_normal:wNw` Expressing an internal floating point number in scientific notation is quite easy: no rounding, and the format is very well defined. First cater for the sign: negative numbers (`#2 = 2`) start with `-`; we then only need to care about positive numbers and `nan`. Then filter the special cases:  $\pm 0$  are represented as `0`; infinities are converted to a number slightly larger than the largest after an “`invalid_operation`” exception; `nan` is represented as `0` after an “`invalid_operation`” exception. In the normal case, decrement the exponent

and unbrace the 4 brace groups, then in a second step grab the first digit (previously hidden in braces) to order the various parts correctly.

```

28182 \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
28183 {
28184     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28185     \if_case:w #1 \exp_stop_f:
28186         \__fp_case_return:nw { 0.00000000000000e0 }
28187     \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
28188     \or:
28189         \__fp_case_use:nw
28190         {
28191             \__fp_invalid_operation:nnw
28192                 { \fp_to_scientific:N \c__fp_overflowing_fp }
28193                 { fp_to_scientific }
28194         }
28195     \or:
28196         \__fp_case_use:nw
28197         {
28198             \__fp_invalid_operation:nnw
28199                 { \fp_to_scientific:N \c_zero_fp }
28200                 { fp_to_scientific }
28201         }
28202     \fi:
28203     \s__fp \__fp_chk:w #1 #2
28204 }
28205 \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
28206     \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28207 {
28208     \exp_after:wN \__fp_to_scientific_normal:wNw
28209     \exp_after:wN e
28210     \int_value:w \__fp_int_eval:w #2 - 1
28211     ; #3 #4 #5 #6 ;
28212 }
28213 \cs_new:Npn \__fp_to_scientific_normal:wNw #1 ; #2#3;
28214     { #2.#3 #1 }

(End definition for \__fp_to_scientific:w, \__fp_to_scientific_normal:wnnnnn, and \__fp_to_scientific_normal:wNw.)
```

## 76.4 Decimal representation

**\fp\_to\_decimal:N** All three public variants are based on the same \\_\_fp\_to\_decimal\_dispatch:w after evaluating their argument to an internal floating point.

```

\fp_to_decimal:c
\fp_to_decimal:n
28215 \cs_new:Npn \fp_to_decimal:N #1
28216     { \exp_after:wN \__fp_to_decimal_dispatch:w #1 }
28217     \cs_generate_variant:Nn \fp_to_decimal:N { c }
28218     \cs_new:Npn \fp_to_decimal:n
28219     {
28220         \exp_after:wN \__fp_to_decimal_dispatch:w
28221         \exp:w \exp_end_continue_f:w \__fp_parse:n
28222     }
```

(End definition for \fp\_to\_decimal:N and \fp\_to\_decimal:n. These functions are documented on page 242.)

```
\_\_fp\_to\_decimal_dispatch:w
```

We allow tuples.

```
28223 \cs_new:Npn \_\_fp\_to\_decimal_dispatch:w #1
28224 {
28225     \_\_fp_change_func_type:NNN
28226     #1 \_\_fp\_to\_decimal:w \_\_fp\_to\_decimal_recover:w
28227     #1
28228 }
28229 \cs_new:Npn \_\_fp\_to\_decimal_recover:w #1 #2 ;
28230 {
28231     \_\_fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
28232     nan
28233 }
28234 \cs_new:Npn \_\_fp\_tuple\_to\_decimal:w
28235 { \_\_fp\_tuple\_convert:Nw \_\_fp\_to\_decimal_dispatch:w }
```

(End definition for `\_\_fp\_to\_decimal_dispatch:w`, `\_\_fp\_to\_decimal_recover:w`, and `\_\_fp\_tuple\_to\_decimal:w`.)

```
\_\_fp\_to\_decimal:w
\_\_fp\_to\_decimal\_normal:wnnnn
\_\_fp\_to\_decimal\_large:Nnnw
\_\_fp\_to\_decimal\_huge:wnnnn
```

The structure is similar to `\_\_fp\_to\_scientific:w`. Insert `-` for negative numbers. Zero gives 0,  $\pm\infty$  and `nan` yield an “invalid operation” exception; note that  $\pm\infty$  produces a very large output, which we don’t expand now since it most likely won’t be needed. Normal numbers with an exponent in the range [1, 15] have that number of digits before the decimal separator: “decimate” them, and remove leading zeros with `\int_value:w`, then trim trailing zeros and dot. Normal numbers with an exponent 16 or larger have no decimal separator, we only need to add trailing zeros. When the exponent is non-positive, the result should be `0.(zeros)(digits)`, trimmed.

```
28236 \cs_new:Npn \_\_fp\_to\_decimal:w \s__fp \_\_fp\_chk:w #1#2
28237 {
28238     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28239     \if_case:w #1 \exp_stop_f:
28240         \_\_fp_case_return:nw { 0 }
28241     \or: \exp_after:wN \_\_fp\_to\_decimal\_normal:wnnnnn
28242     \or:
28243         \_\_fp_case_use:nw
28244         {
28245             \_\_fp_invalid_operation:nnw
28246             { \fp_to_decimal:N \c__fp_overflowing_fp }
28247             { fp_to_decimal }
28248         }
28249     \or:
28250         \_\_fp_case_use:nw
28251         {
28252             \_\_fp_invalid_operation:nnw
28253             { 0 }
28254             { fp_to_decimal }
28255         }
28256     \fi:
28257     \s__fp \_\_fp\_chk:w #1 #2
28258 }
28259 \cs_new:Npn \_\_fp\_to\_decimal\_normal:wnnnnn
28260     \s__fp \_\_fp\_chk:w 1 #1 #2 #3#4#5#6 ;
28261 {
28262     \int_compare:nNnTF {#2} > 0
28263     {
```

```

28264     \int_compare:nNnTF {#2} < \c_fp_prec_int
28265     {
28266         \fp_decimate:nNnnn { \c_fp_prec_int - #2 }
28267             \fp_to_decimal_large:Nnnw
28268     }
28269     {
28270         \exp_after:wN \exp_after:wN
28271             \exp_after:wN \fp_to_decimal_huge:wnnnn
28272                 \prg_replicate:nn { #2 - \c_fp_prec_int } { 0 } ;
28273     }
28274     {#3} {#4} {#5} {#6}
28275 }
28276 {
28277     \exp_after:wN \fp_trim_zeros:w
28278     \exp_after:wN 0
28279     \exp_after:wN .
28280     \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
28281         #3#4#5#6 ;
28282     }
28283 }
28284 \cs_new:Npn \fp_to_decimal_large:Nnnw #1#2#3#4;
28285 {
28286     \exp_after:wN \fp_trim_zeros:w \int_value:w
28287     \if_int_compare:w #2 > \c_zero_int
28288         #2
28289     \fi:
28290     \exp_stop_f:
28291         #3.#4 ;
28292 }
28293 \cs_new:Npn \fp_to_decimal_huge:wnnnn #1; #2#3#4#5 { #2#3#4#5 #1 }

(End definition for \fp_to_decimal:w and others.)

```

## 76.5 Token list representation

**\fp\_to\_tl:N** These three public functions evaluate their argument, then pass it to \fp\_to\_tl\_dispatch:w.

```

\fp_to_tl:c 28294 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \fp_to_tl_dispatch:w #1 }
28295 \cs_generate_variant:Nn \fp_to_tl:N { c }
28296 \cs_new:Npn \fp_to_tl:n
28297 {
28298     \exp_after:wN \fp_to_tl_dispatch:w
28299     \exp:w \exp_end_continue_f:w \fp_parse:n
28300 }

```

(End definition for \fp\_to\_tl:N and \fp\_to\_tl:n. These functions are documented on page 243.)

\fp\_to\_tl\_dispatch:w We allow tuples.

```

\fp_to_tl_recover:w 28301 \cs_new:Npn \fp_to_tl_dispatch:w #1
28302     { \fp_change_func_type:NNN #1 \fp_to_tl:w \fp_to_tl_recover:w #1 }
28303 \cs_new:Npn \fp_to_tl_recover:w #1 #2 ;
28304 {
28305     \fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }

```

```

28306      nan
28307    }
28308 \cs_new:Npn \__fp_tuple_to_tl:w
28309   { \__fp_tuple_convert:Nw \__fp_to_tl_dispatch:w }

(End definition for \__fp_to_tl_dispatch:w, \__fp_to_tl_recover:w, and \__fp_tuple_to_tl:w.)

\__fp_to_tl:w A structure similar to \__fp_to_scientific_dispatch:w and \__fp_to_decimal_
\__fp_to_tl_normal:nnnnn dispatch:w, but without the “invalid operation” exception. First filter special cases.
\__fp_to_tl_scientific:wnnnnn We express normal numbers in decimal notation if the exponent is in the range [-2, 16],
\__fp_to_tl_scientific:wNw and otherwise use scientific notation.

28310 \cs_new:Npn \__fp_to_tl:w \s__fp \__fp_chk:w #1#2
28311   {
28312     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
28313     \if_case:w #1 \exp_stop_f:
28314       \__fp_case_return:nw { 0 }
28315     \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
28316     \or: \__fp_case_return:nw { inf }
28317     \else: \__fp_case_return:nw { nan }
28318     \fi:
28319   }
28320 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
28321   {
28322     \int_compare:nTF
28323       { -2 <= #1 <= \c_fp_prec_int }
28324       { \__fp_to_decimal_normal:wnnnnn }
28325       { \__fp_to_tl_scientific:wnnnnn }
28326     \s__fp \__fp_chk:w 1 0 {#1}
28327   }
28328 \cs_new:Npn \__fp_to_tl_scientific:wnnnnn
28329   \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
28330   {
28331     \exp_after:wN \__fp_to_tl_scientific:wNw
28332     \exp_after:wN e
28333     \int_value:w \__fp_int_eval:w #2 - 1
28334     ; #3 #4 #5 #6 ;
28335   }
28336 \cs_new:Npn \__fp_to_tl_scientific:wNw #1 ; #2#3;
28337   { \__fp_trim_zeros:w #2.#3 ; #1 }

(End definition for \__fp_to_tl:w and others.)

```

## 76.6 Formatting

This is not implemented yet, as it is not yet clear what a correct interface would be, for this kind of structured conversion from a floating point (or other types of variables) to a string. Ideas welcome.

## 76.7 Convert to dimension or integer

\fp\_to\_dim:N All three public variants are based on the same \\_\_fp\_to\_dim\_dispatch:w after evaluating their argument to an internal floating point. We only allow floating point numbers,

\fp\_to\_dim:c

\fp\_to\_dim:n

\\_\_fp\_to\_dim\_dispatch:w

\\_\_fp\_to\_dim\_recover:w

\\_\_fp\_to\_dim:w

not tuples.

```

28338 \cs_new:Npn \fp_to_dim:N #1
28339   { \exp_after:wN \__fp_to_dim_dispatch:w #1 }
28340 \cs_generate_variant:Nn \fp_to_dim:N { c }
28341 \cs_new:Npn \fp_to_dim:n
28342   {
28343     \exp_after:wN \__fp_to_dim_dispatch:w
28344     \exp:w \exp_end_continue_f:w \__fp_parse:n
28345   }
28346 \cs_new:Npn \__fp_to_dim_dispatch:w #1#2 ;
28347   {
28348     \__fp_change_func_type:NNN #1 \__fp_to_dim:w \__fp_to_dim_recover:w
28349     #1 #2 ;
28350   }
28351 \cs_new:Npn \__fp_to_dim_recover:w #1
28352   { \__fp_invalid_operation:nnw { Opt } { fp_to_dim } }
28353 \cs_new:Npn \__fp_to_dim:w #1 ; { \__fp_to_decimal:w #1 ; pt }
```

(End definition for `\fp_to_dim:N` and others. These functions are documented on page 242.)

**\fp\_to\_int:N** For the most part identical to `\fp_to_dim:N` but without `pt`, and where `\__fp_to_int:w` does more work. To convert to an integer, first round to 0 places (to the nearest integer), then express the result as a decimal number: the definition of `\__fp_to_decimal_dispatch:w` is such that there are no trailing dot nor zero.

```

\__fp_to_int_dispatch:w
\__fp_to_int_recover:w
28354 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \__fp_to_int_dispatch:w #1 }
28355 \cs_generate_variant:Nn \fp_to_int:N { c }
28356 \cs_new:Npn \fp_to_int:
28357   {
28358     \exp_after:wN \__fp_to_int_dispatch:w
28359     \exp:w \exp_end_continue_f:w \__fp_parse:n
28360   }
28361 \cs_new:Npn \__fp_to_int_dispatch:w #1#2 ;
28362   {
28363     \__fp_change_func_type:NNN #1 \__fp_to_int:w \__fp_to_int_recover:w
28364     #1 #2 ;
28365   }
28366 \cs_new:Npn \__fp_to_int_recover:w #1
28367   { \__fp_invalid_operation:nnw { 0 } { fp_to_int } }
28368 \cs_new:Npn \__fp_to_int:w #1;
28369   {
28370     \exp_after:wN \__fp_to_decimal:w \exp:w \exp_end_continue_f:w
28371     \__fp_round:Nnw \__fp_round_to_nearest>NNN #1; { 0 }
28372 }
```

(End definition for `\fp_to_int:N` and others. These functions are documented on page 242.)

## 76.8 Convert from a dimension

**\dim\_to\_fp:n** The dimension expression (which can in fact be a glue expression) is evaluated, converted to a number (*i.e.*, expressed in scaled points), then multiplied by  $2^{-16} = 0.0000152587890625$  to give a value expressed in points. The auxiliary `\__fp_mul_npos_o:Nww` expects the desired *(final sign)* and two floating point operands (of the form

```

\__fp_from_dim_test:ww
\__fp_from_dim:wNw
\__fp_from_dim:wNNnnnnnn
\__fp_from_dim:wnnnnnnwNw
```

`\s_fp ... ;`) as arguments. This set of functions is also used to convert dimension registers to floating points while parsing expressions: in this context there is an additional exponent, which is the first argument of `\_fp_from_dim_test:ww`, and is combined with the exponent  $-4$  of  $2^{-16}$ . There is also a need to expand afterwards: this is performed by `\_fp_mul_npos_o:Nww`, and cancelled by `\prg_do_nothing:` here.

```

28373 \cs_new:Npn \dim_to_fp:n #1
28374 {
28375     \exp_after:wN \_fp_from_dim_test:ww
28376     \exp_after:wN 0
28377     \exp_after:wN ,
28378     \int_value:w \tex_glueexpr:D #1 ;
28379 }
28380 \cs_new:Npn \_fp_from_dim_test:ww #1, #2
28381 {
28382     \if_meaning:w 0 #2
28383         \_fp_case_return:nw { \exp_after:wN \c_zero_fp }
28384     \else:
28385         \exp_after:wN \_fp_from_dim:wNw
28386         \int_value:w \_fp_int_eval:w #1 - 4
28387         \if_meaning:w - #2
28388             \exp_after:wN , \exp_after:wN 2 \int_value:w
28389         \else:
28390             \exp_after:wN , \exp_after:wN 0 \int_value:w #2
28391         \fi:
28392     \fi:
28393 }
28394 \cs_new:Npn \_fp_from_dim:wNw #1,#2#3;
28395 {
28396     \_fp_pack_twice_four:wNNNNNNNN \_fp_from_dim:wNNnnnnnn ;
28397     #3 000 0000 00 {10}987654321; #2 {#1}
28398 }
28399 \cs_new:Npn \_fp_from_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
28400 { \_fp_from_dim:wnnnnwNn #1 {#2#300} {0000} ; }
28401 \cs_new:Npn \_fp_from_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
28402 {
28403     \_fp_mul_npos_o:Nww #7
28404     \s_fp \_fp_chk:w 1 #7 {#5} #1 ;
28405     \s_fp \_fp_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
28406     \prg_do_nothing:
28407 }
```

(End definition for `\dim_to_fp:n` and others. This function is documented on page 214.)

## 76.9 Use and eval

**\fp\_use:N** Those public functions are simple copies of the decimal conversions.  
**\fp\_use:c** `\cs_new_eq:NN \fp_use:N \fp_to_decimal:N`  
**\fp\_eval:n** `\cs_generate_variant:Nn \fp_use:N { c }`  
`\cs_new_eq:NN \fp_eval:n \fp_to_decimal:n`

(End definition for `\fp_use:N` and `\fp_eval:n`. These functions are documented on page 243.)

**\fp\_sign:n** Trivial but useful. See the implementation of \fp\_add:Nn for an explanation of why to use \\_\_fp\_parse:n, namely, for better error reporting.

```
28411 \cs_new:Npn \fp_sign:n #1
28412   { \fp_to_decimal:n { sign \__fp_parse:n {#1} } }
```

(End definition for \fp\_sign:n. This function is documented on page 242.)

**\fp\_abs:n** Trivial but useful. See the implementation of \fp\_add:Nn for an explanation of why to use \\_\_fp\_parse:n, namely, for better error reporting.

```
28413 \cs_new:Npn \fp_abs:n #1
28414   { \fp_to_decimal:n { abs \__fp_parse:n {#1} } }
```

(End definition for \fp\_abs:n. This function is documented on page 258.)

**\fp\_max:nn** Similar to \fp\_abs:n, for consistency with \int\_max:nn, etc.

**\fp\_min:nn**

```
28415 \cs_new:Npn \fp_max:nn #1#2
28416   { \fp_to_decimal:n { max ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
28417 \cs_new:Npn \fp_min:nn #1#2
28418   { \fp_to_decimal:n { min ( \__fp_parse:n {#1} , \__fp_parse:n {#2} ) } }
```

(End definition for \fp\_max:nn and \fp\_min:nn. These functions are documented on page 258.)

## 76.10 Convert an array of floating points to a comma list

\\_\_fp\_array\_to\_clist:n  
\\_\_fp\_array\_to\_clist\_loop:Nw Converts an array of floating point numbers to a comma-list. If speed here ends up irrelevant, we can simplify the code for the auxiliary to become

```
\cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
{
  \use_none:n #1
  { , ~ } \fp_to_tl:n { #1 #2 ; }
  \__fp_array_to_clist_loop:Nw
}
```

The \use\_ii:nn function is expanded after \\_\_fp\_expand:n is done, and it removes ,~ from the start of the representation.

```
28419 \cs_new:Npn \__fp_array_to_clist:n #1
28420   {
28421     \tl_if_empty:nF {#1}
28422     {
28423       \exp_last_unbraced:N \use_ii:nn
28424       {
28425         \__fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
28426         \prg_break_point:
28427       }
28428     }
28429   }
28430 \cs_new:Npn \__fp_array_to_clist_loop:Nw #1#2;
28431   {
28432     \use_none:n #1
28433     , ~
```

```
28434      \exp_not:f { \_fp_to_tl_dispatch:w #1 #2 ; }
28435      \_fp_array_to_clist_loop:Nw
28436  }
28437  ⟨/package⟩
```

(End definition for `\_fp_array_to_clist:n` and `\_fp_array_to_clist_loop:Nw`.)

# Chapter 77

## I3fp-random Implementation

```
28438  <*package>
28439  <@@=fp>
```

`\__fp_parse_word_rand:N` Those functions may receive a variable number of arguments. We won't use the argument ?.

```
28440  \cs_new:Npn \__fp_parse_word_rand:N
28441    { \__fp_parse_function>NNN \__fp_rand_o:Nw ? }
28442  \cs_new:Npn \__fp_parse_word_randint:N
28443    { \__fp_parse_function>NNNN \__fp_randint_o:Nw ? }
```

(End definition for `\__fp_parse_word_rand:N` and `\__fp_parse_word_randint:N`.)

### 77.1 Engine support

Most engines provide random numbers, but not all. We write the test twice simply in order to write the `false` branch first.

```
28444  \sys_if_rand_exist:F
28445  {
28446    \msg_new:nnn { kernel } { fp-no-random }
28447      { Random~numbers~unavailable~for~#1 }
28448    \cs_new:Npn \__fp_rand_o:Nw ? #1 @
28449    {
28450      \msg_expandable_error:nnn { kernel } { fp-no-random }
28451        { fp-rand }
28452      \exp_after:wN \c_nan_fp
28453    }
28454    \cs_new_eq:NN \__fp_randint_o:Nw \__fp_rand_o:Nw
28455    \cs_new:Npn \int_rand:nn #1#2
28456    {
28457      \msg_expandable_error:nnn { kernel } { fp-no-random }
28458        { \int_rand:nn {#1} {#2} }
28459      \int_eval:n {#1}
28460    }
28461    \cs_new:Npn \int_rand:n #1
28462    {
28463      \msg_expandable_error:nnn { kernel } { fp-no-random }
28464        { \int_rand:n {#1} }
```

```

28465     1
28466   }
28467   }
28468 \sys_if_rand_exist:T
28469   {

```

Obviously, every word “random” below means “pseudo-random”, as we have no access to entropy (except a very unreliable source of entropy: the time it takes to run some code).

The primitive random number generator (RNG) is provided as `\tex_uniformdeviate:D`. Under the hood, it maintains an array of 55 28-bit numbers, updated with a linear recursion relation (similar to Fibonacci numbers) modulo  $2^{28}$ . When `\tex_uniformdeviate:D <integer>` is called (for brevity denote by  $N$  the `<integer>`), the next 28-bit number is read from the array, scaled by  $N/2^{28}$ , and rounded. To prevent 0 and  $N$  from appearing half as often as other numbers, they are both mapped to the result 0.

This process means that `\tex_uniformdeviate:D` only gives a uniform distribution from 0 to  $N-1$  if  $N$  is a divisor of  $2^{28}$ , so we will mostly call the RNG with such power of 2 arguments. If  $N$  does not divide  $2^{28}$ , then the relative non-uniformity (difference between probabilities of getting different numbers) is about  $N/2^{28}$ . This implies that detecting deviation from  $1/N$  of the probability of a fixed value  $X$  requires about  $2^{56}/N$  random trials. But collective patterns can reduce this to about  $2^{56}/N^2$ . For instance with  $N = 3 \times 2^k$ , the modulo 3 repartition of such random numbers is biased with a non-uniformity about  $2^k/2^{28}$  (which is much worse than the circa  $3/2^{28}$  non-uniformity from taking directly  $N = 3$ ). This is detectable after about  $2^{56}/2^{2k} = 9 \cdot 2^{56}/N^2$  random numbers. For  $k = 15$ ,  $N = 98304$ , this means roughly  $2^{26}$  calls to the RNG (experimentally this takes at the very least 16 seconds on a 2 giga-hertz processor). While this bias is not quite problematic, it is uncomfortably close to being so, and it becomes worse as  $N$  is increased. In our code, we shall thus combine several results from the RNG.

The RNG has three types of unexpected correlations. First, everything is linear modulo  $2^{28}$ , hence the lowest  $k$  bits of the random numbers only depend on the lowest  $k$  bits of the seed (and of course the number of times the RNG was called since setting the seed). The recommended way to get a number from 0 to  $N-1$  is thus to scale the raw 28-bit integer, as the engine’s RNG does. We will go further and in fact typically we discard some of the lowest bits.

Second, suppose that we call the RNG with the same argument  $N$  to get a set of  $K$  integers in  $[0, N-1]$  (throwing away repeats), and suppose that  $N > K^3$  and  $K > 55$ . The recursion used to construct more 28-bit numbers from previous ones is linear:  $x_n = x_{n-55} - x_{n-24}$  or  $x_n = x_{n-55} - x_{n-24} + 2^{28}$ . After rescaling and rounding we find that the result  $N_n \in [0, N-1]$  is among  $N_{n-55} - N_{n-24} + \{-1, 0, 1\}$  modulo  $N$  (a more detailed analysis shows that 0 appears with frequency close to  $3/4$ ). The resulting set thus has more triplets  $(a, b, c)$  than expected obeying  $a = b + c$  modulo  $N$ . Namely it will have of order  $(K-55) \times 3/4$  such triplets, when one would expect  $K^3/(6N)$ . This starts to be detectable around  $N = 2^{18} > 55^3$  (earlier if one keeps track of positions too, but this is more subtle than it looks because the array of 28-bit integers is read backwards by the engine). Hopefully the correlation is subtle enough to not affect realistic documents so we do not specifically mitigate against this. Since we typically use two calls to the RNG per `\int_rand:nn` we would need to investigate linear relations between the  $x_{2n}$  on the one hand and between the  $x_{2n+1}$  on the other hand. Such relations will have more complicated coefficients than  $\pm 1$ , which alleviates the issue.

Third, consider successive batches of 165 calls to the RNG (with argument  $2^{28}$  or with argument 2 for instance), then most batches have more odd than even numbers. Note

that this does not mean that there are more odd than even numbers overall. Similar issues are discussed in Knuth's TAOCP volume 2 near exercise 3.3.2-31. We do not have any mitigation strategy for this.

Ideally, our algorithm should be:

- Uniform. The result should be as uniform as possible assuming that the RNG's underlying 28-bit integers are uniform.
- Uncorrelated. The result should not have detectable correlations between different seeds, similar to the lowest-bit ones mentioned earlier.
- Quick. The algorithm should be fast in  $\text{\TeX}$ , so no “bit twiddling”, but “digit twiddling” is ok.
- Simple. The behaviour must be documentable precisely.
- Predictable. The number of calls to the RNG should be the same for any `\int-rand:nn`, because then the algorithm can be modified later without changing the result of other uses of the RNG.
- Robust. It should work even for `\int-rand:nn { - \c_max_int } { \c_max_int }` where the range is not representable as an integer. In fact, we also provide later a floating-point `randint` whose range can go all the way up to  $2 \times 10^{16} - 1$  possible values.

Some of these requirements conflict. For instance, uniformity cannot be achieved with a fixed number of calls to the RNG.

Denote by `random(N)` one call to `\tex_uniformdeviate:D` with argument  $N$ , and by `ediv(p, q)` the  $\varepsilon$ - $\text{\TeX}$  rounding division giving  $\lfloor p/q + 1/2 \rfloor$ . Denote by  $\langle \min \rangle$ ,  $\langle \max \rangle$  and  $R = \langle \max \rangle - \langle \min \rangle + 1$  the arguments of `\int_min:nn` and the number of possible outcomes. Note that  $R \in [1, 2^{32} - 1]$  cannot necessarily be represented as an integer (however,  $R - 2^{31}$  can). Our strategy is to get two 28-bit integers  $X$  and  $Y$  from the RNG, split each into 14-bit integers, as  $X = X_1 \times 2^{14} + X_0$  and  $Y = Y_1 \times 2^{14} + Y_0$  then return essentially  $\langle \min \rangle + \lfloor R(X_1 \times 2^{-14} + Y_1 \times 2^{-28} + Y_0 \times 2^{-42} + X_0 \times 2^{-56}) \rfloor$ . For small  $R$  the  $X_0$  term has a tiny effect so we ignore it and we can compute  $R \times Y/2^{28}$  much more directly by `random(R)`.

- If  $R \leq 2^{17} - 1$  then return  $\text{ediv}(R \text{random}(2^{14}) + \text{random}(R) + 2^{13}, 2^{14}) - 1 + \langle \min \rangle$ . The shifts by  $2^{13}$  and  $-1$  convert  $\varepsilon$ - $\text{\TeX}$  division to truncated division. The bound on  $R$  ensures that the number obtained after the shift is less than `\c_max_int`. The non-uniformity is at most of order  $2^{17}/2^{42} = 2^{-25}$ .
- Split  $R = R_2 \times 2^{28} + R_1 \times 2^{14} + R_0$ , where  $R_2 \in [0, 15]$ . Compute  $\langle \min \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$  then map a result of  $\langle \max \rangle + 1$  to  $\langle \min \rangle$ . Writing each `ediv` in terms of truncated division with a shift, and using  $\lfloor (p + \lfloor r/s \rfloor)/q \rfloor = \lfloor (ps + r)/(sq) \rfloor$ , what we compute is equal to  $\lfloor \langle \text{exact} \rangle + 2^{-29} + 2^{-15} + 2^{-1} \rfloor$  with  $\langle \text{exact} \rangle = \langle \min \rangle + R \times 0.X_1 Y_1 X_0$ . Given we map  $\langle \max \rangle + 1$  to  $\langle \min \rangle$ , the shift has no effect on uniformity. The non-uniformity is bounded by  $R/2^{56} < 2^{-24}$ . It may be possible to speed up the code by dropping tiny terms such as  $R_0 X_0$ , but the analysis of non-uniformity proves too difficult.

To avoid the overflow when the computation yields  $\langle \max \rangle + 1$  with  $\langle \max \rangle = 2^{31} - 1$  (note that  $R$  is then arbitrary), we compute the result in two pieces. Compute

$\langle \text{first} \rangle = \langle \text{min} \rangle + R_2 X_1 2^{14}$  if  $R_2 < 8$  or  $\langle \text{min} \rangle + 8X_1 2^{14} + (R_2 - 8)X_1 2^{14}$  if  $R_2 \geq 8$ , the expressions being chosen to avoid overflow. Compute  $\langle \text{second} \rangle = R_2 Y_1 + R_1 X_1 + \text{ediv}(\dots)$ , at most  $R_2 2^{14} + R_1 2^{14} + R_0 \leq 2^{28} + 15 \times 2^{14} - 1$ , not at risk of overflowing. We have  $\langle \text{first} \rangle + \langle \text{second} \rangle = \langle \text{max} \rangle + 1 = \langle \text{min} \rangle + R$  if and only if  $\langle \text{second} \rangle = R 12^{14} + R_0 + R_2 2^{14}$  and  $2^{14} R_2 X_1 = 2^{28} R_2 - 2^{14} R_2$  (namely  $R_2 = 0$  or  $X_1 = 2^{14} - 1$ ). In that case, return  $\langle \text{min} \rangle$ , otherwise return  $\langle \text{first} \rangle + \langle \text{second} \rangle$ , which is safe because it is at most  $\langle \text{max} \rangle$ . Note that the decision of what to return does not need  $\langle \text{first} \rangle$  explicitly so we don't actually compute it, just put it in an integer expression in which  $\langle \text{second} \rangle$  is eventually added (or not).

- To get a floating point number in  $[0, 1)$  just call the  $R = 10000 \leq 2^{17} - 1$  procedure above to produce four blocks of four digits.
- To get an integer floating point number in a range (whose size can be up to  $2 \times 10^{16} - 1$ ), work with fixed-point numbers: get six times four digits to build a fixed point number, multiply by  $R$  and add  $\langle \text{min} \rangle$ . This requires some care because l3fp-extended only supports non-negative numbers.

`\c_kernel_randint_max_int` Constant equal to  $2^{17} - 1$ , the maximal size of a range that `\int_range:nn` can do with its “simple” algorithm.

```
28470      \int_const:Nn \c_kernel_randint_max_int { 131071 }
```

(End definition for `\c_kernel_randint_max_int`.)

`\_kernel_randint:n` Used in an integer expression, `\_kernel_randint:n {R}` gives a random number  $1 + \lfloor (R \text{random}(2^{14}) + \text{random}(R)) / 2^{14} \rfloor$  that is in  $[1, R]$ . Previous code was computing  $\lfloor p / 2^{14} \rfloor$  as  $\text{ediv}(p - 2^{13}, 2^{14})$  but that wrongly gives  $-1$  for  $p = 0$ .

```
28471      \cs_new:Npn \_kernel_randint:n #1
28472      {
28473          (#1 * \tex_uniformdeviate:D 16384
28474          + \tex_uniformdeviate:D #1 + 8192 ) / 16384
28475      }
```

(End definition for `\_kernel_randint:n`.)

`\_fp_rand_myriads:n` Used as `\_fp_rand_myriads:n {XXX}` with one letter X (specifically) per block of four digit we want; it expands to ; followed by the requested number of brace groups, each containing four (pseudo-random) digits. Digits are produced as a random number in  $[10000, 19999]$  for the usual reason of preserving leading zeros.

```
28476      \cs_new:Npn \_fp_rand_myriads:n #1
28477      { \_fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
28478      \cs_new:Npn \_fp_rand_myriads_loop:w #1 X
28479      {
28480          #1
28481          \exp_after:wN \_fp_rand_myriads_get:w
28482          \int_value:w \_fp_int_eval:w 9999 +
28483              \_kernel_randint:n { 10000 }
28484              \_fp_rand_myriads_loop:w
28485      }
28486      \cs_new:Npn \_fp_rand_myriads_get:w 1 #1 ; { ; {#1} }
```

(End definition for `\_fp_rand_myriads:n`, `\_fp_rand_myriads_loop:w`, and `\_fp_rand_myriads_get:w`.)

## 77.2 Random floating point

```
\_\_fp\_rand_o:Nw
\_\_fp\_rand_o:w
```

First we check that `random` was called without argument. Then get four blocks of four digits and convert that fixed point number to a floating point number (this correctly sets the exponent). This has a minor bug: if all of the random numbers are zero then the result is correctly 0 but it raises the `underflow` flag; it should not do that.

```
28487      \cs_new:Npn \_\_fp\_rand_o:Nw ? #1 @
28488      {
28489          \tl_if_empty:nTF {#1}
28490          {
28491              \exp_after:wN \_\_fp\_rand_o:w
28492              \exp:w \exp_end_continue_f:w
28493              \_\_fp\_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
28494          }
28495          {
28496              \msg_expandable_error:nnnnn
28497                  { fp } { num-args } { rand() } { 0 } { 0 }
28498              \exp_after:wN \c_nan_fp
28499          }
28500      }
28501      \cs_new:Npn \_\_fp\_rand_o:w ;
28502      {
28503          \exp_after:wN \_\_fp\_sanitize:Nw
28504          \exp_after:wN 0
28505          \int_value:w \_\_fp\_int_eval:w \c_zero_int
28506          \_\_fp_fixed_to_float_o:wN
28507      }
```

(End definition for `\_\_fp\_rand_o:Nw` and `\_\_fp\_rand_o:w`.)

## 77.3 Random integer

Enforce that there is one argument (then add first argument 1) or two arguments. Call `\_\_fp\_randint_badarg:w` on each; this function inserts 1 \exp\_stop\_f: to end the `\if_case:w` statement if either the argument is not an integer or if its absolute value is  $\geq 10^{16}$ . Also bail out if `\_\_fp_compare_back:ww` yields 1, meaning that the bounds are not in the right order. Otherwise an auxiliary converts each argument times  $10^{-16}$  (hence the shift in exponent) to a 24-digit fixed point number (see l3fp-extended). Then compute the number of choices,  $\langle max \rangle + 1 - \langle min \rangle$ . Create a random 24-digit fixed-point number with `\_\_fp\_rand_myriads:n`, then use a fused multiply-add instruction to multiply the number of choices to that random number and add it to  $\langle min \rangle$ . Then truncate to 16 digits (namely select the integer part of  $10^{16}$  times the result) before converting back to a floating point number (`\_\_fp\_sanitize:Nw` takes care of zero). To avoid issues with negative numbers, add 1 to all fixed point numbers (namely  $10^{16}$  to the integers they represent), except of course when it is time to convert back to a float.

```
28508      \cs_new:Npn \_\_fp\_randint_o:Nw ?
28509      {
28510          \_\_fp_parse_function_one_two:nnw
28511              { randint }
28512              { \_\_fp\_randint_default:w \_\_fp\_randint_o:w }
28513      }
28514      \cs_new:Npn \_\_fp\_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
```

```

28515 \cs_new:Npn \__fp_randint_badarg:w \s_fp \__fp_chk:w #1#2#3;
28516 {
28517   \__fp_int:wTF \s_fp \__fp_chk:w #1#2#3;
28518   {
28519     \if_meaning:w 1 #1
28520       \if_int_compare:w
28521         \__fp_use_i_until_s:nw #3 ; > \c_fp_prec_int
28522         \c_one_int
28523       \fi:
28524     \fi:
28525   }
28526   { \c_one_int }
28527 }
28528 \cs_new:Npn \__fp_randint_o:w #1; #2; @
28529 {
28530   \if_case:w
28531     \__fp_randint_badarg:w #1;
28532     \__fp_randint_badarg:w #2;
28533     \if:w 1 \__fp_compare_back:ww #2; #1; \c_one_int \fi:
28534     \c_zero_int
28535     \__fp_randint_auxi_o:ww #1; #2;
28536   \or:
28537     \__fp_invalid_operation_tl_o:ff
28538       { randint } { \__fp_array_to_clist:n { #1; #2; } }
28539     \exp:w
28540   \fi:
28541   \exp_after:wN \exp_end:
28542 }
28543 \cs_new:Npn \__fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
28544 {
28545   \fi:
28546   \__fp_randint_auxii:wn #2 ;
28547   { \__fp_randint_auxii:wn #1 ; \__fp_randint_auxiii_o:ww }
28548 }
28549 \cs_new:Npn \__fp_randint_auxii:wn \s_fp \__fp_chk:w #1#2#3#4 ;
28550 {
28551   \if_meaning:w 0 #1
28552     \exp_after:wN \use_i:nn
28553   \else:
28554     \exp_after:wN \use_ii:nn
28555   \fi:
28556   { \exp_after:wN \__fp_fixed_continue:wn \c_fp_one_fixed_tl }
28557 {
28558   \exp_after:wN \__fp_ep_to_fixed:wwn
28559   \int_value:w \__fp_int_eval:w
28560     #3 - \c_fp_prec_int , #4 {0000} {0000} ;
28561 {
28562   \if_meaning:w 0 #2
28563     \exp_after:wN \use_i:nnnn
28564     \exp_after:wN \__fp_fixed_add_one:wn
28565   \fi:
28566     \exp_after:wN \__fp_fixed_sub:wwn \c_fp_one_fixed_tl
28567 }
28568 \__fp_fixed_continue:wn

```

```

28569         }
28570     }
28571 \cs_new:Npn \__fp_randint_auxiii_o:ww #1 ; #2 ;
28572 {
28573     \__fp_fixed_add:wwn #2 ;
28574     {0000} {0000} {0000} {0001} {0000} {0000} ;
28575     \__fp_fixed_sub:wwn #1 ;
28576 {
28577     \exp_after:wN \use_i:nn
28578     \exp_after:wN \__fp_fixed_mul_add:www
28579     \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } ;
28580 }
28581 #1 ;
28582 \__fp_randint_auxiv_o:ww
28583 #2 ;
28584 \__fp_randint_auxv_o:w #1 ; @
28585 }
28586 \cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
28587 {
28588     \if_int_compare:w
28589         \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
28590         \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
28591         #3#4 > #8#9 \exp_stop_f:
28592         \__fp_use_i_until_s:nw
28593     \fi:
28594     \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
28595 }
28596 \cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @
28597 {
28598     \exp_after:wN \__fp_sanitize:Nw
28599     \int_value:w
28600     \if_int_compare:w #1 < 10000 \exp_stop_f:
28601         2
28602     \else:
28603         0
28604         \exp_after:wN \exp_after:wN
28605         \exp_after:wN \__fp_reverse_args:Nww
28606     \fi:
28607     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_one_fixed_tl
28608     {#1} {#2} {#3} {#4} {0000} {0000} ;
28609 {
28610     \exp_after:wN \exp_stop_f:
28611     \int_value:w \__fp_int_eval:w \c__fp_prec_int
28612         \__fp_fixed_to_float_o:wN
28613     }
28614     0
28615     \exp:w \exp_after:wN \exp_end:
28616 }

```

(End definition for `\__fp_randint_o:Nw` and others.)

**\int\_rand:nn** `\__fp_randint:ww` Evaluate the argument and filter out the case where the lower bound #1 is more than the upper bound #2. Then determine whether the range is narrower than `\c__kernel_randint_max_int`; #2-#1 may overflow for very large positive #2 and negative #1. If the

range is narrow, call `\_kernel_randint:n {⟨choices⟩}` where `⟨choices⟩` is the number of possible outcomes. If the range is wide, use somewhat slower code.

```

28617      \cs_new:Npn \int_rand:nn #1#2
28618      {
28619          \int_eval:n
28620          {
28621              \exp_after:wN \_fp_randint:ww
28622              \int_value:w \int_eval:n {#1} \exp_after:wN ;
28623              \int_value:w \int_eval:n {#2} ;
28624          }
28625      }
28626      \cs_new:Npn \_fp_randint:ww #1; #2;
28627      {
28628          \if_int_compare:w #1 > #2 \exp_stop_f:
28629              \msg_expandable_error:nnnn
28630                  { kernel } { randint-backward-range } {#1} {#2}
28631              \_fp_randint:ww #2; #1;
28632          \else:
28633              \if_int_compare:w \_fp_int_eval:w #2
28634                  \if_int_compare:w #1 > \c_zero_int
28635                      - #1 < \_fp_int_eval:w
28636                  \else:
28637                      < \_fp_int_eval:w #1 +
28638                  \fi:
28639                  \c_kernel_randint_max_int
28640                  \_fp_int_eval_end:
28641                  \_kernel_randint:n
28642                      { \_fp_int_eval:w #2 - #1 + 1 \_fp_int_eval_end: }
28643                      - 1 + #1
28644                  \else:
28645                      \_kernel_randint:nn {#1} {#2}
28646                  \fi:
28647                  \fi:
28648      }

```

(End definition for `\int_rand:nn` and `\_fp_randint:ww`. This function is documented on page 166.)

`\_kernel_randint:nn` Any  $n \in [-2^{31} + 1, 2^{31} - 1]$  is uniquely written as  $2^{14}n_1 + n_2$  with  $n_1 \in [-2^{17}, 2^{17} - 1]$  and  $n_2 \in [0, 2^{14} - 1]$ . Calling `\_fp_randint_split_o:Nw n`; gives  $n_1$ ;  $n_2$ ; and expands the next token once. We do this for two random numbers and apply `\_fp_randint-split_o:Nw` twice to fully decompose the range  $R$ . One subtlety is that we compute  $R - 2^{31} = \langle \max \rangle - \langle \min \rangle - (2^{31} - 1) \in [-2^{31} + 1, 2^{31} - 1]$  rather than  $R$  to avoid overflow.

Then we have `\_fp_randint_wide_aux:w ⟨X1⟩;⟨X0⟩;⟨Y1⟩;⟨Y0⟩;⟨R2⟩;⟨R1⟩;⟨R0⟩;`. and we apply the algorithm described earlier.

```

28649      \cs_new:Npn \_kernel_randint:nn #1#2
28650      {
28651          #1
28652          \exp_after:wN \_fp_randint_wide_aux:w
28653          \int_value:w
28654          \exp_after:wN \_fp_randint_split_o:Nw
28655          \tex_uniformdeviate:D 268435456 ;
28656          \int_value:w
28657          \exp_after:wN \_fp_randint_split_o:Nw

```

```

28658   \tex_uniformdeviate:D 268435456 ;
28659   \int_value:w
28660   \exp_after:wN \__fp_randint_split_o:Nw
28661   \int_value:w \__fp_int_eval:w 131072 +
28662   \exp_after:wN \__fp_randint_split_o:Nw
28663   \int_value:w
28664   \__kernel_int_add:nnn {\#2} { -\#1 } { -\c_max_int } ;
28665   .
28666 }
28667 \cs_new:Npn \__fp_randint_split_o:Nw #1#2 ;
28668 {
28669   \if_meaning:w 0 #1
28670     0 \exp_after:wN ; \int_value:w 0
28671   \else:
28672     \exp_after:wN \__fp_randint_split_aux:w
28673     \int_value:w \__fp_int_eval:w (#1#2 - 8192) / 16384 ;
28674     + #1#2
28675   \fi:
28676   \exp_after:wN ;
28677 }
28678 \cs_new:Npn \__fp_randint_split_aux:w #1 ;
28679 {
28680   #1 \exp_after:wN ;
28681   \int_value:w \__fp_int_eval:w - #1 * 16384
28682 }
28683 \cs_new:Npn \__fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
28684 {
28685   \exp_after:wN \__fp_randint_wide_auxii:w
28686   \int_value:w \__fp_int_eval:w #5 * #3 + #6 * #1 +
28687   (#5 * #4 + #6 * #3 + #7 * #1 +
28688   (#5 * #2 + #7 * #3 +
28689   (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
28690   ) / 16384 \exp_after:wN ;
28691   \int_value:w \__fp_int_eval:w (#5 + #6) * 16384 + #7 ;
28692   #1 ; #5 ;
28693 }
28694 \cs_new:Npn \__fp_randint_wide_auxii:w #1; #2; #3; #4;
28695 {
28696   \if_int_odd:w 0
28697     \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
28698     \if_int_compare:w #4 = \c_zero_int 1 \fi:
28699     \if_int_compare:w #3 = 16383 ~ 1 \fi:
28700     \exp_stop_f:
28701     \exp_after:wN \prg_break:
28702   \fi:
28703   \if_int_compare:w #4 < 8 \exp_stop_f:
28704     + #4 * #3 * 16384
28705   \else:
28706     + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
28707   \fi:
28708   + #1
28709   \prg_break_point:
28710 }

```

(End definition for `\__kernel_randint:nn` and others.)

**\int\_rand:n** Similar to \int\_rand:nn, but needs fewer checks.

```
28711      \cs_new:Npn \int_rand:n #1
28712      {
28713          \int_eval:n
28714              { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
28715      }
28716      \cs_new:Npn \__fp_randint:n #1
28717      {
28718          \if_int_compare:w #1 < \c_one_int
28719              \msg_expandable_error:nnnn
28720                  { kernel } { randint-backward-range } { 1 } { #1 }
28721              \__fp_randint:ww #1; 1;
28722          \else:
28723              \if_int_compare:w #1 > \c_kernel_randint_max_int
28724                  \__kernel_randint:nn { 1 } {#1}
28725              \else:
28726                  \__kernel_randint:n {#1}
28727              \fi:
28728          \fi:
28729      }
```

(End definition for \int\_rand:n and \\_\_fp\_randint:n. This function is documented on page 166.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

```
28730      }
28731  
```

# Chapter 78

## I3fparray implementation

```
28732  <*package>
```

```
28733  <@@=fp>
```

In analogy to I3intarray it would make sense to have `<@@=fparray>`, but we need direct access to `\_fp_parse:n` from I3fp-parse, and a few other (less crucial) internals of the I3fp family.

### 78.1 Allocating arrays

There are somewhat more than  $(2^{31} - 1)^2$  floating point numbers so we store each floating point number as three entries in integer arrays. To avoid having to multiply indices by three or to add 1 etc, a floating point array is just a token list consisting of three tokens: integer arrays of the same size.

`\g_fp_array_int` Used to generate unique names for the three integer arrays.

```
28734 \int_new:N \g_fp_array_int
```

*(End definition for \g\_fp\_array\_int.)*

`\l_fp_array_loop_int` Used to loop in `\_fp_array_gzero:N`.

```
28735 \int_new:N \l_fp_array_loop_int
```

*(End definition for \l\_fp\_array\_loop\_int.)*

`\fparray_new:Nn` Build a three-token token list, then define all three tokens to be integer arrays of the same size. No need to initialize the data: the integer arrays start with zeros, and three zeros denote precisely `\c_zero_fp`, as we want.

```
28736 \cs_new_protected:Npn \fparray_new:Nn #1#2
28737  {
28738  \tl_new:N #1
28739  \prg_replicate:nn { 3 }
28740  {
28741  \int_gincr:N \g_fp_array_int
28742  \exp_args:NNc \tl_gput_right:Nn #1
28743  { g_fp_array_ \_fp_int_to_roman:w \g_fp_array_int _intarray }
28744  }
28745  \exp_last_unbraced:Nfo \_fp_array_new:nNNNN
28746  { \int_eval:n {#2} } #1 #1
```

```

28747      }
28748 \cs_generate_variant:Nn \fparray_new:Nn { c }
28749 \cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
28750 {
28751     \int_compare:nNnTF {#1} < 0
28752     {
28753         \msg_error:nnn { kernel } { negative-array-size } {#1}
28754         \cs_undefine:N #1
28755         \int_gsub:Nn \g__fp_array_int { 3 }
28756     }
28757     {
28758         \intarray_new:Nn #2 {#1}
28759         \intarray_new:Nn #3 {#1}
28760         \intarray_new:Nn #4 {#1}
28761     }
28762 }

```

(End definition for `\fparray_new:Nn` and `\__fp_array_new:nNNN`. This function is documented on page 261.)

**\fparray\_count:N** Size of any of the intarrays, here we pick the third.

```

\fparray_count:c
28763 \cs_new:Npn \fparray_count:N #1
28764 {
28765     \exp_after:wN \use_i:nnn
28766     \exp_after:wN \intarray_count:N #1
28767 }
28768 \cs_generate_variant:Nn \fparray_count:N { c }

```

(End definition for `\fparray_count:N`. This function is documented on page 261.)

## 78.2 Array items

See the `\intarray` analogue: only names change. The functions `\fparray_gset:Nnn` and `\fparray_item:Nn` share bounds checking. The T branch is used if #3 is within bounds of the array #2.

```

\__fp_array_bounds:NNnTF
\__fp_array_bounds_error:NNn
28769 \cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
28770 {
28771     \if_int_compare:w 1 > #3 \exp_stop_f:
28772         \__fp_array_bounds_error:NNn #1 #2 {#3}
28773         #5
28774     \else:
28775         \if_int_compare:w #3 > \fparray_count:N #2 \exp_stop_f:
28776             \__fp_array_bounds_error:NNn #1 #2 {#3}
28777             #5
28778         \else:
28779             #4
28780         \fi:
28781     \fi:
28782 }
28783 \cs_new:Npn \__fp_array_bounds_error:NNn #1#2#3
28784 {
28785     #1 { kernel } { out-of-bounds }
28786     { \token_to_str:N #2 } {#3} { \fparray_count:N #2 }
28787 }

```

(End definition for `\_fp_array_bounds:NnTF` and `\_fp_array_bounds_error:Nn`.)

`\fpararray_gset:Nnn`  
`\fpararray_gset:cnn`

Evaluate, then store exponent in one intarray, sign and 8 digits of mantissa in the next, and 8 trailing digits in the last.

```
28788 \cs_new_protected:Npn \fpararray_gset:Nnn #1#2#3
28789 {
2890     \exp_after:wN \exp_after:wN
2891     \exp_after:wN \_fp_array_gset:NNNNww
2892     \exp_after:wN #1
2893     \exp_after:wN #1
2894     \int_value:w \int_eval:n {#2} \exp_after:wN ;
2895     \exp:w \exp_end_continue_f:w \_fp_parse:n {#3}
2896 }
2897 \cs_generate_variant:Nn \fpararray_gset:Nnn { c }
2898 \cs_new_protected:Npn \_fp_array_gset:NNNNww #1#2#3#4#5 ; #6 ;
2899 {
2900     \_fp_array_bounds:NnTF \msg_error:nnxxx #4 {#5}
2901     {
2902         \exp_after:wN \_fp_change_func_type:NNN
2903         \_fp_use_i_until_s:nw #6 ;
2904         \_fp_array_gset:w
2905         \_fp_array_gset_recover:Nw
2906         #6 ; {#5} #1 #2 #3
2907     }
2908     {
2909 }
2910 \cs_new_protected:Npn \_fp_array_gset_recover:Nw #1#2 ;
2911 {
2912     \_fp_error:nffn { unknown-type } { \tl_to_str:n { #2 ; } } { } { }
2913     \exp_after:wN #1 \c_nan_fp
2914 }
2915 \cs_new_protected:Npn \_fp_array_gset:w \s_fp \_fp_chk:w #1#2
2916 {
2917     \if_case:w #1 \exp_stop_f:
2918         \_fp_case_return:nw { \_fp_array_gset_special:nnNNN {#2} }
2919         \or: \exp_after:wN \_fp_array_gset_normal:w
2920         \or: \_fp_case_return:nw { \_fp_array_gset_special:nnNNN { #2 3 } }
2921         \or: \_fp_case_return:nw { \_fp_array_gset_special:nnNNN { 1 } }
2922         \fi:
2923         \s_fp \_fp_chk:w #1 #2
2924     }
2925 \cs_new_protected:Npn \_fp_array_gset_normal:w
2926     \s_fp \_fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
2927 {
2928     \_kernel_intarray_gset:Nnn #7 {#6} {#2}
2929     \_kernel_intarray_gset:Nnn #8 {#6}
2930     { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
2931     \_kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
2932 }
2933 \cs_new_protected:Npn \_fp_array_gset_special:nnNNN #1#2#3#4#5
2934 {
2935     \_kernel_intarray_gset:Nnn #3 {#2} {#1}
2936     \_kernel_intarray_gset:Nnn #4 {#2} {0}
2937     \_kernel_intarray_gset:Nnn #5 {#2} {0}
```

```
28838 }
```

(End definition for `\fparray_gset:Nnn` and others. This function is documented on page 261.)

### `\fparray_gzero:N`

```
28839 \cs_new_protected:Npn \fparray_gzero:N #1
28840 {
28841     \int_zero:N \l__fp_array_loop_int
28842     \prg_replicate:nn { \fparray_count:N #1 }
28843     {
28844         \int_incr:N \l__fp_array_loop_int
28845         \exp_after:wN \__fp_array_gset_special:nnNNN
28846         \exp_after:wN 0
28847         \exp_after:wN \l__fp_array_loop_int
28848         #1
28849     }
28850 }
28851 \cs_generate_variant:Nn \fparray_gzero:N { c }
```

(End definition for `\fparray_gzero:N`. This function is documented on page 261.)

### `\fparray_item:Nn`

`\fparray_item:cn`

### `\fparray_item_to_tl:Nn`

`\fparray_item_to_tl:cn`

`\__fp_array_item:NwN`

`\__fp_array_item>NNNNN`

`\__fp_array_item:N`

`\__fp_array_item:w`

`\__fp_array_item_special:w`

`\__fp_array_item_normal:w`

```
28852 \cs_new:Npn \fparray_item:Nn #1#2
28853 {
28854     \exp_after:wN \__fp_array_item:NwN
28855     \exp_after:wN #1
28856     \int_value:w \int_eval:n {#2} ;
28857     \__fp_to_decimal:w
28858 }
28859 \cs_generate_variant:Nn \fparray_item:Nn { c }
28860 \cs_new:Npn \fparray_item_to_tl:Nn #1#2
28861 {
28862     \exp_after:wN \__fp_array_item:NwN
28863     \exp_after:wN #1
28864     \int_value:w \int_eval:n {#2} ;
28865     \__fp_to_tl:w
28866 }
28867 \cs_generate_variant:Nn \fparray_item_to_tl:Nn { c }
28868 \cs_new:Npn \__fp_array_item:NwN #1#2 ; #3
28869 {
28870     \__fp_array_bounds>NNnTF \msg_expandable_error:nnffff #1 {#2}
28871     { \exp_after:wN \__fp_array_item>NNNNN #1 {#2} #3 }
28872     { \exp_after:wN #3 \c_nan_fp }
28873 }
28874 \cs_new:Npn \__fp_array_item>NNNNN #1#2#3#4
28875 {
28876     \exp_after:wN \__fp_array_item:N
28877     \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
28878     \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
28879     \int_value:w \__kernel_intarray_item:Nn #1 {#4} ;
28880 }
28881 \cs_new:Npn \__fp_array_item:N #1
28882 {
28883     \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
28884     \__fp_array_item:w #1
```

```

28885      }
28886 \cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
28887  {
28888    \exp_after:wN \__fp_array_item_normal:w
28889    \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
28900    #7 ; {#2#3#4#5} {#6} ;
28901  }
28902 \cs_new:Npn \__fp_array_item_special:w #1 ; #2 ; #3 ; #4
28903  {
28904    \exp_after:wN #4
28905    \exp:w \exp_end_continue_f:w
28906    \if_case:w #3 \exp_stop_f:
28907      \exp_after:wN \c_zero_fp
28908      \or: \exp_after:wN \c_nan_fp
28909      \or: \exp_after:wN \c_minus_zero_fp
28910      \or: \exp_after:wN \c_inf_fp
28911      \else: \exp_after:wN \c_minus_inf_fp
28912      \fi:
28913  }
28914 \cs_new:Npn \__fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
28915  { #9 \s_fp \__fp_chk:w 1 #1 {#8} #7 {#2#3#4#5} {#6} ; }

```

(End definition for `\fparray_item:Nn` and others. These functions are documented on page 261.)

```
28906 </package>
```

# Chapter 79

## I3cctab implementation

28907 `<*package>`

28908 `<@@=cctab>`

As LuaTeX offers engine support for category code tables, and this is entirely lacking from the other engines, we need two complementary approaches. (Some future X<sub>E</sub>T<sub>E</sub>X may add support, at which point the conditionals below would be different.)

### 79.1 Variables

`\g__cctab_stack_seq` List of catcode tables saved by nested `\cctab_begin:N`, to restore catcodes at the matching `\cctab_end:.`. When popped from the `\g__cctab_stack_seq` the table numbers are stored in `\g__cctab_unused_seq` for later reuse.

28909 `\seq_new:N \g__cctab_stack_seq`  
28910 `\seq_new:N \g__cctab_unused_seq`

(End definition for `\g__cctab_stack_seq` and `\g__cctab_unused_seq`.)

`\g__cctab_group_seq` A stack to store the group level when a catcode table started.

28911 `\seq_new:N \g__cctab_group_seq`

(End definition for `\g__cctab_group_seq`.)

`\g__cctab_allocate_int` Integer to keep track of what category code table to allocate. In LuaTeX it is only used in format mode to implement `\cctab_new:N`. In other engines it is used to make csnames for dynamic tables.

28912 `\int_new:N \g__cctab_allocate_int`

(End definition for `\g__cctab_allocate_int`.)

`\l__cctab_internal_a_t1` Scratch space. For instance, when popping `\g__cctab_stack_seq/\g__cctab_unused_seq`, consists of the catcodetable number (integer denotation) in LuaTeX, or of an intarray variable (as a single token) in other engines.

28913 `\tl_new:N \l__cctab_internal_a_t1`  
28914 `\tl_new:N \l__cctab_internal_b_t1`

(End definition for `\l__cctab_internal_a_t1` and `\l__cctab_internal_b_t1`.)

\g\_\_cctab\_endlinechar\_prop In LuaTeX we store the \endlinechar associated to each \catcodetable in a property list, unless it is the default value 13.

```
28915 \prop_new:N \g__cctab_endlinechar_prop
```

*(End definition for \g\_\_cctab\_endlinechar\_prop.)*

## 79.2 Allocating category code tables

\cctab\_new:N  
\cctab\_new:c  
\\_\_cctab\_new:N  
\\_\_cctab\_gstore:Nnn

The \\_\_cctab\_new:N auxiliary allocates a new catcode table but does not attempt to set its value consistently across engines. It is used both in \cctab\_new:N, which sets catcodes to iniTeX values, and in \cctab\_begin:N/\cctab\_end: for dynamically allocated tables.

First, the LuaTeX case. Creating a new category code table is done like other registers. In ConTeXt, \newcatcodetable does not include the initialisation, so that is added explicitly.

```
28916 \sys_if_engine_luatex:TF
28917 {
28918   \cs_new_protected:Npn \cctab_new:N #1
28919   {
28920     \__kernel_chk_if_free_cs:N #1
28921     \__cctab_new:N #1
28922   }
28923   \cs_new_protected:Npn \__cctab_new:N #1
28924   {
28925     \newcatcodetable #1
28926     \tex_initcatcodetable:D #1
28927   }
28928 }
```

Now the case for other engines. Here, each table is an integer array. Following the LuaTeX pattern, a new table starts with iniTeX codes. The index base is out-by-one, so we have an internal function to handle that. The iniTeX \endlinechar is 13.

```
28929 {
28930   \cs_new_protected:Npn \__cctab_new:N #1
28931   {
28932     \intarray_new:Nn #1 { 257 } }
28933   \cs_new_protected:Npn \__cctab_gstore:Nnn #1#2#3
28934   {
28935     \intarray_gset:Nnn #1 { \int_eval:n { #2 + 1 } } { #3 } }
28936   \cs_new_protected:Npn \cctab_new:N #1
28937   {
28938     \__kernel_chk_if_free_cs:N #1
28939     \__cctab_new:N #1
28940     \int_step_inline:nn { 256 }
28941       { \__kernel_intarray_gset:Nnn #1 {##1} { 12 } }
28942     \__kernel_intarray_gset:Nnn #1 { 257 } { 13 }
28943     \__cctab_gstore:Nnn #1 { 0 } { 9 }
28944     \__cctab_gstore:Nnn #1 { 13 } { 5 }
28945     \__cctab_gstore:Nnn #1 { 32 } { 10 }
28946     \__cctab_gstore:Nnn #1 { 37 } { 14 }
28947     \int_step_inline:nnn { 65 } { 90 }
28948       { \__cctab_gstore:Nnn #1 {##1} { 11 } }
28949     \__cctab_gstore:Nnn #1 { 92 } { 0 }
28950     \int_step_inline:nnn { 97 } { 122 }
28951       { \__cctab_gstore:Nnn #1 {##1} { 11 } }
28952     \__cctab_gstore:Nnn #1 { 127 } { 15 }
```

```

28951     }
28952   }
28953 \cs_generate_variant:Nn \cctab_new:N { c }

```

(End definition for `\cctab_new:N`, `\_cctab_new:N`, and `\_cctab_gstore:Nnn`. This function is documented on page 262.)

### 79.3 Saving category code tables

`\_cctab_gset:n` In various functions we need to save the current catcodes (globally) in a table. In LuaTeX, saving the catcodes is a primitives, but the `\endlinechar` needs more work: to avoid filling `\g_cctab_endlinechar_prop` with many entries we special-case the default value 13. In other engines we store 256 current catcodes and the `\endlinechar` in an intarray variable.

```

28954 \sys_if_engine_luatex:TF
28955   {
28956     \cs_new_protected:Npn \_cctab_gset:n #1
28957       { \exp_args:Nf \_cctab_gset_aux:n { \int_eval:n {#1} } }
28958     \cs_new_protected:Npn \_cctab_gset_aux:n #1
28959       {
28960         \tex_savecatcodetable:D #1 \scan_stop:
28961         \int_compare:nNnTF { \tex_endlinechar:D } = { 13 }
28962           { \prop_gremove:Nn \g_cctab_endlinechar_prop {#1} }
28963           {
28964             \prop_gput:NnV \g_cctab_endlinechar_prop {#1}
28965               \tex_endlinechar:D
28966             }
28967           }
28968       }
28969     {
28970       \cs_new_protected:Npn \_cctab_gset:n #1
28971         {
28972           \int_step_inline:nn { 256 }
28973             {
28974               \_kernel_intarray_gset:Nnn #1 {##1}
28975                 { \char_value_catcode:n { ##1 - 1 } }
28976             }
28977             \_kernel_intarray_gset:Nnn #1 { 257 }
28978               { \tex_endlinechar:D }
28979             }
28980           }

```

(End definition for `\_cctab_gset:n` and `\_cctab_gset_aux:n`.)

`\cctab_gset:Nn` Category code tables are always global, so only one version of assignments is needed.  
`\cctab_gset:cn` Simply run the setup in a group and save the result in a category code table #1, provided it is valid. The internal function is defined above depending on the engine.

```

28981 \cs_new_protected:Npn \cctab_gset:Nn #1#2
28982   {
28983     \_cctab_chk_if_valid:NT #1
28984       {
28985         \group_begin:
28986           \cctab_select:N \c_initex_cctab

```

```

28987      #2 \scan_stop:
28988      \__cctab_gset:N {#1}
28989      \group_end:
28990    }
28991  }
28992 \cs_generate_variant:Nn \cctab_gset:Nn { c }

```

(End definition for `\cctab_gset:Nn`. This function is documented on page 262.)

## 79.4 Using category code tables

In LuaTeX, we must ensure that the saved tables are read-only. This is done by applying the saved table, then switching immediately to a scratch table. Any later catcode assignment will affect that scratch table rather than the saved one. If we simply switched to the saved tables, then `\char_set_catcode_other:N` in the example below would change `\c_document_cctab` and a later use of that table would give the wrong category code to `_`.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
    \char_set_catcode_other:N \
  \cctab_end:
  \cctab_begin:N \c_document_cctab
    \int_compare:nTF { \char_value_catcode:n { '_ } = 8 }
      { \TRUE } { \ERROR }
  \cctab_end:
}

```

We must also make sure that a scratch table is never reused in a nested group: in the following example, the scratch table used by the first `\cctab_begin:N` would be changed globally by the second one issuing `\savecatcodetable`, and after `\group_end:` the wrong category codes (those of `\c_str_cctab`) would be imposed. Note that the inner `\cctab_end:` restores the correct catcodes only locally, so the problem really comes up because of the different grouping level. The simplest is to use a scratch table labeled by the `\currentgrouplevel`. We initialize one of them as an example.

```

\use:n
{
  \cctab_begin:N \c_document_cctab
  \group_begin:
    \cctab_begin:N \c_str_cctab
    \cctab_end:
  \group_end:
  \cctab_end:
}

28993 \sys_if_engine_luatex:T
28994  {
28995    \__cctab_new:N \g__cctab_internal_cctab
28996    \cs_new:Npn \__cctab_internal_cctab_name:
      {

```

```

28998     g__cctab_internal
28999     \tex_roman numeral:D \tex_currentgroup_level:D
29000     _cctab
29001   }
29002 }

```

(End definition for `\g__cctab_internal_cctab` and `\__cctab_internal_cctab_name`.)

`\cctab_select:N`  
`\cctab_select:c`  
`\__cctab_select:N`

The public function simply checks the `\cctab var` exists before using the engine-dependent `\__cctab_select:N`. Skipping these checks would result in low-level engine-dependent errors. First, the LuaTeX case. In other engines, selecting a catcode table is a matter of doing 256 catcode assignments and setting the `\endlinechar`.

```

29003 \cs_new_protected:Npn \cctab_select:N #1
29004   { \__cctab_chk_if_valid:NT #1 { \__cctab_select:N #1 } }
29005 \cs_generate_variant:Nn \cctab_select:N { c }
29006 \sys_if_engine_luatex:TF
29007   {
29008     \cs_new_protected:Npn \__cctab_select:N #1
29009       {
29010         \tex_catcodetable:D #1
29011         \prop_get:NVNTF \g__cctab_endlinechar_prop #1 \l__cctab_internal_a_tl
29012           { \int_set:Nn \tex_endlinechar:D { \l__cctab_internal_a_tl } }
29013           { \int_set:Nn \tex_endlinechar:D { 13 } }
29014         \cs_if_exist:cF { \__cctab_internal_cctab_name: }
29015           { \exp_args:Nc \__cctab_new:N { \__cctab_internal_cctab_name: } }
29016         \exp_args:Nc \tex_savecatcodetable:D { \__cctab_internal_cctab_name: }
29017         \exp_args:Nc \tex_catcodetable:D { \__cctab_internal_cctab_name: }
29018       }
29019     }
29020   {
29021     \cs_new_protected:Npn \__cctab_select:N #1
29022       {
29023         \int_step_inline:nn { 256 }
29024           {
29025             \char_set_catcode:nn { ##1 - 1 }
29026               { \__kernel_intarray_item:Nn #1 {##1} }
29027           }
29028           \int_set:Nn \tex_endlinechar:D
29029             { \__kernel_intarray_item:Nn #1 { 257 } }
29030       }
29031   }

```

(End definition for `\cctab_select:N` and `\__cctab_select:N`. This function is documented on page 263.)

`\g__cctab_next_cctab`  
`\__cctab_begin_aux:`

For `\cctab_begin:N/\cctab_end:` we will need to allocate dynamic tables. This is done here by `\__cctab_begin_aux:`, which puts a table number (in LuaTeX) or name (in other engines) into `\l__cctab_internal_a_tl`. In LuaTeX this simply calls `\__cctab_new:N` and uses the resulting catcodetable number; in other engines we need to give a name to the intarray variable and use that. In LuaTeX, to restore catcodes at `\cctab_end:` we cannot just set `\catcodetable` to its value before `\cctab_begin:N`, because that table may have been altered by other code in the mean time. So we must make sure to save the catcodes in a table we control and restore them at `\cctab_end::`

```
29032 \sys_if_engine_luatex:TF
```

```

29033   {
29034     \cs_new_protected:Npn \__cctab_begin_aux:
29035     {
29036       \__cctab_new:N \g__cctab_next_cctab
29037       \tl_set:NV \l__cctab_internal_a_tl \g__cctab_next_cctab
29038       \cs_undefine:N \g__cctab_next_cctab
29039     }
29040   }
29041   {
29042     \cs_new_protected:Npn \__cctab_begin_aux:
29043     {
29044       \int_gincr:N \g__cctab_allocate_int
29045       \exp_args:Nc \__cctab_new:N
29046         { \g__cctab_\int_use:N \g__cctab_allocate_int _cctab }
29047       \exp_args:NNc \tl_set:Nn \l__cctab_internal_a_tl
29048         { \g__cctab_\int_use:N \g__cctab_allocate_int _cctab }
29049     }
29050   }

```

(End definition for `\g__cctab_next_cctab` and `\__cctab_begin_aux`.)

**\cctab\_begin:N** Check the `\cctab var` exists, to avoid low-level errors. Get in `\l__cctab_internal_a_tl` the number/name of a dynamic table, either from `\g__cctab_unused_seq` where we save tables that are not currently in use, or from `\__cctab_begin_aux`: if none are available. Then save the current catcodes into the table (pointed to by) `\l__cctab_internal_a_tl` and save that table number in a stack before selecting the desired catcodes.

```

29051   \cs_new_protected:Npn \cctab_begin:N #
29052   {
29053     \__cctab_chk_if_valid:NT #1
29054     {
29055       \seq_gpop:NNF \g__cctab_unused_seq \l__cctab_internal_a_tl
29056         { \__cctab_begin_aux: }
29057       \exp_args:Nx \__cctab_chk_group_begin:n
29058         { \__cctab_nesting_number:N \l__cctab_internal_a_tl }
29059       \seq_gpush:NV \g__cctab_stack_seq \l__cctab_internal_a_tl
29060       \exp_args:NV \__cctab_gset:n \l__cctab_internal_a_tl
29061       \__cctab_select:N #1
29062     }
29063   }
29064   \cs_generate_variant:Nn \cctab_begin:N { c }

```

(End definition for `\cctab_begin:N`. This function is documented on page [262](#).)

**\cctab\_end:** Make sure a `\cctab_begin:N` was used some time earlier, get in `\l__cctab_internal_a_tl` the catcode table number/name in which the prevailing catcodes were stored, then restore these catcodes. The dynamic table is now unused hence stored in `\g__cctab_unused_seq` for recycling by later `\cctab_begin:N`.

```

29065   \cs_new_protected:Npn \cctab_end:
29066   {
29067     \seq_gpop:NNTF \g__cctab_stack_seq \l__cctab_internal_a_tl
29068     {
29069       \seq_gpush:NV \g__cctab_unused_seq \l__cctab_internal_a_tl
29070       \exp_args:Nx \__cctab_chk_group_end:n
29071         { \__cctab_nesting_number:N \l__cctab_internal_a_tl }

```

```

29072      \__cctab_select:N \l__cctab_internal_a_tl
29073  }
29074  { \msg_error:nn { cctab } { extra-end } }
29075 }

```

(End definition for `\cctab_end:`. This function is documented on page 263.)

`\__cctab_chk_group_begin:n`  
`\__cctab_chk_group_end:n`

Catcode tables are not allowed to be intermixed with groups, so here we check that they are properly nested regarding TeX groups. `\__cctab_chk_group_begin:n` stores the current group level in a stack, and locally defines a dummy control sequence `\__cctab_group_{cctab-level}_chk:`.

`\__cctab_chk_group_end:n` pops the stack, and compares the returned value with `\tex_currentgrouplevel:D`. If they differ, `\cctab_end:` is in a different grouping level than the matching `\cctab_begin:N`. If they are the same, both happened at the same level, however a group might have ended and another started between `\cctab_begin:N` and `\cctab_end::`:

```

\group_begin:
  \cctab_begin:N \c_document_cctab
\group_end:
\group_begin:
  \cctab_end:
\group_end:

```

In this case checking `\tex_currentgrouplevel:D` is not enough, so we locally define `\__cctab_group_{cctab-level}_chk:`, and then check if it exist in `\cctab_end::`. If it doesn't, we know there was a group end where it shouldn't.

The `\cctab_level` in the sentinel macro above cannot be replaced by the more convenient `\tex_currentgrouplevel:D` because with the latter we might be tricked. Suppose:

```

\group_begin:
  \cctab_begin:N \c_code_cctab % A
\group_end:
\group_begin:
  \cctab_begin:N \c_code_cctab % B
  \cctab_end: % C
  \cctab_end: % D
\group_end:

```

The line marked with A would start a `cctab` with a sentinel token named `\__cctab_group_1_chk:`, which would disappear at the `\group_end:` that follows. But B would create the same sentinel token, since both are at the same group level. Line C would end the `cctab` from line B correctly, but so would line D because line B created the same sentinel token. Using `\cctab_level` works correctly because it signals that certain `cctab` level was activated somewhere, but if it doesn't exist when the `\cctab_end:` is reached, we had a problem.

Unfortunately these tests only flag the wrong usage at the `\cctab_end:`, which might be far from the `\cctab_begin:N`. However it isn't possible to signal the wrong usage at the `\group_end:` without using `\tex_aftergroup:D`, which is unsafe in certain types of groups.

The three cases checked here just raise an error, and no recovery is attempted: usually interleaving groups and catcode tables will work predictably.

```

29076 \cs_new_protected:Npn \__cctab_chk_group_begin:n #1
29077 {
29078     \seq_gpush:Nx \g__cctab_group_seq
29079         { \int_use:N \tex_currentgrouplevel:D }
29080     \cs_set_eq:cN { __cctab_group_ #1 _chk: } \prg_do_nothing:
29081 }
29082 \cs_new_protected:Npn \__cctab_chk_group_end:n #1
29083 {
29084     \seq_gpop>NN \g__cctab_group_seq \l__cctab_internal_b_tl
29085     \bool_lazy_and:nnF
29086     {
29087         \int_compare_p:nNn
29088             { \tex_currentgrouplevel:D } = { \l__cctab_internal_b_tl }
29089     }
29090     { \cs_if_exist_p:c { __cctab_group_ #1 _chk: } }
29091     {
29092         \msg_error:nnx { cctab } { group-mismatch }
29093         {
29094             \int_sign:n
29095                 { \tex_currentgrouplevel:D - \l__cctab_internal_b_tl }
29096         }
29097     }
29098     \cs_undefine:c { __cctab_group_ #1 _chk: }
29099 }

```

(End definition for `\__cctab_chk_group_begin:n` and `\__cctab_chk_group_end:n`.)

`\__cctab_nesting_number:N` This macro returns the numeric index of the current catcode table. In LuaTeX this is just the argument, which is a count reference to a `\catcodetable` register. In other engines, the number is extracted from the `cctab` variable.

```

29100 \sys_if_engine_luatex:TF
29101     { \cs_new:Npn \__cctab_nesting_number:N #1 {#1} }
29102     {
29103         \cs_new:Npn \__cctab_nesting_number:N #1
29104         {
29105             \exp_after:wN \exp_after:wN \exp_after:wN \__cctab_nesting_number:w
29106                 \exp_after:wN \token_to_str:N #1
29107         }
29108         \use:x
29109         {
29110             \cs_new:Npn \exp_not:N \__cctab_nesting_number:w
29111                 ##1 \tl_to_str:n { g__cctab_ } ##2 \tl_to_str:n { _cctab } {##2}
29112         }
29113     }

```

(End definition for `\__cctab_nesting_number:N` and `\__cctab_nesting_number:w`.)

Finally, install some code at the end of the TeX run to check that all `\cctab_begin:N` were ended by some `\cctab_end:`.

```

29114 \cs_if_exist:NT \hook_gput_code:nnn
29115     {
29116         \hook_gput_code:nnn { enddocument/end } { cctab }
29117         {
29118             \seq_if_empty:NF \g__cctab_stack_seq
29119                 { \msg_error:nn { cctab } { missing-end } }

```

```

29120      }
29121  }

```

**\cctab\_item:Nn** Evaluate the integer argument only once. In most engines the `cctab` variable only has 256 entries so we only look up the catcode for these entries, otherwise we use the current catcode. In particular, for out-of-range values we use whatever fall-back `\char_value_catcode:n`. In LuaTeX, we use the `tex.getcatcode` function.

```

29122 \cs_new:Npn \cctab_item:Nn #1#2
29123   { \exp_args:Nf \__cctab_item:nN { \int_eval:n {#2} } #1 }
29124 \sys_if_engine_luatex:TF
29125   {
29126     \cs_new:Npn \__cctab_item:nN #1#2
29127       { \lua_now:e { tex.print(-2, tex.getcatcode(\int_use:N #2, #1)) } }
29128   }
29129   {
29130     \cs_new:Npn \__cctab_item:nN #1#2
29131       {
29132         \int_compare:nNnTF {#1} < { 256 }
29133           { \intarray_item:Nn #2 { #1 + 1 } }
29134           { \char_value_catcode:n {#1} }
29135       }
29136   }
29137 \cs_generate_variant:Nn \cctab_item:Nn { c }

```

(End definition for `\cctab_item:Nn`. This function is documented on page 263.)

## 79.5 Category code table conditionals

**\cctab\_if\_exist\_p:N** Checks whether a  $\langle cctab var \rangle$  is defined.

```

29138 \prg_new_eq_conditional:NNN \cctab_if_exist:N \cs_if_exist:N
29139   { TF , T , F , p }
29140 \prg_new_eq_conditional:NNN \cctab_if_exist:c \cs_if_exist:c
29141   { TF , T , F , p }

```

(End definition for `\cctab_if_exist:NTF`. This function is documented on page 263.)

**\\_\_cctab\_chk\_if\_valid:NTF** Checks whether the argument is defined and whether it is a valid  $\langle cctab var \rangle$ . In LuaTeX the validity of the  $\langle cctab var \rangle$  is checked by the engine, which complains if the argument is not a `\chardef`ed` constant. In other engines, check if the given command is an intarray variable (the underlying definition is a copy of the `cmr10` font).

```

29142 \prg_new_protected_conditional:Npnn \__cctab_chk_if_valid:N #1
29143   { TF , T , F }
29144   {
29145     \cctab_if_exist:NTF #1
29146     {
29147       \__cctab_chk_if_valid_aux:NTF #1
29148         { \prg_return_true: }
29149         {
29150           \msg_error:nnx { cctab } { invalid-cctab }
29151             { \token_to_str:N #1 }
29152           \prg_return_false:
29153         }
29154     }

```

```

29155     {
29156         \msg_error:nnx { kernel } { command-not-defined }
29157             { \token_to_str:N #1 }
29158         \prg_return_false:
29159     }
29160 }
29161 \sys_if_engine_luatex:TF
29162 {
29163     \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
29164     {
29165         \int_compare:nNnTF {#1-1} < { \e@alloc@ccodetable@count }
29166     }
29167     \cs_if_exist:NT \c_syst_catcodes_n
29168     {
29169         \cs_gset_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
29170         {
29171             \int_compare:nTF { #1 <= \c_syst_catcodes_n }
29172         }
29173     }
29174 }
29175 {
29176     \cs_new_protected:Npn \__cctab_chk_if_valid_aux:NTF #1
29177     {
29178         \exp_args:Nf \str_if_in:nnTF
29179             { \cs_meaning:N #1 }
29180             { select~font~cmr10~at~ }
29181     }
29182 }

```

(End definition for `\__cctab_chk_if_valid:NTF` and `\__cctab_chk_if_valid_aux:NTF`.)

## 79.6 Constant category code tables

`\cctab_const:Nn` Creates a new  $\langle cctab\ var \rangle$  then sets it with the current and user-supplied codes.  
`\cctab_const:cn`

```

29183 \cs_new_protected:Npn \cctab_const:Nn #1#2
29184 {
29185     \cctab_new:N #1
29186     \cctab_gset:Nn #1 {#2}
29187 }
29188 \cs_generate_variant:Nn \cctab_const:Nn { c }

```

(End definition for `\cctab_const:Nn`. This function is documented on page 262.)

`\c_initex_cctab` Creating category code tables means thinking starting from initEX. For all-other and the standard “string” tables that’s easy.  
`\c_other_cctab`  
`\c_str_cctab`

```

29189 \cctab_new:N \c_initex_cctab
29190 \cctab_const:Nn \c_other_cctab
29191 {
29192     \cctab_select:N \c_initex_cctab
29193     \int_set:Nn \tex_endlinechar:D { -1 }
29194     \int_step_inline:nnn { 0 } { 127 }
29195         { \char_set_catcode_other:n {#1} }
29196 }

```

```

29197 \cctab_const:Nn \c_str_cctab
29198 {
29199     \cctab_select:N \c_other_cctab
29200     \char_set_catcode_space:n { 32 }
29201 }
```

(End definition for `\c_initex_cctab`, `\c_other_cctab`, and `\c_str_cctab`. These variables are documented on page 263.)

`\c_code_cctab`  
`\c_document_cctab`

To pick up document-level category codes, we need to delay set up to the end of the format, where that's possible. Also, as there are a *lot* of category codes to set, we avoid using the official interface and store the document codes using internal code. Depending on whether we are in the hook or not, the catcodes may be code or document, so we explicitly set up both correctly.

```

29202 \cs_if_exist:NTF \expl@finalise@setup@@
29203 { \tl_gput_right:Nn \expl@finalise@setup@@ }
29204 { \use:n }
29205 {
29206     \__cctab_new:N \c_code_cctab
29207     \group_begin:
29208         \int_set:Nn \tex_endlinechar:D { 32 }
29209         \char_set_catcode_invalid:n { 0 }
29210         \bool_lazy_or:nnTF
29211             { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
29212             { \int_step_function:nN { 31 } \char_set_catcode_invalid:n }
29213             { \int_step_function:nN { 31 } \char_set_catcode_active:n }
29214         \int_step_function:nnN { 33 } { 64 } \char_set_catcode_other:n
29215         \int_step_function:nnN { 65 } { 90 } \char_set_catcode_letter:n
29216         \int_step_function:nnN { 91 } { 96 } \char_set_catcode_other:n
29217         \int_step_function:nnN { 97 } { 122 } \char_set_catcode_letter:n
29218         \char_set_catcode_ignore:n { 9 } % tab
29219         \char_set_catcode_other:n { 10 } % lf
29220         \char_set_catcode_active:n { 12 } % ff
29221         \char_set_catcode_end_line:n { 13 } % cr
29222         \char_set_catcode_ignore:n { 32 } % space
29223         \char_set_catcode_parameter:n { 35 } % hash
29224         \char_set_catcode_math_toggle:n { 36 } % dollar
29225         \char_set_catcode_comment:n { 37 } % percent
29226         \char_set_catcode_alignment:n { 38 } % ampersand
29227         \char_set_catcode_letter:n { 58 } % colon
29228         \char_set_catcode_escape:n { 92 } % backslash
29229         \char_set_catcode_math_superscript:n { 94 } % circumflex
29230         \char_set_catcode_letter:n { 95 } % underscore
29231         \char_set_catcode_group_begin:n { 123 } % left brace
29232         \char_set_catcode_other:n { 124 } % pipe
29233         \char_set_catcode_group_end:n { 125 } % right brace
29234         \char_set_catcode_space:n { 126 } % tilde
29235         \char_set_catcode_invalid:n { 127 } % ^^?
29236         \bool_lazy_or:nnF
29237             { \sys_if_engine_xetex_p: } { \sys_if_engine_luatex_p: }
29238             { \int_step_function:nnN { 128 } { 255 } \char_set_catcode_active:n }
29239         \__cctab_gset:n { \c_code_cctab }
29240     \group_end:
29241     \cctab_const:Nn \c_document_cctab
```

```

29242 {
29243   \cctab_select:N \c_code_cctab
29244   \int_set:Nn \tex_endlinechar:D { 13 }
29245   \char_set_catcode_space:n           { 9 }
29246   \char_set_catcode_space:n           { 32 }
29247   \char_set_catcode_other:n          { 58 }
29248   \char_set_catcode_math_subscript:n { 95 }
29249   \char_set_catcode_active:n         { 126 }
29250 }
29251 }

```

(End definition for `\c_code_cctab` and `\c_document_cctab`. These variables are documented on page 263.)

## 79.7 Messages

```

29252 \msg_new:nnnn { cctab } { stack-full }
29253   { The~category~code~table~stack~is~exhausted. }
29254   {
29255     LaTeX~has~been~asked~to~switch~to~a~new~category~code~table,~
29256     but~there~is~no~more~space~to~do~this!
29257   }
29258 \msg_new:nnnn { cctab } { extra-end }
29259   { Extra~\iow_char:N\\cctab_end:~ignored~\msg_line_context:. }
29260   {
29261     LaTeX~came~across~a~\iow_char:N\\cctab_end:~without~a~matching~
29262     \iow_char:N\\cctab_begin:N.~This~command~will~be~ignored.
29263   }
29264 \msg_new:nnnn { cctab } { missing-end }
29265   { Missing~\iow_char:N\\cctab_end:~before~end~of~TeX~run. }
29266   {
29267     LaTeX~came~across~more~\iow_char:N\\cctab_begin:N~than~
29268     \iow_char:N\\cctab_end:.
29269   }
29270 \msg_new:nnnn { cctab } { invalid-cctab }
29271   { Invalid~\iow_char:N\\catcode~table. }
29272   {
29273     You~can~only~switch~to~a~\iow_char:N\\catcode~table~that~is~
29274     initialized~using~\iow_char:N\\cctab_new:N~or~
29275     \iow_char:N\\cctab_const:Nn.
29276   }
29277 \msg_new:nnnn { cctab } { group-mismatch }
29278   {
29279     \iow_char:N\\cctab_end:~occurred~in~a~
29280     \int_case:nn {#1}
29281     {
29282       { 0 } { different-group }
29283       { 1 } { higher-group-level }
29284       { -1 } { lower-group-level }
29285     } ~than~
29286     the~matching~\iow_char:N\\cctab_begin:N.
29287   }
29288   {
29289     Catcode~tables~and~groups~must~be~properly~nested,~but~

```

```
29290      you~tried~to~interleave~them.~LaTeX~will~try~to~proceed,~  
29291      but~results~may~be~unexpected.  
29292  }  
29293 \prop_gput:Nnn \g_msg_module_name_prop { cctab } { LaTeX }  
29294 \prop_gput:Nnn \g_msg_module_type_prop { cctab } { }  
29295 </package>
```

# Chapter 80

## I3unicode implementation

```
29296  <*package>
29297  <@@=codepoint>
```

### 80.1 User functions

`\codepoint_str_generate:n`  
  `\_codepoint_str_generate:nnn`  
`\codepoint_generate:nn`  
`\__codepoint_generate:nnnn`  
`\__codepoint_generate:n`

Conversion of a codepoint to a character (Unicode engines) or to one or more bytes (8-bit engines) is required. For loading the data, all that is needed is the form which creates strings: these are outside the group as they will also be used when looking up data in the hash table storage at point-of-use. Later, we will also need functions that can generate character tokens for document use: those are defined below, in the data recovery setup.

```
29298  \bool_lazy_or:nnTF
29299  { \sys_if_engine_luatex_p: }
29300  { \sys_if_engine_xetex_p: }
29301  {
29302  \cs_new:Npn \codepoint_str_generate:n #1
29303  {
29304  \int_compare:nNnTF {#1} = { '\ }
29305  { ~ }
29306  { \char_generate:nn {#1} { 12 } }
29307  }
29308  \cs_new:Npn \codepoint_generate:nn #1#2
29309  {
29310  \int_compare:nNnTF {#1} = { '\ }
29311  { ~ }
29312  {
29313  \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
29314  { \char_generate:nn {#1} {#2} }
29315  }
29316  }
29317  }
29318  {
29319  \cs_new:Npn \codepoint_str_generate:n #1
29320  {
29321  \int_compare:nNnTF {#1} = { '\ }
29322  { ~ }
29323  {
29324  \use:e
```

```

29325      {
29326          \exp_not:N \__codepoint_str_generate:nnnn
29327              \__kernel_codepoint_to_bytes:n {#1}
29328      }
29329  }
29330
29331 \cs_new:Npn \__codepoint_str_generate:nnnn #1#2#3#4
29332  {
29333      \char_generate:nn {#1} { 12 }
29334      \tl_if_blank:nF {#2}
29335      {
29336          \char_generate:nn {#2} { 12 }
29337          \tl_if_blank:nF {#3}
29338          {
29339              \char_generate:nn {#3} { 12 }
29340              \tl_if_blank:nF {#4}
29341                  { \char_generate:nn {#4} { 12 } }
29342          }
29343      }
29344  }
29345 \cs_new:Npn \codepoint_generate:nn #1#2
29346  {
29347      \int_compare:nNnTF {#1} = { '\ }
29348          { ~ }
29349          {
29350              \int_compare:nNnTF {#1} < { "80 }
29351          {
29352              \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
29353                  { \char_generate:nn {#1} {#2} }
29354          }
29355          {
29356              \use:e
29357                  {
29358                      \exp_not:N \__codepoint_generate:nnnn
29359                          \__kernel_codepoint_to_bytes:n {#1}
29360                  }
29361          }
29362      }
29363  }
29364 \cs_new:Npn \__codepoint_generate:nnnn #1#2#3#4
29365  {
29366      \__kernel_exp_not:w \exp_after:wN
29367          {
29368              \tex_expanded:D
29369                  {
29370                      \__codepoint_generate:n {#1}
29371                      \__codepoint_generate:n {#2}
29372                      \tl_if_blank:nF {#3}
29373                      {
29374                          \__codepoint_generate:n {#3}
29375                          \tl_if_blank:nF {#4}
29376                              { \__codepoint_generate:n {#4} }
29377                      }
29378      }

```

```

29379     }
29380   }
29381 \cs_new:Npn \__codepoint_generate:n #1
29382   {
29383     \__kernel_exp_not:w \exp_after:wN \exp_after:wN \exp_after:wN
29384       { \char_generate:nn {#1} { 13 } }
29385   }
29386 }
```

(End definition for `\codepoint_str_generate:n` and others. These functions are documented on page 266.)

This code converts a codepoint into the correct UTF-8 representation. In terms of the algorithm itself, see <https://en.wikipedia.org/wiki/UTF-8> for the octet pattern.

```

\kernel_codepoint_to_bytes:n
\__codepoint_to_bytes_auxi:n
\__codepoint_to_bytes_auxii:n
\__codepoint_to_bytes_outputi:nw
\__codepoint_to_bytes_outputii:nw
\__codepoint_to_bytes_outputiii:nw
\__codepoint_to_bytes_outputiv:nw
\__codepoint_to_bytes_outputv:nm
\__codepoint_to_bytes_output:fnn
\__codepoint_to_bytes_end:
```

```

29387 \cs_new:Npn \__kernel_codepoint_to_bytes:n #1
29388   {
29389     \exp_args:Nf \__codepoint_to_bytes_auxi:n
29390       { \int_eval:n {#1} }
29391   }
29392 \cs_new:Npn \__codepoint_to_bytes_auxi:n #1
29393   {
29394     \if_int_compare:w #1 > "80 \exp_stop_f:
29395       \if_int_compare:w #1 < "800 \exp_stop_f:
29396         \__codepoint_to_bytes_outputi:nw
29397           { \__codepoint_to_bytes_auxii:Nnn C {#1} { 64 } }
29398         \__codepoint_to_bytes_outputii:nw
29399           { \__codepoint_to_bytes_auxiii:n {#1} }
29400     \else:
29401       \if_int_compare:w #1 < "10000 \exp_stop_f:
29402         \__codepoint_to_bytes_outputi:nw
29403           { \__codepoint_to_bytes_auxii:Nnn E {#1} { 64 * 64 } }
29404         \__codepoint_to_bytes_outputii:nw
29405           {
29406             \__codepoint_to_bytes_auxiii:n
29407               { \int_div_truncate:nn {#1} { 64 } }
29408           }
29409         \__codepoint_to_bytes_outputiii:nw
29410           { \__codepoint_to_bytes_auxiii:n {#1} }
29411     \else:
29412       \__codepoint_to_bytes_outputi:nw
29413       {
29414         \__codepoint_to_bytes_auxii:Nnn F
29415           {#1} { 64 * 64 * 64 }
29416       }
29417     \__codepoint_to_bytes_outputii:nw
29418     {
29419       \__codepoint_to_bytes_auxiii:n
29420         { \int_div_truncate:nn {#1} { 64 * 64 } }
29421     }
29422     \__codepoint_to_bytes_outputiii:nw
29423     {
29424       \__codepoint_to_bytes_auxiii:n
29425         { \int_div_truncate:nn {#1} { 64 } }
29426     }
```

```

29427     \__codepoint_to_bytes_outputiv:nw
29428         { \__codepoint_to_bytes_auxiii:n {#1} }
29429     \fi:
29430     \fi:
29431     \else:
29432         \__codepoint_to_bytes_outputi:nw {#1}
29433     \fi:
29434     \__codepoint_to_bytes_end: { } { } { } { }
29435   }
29436 \cs_new:Npn \__codepoint_to_bytes_auxii:Nnn #1#2#3
29437   { "#10 + \int_div_truncate:nn {#2} {#3} }
29438 \cs_new:Npn \__codepoint_to_bytes_auxiii:n #1
29439   { \int_mod:nn {#1} { 64 } + 128 }
29440 \cs_new:Npn \__codepoint_to_bytes_outputi:nw
29441   #1 #2 \__codepoint_to_bytes_end: #3
29442   { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
29443 \cs_new:Npn \__codepoint_to_bytes_outputii:nw
29444   #1 #2 \__codepoint_to_bytes_end: #3#4
29445   { \__codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
29446 \cs_new:Npn \__codepoint_to_bytes_outputiii:nw
29447   #1 #2 \__codepoint_to_bytes_end: #3#4#5
29448   {
29449     \__codepoint_to_bytes_output:fnn
29450       { \int_eval:n {#1} } { {#3} {#4} } {#2}
29451   }
29452 \cs_new:Npn \__codepoint_to_bytes_outputiv:nw
29453   #1 #2 \__codepoint_to_bytes_end: #3#4#5#6
29454   {
29455     \__codepoint_to_bytes_output:fnn
29456       { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
29457   }
29458 \cs_new:Npn \__codepoint_to_bytes_output:nnn #1#2#3
29459   {
29460     #3
29461     \__codepoint_to_bytes_end: #2 {#1}
29462   }
29463 \cs_generate_variant:Nn \__codepoint_to_bytes_output:nnn { f }
29464 \cs_new:Npn \__codepoint_to_bytes_end: { }

(End definition for \__kernel_codepoint_to_bytes:n and others.)

```

### **\codepoint\_to\_nfd:n**

```

\__codepoint_to_nfd:nn
\__codepoint_to_nfd:nnn
\__codepoint_to_nfd:nnnn
  \__codepoint_to_nfd_generate:nn
    \__codepoint_to_nfd_generate:n
\__codepoint_to_nfd_generate:nnnn

```

Converted to NFD is a potentially-recursive process: the key is to check if we get the input codepoint back again. As far as possible, we use the same path for all engines.

```

29465 \cs_new:Npn \codepoint_to_nfd:n #1
29466   { \__codepoint_to_nfd:nn {#1} { \char_value_catcode:n {#1} } }
29467 \cs_new:Npn \__codepoint_to_nfd:nn #1#2
29468   {
29469     \exp_args:Ne \__codepoint_to_nfd:nnn
29470       { \__codepoint_nfd:n {#1} } {#1} {#2}
29471   }
29472 \cs_new:Npn \__codepoint_to_nfd:nnn #1#2#3 { \__codepoint_to_nfd:nnnn #1 {#2} {#3} }
29473 \cs_new:Npn \__codepoint_to_nfd:nnnn #1#2#3#4
29474   {
29475     \int_compare:nNnTF {#1} = {#3}

```

```

29476 { \codepoint_generate:nn {#1} {#4} }
29477 {
29478     \__codepoint_to_nfd:nn {#1} {#4}
29479     \tl_if_blank:nF {#2}
29480         { \__codepoint_to_nfd:nn {#2} {#4} }
29481     }
29482 }
```

(End definition for `\codepoint_to_nfd:n` and others. This function is documented on page 266.)

## 80.2 Data loader

Text operations requires data from the Unicode Consortium. Data read into Unicode engine formats is at best a small part of what we need, so there is a loader here to set up the appropriate data structures.

Where we need data for most or all of the Unicode range, we use the two-stage table approach recommended by the Unicode Consortium and demonstrated in a model implementation in Python in [https://www.strchr.com/multi-stage\\_tables](https://www.strchr.com/multi-stage_tables). This approach uses the `intarray` (`fontdimen`-based) data type as it is fast for random access and avoids significant hash table usage. In contrast, where only a small subset of codepoints are required, storage as macros is preferable. There is also some consideration of the effort needed to load data: see for example the grapheme breaking information, which would be problematic to convert into a two-stage table but which can be used with reasonable performance in a small number of comma lists (at the cost that breaking at higher codepoint Hangul characters will be slightly slow).

`\c__codepoint_block_size_int` Choosing the block size for the blocks in the two-stage approach is non-trivial: depending on the data stored, the optimal size for memory usage will vary. At the same time, for us there is also the question of load-time: larger blocks require longer comma lists as intermediates, so are slower. As this is going to be needed to use the data, we set it up outside of the group for clarity.

```
29483 \int_const:Nn \c__codepoint_block_size_int { 64 }
```

(End definition for `\c__codepoint_block_size_int`.)

Parsing the data files can be the same way for all engines, but where they are stored as character tokens, the construction method depends on whether they are Unicode or 8-bit internally. Parsing is therefore done by common functions, with some data storage using engine-specific auxiliaries.

As only the data needs to remain at the end of this process, everything is set up inside a group. The only thing that is outside is creating a stream: they are global anyway and it is best to force a stream for all engines.

```
\g__codepoint_data_ior
29484 \ior_new:N \g__codepoint_data_ior
```

(End definition for `\g__codepoint_data_ior`.)

We need some setup for the two-part table approach. The number of blocks we need will be variable, but the resulting size of the stage one table is predictable. For performance reasons, we therefore create the stage one tables now so they can be used immediately, and will later rename them as a constant tables. For each two-stage table construction, we need a comma list to hold the partial block and a couple of integers to

track where we are up to. To avoid burning registers, the latter are stored in macros and are “fake” integers. We also avoid any `new` functions, keeping as much as possible local.

As we need both positive and negative values, case data requires one two-stage table for each transformation. In contrasts, general Unicode properties could be stored in one table with appropriate combination rules: that is not done at present but is likely to be added over time. Here, all that is needed is additional entries into the comma-list to create the structures.

Notice that in the standard `expl3` way we are indexes position not offset: that does mean a little work later.

```

29485 \group_begin:
29486   \clist_map_inline:nn
29487     { uppercase , lowercase }
29488   {
29489     \cs_set_nopar:cpn { l__codepoint_ #1 _block_clist } { }
29490     \cs_set_nopar:cpn { l__codepoint_ #1 _block_t1 } { 1 }
29491     \cs_set_nopar:cpn { l__codepoint_ #1 _pos_t1 } { 0 }
29492     \intarray_new:cn { g__codepoint_ #1 _index_intarray }
29493       { \int_div_truncate:nn { "110000 } \c__codepoint_block_size_int }
29494   }

```

We need an integer value when matching the current block to those we have already seen, and a way to track codepoints for handling ranges. Again, we avoid using up registers or creating global names.

```

29495   \cs_set_nopar:Npn \l__codepoint_next_codepoint_fint_t1 { 0 }
29496   \cs_set_nopar:Npn \l__codepoint_matched_block_t1 { 0 }

```

Parse the main Unicode data file and pull out the NFD and case changing data. The NFD data is stored on using the hash table approach and can yield a predictable number of code points: one or two. We also need the case data, which will be modified further below. To allow for finding rangs, the description of the codepoint needs to be carried forward.

```

29497   \cs_set_protected:Npn \__codepoint_data_auxi:w
29498     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
29499   {
2950     \tl_if_blank:nF {#6}
2951   {
2952     \tl_if_head_eq_charcode:nNF {#6} < % >
2953       { \__codepoint_data_auxii:w #1 ; #6 ~ \q_stop }
2954   }
2955   \__codepoint_data_auxiii:w #1 ; #2 ;
2956 }
2957 \cs_set_protected:Npn \__codepoint_data_auxii:w #1 ; #2 ~ #3 \q_stop
2958 {
2959   \tl_const:cx
2960     { c__codepoint_nfd_ \codepoint_str_generate:n {"#1} _tl }
2961   {
2962     {"#2}
2963     { \tl_if_blank:nF {#3} {"#3} }
2964   }
2965 }

```

The case data is going to be stored as an offset from the parent character, rather than an absolute value. We therefore deal with that plus the situation where a codepoint has no mapping data in oen shot.

```

29516 \cs_set_protected:Npn \__codepoint_data_auxiii:w
29517   #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ~ \q_stop
29518   {
29519     \use:e
29520     {
29521       \__codepoint_data_auxiv:w
29522         #1 ; #2 ;
29523         \__codepoint_data_offset:nn {#1} {#6} ;
29524         \__codepoint_data_offset:nn {#1} {#7} ;
29525         #8;
29526     }
29527   }
29528 \cs_set:Npn \__codepoint_data_offset:nn #1#2
29529   {
29530     \tl_if_blank:nTF {#2}
29531       { 0 }
29532       { \int_eval:n { "#2 - "#1 } }
29533   }

```

To deal with ranges, we track the position of the next codepoint expected. If there is a gap, we deal with that separately: it could be a range or an unused part of the Unicode space. As such, we deal with the current codepoint here whether or not there is range to fill in. Upper- and lowercase data go into the two-stage table, any titlecase exception is just stored in a macro. The data for the codepoint is added to the current block, and if that is now complete we move on to save the block. The case exceptions are all stored as codepoints, with a fixed number of balanced text as we know that there are never more than three.

```

29534 \cs_set_protected:Npn \__codepoint_data_auxiv:w #1 ; #2 ; #3 ; #4 ; #5 ;
29535   {
29536     \int_compare:nNnT {"#1} > \l__codepoint_next_codepoint_fint_tl
29537       {
29538         \__codepoint_data_auxvi:nnnw {#1} {#3} {#4}
29539           #2 Last> \q_stop
29540       }
29541       \__codepoint_add:nn { uppercase } {#3}
29542       \__codepoint_add:nn { lowercase } {#4}
29543       \int_compare:nNnF {#3} = { \__codepoint_data_offset:nn {#1} {#5} }
29544         {
29545           \tl_const:cx
29546             { c__codepoint_titlecase_ \codepoint_str_generate:n {"#1} _tl }
29547             { {"#5} { } { } }
29548         }
29549         \tl_set:Nx \l__codepoint_next_codepoint_fint_tl
29550           { \int_eval:n { "#1 + 1 } }
29551       }
29552 \cs_set_protected:Npn \__codepoint_add:nn #1#2
29553   {
29554     \clist_put_right:cn { l__codepoint_ #1 _block_clist } {#2}
29555     \int_compare:nNnT { \clist_count:c { l__codepoint_ #1 _block_clist } }
29556       = \c__codepoint_block_size_int
29557       { \__codepoint_save_blocks:nn {#1} { 1 } }
29558   }

```

Distinguish between a range and a gap, and pass on the appropriate value(s).

```

29559   \cs_set_protected:Npn \__codepoint_data_auxvi:nnnw #1#2#3#4 Last> #5 \q_stop
29560   {
29561     \tl_if_blank:nTF {#5}
29562     {
29563       \__codepoint_range:nnn {#1} { uppercase } { 0 }
29564       \__codepoint_range:nnn {#1} { lowercase } { 0 }
29565     }
29566     {
29567       \__codepoint_range:nnn {#1} { uppercase } {#2}
29568       \__codepoint_range:nnn {#1} { lowercase } {#3}
29569     }
29570   }

```

Calculated the length of the range and the space remaining in the current block.

```

29571   \cs_set_protected:Npn \__codepoint_range:nnn #1
29572   {
29573     \exp_args:Nf \__codepoint_range_aux:nnn
29574     { \int_eval:n { "#1 - \l__codepoint_next_codepoint_fint_t1 } }
29575   }
29576   \cs_set_protected:Npn \__codepoint_range_aux:nnn #1#2
29577   {
29578     \exp_args:Nf \__codepoint_range:nnnn
29579     {
29580       \int_min:nn
29581         {#1}
29582       {
29583         \c__codepoint_block_size_int
29584         - \clist_count:c { \l__codepoint_ #2 _block_clist }
29585       }
29586     }
29587     {#1} {#2}
29588   }

```

Here we want to do three things: add to and possibly complete the current block, add complete blocks quickly, then finish up the range in a final open block. We need to avoid as far as possible avoid dealing with ever single codepoint, so the middle step is optimised.

```

29589   \cs_set_protected:Npn \__codepoint_range:nnnn #1#2#3#4
29590   {
29591     \prg_replicate:nn {#1}
29592       { \clist_put_right:cn { \l__codepoint_ #3 _block_clist } {#4} }
29593     \int_compare:nNnT { \clist_count:c { \l__codepoint_ #3 _block_clist } }
29594       = \c__codepoint_block_size_int
29595       { \__codepoint_save_blocks:nn {#3} { 1 } }
29596     \int_compare:nNnF
29597       { \int_div_truncate:nn { #2 - #1 } \c__codepoint_block_size_int } = 0
29598       {
29599         \tl_set:cx { \l__codepoint_ #3 _block_clist }
29600         {
29601           \exp_args:NNe \use:nn \use_none:n
29602             { \prg_replicate:nn { \c__codepoint_block_size_int } { , #4 } }
29603           }
29604         \__codepoint_save_blocks:nn {#3}
29605           { \int_div_truncate:nn { (#2 - #1) } \c__codepoint_block_size_int }
29606         }
29607       \prg_replicate:nn

```

```

29608     { \int_mod:nn { #2 - #1 } \c__codepoint_block_size_int }
29609     { \clist_put_right:cx { l__codepoint_ #3 _block_clist } {#4} }
29610 }

```

To allow rapid comparison, each completed block is stored locally as a comma list: once all of the blocks have been created, they are converted into an `intarray` in one step. The aim here is to check the current block against those we've already used, and either match to an existing block or save a new block.

```

29611 \cs_set_protected:Npn \__codepoint_save_blocks:nn #1#2
29612 {
29613     \tl_set_eq:Nc \l__codepoint_matched_block_tl { l__codepoint_ #1 _block_tl }
29614     \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
29615     {
29616         \tl_if_eq:ccT { l__codepoint_ #1 _block_clist }
29617         { l__codepoint_ #1 _block_ ##1 _clist }
29618         { \tl_set:Nn \l__codepoint_matched_block_tl {##1} }
29619     }
29620     \int_compare:nNnT
29621     { \tl_use:c { l__codepoint_ #1 _block_tl } } = \l__codepoint_matched_block_tl
29622     {
29623         \clist_set_eq:cc
29624         {
29625             l__codepoint_ #1 _block_
29626             \tl_use:c { l__codepoint_ #1 _block_tl } _clist
29627         }
29628         { l__codepoint_ #1 _block_clist }
29629         \tl_set:cx { l__codepoint_ #1 _block_tl }
29630         { \int_eval:n { \tl_use:c { l__codepoint_ #1 _block_tl } + 1 } }
29631     }
29632     \prg_replicate:nn {#2}
29633     {
29634         \tl_set:cx { l__codepoint_ #1 _pos_tl }
29635         { \int_eval:n { \tl_use:c { l__codepoint_ #1 _pos_tl } + 1 } }
29636         \exp_args:Nc \__kernel_intarray_gset:Nnn
29637         { g__codepoint_ #1 _index_intarray }
29638         { \tl_use:c { l__codepoint_ #1 _pos_tl } }
29639         \l__codepoint_matched_block_tl
29640     }
29641     \clist_clear:c { l__codepoint_ #1 _block_clist }
29642 }

```

Close out the final block, rename the first stage table, then combine all of the block comma-lists into one large second-stage table with offsets. As we use an index not an offset, there is a little back-and-forward to do.

```

29643 \cs_set_protected:Npn \__codepoint_finalise_blocks:
29644 {
29645     \clist_map_inline:nn { uppercase , lowercase }
29646     {
29647         \__codepoint_range:nnn { 110000 } {##1} { 0 }
29648         \__codepoint_finalise_blocks:n {##1}
29649     }
29650 }
29651 \cs_set_protected:Npn \__codepoint_finalise_blocks:n #1
29652 {

```

```

29653 \cs_gset_eq:cc { c__codepoint_ #1 _index_intarray } { g_codepoint_ #1 _index_intarra
29654 \cs_undefine:c { g_codepoint_ #1 _index_intarray }
29655 \intarray_new:cn { g_codepoint_ #1 _blocks_intarray }
29656     { ( \tl_use:c { l__codepoint_ #1 _block_tl } - 1 ) * \c_codepoint_block_size_int
29657 \int_step_inline:nn { \tl_use:c { l__codepoint_ #1 _block_tl } - 1 }
29658     {
29659         \exp_args:Nv \__codepoint_finalise_blocks:nnn
29660             { l__codepoint_ #1 _block_ ##1 _clist }
29661             {##1} {#1}
29662     }
29663 \cs_gset_eq:cc { c__codepoint_ #1 _blocks_intarray }
29664     { g_codepoint_ #1 _blocks_intarray }
29665 \cs_undefine:c { g_codepoint_ #1 _blocks_intarray }
29666 }
29667 \cs_set_protected:Npn \__codepoint_finalise_blocks:nnn #1#2#3
29668 {
29669     \exp_args:Nnf \__codepoint_finalise_blocks:nnnw { 1 }
29670         { \int_eval:n { ( #2 - 1 ) * \c_codepoint_block_size_int } }
29671         {#3}
29672         #1 , \q_recursion_tail , \q_recursion_stop
29673 }
29674 \cs_set_protected:Npn \__codepoint_finalise_blocks:nnnw #1#2#3#4 ,
29675 {
29676     \quark_if_recursion_tail_stop:n {#4}
29677     \intarray_gset:cnn { g_codepoint_ #3 _blocks_intarray }
29678         { #1 + #2 }
29679         {#4}
29680     \exp_args:Nf \__codepoint_finalise_blocks:nnnw
29681         { \int_eval:n { #1 + 1 } } {#2} {#3}
29682 }

```

With the setup done, read the main data file: it's easiest to do that as a token list with spaces retained.

```

29683 \ior_open:Nn \g_codepoint_data_ior { UnicodeData.txt }
29684 \group_begin:
29685     \char_set_catcode_space:n { '\ }%
29686     \ior_map_variable>NNn \g_codepoint_data_ior \l_codepoint_tma_t1
29687     {%
29688         \if_meaning:w \l_codepoint_tma_t1 \c_space_t1
29689             \exp_after:wN \ior_map_break:
29690             \fi:
29691             \exp_after:wN \__codepoint_data_auxi:w \l_codepoint_tma_t1 \q_stop
29692         }%
29693         \__codepoint_finalise_blocks:
29694     \group_end:
29695 \group_end:

```

`\__kernel_codepoint_data:nn` `\__codepoint_data:nnn` Recover data from a two-stage table: entirely generic as this applies to all tables (as we use the same block size for all of them). Notice that as we use indices not offsets we have to shuffle out-by-one issues. This function is needed *before* loading the special casing data, as there we need to be able to check the standard case mappings.

```

29696 \cs_new:Npn \__kernel_codepoint_data:nn #1#2
29697     {
29698         \exp_args:Nf \__codepoint_data:nnn

```

```

29699 {
29700   \int_eval:n
29701   {
29702     \c__codepoint_block_size_int *
29703     (
29704       \intarray_item:cn { c__codepoint_ #1 _index_intarray }
29705       {
29706         \int_div_truncate:nn {#2}
29707         \c__codepoint_block_size_int
29708         + 1
29709       }
29710       - 1
29711     )
29712   }
29713   {#2} {#1}
29714 }
29715 \cs_new:Npn \__codepoint_data:nnn #1#2#3
29716 {
29717   \intarray_item:cn { c__codepoint_ #3 _blocks_intarray }
29718   { #1 + \int_mod:nn {#2} \c__codepoint_block_size_int + 1 }
29719 }
29720 }
```

(End definition for `\__kernel_codepoint_data:nn` and `\__codepoint_data:nnn`.)

The other data files all use C-style comments so we have to worry about # tokens (and reading as strings). The set up for case folding is in two parts. For the basic (core) mappings, folding is the same as lower casing in most positions so only store the differences. For the more complex foldings, always store the result, splitting up the two or three code points in the input as required.

```

29721 \group_begin:
29722   \ior_open:Nn \g__codepoint_data_ior { CaseFolding.txt }
29723   \cs_set_protected:Npn \__codepoint_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
29724   {
29725     \if:w \tl_head:n { #2 ? } C
29726     \reverse_if:N \if_int_compare:w
29727       \int_eval:n { \__kernel_codepoint_data:nn { lowercase } {"#1} + "#1 }
29728       = "#3 ~
29729       \tl_const:cx
29730         { c__codepoint_casifold_ \codepoint_str_generate:n {"#1} _tl }
29731         { {"#3} { } { } }
29732     \fi:
29733   \else:
29734     \if:w \tl_head:n { #2 ? } F
29735       \__codepoint_data_auxii:w #1 ~ #3 ~ \q_stop
29736     \fi:
29737   \fi:
29738 }
```

Here, #4 can have have a trailing space, so we tidy up a bit at the cost of speed for these small number of cases it applies to.

```

29739   \cs_set_protected:Npn \__codepoint_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
29740   {
29741     \tl_const:cx { c__codepoint_casifold_ \codepoint_str_generate:n {"#1} _tl }
29742     {
```

```

29743     {"#2}
29744     {"#3}
29745     { \tl_if_blank:nF {#4} { " \int_to_Hex:n {"#4} } }
29746   }
29747 }
29748 \ior_str_map_inline:Nn \g__codepoint_data_ior
29749 {
29750   \reverse_if:N \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
29751   \__codepoint_data_auxi:w #1 \q_stop
29752   \fi:
29753 }
29754 \ior_close:N \g__codepoint_data_ior

```

For upper- and lowercasing special situations, there is a bit more to do as we also have titlecasing to consider, plus we need to stop part-way through the file.

```

29755   \ior_open:Nn \g__codepoint_data_ior { SpecialCasing.txt }
29756   \cs_set_protected:Npn \__codepoint_data_auxi:w
29757     #1 ;~ #2 ;~ #3 ;~ #4 ;~ #5 \q_stop
29758   {
29759     \use:n { \__codepoint_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
29760     \use:n { \__codepoint_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
29761     \str_if_eq:nnF {#3} {#4}
29762       { \use:n { \__codepoint_data_auxii:w #1 ~ title ~ #3 ~ } ~ \q_stop }
29763   }
29764   \cs_set_protected:Npn \__codepoint_data_auxii:w
29765     #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
29766   {
29767     \tl_if_empty:nF {#4}
29768     {
29769       \tl_const:cx { c__codepoint_ #2 case_ \codepoint_str_generate:n {"#1} _tl }
29770       {
29771         {"#3}
29772         {"#4}
29773         { \tl_if_blank:nF {#5} {"#5} }
29774       }
29775     }
29776   }
29777 \ior_str_map_inline:Nn \g__codepoint_data_ior
29778   {
29779     \str_if_eq:eeTF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
29780     {
29781       \str_if_eq:eeT
29782         {"#1}
29783         { \c_hash_str \c_space_tl Conditional~Mappings }
29784         { \ior_map_break: }
29785     }
29786     { \__codepoint_data_auxi:w #1 \q_stop }
29787   }
29788 \ior_close:N \g__codepoint_data_ior
29789 \group_end:

```

```

\_kernel_codepoint_case:nn
  \__codepoint_case:nnn
  \__codepoint_uppercase:n
  \__codepoint_lowercase:n
  \__codepoint_titlecase:n
  \__codepoint_casifold:n
  \@_case:nn

```

With the core data files loaded, there is now a need to provide access to this information for other modules. That is done here such that case folding can also be covered. At this level, all that needs to be returned is the

```

29790 \cs_new:Npn \__kernel_codepoint_case:nn #1#2
29791 {
29792     \exp_args:Ne \__codepoint_case:nnn
29793     { \codepoint_str_generate:n {#2} } {#1} {#2}
29794 }
29795 \cs_new:Npn \__codepoint_case:nnn #1#2#3
29796 {
29797     \cs_if_exist:cTF { c__codepoint_ #2 _ #1 _tl }
29798     {
29799         \tl_use:c
29800         { c__codepoint_ #2 _ #1 _tl }
29801     }
29802     { \use:c { __codepoint_ #2 :n } {#3} }
29803 }
29804 \cs_new:Npn \__codepoint_uppercase:n { \__codepoint_case:nn { uppercase } }
29805 \cs_new:Npn \__codepoint_lowercase:n { \__codepoint_case:nn { lowercase } }
29806 \cs_new:Npn \__codepoint_titlecase:n { \__codepoint_case:nn { uppercase } }
29807 \cs_new:Npn \__codepoint_casifold:n { \__codepoint_case:nn { lowercase } }
29808 \cs_new:Npn \__codepoint_case:nn #1#2
29809 {
29810     { \int_eval:n { \__kernel_codepoint_data:nn {#1} {#2} + #2 } }
29811     { }
29812     { }
29813 }

```

(End definition for `\__kernel_codepoint_case:nn` and others. This function is documented on page ??.)

`\__codepoint_nfd:n` A simple interface.

```

\__codepoint_nfd:nn
29814 \cs_new:Npn \__codepoint_nfd:n #1
29815 { \exp_args:Ne \__codepoint_nfd:nn { \codepoint_str_generate:n {#1} } {#1} }
29816 \cs_new:Npn \__codepoint_nfd:nn #1#2
29817 {
29818     \tl_if_exist:cTF { c__codepoint_nfd_ #1 _tl }
29819     { \tl_use:c { c__codepoint_nfd_ #1 _tl } }
29820     { {#2} { } }
29821 }

```

(End definition for `\__codepoint_nfd:n` and `\__codepoint_nfd:nn`.)

29822 `<@=text>`

Read the Unicode grapheme data. This is quite easy to handle and we only need codepoints, not characters, so there is no need to worry about the engine in use. As reading as a string is most convenient, we have to do some work to remove spaces: the hardest part of the entire process!

```

29823 \ior_new:N \g__text_data_ior
29824 \group_begin:
29825     \ior_open:Nn \g__text_data_ior { GraphemeBreakProperty.txt }
29826     \cs_set_nopar:Npn \l__text_tmpa_str { }
29827     \cs_set_nopar:Npn \l__text_tmpb_str { }
29828     \cs_set_protected:Npn \__text_data_auxi:w #1 ;~ #2 ~ #3 \q_stop
29829     {
29830         \str_if_eq:VnF \l__text_tmpb_str {#2}
29831     }

```

```

29832   \str_if_empty:NF \l__text_tmpb_str
29833   {
29834     \clist_const:cx { c__text_grapheme_ \l__text_tmpb_str _clist }
29835     { \exp_after:wN \use_none:n \l__text_tmpa_str }
29836     \cs_set_nopar:Npn \l__text_tmpa_str { }
29837   }
29838   \cs_set_nopar:Npn \l__text_tmpb_str {\#2}
29839   }
29840   \__text_data_auxii:w #1 .. #1 .. #1 \q_stop
29841   }
29842   \cs_set_protected:Npn \__text_data_auxii:w #1 .. #2 .. #3 \q_stop
29843   {
29844     \cs_set_nopar:Npx \l__text_tmpa_str
29845     {
29846       \l__text_tmpa_str ,
29847       \tl_trim_spaces:n {\#1} .. \tl_trim_spaces:n {\#2}
29848     }
29849   }
29850   \ior_str_map_inline:Nn \g__text_data_ior
29851   {
29852     \str_if_eq:eeF { \tl_head:w #1 \c_hash_str \q_stop } { \c_hash_str }
29853     {
29854       \tl_if_blank:nF {\#1}
29855       { \__text_data_auxi:w #1 \q_stop }
29856     }
29857   }
29858   \ior_close:N \g__text_data_ior
29859 \group_end:
29860 </package>

```

# Chapter 81

## I3text implementation

```
29861  <*package>
29862  <@@=text>
29863  \cs_generate_variant:Nn \tl_if_head_eq_meaning_p:nN { o }
```

### 81.1 Internal auxiliaries

\s\_\_text\_stop Internal scan marks.

```
29864  \scan_new:N \s__text_stop
```

(End definition for \s\_\_text\_stop.)

\q\_\_text\_nil Internal quarks.

```
29865  \quark_new:N \q__text_nil
```

(End definition for \q\_\_text\_nil.)

\\_\_text\_quark\_if\_nil\_p:n Branching quark conditional.

\\_\_text\_quark\_if\_nil:nTF 29866 \kernel\_quark\_new\_conditional:Nn \\_\_text\_quark\_if\_nil:n { TF }

(End definition for \\_\_text\_quark\_if\_nil:nTF.)

\q\_\_text\_recursion\_tail Internal recursion quarks.

\q\_\_text\_recursion\_stop 29867 \quark\_new:N \q\_\_text\_recursion\_tail

```
29868  \quark_new:N \q__text_recursion_stop
```

(End definition for \q\_\_text\_recursion\_tail and \q\_\_text\_recursion\_stop.)

\\_\_text\_use\_i\_delimit\_by\_q\_recursion\_stop:nw Functions to gobble up to a quark.

```
29869 \cs_new:Npn \__text_use_i_delimit_by_q_recursion_stop:nw
```

```
29870  #1 #2 \q__text_recursion_stop {#1}
```

(End definition for \\_\_text\_use\_i\_delimit\_by\_q\_recursion\_stop:nw.)

\\_\_text\_if\_q\_recursion\_tail\_stop\_do:Nn Functions to query recursion quarks.

```
29871 \kernel_quark_new_test:N \__text_if_q_recursion_tail_stop_do:Nn
```

```
29872 \kernel_quark_new_test:N \__text_if_q_recursion_tail_stop_do:nn
```

(End definition for `\_text_if_q_recursion_tail_stop:NN` and `\_text_if_q_recursion_tail_stop:nn`.)

`\s_text_recursion_tail` Internal scan marks quarks.

`\s_text_recursion_stop`

(End definition for `\s_text_recursion_tail` and `\s_text_recursion_stop`.)

`\_text_use_i_delimit_by_s_recursion_stop:nw`

Functions to gobble up to a scan mark.

`29875 \cs_new:Npn \_text_use_i_delimit_by_s_recursion_stop:nw`  
`29876 #1 #2 \s_text_recursion_stop {#1}`

(End definition for `\_text_use_i_delimit_by_s_recursion_stop:nw`.)

`\_text_if_s_recursion_tail_stop:NN`

Functions to query recursion scan marks. Slower than a quark test but needed to avoid issues in the outer expansion loop with unterminated `\romannumeral` primitives.

`29877 \cs_new:Npn \_text_if_s_recursion_tail_stop:NN #1`  
`29878 {`  
`29879 \bool_lazy_and:nnTF`  
`29880 { \cs_if_eq_p:NN \s_text_recursion_tail #1 }`  
`29881 { \str_if_eq_p:nn { \s_text_recursion_tail } {#1} }`  
`29882 { \_text_use_i_delimit_by_s_recursion_stop:nw }`  
`29883 { \use_none:n }`  
`29884 }`

(End definition for `\_text_if_s_recursion_tail_stop:NN`.)

## 81.2 Utilities

The idea here is to take a token and ensure that if it's an implicit char, we output the explicit version. Otherwise, the token needs to be unchanged. First, we have to split between control sequences and everything else.

`29885 \group_begin:`  
`29886 \char_set_catcode_active:n { 0 }`  
`29887 \cs_new:Npn \_text_token_to_explicit:N #1`  
`29888 {`  
`29889 \if_catcode:w \exp_not:N #1`  
`29890 \if_catcode:w \scan_stop: \exp_not:N #1`  
`29891 \scan_stop:`  
`29892 \else:`  
`29893 \exp_not:N ^^@`  
`29894 \fi:`  
`29895 \exp_after:wN \_text_token_to_explicit_cs:N`  
`29896 \else:`  
`29897 \exp_after:wN \_text_token_to_explicit_char:N`  
`29898 \fi:`  
`29899 #1`  
`29900 }`  
`29901 \group_end:`

For control sequences, we can check for macros versus other cases using `\if_meaning:w`, then explicitly check for `\chardef` and `\mathchardef`.

```

29902 \cs_new:Npn __text_token_to_explicit_cs:N #1
29903 {
29904     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
29905         \exp_after:wN \use:nn \exp_after:wN
29906             __text_token_to_explicit_cs_aux:N
29907     \else:
29908         \exp_after:wN \exp_not:n
29909     \fi:
29910     {#1}
29911 }
29912 \cs_new:Npn __text_token_to_explicit_cs_aux:N #1
29913 {
29914     \bool_lazy_or:nnTF
29915         { \token_if_chardef_p:N #1 }
29916         { \token_if_mathchardef_p:N #1 }
29917         {
29918             \char_generate:nn {#1}
29919                 { \char_value_catcode:n {#1} }
29920         }
29921     {#1}
29922 }
```

For character tokens, we need to filter out the implicit characters from those that are explicit. That's done here, then if necessary we work out the category code and generate the char. To avoid issues with alignment tabs, that one is done by elimination rather than looking up the code explicitly. The trick with finding the charcode is that the TeX messages are either the `<something>` character `<char>` or the `<type>` `<char>`.

```

29923 \cs_new:Npn __text_token_to_explicit_char:N #1
29924 {
29925     \if:w
29926         \if_catcode:w ^ \exp_args:No \str_tail:n { \token_to_str:N #1 } ^
29927             \token_to_str:N #1 #1
29928         \else:
29929             AB
29930         \fi:
29931         \exp_after:wN \exp_not:n
29932     \else:
29933         \exp_after:wN __text_token_to_explicit:n
29934     \fi:
29935     {#1}
29936 }
29937 \cs_new:Npn __text_token_to_explicit:n #1
29938 {
29939     \exp_after:wN __text_token_to_explicit_auxi:w
29940     \int_value:w
29941         \if_catcode:w \c_group_begin_token #1 1 \else:
29942             \if_catcode:w \c_group_end_token #1 2 \else:
29943                 \if_catcode:w \c_math_toggle_token #1 3 \else:
29944                     \if_catcode:w ## #1 6 \else:
29945                         \if_catcode:w ^ #1 7 \else:
29946                             \if_catcode:w \c_math_subscript_token #1 8 \else:
29947                                 \if_catcode:w \c_space_token #1 10 \else:
```

```

29948     \if_catcode:w A #1 11 \else:
29949     \if_catcode:w + #1 12 \else:
29950         4 \fi: \fi: \fi: \fi: \fi: \fi: \fi: \fi:
29951     \exp_after:wN ;
29952     \token_to_meaning:N #1 \s__text_stop
29953 }
29954 \cs_new:Npn \__text_token_to_explicit_auxi:w #1 ; #2 \s__text_stop
29955 {
29956     \char_generate:nn
29957     {
29958         \if_int_compare:w #1 < 9 \exp_stop_f:
29959             \exp_after:wN \__text_token_to_explicit_auxii:w
29960         \else:
29961             \exp_after:wN \__text_token_to_explicit_auxiii:w
29962             \fi:
29963             #2
29964         }
29965         {#1}
29966     }
29967 \exp_last_unbraced:NNNNo \cs_new:Npn \__text_token_to_explicit_auxii:w
29968     #1 { \tl_to_str:n { character ~ } } { ' }
29969 \cs_new:Npn \__text_token_to_explicit_auxiii:w #1 ~ #2 ~ { ' }

(End definition for \__text_token_to_explicit:N and others.)

```

\\_\_text\_char\_catcode:N An idea from l3char: we need to get the category code of a specific token, not the general case.

```

29970 \cs_new:Npn \__text_char_catcode:N #1
29971 {
29972     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
29973         3
29974     \else:
29975         \if_catcode:w \exp_not:N #1 \c_alignment_token
29976             4
29977     \else:
29978         \if_catcode:w \exp_not:N #1 \c_math_superscript_token
29979             7
29980     \else:
29981         \if_catcode:w \exp_not:N #1 \c_math_subscript_token
29982             8
29983     \else:
29984         \if_catcode:w \exp_not:N #1 \c_space_token
29985             10
29986     \else:
29987         \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
29988             11
29989     \else:
29990         \if_catcode:w \exp_not:N #1 \c_catcode_other_token
29991             12
29992     \else:
29993         13
29994     \fi:
29995     \fi:
29996 
```

```

29997     \fi:
29998     \fi:
29999     \fi:
30000     \fi:
30001 }

```

(End definition for `\_text_catcode:N.`)

`\_text_if_expandable:NTF` Test for tokens that make sense to expand here: that is more restrictive than the engine view.

```

30002 \prg_new_conditional:Npnn \_text_if_expandable:N #1 { T , F , TF }
30003 {
30004     \token_if_expandable:NTF #1
30005     {
30006         \bool_lazy_any:nTF
30007         {
30008             { \token_if_protected_macro_p:N      #1 }
30009             { \token_if_protected_long_macro_p:N #1 }
30010             { \token_if_eq_meaning_p:NN \q_text_recursion_tail #1 }
30011         }
30012         { \prg_return_false: }
30013         { \prg_return_true: }
30014     }
30015     { \prg_return_false: }
30016 }

```

(End definition for `\_text_if_expandable:NTF.`)

### 81.3 Codepoint utilities

For working with codepoints in an engine-neutral way.

`\_text_codepoint_process:nN` Grab a codepoint and apply some code to it: here `#1` should expect one following *balanced text*.

```

\text_codepoint_process:nNN
\text_codepoint_process:nNNN
\text_codepoint_process:nNNNN
30017 \bool_lazy_or:nnTF
30018 { \sys_if_engine_luatex_p: }
30019 { \sys_if_engine_xetex_p: }
30020 {
30021     \cs_new:Npn \_text_codepoint_process:nN #1#2 { #1 {#2} }
30022 }
30023 {
30024     \cs_new:Npn \_text_codepoint_process:nN #1#2
30025     {
30026         \int_compare:nNnTF { '#2 } > { "80 }
30027         {
30028             \int_compare:nNnTF { '#2 } < { "E0 }
30029             { \_text_codepoint_process:nNN }
30030             {
30031                 \int_compare:nNnTF { '#2 } < { "F0 }
30032                 { \_text_codepoint_process:nNNN }
30033                 { \_text_codepoint_process:nNNNN }
30034             }
30035 }

```

```

30036     { \use:n }
30037         {#1} #2
30038     }
30039     \cs_new:Npn \__text_codepoint_process:nNN #1#2#3
30040         { #1 {#2#3} }
30041     \sys_if_engine_ptex:TF
30042     {
30043         \cs_gset:Npn \__text_codepoint_process:nN #1#2
30044         {
30045             \int_compare:nNnTF { '#2 } > { "80 }
30046                 {
30047                     \int_compare:nNnTF { '#2 } < { "E0 }
30048                         { \__text_codepoint_process:nNN }
30049                         { \use:n }
30050                 }
30051             { \use:n }
30052                 {#1} #2
30053         }
30054     }
30055     {
30056         \cs_new:Npn \__text_codepoint_process:nNNN #1#2#3#4
30057             { #1 {#2#3#4} }
30058         \cs_new:Npn \__text_codepoint_process:nNNNN #1#2#3#4#5
30059             { #1 {#2#3#4#5} }
30060     }
30061 }
```

(End definition for `\__text_codepoint_process:nN` and others.)

`\__text_codepoint_compare_p:nNn` Allows comparison for all engines using a first “character” followed by a codepoint.

```

30062 \bool_lazy_or:nNnTF
30063 { \sys_if_engine_luatex_p: }
30064 { \sys_if_engine_xetex_p: }
30065 {
30066     \prg_new_conditional:Npnn
30067         \__text_codepoint_compare:nNn #1#2#3 { TF , p }
30068     {
30069         \int_compare:nNnTF { '#1 } #2 {#3}
30070             \prg_return_true: \prg_return_false:
30071     }
30072     \cs_new:Npn \__text_codepoint_from_chars:Nw #1 {'#1}
30073 }
30074 {
30075     \prg_new_conditional:Npnn
30076         \__text_codepoint_compare:nNn #1#2#3 { TF , p }
30077     {
30078         \int_compare:nNnTF { \__text_codepoint_from_chars:Nw #1 }
30079             #2 {#3}
30080             \prg_return_true: \prg_return_false:
30081     }
30082     \cs_new:Npn \__text_codepoint_from_chars:Nw #1
30083     {
30084         \if_int_compare:w '#1 > "80 \exp_stop_f:
30085             \if_int_compare:w '#1 < "E0 \exp_stop_f:
```

```

30086   \exp_after:wN \exp_after:wN \exp_after:wN
30087     \__text_codepoint_from_chars>NN
30088 \else:
30089   \if_int_compare:w '#1 < "F0 \exp_stop_f:
30090     \exp_after:wN \exp_after:wN \exp_after:wN
30091     \exp_after:wN \exp_after:wN \exp_after:wN
30092     \exp_after:wN \__text_codepoint_from_chars>NNNN
30093 \else:
30094   \exp_after:wN \exp_after:wN \exp_after:wN
30095   \exp_after:wN \exp_after:wN \exp_after:wN
30096   \exp_after:wN \__text_codepoint_from_chars>NNNN
30097 \fi:
30098 \fi:
30099 \else:
30100   \exp_after:wN \__text_codepoint_from_chars:N
30101 \fi:
30102 #1
30103 }
30104 \cs_new:Npn \__text_codepoint_from_chars:N #1 { '#1 }
30105 \cs_new:Npn \__text_codepoint_from_chars>NN #1#2
30106 { ('#1 - "CO) * "40 + '#2 - "80 }

30107

```

Avoid high chars with pTeX.

```

30108   \sys_if_engine_ptex:TF
30109   {
30110     \cs_gset:Npn \__text_codepoint_from_chars:Nw #1
30111     {
30112       \if_int_compare:w '#1 > "80 \exp_stop_f:
30113       \if_int_compare:w '#1 < "E0 \exp_stop_f:
30114         \exp_after:wN \exp_after:wN \exp_after:wN
30115           \__text_codepoint_from_chars>NN
30116     \else:
30117       \exp_after:wN \exp_after:wN \exp_after:wN
30118         \__text_codepoint_from_chars:N
30119     \fi:
30120   \else:
30121     \exp_after:wN \__text_codepoint_from_chars:N
30122   \fi:
30123     #1
30124   }
30125 }
30126 {
30127   \cs_new:Npn \__text_codepoint_from_chars>NNN #1#2#3
30128   { ('#1 - "E0) * "1000 + ('#2 - "80) * "40 + '#3 - "80 }
30129   \cs_new:Npn \__text_codepoint_from_chars>NNNN #1#2#3#4
30130   {
30131     ('#1 - "F0) * "40000
30132     + ('#2 - "80) * "1000
30133     + ('#3 - "80) * "40
30134     + '#4 - "80
30135   }
30136 }
30137

```

(End definition for `\_text_codepoint_compare:nNnTF` and others.)

## 81.4 Configuration variables

`\l_text_accents_tl` Special cases for accents and letter-like symbols, which in some cases will need to be converted further.

```
30138 \tl_new:N \l_text_accents_tl
30139 \tl_set:Nn \l_text_accents_tl
30140 { ` ' ^ ~ = \u \. " \r \H \v \d \c \k \b \t }
30141 \tl_new:N \l_text_letterlike_tl
30142 \tl_set:Nn \l_text_letterlike_tl
30143 {
30144   \AA \aa
30145   \AE \ae
30146   \DH \dh
30147   \DJ \dj
30148   \IJ \ij
30149   \L \l
30150   \NG \ng
30151   \O \o
30152   \OE \oe
30153   \SS \ss
30154   \TH \th
30155 }
```

(End definition for `\l_text_accents_tl` and `\l_text_letterlike_tl`. These variables are documented on page 270.)

`\l_text_case_exclude_arg_tl` Non-text arguments, including covering the case of `\protected@edef` applied to `\cite`.

```
30156 \tl_new:N \l_text_case_exclude_arg_tl
30157 \tl_set:Nx \l_text_case_exclude_arg_tl
30158 {
30159   \exp_not:n { \begin \cite \end \label \ref }
30160   \exp_not:c { cite ~ }
30161   \exp_not:n { \babelshorthand }
30162 }
```

(End definition for `\l_text_case_exclude_arg_tl`. This variable is documented on page 270.)

`\l_text_math_arg_tl` Math mode as arguments.

```
30163 \tl_new:N \l_text_math_arg_tl
30164 \tl_set:Nn \l_text_math_arg_tl { \ensuremath }
```

(End definition for `\l_text_math_arg_tl`. This variable is documented on page 270.)

`\l_text_math_delims_tl` Paired math mode delimiters.

```
30165 \tl_new:N \l_text_math_delims_tl
30166 \tl_set:Nn \l_text_math_delims_tl { $ $ \( \) }
```

(End definition for `\l_text_math_delims_tl`. This variable is documented on page 270.)

**\l\_text\_expand\_exclude\_tl** Commands which need not to expand. We start with a somewhat historical list, and tidy up if possible.

```

30167 \tl_new:N \l_text_expand_exclude_tl
30168 \tl_set:Nn \l_text_expand_exclude_tl
30169   { \begin{cite} \end{label} \ref }
30170 \bool_lazy_and:nnT
30171   { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
30172   { \tl_if_exist_p:N \expl@finalise@setup@@ }
30173   {
30174     \tl_gput_right:Nn \expl@finalise@setup@@
30175     {
30176       \tl_gput_right:Nn \kernel@after@begindocument
30177       {
30178         \group_begin:
30179           \cs_set_protected:Npn \__text_tmp:w #1
30180           {
30181             \tl_clear:N \l_text_expand_exclude_tl
30182             \tl_map_inline:nn {#1}
30183             {
30184               \bool_lazy_any:nF
30185               {
30186                 \token_if_protected_macro_p:N ##1
30187                 \token_if_protected_long_macro_p:N ##1
30188                 {
30189                   \str_if_eq_p:ee
30190                     { \cs_replacement_spec:N ##1 }
30191                     { \exp_not:n { \protect ##1 } \c_space_tl }
30192                   }
30193                 }
30194               { \tl_put_right:Nn \l_text_expand_exclude_tl {##1} }
30195             }
30196           }
30197           \exp_args:NV \__text_tmp:w \l_text_expand_exclude_tl
30198           \exp_args:NNNV \group_end:
30199             \tl_set:Nn \l_text_expand_exclude_tl \l_text_expand_exclude_tl
30200           }
30201         }
30202       }
30203     }

```

(End definition for `\l_text_expand_exclude_tl`. This variable is documented on page [270](#).)

**\l\_\_text\_math\_mode\_tl** Used to control math mode output: internal as there is a dedicated setter.

```
30203 \tl_new:N \l__text_math_mode_tl
```

(End definition for `\l__text_math_mode_tl`.)

## 81.5 Expansion to formatted text

Markers for implicit char handling.

```

30204 \tex_chardef:D \c_text_chardef_space_token = '\ %
30205 \tex_mathchardef:D \c_text_mathchardef_space_token = '\ %
30206 \tex_chardef:D \c_text_chardef_group_begin_token = '{ % '}'
30207 \tex_mathchardef:D \c_text_mathchardef_group_begin_token = '{ % '}'
\c_text_mathchardef_group_end_token

```

```

30208 \tex_chardef:D \c__text_chardef_group_end_token = '\} % '\{
30209 \tex_mathchardef:D \c__text_mathchardef_group_end_token = '\} %

```

(End definition for `\c__text_chardef_space_token` and others.)

```

\text_expand:n
\__text_expand:n
\__text_expand_result:n
\__text_expand_store:n
\__text_expand_store:o
\__text_expand_store:nw
\__text_expand_end:w
\__text_expand_loop:w
\__text_expand_group:n
\__text_expand_space:w
\__text_expand_N_type:N
\__text_expand_N_type_auxi:N
\__text_expand_N_type_auxii:N
\__text_expand_math_search:NNN
\__text_expand_math_loop:Nw
\__text_expand_math:N type>NN
\__text_expand_math_group:Nn
\__text_expand_math_space:Nw
\__text_expand_implicit:N
\__text_expand_explicit:N
\__text_expand_exclude:N
\__text_expand_exclude_switch:Nnnnn
\__text_expand_exclude:nN
\__text_expand_exclude:NN
\__text_expand_exclude:Nw
\__text_expand_exclude:Nnn
\__text_expand_accent:N
\__text_expand_accent:NN
\__text_expand_letterlike:N
\__text_expand_letterlike:NN
\__text_expand_cs:N
\__text_expand_protect:w
\__text_expand_protect:N
\__text_expand_protect:nN
\__text_expand_protect:Nw
\__text_expand_testopt:N
\__text_expand_testopt:NNn
\__text_expand_encoding:N
\__text_expand_encoding_escape:N
\__text_expand_replace:N
\__text_expand_replace:n
\__text_expand_cs_expand:N
\__text_expand_unexpanded:w
\__text_expand_unexpanded_test:w
\__text_expand_unexpanded:N
\__text_expand_unexpanded:n

```

After precautions against & tokens, start a simple loop: that of course means that “text” cannot contain the two recursion quarks. The loop here must be f-type expandable; we have arbitrary user commands which might be protected *and* take arguments, and if the expansion code is used in a typesetting context, that will otherwise explode. (The same issue applies more clearly to case changing: see the example there.) The outer loop has to use scan marks as delimiters to protect against unterminated `\romannumeral` usage in the input.

```

30210 \cs_new:Npn \text_expand:n #1
30211 {
30212   \kernel_exp_not:w \exp_after:wN
30213   {
30214     \exp:w
30215     \__text_expand:n {#1}
30216   }
30217 }
30218 \cs_new:Npn \__text_expand:n #1
30219 {
30220   \group_align_safe_begin:
30221   \__text_expand_loop:w #1
30222   \s_text_recursion_tail \s_text_recursion_stop
30223   \__text_expand_result:n { }
30224 }

```

The approach to making the code f-type expandable is to use a marker result token and to shuffle the collected tokens

```

30225 \cs_new:Npn \__text_expand_store:n #1
30226   { \__text_expand_store:nw {#1} }
30227 \cs_generate_variant:Nn \__text_expand_store:n { o }
30228 \cs_new:Npn \__text_expand_store:nw #1#2 \__text_expand_result:n #3
30229   { #2 \__text_expand_result:n { #3 #1 } }
30230 \cs_new:Npn \__text_expand_end:w #1 \__text_expand_result:n #2
30231   {
30232     \group_align_safe_end:
30233     \exp_end:
30234     #2
30235   }

```

The main loop is a standard “tl action”; groups are handled recursively, while spaces are just passed through. Thus all of the action is in handling N-type tokens.

```

30236 \cs_new:Npn \__text_expand_loop:w #1 \s_text_recursion_stop
30237 {
30238   \tl_if_head_is_N_type:nTF {#1}
30239   { \__text_expand_N_type:N }
30240   {
30241     \tl_if_head_is_group:nTF {#1}
30242     { \__text_expand_group:n }
30243     { \__text_expand_space:w }
30244   }
30245 #1 \s_text_recursion_stop

```

```

30246    }
30247 \cs_new:Npn \__text_expand_group:n #1
30248 {
30249     \__text_expand_store:o
30250     {
30251         \exp_after:wN
30252         {
30253             \exp:w
30254             \__text_expand:n {#1}
30255         }
30256     }
30257     \__text_expand_loop:w
30258 }
30259 \exp_last_unbraced:NNo \cs_new:Npn \__text_expand_space:w \c_space_tl
30260 {
30261     \__text_expand_store:n { ~ }
30262     \__text_expand_loop:w
30263 }

```

Before we get into the real work, we have to watch out for problematic implicit characters: spaces and grouping tokens. Converting these to explicit characters later would lead to real issues as they are *not* N-type. A space is the easy case, so it's dealt with first: just insert the explicit token and continue the loop.

```

30264 \cs_new:Npx \__text_expand_N_type:N #1
30265 {
30266     \exp_not:N \__text_if_s_recursion_tail_stop_do:Nn #1
30267     { \exp_not:N \__text_expand_end:w }
30268 \exp_not:N \bool_lazy_any:nTF
30269 {
30270     { \exp_not:N \token_if_eq_meaning_p:NN #1 \c_space_token }
30271     {
30272         \exp_not:N \token_if_eq_meaning_p:NN #1
30273         \c_text_chardef_space_token
30274     }
30275     {
30276         \exp_not:N \token_if_eq_meaning_p:NN #1
30277         \c_text_mathchardef_space_token
30278     }
30279 }
30280 { \exp_not:N \__text_expand_space:w \c_space_tl }
30281 { \exp_not:N \__text_expand_N_type_auxi:N #1 }
30282 }

```

Implicit {} offer two issues. First, the token could be an implicit brace character: we need to avoid turning that into a brace group, so filter out the cases manually. Then we handle the case where an implicit group is present. That is done in an “open-ended” way: there’s the possibility the closing token is hidden somewhere.

```

30283 \cs_new:Npn \__text_expand_N_type_auxi:N #1
30284 {
30285     \bool_lazy_or:nnTF
30286     { \token_if_eq_meaning_p:NN #1 \c_text_chardef_group_begin_token }
30287     { \token_if_eq_meaning_p:NN #1 \c_text_mathchardef_group_begin_token }
30288     {
30289         \__text_expand_store:o \c_left_brace_str

```

```

30290           \__text_expand_loop:w
30291       }
30292   {
30293     \bool_lazy_or:nnTF
30294       { \token_if_eq_meaning_p:NN #1 \c_text_chardef_group_end_token }
30295       { \token_if_eq_meaning_p:NN #1 \c_text_mathchardef_group_end_token }
30296       {
30297         \__text_expand_store:o \c_right_brace_str
30298         \__text_expand_loop:w
30299       }
30300       { \__text_expand_N_type_auxii:N #1 }
30301     }
30302   }
30303 \cs_new:Npn \__text_expand_N_type_auxii:N #1
30304   {
30305     \token_if_eq_meaning:NNTF #1 \c_group_begin_token
30306     {
30307       { \if_false: } \fi:
30308       \__text_expand_loop:w
30309     }
30310     {
30311       \token_if_eq_meaning:NNTF #1 \c_group_end_token
30312       {
30313         \if_false: { \fi:
30314         \__text_expand_loop:w
30315       }
30316       { \__text_expand_N_type_auxiii:N #1 }
30317     }
30318   }

```

The first step in dealing with N-type tokens is to look for math mode material: that needs to be left alone. The starting function has to be split into two as we need \quark\_if\_recursion\_tail\_stop:N first before we can trigger the search. We then look for matching pairs of delimiters, allowing for the case where math mode starts but does not end. Within math mode, we simply pass all the tokens through unchanged, just checking the N-type ones against the end marker.

```

30319 \cs_new:Npn \__text_expand_N_type_auxiii:N #1
30320   {
30321     \exp_after:wN \__text_expand_math_search:NNN
30322     \exp_after:wN #1 \l_text_math_delims_tl
30323     \q_text_recursion_tail \q_text_recursion_tail
30324     \q_text_recursion_stop
30325   }
30326 \cs_new:Npn \__text_expand_math_search:NNN #1#2#3
30327   {
30328     \__text_if_q_recursion_tail_stop_do:Nn #2
30329     { \__text_expand_explicit:N #1 }
30330     \token_if_eq_meaning:NNTF #1 #2
30331     {
30332       \__text_use_i_delimit_by_q_recursion_stop:nw
30333       {
30334         \__text_expand_store:n {#1}
30335         \__text_expand_math_loop:Nw #3
30336       }

```

```

30337     }
30338     { \__text_expand_math_search:NNN #1 }
30339   }
30340 \cs_new:Npn \__text_expand_math_loop:Nw #1#2 \s__text_recursion_stop
30341   {
30342     \tl_if_head_is_N_type:nTF {#2}
30343     { \__text_expand_math_N_type:NN }
30344     {
30345       \tl_if_head_is_group:nTF {#2}
30346       { \__text_expand_math_group:Nn }
30347       { \__text_expand_math_space:Nw }
30348     }
30349     #1#2 \s__text_recursion_stop
30350   }
30351 \cs_new:Npn \__text_expand_math_N_type:NN #1#2
30352   {
30353     \__text_if_s_recursion_tail_stop_do:Nn #2
30354     { \__text_expand_end:w }
30355     \__text_expand_store:n {#2}
30356     \token_if_eq_meaning:NNTF #2 #1
30357     { \__text_expand_loop:w }
30358     { \__text_expand_math_loop:Nw #1 }
30359   }
30360 \cs_new:Npn \__text_expand_math_group:Nn #1#2
30361   {
30362     \__text_expand_store:n { {#2} }
30363     \__text_expand_math_loop:Nw #1
30364   }
30365 \exp_after:wN \cs_new:Npn \exp_after:wN \__text_expand_math_space:Nw
30366   \exp_after:wN # \exp_after:wN 1 \c_space_tl
30367   {
30368     \__text_expand_store:n { ~ }
30369     \__text_expand_math_loop:Nw #1
30370   }

```

At this stage, either we have a control sequence or a simple character: split and handle. The need to check for non-protected actives arises from handling of legacy input encodings: they need to end up in a representation we can deal with in further processing. The tests for explicit parts of the L<sup>A</sup>T<sub>E</sub>X 2<sub>C</sub> UTF-8 mechanism cover the case of bookmarks, where definitions change and are no longer protected. The same is true for babel shorthands.

```

30371 \cs_new:Npn \__text_expand_explicit:N #1
30372   {
30373     \token_if_cs:NTF #1
30374     { \__text_expand_exclude:N #1 }
30375     {
30376       \bool_lazy_and:nnTF
30377       { \token_if_active_p:N #1 }
30378       {
30379         ! \bool_lazy_any_p:n
30380         {
30381           { \token_if_protected_macro_p:N #1 }
30382           { \token_if_protected_long_macro_p:N #1 }
30383           { \tl_if_head_eq_meaning_p:on {#1} \UTFviii@two@octets }

```

```

30384     { \tl_if_head_eq_meaning_p:on {#1} \UTFviii@three@octets }
30385     { \tl_if_head_eq_meaning_p:on {#1} \UTFviii@four@octets }
30386     { \tl_if_head_eq_meaning_p:on {#1} \active@prefix }
30387   }
30388   }
30389   { \exp_after:wN \__text_expand_loop:w #1 }
30390   {
30391     \__text_expand_store:n {#1}
30392     \__text_expand_loop:w
30393   }
30394 }
30395 }
```

Next we exclude math commands: this is mainly as there *might* be an `\ensuremath`. The switching command for case needs special handling as it has to work by meaning.

```

30396 \cs_new:Npn \__text_expand_exclude:N #1
30397   {
30398     \cs_if_eq:NNTF #1 \text_case_switch:nnnn
30399     { \__text_expand_exclude_switch:Nnnnn #1 }
30400     {
30401       \exp_args:Ne \__text_expand_exclude:nN
30402       {
30403         \exp_not:V \l_text_math_arg_tl
30404         \exp_not:V \l_text_expand_exclude_tl
30405       }
30406       #1
30407     }
30408   }
30409 \cs_new:Npn \__text_expand_exclude_switch:Nnnnn #1#2#3#4#5
30410   {
30411     \__text_expand_store:n { #1 {#2} {#3} {#4} {#5} }
30412     \__text_expand_loop:w
30413   }
30414 \cs_new:Npn \__text_expand_exclude:nN #1#2
30415   {
30416     \__text_expand_exclude>NN #2 #1
30417     \q_text_recursion_tail \q_text_recursion_stop
30418   }
30419 \cs_new:Npn \__text_expand_exclude>NN #1#2
30420   {
30421     \__text_if_q_recursion_tail_stop_do:Nn #2
30422     { \__text_expand Accent:N #1 }
30423     \str_if_eq:nnTF {#1} {#2}
30424     {
30425       \__text_use_i_delimit_by_q_recursion_stop:nw
30426       { \__text_expand_exclude:Nw #1 }
30427     }
30428     { \__text_expand_exclude>NN #1 }
30429   }
30430 \cs_new:Npn \__text_expand_exclude:Nw #1#2#
30431   { \__text_expand_exclude:Nnn #1 {#2} }
30432 \cs_new:Npn \__text_expand_exclude:Nnn #1#2#3
30433   {
30434     \__text_expand_store:n { #1#2 {#3} }
```

```

30435     \__text_expand_loop:w
30436 }

```

Accents.

```

30437 \cs_new:Npn \__text_expand_accent:N #1
30438 {
30439     \exp_after:wN \__text_expand_accent:NN \exp_after:wN
30440         #1 \l_text_accents_tl
30441         \q_text_recursion_tail \q_text_recursion_stop
30442 }
30443 \cs_new:Npn \__text_expand_accent:NN #1#2
30444 {
30445     \__text_if_q_recursion_tail_stop_do:Nn #2
30446     { \__text_expand_letterlike:N #1 }
30447     \cs_if_eq:NNTF #2 #1
30448     {
30449         \__text_use_i_delimit_by_q_recursion_stop:nw
30450         {
30451             \__text_expand_store:n {#1}
30452             \__text_expand_loop:w
30453         }
30454     }
30455     { \__text_expand_accent:NN #1 }
30456 }

```

Another list of exceptions: these ones take no arguments so are easier to handle.

```

30457 \cs_new:Npn \__text_expand_letterlike:N #1
30458 {
30459     \exp_after:wN \__text_expand_letterlike:NN \exp_after:wN
30460         #1 \l_text_letterlike_tl
30461         \q_text_recursion_tail \q_text_recursion_stop
30462 }
30463 \cs_new:Npn \__text_expand_letterlike:NN #1#2
30464 {
30465     \__text_if_q_recursion_tail_stop_do:Nn #2
30466     { \__text_expand_cs:N #1 }
30467     \cs_if_eq:NNTF #2 #1
30468     {
30469         \__text_use_i_delimit_by_q_recursion_stop:nw
30470         {
30471             \__text_expand_store:n {#1}
30472             \__text_expand_loop:w
30473         }
30474     }
30475     { \__text_expand_letterlike:NN #1 }
30476 }

```

$\text{\LaTeX} 2\varepsilon$ 's `\protect` makes life interesting. Where possible, we simply remove it and replace with the “parent” command; of course, the `\protect` might be explicit, in which case we need to leave it alone. That includes the case where it's not even followed by an N-type token. There is also the case of a straight `\@protected@testopt` to cover.

```

30477 \cs_new:Npx \__text_expand_cs:N #1
30478 {
30479     \exp_not:N \str_if_eq:nnTF {#1} { \exp_not:N \protect }
30480     { \exp_not:N \__text_expand_protect:w }

```

```

30481 {
30482   \bool_lazy_and:nnTF
30483     { \cs_if_exist_p:N \fmtname }
30484     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
30485     { \exp_not:N \__text_expand_testopt:N #1 }
30486     { \exp_not:N \__text_expand_replace:N #1 }
30487   }
30488 }
30489 \cs_new:Npn \__text_expand_protect:w #1 \s__text_recursion_stop
30490 {
30491   \tl_if_head_is_N_type:nTF {#1}
30492     { \__text_expand_protect:N }
30493     {
30494       \__text_expand_store:n { \protect }
30495       \__text_expand_loop:w
30496     }
30497     #1 \s__text_recursion_stop
30498   }
30499 \cs_new:Npn \__text_expand_protect:N #1
30500 {
30501   \__text_if_s_recursion_tail_stop_do:Nn #1
30502   {
30503     \__text_expand_store:n { \protect }
30504     \__text_expand_end:w
30505   }
30506   \exp_args:Ne \__text_expand_protect:nN
30507     { \cs_to_str:N #1 } #1
30508   }
30509 \cs_new:Npn \__text_expand_protect:nN #1#2
30510   { \__text_expand_protect:Nw #2 #1 \q__text_nil #1 ~ \q__text_nil \q__text_nil \s__text_st
30511 \cs_new:Npn \__text_expand_protect:Nw #1 #2 ~ \q__text_nil #3 \q__text_nil #4 \s__text_stop
30512 {
30513   \__text_quark_if_nil:nTF {#4}
30514   {
30515     \cs_if_exist:cTF {#2}
30516       { \exp_args:Ne \__text_expand_store:n { \exp_not:c {#2} } }
30517       { \__text_expand_store:n { \protect #1 } }
30518   }
30519   { \__text_expand_store:n { \protect #1 } }
30520   \__text_expand_loop:w
30521 }
30522 \cs_new:Npn \__text_expand_testopt:N #1
30523 {
30524   \token_if_eq_meaning:NNTF #1 \@protected@testopt
30525     { \__text_expand_testopt>NNn }
30526     { \__text_expand_encoding:N #1 }
30527   }
30528 \cs_new:Npn \__text_expand_testopt>NNn #1#2#3
30529 {
30530   \__text_expand_store:n {#1}
30531   \__text_expand_loop:w
30532 }

```

Deal with encoding-specific commands

```

30533 \cs_new:Npn \__text_expand_encoding:N #1
30534 {
30535     \bool_lazy_or:nnTF
30536     { \cs_if_eq_p:NN #1 \@current@cmd }
30537     { \cs_if_eq_p>NN #1 \@changed@cmd }
30538     { \exp_after:wN \__text_expand_loop:w \__text_expand_encoding_escape:NN }
30539     { \__text_expand_replace:N #1 }
30540 }
30541 \cs_new:Npn \__text_expand_encoding_escape:NN #1#2 { \exp_not:n {#1} }

```

See if there is a dedicated replacement, and if there is, insert it.

```

30542 \cs_new:Npn \__text_expand_replace:N #1
30543 {
30544     \bool_lazy_and:nnTF
30545     { \cs_if_exist_p:c { l__text_expand_ \token_to_str:N #1 _tl } }
30546     {
30547         \bool_lazy_or_p:nn
30548         { \token_if_cs_p:N #1 }
30549         { \token_if_active_p:N #1 }
30550     }
30551     {
30552         \exp_args:Nv \__text_expand_replace:n
30553         { l__text_expand_ \token_to_str:N #1 _tl }
30554     }
30555     { \__text_expand_cs_expand:N #1 }
30556 }
30557 \cs_new:Npn \__text_expand_replace:n #1 { \__text_expand_loop:w #1 }

```

Finally, expand any macros which can be: this then loops back around to deal with what they produce. The only issue is if the token is `\exp_not:n`, as that must apply to the following balanced text.

```

30558 \cs_new:Npn \__text_expand_cs_expand:N #1
30559 {
30560     \__text_if_expandable:NTF #1
30561     {
30562         \token_if_eq_meaning:NNTF #1 \exp_not:n
30563         { \__text_expand_unexpanded:w }
30564         { \exp_after:wN \__text_expand_loop:w #1 }
30565     }
30566     {
30567         \__text_expand_store:n {#1}
30568         \__text_expand_loop:w
30569     }
30570 }

```

Since `\exp_not:n` is actually a primitive, it allows a strange syntax and it particular the primitive expands what follows and discards spaces and `\scan_stop:` until finding a braced argument (the opening brace can be implicit but we will not support this here). Here, we repeatedly f-expand after such an `\exp_not:n`, and test what follows. If it is a brace group, then we found the intended argument of `\exp_not:n`. If it is a space, then the next f-expansion will eliminate it. If it is an N-type token then `\__text_expand_unexpanded:N` leaves the token to be expanded if it is expandable, and otherwise removes it, assuming that it is `\scan_stop:`. This silently hides errors when `\exp_not:n`

is incorrectly followed by some non-expandable token other than `\scan_stop`, but this should be pretty rare, and there is no good error recovery anyways.

```

30571 \cs_new:Npn __text_expand_unexpanded:w
30572 {
30573     \exp_after:wN __text_expand_unexpanded_test:w
30574     \exp:w \exp_end_continue_f:w
30575 }
30576 \cs_new:Npn __text_expand_unexpanded_test:w #1 \s__text_recursion_stop
30577 {
30578     \tl_if_head_is_group:nTF {#1}
30579     { __text_expand_unexpanded:n }
30580     {
30581         __text_expand_unexpanded:w
30582         \tl_if_head_is_N_type:nT {#1} { __text_expand_unexpanded:N }
30583     }
30584     #1 \s__text_recursion_stop
30585 }
30586 \cs_new:Npn __text_expand_unexpanded:N #1
30587 {
30588     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
30589     \else:
30590         \exp_after:wN #1
30591     \fi:
30592 }
30593 \cs_new:Npn __text_expand_unexpanded:n #1
30594 {
30595     __text_expand_store:n {#1}
30596     __text_expand_loop:w
30597 }

```

(End definition for `\text_expand:n` and others. This function is documented on page 267.)

`\text_declare_expand_equivalent:Nn`

`\text_declare_expand_equivalent:cn`

Create equivalents to allow replacement.

```

30598 \cs_new_protected:Npn \text_declare_expand_equivalent:Nn #1#2
30599 {
30600     \tl_clear_new:c { l__text_expand_ \token_to_str:N #1 _tl }
30601     \tl_set:cn { l__text_expand_ \token_to_str:N #1 _tl } {#2}
30602 }
30603 \cs_generate_variant:Nn \text_declare_expand_equivalent:Nn { c }

```

(End definition for `\text_declare_expand_equivalent:Nn`. This function is documented on page 267.)

30604 `</package>`

# Chapter 82

## I3text-case implementation

```
30605  <*package>
30606  <@@=text>
```

### 82.1 Case changing

\l\_text\_titlecase\_check\_letter\_bool  
Needed to determine the route used in titlecasing.  
30607 \bool\_new:N \l\_text\_titlecase\_check\_letter\_bool  
30608 \bool\_set\_true:N \l\_text\_titlecase\_check\_letter\_bool

(End definition for \l\_text\_titlecase\_check\_letter\_bool. This variable is documented on page 270.)

\text\_lowercase:n  
\text\_uppercase:n  
\text\_titlecase:n  
\text\_titlecase\_first:n  
\text\_lowercase:nn  
\text\_uppercase:nn  
\text\_titlecase:nn  
\text\_titlecase\_first:nn  
The user level functions here are all wrappers around the internal functions for case changing.  
30609 \cs\_new:Npn \text\_lowercase:n #1  
30610 { \\_\_text\_change\_case:nnn { lower } { } {#1} }  
30611 \cs\_new:Npn \text\_uppercase:n #1  
30612 { \\_\_text\_change\_case:nnn { upper } { } {#1} }  
30613 \cs\_new:Npn \text\_titlecase:n #1  
30614 { \\_\_text\_change\_case:nnn { title } { } {#1} }  
30615 \cs\_new:Npn \text\_titlecase\_first:n #1  
30616 { \\_\_text\_change\_case:nnn { titleonly } { } {#1} }  
30617 \cs\_new:Npn \text\_lowercase:nn #1#2  
30618 { \\_\_text\_change\_case:nnn { lower } {#1} {#2} }  
30619 \cs\_new:Npn \text\_uppercase:nn #1#2  
30620 { \\_\_text\_change\_case:nnn { upper } {#1} {#2} }  
30621 \cs\_new:Npn \text\_titlecase:nn #1#2  
30622 { \\_\_text\_change\_case:nnn { title } {#1} {#2} }  
30623 \cs\_new:Npn \text\_titlecase\_first:nn #1#2  
30624 { \\_\_text\_change\_case:nnn { titleonly } {#1} {#2} }

(End definition for \text\_lowercase:n and others. These functions are documented on page 268.)

\\_\_text\_change\_case:nnn  
\\_\_text\_change\_case\_auxi:nnn  
\\_\_text\_change\_case\_auxii:nnn  
\\_\_text\_change\_case\_BCP:nnn  
\\_\_text\_change\_case\_BCP:nnw  
    \\_\_text\_change\_case\_BCP:nnnnw  
\\_\_text\_change\_case\_store:n  
\\_\_text\_change\_case\_store:o  
\\_\_text\_change\_case\_store:V  
\\_\_text\_change\_case\_store:v  
\\_\_text\_change\_case\_store:e  
\\_\_text\_change\_case\_store:nw  
\\_\_text\_change\_case\_result:n  
    \\_\_text\_change\_case\_end:w  
\\_\_text\_change\_case\_loop:nnw  
    \\_\_text\_change\_case\_break:w  
As for the expansion code, the business end of case changing is the handling of N-type tokens. First, we expand the input fully (so the loops here don't need to worry about awkward look-aheads and the like). Then we split into the different paths.

The code here needs to be f-type expandable to deal with the situation where case changing is applied in running text. There, we might have case changing as a document command and the text containing other non-expandable document commands.

```

\cs_set_eq:NN \MakeLowercase \text_lowercase
...
\MakeLowercase{\enquote*{A} text}

```

If we use an **e**-type expansion and wrap each token in `\exp_not:n`, that would explode: the document command grabs `\exp_not:n` as an argument, and things go badly wrong. So we have to wrap the entire result in exactly one `\exp_not:n`, or rather in the kernel version.

```

30625 \cs_new:Npn \__text_change_case:nnn #1#2#3
30626 {
30627   \__kernel_exp_not:w \exp_after:wN
30628   {
30629     \exp:w
30630     \exp_args:Ne \__text_change_case_auxi:nnn
30631     { \text_expand:n {#3} }
30632     {#1} {#2}
30633   }
30634 }
30635 \cs_new:Npn \__text_change_case_auxi:nnn #1#2#3
30636   { \exp_args:No \__text_change_case_BCP:nnn { \tl_to_str:n {#3} } {#1} {#2} }
30637 \cs_new:Npx \__text_change_case_BCP:nnn #1#2#3
30638   {
30639     \exp_not:N \__text_change_case_BCP:nnw
30640     {#2} {#3} #1 \tl_to_str:n { -x- -x- } \exp_not:N \q_text_stop
30641   }
30642 \use:x
30643   {
30644     \cs_new:Npn \exp_not:N \__text_change_case_BCP:nnw
30645       ##1##2##3 \tl_to_str:n { -x- } ##4 \tl_to_str:n { -x- } ##5
30646     \exp_not:N \q_text_stop
30647   }
30648   { \__text_change_case_BCP:nnnw {#1} {#2} {#4} {#3} #3 -- \q_text_stop }
30649 \cs_new:Npn \__text_change_case_BCP:nnnw #1#2#3#4#5 - #6 - #7 \q_text_stop
30650   {
30651     \cs_if_exist:cTF { __text_change_case_ #2 _ #5 -x- #3 :nnnn }
30652     { \__text_change_case_auxii:nnn {#1} {#2} { #5 -x- #3 } }
30653     { \__text_change_case_auxii:nnn {#1} {#2} {#4} }
30654   }
30655 \cs_new:Npn \__text_change_case_auxii:nnn #1#2#3
30656   {
30657     \group_align_safe_begin:
30658     \cs_if_exist_use:c { __text_change_case_boundary_ #2 _ #3 :Nnnw }
30659     \__text_change_case_loop:nnw {#2} {#3} #1
30660     \q_text_recursion_tail \q_text_recursion_stop
30661     \__text_change_case_result:n { }
30662   }

```

As for expansion, collect up the tokens for future use.

```

30663 \cs_new:Npn \__text_change_case_store:n #1
30664   { \__text_change_case_store:nw {#1} }
30665 \cs_generate_variant:Nn \__text_change_case_store:n { o , e , V , v }
30666 \cs_new:Npn \__text_change_case_store:nw #1#2 \__text_change_case_result:n #3
30667   { #2 \__text_change_case_result:n { #3 #1 } }
30668 \cs_new:Npn \__text_change_case_end:w #1 \__text_change_case_result:n #2

```

```

30669  {
30670  \group_align_safe_end:
30671  \exp_end:
30672  #2
30673  }

```

The main loop is the standard `tl` action type.

```

30674 \cs_new:Npn \__text_change_case_loop:nw #1#2#3 \q__text_recursion_stop
30675  {
30676  \tl_if_head_is_N_type:nTF {#3}
30677  { \__text_change_case_N_type:nnN }
30678  {
30679  \tl_if_head_is_group:nTF {#3}
30680  { \use:c { __text_change_case_group_ #1 :nnn } }
30681  { \__text_change_case_space:nw }
30682  }
30683  {#1} {#2} #3 \q__text_recursion_stop
30684  }
30685 \cs_new:Npn \__text_change_case_break:w #1 \q__text_recursion_tail \q__text_recursion_stop
30686  {
30687  \__text_change_case_store:n {#1}
30688  \__text_change_case_end:w
30689  }

```

For a group, we *could* worry about whether this contains a character or not. However, that would make life very complex for little gain: exactly what a first character is is rather weakly-defined anyway. So if there is a group, we simply assume that a character has been seen, and for title case we switch to the “rest of the tokens” situation. To avoid having too much testing, we use a two-step process here to allow the titlecase functions to be separate.

```

30690 \cs_new:Npn \__text_change_case_group_lower:nnn #1#2#3
30691  {
30692  \__text_change_case_store:o
30693  {
30694  \exp_after:wN
30695  {
30696  \exp:w
30697  \__text_change_case_auxii:nnn {#3} {#1} {#2}
30698  }
30699  }
30700  \__text_change_case_loop:nw {#1} {#2}
30701  }
30702 \cs_new_eq:NN \__text_change_case_group_upper:nnn
30703  \__text_change_case_group_lower:nnn
30704 \cs_new:Npn \__text_change_case_group_title:nnn #1#2#3
30705  {
30706  \__text_change_case_store:o
30707  {
30708  \exp_after:wN
30709  {
30710  \exp:w
30711  \__text_change_case_auxii:nnn {#3} {#1} {#2}
30712  }
30713  }

```

```

30714     \__text_change_case_loop:nw { lower } {#2}
30715   }
30716 \cs_new:Npn \__text_change_case_group_titleonly:nnn #1#2#3
30717   {
30718     \__text_change_case_store:o
30719   {
30720     \exp_after:wN
30721   {
30722     \exp:w
30723     \__text_change_case_auxii:nnn {#3} {#1} {#2}
30724   }
30725   }
30726   \__text_change_case_break:w
30727 }
30728 \use:x
30729 {
30730   \cs_new:Npn \exp_not:N \__text_change_case_space:nnw ##1##2 \c_space_tl
30731 }
30732 {
30733   \__text_change_case_store:n { ~ }
30734   \cs_if_exist_use:c { __text_change_case_boundary_ #1 _ #2 :Nnnw }
30735   \__text_change_case_loop:nw {#1} {#2}
30736 }

```

The first step of handling N-type tokens is to filter out the end-of-loop. That has to be done separately from the first real step as otherwise we pick up the wrong delimiter. The loop here is the same as the `expand` one, just passing the additional data long. If no close-math token is found then the final clean-up is forced (i.e. there is no assumption of “well-behaved” input in terms of math mode).

```

30737 \cs_new:Npn \__text_change_case_N_type:nnN #1#2#3
30738   {
30739     \__text_if_q_recursion_tail_stop_do:Nn #3
30740     { \__text_change_case_end:w }
30741     \__text_change_case_N_type_aux:nnN {#1} {#2} #3
30742   }
30743 \cs_new:Npn \__text_change_case_N_type_aux:nnN #1#2#3
30744   {
30745     \exp_args:NV \__text_change_case_N_type:nnnN
30746     \l_text_math_delims_tl {#1} {#2} #3
30747   }
30748 \cs_new:Npn \__text_change_case_N_type:nnnN #1#2#3#4
30749   {
30750     \__text_change_case_math_search:nnNNN {#2} {#3} #4 #1
30751     \q_text_recursion_tail \q_text_recursion_tail
30752     \q_text_recursion_stop
30753   }
30754 \cs_new:Npn \__text_change_case_math_search:nnNNN #1#2#3#4#5
30755   {
30756     \__text_if_q_recursion_tail_stop_do:Nn #4
30757     { \__text_change_case_cs_check:nnN {#1} {#2} #3 }
30758     \token_if_eq_meaning:NNTF #3 #4
30759   {
30760     \__text_use_i_delimit_by_q_recursion_stop:nw
30761   }

```

```

30762           \__text_change_case_store:n {#3}
30763           \__text_change_case_math_loop:nnNw {#1} {#2} #5
30764       }
30765   }
30766   { \__text_change_case_math_search:nnNNN {#1} {#2} #3 }
30767 }
30768 \cs_new:Npn \__text_change_case_math_loop:nnNw #1#2#3#4 \q__text_recursion_stop
30769 {
30770     \tl_if_head_is_N_type:nTF {#4}
30771     { \__text_change_case_math_N_type:nnNN }
30772     {
30773         \tl_if_head_is_group:nTF {#4}
30774         { \__text_change_case_math_group:nnNn }
30775         { \__text_change_case_math_space:nnNw }
30776     }
30777     {#1} {#2} #3 #4 \q__text_recursion_stop
30778 }
30779 \cs_new:Npn \__text_change_case_math_N_type:nnNN #1#2#3#4
30780 {
30781     \__text_if_q_recursion_tail_stop_do:Nn #4
30782     { \__text_change_case_end:w }
30783     \__text_change_case_store:n {#4}
30784     \token_if_eq_meaning:NNTF #4 #3
30785     { \__text_change_case_loop:nnw {#1} {#2} }
30786     { \__text_change_case_math_loop:nnNw {#1} {#2} #3 }
30787 }
30788 \cs_new:Npn \__text_change_case_math_group:nnNn #1#2#3#4
30789 {
30790     \__text_change_case_store:n { {#4} }
30791     \__text_change_case_math_loop:nnNw {#1} {#2} #3
30792 }
30793 \use:x
30794 {
30795     \cs_new:Npn \exp_not:N \__text_change_case_math_space:nnNw ##1##2##3
30796     \c_space_tl
30797 }
30798 {
30799     \__text_change_case_store:n { ~ }
30800     \__text_change_case_math_loop:nnNw {#1} {#2} #3
30801 }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: the two routes the code may take are then very different.

```

30802 \cs_new:Npn \__text_change_case_cs_check:nnN #1#2#3
30803 {
30804     \token_if_cs:NTF #3
30805     { \__text_change_case_exclude:nnN {#1} {#2} }
30806     {
30807         \__text_codepoint_process:nN
30808         { \use:c { __text_change_case_codepoint_ #1 :nnn } {#1} {#2} }
30809     }
30810     #3
30811 }

```

To deal with a control sequence there is first a need to test if it is on the list which

indicate that case changing should be skipped. That's done using a loop as for the other special cases. If a hit is found then the argument is grabbed and passed through as-is.

```

30812 \cs_new:Npn \__text_change_case_exclude:nnN #1#2#3
30813 {
30814     \exp_args:Ne \__text_change_case_exclude:nnnN
30815     {
30816         \exp_not:V \l_text_math_arg_tl
30817         \exp_not:V \l_text_case_exclude_arg_tl
30818     }
30819     {#1} {#2} #3
30820 }
30821 \cs_new:Npn \__text_change_case_exclude:nnnN #1#2#3#4
30822 {
30823     \__text_change_case_exclude:nnNN {#2} {#3} #4 #1
30824     \q_text_recursion_tail \q_text_recursion_stop
30825 }
30826 \cs_new:Npn \__text_change_case_exclude:nnNN #1#2#3#4
30827 {
30828     \__text_if_q_recursion_tail_stop_do:Nn #4
30829     { \__text_change_case_replace:nnN {#1} {#2} #3 }
30830     \str_if_eq:nnTF {#3} {#4}
30831     {
30832         \__text_use_i_delimit_by_q_recursion_stop:nw
30833         { \__text_change_case_exclude:nnNw {#1} {#2} #3 }
30834     }
30835     { \__text_change_case_exclude:nnNN {#1} {#2} #3 }
30836 }
30837 \cs_new:Npn \__text_change_case_exclude:nnNw #1#2#3#4#
30838 { \__text_change_case_exclude:nnNnn {#1} {#2} {#3} {#4} }
30839 \cs_new:Npn \__text_change_case_exclude:nnNnn #1#2#3#4#5
30840 {
30841     \tl_if_blank:nTF {#4}
30842     { \__text_change_case_store:n { #3 {#5} } }
30843     {
30844         \__text_change_case_store:o
30845         {
30846             \exp_after:wN #3
30847             \exp:w \__text_change_case_auxii:nnn {#4} {#1} {#2}
30848             {#5}
30849         }
30850     }
30851     \__text_change_case_loop:nnw {#1} {#2}
30852 }

```

Deal with any specialist replacement for case changing.

```

30853 \cs_new:Npn \__text_change_case_replace:nnN #1#2#3
30854 {
30855     \cs_if_exist:cTF { l__text_case_ \token_to_str:N #3 _tl }
30856     {
30857         \exp_args:Nv \__text_change_case_replace:nnn
30858         { l__text_case_ \token_to_str:N #3 _tl } {#1} {#2}
30859     }
30860     { \__text_change_case_switch:nnN {#1} {#2} #3 }
30861 }

```

```

30862 \cs_new:Npn \__text_change_case_replace:nnn #1#2#3
30863   { \__text_change_case_loop:nw {#2} {#3} #1 }
Allow for manually-controlled case switching.
30864 \cs_new:Npn \__text_change_case_switch:nnN #1#2#3
30865   {
30866     \cs_if_eq:NNTF #3 \text_case_switch:nnnn
30867       { \use:c { __text_change_case_switch_ #1 :nnNnnn } }
30868       { \use:c { __text_change_case_letterlike_ #1 :nnN } }
30869       {#1} {#2} #3
30870   }
30871 \cs_new:Npn \__text_change_case_switch_lower:nnNnnnn #1#2#3#4#5#6#7
30872   {
30873     \__text_change_case_store:n {#6}
30874     \__text_change_case_loop:nw {#1} {#2}
30875   }
30876 \cs_new:Npn \__text_change_case_switch_upper:nnNnnnn #1#2#3#4#5#6#7
30877   {
30878     \__text_change_case_store:n {#5}
30879     \__text_change_case_loop:nw {#1} {#2}
30880   }
30881 \cs_new:Npn \__text_change_case_switch_title:nnNnnnn #1#2#3#4#5#6#7
30882   {
30883     \__text_change_case_store:n {#7}
30884     \__text_change_case_loop:nw {#1} {#2}
30885   }
30886 \cs_new:Npn \__text_change_case_switch_titleonly:nnNnnnn #1#2#3#4#5#6#7
30887   {
30888     \__text_change_case_store:n {#7}
30889     \__text_change_case_break:w
30890   }

```

Letter-like commands may still be present: they are set up using a simple lookup approach, so can easily be handled with no loop. If there is no hit, we are at the end of the process: we loop around. Letter-like chars are all available only in upper- and lowercase, so titlecasing maps to the uppercase version.

```

30891 \cs_new:Npn \__text_change_case_letterlike_lower:nnN #1#2#3
30892   { \__text_change_case_letterlike:nnnnN {#1} {#1} {#1} {#2} #3 }
30893 \cs_new_eq:NN \__text_change_case_letterlike_upper:nnN
30894   \__text_change_case_letterlike_lower:nnN
30895 \cs_new:Npn \__text_change_case_letterlike_title:nnN #1#2#3
30896   { \__text_change_case_letterlike:nnnnN { upper } { lower } {#1} {#2} #3 }
30897 \cs_new:Npn \__text_change_case_letterlike_titleonly:nnN #1#2#3
30898   { \__text_change_case_letterlike:nnnnN { upper } { end } {#1} {#2} #3 }
30899 \cs_new:Npn \__text_change_case_letterlike:nnnnN #1#2#3#4#5
30900   {
30901     \cs_if_exist:cTF { c__text_ #1 case_ \token_to_str:N #5 _tl }
30902     {
30903       \__text_change_case_store:v
30904         { c__text_ #1 case_ \token_to_str:N #5 _tl }
30905         \use:c { __text_change_case_next_ #2 :nn } {#2} {#4}
30906     }
30907   {
30908     \__text_change_case_store:n {#5}
30909     \cs_if_exist:cTF

```

```

30910      {
30911          c__text_
30912          \str_if_eq:nNTF {#1} { lower } { upper } { lower }
30913          case_ \token_to_str:N #5 _tl
30914      }
30915      { \use:c { __text_change_case_next_ #2 :nn } {#2} {#4} }
30916      { \__text_change_case_loop:nnw {#3} {#4} }
30917  }
30918 }

```

For upper- and lowercase changes, once we get to this stage there are only a couple of questions remaining: is there a language-specific mapping and is there the special case of a terminal sigma. If not, then we pass to a simple codepoint mapping.

```

30919 \cs_new:Npn \__text_change_case_codepoint_lower:nnn #1#2#3
30920  {
30921      \cs_if_exist_use:cF { __text_change_case_lower_ #2 :nnnn }
30922      { \__text_change_case_lower_sigma:nnnn }
30923      {#1} {#1} {#2} {#3}
30924  }
30925 \cs_new:Npn \__text_change_case_codepoint_upper:nnn #1#2#3
30926  {
30927      \cs_if_exist_use:cF { __text_change_case_upper_ #2 :nnnn }
30928      { \__text_change_case_codepoint:nnnn }
30929      {#1} {#1} {#2} {#3}
30930  }

```

If the current character is an uppercase sigma, the a check is made on the next item in the input. If it is N-type and not a control sequence then there is a look-ahead phase: the logic here is simply based on letters or actives (to cover 8-bit engines).

```

30931 \cs_new:Npn \__text_change_case_lower_sigma:nnnn #1#2#3#4
30932  {
30933      \__text_codepoint_compare:nNnTF {#4} = { "03A3" }
30934      { \__text_change_case_lower_sigma:nnnw {#2} }
30935      { \__text_change_case_codepoint:nnnn {#1} {#2} }
30936      {#3} {#4}
30937  }
30938 \cs_new:Npn \__text_change_case_lower_sigma:nnnw #1#2#3#4 \q__text_recursion_stop
30939  {
30940      \tl_if_head_is_N_type:nTF {#4}
30941      { \__text_change_case_lower_sigma:nnnN {#3} }
30942      {
30943          \__text_change_case_store:
30944          { \codepoint_generate:nn { "03C2" } { \__text_char_catcode:N #3 } }
30945          \__text_change_case_loop:nnw
30946      }
30947      {#1} {#2} #4 \q__text_recursion_stop
30948  }
30949 \cs_new:Npn \__text_change_case_lower_sigma:nnnN #1#2#3#4
30950  {
30951      \__text_change_case_store:e
30952      {
30953          \bool_lazy_or:nnTF
30954          { \token_if_letter_p:N #4 }
30955          {
30956              \bool_lazy_and_p:nn

```

```

30957     { \token_if_active_p:N #4 }
30958     { \int_compare_p:nNn {'#4} > { "80 } }
30959   }
30960   { \codepoint_generate:nn { "03C3 } { \text_char_catcode:N #1 } }
30961   { \codepoint_generate:nn { "03C2 } { \text_char_catcode:N #1 } }
30962   }
30963   \text_change_case_loop:nnw {#2} {#3} #4
30964 }

For titlecasing, we need to fully expand the new character to see if it is a letter (or active).
30965 \cs_new:Npn \text_change_case_codepoint_title:nnn #1#2#3
30966   {
30967     \bool_if:NTF \l_text_titlecase_check_letter_bool
30968     {
30969       \tl_if_single:nTF {#3}
30970       {
30971         \bool_lazy_or:nnTF
30972           { \token_if_letter_p:N #3 }
30973           {
30974             \bool_lazy_and_p:nn
30975               { \token_if_active_p:N #3 }
30976               { ! \int_compare_p:nNn {'#3} < { "80 } }
30977             }
30978             { \use:c { \text_change_case_codepoint_ #1 :nn } }
30979             { \text_change_case_codepoint_title:nnnn { title } {#1} }
30980           }
30981           { \use:c { \text_change_case_codepoint_ #1 :nn } }
30982         }
30983         { \use:c { \text_change_case_codepoint_ #1 :nn } }
30984         {#2} {#3}
30985       }
30986     \cs_new_eq:NN \text_change_case_codepoint_titleonly:nnn
30987       \text_change_case_codepoint_title:nnn
30988     \cs_new:Npn \text_change_case_codepoint_title:nn #1#2
30989       { \text_change_case_codepoint_title:nnnn { title } { lower } {#1} {#2} }
30990     \cs_new:Npn \text_change_case_codepoint_titleonly:nn #1#2
30991       { \text_change_case_codepoint_title:nnnn { title } { end } {#1} {#2} }
30992     \cs_new:Npn \text_change_case_codepoint_title:nnnn #1#2#3#4
30993     {
30994       \cs_if_exist_use:cF { \text_change_case_title_ #3 :nnnn }
30995       {
30996         \cs_if_exist_use:cF { \text_change_case_upper_ #3 :nnnn }
30997           { \text_change_case_codepoint:nnnn }
30998       }
30999       {#1} {#2} {#3} {#4}
31000     }
31001 \cs_new:Npn \text_change_case_codepoint:nnnn #1#2#3#4
31002   {
31003     \bool_lazy_and:nnTF
31004       { \tl_if_single_p:n {#4} }
31005       { \token_if_active_p:N #4 }
31006       { \text_change_case_store:n {#4} }
31007       {
31008         \text_change_case_store:e
31009           { \text_change_case_codepoint:nn {#1} {#4} }

```

```

31010      }
31011      \use:c { __text_change_case_next_ #2 :nn } {#2} {#3}
31012    }
31013 \cs_new:Npn __text_change_case_codepoint:nn #1#2
31014  {
31015    __text_change_case_codepoint:fnn
31016    { \int_eval:n { __text_codepoint_from_chars:Nw #2 } } {#1} {#2}
31017  }
31018 \cs_new:Npn __text_change_case_codepoint:nnn #1#2#3
31019  {
31020    \exp_args:Ne __text_change_case_codepoint_aux:nn
31021    { __kernel_codepoint_case:nn { #2 case } {#1} } {#3}
31022  }
31023 \cs_generate_variant:Nn __text_change_case_codepoint:nnn { f }

Avoid high chars with pTEx.

31024 \sys_if_engine_ptex:T
31025  {
31026    \cs_new_eq:NN __text_change_case_codepoint_aux:nnn
31027      __text_change_case_codepoint:nnn
31028    \cs_gset:Npn __text_change_case_codepoint:nnn #1#2#3
31029    {
31030      \int_compare:nNnTF {#1} = { -1 }
31031      { \exp_not:n {#3} }
31032      { __text_change_case_codepoint_aux:nnn {#1} {#2} {#3} }
31033    }
31034  }
31035 \cs_new:Npn __text_change_case_codepoint_aux:nn #1#2
31036  {
31037    \use:e { __text_change_case_codepoint_aux:nnnn #1 {#2} }
31038  }
31039 \cs_new:Npn __text_change_case_codepoint_aux:nnnn #1#2#3#4
31040  {
31041    __text_codepoint_compare:nNnTF {#4} = {#1}
31042    { \exp_not:n {#4} }
31043    {
31044      \codepoint_generate:nn {#1}
31045      { __text_change_case_catcode:nn {#4} {#1} }
31046      \tl_if_blank:nF {#2}
31047      {
31048        \codepoint_generate:nn {#2}
31049        { \char_value_catcode:n {#2} }
31050        \tl_if_blank:nF {#3}
31051        {
31052          \codepoint_generate:nn {#3}
31053          { \char_value_catcode:n {#3} }
31054        }
31055      }
31056    }
31057  }

```

We need to ensure that only valid catcode-extraction is attempted. That's fine with Unicode engines but needs a bit of work with 8-bit ones. The logic is that if the original codepoint was in the ASCII range, we keep the catcode. Otherwise, if the target is in

the ASCII range, we use the standard catcode. If neither are true, we set as 13 on the grounds that this will be what is used anyway!

```

31058 \bool_lazy_or:nnTF
31059   { \sys_if_engine_luatex_p: }
31060   { \sys_if_engine_xetex_p: }
31061   {
31062     \cs_new:Npn \__text_change_case_catcode:nn #1#2
31063       { \__text_char_catcode:N #1 }
31064   }
31065   {
31066     \cs_new:Npn \__text_change_case_catcode:nn #1#2
31067       {
31068         \__text_codepoint_compare:nNnTF {#1} < { "80 "
31069           { \__text_char_catcode:N #1 }
31070           {
31071             \int_compare:nNnTF {#2} < { "80 "
31072               { \char_value_catcode:n {#2} }
31073               { 13 }
31074             }
31075           }
31076         }
31077       \cs_new:Npn \__text_change_case_next_lower:nn #1#2
31078         { \__text_change_case_loop:nnw {#1} {#2} }
31079       \cs_new_eq:NN \__text_change_case_next_upper:nn
31080         \__text_change_case_next_lower:nn
31081       \cs_new_eq:NN \__text_change_case_next_title:nn
31082         \__text_change_case_next_lower:nn
31083       \cs_new_eq:NN \__text_change_case_next_titleonly:nn
31084         \__text_change_case_next_lower:nn
31085       \cs_new:Npn \__text_change_case_next_end:nn #1#2
31086         { \__text_change_case_break:w }

```

(End definition for `\__text_change_case:nnn` and others.)

`\text_declare_case_equivalent:Nn` Create equivalents to allow replacement.

```

31087 \cs_new_protected:Npn \text_declare_case_equivalent:Nn #1#2
31088   {
31089     \tl_clear_new:c { l__text_case_ \token_to_str:N #1 _tl }
31090     \tl_set:cn { l__text_case_ \token_to_str:N #1 _tl } {#2}
31091   }

```

(End definition for `\text_declare_case_equivalent:Nn`. This function is documented on page 269.)

`\text_case_switch:nnnn` Set up the mechanism for manual case switching.

```

31092 \cs_new:Npn \text_case_switch:nnnn #1#2#3#4
31093   {
31094     \__text_case_switch_marker:
31095     #1
31096   }
31097 \cs_new:Npn \__text_case_switch_marker: { }

```

(End definition for `\text_case_switch:nnnn` and `\__text_case_switch_marker:`. This function is documented on page 269.)

```
\_text_change_case_upper_de-x-eszett:nnnn
\text_change_case_upper_de-alt:nnnn
```

A simple alternative version for German.

```
31098 \cs_new:cpn { __text_change_case_upper_de-x-eszett:nnnn } #1#2#3#4
31099   {
31100     \__text_codepoint_compare:nNnTF {#4} = { "00DF }
31101     {
31102       \__text_change_case_store:e
31103         {
31104           \codepoint_generate:nn { "1E9E }
31105             { \__text_change_case_catcode:nn {#4} { "1E9E } }
31106         }
31107         \use:c { __text_change_case_next_ #2 :nn }
31108           {#2} {#3}
31109       }
31110     { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31111   }
31112 \cs_new_eq:cc { __text_change_case_upper_de-alt:nnnn }
31113   { __text_change_case_upper_de-x-eszett:nnnn }
```

(End definition for `\__text_change_case_upper_de-x-eszett:nnnn` and `\__text_change_case_upper_de-alt:nnnn`.)

```
\text_change_case_upper_el:nnnn
\text_change_case_upper_el_aux:nnnn
\text_change_case_upper_el-x-iota:nnnn
\text_change_case_upper_el:nnn
\text_change_case_upper_el:nnnw
\text_change_case_upper_el:nnN
\text_change_case_upper_el_aux:nnnN
t_change_case_upper_el_ypogrammeni:nnnnnw
t_change_case_upper_el_ypogrammeni:nnnnnN
t_change_case_upper_el_ypogrammeni:nnnnnn
\text_change_case_upper_el_dialytika:nnn
\text_change_case_upper_el_dialytika:n
\text_change_case_upper_el_hiatus:nnnw
\text_change_case_upper_el_hiatus:nnN
\text_change_case_upper_el_hiatus:nnn
\text_change_case_upper_el_gobble:nnw
\text_change_case_upper_el_gobble:nnN
\text_change_case_upper_el_gobble:nnn
\text_change_case_if_greek:nTF
\text_change_case_if_greek:n
\text_change_case_if_greek_p:n
\text_change_case_if_greek:nTF
\text_change_case_if_greek_accent_p:n
\text_change_case_if_greek_accent:nTF
\text_change_case_if_greek_accent:n
\text_change_case_if_greek_diacritic_p:n
\text_change_case_if_greek_diacritic:nTF
\text_change_case_if_greek_diacritic:n
\text_change_case_if_takes_dialytika:nTF
\text_change_case_if_takes_dialytika:n
\text_change_case_if_takes_ypogrammeni:nTF
\text_change_case_if_takes_ypogrammeni:n
```

For Greek uppercasing, we need to know if characters *in the Greek range* have accents. That means doing a NFD conversion first, then starting a search. As described by the Unicode CLDR, Greek accents need to be found *after* any U+0308 (diaeresis) and are done in two groups to allow for the canonical ordering. The implementation here follows the data and examples from ICU (<https://icu.unicode.org/design/case/greek-upper>), although necessarily the implementation is somewhat different. The *ypogrammeni* is filtered out here as it is not actually in the Greek range, so gets lost if we leave until later.

```
31114 \cs_new:Npn \__text_change_case_upper_el:nnnn #1#2#3#4
31115   {
31116     \__text_change_case_if_greek:nTF {#4}
31117     {
31118       \exp_args:Ne \__text_change_case_upper_el:nnn
31119         {
31120           \codepoint_to_nfd:n { \__text_codepoint_from_chars:Nw #4 }
31121         }
31122         {#2} {#3}
31123     }
31124   {
31125     \__text_codepoint_compare:nNnTF {#4} = { "0345 }
31126       {
31127         \__text_change_case_store:e
31128           {
31129             \use:c { __text_change_case_upper_ #3 :n } {#4}
31130           }
31131         \__text_change_case_loop:nnw {#2} {#3}
31132       }
31133       { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31134     }
31135   }
31136 \cs_new_eq:cN { __text_change_case_upper_el-x-iota:nnnn }
31137   \__text_change_case_upper_el:nnnn
31138 \cs_new:Npn \__text_change_case_upper_el:n #1
```

```

31139  {
31140    \codepoint_generate:nn { "0399 "
31141      { \char_value_catcode:n { "0399 } }
31142    }
31143 \cs_new:cpn { __text_change_case_upper_el:x-iota:n } #1
31144   { \exp_not:n {#1} }
31145 \cs_new:Npn __text_change_case_upper_el:nnn #1#2#3
31146   {
31147     __text_codepoint_process:nN
31148       { __text_change_case_upper_el:nnnw {#2} {#3} } #1
31149   }

```

At this stage we have the first NFD codepoint as #3. What we need to know is whether after that we have another character, either from the NFD or directly in the input. If not, we store the changed character at this stage.

```

31150 \cs_new:Npn __text_change_case_upper_el:nnnw #1#2#3#4 \q__text_recursion_stop
31151  {
31152    \tl_if_head_is_N_type:nTF {#4}
31153      { __text_change_case_upper_el:nnnN {#3} }
31154    {
31155      __text_change_case_store:e
31156        { __text_change_case_codepoint:nn { upper } {#3} }
31157      __text_change_case_loop:nnw
31158    }
31159    {#1} {#2} #4 \q__text_recursion_stop
31160  }

```

Now, we check the detail of the next codepoint: again we filter out the not-a-char cases, before checking if it's an dialytika, accent or diacritic. (The latter do not have the same hiatus behavior as accents.) There is additional work if the codepoint can take a ypogegrammeni: there, we need to move any ypogegrammeni to after accents (in case the input is not normalised). The ypogegrammeni itself is handled separately.

```

31161 \cs_new:Npn __text_change_case_upper_el:nnnN #1#2#3#4
31162  {
31163    \token_if_cs:NTF #4
31164    {
31165      __text_change_case_store:e
31166        { __text_change_case_codepoint:nn { upper } {#1} }
31167      __text_change_case_loop:nnw {#2} {#3} #4
31168    }
31169    {
31170      __text_change_case_if_takes_ypogegrammeni:nTF {#1}
31171      {
31172        __text_change_case_upper_el_ypogegrammeni:nnnnnw
31173          {#1} {#2} {#3} { } { } #4
31174        }
31175        { __text_change_case_upper_el_aux:nnN {#1} {#2} {#3} #4 }
31176      }
31177    }
31178 \cs_new:Npn __text_change_case_upper_el_ypogegrammeni:nnnnnw
31179   #1#2#3#4#5#6 \q__text_recursion_stop
31180   {
31181     \tl_if_head_is_N_type:nTF {#6}
31182     {

```

```

31183     \__text_change_case_upper_el_ypogegrammeni:nnnnN
31184         {#1} {#2} {#3} {#4} {#5}
31185     }
31186     { \__text_change_case_upper_el_aux:nnnN {#1} {#2} {#3} #4#5 }
31187         #6 \q__text_recursion_stop
31188     }
31189 \cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:nnnnN #1#2#3#4#5#6
31190     {
31191         \token_if_cs:NTF #6
31192             { \__text_change_case_upper_el_aux:nnnN {#1} {#2} {#3} #4#5 #6 }
31193         {
31194             \__text_codepoint_process:nN
31195                 {
31196                     \__text_change_case_upper_el_ypogegrammeni:nnnnn
31197                         {#1} {#2} {#3} {#4} {#5}
31198                 }
31199                 #6
31200             }
31201         }
31202 \cs_new:Npn \__text_change_case_upper_el_ypogegrammeni:nnnnnn #1#2#3#4#5#6
31203     {
31204         \__text_codepoint_compare:nNnTF {#6} = { "0345" }
31205         {
31206             \__text_change_case_upper_el_ypogegrammeni:nnnnnw
31207                 {#1} {#2} {#3} {#4} {#6}
31208         }
31209     {
31210         \bool_lazy_or:nnTF
31211             { \__text_change_case_if_greek_accent_p:n {#6} }
31212             { \__text_change_case_if_greek_diacritic_p:n {#6} }
31213         {
31214             \__text_change_case_upper_el_ypogegrammeni:nnnnnw
31215                 {#1} {#2} {#3} {#4#6} {#5}
31216         }
31217         { \__text_change_case_upper_el_aux:nnnN {#1} {#2} {#3} #4#5 #6 }
31218     }
31219 }
31220 \cs_new:Npn \__text_change_case_upper_el_aux:nnnN #1#2#3#4
31221     {
31222         \__text_codepoint_process:nN
31223             { \__text_change_case_upper_el_aux:nnnn {#1} {#2} {#3} } #4
31224     }
31225 \cs_new:Npn \__text_change_case_upper_el_aux:nnnn #1#2#3#4
31226     {
31227         \__text_codepoint_compare:nNnTF {#4} = { "0308" }
31228             { \__text_change_case_upper_el_dialytika:nnn {#2} {#3} {#1} }
31229             {
31230                 \__text_change_case_if_greek_accent:nTF {#4}
31231                 { \__text_change_case_upper_el_hiatus:nnnw {#2} {#3} {#1} }
31232                 {
31233                     \__text_change_case_if_greek_diacritic:nTF {#4}
31234                     {
31235                         \__text_change_case_store:e
31236                             { \__text_change_case_codepoint:nn { upper } {#1} }
```

```

31237     \__text_change_case_loop:n nw {#2} {#3}
31238 }
31239 {
31240     \__text_codepoint_compare:nNnTF {#4} = { "0345" }
31241     { \__text_change_case_store:e { [XXX] } }
31242     {
31243         \__text_change_case_store:e
31244         { \__text_change_case_codepoint:nn { upper } {#1} }
31245     }
31246     \__text_change_case_loop:n nw {#2} {#3} #4
31247 }
31248 }
31249 }
31250 }
31251 % \end{macrocode}
31252 % We handle \emph{dialytika} in parts as it's also needed for the hiatus.
31253 % We know only two letters take it, so we can shortcut here on the second
31254 % part of the tests.
31255 % \begin{macrocode}
31256 \cs_new:Npn \__text_change_case_upper_el_dialytika:nnn #1#2#3
31257 {
31258     \__text_change_case_if_takes_dialytika:nTF {#3}
31259     { \__text_change_case_upper_el_dialytika:n {#3} }
31260     {
31261         \__text_change_case_store:e
31262         { \__text_change_case_codepoint:nn { upper } {#3} }
31263     }
31264     \__text_change_case_upper_el_gobble:n nw {#1} {#2}
31265 }
31266 \cs_new:Npn \__text_change_case_upper_el_dialytika:n #1
31267 {
31268     \__text_change_case_store:e
31269     {
31270         \bool_lazy_or:nnTF
31271         { \__text_codepoint_compare_p:nNn {#1} = { "0399" } }
31272         { \__text_codepoint_compare_p:nNn {#1} = { "03B9" } }
31273         {
31274             \codepoint_generate:nn { "03AA" }
31275             { \__text_change_case_catcode:nn {#1} { "03AA" } }
31276         }
31277         {
31278             \codepoint_generate:nn { "03AB" }
31279             { \__text_change_case_catcode:nn {#1} { "03AB" } }
31280         }
31281     }
31282 }

```

Adding a hiatus needs some of the same ideas, but if there is not one we skip this code point, hence needing a separate function.

```

31283 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnnw
31284 #1#2#3#4 \q_text_recursion_stop
31285 {
31286     \__text_change_case_store:e
31287     { \__text_change_case_codepoint:nn { upper } {#3} }

```

```

31288     \tl_if_head_is_N_type:nTF {#4}
31289     { \__text_change_case_upper_el_hiatus:nnN }
31290     { \__text_change_case_loop:nnw }
31291     {#1} {#2} #4 \q_text_recursion_stop
31292   }
31293 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnN #1#2#3
31294   {
31295     \token_if_cs:NTF #3
31296     { \__text_change_case_loop:nnw {#1} {#2} #3 }
31297     {
31298       \__text_codepoint_process:nn
31299       { \__text_change_case_upper_el_hiatus:nnn {#1} {#2} } #3
31300     }
31301   }
31302 \cs_new:Npn \__text_change_case_upper_el_hiatus:nnn #1#2#3
31303   {
31304     \__text_change_case_if_takes_dialytika:nTF {#3}
31305     {
31306       \__text_change_case_upper_el_dialytika:n {#3}
31307       \__text_change_case_upper_el_gobble:nnw {#1} {#2}
31308     }
31309     { \__text_change_case_loop:nnw {#1} {#2} #3 }
31310   }

```

For clearing out trailing combining marks after we have dealt with the first one.

```

31311 \cs_new:Npn \__text_change_case_upper_el_gobble:nnw
31312   #1#2#3 \q_text_recursion_stop
31313   {
31314     \tl_if_head_is_N_type:nTF {#3}
31315     { \__text_change_case_upper_el_gobble:nnN }
31316     { \__text_change_case_loop:nnw }
31317     {#1} {#2} #3 \q_text_recursion_stop
31318   }
31319 \cs_new:Npn \__text_change_case_upper_el_gobble:nnN #1#2#3
31320   {
31321     \token_if_cs:NTF #3
31322     { \__text_change_case_loop:nnw {#1} {#2} }
31323     {
31324       \__text_codepoint_process:nn
31325       { \__text_change_case_upper_el_gobble:nnn {#1} {#2} }
31326     }
31327     #3
31328   }
31329 \cs_new:Npn \__text_change_case_upper_el_gobble:nnn #1#2#3
31330   {
31331     \bool_lazy_or:nnTF
31332     { \__text_change_case_if_greek_accent_p:n {#3} }
31333     { \__text_change_case_if_greek_diacritic_p:n {#3} }
31334     { \__text_change_case_upper_el_gobble:nnw {#1} {#2} }
31335     { \__text_change_case_loop:nnw {#1} {#2} #3 }
31336   }

```

Luckily the Greek range is limited and clear.

```

31337 \prg_new_conditional:Npnn \__text_change_case_if_greek:n #1 { TF }
31338   {

```

```

31339     \exp_args:Nf \__text_change_case_if_greek:n
31340         { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
31341     }
31342 \cs_new:Npn \__text_change_case_if_greek:n #1
31343     {
31344         \if_int_compare:w #1 < "0370 \exp_stop_f:
31345             \prg_return_false:
31346         \else:
31347             \if_int_compare:w #1 > "03FF \exp_stop_f:
31348                 \if_int_compare:w #1 < "1F00 \exp_stop_f:
31349                     \prg_return_false:
31350                 \else:
31351                     \if_int_compare:w #1 > "1FFF \exp_stop_f:
31352                         \if_int_compare:w #1 = "2126 \exp_stop_f:
31353                             \prg_return_true:
31354                         \else:
31355                             \prg_return_false:
31356                         \fi:
31357                     \else:
31358                         \prg_return_true:
31359                     \fi:
31360                 \fi:
31361             \else:
31362                 \prg_return_true:
31363             \fi:
31364         \fi:
31365     }

```

We follow ICU in adding a few extras to the accent list here.

```

31366 \prg_new_conditional:Npnn \__text_change_case_if_greek_accent:n #1 { TF , p }
31367     {
31368         \exp_args:Nf \__text_change_case_if_greek_accent:n
31369             { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
31370     }
31371 \cs_new:Npn \__text_change_case_if_greek_accent:n #1
31372     {
31373         \if_int_compare:w #1 = "0300 \exp_stop_f:
31374             \prg_return_true:
31375         \else:
31376             \if_int_compare:w #1 = "0301 \exp_stop_f:
31377                 \prg_return_true:
31378             \else:
31379                 \if_int_compare:w #1 = "0342 \exp_stop_f:
31380                     \prg_return_true:
31381                 \else:
31382                     \if_int_compare:w #1 = "0302 \exp_stop_f:
31383                         \prg_return_true:
31384                     \else:
31385                         \if_int_compare:w #1 = "0303 \exp_stop_f:
31386                             \prg_return_true:
31387                         \else:
31388                             \if_int_compare:w #1 = "0311 \exp_stop_f:
31389                                 \prg_return_true:
31390                             \else:
31391                                 \prg_return_false:

```

```

31392           \fi:
31393           \fi:
31394           \fi:
31395           \fi:
31396           \fi:
31397           \fi:
31398       }
31399 \prg_new_if_greek_diacritic:n
31400   #1 { TF , p }
31401   {
31402     \exp_args:Nf \text_change_case_if_greek_diacritic:n
31403       { \int_eval:n { \text_codepoint_from_chars:Nw #1 } }
31404     }
31405 \cs_new:Npn \text_change_case_if_greek_diacritic:n #1
31406   {
31407     \if_int_compare:w #1 = "0304 \exp_stop_f:
31408       \prg_return_true:
31409     \else:
31410       \if_int_compare:w #1 = "0306 \exp_stop_f:
31411         \prg_return_true:
31412       \else:
31413         \if_int_compare:w #1 = "0313 \exp_stop_f:
31414           \prg_return_true:
31415         \else:
31416           \if_int_compare:w #1 = "0314 \exp_stop_f:
31417             \prg_return_true:
31418           \else:
31419             \if_int_compare:w #1 = "0343 \exp_stop_f:
31420               \prg_return_true:
31421             \else:
31422               \prg_return_false:
31423             \fi:
31424           \fi:
31425         \fi:
31426       \fi:
31427     \fi:
31428   }
31429 \prg_new_if_takes_dialytika:n
31430   #1 { TF }
31431   {
31432     \exp_args:Nf \text_change_case_if_takes_dialytika:n
31433       { \int_eval:n { \text_codepoint_from_chars:Nw #1 } }
31434     }
31435 \cs_new:Npn \text_change_case_if_takes_dialytika:n #1
31436   {
31437     \if_int_compare:w #1 = "0399 \exp_stop_f:
31438       \prg_return_true:
31439     \else:
31440       \if_int_compare:w #1 = "03B9 \exp_stop_f:
31441         \prg_return_true:
31442       \else:
31443         \if_int_compare:w #1 = "03A5 \exp_stop_f:
31444           \prg_return_true:
31445         \else:

```

```

31446          \prg_return_true:
31447      \else:
31448          \prg_return_false:
31449      \fi:
31450          \fi:
31451      \fi:
31452      \fi:
31453  }
31454 \prg_new_conditional:Npnn \__text_change_case_if_takes_ypogegrammeni:n #1 { TF }
31455  {
31456      \exp_args:Nf \__text_change_case_if_takes_ypogegrammeni:n
31457      { \int_eval:n { \__text_codepoint_from_chars:Nw #1 } }
31458  }
31459 \cs_new:Npn \__text_change_case_if_takes_ypogegrammeni:n #1
31460  {
31461      \if_int_compare:w #1 = "03B1 \exp_stop_f:
31462          \prg_return_true:
31463      \else:
31464          \if_int_compare:w #1 = "03B7 \exp_stop_f:
31465              \prg_return_true:
31466          \else:
31467              \if_int_compare:w #1 = "03C9 \exp_stop_f:
31468                  \prg_return_true:
31469              \else:
31470                  \prg_return_false:
31471              \fi:
31472          \fi:
31473      \fi:
31474  }

```

(End definition for `\__text_change_case_upper_el:nnnn` and others.)

`\__text_change_case_boundary_upper_el:Nnnw`  
`\__text_change_case_boundary_upper_el:nnN`  
`\__text_change_case_boundary_upper_el:nnnN`  
`\__text_change_case_boundary_upper_el:nnnw`  
`\__text_change_case_boundary_upper_el:nnnN`

There is one special case in Greek that needs to be picked up based on being an isolated letter. We do that using a test similar to final sigma, but it has to fire off from the space grabber.

```

31475 \cs_new:Npn \__text_change_case_boundary_upper_el:Nnnw
31476  #1#2#3#4 \q_text_recursion_stop
31477  {
31478      \tl_if_head_is_N_type:NTF {#4}
31479      { \__text_change_case_boundary_upper_el:nnN }
31480      { \__text_change_case_loop:nnw }
31481      {#2} {#3} #4 \q_text_recursion_stop
31482  }
31483 \cs_new:Npn \__text_change_case_boundary_upper_el:nnN #1#2#3
31484  {
31485      \token_if_cs:NTF #3
31486      { \__text_change_case_loop:nnw {#1} {#2} }
31487      {
31488          \__text_codepoint_process:nN
31489          { \__text_change_case_boundary_upper_el:nnn {#1} {#2} }
31490      }
31491      #3
31492  }
31493 \cs_new:Npn \__text_change_case_boundary_upper_el:nnn #1#2#3

```

```

31494 {
31495   \bool_lazy_or:nnTF
31496     { \__text_codepoint_compare_p:nNn {#3} = { "03AE" } }
31497     { \__text_codepoint_compare_p:nNn {#3} = { "1F22" } }
31498     { \__text_change_case_boundary_upper_el:nnnw {#1} {#2} {#3} }
31499     { \__text_change_case_loop:nnw {#1} {#2} #3 }
31500   }
31501 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnw
31502   #1#2#3#4 \q__text_recursion_stop
31503   {
31504     \tl_if_head_is_N_type:nTF {#4}
31505       { \__text_change_case_boundary_upper_el:nnnN {#3} }
31506       {
31507         \__text_change_case_store:e
31508         {
31509           \codepoint_generate:nn { "0389" }
31510             { \__text_change_case_catcode:nn {#3} { "0389" } }
31511           }
31512           \__text_change_case_loop:nnw
31513         }
31514         {#1} {#2} #4 \q__text_recursion_stop
31515       }
31516 \cs_new:Npn \__text_change_case_boundary_upper_el:nnnN #1#2#3#4
31517   {
31518     \MARJ
31519     \bool_lazy_or:nnTF
31520       { \token_if_letter_p:N #4 }
31521       {
31522         \bool_lazy_and_p:nn
31523           { \token_if_active_p:N #4 }
31524           { ! \int_compare_p:nNn {'#4} < { "80" } }
31525       }
31526       { \__text_change_case_loop:nnw {#2} {#3} #1#4 }
31527     {
31528       \__text_change_case_store:e
31529       {
31530         \codepoint_generate:nn { "0389" }
31531           { \__text_change_case_catcode:nn {#1} { "0389" } }
31532         }
31533         \__text_change_case_loop:nnw {#2} {#3} #4
31534       }
31535     }

```

(End definition for `\__text_change_case_boundary_upper_el:Nnnw` and others.)

`\__text_change_case_title_el:nnnn` Titlecasing retains accents, but to prevent the uppercasing code from kicking in, there has to be an explicit function here.

```

31536 \cs_new:Npn \__text_change_case_title_el:nnnn #1#2#3#4
31537   { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }

```

(End definition for `\__text_change_case_title_el:nnnn`.)

See <https://www.unicode.org/L2/L2020/20143-armenian-ech-yiwn.pdf>.

```

31538 \cs_new:Npn \__text_change_case_upper_hy:nnnn #1#2#3#4

```

```

\__text_change_case_upper_hy:nnnn
\__text_change_case_title_hy:nnnn
\__text_change_case_upper_hy-x-yiwn:nnnn
\__text_change_case_title_hy-x-yiwn:nnnn

```

```

31539 {
31540   \__text_codepoint_compare:nNnTF {#4} = { "0587 }
31541   {
31542     \__text_change_case_store:e
31543     {
31544       \codepoint_generate:nn { "0535 }
31545       { \__text_change_case_catcode:nn {#4} { "0535 } }
31546       \codepoint_generate:nn { "054E }
31547       { \__text_change_case_catcode:nn {#4} { "054E } }
31548     }
31549     \use:c { __text_change_case_next_ #2 :nn }
31550     {#2} {#3}
31551   }
31552   { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31553 }
31554 \cs_new:Npn \__text_change_case_title_hy:nnnn #1#2#3#4
31555 {
31556   \__text_codepoint_compare:nNnTF {#4} = { "0587 }
31557   {
31558     \__text_change_case_store:e
31559     {
31560       \codepoint_generate:nn { "0535 }
31561       { \__text_change_case_catcode:nn {#4} { "0535 } }
31562       \codepoint_generate:nn { "057E }
31563       { \__text_change_case_catcode:nn {#4} { "057E } }
31564     }
31565     \use:c { __text_change_case_next_ #2 :nn }
31566     {#2} {#3}
31567   }
31568   { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31569 }
31570 \cs_new:cpn { __text_change_case_upper_hy-x-yiwn:nnnn } #1#2#3#4
31571   { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31572 \cs_new_eq:cc { __text_change_case_title_hy-x-yiwn:nnnn }
31573   { __text_change_case_upper_hy-x-yiwn:nnnn }

(End definition for \__text_change_case_upper_hy:nnnn and others.)

```

Simply swaps of characters.

```

31574 \cs_new:cpn { __text_change_case_lower_la-x-medieval:nnnn } #1#2#3#4
31575 {
31576   \__text_codepoint_compare:nNnTF {#4} = { "0056 }
31577   {
31578     \__text_change_case_store:e
31579     { \char_generate:nn { "0075 } { \__text_char_catcode:N #4 } }
31580     \use:c { __text_change_case_next_ #2 :nn }
31581     {#2} {#3}
31582   }
31583   { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31584 }
31585 \cs_new:cpn { __text_change_case_upper_la-x-medieval:nnnn } #1#2#3#4
31586 {
31587   \__text_codepoint_compare:nNnTF {#4} = { "0075 }
31588   {

```

```

31589     \__text_change_case_store:e
31590         { \char_generate:nn { "0056 } { \__text_char_catcode:N #4 } }
31591         \use:c { __text_change_case_next_ #2 :nn }
31592             {#2} {#3}
31593     }
31594     { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31595 }
```

(End definition for `\__text_change_case_lower_lt-x-medieval:nnnn` and `\__text_change_case_upper_-lt-x-medieval:nnnn`.)

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. The first step is a simple match attempt: look for the three uppercase accented letters which should gain a dot-above char in their lowercase form.

```

31596 \cs_new:Npn \__text_change_case_lower_lt:nnnn
31597 {
31598     \exp_args:Ne \__text_change_case_lower_lt_auxi:nnnn
31599     {
31600         \int_case:nn { \__text_codepoint_from_chars:Nw #4 }
31601             {
31602                 { "00CC } { "0300 }
31603                 { "00CD } { "0301 }
31604                 { "0128 } { "0303 }
31605             }
31606         }
31607     {#2} {#3} {#4}
31608 }
```

If there was a hit, output the result with the dot-above and move on. Otherwise, look for one of the three letters that can take a combining accent: I, J and I-ogonek.

```

31609 \cs_new:Npn \__text_change_case_lower_lt_auxi:nnnn #1#2#3#4
31610 {
31611     \tl_if_blank:nTF {#1}
31612     {
31613         \exp_args:Ne \__text_change_case_lower_lt_auxii:nnnn
31614         {
31615             \int_case:nn { \__text_codepoint_from_chars:Nw #4 }
31616                 {
31617                     { "0049 } { "0069 }
31618                     { "004A } { "006A }
31619                     { "012E } { "012F }
31620                 }
31621             }
31622         {#2} {#3} {#4}
31623     }
31624     {
31625         \__text_change_case_store:e
31626             {
31627                 \codepoint_generate:nn { "0069 }
31628                     { \__text_change_case_catcode:nn {#4} { "0069 } }
31629                 \codepoint_generate:nn { "0307 }
31630                     { \__text_change_case_catcode:nn {#4} { "0307 } }
31631                 \codepoint_generate:nn {#1}
```

```

31632         { \__text_change_case_catcode:nn {#4} {#1} }
31633     }
31634     \__text_change_case_loop:nnw {#2} {#3}
31635   }
31636 }

```

Again, branch depending on a hit. If there is one, we output the character then need to look for a combining accent: as usual, we need to be aware of the loop situation.

```

31637 \cs_new:Npn \__text_change_case_lower_lt_auxii:nnnn #1#2#3#4
31638   {
31639     \tl_if_blank:nTF {#1}
31640       { \__text_change_case_codepoint:nnnn {#2} {#2} {#3} {#4} }
31641       {
31642         \__text_change_case_store:e
31643         {
31644           \codepoint_generate:nn {#1}
31645             { \__text_change_case_catcode:nn {#4} {#1} }
31646           }
31647         \__text_change_case_lower_lt:nnw {#2} {#3}
31648       }
31649     }
31650 \cs_new:Npn \__text_change_case_lower_lt:nnw #1#2#3 \q__text_recursion_stop
31651   {
31652     \tl_if_head_is_N_type:nTF {#3}
31653       { \__text_change_case_lower_lt:nnN }
31654       { \__text_change_case_loop:nnw }
31655         {#1} {#2} #3 \q__text_recursion_stop
31656     }
31657 \cs_new:Npn \__text_change_case_lower_lt:nnN #1#2#3
31658   {
31659     \__text_codepoint_process:nn
31660       { \__text_change_case_lower_lt:nnn {#1} {#2} } #3
31661   }
31662 \cs_new:Npn \__text_change_case_lower_lt:nnn #1#2#3
31663   {
31664     \bool_lazy_and:nnT
31665     {
31666       \bool_lazy_or_p:nn
31667         { ! \tl_if_single_p:n {#3} }
31668         { ! \token_if_cs_p:N #3 }
31669     }
31670   {
31671     \bool_lazy_any_p:n
31672     {
31673       { \__text_codepoint_compare_p:nNn {#3} = { "0300" } }
31674       { \__text_codepoint_compare_p:nNn {#3} = { "0301" } }
31675       { \__text_codepoint_compare_p:nNn {#3} = { "0303" } }
31676     }
31677   }
31678   {
31679     \__text_change_case_store:e
31680     {
31681       \codepoint_generate:nn { "0307" }
31682         { \__text_change_case_catcode:nn {#3} { "0307" } }

```

```

31683         }
31684     }
31685     \__text_change_case_loop:nw {#1} {#2} #3
31686 }

```

(End definition for `\__text_change_cases_lower_lt:nnnn` and others.)

The uppercasing version: first find i/j/i-ogonek, then look for the combining char: drop it if present.

```

31687 \cs_new:Npn \__text_change_case_upper_lt:nnnn #1#2#3#4
31688 {
31689     \exp_args:Ne \__text_change_case_upper_lt_aux:nnnn
31690     {
31691         \int_case:nn { \__text_codepoint_from_chars:Nw #4 }
31692         {
31693             { "0069" } { "0049" }
31694             { "006A" } { "004A" }
31695             { "012F" } { "012E" }
31696         }
31697     }
31698     {#2} {#3} {#4}
31699 }
31700 \cs_new:Npn \__text_change_case_upper_lt_aux:nnnn #1#2#3#4
31701 {
31702     \tl_if_blank:nTF {#1}
31703     { \__text_change_case_codepoint:nnnn { upper } {#2} {#3} {#4} }
31704     {
31705         \__text_change_case_store:e
31706         {
31707             \codepoint_generate:nn {#1}
31708             { \__text_change_case_catcode:nn {#4} {#1} }
31709         }
31710         \__text_change_case_upper_lt:nw {#2} {#3}
31711     }
31712 }
31713 \cs_new:Npn \__text_change_case_upper_lt:nnw #1#2#3 \q__text_recursion_stop
31714 {
31715     \tl_if_head_is_N_type:nTF {#3}
31716     { \__text_change_case_upper_lt:nnN }
31717     { \use:c { __text_change_case_next_ #1 :nn } }
31718     {#1} {#2} #3 \q__text_recursion_stop
31719 }
31720 \cs_new:Npn \__text_change_case_upper_lt:nnN #1#2#3
31721 {
31722     \__text_codepoint_process:nN
31723     { \__text_change_case_upper_lt:nnn {#1} {#2} } #3
31724 }
31725 \cs_new:Npn \__text_change_case_upper_lt:nnn #1#2#3
31726 {
31727     \bool_lazy_and:nnTF
31728     {
31729         \bool_lazy_or_p:nn
31730         { ! \tl_if_single_p:n {#3} }
31731         { ! \token_if_cs_p:N #3 }

```

```

31732 }
31733 { \__text_codepoint_compare_p:nNn {#3} = { "0307" } }
31734 { \use:c { __text_change_case_next_ #1 :nn } {#1} {#2} }
31735 { \use:c { __text_change_case_next_ #1 :nn } {#1} {#2} #3 }
31736 }

```

(End definition for `\__text_change_cases_upper_lt:nnnn` and others.)

For Dutch, there is a single look-ahead test for ij when title casing. If the appropriate letters are found, produce IJ and gobble the j/J.

```

\__text_change_case_title_nl:nnnn
\__text_change_case_title_nl_aux:nnnn
\__text_change_case_title_nl:nnN
31737 \cs_new:Npn \__text_change_case_title_nl:nnnn #1#2#3#4
31738 {
31739     \tl_if_single:nTF {#4}
31740     { \__text_change_case_title_nl_aux:nnnn }
31741     { \__text_change_case_codepoint:nnnn }
31742     {#1} {#2} {#3} {#4}
31743 }
31744 \cs_new:Npn \__text_change_case_title_nl_aux:nnnn #1#2#3#4
31745 {
31746     \bool_lazy_or:nnTF
31747     { \int_compare_p:nNn {'#4} = { "0049" } }
31748     { \int_compare_p:nNn {'#4} = { "0069" } }
31749     {
31750         \__text_change_case_store:e
31751         { \char_generate:nn { "0049" } { \__text_char_catcode:N #4 } }
31752         \__text_change_case_title_nl:nnw {#2} {#3}
31753     }
31754     { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31755 }
31756 \cs_new:Npn \__text_change_case_title_nl:nnw #1#2#3 \q__text_recursion_stop
31757 {
31758     \tl_if_head_is_N_type:nTF {#3}
31759     { \__text_change_case_title_nl:nnN }
31760     { \use:c { __text_change_case_next_ #1 :nn } }
31761     {#1} {#2} #3 \q__text_recursion_stop
31762 }
31763 \cs_new:Npn \__text_change_case_title_nl:nnN #1#2#3
31764 {
31765     \bool_lazy_and:nnTF
31766     { ! \token_if_cs_p:N #3 }
31767     {
31768         \bool_lazy_or_p:nn
31769         { \int_compare_p:nNn {'#3} = { "004A" } }
31770         { \int_compare_p:nNn {'#3} = { "006A" } }
31771     }
31772     {
31773         \__text_change_case_store:e
31774         { \char_generate:nn { "004A" } { \__text_char_catcode:N #3 } }
31775         \use:c { __text_change_case_next_ #1 :nn } {#1} {#2}
31776     }
31777     { \use:c { __text_change_case_next_ #1 :nn } {#1} {#2} #3 }
31778 }

```

(End definition for `\__text_change_case_title_nl:nnnn` and others.)

```
\_text_change_case_lower_tr:nnnn
\text_change_case_lower_tr:nnNw
\text_change_case_lower_tr:NnnN
\text_change_case_lower_tr:Nnnn
```

The Turkic languages need special treatment for dotted-i and dotless-i. The lower casing rule can be expressed in terms of searching first for either a dotless-I or a dotted-I. In the latter case the mapping is easy, but in the former there is a second stage search.

```
31779 \cs_new:Npn \text_change_case_lower_tr:nnnn #1#2#3#4
31780 {
31781     \text_codepoint_compare:nNnTF {#4} = { "0049 }
31782     { \text_change_case_lower_tr:nnNw {#1} {#3} #4 }
31783     {
31784         \text_codepoint_compare:nNnTF {#4} = { "0130 }
31785         {
31786             \text_change_case_store:e
31787             {
31788                 \codepoint_generate:nn { "0069 }
31789                 { \text_change_case_catcode:nn {#4} { "0069 } }
31790             }
31791             \text_change_case_loop:nnw {#1} {#3}
31792         }
31793         { \text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31794     }
31795 }
```

After a dotless-I there may be a dot-above character. If there is then a dotted-i should be produced, otherwise output a dotless-i. When the combination is found both the dotless-I and the dot-above char have to be removed from the input.

```
31796 \cs_new:Npn \text_change_case_lower_tr:nnNw #1#2#3#4 \q_text_recursion_stop
31797 {
31798     \tl_if_head_is_N_type:nTF {#4}
31799     { \text_change_case_lower_tr:NnnN #3 {#1} {#2} }
31800     {
31801         \text_change_case_store:e
31802         {
31803             \codepoint_generate:nn { "0131 }
31804             { \text_change_case_catcode:nn {#3} { "0131 } }
31805         }
31806         \text_change_case_loop:nnw {#1} {#2}
31807     }
31808     #4 \q_text_recursion_stop
31809 }
31810 \cs_new:Npn \text_change_case_lower_tr:NnnN #1#2#3#4
31811 {
31812     \text_codepoint_process:nN
31813     { \text_change_case_lower_tr:Nnnn #1 {#2} {#3} } #4
31814 }
31815 \cs_new:Npn \text_change_case_lower_tr:Nnnn #1#2#3#4
31816 {
31817     \bool_lazy_or:nnTF
31818     {
31819         \bool_lazy_and_p:nn
31820         { \tl_if_single_p:n {#4} }
31821         { \token_if_cs_p:N #4 }
31822     }
31823     { ! \text_codepoint_compare_p:nNn {#4} = { "0307 } }
31824 {
31825     \text_change_case_store:e
```

```

31826    {
31827        \codepoint_generate:nn { "0131 }
31828            { \__text_change_case_catcode:nn {#1} { "0131 } }
31829    }
31830    \__text_change_case_loop:nnw {#2} {#3} #4
31831 }
31832 {
31833     \__text_change_case_store:e
31834         {
31835             \codepoint_generate:nn { "0069 }
31836                 { \__text_change_case_catcode:nn {#1} { "0069 } }
31837             }
31838             \__text_change_case_loop:nnw {#2} {#3}
31839         }
31840     }

```

(End definition for `\__text_change_case_lower_tr:nnnn` and others.)

`\__text_change_case_upper_tr:nnnn` Uppercasing is easier: just one exception with no context.

```

31841 \cs_new:Npn \__text_change_case_upper_tr:nnnn #1#2#3#4
31842 {
31843     \__text_codepoint_compare:nNnTF {#4} = { "0069 }
31844     {
31845         \__text_change_case_store:e
31846             {
31847                 \codepoint_generate:nn { "0130 }
31848                     { \__text_change_case_catcode:nn {#4} { "0130 } }
31849                 }
31850                 \use:c { __text_change_case_next_ #2 :nn } {#2} {#3}
31851             }
31852             { \__text_change_case_codepoint:nnnn {#1} {#2} {#3} {#4} }
31853     }

```

(End definition for `\__text_change_case_upper_tr:nnnn`.)

`\__text_change_case_lower_az:nnnn`  
`\__text_change_case_upper_az:nnnn` Straight copies.

```

31854 \cs_new_eq:NN \__text_change_case_lower_az:nnnn
31855     \__text_change_case_lower_tr:nnnn
31856 \cs_new_eq:NN \__text_change_case_upper_az:nnnn
31857     \__text_change_case_upper_tr:nnnn

```

(End definition for `\__text_change_case_lower_az:nnnn` and `\__text_change_case_upper_az:nnnn`.)

The (fixed) look-up mappings for letter-like control sequences.

```

31858 \group_begin:
31859     \cs_set_protected:Npn \__text_change_case_setup:NN #1#2
31860     {
31861         \quark_if_recursion_tail_stop:N #1
31862         \tl_const:cn { c__text_lowercase_ \token_to_str:N #1 _tl }
31863             { #2 }
31864         \tl_const:cn { c__text_uppercase_ \token_to_str:N #2 _tl }
31865             { #1 }
31866         \__text_change_case_setup:NN
31867     }
31868 \__text_change_case_setup:NN

```

```

31869  \AA \aa
31870  \AE \ae
31871  \DH \dh
31872  \DJ \dj
31873  \IJ \ij
31874  \L \l
31875  \NG \ng
31876  \O \o
31877  \OE \oe
31878  \SS \ss
31879  \TH \th
31880  \q_recursion_tail ?
31881  \q_recursion_stop
31882  \tl_if_exist:cn { c__text_uppercase_ \token_to_str:N \i _tl } { I }
31883  \tl_if_exist:cn { c__text_uppercase_ \token_to_str:N \j _tl } { J }
31884 \group_end:

```

To deal with possible encoding-specific extensions to `\@uclclist`, we check at the end of the preamble. This will therefore only apply to L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package mode.

```

31885 \tl_if_exist:NT \expl@finalise@setup@@
31886  {
31887   \tl_gput_right:Nn \expl@finalise@setup@@
31888   {
31889     \tl_gput_right:Nn \kernel@after@begindocument
31890     {
31891       \group_begin:
31892         \cs_set_protected:Npn \__text_change_case_setup:N #1#2
31893         {
31894           \quark_if_recursion_tail_stop:N #1
31895           \tl_if_single_token:nT {#2}
31896           {
31897             \cs_if_exist:cF
31898               { c__text_uppercase_ \token_to_str:N #1 _tl }
31899               {
31900                 \tl_const:cn
31901                   { c__text_uppercase_ \token_to_str:N #1 _tl }
31902                   { #2 }
31903               }
31904             \cs_if_exist:cF
31905               { c__text_lowercase_ \token_to_str:N #2 _tl }
31906               {
31907                 \tl_const:cn
31908                   { c__text_lowercase_ \token_to_str:N #2 _tl }
31909                   { #1 }
31910               }
31911             }
31912             \__text_change_case_setup:Nn
31913           }
31914           \exp_after:wN \__text_change_case_setup:Nn \@uclclist
31915           \q_recursion_tail ?
31916           \q_recursion_stop
31917           \group_end:
31918         }
31919     }

```

31920      }  
31921 </package>

# Chapter 83

## I3text-map implementation

```
31922  <*package>
31923  <@@=text>
```

### 83.1 Mapping to text

```
\text_map_function:nN
\__text_map_function:nN
  \__text_map_loop:Nnw
  \__text_map_group:Nnn
  \__text_map_space:Nnw
  \__text_map_N_type:NnN
\__text_map_codepoint:Nnn
  \__text_map_CR:Nnw
  \__text_map_CR:Nnn
  \__text_map_class:Nnnn
  \__text_map_class:nNnnn
\__text_map_class_loop:Nnnnnw
  \__text_map_class_end:nw
  \__text_map_Control:Nnn
  \__text_map_Extend:Nnn
\__text_map_SpacingMark:Nnn
  \__text_map_Prepend:Nnn
\__text_map_Prepend_aux:Nnn
  \__text_map_Prepend:nNnn
  \__text_map_Prepend_loop:Nnnnw
\__text_map_not_Control:Nnn
  \__text_map_not_Extend:Nnn
  \__text_map_not_SpacingMark:Nnn
\__text_map_not_Prepend:Nnn
  \__text_map_not_L:Nnn
  \__text_map_not_LV:Nnn
  \__text_map_not_V:Nnn
\__text_map_not_LVT:Nnn
  \__text_map_not_T:Nnn
  \__text_map_L:Nnn
  \__text_map_LV:Nnn
  \__text_map_V:Nnn
\__text_map_LVT:Nnn
  \__text_map_T:Nnn
\__text_map_hangul:Nnnnw
\__text_map_hangul:NnnN
  \__text_map_hangul:Nnnnn
\__text_map_hangul_aux:Nnnnnw
  \__text_map_hangul:nNnnnnw
    \__text_map_hangul_loop:Nnnnnnw
```

The standard lead-off for an action loop.

```
31924  \cs_new:Npn \text_map_function:nN #1#2
31925    { \exp_args:Ne \__text_map_function:nN { \text_expand:n {#1} } #2 }
31926  \cs_new:Npn \__text_map_function:nN #1#2
31927    {
31928      \__text_map_loop:Nnw #2 { } #1
31929        \q_text_recursion_tail \q_text_recursion_stop
31930        \prg_break_point:Nn \text_map_break: { }
31931    }
```

The standard set up for an “action” loop. Groups are handled by recursion, spaces are treated similarly: both count as grapheme boundaries. For N-type tokens, we filter out control sequences (again a boundary), then move on to further analysis.

```
31932  \cs_new:Npn \__text_map_loop:Nnw #1#2#3 \q_text_recursion_stop
31933  {
31934    \tl_if_head_is_N_type:nTF {#3}
31935      { \__text_map_N_type:NnN }
31936      {
31937        \tl_if_head_is_group:nTF {#3}
31938          { \__text_map_group:Nnn }
31939          { \__text_map_space:Nnw }
31940      }
31941      #1 {#2} #3 \q_text_recursion_stop
31942  }
31943  \cs_new:Npn \__text_map_group:Nnn #1#2#3
31944  {
31945    \__text_map_output:Nn #1 {#2}
31946    {
31947      \__text_map_loop:Nnw #1 { } #2
31948        \q_text_recursion_tail \q_text_recursion_stop
31949        \prg_break_point:Nn \text_map_break: { }
31950    }
```

```

31951     \__text_map_loop:Nnw #1 { }
31952   }
31953 \use:x
31954   { \cs_new:Npn \exp_not:N \__text_map_space:Nnw ##1##2 \c_space_tl }
31955   {
31956     \__text_map_output:Nn #1 {#2}
31957     #1 { ~ }
31958     \__text_map_loop:Nnw #1 { }
31959   }
31960 \cs_new:Npn \__text_map_N_type:NnN #1#2#3
31961   {
31962     \__text_if_q_recursion_tail_stop_do:Nn #3
31963   {
31964     \__text_map_output:Nn #1 {#2}
31965     \text_map_break:
31966   }
31967 \token_if_cs:NTF #3
31968   {
31969     \__text_map_output:Nn #1 {#2}
31970     #1 {#3}
31971     \__text_map_loop:Nnw #1 { }
31972   }
31973   {
31974     \__text_codepoint_process:nN
31975     { \__text_map_codepoint:Nnn #1 {#2} } #3
31976   }
31977 }

```

We pull out a few special cases here. Carriage returns case needs a bit of context handling so has an auxiliary. Codepoint U+200D is the zero-width joiner, which has no context to concern us: just don't break.

```

31978 \cs_new:Npn \__text_map_codepoint:Nnn #1#2#3
31979   {
31980     \__text_codepoint_compare:nNnTF {#3} = { "0D }
31981   {
31982     \__text_map_output:Nn #1 {#2}
31983     \__text_map_CR:Nnw #1 {#3}
31984   }
31985   {
31986     \__text_codepoint_compare:nNnTF {#3} = { "200D }
31987     { \__text_map_loop:Nnw #1 {#2#3} }
31988     { \__text_map_class:Nnnn #1 {#2} {#3} { Control } }
31989   }
31990 }

```

A carriage return is a boundary unless it is immediately followed by a line feed, in which case that pair is a boundary.

```

31991 \cs_new:Npn \__text_map_CR:Nnw #1#2#3 \q__text_recursion_stop
31992   {
31993     \tl_if_head_is_N_type:nTF {#3}
31994     { \__text_map_CR:NnN #1 {#2} }
31995   {
31996     #1 {#2}
31997     \__text_map_loop:Nnw #1 { }
31998   }

```

```

31999      #3 \q__text_recursion_stop
32000    }
32001 \cs_new:Npn \__text_map_CR:NnN #1#2#3
32002  {
32003    \__text_if_q_recursion_tail_stop_do:Nn #3
32004    {
32005      #1 {#2}
32006      \text_map_break:
32007    }
32008 \bool_lazy_and:nnTF
32009  { ! \token_if_cs_p:N #3 }
32010  { \int_compare_p:nNn { '#3 } = { "0A" } }
32011  {
32012    \__text_map_output:Nn #1 {#2#3}
32013    \__text_map_loop:Nnw #1 { }
32014  }
32015  { \__text_map_loop:Nnw #1 { } #3 }
32016 }

```

There are various classes of character, and we deal with them all in the same general way. We need to example the relevant list of codepoints: if we get a hit, then we do whatever the relevant action is. Otherwise we loop, but only if the current codepoint could still match: the loop stops early otherwise and we move forward.

```

32017 \cs_new:Npn \__text_map_class:Nnnn #1#2#3#4
32018  {
32019    \exp_args:Nv \__text_map_class:nNnnn { c__text_grapheme_ #4 _clist }
32020    #1 {#2} {#3} {#4}
32021  }
32022 \cs_new:Npn \__text_map_class:nNnnn #1#2#3#4#5
32023  {
32024    \__text_map_class_loop:Nnnnw #2 {#3} {#4} {#5}
32025    #1 , \q__text_recursion_tail ... , \q__text_recursion_stop
32026  }
32027 \cs_new:Npn \__text_map_class_loop:Nnnnw #1#2#3#4 #5 .. #6 ,
32028  {
32029    \__text_if_q_recursion_tail_stop_do:nn {#5}
32030    { \use:c { __text_map_not_ #4 :Nnn } #1 {#2} {#3} }
32031    \__text_codepoint_compare:nNnTF {#3} < { "#5" }
32032  {
32033    \__text_map_class_end:nw
32034    { \use:c { __text_map_not_ #4 :Nnn } #1 {#2} {#3} }
32035  }
32036  {
32037    \__text_codepoint_compare:nNnTF {#3} > { "#6" }
32038    { \__text_map_class_loop:Nnnnw #1 {#2} {#3} {#4} }
32039  {
32040    \__text_map_class_end:nw
32041    { \use:c { __text_map_ #4 :Nnn } #1 {#2} {#3} }
32042  }
32043  }
32044  }
32045 \cs_new:Npn \__text_map_class_end:nw #1#2 \q__text_recursion_stop {#1}

```

Break before *and* after.

```
32046 \cs_new:Npn \__text_map_Control:Nnn #1#2#3
```

```

32047    {
32048        \__text_map_output:Nn #1 {#2}
32049        \__text_map_output:Nn #1 {#3}
32050        \__text_map_loop:Nnw #1 { }
32051    }

```

Keep collecting.

```

32052 \cs_new:Npn \__text_map_Extend:Nnn #1#2#3
32053     { \__text_map_loop:Nnw #1 {#2#3} }
32054 \cs_new_eq:NN \__text_map_SpacingMark:Nnn \__text_map_Extend:Nnn

```

Outputting anything earlier, then combine with what follows. The only exclusions are control characters.

```

32055 \cs_new:Npn \__text_map_Prepend:Nnn #1#2#3
32056     {
32057         \__text_map_output:Nn #1 {#2}
32058         \__text_map_lookahead:NnNw #1 {#3} \__text_map_Prepend_aux:Nnn
32059     }
32060 \cs_new:Npn \__text_map_Prepend_aux:Nnn #1#2#3
32061     {
32062         \bool_lazy_or:nnTF
32063             { \__text_codepoint_compare_p:nNn {#3} = { "0A" } }
32064             { \__text_codepoint_compare_p:nNn {#3} = { "0D" } }
32065             {
32066                 #1 {#2}
32067                 \__text_map_loop:Nnw #1 {#3}
32068             }
32069             {
32070                 \exp_args:NV \__text_map_Prepend:nNnn
32071                     \c_text_grapheme_Control_clist
32072                     #1 {#2} {#3}
32073             }
32074     }
32075 \cs_new:Npn \__text_map_Prepend:nNnn #1#2#3#4
32076     {
32077         \__text_map_Prepend_loop:Nnnw #2 {#3} {#4}
32078             #1 , \q_text_recursion_tail ... , \q_text_recursion_stop
32079     }
32080 \cs_new:Npn \__text_map_Prepend_loop:Nnnw #1#2#3 #4 ... #5 ,
32081     {
32082         \__text_if_q_recursion_tail_stop_do:nn {#4}
32083             { \__text_map_loop:Nnw #1 {#2#3} }
32084         \__text_codepoint_compare:nNnTF {#3} < { "#4" }
32085             {
32086                 \__text_map_class_end:nw
32087                     { \__text_map_loop:Nnw #1 {#2#3} }
32088             }
32089             {
32090                 \__text_codepoint_compare:nNnTF {#3} > { "#5" }
32091                     { \__text_map_Prepend_loop:Nnnw #1 {#2} {#3} }
32092                     {
32093                         \__text_map_class_end:nw
32094                             { \__text_map_loop:Nnw #1 {#2} #3 }
32095                     }
32096             }
32097     }

```

```
32097 }
```

Dealing with end-of-class is done such that we can be flexible.

```
32098 \cs_new:Npn \__text_map_not_Control:Nnn #1#2#3
32099   { \__text_map_class:Nnnn #1 {#2} {#3} { Extend } }
32100 \cs_new:Npn \__text_map_not_Extend:Nnn #1#2#3
32101   { \__text_map_class:Nnnn #1 {#2} {#3} { SpacingMark } }
32102 \cs_new:Npn \__text_map_not_SpacingMark:Nnn #1#2#3
32103   { \__text_map_class:Nnnn #1 {#2} {#3} { Prepend } }
32104 \cs_new:Npn \__text_map_not_PrepPEND:Nnn #1#2#3
32105   { \__text_map_class:Nnnn #1 {#2} {#3} { L } }
32106 \cs_new:Npn \__text_map_not_L:Nnn #1#2#3
32107   { \__text_map_class:Nnnn #1 {#2} {#3} { LV } }
32108 \cs_new:Npn \__text_map_not_LV:Nnn #1#2#3
32109   { \__text_map_class:Nnnn #1 {#2} {#3} { V } }
32110 \cs_new:Npn \__text_map_not_V:Nnn #1#2#3
32111   { \__text_map_class:Nnnn #1 {#2} {#3} { LVT } }
32112 \cs_new:Npn \__text_map_not_LVT:Nnn #1#2#3
32113   { \__text_map_class:Nnnn #1 {#2} {#3} { T } }
32114 \cs_new:Npn \__text_map_not_T:Nnn #1#2#3
32115   { \__text_map_class:Nnnn #1 {#2} {#3} { Regional_Indicator } }
32116 \cs_new:Npn \__text_map_not_Regional_Indicator:Nnn #1#2#3
32117   {
32118     \__text_map_output:Nn #1 {#2}
32119     \__text_map_loop:Nnw #1 {#3}
32120 }
```

Hangul needs additional treatment. First we have to deal with the start-of-Hangul position: output what we had up to now, then move the the specialist handler. The idea here is to pick off the different codepoint types one at a time, tracking what else can be considered at each stage until we hit the end of the viable types. Other than that, we just keep building up the Hangul codepoints using a dedicated version of the loop from above.

```
32121 \cs_new:Npn \__text_map_L:Nnn #1#2#3
32122   {
32123     \__text_map_output:Nn #1 {#2}
32124     \__text_map_hangul:Nnnw
32125       #1 {#3} { L ; V ; LV ; LVT }
32126   }
32127 \cs_new:Npn \__text_map_LV:Nnn #1#2#3
32128   {
32129     \__text_map_output:Nn #1 {#2}
32130     \__text_map_hangul:Nnnw
32131       #1 {#3} { V ; T }
32132   }
32133 \cs_new_eq:NN \__text_map_V:Nnn \__text_map_LV:Nnn
32134 \cs_new:Npn \__text_map_LVT:Nnn #1#2#3
32135   {
32136     \__text_map_output:Nn #1 {#2}
32137     \__text_map_hangul:Nnnw
32138       #1 {#3} { T }
32139   }
32140 \cs_new_eq:NN \__text_map_T:Nnn \__text_map_LVT:Nnn
32141 \cs_new:Npn \__text_map_hangul:Nnnw #1#2#3#4 \q__text_recursion_stop
```

```

32142    {
32143        \tl_if_head_is_N_type:nTF {#4}
32144            { \__text_map_hangul:NnnN #1 {#2} {#3} }
32145            {
32146                #1 {#2}
32147                    \__text_map_loop:Nnw #1 { }
32148            }
32149            #4 \q__text_recursion_stop
32150        }
32151 \cs_new:Npn \__text_map_hangul:NnnN #1#2#3#4
32152    {
32153        \__text_if_q_recursion_tail_stop_do:Nn #4
32154        {
32155            #1 {#2}
32156            \text_map_break:
32157        }
32158 \token_if_cs:NTF #4
32159    {
32160        #1 {#2}
32161            \__text_map_loop:Nnw #1 { }
32162    }
32163    {
32164        \__text_codepoint_process:nN
32165            { \__text_map_hangul:Nnnn #1 {#2} {#3} } #4
32166    }
32167 }
32168 \cs_new:Npn \__text_map_hangul:Nnnn #1#2#3#4
32169    {
32170        \__text_map_hangul_aux:Nnnw #1 {#2} {#4}
32171            #3 ; \q_recursion_tail ; \q_recursion_stop
32172    }
32173 \cs_new:Npn \__text_map_hangul_aux:Nnnw #1#2#3#4 ;
32174    {
32175        \quark_if_recursion_tail_stop_do:nn {#4}
32176            { \__text_map_loop:Nnw #1 {#2} #3 }
32177        \exp_args:Nv \__text_map_hangul:nNnnnw { c__text_grapheme_ #4 _clist }
32178            #1 {#2} {#3} {#4}
32179    }
32180 \cs_new:Npn \__text_map_hangul:nNnnnw #1#2#3#4#5#6 \q_recursion_stop
32181    {
32182        \__text_map_hangul_loop:Nnnnnw #2 {#3} {#4} {#5} {#6}
32183            #1 , \q_text_recursion_tail .. , \q_text_recursion_stop
32184    }
32185 \cs_new:Npn \__text_map_hangul_loop:Nnnnnw #1#2#3#4#5 #6 .. #7 ,
32186    {
32187        \__text_if_q_recursion_tail_stop_do:nn {#6}
32188            { \__text_map_hangul_next:Nnnn #1 {#2} {#3} {#5} }
32189        \__text_codepoint_compare:nNnTF {#3} < { "#6 }
32190        {
32191            \__text_map_hangul_end:nw
32192                { \__text_map_hangul_next:Nnnn #1 {#2} {#3} {#5} }
32193        }
32194        {
32195            \__text_codepoint_compare:nNnTF {#3} > { "#7 }

```

```

32196     { \__text_map_hangul_loop:Nnnnnw #1 {#2} {#3} {#4} {#5} }
32197     {
32198         \__text_map_hangul_end:nw
32199             { \use:c { __text_map_hangul_ #4 :Nnn } #1 {#2} {#3} }
32200         }
32201     }
32202 }
32203 \cs_new:Npn \__text_map_hangul_next:Nnnn #1#2#3#4
32204     { \__text_map_hangul_aux:Nnnw #1 {#2} {#3} #4 \q_recursion_stop }
32205 \cs_new:Npn \__text_map_hangul_end:nw #1#2 \q_recursion_stop {#1}
32206 \cs_new:Npn \__text_map_hangul_L:Nnn #1#2#3
32207     {
32208         \__text_map_hangul:Nnnw
32209             #1 {#2#3} { L V { LV } { LVT } }
32210     }
32211 \cs_new:Npn \__text_map_hangul_LV:Nnn #1#2#3
32212     {
32213         \__text_map_hangul:Nnnw
32214             #1 {#2#3} { VT }
32215     }
32216 \cs_new_eq:NN \__text_map_hangul_V:Nnn \__text_map_hangul_LV:Nnn
32217 \cs_new:Npn \__text_map_hangul_LVT:Nnn #1#2#3
32218     {
32219         \__text_map_hangul:Nnnw
32220             #1 {#2#3} { T }
32221     }
32222 \cs_new_eq:NN \__text_map_hangul_T:Nnn \__text_map_hangul_LVT:Nnn

```

The Regional Indicator rule means looking ahead and dealing with the case where there are two in a row. So we use a look ahead to pick them off. As there is only one range the values are hard-coded.

```

32223 \cs_new:Npn \__text_map_Regional_Indicator:Nnn #1#2#3
32224     {
32225         \__text_map_output:Nn #1 {#2}
32226         \__text_map_lookahead:NnNw #1 {#3} \__text_map_Regional_Indicator_aux:Nnn
32227     }
32228 \cs_new:Npn \__text_map_Regional_Indicator_aux:Nnn #1#2#3
32229     {
32230         \bool_lazy_or:nnTF
32231             { \__text_codepoint_compare_p:nNn {#3} < { "1F1E6" } }
32232             { \__text_codepoint_compare_p:nNn {#3} > { "1F1FF" } }
32233         {
32234             \__text_map_loop:Nnw #1 {#2} #3
32235         }
32236         { \__text_map_loop:Nnw #1 {#2#3} }
32237     }

```

A generic loop-ahead setup.

```

32238 \cs_new:Npn \__text_map_lookahead:NnNw #1#2#3#4 \q_recursion_stop
32239     {
32240         \tl_if_head_is_N_type:nTF {#4}
32241             { \__text_map_lookahead:NnNN #1 {#2} #3 }
32242             { \__text_map_loop:Nnw #1 {#2} }
32243             #4 \q_recursion_stop
32244     }

```

```

32245 \cs_new:Npn \__text_map_loookahead:NnNN #1#2#3#4
32246 {
32247     \__text_if_q_recursion_tail_stop_do:Nn #4 { #1 {#2} }
32248     \token_if_cs:NTF #4
32249     {
32250         #1 {#2}
32251         \__text_map_loop:Nnw #1 { }
32252     }
32253     { \__text_codepoint_process:nN { #3 #1 {#2} } }
32254     #4
32255 }
```

For the end of the process.

```

32256 \cs_new:Npn \__text_map_output:Nn #1#2
32257     { \tl_if_blank:nF {#2} { #1 {#2} } }
32258 \cs_new:Npn \text_map_break:
32259     { \prg_map_break:Nn \text_map_break: { } }
32260 \cs_new:Npn \text_map_break:n
32261     { \prg_map_break:Nn \text_map_break: }
```

(End definition for `\text_map_function:nN` and others. These functions are documented on page 271.)

**\text\_map\_inline:nn** The standard non-expandable inline version.

```

32262 \cs_new_protected:Npn \text_map_inline:nn #1#2
32263 {
32264     \int_gincr:N \g_kernel_prg_map_int
32265     \cs_gset_protected:cpn
32266     { __text_map_ \int_use:N \g_kernel_prg_map_int :w } ##1 {#2}
32267     \exp_args:Nnc \text_map_function:nN {#1}
32268     { __text_map_ \int_use:N \g_kernel_prg_map_int :w }
32269     \prg_break_point:Nn \text_map_break:
32270     { \int_gdecr:N \g_kernel_prg_map_int }
```

(End definition for `\text_map_inline:nn`. This function is documented on page 271.)

32272 ⟨/package⟩

# Chapter 84

## I3text-purify implementation

```
32273  <*package>
32274  <@@=text>
```

### 84.1 Purifying text

`\_text_if_recursion_tail_stop:N` Functions to query recursion quarks.

```
32275  \_kernel_quark_new_test:N \_text_if_recursion_tail_stop:N
```

(*End definition for \\_text\_if\_recursion\_tail\_stop:N.*)

`\text_purify:n` As in the other parts of the module, we start off with a standard “action” loop, with expansion applied up-front.

```
32276  \cs_new:Npn \text_purify:n #1
32277  {
32278      \_kernel_exp_not:w \exp_after:wN
32279      {
32280          \exp:w
32281          \exp_args:Ne \text_purify:n
32282          { \text_expand:n {#1} }
32283      }
32284  }
32285  \cs_new:Npn \_text_purify:n #1
32286  {
32287      \group_align_safe_begin:
32288      \_text_purify_loop:w #1
32289      \q_text_recursion_tail \q_text_recursion_stop
32290      \_text_purify_result:n { }
32291  }
```

As for expansion, collect up the tokens for future use.

```
32292  \cs_new:Npn \_text_purify_store:n #1
32293  {
32294      \_text_purify_store:nw {#1}
32295  \cs_new:Npn \_text_purify_store:nw #1#2 \_text_purify_result:n #3
32296  {
32297      \_text_purify_result:n { #3 #1 }
32298  \cs_new:Npn \_text_purify_end:w #1 \_text_purify_result:n #2
32299  {
32300      \group_align_safe_end:
32301      \exp_end:
```

```

32300      #2
32301      }

```

The main loop is a standard “tl action”. Unlike the expansion or case changing, here any groups have to be run inline. Most of the business end is as before in the N-type token processing.

```

32302 \cs_new:Npn \__text_purify_loop:w #1 \q__text_recursion_stop
32303 {
32304     \tl_if_head_is_N_type:nTF {#1}
32305         { \__text_purify_N_type:N }
32306         {
32307             \tl_if_head_is_group:nTF {#1}
32308                 { \__text_purify_group:n }
32309                 { \__text_purify_space:w }
32310         }
32311     #1 \q__text_recursion_stop
32312 }
32313 \cs_new:Npn \__text_purify_group:n #1 { \__text_purify_loop:w #1 }
32314 \exp_last_unbraced:NNo \cs_new:Npn \__text_purify_space:w \c_space_tl
32315 {
32316     \__text_purify_store:n { ~ }
32317     \__text_purify_loop:w
32318 }

```

The first part of handling math mode is exactly the same as in the other functions: look for a start-of-math mode token and if found start a new loop tracking the closing token.

```

32319 \cs_new:Npn \__text_purify_N_type:N #1
32320 {
32321     \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
32322     \__text_purify_N_type_aux:N #1
32323 }
32324 \cs_new:Npn \__text_purify_N_type_aux:N #1
32325 {
32326     \exp_after:wN \__text_purify_math_search:NNN
32327         \exp_after:wN #1 \l_text_math_delims_tl
32328         \q__text_recursion_tail ?
32329         \q__text_recursion_stop
32330 }
32331 \cs_new:Npn \__text_purify_math_search:NNN #1#2#3
32332 {
32333     \__text_if_q_recursion_tail_stop_do:Nn #2
32334         { \__text_purify_math_cmd:N #1 }
32335     \token_if_eq_meaning:NNTF #1 #2
32336         {
32337             \__text_use_i_delimit_by_q_recursion_stop:nw
32338                 { \__text_purify_math_start>NNw #2 #3 }
32339         }
32340         { \__text_purify_math_search:NNN #1 }
32341     }
32342 \cs_new:Npn \__text_purify_math_start>NNw #1#2#3 \q__text_recursion_stop
32343 {
32344     \__text_purify_math_loop>NNw #1#2#3 \q__text_recursion_stop
32345         \__text_purify_math_result:n { }
32346     }
32347 \cs_new:Npn \__text_purify_math_store:n #1

```

```

32348 { \__text_purify_math_store:nw {#1} }
32349 \cs_new:Npn \__text_purify_math_store:nw #1#2 \__text_purify_math_result:n #3
32350 { #2 \__text_purify_math_result:n { #3 #1 } }
32351 \cs_new:Npn \__text_purify_math_end:w #1 \__text_purify_math_result:n #2
32352 {
32353     \__text_purify_store:n { $ #2 $ }
32354     \__text_purify_loop:w #1
32355 }
32356 \cs_new:Npn \__text_purify_math_stop:Nw #1 \__text_purify_math_result:n #2
32357 {
32358     \__text_purify_store:n {#1#2}
32359     \__text_purify_end:w
32360 }
32361 \cs_new:Npn \__text_purify_math_loop>NNw #1#2#3 \q__text_recursion_stop
32362 {
32363     \tl_if_head_is_N_type:NTF {#3}
32364     { \__text_purify_math_N_type:NNN }
32365     {
32366         \tl_if_head_is_group:nTF {#3}
32367         { \__text_purify_math_group:NNn }
32368         { \__text_purify_math_space>NNw }
32369     }
32370     #1#2#3 \q__text_recursion_stop
32371 }
32372 \cs_new:Npn \__text_purify_math_N_type:NNN #1#2#3
32373 {
32374     \__text_if_q_recursion_tail_stop_do:Nn #3
32375     { \__text_purify_math_stop:Nw #1 }
32376     \token_if_eq_meaning:NNTF #3 #2
32377     { \__text_purify_math_end:w }
32378     {
32379         \__text_purify_math_store:n {#3}
32380         \__text_purify_math_loop>NNw #1#2
32381     }
32382 }
32383 \cs_new:Npn \__text_purify_math_group:NNn #1#2#3
32384 {
32385     \__text_purify_math_store:n { {#3} }
32386     \__text_purify_math_loop>NNw #1#2
32387 }
32388 \exp_after:wN \cs_new:Npn \exp_after:wN \__text_purify_math_space>NNw
32389 \exp_after:wN # \exp_after:wN 1
32390 \exp_after:wN # \exp_after:wN 2 \c_space_tl
32391 {
32392     \__text_purify_math_store:n { ~ }
32393     \__text_purify_math_loop>NNw #1#2
32394 }

```

Then handle math mode as an argument: same outcomes, different input syntax.

```

32395 \cs_new:Npn \__text_purify_math_cmd:N #1
32396 {
32397     \exp_after:wN \__text_purify_math_cmd:NN \exp_after:wN #1
32398     \l_text_math_arg_tl \q__text_recursion_tail \q__text_recursion_stop
32399 }
32400 \cs_new:Npn \__text_purify_math_cmd:NN #1#2

```

```

32401  {
32402    \__text_if_q_recursion_tail_stop_do:Nn #2
32403    { \__text_purify_replace:N #1 }
32404    \cs_if_eq:NNTF #2 #1
32405    {
32406      \__text_use_i_delimit_by_q_recursion_stop:nw
32407      { \__text_purify_math_cmd:n }
32408    }
32409    { \__text_purify_math_cmd:NN #1 }
32410  }
32411 \cs_new:Npn \__text_purify_math_cmd:n #1
32412  { \__text_purify_math_end:w \__text_purify_math_result:n {#1} }

```

For N-type tokens, we first look for a string-context replacement before anything else: this can therefore cover anything. Assuming we don't find one, check to see if we can expand control sequences: if not, they have to be dropped. We also allow for L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  \protect: there's an assumption that we don't have \protect { \oops } or similar, but that's also in the expansion code and seems like a reasonable balance.

```

32413 \cs_new:Npn \__text_purify_replace:N #1
32414  {
32415    \bool_lazy_and:nnTF
32416    { \cs_if_exist_p:c { \__text_purify_ \token_to_str:N #1 _tl } }
32417    {
32418      \bool_lazy_or_p:nn
32419      { \token_if_cs_p:N #1 }
32420      { \token_if_active_p:N #1 }
32421    }
32422    {
32423      \exp_args:Nv \__text_purify_replace:n
32424      { \__text_purify_ \token_to_str:N #1 _tl }
32425    }
32426    {
32427      \token_if_cs:NTF #1
32428      { \__text_purify_expand:N #1 }
32429      {
32430        \exp_args:Ne \__text_purify_store:n
32431        { \__text_token_to_explicit:N #1 }
32432        \__text_purify_loop:w
32433      }
32434    }
32435  }
32436 \cs_new:Npn \__text_purify_replace:n #1 { \__text_purify_loop:w #1 }
32437 \cs_new:Npn \__text_purify_expand:N #1
32438  {
32439    \str_if_eq:nnTF {#1} { \protect }
32440    { \__text_purify_protect:N }
32441    { \__text_purify_encoding:N #1 }
32442  }
32443 \cs_new:Npn \__text_purify_protect:N #1
32444  {
32445    \__text_if_q_recursion_tail_stop_do:Nn #1 { \__text_purify_end:w }
32446    \__text_purify_loop:w
32447  }

```

Handle encoding commands, as detailed for expansion.

```

32448 \cs_new:Npn \__text_purify_encoding:N #1
32449  {
32450      \bool_lazy_or:nnTF
32451          { \cs_if_eq_p:NN #1 \@current@cmd }
32452          { \cs_if_eq_p:NN #1 \@changed@cmd }
32453          { \__text_purify_encoding_escape:NN }
32454  {
32455      \__text_if_expandable:NTF #1
32456          { \exp_after:wN \__text_purify_loop:w #1 }
32457          { \__text_purify_loop:w }
32458  }
32459 }
32460 \cs_new:Npn \__text_purify_encoding_escape:NN #1#2
32461  {
32462      \__text_purify_store:n {#1}
32463      \__text_purify_loop:w
32464 }

```

(End definition for `\text_purify:n` and others. This function is documented on page 270.)

```

\text_declare_purify_equivalent:Nn
\text_declare_purify_equivalent:Nx
32465 \cs_new_protected:Npn \text_declare_purify_equivalent:Nn #1#2
32466  {
32467      \tl_clear_new:c { l__text_purify_ \token_to_str:N #1 _tl }
32468      \tl_set:cn { l__text_purify_ \token_to_str:N #1 _tl } {#2}
32469  }
32470 \cs_generate_variant:Nn \text_declare_purify_equivalent:Nn { Nx }

```

(End definition for `\text_declare_purify_equivalent:Nn`. This function is documented on page 270.)

Now pre-define a range of standard commands that need dedicated definitions in purified text. First handle font-related stuff: all of this needs to be disabled.

```

32471 \tl_map_inline:nn
32472  {
32473      \fontencoding
32474      \fontfamily
32475      \fontseries
32476      \fontshape
32477  }
32478  { \text_declare_purify_equivalent:Nn #1 { \use_none:n } }
32479 \text_declare_purify_equivalent:Nn \fontsize { \use_none:nn }
32480 \text_declare_purify_equivalent:Nn \selectfont { }
32481 \text_declare_purify_equivalent:Nn \usefont { \use_none:nnnn }
32482 \tl_map_inline:nn
32483  {
32484      \emph
32485      \text
32486      \textnormal
32487      \textrm
32488      \textsf
32489      \texttt
32490      \textbf
32491      \textmd
32492      \textit
32493      \textsl

```

```

32494     \textup
32495     \textsc
32496     \textulc
32497     }
32498     { \text_declare_purify_equivalent:Nn #1 { \use:n } }
32499 \tl_map_inline:nn
32500     {
32501         \normalfont
32502         \rmfamily
32503         \sffamily
32504         \ttfamily
32505         \bfseries
32506         \mdseries
32507         \itshape
32508         \scshape
32509         \slshape
32510         \upshape
32511         \em
32512         \Huge
32513         \LARGE
32514         \Large
32515         \footnotesize
32516         \huge
32517         \large
32518         \normalsize
32519         \scriptsize
32520         \small
32521         \tiny
32522     }
32523     { \text_declare_purify_equivalent:Nn #1 { } }
32524 \exp_args:Nc \text_declare_purify_equivalent:Nn
32525     { @protected@testopt } { \use_none:nnn }

```

Environments have to be handled by pure expansion.

```
\_text_end_env:n
32526 \text_declare_purify_equivalent:Nn \begin { \use:c }
32527 \text_declare_purify_equivalent:Nn \end { \_text_end_env:n }
32528 \cs_new:Npn \_text_end_env:n #1 { \cs:w end #1 \cs_end: }
```

(End definition for \\_text\_end\_env:n.)

Some common symbols and similar ideas.

```
32529 \text_declare_purify_equivalent:Nn \\ { }
32530 \tl_map_inline:nn
32531     { \{ \} \# \$ \% \_ }
32532     { \text_declare_purify_equivalent:Nx #1 { \cs_to_str:N #1 } }
```

Cross-referencing.

```
32533 \text_declare_purify_equivalent:Nn \label { \use_none:n }
```

Spaces.

```
32534 \group_begin:
32535 \char_set_catcode_active:N \~
32536 \use:n
32537 {
```

```

32538     \group_end:
32539     \text_declare_purify_equivalent:Nx ~ { \c_space_tl }
32540   }
32541 \text_declare_purify_equivalent:Nn \nobreakspace { ~ }
32542 \text_declare_purify_equivalent:Nn \_ { ~ }
32543 \text_declare_purify_equivalent:Nn \, { ~ }

```

## 84.2 Accent and letter-like data for purifying text

In contrast to case changing, both 8-bit and Unicode engines need information for text purification to handle accents and letter-like functions: these all need to be removed. However, the results are of course engine-dependent.

For the letter-like commands, life is relatively easy: they are all simply added as standard exceptions. The only oddity is \SS, which gets converted to two letters. (At some stage an alternative version can presumably be added to babel or similar.)

```

32544 \bool_lazy_or:nnTF
32545   { \sys_if_engine_luatex_p: }
32546   { \sys_if_engine_xetex_p: }
32547   {
32548     \cs_set_protected:Npn \__text_loop:Nn #1#2
32549     {
32550       \quark_if_recursion_tail_stop:N #1
32551       \text_declare_purify_equivalent:Nx #1
32552         {
32553           \char_generate:nn { "#2" }
32554             { \char_value_catcode:n { "#2" } }
32555         }
32556       \__text_loop:Nn
32557     }
32558   }
32559   {
32560     \cs_set_protected:Npn \__text_loop:Nn #1#2
32561     {
32562       \quark_if_recursion_tail_stop:N #1
32563       \text_declare_purify_equivalent:Nx #1
32564         {
32565           \exp_args:Ne \__text_tmp:n
32566             { \__kernel_codepoint_to_bytes:n { "#2" } }
32567         }
32568       \__text_loop:Nn
32569     }
32570     \cs_set:Npn \__text_tmp:n #1 { \__text_tmp:nnnn #1 }
32571     \cs_set:Npn \__text_tmp:nnnn #1#2#3#4
32572     {
32573       \exp_after:wN \exp_after:wN \exp_after:wN
32574         \exp_not:N \char_generate:nn {#1} { 13 }
32575       \exp_after:wN \exp_after:wN \exp_after:wN
32576         \exp_not:N \char_generate:nn {#2} { 13 }
32577     }
32578   }
32579 \__text_loop:Nn
32580   \AA { 00C5 }

```

```

32581   \AE { 00C6 }
32582   \DH { 00D0 }
32583   \DJ { 0110 }
32584   \IJ { 0132 }
32585   \L { 0141 }
32586   \NG { 014A }
32587   \O { 00D8 }
32588   \OE { 0152 }
32589   \TH { 00DE }
32590   \aa { 00E5 }
32591   \ae { 00E6 }
32592   \dh { 00F0 }
32593   \dj { 0111 }
32594   \i { 0131 }
32595   \j { 0237 }
32596   \ij { 0132 }
32597   \l { 0142 }
32598   \ng { 014B }
32599   \o { 00F8 }
32600   \oe { 0153 }
32601   \ss { 00DF }
32602   \th { 00FE }
32603   \q_recursion_tail ?
32604   \q_recursion_stop
32605 \text_declare_purify_equivalent:Nn \ss { SS }

```

\\_\\_text\\_purify\\_accent:NN Accent LICR handling is a little more complex. Accents may exist as pre-composed codepoints or as independent glyphs. The former are all saved as single token lists, whilst for the latter the combining accent needs to be re-ordered compared to the character it applies to.

```

32606 \cs_new:Npn \_\_text_purify_accent:NN #1#2
32607 {
32608   \cs_if_exist:cTF
32609     { c_\_text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
32610     {
32611       \exp_not:v
32612         { c_\_text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
32613     }
32614     {
32615       \exp_not:n {#2}
32616       \exp_not:v { c_\_text_purify_ \token_to_str:N #1 _tl }
32617     }
32618   }
32619 \tl_map_inline:Nn \l_text_accents_tl
32620   { \text_declare_purify_equivalent:Nn #1 { \_\_text_purify_accent:NN #1 } }

```

First set up the combining accents.

```

32621 \group_begin:
32622   \cs_set_protected:Npn \_\_text_loop:Nn #1#2
32623   {
32624     \quark_if_recursion_tail_stop:N #1
32625     \tl_const:cx { c_\_text_purify_ \token_to_str:N #1 _tl }
32626       { \_\_text_tmp:n {#2} }
32627     \_\_text_loop:Nn

```

```

32628      }
32629  \bool_lazy_or:nnTF
32630    { \sys_if_engine_luatex_p: }
32631    { \sys_if_engine_xetex_p: }
32632    {
32633      \cs_set:Npn \__text_tmp:n #1
32634      {
32635        \char_generate:nn { "#1" }
32636        { \char_value_catcode:n { "#1" } }
32637      }
32638    }
32639    {
32640      \cs_set:Npn \__text_tmp:n #1
32641      {
32642        \exp_args:Ne \__text_tmp_aux:n
32643        { \__kernel_codepoint_to_bytes:n { "#1" } }
32644      }
32645      \cs_set:Npn \__text_tmp_aux:n #1 { \__text_tmp:nnnn #1 }
32646      \cs_set:Npn \__text_tmp:nnnn #1#2#3#4
32647      {
32648        \exp_after:wN \exp_after:wN \exp_after:wN
32649        \exp_not:N \char_generate:nn {#1} { 13 }
32650        \exp_after:wN \exp_after:wN \exp_after:wN
32651        \exp_not:N \char_generate:nn {#2} { 13 }
32652      }
32653    }
32654  \__text_loop:Nn
32655    \` { 0300 }
32656    \` { 0301 }
32657    \^ { 0302 }
32658    \~ { 0303 }
32659    \= { 0304 }
32660    \u { 0306 }
32661    \. { 0307 }
32662    \v { 0308 }
32663    \r { 030A }
32664    \H { 030B }
32665    \v { 030C }
32666    \d { 0323 }
32667    \c { 0327 }
32668    \k { 0328 }
32669    \b { 0331 }
32670    \t { 0361 }
32671    \q_recursion_tail { }
32672    \q_recursion_stop

```

Now we handle the pre-composed accents: the list here is taken from `puenc.def`. All of the precomposed cases take a single letter as their second argument. We do not try to cover the case where an accent is added to a “real” dotless-i or -j, or a æ/Æ. Rather, we assume that if the UTF-8 character is used, it will have the real accent character too.

```

32673  \cs_set_protected:Npn \__text_loop>NNn #1#2#3
32674  {
32675    \quark_if_recursion_tail_stop:N #1
32676    \tl_const:cx

```

```

32677      { c__text_purify_ \token_to_str:N #1 _ \token_to_str:N #2 _tl }
32678      { \__text_tmp:n {#3} }
32679      \__text_loop>NNn
32680  }
32681  \__text_loop>NNn
32682    \` A  { 00C0 }
32683    \` A  { 00C1 }
32684    \^ A  { 00C2 }
32685    \~ A  { 00C3 }
32686    \\" A  { 00C4 }
32687    \r A  { 00C5 }
32688    \c C  { 00C7 }
32689    \` E  { 00C8 }
32690    \` E  { 00C9 }
32691    \^ E  { 00CA }
32692    \\" E  { 00CB }
32693    \` I  { 00CC }
32694    \` I  { 00CD }
32695    \^ I  { 00CE }
32696    \\" I  { 00CF }
32697    \~ N  { 00D1 }
32698    \` O  { 00D2 }
32699    \` O  { 00D3 }
32700    \^ O  { 00D4 }
32701    \~ O  { 00D5 }
32702    \\" O  { 00D6 }
32703    \` U  { 00D9 }
32704    \` U  { 00DA }
32705    \^ U  { 00DB }
32706    \\" U  { 00DC }
32707    \` Y  { 00DD }
32708    \` a  { 00E0 }
32709    \` a  { 00E1 }
32710    \^ a  { 00E2 }
32711    \~ a  { 00E3 }
32712    \\" a  { 00E4 }
32713    \r a  { 00E5 }
32714    \c c  { 00E7 }
32715    \` e  { 00E8 }
32716    \` e  { 00E9 }
32717    \^ e  { 00EA }
32718    \\" e  { 00EB }
32719    \` i  { 00EC }
32720    \` \i  { 00EC }
32721    \` i  { 00ED }
32722    \` \i  { 00ED }
32723    \^ i  { 00EE }
32724    \^ \i  { 00EE }
32725    \\" i  { 00EF }
32726    \\" \i  { 00EF }
32727    \~ n  { 00F1 }
32728    \` o  { 00F2 }
32729    \` o  { 00F3 }
32730    \^ o  { 00F4 }

```

32731	\~ o	{ 00F5 }
32732	\" o	{ 00F6 }
32733	\` u	{ 00F9 }
32734	\^ u	{ 00FA }
32735	\~ u	{ 00FB }
32736	\" u	{ 00FC }
32737	\` y	{ 00FD }
32738	\" y	{ 00FF }
32739	\= A	{ 0100 }
32740	\= a	{ 0101 }
32741	\u A	{ 0102 }
32742	\u a	{ 0103 }
32743	\k A	{ 0104 }
32744	\k a	{ 0105 }
32745	\` C	{ 0106 }
32746	\` c	{ 0107 }
32747	\~ C	{ 0108 }
32748	\~ c	{ 0109 }
32749	\. C	{ 010A }
32750	\. c	{ 010B }
32751	\v C	{ 010C }
32752	\v c	{ 010D }
32753	\v D	{ 010E }
32754	\v d	{ 010F }
32755	\= E	{ 0112 }
32756	\= e	{ 0113 }
32757	\u E	{ 0114 }
32758	\u e	{ 0115 }
32759	\. E	{ 0116 }
32760	\. e	{ 0117 }
32761	\k E	{ 0118 }
32762	\k e	{ 0119 }
32763	\v E	{ 011A }
32764	\v e	{ 011B }
32765	\~ G	{ 011C }
32766	\~ g	{ 011D }
32767	\u G	{ 011E }
32768	\u g	{ 011F }
32769	\. G	{ 0120 }
32770	\. g	{ 0121 }
32771	\c G	{ 0122 }
32772	\c g	{ 0123 }
32773	\~ H	{ 0124 }
32774	\~ h	{ 0125 }
32775	\~ I	{ 0128 }
32776	\~ i	{ 0129 }
32777	\~ \i	{ 0129 }
32778	\= I	{ 012A }
32779	\= i	{ 012B }
32780	\= \i	{ 012B }
32781	\u I	{ 012C }
32782	\u i	{ 012D }
32783	\u \i	{ 012D }
32784	\k I	{ 012E }

32785	\k i	{ 012F }
32786	\k \i	{ 012F }
32787	\. I	{ 0130 }
32788	\^ J	{ 0134 }
32789	\^ j	{ 0135 }
32790	\^ \j	{ 0135 }
32791	\c K	{ 0136 }
32792	\c k	{ 0137 }
32793	\' L	{ 0139 }
32794	\' l	{ 013A }
32795	\c L	{ 013B }
32796	\c l	{ 013C }
32797	\v L	{ 013D }
32798	\v l	{ 013E }
32799	\. L	{ 013F }
32800	\. l	{ 0140 }
32801	\' N	{ 0143 }
32802	\' n	{ 0144 }
32803	\c N	{ 0145 }
32804	\c n	{ 0146 }
32805	\v N	{ 0147 }
32806	\v n	{ 0148 }
32807	\= O	{ 014C }
32808	\= o	{ 014D }
32809	\u O	{ 014E }
32810	\u o	{ 014F }
32811	\H O	{ 0150 }
32812	\H o	{ 0151 }
32813	\' R	{ 0154 }
32814	\' r	{ 0155 }
32815	\c R	{ 0156 }
32816	\c r	{ 0157 }
32817	\v R	{ 0158 }
32818	\v r	{ 0159 }
32819	\' S	{ 015A }
32820	\' s	{ 015B }
32821	\^ S	{ 015C }
32822	\^ s	{ 015D }
32823	\c S	{ 015E }
32824	\c s	{ 015F }
32825	\v S	{ 0160 }
32826	\v s	{ 0161 }
32827	\c T	{ 0162 }
32828	\c t	{ 0163 }
32829	\v T	{ 0164 }
32830	\v t	{ 0165 }
32831	\~ U	{ 0168 }
32832	\~ u	{ 0169 }
32833	\= U	{ 016A }
32834	\= u	{ 016B }
32835	\u U	{ 016C }
32836	\u u	{ 016D }
32837	\r U	{ 016E }
32838	\r u	{ 016F }

```

32839 \H U { 0170 }
32840 \H u { 0171 }
32841 \k U { 0172 }
32842 \k u { 0173 }
32843 \^ W { 0174 }
32844 \^ w { 0175 }
32845 \^ Y { 0176 }
32846 \^ y { 0177 }
32847 \^" Y { 0178 }
32848 \^' Z { 0179 }
32849 \^' z { 017A }
32850 \. Z { 017B }
32851 \. z { 017C }
32852 \v Z { 017D }
32853 \v z { 017E }
32854 \v A { 01CD }
32855 \v a { 01CE }
32856 \v I { 01CF }
32857 \v \i { 01DO }
32858 \v i { 01DO }
32859 \v O { 01D1 }
32860 \v o { 01D2 }
32861 \v U { 01D3 }
32862 \v u { 01D4 }
32863 \v G { 01E6 }
32864 \v g { 01E7 }
32865 \v K { 01E8 }
32866 \v k { 01E9 }
32867 \k O { 01EA }
32868 \k o { 01EB }
32869 \v \j { 01FO }
32870 \v j { 01FO }
32871 \^' G { 01F4 }
32872 \^' g { 01F5 }
32873 \^' N { 01F8 }
32874 \^' n { 01F9 }
32875 \^' \AE { 01FC }
32876 \^' \ae { 01FD }
32877 \^' \O { 01FE }
32878 \^' \o { 01FF }
32879 \v H { 021E }
32880 \v h { 021F }
32881 \. A { 0226 }
32882 \. a { 0227 }
32883 \c E { 0228 }
32884 \c e { 0229 }
32885 \. O { 022E }
32886 \. o { 022F }
32887 \= Y { 0232 }
32888 \= y { 0233 }
32889 \q_recursion_tail ? { }
32890 \q_recursion_stop
32891 \group_end:

```

(End definition for \\_\\_text\\_purify\\_accent:NN.)

32892 ⟨/package⟩

# Chapter 85

## I3box implementation

```
32893  <*package>
32894  <@@=box>
```

### 85.1 Support code

`\__box_dim_eval:w` Evaluating a dimension expression expandably. The only difference with `\dim_eval:n` is the lack of `\dim_use:N`, to produce an internal dimension rather than expand it into characters.

```
32895  \cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
32896  \cs_new:Npn \__box_dim_eval:n #1
32897    { \__box_dim_eval:w #1 \scan_stop: }
```

(End definition for `\__box_dim_eval:w` and `\__box_dim_eval:n`.)

`\__kernel_kern:n` We need kerns in a few places. At present, we don't have a module for this concept, so it goes in at first use: here. The idea is to avoid repeated use of the bare primitive.

```
32898  \cs_new_protected:Npn \__kernel_kern:n #1
32899    { \tex_kern:D \__box_dim_eval:n {#1} }
```

(End definition for `\__kernel_kern:n`.)

### 85.2 Creating and initialising boxes

The following test files are used for this code: `m3box001.lvt`.

`\box_new:N` Defining a new `<box>` register: remember that box 255 is not generally available.

```
32900  \cs_new_protected:Npn \box_new:N #1
32901    {
32902      \__kernel_chk_if_free_cs:N #1
32903      \cs:w newbox \cs_end: #1
32904    }
32905  \cs_generate_variant:Nn \box_new:N { c }
```

Clear a  $\langle box \rangle$  register.

```
32906 \cs_new_protected:Npn \box_clear:N #1
32907   { \box_set_eq:NN #1 \c_empty_box }
32908 \cs_new_protected:Npn \box_gclear:N #1
32909   { \box_gset_eq:NN #1 \c_empty_box }
32910 \cs_generate_variant:Nn \box_clear:N { c }
32911 \cs_generate_variant:Nn \box_gclear:N { c }
```

Clear or new.

```
32912 \cs_new_protected:Npn \box_clear_new:N #1
32913   { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
32914 \cs_new_protected:Npn \box_gclear_new:N #1
32915   { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
32916 \cs_generate_variant:Nn \box_clear_new:N { c }
32917 \cs_generate_variant:Nn \box_gclear_new:N { c }
```

Assigning the contents of a box to be another box.

```
32918 \cs_new_protected:Npn \box_set_eq:NN #1#2
32919   { \tex_setbox:D #1 \tex_copy:D #2 }
32920 \cs_new_protected:Npn \box_gset_eq:NN #1#2
32921   { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
32922 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
32923 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
```

Assigning the contents of a box to be another box, then drops the original box.

```
32924 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
32925   { \tex_setbox:D #1 \tex_box:D #2 }
32926 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
32927   { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
32928 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
32929 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }

Copies of the cs functions defined in l3basics.
```

```
32930 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
32931   { TF , T , F , p }
32932 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
32933   { TF , T , F , p }
```

### 85.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a  $\langle box \rangle$  register.

```
32934 \cs_new_eq:NN \box_ht:N \tex_ht:D
32935 \cs_new_eq:NN \box_dp:N \tex_dp:D
32936 \cs_new_eq:NN \box_wd:N \tex_wd:D
32937 \cs_generate_variant:Nn \box_ht:N { c }
32938 \cs_generate_variant:Nn \box_dp:N { c }
32939 \cs_generate_variant:Nn \box_wd:N { c }
```

The  $\backslash box\_ht:N$  and  $\backslash box\_dp:N$  primitives do not expand but rather are suitable for use after  $\backslash the$  or inside dimension expressions. Here we obtain the same behaviour by using  $\backslash \_box\_dim\_eval:n$  (basically  $\backslash dimexpr$ ) rather than  $\backslash dim\_eval:n$  (basically  $\backslash the \backslash dimexpr$ ).

```

32940 \cs_new_protected:Npn \box_ht_plus_dp:N #1
32941   { \__box_dim_eval:n { \box_ht:N #1 + \box_dp:N #1 } }
32942 \cs_generate_variant:Nn \box_ht_plus_dp:N { c }

\box_set_ht:Nn
\box_set_ht:cn
\box_gset_ht:Nn
\box_gset_ht:cn
\box_set_dp:Nn
\box_set_dp:cn
\box_gset_dp:Nn
\box_gset_dp:cn
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn
32943 \cs_new_protected:Npn \box_set_dp:Nn #1#2
32944   {
32945     \tex_setbox:D #1 = \tex_copy:D #1
32946     \box_dp:N #1 \__box_dim_eval:n {#2}
32947   }
32948 \cs_generate_variant:Nn \box_set_dp:Nn { c }
32949 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
32950   { \box_dp:N #1 \__box_dim_eval:n {#2} }
32951 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
32952 \cs_new_protected:Npn \box_set_ht:Nn #1#2
32953   {
32954     \tex_setbox:D #1 = \tex_copy:D #1
32955     \box_ht:N #1 \__box_dim_eval:n {#2}
32956   }
32957 \cs_generate_variant:Nn \box_set_ht:Nn { c }
32958 \cs_new_protected:Npn \box_gset_ht:Nn #1#2
32959   { \box_ht:N #1 \__box_dim_eval:n {#2} }
32960 \cs_generate_variant:Nn \box_gset_ht:Nn { c }
32961 \cs_new_protected:Npn \box_set_wd:Nn #1#2
32962   {
32963     \tex_setbox:D #1 = \tex_copy:D #1
32964     \box_wd:N #1 \__box_dim_eval:n {#2}
32965   }
32966 \cs_generate_variant:Nn \box_set_wd:Nn { c }
32967 \cs_new_protected:Npn \box_gset_wd:Nn #1#2
32968   { \box_wd:N #1 \__box_dim_eval:n {#2} }
32969 \cs_generate_variant:Nn \box_gset_wd:Nn { c }

```

## 85.4 Using boxes

Using a  $\langle box \rangle$ . These are just TeX primitives with meaningful names.

```

\box_use_drop:N
\box_use_drop:c
\box_use:N
\box_use:c
32970 \cs_new_eq:NN \box_use_drop:N \tex_box:D
32971 \cs_new_eq:NN \box_use:N \tex_copy:D
32972 \cs_generate_variant:Nn \box_use_drop:N { c }
32973 \cs_generate_variant:Nn \box_use:N { c }

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

\box_move_left:nn
\box_move_right:nn
\box_move_up:nn
\box_move_down:nn
32974 \cs_new_protected:Npn \box_move_left:nn #1#2
32975   { \tex_moveleft:D \__box_dim_eval:n {#1} #2 }
32976 \cs_new_protected:Npn \box_move_right:nn #1#2
32977   { \tex_moveright:D \__box_dim_eval:n {#1} #2 }
32978 \cs_new_protected:Npn \box_move_up:nn #1#2
32979   { \tex_raise:D \__box_dim_eval:n {#1} #2 }
32980 \cs_new_protected:Npn \box_move_down:nn #1#2
32981   { \tex_lower:D \__box_dim_eval:n {#1} #2 }

```

## 85.5 Box conditionals

The primitives for testing if a  $\langle box \rangle$  is empty/void or which type of box it is.

```
32982 \cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
32983 \cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
32984 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
\if_box_empty:N

32985 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
32986   { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
32987 \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
32988   { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
32989 \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
32990   { c } { p , T , F , TF }
32991 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
32992   { c } { p , T , F , TF }

\box_if_horizontal_p:N Testing if a  $\langle box \rangle$  is empty/void.
\box_if_horizontal_p:c
\box_if_horizontal:NTF
\box_if_horizontal:cTF
\box_if_vertical_p:N
\box_if_vertical_p:c
\box_if_vertical:NTF
\box_if_vertical:cTF
```

```
32993 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
32994   { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
32995 \prg_generate_conditional_variant:Nnn \box_if_empty:N
32996   { c } { p , T , F , TF }
```

(End definition for `\box_new:N` and others. These functions are documented on page 273.)

## 85.6 The last box inserted

`\box_set_to_last:N` Set a box to the previous box.

```
32997 \cs_new_protected:Npn \box_set_to_last:N #1
32998   { \tex_setbox:D #1 \tex_lastbox:D }
32999 \cs_new_protected:Npn \box_gset_to_last:N #1
33000   { \tex_global:D \tex_setbox:D #1 \tex_lastbox:D }
33001 \cs_generate_variant:Nn \box_set_to_last:N { c }
33002 \cs_generate_variant:Nn \box_gset_to_last:N { c }
```

(End definition for `\box_set_to_last:N` and `\box_gset_to_last:N`. These functions are documented on page 276.)

## 85.7 Constant boxes

`\c_empty_box` A box we never use.

```
33003 \box_new:N \c_empty_box
```

(End definition for `\c_empty_box`. This variable is documented on page 276.)

## 85.8 Scratch boxes

```
\l_tmpa_box  Scratch boxes.  
\l_tmpb_box  
\g_tmpa_box  
\g_tmpb_box  
33004 \box_new:N \l_tmpa_box  
33005 \box_new:N \l_tmpb_box  
33006 \box_new:N \g_tmpa_box  
33007 \box_new:N \g_tmpb_box
```

(End definition for `\l_tmpa_box` and others. These variables are documented on page 276.)

## 85.9 Viewing box contents

TEX's `\showbox` is not really that helpful in many cases, and it is also inconsistent with other IATEX<sup>3</sup> `show` functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

```
\box_show:N  Essentially a wrapper around the internal function, but evaluating the breadth and depth  
\box_show:c  arguments now outside the group.  
\box_show:Nnn  
\box_show:cnn  
33008 \cs_new_protected:Npn \box_show:N #1  
33009 { \box_show:Nnn #1 \c_max_int \c_max_int }  
33010 \cs_generate_variant:Nn \box_show:N { c }  
33011 \cs_new_protected:Npn \box_show:Nnn #1#2#3  
33012 { \__box_show>NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }  
33013 \cs_generate_variant:Nn \box_show:Nnn { c }
```

(End definition for `\box_show:N` and `\box_show:Nnn`. These functions are documented on page 277.)

```
\box_log:N  Getting TEX to write to the log without interruption the run is done by altering the  
\box_log:c  interaction mode. For that, the  $\varepsilon$ -TEX extensions are needed.  
\box_log:Nnn  
\box_log:cnn  
\__box_log:nNnn  
33014 \cs_new_protected:Npn \box_log:N #1  
33015 { \box_log:Nnn #1 \c_max_int \c_max_int }  
33016 \cs_generate_variant:Nn \box_log:N { c }  
33017 \cs_new_protected:Npn \box_log:Nnn  
33018 { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }  
33019 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4  
33020 {  
33021 \int_set:Nn \tex_interactionmode:D { 0 }  
33022 \__box_show>NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }  
33023 \int_set:Nn \tex_interactionmode:D {#1}  
33024 }  
33025 \cs_generate_variant:Nn \box_log:Nnn { c }
```

(End definition for `\box_log:N`, `\box_log:Nnn`, and `\__box_log:nNnn`. These functions are documented on page 277.)

```
\__box_show>NNnn  
\__box_show>NNff  
The internal auxiliary to actually do the output uses a group to deal with breadth and  
depth values. The \use:n here gives better output appearance. Setting \tracingonline  
and \errorcontextlines is used to control what appears in the terminal.
```

```
33026 \cs_new_protected:Npn \__box_show>NNnn #1#2#3#4  
33027 {  
33028 \box_if_exist:NTF #2  
33029 {  
33030 \group_begin:
```

```

33031     \int_set:Nn \tex_showboxbreadth:D {#3}
33032     \int_set:Nn \tex_showboxdepth:D {#4}
33033     \int_set:Nn \tex_tracingonline:D {#1}
33034     \int_set:Nn \tex_errorcontextlines:D { -1 }
33035     \tex_showbox:D \use:n {#2}
33036     \group_end:
33037 }
33038 {
33039     \msg_error:nnx { kernel } { variable-not-defined }
33040     { \token_to_str:N #2 }
33041 }
33042 }
33043 \cs_generate_variant:Nn \__box_show:NNnn { NNff }

(End definition for \__box_show:NNnn.)

```

## 85.10 Horizontal mode boxes

**\hbox:n** (*The test suite for this command, and others in this file, is m3box002.lvt.*)  
Put a horizontal box directly into the input stream.

```

33044 \cs_new_protected:Npn \hbox:n #1
33045     { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }

(End definition for \hbox:n. This function is documented on page 277.)

```

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
33046 \cs_new_protected:Npn \hbox_set:Nn #1#2
33047 {
33048     \tex_setbox:D #1 \tex_hbox:D
33049     { \color_group_begin: #2 \color_group_end: }
33050 }
33051 \cs_new_protected:Npn \hbox_gset:Nn #1#2
33052 {
33053     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
33054     { \color_group_begin: #2 \color_group_end: }
33055 }
33056 \cs_generate_variant:Nn \hbox_set:Nn { c }
33057 \cs_generate_variant:Nn \hbox_gset:Nn { c }

(End definition for \hbox_set:Nn and \hbox_gset:Nn. These functions are documented on page 277.)

```

**\hbox\_set\_to\_wd:Nnn** Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```

\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
33058 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
33059 {
33060     \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
33061     { \color_group_begin: #3 \color_group_end: }
33062 }
33063 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
33064 {
33065     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
33066     { \color_group_begin: #3 \color_group_end: }
33067 }
33068 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
33069 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnn { c }

```

(End definition for `\hbox_set_to_wd:Nnn` and `\hbox_gset_to_wd:Nnn`. These functions are documented on page 278.)

`\hbox_set:Nw` Storing material in a horizontal box. This type is useful in environment definitions.

```
33070 \cs_new_protected:Npn \hbox_set:Nw #1
33071 {
33072     \tex_setbox:D #1 \tex_hbox:D
33073     \c_group_begin_token
33074     \color_group_begin:
33075 }
33076 \cs_new_protected:Npn \hbox_gset:Nw #1
33077 {
33078     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
33079     \c_group_begin_token
33080     \color_group_begin:
33081 }
33082 \cs_generate_variant:Nn \hbox_set:Nw { c }
33083 \cs_generate_variant:Nn \hbox_gset:Nw { c }
33084 \cs_new_protected:Npn \hbox_set_end:
33085 {
33086     \color_group_end:
33087     \c_group_end_token
33088 }
33089 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:
```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page 278.)

`\hbox_set_to_wd:Nnw` Combining the above ideas.

```
33090 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
33091 {
33092     \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
33093     \c_group_begin_token
33094     \color_group_begin:
33095 }
33096 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
33097 {
33098     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
33099     \c_group_begin_token
33100     \color_group_begin:
33101 }
33102 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
33103 \cs_generate_variant:Nn \hbox_gset_to_wd:Nnw { c }
```

(End definition for `\hbox_set_to_wd:Nnw` and `\hbox_gset_to_wd:Nnw`. These functions are documented on page 278.)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

```
33104 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
33105 {
33106     \tex_hbox:D to \__box_dim_eval:n {#1}
33107     { \color_group_begin: #2 \color_group_end: }
33108 }
33109 \cs_new_protected:Npn \hbox_to_zero:n #1
33110 {
33111     \tex_hbox:D to \c_zero_dim
```

```

33112     { \color_group_begin: #1 \color_group_end: }
33113 }
```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 277.)

**\hbox\_overlap\_center:n** Put a zero-sized box with the contents pushed against one side (which makes it stick out on the other) directly into the input stream.

```

33114 \cs_new_protected:Npn \hbox_overlap_center:n #1
33115   { \hbox_to_zero:n { \tex_hss:D #1 \tex_hss:D } }
33116 \cs_new_protected:Npn \hbox_overlap_left:n #1
33117   { \hbox_to_zero:n { \tex_hss:D #1 } }
33118 \cs_new_protected:Npn \hbox_overlap_right:n #1
33119   { \hbox_to_zero:n { #1 \tex_hss:D } }
```

(End definition for `\hbox_overlap_center:n`, `\hbox_overlap_left:n`, and `\hbox_overlap_right:n`. These functions are documented on page 278.)

**\hbox\_unpack:N** Unpacking a box and if requested also clear it.

```

33120 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
33121 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
33122 \cs_generate_variant:Nn \hbox_unpack:N { c }
33123 \cs_generate_variant:Nn \hbox_unpack_drop:N { c }
```

(End definition for `\hbox_unpack:N` and `\hbox_unpack_drop:N`. These functions are documented on page 278.)

## 85.11 Vertical mode boxes

TeX ends these boxes directly with the internal `end_graf` routine. This means that there is no `\par` at the end of vertical boxes unless we insert one. Thus all vertical boxes include a `\par` just before closing the color group.

**\vbox:n** The following test files are used for this code: `m3box003.lvt`.

The following test files are used for this code: `m3box003.lvt`.

**\vbox\_top:n** Put a vertical box directly into the input stream.

```

33124 \cs_new_protected:Npn \vbox:n #1
33125   { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
33126 \cs_new_protected:Npn \vbox_top:n #1
33127   { \tex_vtop:D { \color_group_begin: #1 \par \color_group_end: } }
```

(End definition for `\vbox:n` and `\vbox_top:n`. These functions are documented on page 279.)

**\vbox\_to\_ht:nn** Put a vertical box directly into the input stream.

```

33128 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
33129   {
33130     \tex_vbox:D to \__box_dim_eval:n {#1}
33131     { \color_group_begin: #2 \par \color_group_end: }
33132   }
33133 \cs_new_protected:Npn \vbox_to_zero:n #1
33134   {
33135     \tex_vbox:D to \c_zero_dim
33136     { \color_group_begin: #1 \par \color_group_end: }
33137 }
```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 279.)

`\vbox_set:Nn` Storing material in a vertical box with a natural height.

```
33138 \cs_new_protected:Npn \vbox_set:Nn #1#2
33139 {
33140     \tex_setbox:D #1 \tex_vbox:D
33141     { \color_group_begin: #2 \par \color_group_end: }
33142 }
33143 \cs_new_protected:Npn \vbox_gset:Nn #1#2
33144 {
33145     \tex_global:D \tex_setbox:D #1 \tex_vbox:D
33146     { \color_group_begin: #2 \par \color_group_end: }
33147 }
33148 \cs_generate_variant:Nn \vbox_set:Nn { c }
33149 \cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 279.)

`\vbox_set_top:Nn` Storing material in a vertical box with a natural height and reference point at the baseline of the first object in the box.

```
33150 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
33151 {
33152     \tex_setbox:D #1 \tex_vtop:D
33153     { \color_group_begin: #2 \par \color_group_end: }
33154 }
33155 \cs_new_protected:Npn \vbox_gset_top:Nn #1#2
33156 {
33157     \tex_global:D \tex_setbox:D #1 \tex_vtop:D
33158     { \color_group_begin: #2 \par \color_group_end: }
33159 }
33160 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
33161 \cs_generate_variant:Nn \vbox_gset_top:Nn { c }
```

(End definition for `\vbox_set_top:Nn` and `\vbox_gset_top:Nn`. These functions are documented on page 279.)

`\vbox_set_to_ht:Nnn` Storing material in a vertical box with a specified height.

```
33162 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
33163 {
33164     \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
33165     { \color_group_begin: #3 \par \color_group_end: }
33166 }
33167 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
33168 {
33169     \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
33170     { \color_group_begin: #3 \par \color_group_end: }
33171 }
33172 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
33173 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnn { c }
```

(End definition for `\vbox_set_to_ht:Nnn` and `\vbox_gset_to_ht:Nnn`. These functions are documented on page 279.)

```

\ vbox_set:Nw Storing material in a vertical box. This type is useful in environment definitions.
\ vbox_set:cw
\ vbox_gset:Nw
\ vbox_gset:cw
\ vbox_set_end:
\ vbox_gset_end:
33174 \cs_new_protected:Npn \vbox_set:Nw #1
33175 {
33176     \tex_setbox:D #1 \tex_vbox:D
33177     \c_group_begin_token
33178     \color_group_begin:
33179 }
33180 \cs_new_protected:Npn \vbox_gset:Nw #1
33181 {
33182     \tex_global:D \tex_setbox:D #1 \tex_vbox:D
33183     \c_group_begin_token
33184     \color_group_begin:
33185 }
33186 \cs_generate_variant:Nn \vbox_set:Nw { c }
33187 \cs_generate_variant:Nn \vbox_gset:Nw { c }
33188 \cs_new_protected:Npn \vbox_set_end:
33189 {
33190     \par
33191     \color_group_end:
33192     \c_group_end_token
33193 }
33194 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:

```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page [279](#).)

`\vbox_set_to_ht:Nnw` A combination of the above ideas.

```

\ vbox_set_to_ht:cw
\ vbox_gset_to_ht:Nnw
\ vbox_gset_to_ht:cw
33195 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
33196 {
33197     \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
33198     \c_group_begin_token
33199     \color_group_begin:
33200 }
33201 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
33202 {
33203     \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
33204     \c_group_begin_token
33205     \color_group_begin:
33206 }
33207 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
33208 \cs_generate_variant:Nn \vbox_gset_to_ht:Nnw { c }

```

(End definition for `\vbox_set_to_ht:Nnw` and `\vbox_gset_to_ht:Nnw`. These functions are documented on page [280](#).)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.

```

\ vbox_unpack:c
\ vbox_unpack_drop:N
\ vbox_unpack_drop:c
33209 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
33210 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
33211 \cs_generate_variant:Nn \vbox_unpack:N { c }
33212 \cs_generate_variant:Nn \vbox_unpack_drop:N { c }

```

(End definition for `\vbox_unpack:N` and `\vbox_unpack_drop:N`. These functions are documented on page [280](#).)

```

\ vbox_set_split_to_ht:NNn Splitting a vertical box in two.
\ vbox_set_split_to_ht:cNn
\ vbox_set_split_to_ht:Ncn
\ vbox_set_split_to_ht:ccn
\ vbox_gset_split_to_ht:NNn
\ vbox_gset_split_to_ht:cNn
\ vbox_gset_split_to_ht:Ncn
\ vbox_gset_split_to_ht:ccn

33213 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
33214   { \tex_setbox:D #1 \tex_vsprint:D #2 to \__box_dim_eval:n {#3} }
33215 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
33216 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
33217   {
33218     \tex_global:D \tex_setbox:D #1
33219     \tex_vsprint:D #2 to \__box_dim_eval:n {#3}
33220   }
33221 \cs_generate_variant:Nn \vbox_gset_split_to_ht:NNn { c , Nc , cc }

(End definition for \vbox_set_split_to_ht:NNn and \vbox_gset_split_to_ht:NNn. These functions are documented on page 280.)

```

## 85.12 Affine transformations

\l\_box\_angle\_fp When rotating boxes, the angle itself may be needed by the engine-dependent code. This is done using the `fp` module so that the value is tidied up properly.

```
33222 \fp_new:N \l_box_angle_fp
```

(End definition for \l\_box\_angle\_fp.)

\l\_box\_cos\_fp \l\_box\_sin\_fp These are used to hold the calculated sine and cosine values while carrying out a rotation.

```
33223 \fp_new:N \l_box_cos_fp
33224 \fp_new:N \l_box_sin_fp
```

(End definition for \l\_box\_cos\_fp and \l\_box\_sin\_fp.)

\l\_box\_top\_dim These are the positions of the four edges of a box before manipulation.

```
33225 \dim_new:N \l_box_top_dim
33226 \dim_new:N \l_box_bottom_dim
33227 \dim_new:N \l_box_left_dim
33228 \dim_new:N \l_box_right_dim
```

(End definition for \l\_box\_top\_dim and others.)

\l\_box\_top\_new\_dim These are the positions of the four edges of a box after manipulation.

```
33229 \dim_new:N \l_box_top_new_dim
33230 \dim_new:N \l_box_bottom_new_dim
33231 \dim_new:N \l_box_left_new_dim
33232 \dim_new:N \l_box_right_new_dim
```

(End definition for \l\_box\_top\_new\_dim and others.)

\l\_box\_internal\_box Scratch space, but also needed by some parts of the driver.

```
33233 \box_new:N \l_box_internal_box
```

(End definition for \l\_box\_internal\_box.)

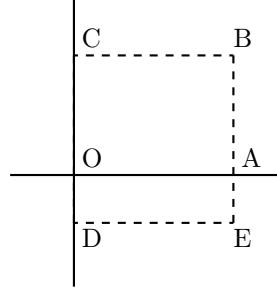


Figure 1: Co-ordinates of a box prior to rotation.

```

\box_rotate:Nn          Rotation of a box starts with working out the relevant sine and cosine. The actual
\box_rotate:cn          rotation is in an auxiliary to keep the flow slightly clearer
\box_grotate:Nn          33234 \cs_new_protected:Npn \box_rotate:Nn #1#2
\box_grotate:cn          33235 { \__box_rotate:NnN #1 {#2} \hbox_set:Nn }
\__box_rotate:NnN         33236 \cs_generate_variant:Nn \box_rotate:Nn { c }
\__box_rotate:N           33237 \cs_new_protected:Npn \box_grotate:Nn #1#2
\__box_rotate_xdir:nN    33238 { \__box_rotate:NnN #1 {#2} \hbox_gset:Nn }
\__box_rotate_ydir:nN    33239 \cs_generate_variant:Nn \box_grotate:Nn { c }
\__box_rotate_quadrant_one: 33240 \cs_new_protected:Npn \__box_rotate:NnN #1#2#3
\__box_rotate_quadrant_two: 33241 {
\__box_rotate_quadrant_three: 33242 #3 #1
\__box_rotate_quadrant_four: 33243 {
\__box_rotate_quadrant_one: 33244 \fp_set:Nn \l__box_angle_fp {#2}
\__box_rotate_quadrant_two: 33245 \fp_set:Nn \l__box_sin_fp { sind ( \l__box_angle_fp ) }
\__box_rotate_quadrant_three: 33246 \fp_set:Nn \l__box_cos_fp { cosd ( \l__box_angle_fp ) }
\__box_rotate_quadrant_four: 33247 \__box_rotate:N #1
\__box_rotate_quadrant_one: 33248 }
\__box_rotate_quadrant_two: 33249 }

```

The edges of the box are then recorded: the left edge is always at zero. Rotation of the four edges then takes place: this is most efficiently done on a quadrant basis.

```

33250 \cs_new_protected:Npn \__box_rotate:N #1
33251 {
33252 \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
33253 \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
33254 \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
33255 \dim_zero:N \l__box_left_dim

```

The next step is to work out the  $x$  and  $y$  coordinates of vertices of the rotated box in relation to its original coordinates. The box can be visualized with vertices  $B$ ,  $C$ ,  $D$  and  $E$  is illustrated (Figure 1). The vertex  $O$  is the reference point on the baseline, and in this implementation is also the centre of rotation. The formulae are, for a point  $P$  and angle  $\alpha$ :

$$\begin{aligned}
P'_x &= P_x - O_x \\
P'_y &= P_y - O_y \\
P''_x &= (P'_x \cos(\alpha)) - (P'_y \sin(\alpha)) \\
P''_y &= (P'_x \sin(\alpha)) + (P'_y \cos(\alpha)) \\
P'''_x &= P''_x + O_x + L_x \\
P'''_y &= P''_y + O_y
\end{aligned}$$

The “extra” horizontal translation  $L_x$  at the end is calculated so that the leftmost point of the resulting box has  $x$ -coordinate 0. This is desirable as TeX boxes must have the reference point at the left edge of the box. (As  $O$  is always  $(0,0)$ , this part of the calculation is omitted here.)

```

33256      \fp_compare:nNnTF \l_box_sin_fp > \c_zero_fp
33257      {
33258          \fp_compare:nNnTF \l_box_cos_fp > \c_zero_fp
33259          { \__box_rotate_quadrant_one: }
33260          { \__box_rotate_quadrant_two: }
33261      }
33262      {
33263          \fp_compare:nNnTF \l_box_cos_fp < \c_zero_fp
33264          { \__box_rotate_quadrant_three: }
33265          { \__box_rotate_quadrant_four: }
33266      }

```

The position of the box edges are now known, but the box at this stage be misplaced relative to the current TeX reference point. So the content of the box is moved such that the reference point of the rotated box is in the same place as the original.

```

33267      \hbox_set:Nn \l_box_internal_box { \box_use:N #1 }
33268      \hbox_set:Nn \l_box_internal_box
33269      {
33270          \__kernel_kern:n { -\l_box_left_new_dim }
33271          \hbox:n
33272          {
33273              \__box_backend_rotate:Nn
33274                  \l_box_internal_box
33275                  \l_box_angle_fp
33276          }
33277      }

```

Tidy up the size of the box so that the material is actually inside the bounding box. The result can then be used to reset the original box.

```

33278      \box_set_ht:Nn \l_box_internal_box { \l_box_top_new_dim }
33279      \box_set_dp:Nn \l_box_internal_box { -\l_box_bottom_new_dim }
33280      \box_set_wd:Nn \l_box_internal_box
33281      { \l_box_right_new_dim - \l_box_left_new_dim }
33282      \box_use_drop:N \l_box_internal_box
33283  }

```

These functions take a general point (#1,#2) and rotate its location about the origin, using the previously-set sine and cosine values. Each function gives only one component of the location of the updated point. This is because for rotation of a box each step needs only one value, and so performance is gained by avoiding working out both  $x'$  and  $y'$  at the same time. Contrast this with the equivalent function in the l3coffins module, where both parts are needed.

```

33284  \cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
33285  {
33286      \dim_set:Nn #3
33287      {
33288          \fp_to_dim:n
33289          {
33290              \l_box_cos_fp * \dim_to_fp:n {#1}
33291              - \l_box_sin_fp * \dim_to_fp:n {#2}

```

```

33292         }
33293     }
33294   }
33295 \cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
33296   {
33297     \dim_set:Nn #3
33298     {
33299       \fp_to_dim:n
33300       {
33301         \l__box_sin_fp * \dim_to_fp:n {#1}
33302         + \l__box_cos_fp * \dim_to_fp:n {#2}
33303       }
33304     }
33305   }

```

Rotation of the edges is done using a different formula for each quadrant. In every case, the top and bottom edges only need the resulting  $y$ -values, whereas the left and right edges need the  $x$ -values. Each case is a question of picking out which corner ends up at with the maximum top, bottom, left and right value. Doing this by hand means a lot less calculating and avoids lots of comparisons.

```

33306 \cs_new_protected:Npn \__box_rotate_quadrant_one:
33307   {
33308     \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
33309     \l__box_top_new_dim
33310     \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
33311     \l__box_bottom_new_dim
33312     \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
33313     \l__box_left_new_dim
33314     \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
33315     \l__box_right_new_dim
33316   }
33317 \cs_new_protected:Npn \__box_rotate_quadrant_two:
33318   {
33319     \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
33320     \l__box_top_new_dim
33321     \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
33322     \l__box_bottom_new_dim
33323     \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
33324     \l__box_left_new_dim
33325     \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
33326     \l__box_right_new_dim
33327   }
33328 \cs_new_protected:Npn \__box_rotate_quadrant_three:
33329   {
33330     \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
33331     \l__box_top_new_dim
33332     \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
33333     \l__box_bottom_new_dim
33334     \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
33335     \l__box_left_new_dim
33336     \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
33337     \l__box_right_new_dim
33338   }
33339 \cs_new_protected:Npn \__box_rotate_quadrant_four:

```

```

33340  {
33341    \__box_rotate_ydir:nnN \l_box_left_dim  \l_box_top_dim
33342      \l_box_top_new_dim
33343    \__box_rotate_ydir:nnN \l_box_right_dim \l_box_bottom_dim
33344      \l_box_bottom_new_dim
33345    \__box_rotate_xdir:nnN \l_box_left_dim  \l_box_bottom_dim
33346      \l_box_left_new_dim
33347    \__box_rotate_xdir:nnN \l_box_right_dim \l_box_top_dim
33348      \l_box_right_new_dim
33349  }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 284.)

`\l_box_scale_x_fp`  
`\l_box_scale_y_fp`

```

33350 \fp_new:N \l_box_scale_x_fp
33351 \fp_new:N \l_box_scale_y_fp

```

(End definition for `\l_box_scale_x_fp` and `\l_box_scale_y_fp`.)

Resizing a box starts by working out the various dimensions of the existing box.

```

33352 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:Nnn
33353  {
33354    \__box_resize_to_wd_and_ht_plus_dp:NnnN #1{#2}{#3}
33355      \hbox_set:Nn
33356  }
33357 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
33358 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1{#2}{#3}
33359  {
33360    \__box_resize_to_wd_and_ht_plus_dp:NnnN #1{#2}{#3}
33361      \hbox_gset:Nn
33362  }
33363 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
33364 \cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1{#2}{#3}{#4}
33365  {
33366    #4 #1
33367  {
33368    \__box_resize_set_corners:N #1

```

The *x*-scaling and resulting box size is easy enough to work out: the dimension is that given as #2, and the scale is simply the new width divided by the old one.

```

33369   \fp_set:Nn \l_box_scale_x_fp
33370     { \dim_to_fp:n {#2} / \dim_to_fp:n { \l_box_right_dim } }

```

The *y*-scaling needs both the height and the depth of the current box.

```

33371   \fp_set:Nn \l_box_scale_y_fp
33372  {
33373    \dim_to_fp:n {#3}
33374    / \dim_to_fp:n { \l_box_top_dim - \l_box_bottom_dim }
33375  }

```

Hand off to the auxiliary which does the rest of the work.

```

33376   \__box_resize:N #1
33377  }
33378  }
33379 \cs_new_protected:Npn \__box_resize_set_corners:N #1
33380  {

```

```

33381      \dim_set:Nn \l__box_top_dim   { \box_ht:N #1 }
33382      \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
33383      \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
33384      \dim_zero:N \l__box_left_dim
33385  }

```

With at least one real scaling to do, the next phase is to find the new edge co-ordinates. In the  $x$  direction this is relatively easy: just scale the right edge. In the  $y$  direction, both dimensions have to be scaled, and this again needs the absolute scale value. Once that is all done, the common resize/rescale code can be employed.

```

33386 \cs_new_protected:Npn \__box_resize:N #1
33387  {
33388      \__box_resize>NNN \l__box_right_new_dim
33389          \l__box_scale_x_fp \l__box_right_dim
33390      \__box_resize>NNN \l__box_bottom_new_dim
33391          \l__box_scale_y_fp \l__box_bottom_dim
33392      \__box_resize>NNN \l__box_top_new_dim
33393          \l__box_scale_y_fp \l__box_top_dim
33394      \__box_resize_common:N #1
33395  }
33396 \cs_new_protected:Npn \__box_resize>NNN #1#2#3
33397  {
33398      \dim_set:Nn #1
33399          { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
33400  }

```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 283.)

`\box_resize_to_ht:Nn`  
`\box_resize_to_ht:cn`  
`\box_gresize_to_ht:Nn`  
`\box_gresize_to_ht:cn`

`\__box_resize_to_ht:Nnn`  
`\box_resize_to_ht_plus_dp:Nn`  
`\box_resize_to_ht_plus_dp:cn`  
`\box_gresize_to_ht_plus_dp:Nn`  
`\box_gresize_to_ht_plus_dp:cn`  
`\__box_resize_to_ht_plus_dp:Nnn`  
`\box_resize_to_wd:Nn`  
`\box_resize_to_wd:cn`  
`\box_gresize_to_wd:Nn`  
`\box_gresize_to_wd:cn`  
`\__box_resize_to_wd:Nnn`  
`\box_resize_to_wd_and_ht:Nnn`  
`\box_resize_to_wd_and_ht:cnn`  
`\box_gresize_to_wd_and_ht:Nnn`  
`\box_gresize_to_wd_and_ht:cnn`  
`\__box_resize_to_wd_ht:NnnN`

Scaling to a (total) height or to a width is a simplified version of the main resizing operation, with the scale simply copied between the two parts. The internal auxiliary is called using the scaling value twice, as the sign for both parts is needed (as this allows the same internal code to be used as for the general case).

```

33401 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
33402  { \__box_resize_to_ht:Nnn #1 {#2} \hbox_set:Nn }
33403 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
33404 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2
33405  { \__box_resize_to_ht:Nnn #1 {#2} \hbox_gset:Nn }
33406 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c }
33407 \cs_new_protected:Npn \__box_resize_to_ht:Nnn #1#2#3
33408  {
33409      #3 #1
33410      {
33411          \__box_resize_set_corners:N #1
33412          \fp_set:Nn \l__box_scale_y_fp
33413              {
33414                  \dim_to_fp:n {#2}
33415                  / \dim_to_fp:n { \l__box_top_dim }
33416              }
33417          \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
33418          \__box_resize:N #1
33419      }
33420  }
33421 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2

```

```

33422 { \__box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
33423 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
33424 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
33425 { \__box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
33426 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
33427 \cs_new_protected:Npn \__box_resize_to_ht_plus_dp:NnN #1#2#3
33428 {
33429 #3 #1
33430 {
33431 \__box_resize_set_corners:N #1
33432 \fp_set:Nn \l__box_scale_y_fp
33433 {
33434 \dim_to_fp:n {#2}
33435 / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
33436 }
33437 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
33438 \__box_resize:N #1
33439 }
33440 }
33441 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
33442 { \__box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
33443 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
33444 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
33445 { \__box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
33446 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
33447 \cs_new_protected:Npn \__box_resize_to_wd:NnN #1#2#3
33448 {
33449 #3 #1
33450 {
33451 \__box_resize_set_corners:N #1
33452 \fp_set:Nn \l__box_scale_x_fp
33453 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
33454 \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
33455 \__box_resize:N #1
33456 }
33457 }
33458 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
33459 { \__box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
33460 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
33461 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
33462 { \__box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
33463 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
33464 \cs_new_protected:Npn \__box_resize_to_wd_and_ht:NnnN #1#2#3#4
33465 {
33466 #4 #1
33467 {
33468 \__box_resize_set_corners:N #1
33469 \fp_set:Nn \l__box_scale_x_fp
33470 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
33471 \fp_set:Nn \l__box_scale_y_fp
33472 {
33473 \dim_to_fp:n {#3}
33474 / \dim_to_fp:n { \l__box_top_dim }
33475 }

```

```

33476           \__box_resize:N #1
33477       }
33478   }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 282.)

```

\box_scale:Nnn
\box_scale:cnn
\box_gscale:Nnn
\box_gscale:cnn
\__box_scale:NnnN
\__box_scale:N

```

When scaling a box, setting the scaling itself is easy enough. The new dimensions are also relatively easy to find, allowing only for the need to keep them positive in all cases. Once that is done then after a check for the trivial scaling a hand-off can be made to the common code. The code here is split into two as this allows sharing with the auto-resizing functions.

```

33479 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
33480   { \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
33481 \cs_generate_variant:Nn \box_scale:Nnn { c }
33482 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
33483   { \__box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
33484 \cs_generate_variant:Nn \box_gscale:Nnn { c }
33485 \cs_new_protected:Npn \__box_scale:NnnN #1#2#3#4
33486   {
33487     #4 #1
33488   {
33489     \fp_set:Nn \l__box_scale_x_fp {#2}
33490     \fp_set:Nn \l__box_scale_y_fp {#3}
33491     \__box_scale:N #1
33492   }
33493   }
33494 \cs_new_protected:Npn \__box_scale:N #1
33495   {
33496     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
33497     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
33498     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
33499     \dim_zero:N \l__box_left_dim
33500     \dim_set:Nn \l__box_top_new_dim
33501       { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
33502     \dim_set:Nn \l__box_bottom_new_dim
33503       { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
33504     \dim_set:Nn \l__box_right_new_dim
33505       { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
33506     \__box_resize_common:N #1
33507   }

```

(End definition for `\box_scale:Nnn` and others. These functions are documented on page 284.)

Although autosizing a box uses dimensions, it has more in common in implementation with scaling. As such, most of the real work here is done elsewhere.

```

33508 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
33509   { \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
33510 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
33511 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
33512   { \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
33513 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
33514 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
33515   {
33516     \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

33517     \hbox_set:Nn
33518   }
33519   \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
33520   \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
33521   {
33522     \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
33523     \hbox_gset:Nn
33524   }
33525   \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
33526   \cs_new_protected:Npn \__box_autosize:NnnnN #1#2#3#4#5
33527   {
33528     #5 #1
33529   {
33530     \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
33531     \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
33532     \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
33533     { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
33534     { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
33535     \__box_scale:N #1
33536   }
33537 }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page 282.)

`\__box_resize_common:N` The main resize function places its input into a box which start off with zero width, and includes the handles for engine rescaling.

```

33538 \cs_new_protected:Npn \__box_resize_common:N #1
33539   {
33540     \hbox_set:Nn \l__box_internal_box
33541     {
33542       \__box_backend_scale:Nnn
33543       #1
33544       \l__box_scale_x_fp
33545       \l__box_scale_y_fp
33546     }

```

The new height and depth can be applied directly.

```

33547   \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
33548   {
33549     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
33550     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
33551   }
33552   {
33553     \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
33554     \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
33555   }

```

Things are not quite as obvious for the width, as the reference point needs to remain unchanged. For positive scaling factors resizing the box is all that is needed. However, for case of a negative scaling the material must be shifted such that the reference point ends up in the right place.

```

33556   \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
33557   {
33558     \hbox_to_wd:nn { \l__box_right_new_dim }

```

```

33559   {
33560     \__kernel_kern:n { \l__box_right_new_dim }
33561     \box_use_drop:N \l__box_internal_box
33562     \tex_hss:D
33563   }
33564 }
33565 {
33566   \box_set_wd:Nn \l__box_internal_box { \l__box_right_new_dim }
33567   \hbox:n
33568   {
33569     \__kernel_kern:n { 0pt }
33570     \box_use_drop:N \l__box_internal_box
33571     \tex_hss:D
33572   }
33573 }
33574 }
```

(End definition for `\__box_resize_common:N`.)

33575 `</package>`

# Chapter 86

## I3coffins Implementation

```
33576  <*package>
33577  <@@=coffin>
```

### 86.1 Coffins: data structures and general variables

\l\_\_coffin\_internal\_box    Scratch variables.

```
33578  \box_new:N \l__coffin_internal_box
33579  \dim_new:N \l__coffin_internal_dim
33580  \tl_new:N \l__coffin_internal_tl
```

(End definition for \l\_\_coffin\_internal\_box, \l\_\_coffin\_internal\_dim, and \l\_\_coffin\_internal\_tl.)

\c\_\_coffin\_corners\_prop    The “corners”; of a coffin define the real content, as opposed to the TeX bounding box. They all start off in the same place, of course.

```
33581  \prop_const_from_keyval:Nn \c__coffin_corners_prop
33582  {
33583      tl = { Opt } { Opt } ,
33584      tr = { Opt } { Opt } ,
33585      bl = { Opt } { Opt } ,
33586      br = { Opt } { Opt } ,
33587 }
```

(End definition for \c\_\_coffin\_corners\_prop.)

\c\_\_coffin\_poles\_prop    Pole positions are given for horizontal, vertical and reference-point based values.

```
33588  \prop_const_from_keyval:Nn \c__coffin_poles_prop
33589  {
33590      l = { Opt } { Opt } { Opt } { 1000pt } ,
33591      hc = { Opt } { Opt } { Opt } { 1000pt } ,
33592      r = { Opt } { Opt } { Opt } { 1000pt } ,
33593      b = { Opt } { Opt } { 1000pt } { Opt } ,
33594      vc = { Opt } { Opt } { 1000pt } { Opt } ,
33595      t = { Opt } { Opt } { 1000pt } { Opt } ,
33596      B = { Opt } { Opt } { 1000pt } { Opt } ,
33597      H = { Opt } { Opt } { 1000pt } { Opt } ,
33598      T = { Opt } { Opt } { 1000pt } { Opt } ,
33599 }
```

*(End definition for \c\_coffin\_poles\_prop.)*

\l\_coffin\_slope\_A\_fp  
\l\_coffin\_slope\_B\_fp  
33600 \fp\_new:N \l\_coffin\_slope\_A\_fp  
33601 \fp\_new:N \l\_coffin\_slope\_B\_fp

*(End definition for \l\_coffin\_slope\_A\_fp and \l\_coffin\_slope\_B\_fp.)*

\l\_coffin\_error\_bool  
For propagating errors so that parts of the code can work around them.

33602 \bool\_new:N \l\_coffin\_error\_bool

*(End definition for \l\_coffin\_error\_bool.)*

\l\_coffin\_offset\_x\_dim  
\l\_coffin\_offset\_y\_dim  
The offset between two sets of coffin handles when typesetting. These values are corrected from those requested in an alignment for the positions of the handles.

33603 \dim\_new:N \l\_coffin\_offset\_x\_dim  
33604 \dim\_new:N \l\_coffin\_offset\_y\_dim

*(End definition for \l\_coffin\_offset\_x\_dim and \l\_coffin\_offset\_y\_dim.)*

\l\_coffin\_pole\_a\_t1  
\l\_coffin\_pole\_b\_t1  
Needed for finding the intersection of two poles.

33605 \tl\_new:N \l\_coffin\_pole\_a\_t1  
33606 \tl\_new:N \l\_coffin\_pole\_b\_t1

*(End definition for \l\_coffin\_pole\_a\_t1 and \l\_coffin\_pole\_b\_t1.)*

\l\_coffin\_x\_dim  
\l\_coffin\_y\_dim  
\l\_coffin\_x\_prime\_dim  
\l\_coffin\_y\_prime\_dim  
For calculating intersections and so forth.

33607 \dim\_new:N \l\_coffin\_x\_dim  
33608 \dim\_new:N \l\_coffin\_y\_dim  
33609 \dim\_new:N \l\_coffin\_x\_prime\_dim  
33610 \dim\_new:N \l\_coffin\_y\_prime\_dim

*(End definition for \l\_coffin\_x\_dim and others.)*

## 86.2 Basic coffin functions

There are a number of basic functions needed for creating coffins and placing material in them. This all relies on the following data structures.

\\_coffin\_to\_value:N  
Coffins are a two-part structure and we rely on the internal nature of box allocation to make everything work. As such, we need an interface to turn coffin identifiers into numbers. For the purposes here, the signature allowed is N despite the nature of the underlying primitive.

33611 \cs\_new\_eq:NN \\_coffin\_to\_value:N \tex\_number:D

*(End definition for \\_coffin\_to\_value:N.)*

**\coffin\_if\_exist:p:N** Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. A cleaner way to handle this is provided here: both the box and the coffin structure are checked.

```

33612 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }
33613   {
33614     \cs_if_exist:NTF #1
33615       {
33616         \cs_if_exist:cTF { coffin ~ \__coffin_to_value:N #1 ~ poles }
33617           { \prg_return_true: }
33618           { \prg_return_false: }
33619       }
33620       { \prg_return_false: }
33621     }
33622 \prg_generate_conditional_variant:Nnn \coffin_if_exist:N
33623   { c } { p , T , F , TF }
```

(End definition for `\coffin_if_exist:NTF`. This function is documented on page 285.)

**\\_\_coffin\_if\_exist:NT** Several of the higher-level coffin functions would give multiple errors if the coffin does not exist. So a wrapper is provided to deal with this correctly, issuing an error on erroneous use.

```

33624 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
33625   {
33626     \coffin_if_exist:NTF #1
33627       { #2 }
33628       {
33629         \msg_error:nnx { coffin } { unknown }
33630           { \token_to_str:N #1 }
33631       }
33632     }
```

(End definition for `\__coffin_if_exist:NT`.)

**\coffin\_clear:N** Clearing coffins means emptying the box and resetting all of the structures.

```

33633 \cs_new_protected:Npn \coffin_clear:N #1
33634   {
33635     \__coffin_if_exist:NT #1
33636       {
33637         \box_clear:N #1
33638         \__coffin_reset_structure:N #1
33639       }
33640     }
33641 \cs_generate_variant:Nn \coffin_clear:N { c }
33642 \cs_new_protected:Npn \coffin_gclear:N #1
33643   {
33644     \__coffin_if_exist:NT #1
33645       {
33646         \box_gclear:N #1
33647         \__coffin_greset_structure:N #1
33648       }
33649     }
33650 \cs_generate_variant:Nn \coffin_gclear:N { c }
```

(End definition for `\coffin_clear:N` and `\coffin_gclear:N`. These functions are documented on page 285.)

**\coffin\_new:N** Creating a new coffin means making the underlying box and adding the data structures. The `\debug_suspend:` and `\debug_resume:` functions prevent `\prop_gclear_new:c` from writing useless information to the log file.

```

33651 \cs_new_protected:Npn \coffin_new:N #1
33652 {
33653     \box_new:N #1
33654     \debug_suspend:
33655     \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
33656     \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
33657     \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
33658         \c_coffin_corners_prop
33659     \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
33660         \c_coffin_poles_prop
33661     \debug_resume:
33662 }
33663 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for `\coffin_new:N`. This function is documented on page 285.)

**\hcoffin\_set:Nn** Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```

33664 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
33665 {
33666     \__coffin_if_exist:NT #1
33667     {
33668         \hbox_set:Nn #1
33669         {
33670             \color_ensure_current:
33671             #2
33672         }
33673         \__coffin_update:N #1
33674     }
33675 }
33676 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
33677 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
33678 {
33679     \__coffin_if_exist:NT #1
33680     {
33681         \hbox_gset:Nn #1
33682         {
33683             \color_ensure_current:
33684             #2
33685         }
33686         \__coffin_gupdate:N #1
33687     }
33688 }
33689 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }

```

(End definition for `\hcoffin_set:Nn` and `\hcoffin_gset:Nn`. These functions are documented on page 286.)

**\vcoffin\_set:Nnn** Setting vertical coffins is more complex. First, the material is typeset with a given width. The default handles and poles are set as for a horizontal coffin, before finding the top baseline using a temporary box. No `\color_ensure_current:` here as that would add a

```

\__coffin_set_vertical:NnnNN
\__coffin_set_vertical_aux:

```

whatsit to the start of the vertical box and mess up the location of the T pole (see *T<sub>EX</sub> by Topic* for discussion of the \vtop primitive, used to do the measuring).

```

33690 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
33691 {
33692     \__coffin_set_vertical:NnnNN #1 {#2} {#3}
33693     \vbox_set:Nn \__coffin_update:N
33694 }
33695 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
33696 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
33697 {
33698     \__coffin_set_vertical:NnnNN #1 {#2} {#3}
33699     \vbox_gset:Nn \__coffin_gupdate:N
33700 }
33701 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
33702 \cs_new_protected:Npn \__coffin_set_vertical:NnnNN #1#2#3#4#5
33703 {
33704     \__coffin_if_exist:NT #1
33705 {
33706     #4 #1
33707     {
33708         \dim_set:Nn \tex_hsize:D {#2}
33709         \__coffin_set_vertical_aux:
33710         #3
33711     }
33712     #5 #1
33713     \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
33714     \__coffin_set_pole:Nnx #1 { T }
33715     {
33716         { Opt }
33717         {
33718             \dim_eval:n
33719             { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
33720         }
33721         { 1000pt }
33722         { Opt }
33723     }
33724     \box_clear:N \l__coffin_internal_box
33725 }
33726 }
33727 \cs_new_protected:Npx \__coffin_set_vertical_aux:
33728 {
33729     \bool_lazy_and:nnT
33730     { \cs_if_exist_p:N \fmtname }
33731     { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
33732     {
33733         \dim_set_eq:NN \exp_not:N \linewidth \tex_hsize:D
33734         \dim_set_eq:NN \exp_not:N \columnwidth \tex_hsize:D
33735     }
33736 }

```

(End definition for \vcoffin\_set:Nnn and others. These functions are documented on page 286.)

**\hcoffin\_set:Nw** These are the “begin”/“end” versions of the above: watch the grouping!

**\hcoffin\_set:cw** 33737 \cs\_new\_protected:Npn \hcoffin\_set:Nw #1

**\hcoffin\_gset:Nw**

**\hcoffin\_gset:cw**

**\hcoffin\_set\_end:**

**\hcoffin\_gset\_end:**

```

33738  {
33739    \__coffin_if_exist:NT #1
33740    {
33741      \hbox_set:Nw #1 \color_ensure_current:
33742      \cs_set_protected:Npn \hcoffin_set_end:
33743      {
33744        \hbox_set_end:
33745        \__coffin_update:N #1
33746      }
33747    }
33748  }
33749 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
33750 \cs_new_protected:Npn \hcoffin_gset:Nw #1
33751  {
33752    \__coffin_if_exist:NT #1
33753    {
33754      \hbox_gset:Nw #1 \color_ensure_current:
33755      \cs_set_protected:Npn \hcoffin_gset_end:
33756      {
33757        \hbox_gset_end:
33758        \__coffin_gupdate:N #1
33759      }
33760    }
33761  }
33762 \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
33763 \cs_new_protected:Npn \hcoffin_set_end: { }
33764 \cs_new_protected:Npn \hcoffin_gset_end: { }

```

(End definition for `\hcoffin_set:Nw` and others. These functions are documented on page 286.)

`\vcoffin_set:Nnw` The same for vertical coffins.

```

33765 \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
33766  {
33767    \__coffin_set_vertical:NnNNNNw #1 {#2} \vbox_set:Nw
33768    \vcoffin_set_end:
33769    \vbox_set_end: \__coffin_update:N
33770  }
33771 \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
33772 \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
33773  {
33774    \__coffin_set_vertical:NnNNNNw #1 {#2} \vbox_gset:Nw
33775    \vcoffin_gset_end:
33776    \vbox_gset_end: \__coffin_gupdate:N
33777  }
33778 \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
33779 \cs_new_protected:Npn \__coffin_set_vertical:NnNNNNw #1#2#3#4#5#6
33780  {
33781    \__coffin_if_exist:NT #1
33782    {
33783      #3 #1
33784      \dim_set:Nn \tex_hsize:D {#2}
33785      \__coffin_set_vertical_aux:
33786      \cs_set_protected:Npn #4
33787      {

```

```

33788 #5
33789 #6 #1
33790 \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
33791 \__coffin_set_pole:Nnx #1 { T }
33792 {
33793     { Opt }
33794     {
33795         \dim_eval:n
33796             { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
33797         }
33798         { 1000pt }
33799         { Opt }
33800     }
33801     \box_clear:N \l__coffin_internal_box
33802 }
33803 }
33804 }
33805 \cs_new_protected:Npn \vcoffin_set_end: { }
33806 \cs_new_protected:Npn \vcoffin_gset_end: { }

(End definition for \vcoffin_set:Nnw and others. These functions are documented on page 286.)
```

### \coffin\_set\_eq:NN

\coffin\_set\_eq:Nc

\coffin\_set\_eq:cN

\coffin\_set\_eq:cc

### \coffin\_gset\_eq:NN

\coffin\_gset\_eq:Nc

\coffin\_gset\_eq:cN

\coffin\_gset\_eq:cc

Setting two coffins equal is just a wrapper around other functions.

```

33807 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
33808 {
33809     \__coffin_if_exist:NT #1
33810     {
33811         \box_set_eq:NN #1 #2
33812         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
33813             { coffin ~ \__coffin_to_value:N #2 ~ corners }
33814             \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
33815             { coffin ~ \__coffin_to_value:N #2 ~ poles }
33816     }
33817 }
33818 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
33819 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
33820 {
33821     \__coffin_if_exist:NT #1
33822     {
33823         \box_gset_eq:NN #1 #2
33824         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
33825             { coffin ~ \__coffin_to_value:N #2 ~ corners }
33826             \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
33827             { coffin ~ \__coffin_to_value:N #2 ~ poles }
33828     }
33829 }
33830 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }
```

(End definition for \coffin\_set\_eq:NN and \coffin\_gset\_eq:NN. These functions are documented on page 285.)

### \c\_empty\_coffin

\l\_\_coffin\_aligned\_coffin

\l\_\_coffin\_aligned\_internal\_coffin

Special coffins: these cannot be set up earlier as they need \coffin\_new:N. The empty coffin is set as a box as the full coffin-setting system needs some material which is not yet available. The empty coffin is created entirely by hand: not everything is in place yet.

```

33831 \coffin_new:N \c_empty_coffin
33832 \coffin_new:N \l__coffin_aligned_coffin
33833 \coffin_new:N \l__coffin_aligned_internal_coffin

(End definition for \c_empty_coffin, \l__coffin_aligned_coffin, and \l__coffin_aligned_internal_coffin. This variable is documented on page 289.)

```

\l\_tmpa\_coffin The usual scratch space.  
\l\_tmpb\_coffin  
\g\_tmpa\_coffin  
\g\_tmpb\_coffin

```

33834 \coffin_new:N \l_tmpa_coffin
33835 \coffin_new:N \l_tmpb_coffin
33836 \coffin_new:N \g_tmpa_coffin
33837 \coffin_new:N \g_tmpb_coffin

```

(End definition for \l\_tmpa\_coffin and others. These variables are documented on page 289.)

## 86.3 Measuring coffins

\coffin\_dp:N Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.  
\coffin\_dp:c  
\coffin\_ht:N  
\coffin\_ht:c  
\coffin\_wd:N  
\coffin\_wd:c

```

33838 \cs_new_eq:NN \coffin_dp:N \box_dp:N
33839 \cs_new_eq:NN \coffin_dp:c \box_dp:c
33840 \cs_new_eq:NN \coffin_ht:N \box_ht:N
33841 \cs_new_eq:NN \coffin_ht:c \box_ht:c
33842 \cs_new_eq:NN \coffin_wd:N \box_wd:N
33843 \cs_new_eq:NN \coffin_wd:c \box_wd:c

```

(End definition for \coffin\_dp:N, \coffin\_ht:N, and \coffin\_wd:N. These functions are documented on page 288.)

## 86.4 Coffins: handle and pole management

\\_\\_coffin\_get\_pole:NnN A simple wrapper around the recovery of a coffin pole, with some error checking and recovery built-in.

```

33844 \cs_new_protected:Npn \_\_coffin_get_pole:NnN #1#2#3
33845 {
33846     \prop_get:cnNF
33847         { coffin ~ \_\_coffin_to_value:N #1 ~ poles } {#2} #3
33848     {
33849         \msg_error:nnxx { coffin } { unknown-pole }
33850             { \exp_not:n {#2} } { \token_to_str:N #1 }
33851             \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
33852     }
33853 }

```

(End definition for \\_\\_coffin\_get\_pole:NnN.)

\\_\\_coffin\_reset\_structure:N \\_\\_coffin\_greset\_structure:N Resetting the structure is a simple copy job.

```

33854 \cs_new_protected:Npn \_\_coffin_reset_structure:N #1
33855 {
33856     \prop_set_eq:cN { coffin ~ \_\_coffin_to_value:N #1 ~ corners }
33857         \c\_coffin_corners_prop
33858     \prop_set_eq:cN { coffin ~ \_\_coffin_to_value:N #1 ~ poles }
33859         \c\_coffin_poles_prop

```

```

33860    }
33861 \cs_new_protected:Npn \__coffin_greset_structure:N #1
33862 {
33863     \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
33864         \c_coffin_corners_prop
33865     \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
33866         \c_coffin_poles_prop
33867 }

```

(End definition for `\__coffin_reset_structure:N` and `\__coffin_greset_structure:N`.)

```

\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
\coffin_set_vertical_pole:Nnn
\coffin_set_vertical_pole:cnn
\coffin_gset_vertical_pole:Nnn
\coffin_gset_vertical_pole:cnn
\__coffin_set_vertical_pole:NnnN
\__coffin_set_pole:Nnn
\__coffin_set_pole:Nnx

```

Setting the pole of a coffin at the user/designer level requires a bit more care. The idea here is to provide a reasonable interface to the system, then to do the setting with full expansion. The three-argument version is used internally to do a direct setting.

```

33868 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
33869 {
33870     \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cnx
33871 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
33872 {
33873     \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cnx
33874 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
33875 \cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
33876 {
33877     \__coffin_if_exist:NT #1
33878     {
33879         #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
33880             {#2}
33881             {
33882                 { Opt } { \dim_eval:n {#3} }
33883                 { 1000pt } { Opt }
33884             }
33885     }
33886 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
33887 {
33888     \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cnx
33889 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
33890 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
33891 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
33892 \cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
33893 {
33894     \__coffin_if_exist:NT #1
33895     {
33896         #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
33897             {#2}
33898             {
33899                 { \dim_eval:n {#3} } { Opt }
33900                 { Opt } { 1000pt }
33901             }
33902     }
33903 }
33904 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
33905 {
33906     \prop_put:cnn { coffin ~ \__coffin_to_value:N #1 ~ poles }
33907         {#2} {#3}

```

```

33908     }
33909 \cs_generate_variant:Nn \__coffin_set_pole:Nnn { Nnx }

```

(End definition for `\coffin_set_horizontal_pole:Nnn` and others. These functions are documented on page 286.)

`\__coffin_update:N` Simple shortcuts.

```

\__coffin_gupdate:N
33910 \cs_new_protected:Npn \__coffin_update:N #1
33911 {
33912     \__coffin_reset_structure:N #1
33913     \__coffin_update_corners:N #1
33914     \__coffin_update_poles:N #1
33915 }
33916 \cs_new_protected:Npn \__coffin_gupdate:N #1
33917 {
33918     \__coffin_greset_structure:N #1
33919     \__coffin_gupdate_corners:N #1
33920     \__coffin_gupdate_poles:N #1
33921 }

```

(End definition for `\__coffin_update:N` and `\__coffin_gupdate:N`.)

`\__coffin_update_corners:N` Updating the corners of a coffin is straight-forward as at this stage there can be no rotation. So the corners of the content are just those of the underlying TeX box.

```

\__coffin_gupdate_corners:N
\__coffin_update_corners:NN
\__coffin_update_corners:NNN
33922 \cs_new_protected:Npn \__coffin_update_corners:N #1
33923 {
33924     \__coffin_update_corners:NN #1 \prop_put:Nnx
33925     \__coffin_update_corners:NN #1 \prop_gput:Nnx
33926 \cs_new_protected:Npn \__coffin_update_corners:NN #1#2
33927 {
33928     \exp_args:Nc \__coffin_update_corners:NNN
33929     { coffin ~ \__coffin_to_value:N #1 ~ corners }
33930     #1 #2
33931 }
33932 \cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
33933 {
33934     #3 #1
33935     { tl }
33936     { { Opt } { \dim_eval:n { \box_ht:N #2 } } }
33937     #3 #1
33938     { tr }
33939     {
33940         { \dim_eval:n { \box_wd:N #2 } }
33941         { \dim_eval:n { \box_ht:N #2 } }
33942     }
33943     #3 #1
33944     { bl }
33945     { { Opt } { \dim_eval:n { -\box_dp:N #2 } } }
33946     #3 #1
33947     { br }
33948     {
33949         { \dim_eval:n { \box_wd:N #2 } }
33950         { \dim_eval:n { -\box_dp:N #2 } }
33951     }
33952 }

```

(End definition for `\_coffin_update_corners:N` and others.)

`\_coffin_update_poles:N`  
`\_coffin_gupdate_poles:N`  
`\_coffin_update_poles:NN`  
`\_coffin_update_poles:NNN`

This function is called when a coffin is set, and updates the poles to reflect the nature of size of the box. Thus this function only alters poles where the default position is dependent on the size of the box. It also does not set poles which are relevant only to vertical coffins.

```
33953 \cs_new_protected:Npn \_coffin_update_poles:N #1
33954   { \_coffin_update_poles:NN #1 \prop_put:Nnx }
33955 \cs_new_protected:Npn \_coffin_gupdate_poles:N #1
33956   { \_coffin_update_poles:NN #1 \prop_gput:Nnx }
33957 \cs_new_protected:Npn \_coffin_update_poles:NN #1#2
33958   {
33959     \exp_args:Nc \_coffin_update_poles:NNN
33960       { coffin ~ \_coffin_to_value:N #1 ~ poles }
33961       #1 #2
33962   }
33963 \cs_new_protected:Npn \_coffin_update_poles:NNN #1#2#3
33964   {
33965     #3 #1 { hc }
33966     {
33967       { \dim_eval:n { 0.5 \box_wd:N #2 } }
33968       { Opt } { Opt } { 1000pt }
33969     }
33970     #3 #1 { r }
33971     {
33972       { \dim_eval:n { \box_wd:N #2 } }
33973       { Opt } { Opt } { 1000pt }
33974     }
33975     #3 #1 { vc }
33976     {
33977       { Opt }
33978       { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
33979       { 1000pt }
33980       { Opt }
33981     }
33982     #3 #1 { t }
33983     {
33984       { Opt }
33985       { \dim_eval:n { \box_ht:N #2 } }
33986       { 1000pt }
33987       { Opt }
33988     }
33989     #3 #1 { b }
33990     {
33991       { Opt }
33992       { \dim_eval:n { -\box_dp:N #2 } }
33993       { 1000pt }
33994       { Opt }
33995     }
33996 }
```

(End definition for `\_coffin_update_poles:N` and others.)

## 86.5 Coffins: calculation of pole intersections

The lead off in finding intersections is to recover the two poles and then hand off to the auxiliary for the actual calculation. There may of course not be an intersection, for which an error trap is needed.

```

33997 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
33998 {
33999   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
34000   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
34001   \bool_set_false:N \l__coffin_error_bool
34002   \exp_last_two_unbraced:Noo
34003   \__coffin_calculate_intersection:nnnnnnnn
34004     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
34005   \bool_if:NT \l__coffin_error_bool
34006   {
34007     \msg_error:nn { coffin } { no-pole-intersection }
34008     \dim_zero:N \l__coffin_x_dim
34009     \dim_zero:N \l__coffin_y_dim
34010   }
34011 }
```

The two poles passed here each have four values (as dimensions),  $(a, b, c, d)$  and  $(a', b', c', d')$ . These are arguments 1–4 and 5–8, respectively. In both cases  $a$  and  $b$  are the co-ordinates of a point on the pole and  $c$  and  $d$  define the direction of the pole. Finding the intersection depends on the directions of the poles, which are given by  $d/c$  and  $d'/c'$ . However, if one of the poles is either horizontal or vertical then one or more of  $c$ ,  $d$ ,  $c'$  and  $d'$  are zero and a special case is needed.

```

34012 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnnn
34013   #1#2#3#4#5#6#7#8
34014 {
34015   \dim_compare:nNnTF {#3} = \c_zero_dim
```

The case where the first pole is vertical. So the  $x$ -component of the interaction is at  $a$ . There is then a test on the second pole: if it is also vertical then there is an error.

```

34016 {
34017   \dim_set:Nn \l__coffin_x_dim {#1}
34018   \dim_compare:nNnTF {#7} = \c_zero_dim
34019   { \bool_set_true:N \l__coffin_error_bool }
```

The second pole may still be horizontal, in which case the  $y$ -component of the intersection is  $b'$ . If not,

$$y = \frac{d'}{c'} (a - a') + b'$$

with the  $x$ -component already known to be #1.

```

34020 {
34021   \dim_set:Nn \l__coffin_y_dim
34022   {
34023     \dim_compare:nNnTF {#8} = \c_zero_dim
34024     {#6}
34025     {
34026       \fp_to_dim:n
34027       {
34028         ( \dim_to_fp:n {#8} / \dim_to_fp:n {#7} )
34029         * ( \dim_to_fp:n {#1} - \dim_to_fp:n {#5} )
```

```

34030           + \dim_to_fp:n {#6}
34031       }
34032   }
34033   }
34034   }
34035 }

```

If the first pole is not vertical then it may be horizontal. If so, then the procedure is essentially the same as that already done but with the  $x$ - and  $y$ -components interchanged.

```

34036 {
34037   \dim_compare:nNnTF {#4} = \c_zero_dim
34038   {
34039     \dim_set:Nn \l__coffin_y_dim {#2}
34040     \dim_compare:nNnTF {#8} = { \c_zero_dim }
34041     { \bool_set_true:N \l__coffin_error_bool }
34042   }

```

Now we deal with the case where the second pole may be vertical, or if not we have

$$x = \frac{c'}{d'} (b - b') + a'$$

which is again handled by the same auxiliary.

```

34043   \dim_set:Nn \l__coffin_x_dim
34044   {
34045     \dim_compare:nNnTF {#7} = \c_zero_dim
34046     {#5}
34047     {
34048       \fp_to_dim:n
34049       {
34050         ( \dim_to_fp:n {#7} / \dim_to_fp:n {#8} )
34051         * ( \dim_to_fp:n {#4} - \dim_to_fp:n {#6} )
34052         + \dim_to_fp:n {#5}
34053       }
34054     }
34055   }
34056 }
34057

```

The first pole is neither horizontal nor vertical. To avoid even more complexity, we now work out both slopes and pass to an auxiliary.

```

34058   {
34059     \use:x
34060     {
34061       \__coffin_calculate_intersection:nnnnnn
34062       { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
34063       { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
34064     }
34065     {#1} {#2} {#5} {#6}
34066   }
34067 }
34068

```

Assuming the two poles are not parallel, then the intersection point is found in two steps. First we find the  $x$ -value with

$$x = \frac{sa - s'a' - b + b'}{s - s'}$$

and then finding the  $y$ -value with

$$y = s(x - a) + b$$

```

34069 \cs_set_protected:Npn \__coffin_calculate_intersection:nnnnnnn #1#2#3#4#5#6
34070 {
34071     \fp_compare:nNnTF {#1} = {#2}
34072         { \bool_set_true:N \l__coffin_error_bool }
34073     {
34074         \dim_set:Nn \l__coffin_x_dim
34075             {
34076                 \fp_to_dim:n
34077                     {
34078                         (
34079                             #1 * \dim_to_fp:n {#3}
34080                             - #2 * \dim_to_fp:n {#5}
34081                             - \dim_to_fp:n {#4}
34082                             + \dim_to_fp:n {#6}
34083                         )
34084                         /
34085                         ( #1 - #2 )
34086                     }
34087                 }
34088             \dim_set:Nn \l__coffin_y_dim
34089                 {
34090                     \fp_to_dim:n
34091                         {
34092                             #1 * ( \l__coffin_x_dim - \dim_to_fp:n {#3} )
34093                             + \dim_to_fp:n {#4}
34094                         }
34095                 }
34096             }
34097         }

```

(End definition for `\__coffin_calculate_intersection:Nnn`, `\__coffin_calculate_intersection:nnnnnnnn`, and `\__coffin_calculate_intersection:nnnnnnn`.)

## 86.6 Affine transformations

`\l__coffin_sin_fp` Used for rotations to get the sine and cosine values.

`\l__coffin_cos_fp`

```

34098 \fp_new:N \l__coffin_sin_fp
34099 \fp_new:N \l__coffin_cos_fp

```

(End definition for `\l__coffin_sin_fp` and `\l__coffin_cos_fp`.)

`\l__coffin_bounding_prop` A property list for the bounding box of a coffin. This is only needed during the rotation, so there is just the one.

```

34100 \prop_new:N \l__coffin_bounding_prop

```

(End definition for `\l__coffin_bounding_prop`.)

`\l__coffin_corners_prop` Used to avoid needing to track scope for intermediate steps.

`\l__coffin_poles_prop`

```

34101 \prop_new:N \l__coffin_corners_prop
34102 \prop_new:N \l__coffin_poles_prop

```

(End definition for `\l_coffin_corners_prop` and `\l_coffin_poles_prop`.)

`\l_coffin_bounding_shift_dim` The shift of the bounding box of a coffin from the real content.

34103 `\dim_new:N \l_coffin_bounding_shift_dim`

(End definition for `\l_coffin_bounding_shift_dim`.)

`\l_coffin_left_corner_dim` These are used to hold maxima for the various corner values: these thus define the minimum size of the bounding box after rotation.

34104 `\dim_new:N \l_coffin_left_corner_dim`

34105 `\dim_new:N \l_coffin_right_corner_dim`

34106 `\dim_new:N \l_coffin_bottom_corner_dim`

34107 `\dim_new:N \l_coffin_top_corner_dim`

(End definition for `\l_coffin_left_corner_dim` and others.)

`\coffin_rotate:Nn`

`\coffin_rotate:cn`

`\coffin_grotate:Nn`

`\coffin_grotate:cn`

`\__coffin_rotate:NnNNN`

34108 `\cs_new_protected:Npn \coffin_rotate:Nn #1#2`

34109 { `\__coffin_rotate:NnNNN #1 {#2} \box_rotate:Nn \prop_set_eq:cN \hbox_set:Nn }`

34110 `\cs_generate_variant:Nn \coffin_rotate:Nn { c }`

34111 `\cs_new_protected:Npn \coffin_grotate:Nn #1#2`

34112 { `\__coffin_rotate:NnNNN #1 {#2} \box_grotate:Nn \prop_gset_eq:cN \hbox_gset:Nn }`

34113 `\cs_generate_variant:Nn \coffin_grotate:Nn { c }`

34114 `\cs_new_protected:Npn \__coffin_rotate:NnNNN #1#2#3#4#5`

34115 {

34116 `\fp_set:Nn \l_coffin_sin_fp { sind ( #2 ) }`

34117 `\fp_set:Nn \l_coffin_cos_fp { cosd ( #2 ) }`

Use a local copy of the property lists to avoid needing to pass the name and scope around.

34118 `\prop_set_eq:Nc \l_coffin_corners_prop`

34119 { `coffin ~ \__coffin_to_value:N #1 ~ corners }`

34120 `\prop_set_eq:Nc \l_coffin_poles_prop`

34121 { `coffin ~ \__coffin_to_value:N #1 ~ poles }`

The corners and poles of the coffin can now be rotated around the origin. This is best achieved using mapping functions.

34122 `\prop_map_inline:Nn \l_coffin_corners_prop`

34123 { `\__coffin_rotate_corner:Nnnn #1 {##1} ##2 }`

34124 `\prop_map_inline:Nn \l_coffin_poles_prop`

34125 { `\__coffin_rotate_pole:Nnnnnn #1 {##1} ##2 }`

The bounding box of the coffin needs to be rotated, and to do this the corners have to be found first. They are then rotated in the same way as the corners of the coffin material itself.

34126 `\__coffin_set_bounding:N #1`

34127 `\prop_map_inline:Nn \l_coffin_bounding_prop`

34128 { `\__coffin_rotate_bounding:nnn {##1} ##2 }`

At this stage, there needs to be a calculation to find where the corners of the content and the box itself will end up.

34129 `\__coffin_find_corner_maxima:N #1`

34130 `\__coffin_find_bounding_shift:`

34131 `#3 #1 {#2}`

The correction of the box position itself takes place here. The idea is that the bounding box for a coffin is tight up to the content, and has the reference point at the bottom-left. The  $x$ -direction is handled by moving the content by the difference in the positions of the bounding box and the content left edge. The  $y$ -direction is dealt with by moving the box down by any depth it has acquired. The internal box is used here to allow for the next step.

```

34132      \hbox_set:Nn \l_coffin_internal_box
34133      {
34134          \__kernel_kern:n
34135          { \l_coffin_bounding_shift_dim - \l_coffin_left_corner_dim }
34136          \box_move_down:nn { \l_coffin_bottom_corner_dim }
34137          { \box_use:N #1 }
34138      }

```

If there have been any previous rotations then the size of the bounding box will be bigger than the contents. This can be corrected easily by setting the size of the box to the height and width of the content. As this operation requires setting box dimensions and these transcend grouping, the safe way to do this is to use the internal box and to reset the result into the target box.

```

34139      \box_set_ht:Nn \l_coffin_internal_box
34140          { \l_coffin_top_corner_dim - \l_coffin_bottom_corner_dim }
34141      \box_set_dp:Nn \l_coffin_internal_box { Opt }
34142      \box_set_wd:Nn \l_coffin_internal_box
34143          { \l_coffin_right_corner_dim - \l_coffin_left_corner_dim }
34144          #5 #1 { \box_use_drop:N \l_coffin_internal_box }

```

The final task is to move the poles and corners such that they are back in alignment with the box reference point.

```

34145      \prop_map_inline:Nn \l_coffin_corners_prop
34146          { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
34147      \prop_map_inline:Nn \l_coffin_poles_prop
34148          { \__coffin_shift_pole:Nnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

34149      #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
34150          \l_coffin_corners_prop
34151      #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
34152          \l_coffin_poles_prop
34153      }

```

(End definition for `\coffin_rotate:Nn`, `\coffin_grotate:Nn`, and `\__coffin_rotate:NnNNN`. These functions are documented on page 287.)

## `\__coffin_set_bounding:N`

The bounding box corners for a coffin are easy enough to find: this is the same code as for the corners of the material itself, but using a dedicated property list.

```

34154 \cs_new_protected:Npn \__coffin_set_bounding:N #1
34155  {
34156      \prop_put:Nnx \l_coffin_bounding_prop { tl }
34157          { { Opt } { \dim_eval:n { \box_ht:N #1 } } } }
34158      \prop_put:Nnx \l_coffin_bounding_prop { tr }
34159          {
34160              { \dim_eval:n { \box_wd:N #1 } }
34161              { \dim_eval:n { \box_ht:N #1 } }
34162          }

```

```

34163   \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
34164   \prop_put:Nnx \l__coffin_bounding_prop { bl }
34165     { { Opt } { \dim_use:N \l__coffin_internal_dim } }
34166   \prop_put:Nnx \l__coffin_bounding_prop { br }
34167     {
34168       { \dim_eval:n { \box_wd:N #1 } }
34169       { \dim_use:N \l__coffin_internal_dim }
34170     }
34171   }

```

(End definition for `\__coffin_set_bounding:N`.)

`\__coffin_rotate_bounding:nnn`  
`\__coffin_rotate_corner:Nnnn`

Rotating the position of the corner of the coffin is just a case of treating this as a vector from the reference point. The same treatment is used for the corners of the material itself and the bounding box.

```

34172 \cs_new_protected:Npn \__coffin_rotate_bounding:nnn #1#2#3
34173   {
34174     \__coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
34175     \prop_put:Nnx \l__coffin_bounding_prop {#1}
34176       { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
34177   }
34178 \cs_new_protected:Npn \__coffin_rotate_corner:Nnnn #1#2#3#4
34179   {
34180     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
34181     \prop_put:Nnx \l__coffin_corners_prop {#2}
34182       { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
34183   }

```

(End definition for `\__coffin_rotate_bounding:nnn` and `\__coffin_rotate_corner:Nnnn`.)

`\__coffin_rotate_pole:Nnnnnn`

Rotating a single pole simply means shifting the co-ordinate of the pole and its direction. The rotation here is about the bottom-left corner of the coffin.

```

34184 \cs_new_protected:Npn \__coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
34185   {
34186     \__coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
34187     \__coffin_rotate_vector:nnNN {#5} {#6}
34188       \l__coffin_x_prime_dim \l__coffin_y_prime_dim
34189     \prop_put:Nnx \l__coffin_poles_prop {#2}
34190       {
34191         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
34192         { \dim_use:N \l__coffin_x_prime_dim }
34193         { \dim_use:N \l__coffin_y_prime_dim }
34194       }
34195   }

```

(End definition for `\__coffin_rotate_pole:Nnnnnn`.)

`\__coffin_rotate_vector:nnNN`

A rotation function, which needs only an input vector (as dimensions) and an output space. The values `\l__coffin_cos_fp` and `\l__coffin_sin_fp` should previously have been set up correctly. Working this way means that the floating point work is kept to a minimum: for any given rotation the sin and cosine values do no change, after all.

```

34196 \cs_new_protected:Npn \__coffin_rotate_vector:nnNN #1#2#3#4
34197   {
34198     \dim_set:Nn #3

```

```

34199 {
34200   \fp_to_dim:n
34201   {
34202     \dim_to_fp:n {#1} * \l__coffin_cos_fp
34203     - \dim_to_fp:n {#2} * \l__coffin_sin_fp
34204   }
34205 }
34206 \dim_set:Nn #4
34207 {
34208   \fp_to_dim:n
34209   {
34210     \dim_to_fp:n {#1} * \l__coffin_sin_fp
34211     + \dim_to_fp:n {#2} * \l__coffin_cos_fp
34212   }
34213 }
34214 }

```

(End definition for `\__coffin_rotate_vector:nnNN`.)

`\__coffin_find_corner_maxima:N`  
`\__coffin_find_corner_maxima_aux:nn`

The idea here is to find the extremities of the content of the coffin. This is done by looking for the smallest values for the bottom and left corners, and the largest values for the top and right corners. The values start at the maximum dimensions so that the case where all are positive or all are negative works out correctly.

```

34215 \cs_new_protected:Npn \__coffin_find_corner_maxima:N #1
34216 {
34217   \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
34218   \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
34219   \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
34220   \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
34221   \prop_map_inline:Nn \l__coffin_corners_prop
34222   { \__coffin_find_corner_maxima_aux:nn ##2 }
34223 }
34224 \cs_new_protected:Npn \__coffin_find_corner_maxima_aux:nn #1#2
34225 {
34226   \dim_set:Nn \l__coffin_left_corner_dim
34227   { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
34228   \dim_set:Nn \l__coffin_right_corner_dim
34229   { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
34230   \dim_set:Nn \l__coffin_bottom_corner_dim
34231   { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
34232   \dim_set:Nn \l__coffin_top_corner_dim
34233   { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
34234 }

```

(End definition for `\__coffin_find_corner_maxima:N` and `\__coffin_find_corner_maxima_aux:nn`.)

`\__coffin_find_bounding_shift:`  
`\__coffin_find_bounding_shift_aux:nn`

The approach to finding the shift for the bounding box is similar to that for the corners. However, there is only one value needed here and a fixed input property list, so things are a bit clearer.

```

34235 \cs_new_protected:Npn \__coffin_find_bounding_shift:
34236 {
34237   \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
34238   \prop_map_inline:Nn \l__coffin_bounding_prop
34239   { \__coffin_find_bounding_shift_aux:nn ##2 }

```

```

34240     }
34241 \cs_new_protected:Npn \__coffin_find_bounding_shift_aux:nn #1#2
34242 {
34243     \dim_set:Nn \l__coffin_bounding_shift_dim
34244     { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
34245 }

```

(End definition for `\__coffin_find_bounding_shift:` and `\__coffin_find_bounding_shift_aux:nn`.)

`\__coffin_shift_corner:Nnnn`  
`\__coffin_shift_pole:Nnnnnn`

Shifting the corners and poles of a coffin means subtracting the appropriate values from the  $x$ - and  $y$ -components. For the poles, this means that the direction vector is unchanged.

```

34246 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
34247 {
34248     \prop_put:Nnx \l__coffin_corners_prop {#2}
34249     {
34250         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
34251         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
34252     }
34253 }
34254 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
34255 {
34256     \prop_put:Nnx \l__coffin_poles_prop {#2}
34257     {
34258         { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
34259         { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
34260         {#5} {#6}
34261     }
34262 }

```

(End definition for `\__coffin_shift_corner:Nnnn` and `\__coffin_shift_pole:Nnnnnn`.)

`\l__coffin_scale_x_fp`  
`\l__coffin_scale_y_fp`

Storage for the scaling factors in  $x$  and  $y$ , respectively.

```

34263 \fp_new:N \l__coffin_scale_x_fp
34264 \fp_new:N \l__coffin_scale_y_fp

```

(End definition for `\l__coffin_scale_x_fp` and `\l__coffin_scale_y_fp`.)

`\l__coffin_scaled_total_height_dim`  
`\l__coffin_scaled_width_dim`

When scaling, the values given have to be turned into absolute values.

```

34265 \dim_new:N \l__coffin_scaled_total_height_dim
34266 \dim_new:N \l__coffin_scaled_width_dim

```

(End definition for `\l__coffin_scaled_total_height_dim` and `\l__coffin_scaled_width_dim`.)

```

\coffin_resize:Nnn
\coffin_resize:cnn
\coffin_gresize:Nnn
\coffin_gresize:cnn
\__coffin_resize:NnnNN

```

Resizing a coffin begins by setting up the user-friendly names for the dimensions of the coffin box. The new sizes are then turned into scale factor. This is the same operation as takes place for the underlying box, but that operation is grouped and so the same calculation is done here.

```

34267 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
34268 {
34269     \__coffin_resize:NnnNN #1 {#2} {#3}
34270     \box_resize_to_wd_and_ht_plus_dp:Nnn
34271     \prop_set_eq:cN
34272 }

```

```

34273 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
34274 \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
34275 {
34276     \__coffin_resize:NnnNN #1 {#2} {#3}
34277     \box_gresize_to_wd_and_ht_plus_dp:Nnn
34278     \prop_gset_eq:cN
34279 }
34280 \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
34281 \cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
34282 {
34283     \fp_set:Nn \l__coffin_scale_x_fp
34284     { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
34285     \fp_set:Nn \l__coffin_scale_y_fp
34286     {
34287         \dim_to_fp:n {#3}
34288         / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
34289     }
34290     #4 #1 {#2} {#3}
34291     \__coffin_resize_common:NnnN #1 {#2} {#3} #5
34292 }

```

(End definition for `\coffin_resize:Nnn`, `\coffin_gresize:Nnn`, and `\__coffin_resize:NnnNN`. These functions are documented on page 287.)

`\__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

34293 \cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
34294 {
34295     \prop_set_eq:Nc \l__coffin_corners_prop
34296     { coffin ~ \__coffin_to_value:N #1 ~ corners }
34297     \prop_set_eq:Nc \l__coffin_poles_prop
34298     { coffin ~ \__coffin_to_value:N #1 ~ poles }
34299     \prop_map_inline:Nn \l__coffin_corners_prop
34300     { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
34301     \prop_map_inline:Nn \l__coffin_poles_prop
34302     { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }

```

Negative  $x$ -scaling values place the poles in the wrong location: this is corrected here.

```

34303 \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
34304 {
34305     \prop_map_inline:Nn \l__coffin_corners_prop
34306     { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
34307     \prop_map_inline:Nn \l__coffin_poles_prop
34308     { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
34309 }
34310 #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
34311     \l__coffin_corners_prop
34312 #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
34313     \l__coffin_poles_prop
34314 }

```

(End definition for `\__coffin_resize_common:NnnN`.)

`\coffin_scale:Nnn`  
`\coffin_scale:cnn`  
`\coffin_gscale:Nnn`  
`\coffin_gscale:cnn`  
`\coffin_scale:NnnNN` For scaling, the opposite calculation is done to find the new dimensions for the coffin. Only the total height is needed, as this is the shift required for corners and poles. The

scaling is done the TeX way as this works properly with floating point values without needing to use the fp module.

```

34315 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
34316   { \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
34317 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
34318 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
34319   { \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
34320 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
34321 \cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
34322   {
34323     \fp_set:Nn \l__coffin_scale_x_fp {#2}
34324     \fp_set:Nn \l__coffin_scale_y_fp {#3}
34325     #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
34326     \dim_set:Nn \l__coffin_internal_dim
34327       { \coffin_ht:N #1 + \coffin_dp:N #1 }
34328     \dim_set:Nn \l__coffin_scaled_total_height_dim
34329       { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
34330     \dim_set:Nn \l__coffin_scaled_width_dim
34331       { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
34332     \__coffin_resize_common:NnnN #1
34333       { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
34334     #5
34335   }

```

(End definition for \coffin\_scale:Nnn, \coffin\_gscale:Nnn, and \coffin\_scale:NnnNN. These functions are documented on page 287.)

\\_\_coffin\_scale\_vector:nnNN

This function scales a vector from the origin using the pre-set scale factors in  $x$  and  $y$ . This is a much less complex operation than rotation, and as a result the code is a lot clearer.

```

34336 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
34337   {
34338     \dim_set:Nn #3
34339       { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
34340     \dim_set:Nn #4
34341       { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
34342   }

```

(End definition for \\_\_coffin\_scale\_vector:nnNN.)

\\_\_coffin\_scale\_corner:Nnnn  
\\_\_coffin\_scale\_pole:Nnnnnn

Scaling both corners and poles is a simple calculation using the preceding vector scaling.

```

34343 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
34344   {
34345     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
34346     \prop_put:Nnx \l__coffin_corners_prop {#2}
34347       { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
34348   }
34349 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
34350   {
34351     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
34352     \prop_put:Nnx \l__coffin_poles_prop {#2}
34353       {
34354         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
34355         {#5} {#6}

```

```

34356     }
34357 }
```

(End definition for `\_coffin_scale_corner:Nnnn` and `\_coffin_scale_pole:Nnnnnn`.)

`\_coffin_x_shift_corner:Nnnn`  
`\_coffin_x_shift_pole:Nnnnnn`

These functions correct for the *x* displacement that takes place with a negative horizontal scaling.

```

34358 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
34359   {
34360     \prop_put:Nnx \l_coffin_corners_prop {#2}
34361       {
34362         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
34363       }
34364   }
34365 \cs_new_protected:Npn \_coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
34366   {
34367     \prop_put:Nnx \l_coffin_poles_prop {#2}
34368       {
34369         { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
34370         {#5} {#6}
34371       }
34372 }
```

(End definition for `\_coffin_x_shift_corner:Nnnn` and `\_coffin_x_shift_pole:Nnnnnn`.)

## 86.7 Aligning and typesetting of coffins

`\coffin_join:NnnNnnn`  
`\coffin_join:cnnNnnn`  
`\coffin_join:Nnncnnnn`  
`\coffin_join:cnnccnnnn`  
`\coffin_gjoin:NnnNnnnn`  
`\coffin_gjoin:cnnNnnnn`  
`\coffin_gjoin:Nnnccnnnn`  
`\coffin_gjoin:cnnccnnnn`  
`\_coffin_join:NnnNnnnnN`

This command joins two coffins, using a horizontal and vertical pole from each coffin and making an offset between the two. The result is stored as the as a third coffin, which has all of its handles reset to standard values. First, the more basic alignment function is used to get things started.

```

34373 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
34374   {
34375     \_coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
34376       \coffin_set_eq:NN
34377   }
34378 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnnc , cnnc }
34379 \cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
34380   {
34381     \_coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
34382       \coffin_gset_eq:NN
34383   }
34384 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
34385 \cs_new_protected:Npn \_coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
34386   {
34387     \_coffin_align:NnnNnnnnN
34388     #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l_coffin_aligned_coffin
```

Correct the placement of the reference point. If the *x*-offset is negative then the reference point of the second box is to the left of that of the first, which is corrected using a kern. On the right side the first box might stick out, which would show up if it is wider than the sum of the *x*-offset and the width of the second box. So a second kern may be needed.

```

34389     \hbox_set:Nn \l_coffin_aligned_coffin
```

```

34390 {
34391   \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
34392     { \__kernel_kern:n { -\l__coffin_offset_x_dim } }
34393   \hbox_unpack:N \l__coffin_aligned_coffin
34394   \dim_set:Nn \l__coffin_internal_dim
34395     { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
34396   \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
34397     { \__kernel_kern:n { -\l__coffin_internal_dim } }
34398 }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

34399   \__coffin_reset_structure:N \l__coffin_aligned_coffin
34400   \prop_clear:c
34401   {
34402     \coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
34403     \c_space_tl corners
34404   }
34405   \__coffin_update_poles:N \l__coffin_aligned_coffin

```

The structures of the parent coffins are now transferred to the new coffin, which requires that the appropriate offsets are applied. That then depends on whether any shift was needed.

```

34406   \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
34407   {
34408     \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
34409     \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
34410     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
34411     \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
34412   }
34413   {
34414     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
34415     \__coffin_offset_poles:Nnn #4
34416       { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
34417     \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
34418     \__coffin_offset_corners:Nnn #4
34419       { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
34420   }
34421   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
34422   #9 #1 \l__coffin_aligned_coffin
34423 }

```

(End definition for `\coffin_join:NnnNnnnn`, `\coffin_gjoin:NnnNnnnn`, and `\__coffin_join:NnnNnnnn`. These functions are documented on page [288](#).)

`\coffin_attach:NnnNnnnn`  
`\coffin_attach:cnnNnnnn`  
`\coffin_attach:Nnnccnnnn`  
`\coffin_attach:cnnccnnnn`

`\coffin_gattach:NnnNnnnn`  
`\coffin_gattach:cnnNnnnn`  
`\coffin_gattach:Nnnccnnnn`  
`\coffin_gattach:cnnccnnnn`  
`\__coffin_attach:NnnNnnnnN`  
`\__coffin_attach_mark:NnnNnnnn`

```

34424 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
34425   {
34426     \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
34427     \coffin_set_eq:NN
34428   }
34429 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }

```

```

34430 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
34431 {
34432     \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
34433     \coffin_gset_eq:NN
34434 }
34435 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
34436 \cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
34437 {
34438     \__coffin_align:NnnNnnnnN
34439         #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
34440     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
34441     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
34442     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
34443     \__coffin_reset_structure:N \l__coffin_aligned_coffin
34444     \prop_set_eq:cc
34445     {
34446         coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
34447         \c_space_tl corners
34448     }
34449     { coffin ~ \__coffin_to_value:N #1 ~ corners }
34450     \__coffin_update_poles:N \l__coffin_aligned_coffin
34451     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
34452     \__coffin_offset_poles:Nnn #4
34453         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
34454     \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
34455     #9 #1 \l__coffin_aligned_coffin
34456 }
34457 \cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
34458 {
34459     \__coffin_align:NnnNnnnnN
34460         #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
34461     \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
34462     \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
34463     \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
34464     \box_set_eq:NN #1 \l__coffin_aligned_coffin
34465 }

```

(End definition for `\coffin_attach:NnnNnnnn` and others. These functions are documented on page 287.)

`\__coffin_align:NnnNnnnnN`

The internal function aligns the two coffins into a third one, but performs no corrections on the resulting coffin poles. The process begins by finding the points of intersection for the poles for each of the input coffins. Those for the first coffin are worked out after those for the second coffin, as this allows the ‘primed’ storage area to be used for the second coffin. The ‘real’ box offsets are then calculated, before using these to re-box the input coffins. The default poles are then set up, but the final result depends on how the bounding box is being handled.

```

34466 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
34467 {
34468     \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
34469     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
34470     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
34471     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
34472     \dim_set:Nn \l__coffin_offset_x_dim

```

```

34473      { \l_coffin_x_dim - \l_coffin_x_prime_dim + #7 }
34474      \dim_set:Nn \l_coffin_offset_y_dim
34475          { \l_coffin_y_dim - \l_coffin_y_prime_dim + #8 }
34476      \hbox_set:Nn \l_coffin_aligned_internal_coffin
34477          {
34478              \box_use:N #1
34479                  \__kernel_kern:n { -\box_wd:N #1 }
34480                  \__kernel_kern:n { \l_coffin_offset_x_dim }
34481                  \box_move_up:nn { \l_coffin_offset_y_dim } { \box_use:N #4 }
34482          }
34483      \coffin_set_eq:NN #9 \l_coffin_aligned_internal_coffin
34484  }

```

(End definition for `\__coffin_align:NnnNnnnnN`.)

```
\__coffin_offset_poles:Nnn
\__coffin_offset_pole:Nnnnnn
```

Transferring structures from one coffin to another requires that the positions are updated by the offset between the two coffins. This is done by mapping over the property list of the source coffins, moving as appropriate and saving to the new coffin data structures. The test for a - means that the structures from the parent coffins are uniquely labelled and do not depend on the order of alignment. The pay off for this is that - should not be used in coffin pole or handle names, and that multiple alignments do not result in a whole set of values.

```

34485 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
34486  {
34487      \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
34488          { \__coffin_offset_pole:Nnnnnn #1 {##1} ##2 {#2} {#3} }
34489  }
34490 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
34491  {
34492      \dim_set:Nn \l_coffin_x_dim { #3 + #7 }
34493      \dim_set:Nn \l_coffin_y_dim { #4 + #8 }
34494      \tl_if_in:nnTF {#2} { - }
34495          { \tl_set:Nn \l_coffin_internal_tl { {#2} } }
34496          { \tl_set:Nn \l_coffin_internal_tl { { #1 - #2 } } }
34497      \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l_coffin_aligned_coffin
34498          { \l_coffin_internal_tl }
34499          {
34500              { \dim_use:N \l_coffin_x_dim } { \dim_use:N \l_coffin_y_dim }
34501              {##5} {#6}
34502          }
34503  }

```

(End definition for `\__coffin_offset_poles:Nnn` and `\__coffin_offset_pole:Nnnnnnn`.)

```
\__coffin_offset_corners:Nnn
\__coffin_offset_corner:Nnnnn
```

Saving the offset corners of a coffin is very similar, except that there is no need to worry about naming: every corner can be saved here as order is unimportant.

```

34504 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
34505  {
34506      \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ corners }
34507          { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
34508  }
34509 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
34510  {
34511      \prop_put:cnx

```

```

34512 {
34513   coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
34514   \c_space_tl corners
34515 }
34516 { #1 - #2 }
34517 {
34518   { \dim_eval:n { #3 + #5 } }
34519   { \dim_eval:n { #4 + #6 } }
34520 }
34521 }

```

(End definition for `\__coffin_offset_corners:Nnn` and `\__coffin_offset_corner:Nnnnn`.)

The T and B poles need to be recalculated after alignment. These functions find the larger absolute value for the poles, but this is of course only logical when the poles are horizontal.

```

34522 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
34523 {
34524   \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_tl
34525   \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_tl
34526   \exp_last_two_unbraced:Nno \__coffin_update_T:nnnnnnnnN
34527     \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
34528   \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_tl
34529   \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_tl
34530   \exp_last_two_unbraced:Nno \__coffin_update_B:nnnnnnnnN
34531     \l__coffin_pole_a_tl \l__coffin_pole_b_tl #3
34532 }
34533 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
34534 {
34535   \dim_compare:nNnTF {#2} < {#6}
34536   {
34537     \__coffin_set_pole:Nnx #9 { T }
34538       { { Opt } {#6} { 1000pt } { Opt } }
34539   }
34540   {
34541     \__coffin_set_pole:Nnx #9 { T }
34542       { { Opt } {#2} { 1000pt } { Opt } }
34543   }
34544 }
34545 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
34546 {
34547   \dim_compare:nNnTF {#2} < {#6}
34548   {
34549     \__coffin_set_pole:Nnx #9 { B }
34550       { { Opt } {#2} { 1000pt } { Opt } }
34551   }
34552   {
34553     \__coffin_set_pole:Nnx #9 { B }
34554       { { Opt } {#6} { 1000pt } { Opt } }
34555   }
34556 }

```

(End definition for `\__coffin_update_vertical_poles:NNN`, `\__coffin_update_T:nnnnnnnnN`, and `\__coffin_update_B:nnnnnnnnN`.)

```
\c__coffin_empty_coffin An empty-but-horizontal coffin.
34557 \coffin_new:N \c__coffin_empty_coffin
34558 \tex_setbox:D \c__coffin_empty_coffin = \tex_hbox:D { }

(End definition for \c__coffin_empty_coffin.)
```

**\coffin\_typeset:Nnnnn**  
**\coffin\_typeset:cnnnn**

Typesetting a coffin means aligning it with the current position, which is done using a coffin with no content at all. As well as aligning to the empty coffin, there is also a need to leave vertical mode, if necessary.

```
34559 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
34560 {
34561     \mode_leave_vertical:
34562     \__coffin_align:NnnNnnnnN \c__coffin_empty_coffin { H } { 1 }
34563         #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
34564     \box_use_drop:N \l__coffin_aligned_coffin
34565 }
34566 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }
```

(End definition for \coffin\_typeset:Nnnnn. This function is documented on page 288.)

## 86.8 Coffin diagnostics

**\l\_\_coffin\_display\_coffin** Used for printing coffins with data structures attached.

```
34567 \coffin_new:N \l__coffin_display_coffin
34568 \coffin_new:N \l__coffin_display_coord_coffin
34569 \coffin_new:N \l__coffin_display_pole_coffin
```

(End definition for \l\_\_coffin\_display\_coffin, \l\_\_coffin\_display\_coord\_coffin, and \l\_\_coffin\_display\_pole\_coffin.)

**\l\_\_coffin\_display\_handles\_prop** This property list is used to print coffin handles at suitable positions. The offsets are expressed as multiples of the basic offset value, which therefore acts as a scale-factor.

```
34570 \prop_new:N \l__coffin_display_handles_prop
34571 \prop_put:Nnn \l__coffin_display_handles_prop { tl }
34572     { { b } { r } { -1 } { 1 } }
34573 \prop_put:Nnn \l__coffin_display_handles_prop { thc }
34574     { { b } { hc } { 0 } { 1 } }
34575 \prop_put:Nnn \l__coffin_display_handles_prop { tr }
34576     { { b } { 1 } { 1 } { 1 } }
34577 \prop_put:Nnn \l__coffin_display_handles_prop { vcl }
34578     { { vc } { r } { -1 } { 0 } }
34579 \prop_put:Nnn \l__coffin_display_handles_prop { vchc }
34580     { { vc } { hc } { 0 } { 0 } }
34581 \prop_put:Nnn \l__coffin_display_handles_prop { vcr }
34582     { { vc } { 1 } { 1 } { 0 } }
34583 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
34584     { { t } { r } { -1 } { -1 } }
34585 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
34586     { { t } { hc } { 0 } { -1 } }
34587 \prop_put:Nnn \l__coffin_display_handles_prop { br }
34588     { { t } { 1 } { 1 } { -1 } }
34589 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
34590     { { t } { r } { -1 } { -1 } }
```

```

34591 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
34592   { { t } { hc } { 0 } { -1 } }
34593 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
34594   { { t } { 1 } { 1 } { -1 } }
34595 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
34596   { { vc } { r } { -1 } { 1 } }
34597 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
34598   { { vc } { hc } { 0 } { 1 } }
34599 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
34600   { { vc } { 1 } { 1 } { 1 } }
34601 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
34602   { { b } { r } { -1 } { -1 } }
34603 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
34604   { { b } { hc } { 0 } { -1 } }
34605 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
34606   { { b } { 1 } { 1 } { -1 } }

```

(End definition for `\l__coffin_display_handles_prop`.)

`\l_coffin_display_offset_dim` The standard offset for the label from the handle position when displaying handles.

```

34607 \dim_new:N \l__coffin_display_offset_dim
34608 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End definition for `\l__coffin_display_offset_dim`.)

`\l__coffin_display_x_dim` `\l__coffin_display_y_dim` As the intersections of poles have to be calculated to find which ones to print, there is a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

34609 \dim_new:N \l__coffin_display_x_dim
34610 \dim_new:N \l__coffin_display_y_dim

```

(End definition for `\l__coffin_display_x_dim` and `\l__coffin_display_y_dim`.)

`\l__coffin_display_poles_prop` A property list for printing poles: various things need to be deleted from this to get a “nice” output.

```
34611 \prop_new:N \l__coffin_display_poles_prop
```

(End definition for `\l__coffin_display_poles_prop`.)

`\l__coffin_display_font_t1` Stores the settings used to print coffin data: this keeps things flexible.

```

34612 \tl_new:N \l__coffin_display_font_t1
34613 \bool_lazy_and:nnT
34614   { \cs_if_exist_p:N \fmtname }
34615   { \str_if_eq_p:Vn \fmtname { LaTeXe } }
34616   {
34617     \tl_set:Nn \l__coffin_display_font_t1
34618       { \sffamily \tiny }
34619   }

```

(End definition for `\l__coffin_display_font_t1`.)

`\__coffin_rule:nn` Abstract out creation of rules here until there is a higher-level interface.

```

34620 \cs_new_protected:Npn \__coffin_rule:nn #1#2
34621   {
34622     \mode_leave_vertical:
34623     \hbox:n { \tex_vrule:D width #1 height #2 \scan_stop: }
34624   }

```

(End definition for `\_coffin_rule:nn`.)

`\coffin_mark_handle:Nnnn`  
`\coffin_mark_handle:cnnn`  
  `\_coffin_mark_handle_aux:nnnnNnn`

Marking a single handle is relatively easy. The standard attachment function is used, meaning that there are two calculations for the location. However, this is likely to be okay given the load expected. Contrast with the more optimised version for showing all handles which comes next.

```
34625 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
34626 {
34627   \hcoffin_set:Nn \l__coffin_display_pole_coffin
34628   {
34629     \color_select:n {#4}
34630     \_coffin_rule:nn { 1pt } { 1pt }
34631   }
34632   \_coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
34633     \l__coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }
34634   \hcoffin_set:Nn \l__coffin_display_coord_coffin
34635   {
34636     \color_select:n {#4}
34637     \l__coffin_display_font_tl
34638     ( \tl_to_str:n { #2 , #3 } )
34639   }
34640   \prop_get:NnN \l__coffin_display_handles_prop
34641     { #2 #3 } \l__coffin_internal_tl
34642   \quark_if_no_value:NTF \l__coffin_internal_tl
34643   {
34644     \prop_get:NnN \l__coffin_display_handles_prop
34645       { #3 #2 } \l__coffin_internal_tl
34646     \quark_if_no_value:NTF \l__coffin_internal_tl
34647     {
34648       \_coffin_attach_mark:NnnNnnnn #1 {#2} {#3}
34649         \l__coffin_display_coord_coffin { 1 } { vc }
34650           { 1pt } { Opt }
34651     }
34652   {
34653     \exp_last_unbraced:No \_coffin_mark_handle_aux:nnnnNnn
34654       \l__coffin_internal_tl #1 {#2} {#3}
34655   }
34656 }
34657 {
34658   \exp_last_unbraced:No \_coffin_mark_handle_aux:nnnnNnn
34659     \l__coffin_internal_tl #1 {#2} {#3}
34660   }
34661 }
34662 \cs_new_protected:Npn \_coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7
34663 {
34664   \_coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
34665     \l__coffin_display_coord_coffin {#1} {#2}
34666     { #3 \l__coffin_display_offset_dim }
34667     { #4 \l__coffin_display_offset_dim }
34668   }
34669 \cs_generate_variant:Nn \coffin_mark_handle:Nnnn { c }
```

(End definition for `\coffin_mark_handle:Nnnn` and `\_coffin_mark_handle_aux:nnnnNnn`. This function is documented on page [289](#).)

```
\coffin_display_handles:Nn
```

```
\coffin_display_handles:cn
```

```
\_coffin_display_handles_aux:nnnnn
```

```
\_coffin_display_handles_aux:nnnn
```

```
\_coffin_display_attach:NNnnn
```

Printing the poles starts by removing any duplicates, for which the H poles is used as the definitive version for the baseline and bottom. Two loops are then used to find the combinations of handles for all of these poles. This is done such that poles are removed during the loops to avoid duplication.

```
34670 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
34671 {
34672     \hcoffin_set:Nn \l__coffin_display_pole_coffin
34673     {
34674         \color_select:n {#2}
34675         \__coffin_rule:nn { 1pt } { 1pt }
34676     }
34677     \prop_set_eq:Nc \l__coffin_display_poles_prop
34678     { coffin ~ \__coffin_to_value:N #1 ~ poles }
34679     \__coffin_get_pole:NnN #1 { H } \l__coffin_pole_a_tl
34680     \__coffin_get_pole:NnN #1 { T } \l__coffin_pole_b_tl
34681     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
34682     { \prop_remove:Nn \l__coffin_display_poles_prop { T } }
34683     \__coffin_get_pole:NnN #1 { B } \l__coffin_pole_b_tl
34684     \tl_if_eq:NNT \l__coffin_pole_a_tl \l__coffin_pole_b_tl
34685     { \prop_remove:Nn \l__coffin_display_poles_prop { B } }
34686     \coffin_set_eq:NN \l__coffin_display_coffin #1
34687     \prop_map_inline:Nn \l__coffin_display_poles_prop
34688     {
34689         \prop_remove:Nn \l__coffin_display_poles_prop {##1}
34690         \__coffin_display_handles_aux:nnnnnn {##1} ##2 {##2}
34691     }
34692     \box_use_drop:N \l__coffin_display_coffin
34693 }
```

For each pole there is a check for an intersection, which here does not give an error if none is found. The successful values are stored and used to align the pole coffin with the main coffin for output. The positions are recovered from the preset list if available.

```
34694 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
34695 {
34696     \prop_map_inline:Nn \l__coffin_display_poles_prop
34697     {
34698         \bool_set_false:N \l__coffin_error_bool
34699         \__coffin_calculate_intersection:nnnnnnnn {##2} {##3} {##4} {##5} ##2
34700         \bool_if:NF \l__coffin_error_bool
34701         {
34702             \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
34703             \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
34704             \__coffin_display_attach:NNnnn
34705             \l__coffin_display_pole_coffin { hc } { vc }
34706             { Opt } { Opt }
34707             \hcoffin_set:Nn \l__coffin_display_coord_coffin
34708             {
34709                 \color_select:n {#6}
34710                 \l__coffin_display_font_tl
34711                 ( \tl_to_str:n { #1 , ##1 } )
34712             }
34713             \prop_get:Nn \l__coffin_display_handles_prop
34714             { #1 ##1 } \l__coffin_internal_tl
34715             \quark_if_no_value:NTF \l__coffin_internal_tl
```

```

34716    {
34717        \prop_get:NnN \l__coffin_display_handles_prop
34718            { ##1 #1 } \l__coffin_internal_tl
34719        \quark_if_no_value:NTF \l__coffin_internal_tl
34720            {
34721                \__coffin_display_attach:Nnnnn
34722                    \l__coffin_display_coord_coffin { 1 } { vc }
34723                        { 1pt } { Opt }
34724                }
34725            {
34726                \exp_last_unbraced:No
34727                    \__coffin_display_handles_aux:nnnn
34728                        \l__coffin_internal_tl
34729            }
34730        }
34731    {
34732        \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
34733            \l__coffin_internal_tl
34734    }
34735}
34736}
34737}
34738 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
34739{
34740    \__coffin_display_attach:Nnnnn
34741        \l__coffin_display_coord_coffin {#1} {#2}
34742        { #3 \l__coffin_display_offset_dim }
34743        { #4 \l__coffin_display_offset_dim }
34744}
34745 \cs_generate_variant:Nn \coffin_display_handles:Nn { c }
```

This is a dedicated version of `\coffin_attach:NnnNnnnn` with a hard-wired first coffin. As the intersection is already known and stored for the display coffin the code simply uses it directly, with no calculation.

```

34746 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
34747{
34748    \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
34749    \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
34750    \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
34751    \dim_set:Nn \l__coffin_offset_x_dim
34752        { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
34753    \dim_set:Nn \l__coffin_offset_y_dim
34754        { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
34755    \hbox_set:Nn \l__coffin_aligned_coffin
34756    {
34757        \box_use:N \l__coffin_display_coffin
34758        \__kernel_kern:n { -\box_wd:N \l__coffin_display_coffin }
34759        \__kernel_kern:n { \l__coffin_offset_x_dim }
34760        \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
34761    }
34762    \box_set_ht:Nn \l__coffin_aligned_coffin
34763        { \box_ht:N \l__coffin_display_coffin }
34764    \box_set_dp:Nn \l__coffin_aligned_coffin
34765        { \box_dp:N \l__coffin_display_coffin }
```

```

34766     \box_set_wd:Nn \l__coffin_aligned_coffin
34767     { \box_wd:N \l__coffin_display_coffin }
34768     \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
34769 }

```

(End definition for `\coffin_display_handles:Nn` and others. This function is documented on page 288.)

### `\coffin_show_structure:N`

`\coffin_show_structure:c`

### `\coffin_log_structure:N`

`\coffin_log_structure:c`

`\_\coffin_show_structure>NN`

For showing the various internal structures attached to a coffin in a way that keeps things relatively readable. If there is no apparent structure then the code complains.

```

34770 \cs_new_protected:Npn \coffin_show_structure:N
34771 { \_\coffin_show_structure:NN \msg_show:nxxxxx }
34772 \cs_generate_variant:Nn \coffin_show_structure:N { c }
34773 \cs_new_protected:Npn \coffin_log_structure:N
34774 { \_\coffin_show_structure:NN \msg_log:nxxxxx }
34775 \cs_generate_variant:Nn \coffin_log_structure:N { c }
34776 \cs_new_protected:Npn \_\coffin_show_structure:NN #1#2
34777 {
34778     \_\coffin_if_exist:NT #2
34779 {
34780     #1 { coffin } { show }
34781     { \token_to_str:N #2 }
34782 {
34783     \iow_newline: >- ht ~== \dim_eval:n { \coffin_ht:N #2 }
34784     \iow_newline: >- dp ~== \dim_eval:n { \coffin_dp:N #2 }
34785     \iow_newline: >- wd ~== \dim_eval:n { \coffin_wd:N #2 }
34786 }
34787 {
34788     \prop_map_function:cN
34789     { coffin ~ \_\coffin_to_value:N #2 ~ poles }
34790     \msg_show_item_unbraced:nn
34791 }
34792 {
34793 }
34794 }

```

(End definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `\_\coffin_show_structure>NN`. These functions are documented on page 289.)

### `\coffin_show:N`

`\coffin_show:c`

### `\coffin_log:N`

`\coffin_log:c`

`\coffin_show:Nnn`

`\coffin_show:cnn`

### `\coffin_log:Nnn`

`\coffin_log:cnn`

`\_\coffin_show>NNNnn`

Essentially a combination of `\coffin_show_structure:N` and `\box_show:Nnn`, but we need to avoid having two prompts, so we use `\msg_term:nxxxxx` instead of `\msg_show:nxxxxx` in the `show` case.

```

34795 \cs_new_protected:Npn \coffin_show:N #1
34796 { \coffin_show:Nnn #1 \c_max_int \c_max_int }
34797 \cs_generate_variant:Nn \coffin_show:N { c }
34798 \cs_new_protected:Npn \coffin_log:N #1
34799 { \coffin_log:Nnn #1 \c_max_int \c_max_int }
34800 \cs_generate_variant:Nn \coffin_log:N { c }
34801 \cs_new_protected:Npn \coffin_show:Nnn
34802 { \_\coffin_show:NNNnn \msg_term:nxxxxx \box_show:Nnn }
34803 \cs_generate_variant:Nn \coffin_show:Nnn { c }
34804 \cs_new_protected:Npn \coffin_log:Nnn
34805 { \_\coffin_show:NNNnn \msg_log:nxxxxx \box_show:Nnn }
34806 \cs_generate_variant:Nn \coffin_log:Nnn { c }
34807 \cs_new_protected:Npn \_\coffin_show:NNNnn #1#2#3#4#5

```

```

34808  {
34809    \__coffin_if_exist:NT #3
34810    {
34811      \__coffin_show_structure:NN #1 #3
34812      #2 #3 {#4} {#5}
34813    }
34814  }

```

(End definition for `\coffin_show:N` and others. These functions are documented on page 289.)

## 86.9 Messages

```

34815 \msg_new:nnnn { coffin } { no-pole-intersection }
34816   { No-intersection-between-coffin-poles. }
34817   {
34818     LaTeX-was-asked-to-find-the-intersection-between-two-poles,-
34819     but-they-do-not-have-a-unique-meeting-point:-
34820     the-value-(Opt,-Opt)-will-be-used.
34821   }
34822 \msg_new:nnnn { coffin } { unknown }
34823   { Unknown-coffin-'#1'. }
34824   { The-coffin-'#1'-was-never-defined. }
34825 \msg_new:nnnn { coffin } { unknown-pole }
34826   { Pole-'#1'-unknown-for-coffin-'#2'. }
34827   {
34828     LaTeX-was-asked-to-find-a-typesetting-pole-for-a-coffin,-
34829     but-either-the-coffin-does-not-exist-or-the-pole-name-is-wrong.
34830   }
34831 \msg_new:nnn { coffin } { show }
34832   {
34833     Size-of-coffin-#1 : #2 \\
34834     Poles-of-coffin-#1 : #3 .
34835   }
34836 
```

`</package>`

# Chapter 87

## I3color Implementation

```
34837  <*package>
34838  <@@=color>
```

### 87.1 Basics

\l\_color\_current\_tl The color currently active for foreground (text, *etc.*) material. This is stored in the form of a color model followed by one or more values. There are four pre-defined models, three of which take numerical values in the range [0, 1]:

- **gray** *<gray>* Grayscale color with the *<gray>* value running from 0 (fully black) to 1 (fully white)
- **cmyk** *<cyan>* *<magenta>* *<yellow>* *<black>*
- **rgb** *<red>* *<green>* *<blue>*

Notice that the value are separated by spaces. There is a fourth pre-defined model using a string value and a numerical one:

- **spot** *<name>* *<tint>* A pre-defined spot color, where the *<name>* should be a pre-defined string color name and the *<tint>* should be in the range [0, 1].

Additional models may be created to allow mixing of spot colors. The number of data entries these require will depend on the number of colors to be mixed.

**TeXhackers note:** The content of \l\_color\_current\_tl comprises two brace groups, the first containing the color model and the second containing the value(s) applicable in that model.

(End definition for \l\_color\_current\_tl.)

\color\_group\_begin: Grouping for color is the same as using the basic \group\_begin: and \group\_end: functions. However, for semantic reasons, they are renamed here.

```
34839  \cs_new_eq:NN \color_group_begin: \group_begin:
34840  \cs_new_eq:NN \color_group_end:   \group_end:
```

(End definition for \color\_group\_begin: and \color\_group\_end:. These functions are documented on page 291.)

<code>\color_ensure_current:</code>	A driver-independent wrapper for setting the foreground color to the current color “now”.
	<pre>34841 \cs_new_protected:Npn \color_ensure_current: 34842   { \__color_select:N \l__color_current_tl }</pre>
	(End definition for <code>\color_ensure_current</code> .. This function is documented on page 291.)
<code>\s_color_stop</code>	Internal scan marks.
	<pre>34843 \scan_new:N \s_color_stop</pre>
	(End definition for <code>\s_color_stop</code> .)
<code>\__color_select:N</code> <code>\__color_select_math:N</code> <code>\__color_select:nn</code>	Take an internal color specification and pass it to the driver. This code is needed to ensure the current color but will also be used by the higher-level material.
	<pre>34844 \cs_new_protected:Npn \__color_select:N #1 34845   { 34846     \exp_after:wN \__color_select:nn #1 34847     \group_insert_after:N \__color_backend_reset: 34848   } 34849 \cs_new_protected:Npn \__color_select_math:N #1 34850   { \exp_after:wN \__color_select:nn #1 } 34851 \cs_new_protected:Npn \__color_select:nn #1#2 34852   { \use:c { __color_backend_select_ #1 :n } {#2} }</pre>
	(End definition for <code>\__color_select:N</code> , <code>\__color_select_math:N</code> , and <code>\__color_select:nn</code> .)
<code>\l__color_current_tl</code>	The current color, with the model and
	<pre>34853 \tl_new:N \l__color_current_tl 34854 \tl_set:Nn \l__color_current_tl { { gray } { 0 } }</pre>
	(End definition for <code>\l__color_current_tl</code> .)

## 87.2 Predefined color names

The ability to predefine colors with a name is a key part of this module and means there has to be a method for storing the results. At first sight, it seems natural to follow the usual `expl3` model and create a `color` variable type for the process. That would then allow both local and global colors, constant colors and the like. However, these names need to be accessible in some form at the user level, for selection of colors either simply by name or as part of a more complex expression. This does not require that the full name is exposed but does require that they can be looked up in a predictable way. As such, it is more useful to expose just the color names as part of the interface, with the result that only local color names can be created. (This is also seen for example in key creation in `l3keys`.) As a result, color names are declarative (no `new` functions).

Since there is no need to manipulate colors *en masse*, each is stored in a two-part structure: a `prop` for the colors themselves, and a `tl` for the default model for each color.

## 87.3 Setup

```
\l_color_internal_int
\l_color_internal_tl
 34855 \int_new:N \l_color_internal_int
 34856 \tl_new:N \l_color_internal_tl

(End definition for \l_color_internal_int and \l_color_internal_tl.)
```

\s\_color\_mark Internal scan marks. \s\_color\_stop is already defined in l3color-base.  
34857 \scan\_new:N \s\_color\_mark

(End definition for \s\_color\_mark.)

## 87.4 Utility functions

\color\_if\_exist\_p:n A simple wrapper to avoid needing to have the lookup repeated in too many places. To  
guard against a color created in a group, we need to test for entries in the prop.

```
34858 \prg_new_conditional:Npnn \color_if_exist:n #1 { p , T, F, TF }
 34859 {
 34860   \prop_if_exist:cTF { l_color_named_ #1 _prop }
 34861   {
 34862     \prop_if_empty:cTF { l_color_named_ #1 _prop }
 34863     \prg_return_false:
 34864     \prg_return_true:
 34865   }
 34866   \prg_return_false:
 34867 }
```

(End definition for \color\_if\_exist:nTF. This function is documented on page 294.)

\\_\_color\_model:N Simple abstractions.

```
34868 \cs_new:Npn \__color_model:N #1 { \exp_after:wN \use_i:nn #1 }
 34869 \cs_new:Npn \__color_values:N #1 { \exp_after:wN \use_ii:nn #1 }
```

(End definition for \\_\_color\_model:N and \\_\_color\_values:N.)

\\_\_color\_extract:nNN Recover the values for the standard model for a color.

```
34870 \cs_new_protected:Npn \__color_extract:nNN #1#2#3
 34871 {
 34872   \tl_set_eq:Nc #2 { l_color_named_ #1 _tl }
 34873   \prop_get:cVN { l_color_named_ #1 _prop } #2 #3
 34874 }
 34875 \cs_generate_variant:Nn \__color_extract:nNN { V }
```

(End definition for \\_\_color\_extract:nNN.)

## 87.5 Model conversion

```
\__color_convert:nnN
\__color_convert:VVN
\__color_convert:nnN
\__color_convert:nVnN
\__color_convert:nnVN
t_rgb_rgb:w_____\__color_convert_rgb_cmyk:w
    \__color_convert_rgb_cmyk:nnn
    \__color_convert_rgb_cmyk:nnnn
```

Model conversion is carried out using standard formulae for base models, as described in the manual for `xcolor` (see also the *PostScript Language Reference Manual*). For other models direct conversion might not be defined, so we go through the fallback models if necessary.

```
34876 \cs_new_protected:Npn \__color_convert:nnN #1#2#3
34877     { \__color_convert:nnVN {#1} {#2} #3 #3 }
34878 \cs_generate_variant:Nn \__color_convert:nnN { VV }
34879 \cs_generate_variant:Nn \exp_last_unbraced:Nf { c }
34880 \cs_new_protected:Npn \__color_convert:nnnN #1#2#3#4
34881     {
34882         \tl_set:Nx #4
34883         {
34884             \cs_if_exist_use:cTF { __color_convert_ #1 _ #2 :w }
34885                 { #3 \s_color_stop }
34886             {
34887                 \cs_if_exist:cTF { __color_convert_ \use:c { c_colorFallback_ #1 _tl } _ #2 :
34888                     {
34889                         \exp_last_unbraced:cf
34890                             { __color_convert_ \use:c { c_colorFallback_ #1 _tl } _ #2 :w }
34891                             { \use:c { __color_convert_ #1 _ \use:c { c_colorFallback_ #1 _tl } :w
34892                                 \s_color_stop
34893                         }
34894                         {
34895                             \exp_last_unbraced:cf
34896                             { __color_convert_ \use:c { c_colorFallback_ #2 _tl } _ #2 :w }
34897                             {
34898                                 \cs_if_exist_use:cTF { __color_convert_ #1 _ \use:c { c_colorFallback_
34899                                     { #3 \s_color_stop }
34900                                     {
34901                                         \exp_last_unbraced:cf
34902                                             { __color_convert_ \use:c { c_colorFallback_ #1 _tl } _ \use:c
34903                                             { \use:c { __color_convert_ #1 _ \use:c { c_colorFallback_ #1 _tl } :w
34904                                                 \s_color_stop
34905                                         }
34906                                         }
34907                                         \s_color_stop
34908                                     }
34909                                     }
34910                                 }
34911                             }
34912 \cs_generate_variant:Nn \__color_convert:nnnn { nV , nnV }
34913 \cs_new:Npn \__color_convert_gray_gray:w #1 \s_color_stop
34914     { #1 }
34915 \cs_new:Npn \__color_convert_gray_rgb:w #1 \s_color_stop
34916     { #1 ~ #1 ~ #1 }
34917 \cs_new:Npn \__color_convert_gray_cmyk:w #1 \s_color_stop
34918     { 0 ~ 0 ~ 0 ~ \fp_eval:n { 1 - #1 } }
```

These rather odd values are based on NTSC television: the set are used for the cmyk conversion.

```
34919 \cs_new:Npn \__color_convert_rgb_gray:w #1 ~ #2 ~ #3 \s_color_stop
34920     { \fp_eval:n { 0.3 * #1 + 0.59 * #2 + 0.11 * #3 } }
```

```

34921 \cs_new:Npn \__color_convert_rgb_rgb:w #1 \s_color_stop
34922 { #1 }

```

The conversion from `rgb` to `cmyk` is the most complex: a two-step procedure which requires *black generation* and *undercolor removal* functions. The PostScript reference describes them as device-dependent, but following `xcolor` we assume they are linear. Moreover, as the likelihood of anyone using a non-unitary matrix here is tiny, we simplify and treat those two concepts as no-ops. To allow code sharing with parsing of `cmy` values, we have an intermediate function here (`\__color_convert_rgb_cmyk:nnn`) which actually takes `cmy` values as input.

```

34923 \cs_new:Npn \__color_convert_rgb_cmyk:w #1 ~ #2 ~ #3 \s_color_stop
34924 {
34925     \exp_args:Neee \__color_convert_rgb_cmyk:nnn
34926         { \fp_eval:n { 1 - #1 } }
34927         { \fp_eval:n { 1 - #2 } }
34928         { \fp_eval:n { 1 - #3 } }
34929     }
34930 \cs_new:Npn \__color_convert_rgb_cmyk:nnn #1#2#3
34931 {
34932     \exp_args:Ne \__color_convert_rgb_cmyk:nnnn
34933         { \fp_eval:n { min( #1, #2 , #3 ) } } {#1} {#2} {#3}
34934 }
34935 \cs_new:Npn \__color_convert_rgb_cmyk:nnnn #1#2#3#4
34936 {
34937     \fp_eval:n { min ( 1 , max ( 0 , #2 - #1 ) ) } \c_space_tl
34938     \fp_eval:n { min ( 1 , max ( 0 , #3 - #1 ) ) } \c_space_tl
34939     \fp_eval:n { min ( 1 , max ( 0 , #4 - #1 ) ) } \c_space_tl
34940     #1
34941 }
34942 \cs_new:Npn \__color_convert_cmyk_gray:w #1 ~ #2 ~ #3 ~ #4 \s_color_stop
34943     { \fp_eval:n { 1 - min ( 1 , 0.3 * #1 + 0.59 * #2 + 0.11 * #3 + #4 ) } }
34944 \cs_new:Npn \__color_convert_cmyk_rgb:w #1 ~ #2 ~ #3 ~ #4 \s_color_stop
34945 {
34946     \fp_eval:n { 1 - min ( 1 , #1 + #4 ) } \c_space_tl
34947     \fp_eval:n { 1 - min ( 1 , #2 + #4 ) } \c_space_tl
34948     \fp_eval:n { 1 - min ( 1 , #3 + #4 ) }
34949 }
34950 \cs_new:Npn \__color_convert_cmyk_cmyk:w #1 \s_color_stop
34951 { #1 }

```

(End definition for `\__color_convert:nnN` and others.)

## 87.6 Color expressions

<code>\l_color_model_tl</code> <code>\l_color_value_tl</code> <code>\l_color_next_model_tl</code> <code>\l_color_next_value_tl</code>	Working space to store the color data whilst doing calculations: keeping it on the stack is attractive but gets tricky (return is non-trivial).
--	---

```

34952 \tl_new:N \l_color_model_tl
34953 \tl_new:N \l_color_value_tl
34954 \tl_new:N \l_color_next_model_tl
34955 \tl_new:N \l_color_next_value_tl

```

(End definition for `\l_color_model_tl` and others.)

```

\__color_parse:nN
\__color_parse_aux:nN
\__color_parse_eq:Nn
\__color_parse_eq:nNn
    \__color_parse:Nw
\__color_parse_loop_init:Nnn
    \__color_parse_loop:w
\__color_parse_loop:nn
    \__color_parse_gray:n
    \__color_parse_std:n
\__color_parse_break:w
    \__color_parse_end:
\__color_parse_mix:Nnnn
\__color_parse_mix:NVn
\__color_parse_mix:nNnn

```

```

\__color_parse_mix_gray:nw
\__color_parse_mix_rgb:nw
\__color_parse_mix_cmyk:nw

```

The main function for parsing color expressions removes actives but otherwise expands, then starts working through the expression itself. At the end, we apply the payload.

```

34956 \cs_new_protected:Npx \__color_parse:nN #1#2
34957 {
34958     \tl_set:Nx \exp_not:c { l__color_named_. _tl }
34959         { \exp_not:N \__color_model:N \exp_not:N \l__color_current_tl }
34960     \prop_put:NVx \exp_not:c { l__color_named_. _prop }
34961         \exp_not:c { l__color_named_. _tl }
34962         { \exp_not:N \__color_values:N \exp_not:N \l__color_current_tl }
34963     \exp_not:N \exp_args:Ne \exp_not:N \__color_parse_aux:nN
34964         { \exp_not:N \tl_to_str:n {#1} } #2
34965 }

```

Before going to all of the effort of parsing an expression, these two precursor functions look for a pre-defined name, either on its own or with a trailing ! (which is the same thing).

```

34966 \cs_new_protected:Npn \__color_parse_aux:nN #1#2
34967 {
34968     \color_if_exist:nTF {#1}
34969         { \__color_parse_set_eq:Nn #2 {#1} }
34970         { \__color_parse:Nw #2#1 ! \s__color_stop }
34971     \__color_check_model:N #2
34972 }
34973 \cs_new_protected:Npn \__color_parse_set_eq:Nn #1#2
34974 {
34975     \tl_if_empty:NTF \l_color_fixed_model_tl
34976         { \exp_args:Nv \__color_parse_set_eq:nNn { l__color_named_. #2 _tl } }
34977         { \exp_args:NV \__color_parse_set_eq:nNn \l_color_fixed_model_tl }
34978             #1 {#2}
34979 }

```

Here, we have to allow for the case where there is a fixed model: that can't be swept up by generic conversion as we are dealing with a named color.

```

34980 \cs_new_protected:Npn \__color_parse_set_eq:nNn #1#2#3
34981 {
34982     \prop_get:cnNTF
34983         { l__color_named_. #3 _prop } {#1}
34984     \l_color_value_tl
34985         { \tl_set:Nx #2 { {#1} { \l__color_value_tl } } }
34986         {
34987             \tl_set_eq:Nc \l_color_model_tl { l__color_named_. #3 _tl }
34988             \prop_get:cVN { l__color_named_. #3 _prop } \l_color_model_tl
34989                 \l__color_value_tl
34990             \__color_convert:nnN
34991                 \l_color_model_tl {#1} \l__color_value_tl
34992             \tl_set:Nx #2
34993                 {
34994                     {#1}
34995                     { \l__color_value_tl }
34996                 }
34997             }
34998         }
34999 \cs_new_protected:Npn \__color_parse:Nw #1#2 ! #3 \s__color_stop
35000 {

```

```

35001     \color_if_exist:nTF {#2}
35002     {
35003         \tl_if_blank:nTF {#3}
35004             { \__color_parse_set_eq:Nn #1 {#2} }
35005             { \__color_parse_loop_init:Nnn #1 {#2} {#3} }
35006     }
35007     {
35008         \msg_error:nnn { color } { unknown-color } {#2}
35009         \tl_set:Nn \l__color_current_tl { { gray } { 0 } }
35010     }
35011 }

```

Once we establish that a full parse is needed, the next job is to get the detail of the first color. That will determine the model we use for the calculation: splitting here makes checking that a bit easier.

```

35012 \cs_new_protected:Npn \__color_parse_loop_init:Nnn #1#2#3
35013 {
35014     \group_begin:
35015         \__color_extract:nNN {#2} \l__color_model_tl \l__color_value_tl
35016         \__color_parse_loop:w #3 ! ! ! \s__color_stop
35017         \tl_set:Nx \l__color_internal_tl
35018             { { \l__color_model_tl } { \l__color_value_tl } }
35019         \exp_args:NNNV \group_end:
35020         \tl_set:Nn #1 \l__color_internal_tl
35021 }

```

This is the loop proper: there can be an open-ended set of colors to parse, separated by ! tokens. There are a few cases to look out for. At the end of the expression and with we find a mix of 100 then we simply skip the next color entirely (we can't stop the loop as there might be a further valid color to mix in). On the other hand, if we get a mix of 0 then drop everything so far and start again. There is also a trailing white to "read in" if the final explicit data is a mix. Those conditions are separate from actually looping, which is therefore sorted out by checking if we have further data to process: in contrast to `xcolor`, we don't allow !! so the test can be simplified.

```

35022 \cs_new_protected:Npn \__color_parse_loop:w #1 ! #2 ! #3 ! #4 ! #5 \s__color_stop
35023 {
35024     \bool_lazy_or:nnF
35025         { \tl_if_blank_p:n {#1} }
35026         { \int_compare_p:nNn {#1} = { 100 } }
35027     {
35028         \int_compare:nNnTF {#1} = { 0 }
35029             {
35030                 \tl_if_blank:nTF {#2}
35031                     { \__color_extract:nNN { white } }
35032                     { \__color_extract:nNN {#2} }
35033                     \l__color_model_tl \l__color_value_tl
35034             }
35035             {
35036                 \use:x
35037                     {
35038                         \__color_parse_loop:nn {#1}
35039                         { \tl_if_blank:nTF {#2} { white } {#2} }
35040                     }
35041             }

```

```

35042      }
35043      \tl_if_blank:nF {#3}
35044      { \__color_parse_loop:w #3 ! #4 ! #5 \s__color_stop }
35045      \__color_parse_end:
35046    }

```

The “payload” of calculation in the loop first. If the model for the upcoming color is different from that of the existing (partial) color, convert the model. For `gray` the two are flipped round so that the outcome is something with “real” color. We are then in a position to do the actual calculation itself. The two auxiliaries here give us a way to break the loop should an invalid name be found.

```

35047  \cs_new_protected:Npn \__color_parse_loop:nn #1#2
35048  {
35049    \color_if_exist:nTF {#2}
35050    {
35051      \__color_extract:nNN {#2} \l__color_next_model_tl \l__color_next_value_tl
35052      \tl_if_eq:NNF \l__color_model_tl \l__color_next_model_tl
35053      {
35054        \str_if_eq:VnTF \l__color_model_tl { gray }
35055        { \__color_parse_gray:n {#2} }
35056        { \__color_parse_std:n {#2} }
35057      }
35058      \tl_set:Nx \l__color_value_tl
35059      {
35060        \__color_parse_mix:NVN
35061        \l__color_model_tl \l__color_value_tl \l__color_next_value_tl {#1}
35062      }
35063    }
35064    {
35065      \msg_error:nnn { color } { unknown-color } {#2}
35066      \__color_extract:nNN { black } \l__color_model_tl \l__color_value_tl
35067      \__color_parse_break:w
35068    }
35069  }

```

The `gray` model needs special handling: the models need to be swapped: we do that using a dedicated function.

```

35070  \cs_new_protected:Npn \__color_parse_gray:n #1
35071  {
35072    \tl_set_eq:NN \l__color_model_tl \l__color_next_model_tl
35073    \tl_set:Nn \l__color_next_model_tl { gray }
35074    \exp_args:NnV \__color_convert:nnN { gray } \l__color_model_tl
35075    \l__color_value_tl
35076    \prop_get:cVN { l__color_named_ #1 _prop } \l__color_model_tl
35077    \l__color_next_value_tl
35078  }
35079  \cs_new_protected:Npn \__color_parse_std:n #1
35080  {
35081    \prop_get:cVNF { l__color_named_ #1 _prop }
35082    \l__color_model_tl
35083    \l__color_next_value_tl
35084    {
35085      \__color_convert:VVN
35086      \l__color_next_model_tl

```

```

35087           \l_color_model_tl
35088           \l_color_next_value_tl
35089       }
35090   }
35091 \cs_new_protected:Npn \__color_parse_break:w #1 \__color_parse_end: { }
35092 \cs_new_protected:Npn \__color_parse_end: { }

```

Do the vector arithmetic: mainly a question of shuffling input, along with one pre-calculation to keep down the use of division.

```

35093 \cs_new:Npn \__color_parse_mix:Nnnn #1#2#3#4
35094 {
35095     \exp_args:Nf \__color_parse_mix:nNnn
35096     { \fp_eval:n { #4 / 100 } }
35097     #1 {#2} {#3}
35098 }
35099 \cs_generate_variant:Nn \__color_parse_mix:Nnnn { NVV }
35100 \cs_new:Npn \__color_parse_mix:nNnn #1#2#3#4
35101 {
35102     \use:c { __color_parse_mix_ #2 :nw } {#1}
35103     #3 \s_color_mark #4 \s_color_stop
35104 }
35105 \cs_new:Npn \__color_parse_mix_gray:nw #1#2 \s_color_mark #3 \s_color_stop
35106 { \fp_eval:n { #2 * #1 + #3 * ( 1 - #1 ) } }
35107 \cs_new:Npn \__color_parse_mix_rgb:nw
35108 #1#2 ~ #3 ~ #4 \s_color_mark #5 ~ #6 ~ #7 \s_color_stop
35109 {
35110     \fp_eval:n { #2 * #1 + #5 * ( 1 - #1 ) } \c_space_tl
35111     \fp_eval:n { #3 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
35112     \fp_eval:n { #4 * #1 + #7 * ( 1 - #1 ) }
35113 }
35114 \cs_new:Npn \__color_parse_mix_cmyk:nw
35115 #1#2 ~ #3 ~ #4 ~ #5 \s_color_mark #6 ~ #7 ~ #8 ~ #9 \s_color_stop
35116 {
35117     \fp_eval:n { #2 * #1 + #6 * ( 1 - #1 ) } \c_space_tl
35118     \fp_eval:n { #3 * #1 + #7 * ( 1 - #1 ) } \c_space_tl
35119     \fp_eval:n { #4 * #1 + #8 * ( 1 - #1 ) } \c_space_tl
35120     \fp_eval:n { #5 * #1 + #9 * ( 1 - #1 ) }
35121 }

```

(End definition for `\__color_parse:nN` and others.)

```

\__color_parse_model_gray:w
\__color_parse_model_rgb:w
\__color_parse_model_cmyk:w
\__color_parse_number:n
\__color_parse_number:w

```

Turn the input into internal form, also tidying up the number quickly.

```

35122 \cs_new:Npn \__color_parse_model_gray:w #1 , #2 \s_color_stop
35123 { { gray } { \__color_parse_number:n {#1} } }
35124 \cs_new:Npn \__color_parse_model_rgb:w #1 , #2 , #3 , #4 \s_color_stop
35125 {
35126     { rgb }
35127     {
35128         \__color_parse_number:n {#1} ~
35129         \__color_parse_number:n {#2} ~
35130         \__color_parse_number:n {#3}
35131     }
35132 }
35133 \cs_new:Npn \__color_parse_model_cmyk:w #1 , #2 , #3 , #4 , #5 \s_color_stop
35134 {

```

```

35135     { cmyk }
35136     {
35137         \__color_parse_number:n {#1} ~
35138         \__color_parse_number:n {#2} ~
35139         \__color_parse_number:n {#3} ~
35140         \__color_parse_number:n {#4}
35141     }
35142   }
35143 \cs_new:Npn \__color_parse_number:n #1
35144   { \__color_parse_number:w #1 . 0 . \s__color_stop }
35145 \cs_new:Npn \__color_parse_number:w #1 . #2 . #3 \s__color_stop
35146   { \tl_if_blank:nTF {#1} { 0 } {#1} . #2 }

(End definition for \__color_parse_model_gray:w and others.)

```

```

\__color_parse_model_Gray:w
\__color_parse_model_hsb:w
\__color_parse_model_Hsb:w
\__color_parse_model_HSB:w
\__color_parse_model_HTML:w
\__color_parse_model_RGB:w
\__color_parse_model_hsb:nnn
  \__color_parse_model_hsb_aux:nnn
    \__color_parse_model_hsb:nnnn
      \__color_parse_model_hsb:nnnnn
        \__color_parse_model_hsb_0:nnnn
        \__color_parse_model_hsb_1:nnnn
        \__color_parse_model_hsb_2:nnnn
        \__color_parse_model_hsb_3:nnnn
        \__color_parse_model_hsb_4:nnnn
        \__color_parse_model_hsb_5:nnnn
\__color_parse_model_wave:w
  \__color_parse_model_wave_auxi:nn
  \__color_parse_model_wave_auxii:nn
  \__color_parse_model_wave_rho:n

```

The conversion here is non-trivial but is described at length in the `xcolor` manual. For ease, we calculate the integer and fractional parts of the hue first, then use them to work out the possible values for  $r$ ,  $g$  and  $b$  before putting them in the correct places.

```

35156 \cs_new:Npn \__color_parse_model_hsb:nnn #1#2#3
35157   {
35158     { rgb }
35159     {
35160       \exp_args:Ne \__color_parse_model_hsb_aux:nnn
35161         { \fp_eval:n { 6 * #1 } } {#2} {#3}
35162     }
35163   }
35164 \cs_new:Npn \__color_parse_model_hsb_aux:nnn #1#2#3
35165   {
35166     \exp_args:Nee \__color_parse_model_hsb_aux:nnnn
35167       { \fp_eval:n { floor(#1) } } { \fp_eval:n { #1 - floor(#1) } }
35168       {#2} {#3}
35169   }
35170 \cs_new:Npn \__color_parse_model_hsb_aux:nnnn #1#2#3#4
35171   {
35172     \use:e
35173     {
35174       \exp_not:N \__color_parse_model_hsb_aux:nnnnn
35175         { \__color_parse_number:n {#4} }
35176         { \fp_eval:n { round(#4 * (1 - #3) ,5) } }
35177         { \fp_eval:n { round(#4 * ( 1 - #3 * #2 ) ,5) } }
35178         { \fp_eval:n { round(#4 * ( 1 - #3 * (1 - #2) ) ,5) } }
35179         {#1}
35180     }
35181   }

```

```

35182 \cs_new:Npn \__color_parse_model_hsb_aux:nnnnn #1#2#3#4#5
35183   { \use:c { __color_parse_model_hsb_ #5 :nnnn } {#1} {#2} {#3} {#4} }
35184 \cs_new:cpn { __color_parse_model_hsb_0:nnnn } #1#2#3#4 { #1 ~ #4 ~ #2 }
35185 \cs_new:cpn { __color_parse_model_hsb_1:nnnn } #1#2#3#4 { #3 ~ #1 ~ #2 }
35186 \cs_new:cpn { __color_parse_model_hsb_2:nnnn } #1#2#3#4 { #2 ~ #1 ~ #4 }
35187 \cs_new:cpn { __color_parse_model_hsb_3:nnnn } #1#2#3#4 { #2 ~ #3 ~ #1 }
35188 \cs_new:cpn { __color_parse_model_hsb_4:nnnn } #1#2#3#4 { #4 ~ #2 ~ #1 }
35189 \cs_new:cpn { __color_parse_model_hsb_5:nnnn } #1#2#3#4 { #1 ~ #2 ~ #3 }
35190 \cs_new:cpn { __color_parse_model_hsb_6:nnnn } #1#2#3#4 { #1 ~ #2 ~ #2 }
35191 \cs_new:Npn \__color_parse_model_HSB:w #1 , #2 , #3 , #4 \s__color_stop
35192   {
35193     \exp_args:Nee \__color_parse_model_hsb:nnn
35194       { \fp_eval:n {#1 / 240} }
35195       { \fp_eval:n {#2 / 240} }
35196       { \fp_eval:n {#3 / 240} }
35197   }
35198 \cs_new:Npn \__color_parse_model_HTML:w #1 , #2 \s__color_stop
35199   { __color_parse_model_HTML_aux:w #1 0 0 0 0 0 \s__color_stop }
35200 \cs_new:Npn \__color_parse_model_HTML_aux:w #1#2#3#4#6#7 \s__color_stop
35201   {
35202     { rgb }
35203     {
35204       \fp_eval:n { round(\int_from_hex:n {#1#2} / 255,5) } ~
35205       \fp_eval:n { round(\int_from_hex:n {#3#4} / 255,5) } ~
35206       \fp_eval:n { round(\int_from_hex:n {#5#6} / 255,5) }
35207     }
35208   }
35209 \cs_new:Npn \__color_parse_model_RGB:w #1 , #2 , #3 , #4 \s__color_stop
35210   {
35211     { rgb }
35212     {
35213       \fp_eval:n { round(#1 / 255,5) } ~
35214       \fp_eval:n { round(#2 / 255,5) } ~
35215       \fp_eval:n { round(#3 / 255,5) }
35216     }
35217   }

```

Following the description in the `xcolor` manual. As we always use `rgb`, there is no need to find the sixth, we just pas the information straight to the `hsb` auxiliary defined earlier.

```

35218 \cs_new:Npn \__color_parse_model_wave:w #1 , #2 \s__color_stop
35219   {
35220     { rgb }
35221     {
35222       \fp_compare:nNnTF {#1} < { 420 }
35223         { __color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 380) / 40 } }
35224       }
35225       {
35226         \fp_compare:nNnTF {#1} > { 700 }
35227           { __color_parse_model_wave_auxi:nn {#1} { 0.3 + 0.7 * (#1 - 780) / -80 } }
35228           { __color_parse_model_wave_auxi:nn {#1} { 1 } }
35229       }
35230     }
35231   }
35232 \cs_new:Npn \__color_parse_model_wave_auxi:nn #1#2

```

```

35233   {
35234     \fp_compare:nNnTF {#1} < { 440 }
35235     {
35236       \__color_parse_model_wave_auxii:nn
35237         { 4 + \__color_parse_model_wave_rho:n { (#1 - 440) / -60 } }
35238         {#2}
35239     }
35240   {
35241     \fp_compare:nNnTF {#1} < { 490 }
35242     {
35243       \__color_parse_model_wave_auxii:nn
35244         { 4 - \__color_parse_model_wave_rho:n { (#1 - 440) / 50 } }
35245         {#2}
35246     }
35247   {
35248     \fp_compare:nNnTF {#1} < { 510 }
35249     {
35250       \__color_parse_model_wave_auxii:nn
35251         { 2 + \__color_parse_model_wave_rho:n { (#1 - 510) / -20 } }
35252         {#2}
35253     }
35254   {
35255     \fp_compare:nNnTF {#1} < { 580 }
35256     {
35257       \__color_parse_model_wave_auxii:nn
35258         { 2 - \__color_parse_model_wave_rho:n { (#1 - 510) / 70 } }
35259         {#2}
35260     }
35261   {
35262     \fp_compare:nNnTF {#1} < { 645 }
35263     {
35264       \__color_parse_model_wave_auxii:nn
35265         { \__color_parse_model_wave_rho:n { (#1 - 645) / -65 } }
35266         {#2}
35267     }
35268     { \__color_parse_model_wave_auxii:nn { 0 } {#2} }
35269   }
35270 }
35271 }
35272 }
35273 \cs_new:Npn \__color_parse_model_wave_auxii:nn #1#2
35274   {
35275     \exp_args:Neee \__color_parse_model_hsb_aux:nnn
35276       { \fp_eval:n {#1} }
35277       { 1 }
35278       { \__color_parse_model_wave_rho:n {#2} }
35279   }
35280 }
35281 \cs_new:Npn \__color_parse_model_wave_rho:n #1
35282   { \fp_eval:n { min(1, max(0,#1) ) } }

```

(End definition for `\__color_parse_model_Gray:w` and others.)

`\__color_parse_model_cmy:w` Simply pass data to the conversion functions.

```

35283 \cs_new:Npn \__color_parse_model_cmy:w #1 , #2 , #3 , #4 \s__color_stop
35284 {
35285     { cmyk }
35286     { \__color_convert_rgb_cmyk:nnn {#1} {#2} {#3} }
35287 }

```

(End definition for `\__color_parse_model_cmy:w`.)

```

\__color_parse_model_tHsb:w
\__color_parse_model_tHsb:n
\__color_parse_model_tHsb:nw

```

There are three stages to the process here: bring the `tH` argument into the normal range, divide through to get to `hsb` and finally convert that to `rgb`. The final stage can be delegated to the parsing function for `hsb`, and the conversion from `Hsb` to `hsb` is trivial, so the main focus here is the first stage. We use a simple expandable loop to do the work, and we implement the equation given in the `xcolor` manual (number 85 there) as a simple expression.

```

35288 \cs_new:Npn \__color_parse_model_tHsb:w #1 , #2 , #3 , #4 \s__color_stop
35289 {
35290     \exp_args:Ne \__color_parse_model_hsb:nnn
35291     { \__color_parse_model_tHsb:n {#1} } {#2} {#3}
35292 }
35293 \cs_new:Npn \__color_parse_model_tHsb:n #1
35294 {
35295     \__color_parse_model_tHsb:nw {#1}
35296     0 , 0 ;
35297     60 , 30 ;
35298     120 , 60 ;
35299     180 , 120 ;
35300     210 , 180 ;
35301     240 , 240 ;
35302     360 , 360 ;
35303     \q_recursion_tail ,
35304     \q_recursion_stop
35305 }
35306 \cs_new:Npn \__color_parse_model_tHsb:nw #1 #2 , #3 ; #4 , #5 ;
35307 {
35308     \quark_if_recursion_tail_stop_do:nn {#4} { 0 }
35309     \fp_compare:nNnTF {#1} > {#4}
35310     { \__color_parse_model_tHsb:nw {#1} #4 , #5 ; }
35311     {
35312         \use_i_delimit_by_q_recursion_stop:nw
35313         { \fp_eval:n { ((#1 - #2) / (#4 - #2) * (#5 - #3) + #3) / 360 } }
35314     }
35315 }

```

(End definition for `\__color_parse_model_tHsb:w`, `\__color_parse_model_tHsb:n`, and `\__color_parse_model_tHsb:nw`.)

```
\__color_parse_model_&spot:w
```

We cannot extract data here from that passed by `xcolor`, so we fall back on a black tint.

```

35316 \cs_new:cpn { __color_parse_model_&spot:w } #1 , #2 \s__color_stop
35317 { { gray } { #1 } }

```

(End definition for `\__color_parse_model_&spot:w`.)

## 87.7 Selecting colors (and color models)

`\l_color_fixed_model_tl`

For selecting a single fixed model.

```
35318 \tl_new:N \l_color_fixed_model_tl
```

(End definition for `\l_color_fixed_model_tl`. This variable is documented on page 294.)

`\__color_check_model:N`

Check that the model in use is the one required.

```
35319 \cs_new_protected:Npn \__color_check_model:N #1
35320 {
35321   \tl_if_empty:NF \l_color_fixed_model_tl
35322   {
35323     \exp_after:wN \__color_check_model:nn #1
35324     \tl_if_eq:NNF \l_color_model_tl \l_color_fixed_model_tl
35325     {
35326       \__color_convert:VVN \l_color_model_tl \l_color_fixed_model_tl
35327       \l_color_value_tl
35328     }
35329     \tl_set:Nx #1
35330     { { \l_color_fixed_model_tl } { \l_color_value_tl } }
35331   }
35332 }
35333 \cs_new_protected:Npn \__color_check_model:nn #1#2
35334 {
35335   \tl_set:Nn \l_color_model_tl {#1}
35336   \tl_set:Nn \l_color_value_tl {#2}
35337 }
```

(End definition for `\__color_check_model:N` and `\__color_check_model:nn`.)

`\__color_finalise_current:`

A backend-neutral location for “last minute” manipulations before handing off to the backend code. We set the special `.` syntax here: this will therefore always be available. The finalisation is separate from the main function so it can also be applied to *e.g.* page color.

```
35338 \cs_new_protected:Npx \__color_finalise_current:
35339 {
35340   \tl_set:Nx \exp_not:c { l_color_named_._tl }
35341   { \exp_not:N \__color_model:N \exp_not:N \l_color_current_tl }
35342   \prop_clear:N \exp_not:c { l_color_named_._prop }
35343   \prop_put:NVx \exp_not:c { l_color_named_._prop }
35344   \exp_not:c { l_color_named_._tl }
35345   { \exp_not:N \__color_values:N \exp_not:N \l_color_current_tl }
35346 }
```

(End definition for `\__color_finalise_current:..`)

`\color_select:n`  
`\color_select:nn`

`\__color_select_main:Nw`

`\__color_select_loop:Nw`

`\__color_select:nnN`

`\__color_select_swap:Nnn`

Parse the input expressions then get the backend to actually activate them. The main complexity here is the need to check through multiple models. That is done “locally” here as the approach is subtly different to when different models are being stored.

```
35347 \cs_new_protected:Npn \color_select:n #1
35348 {
35349   \__color_parse:nN {#1} \l_color_current_tl
35350   \__color_finalise_current:
35351   \__color_select:N \l_color_current_tl
```

```

35352     }
35353 \cs_new_protected:Npn \color_select:nn #1#2
35354 {
35355     \__color_select_main:Nw \l__color_current_tl
35356     #1 / \s__color_mark #2 / \s__color_stop
35357     \__color_finalise_current:
35358     \__color_select:N \l__color_current_tl
35359 }

```

If the first color model is the fixed one, or if there is no fixed model, we don't need most of the data: just set up and apply the backend function.

```

35360 \cs_new_protected:Npn \__color_select_main:Nw
35361 #1 #2 / #3 / #4 \s__color_mark #5 / #6 / #7 \s__color_stop
35362 {
35363     \__color_select:nnN {#2} {#5} #1
35364     \bool_lazy_or:nnF
35365     { \tl_if_empty_p:N \l_color_fixed_model_tl }
35366     { \str_if_eq_p:nV {#2} \l_color_fixed_model_tl }
35367     { \__color_select_loop:Nw #1 #3 / #4 \s__color_mark #6 / #7 \s__color_stop }
35368 }

```

If a fixed model applies, we need to check each possible value in order. If there is no hit at all, fall back on the generic formula-based interchange.

```

35369 \cs_new_protected:Npn \__color_select_loop:Nw
35370 #1 #2 / #3 \s__color_mark #4 / #5 \s__color_stop
35371 {
35372     \str_if_eq:nVTF {#2} \l_color_fixed_model_tl
35373     { \__color_select:nnN {#2} {#4} #1 }
35374     {
35375         \tl_if_blank:nTF {#2}
35376         { \exp_after:wN \__color_select_swap:Nnn \exp_after:wN #1 #1 }
35377         { \__color_select_loop:Nw #1 #3 \s__color_mark #5 \s__color_stop }
35378     }
35379 }
35380 \cs_new_protected:Npn \__color_select:nnN #1#2#3
35381 {
35382     \cs_if_exist:cTF { __color_parse_model_ #1 :w }
35383     {
35384         \tl_set:Nx #3
35385         { \use:c { __color_parse_model_ #1 :w } #2 , 0 , 0 , 0 , 0 \s__color_stop }
35386     }
35387     { \msg_error:nnn { color } { unknown-model } {#1} }
35388 }
35389 \cs_new_protected:Npn \__color_select_swap:Nnn #1#2#3
35390 {
35391     \__color_convert:nVnN {#2} \l_color_fixed_model_tl {#3} \l__color_value_tl
35392     \tl_set:Nx #1
35393     { { \l_color_fixed_model_tl } { \l__color_value_tl } }
35394 }

```

(End definition for `\color_select:n` and others. These functions are documented on page 294.)

## 87.8 Math color

The approach here is the same as for the L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  `\mathcolor` command, but as we are working at the `expl3` level we can make some minor changes.

`\l_color_math_active_tl` Tokens representing active sub/superscripts.

```
35395 \tl_new:N \l_color_math_active_tl
35396 \tl_set:Nn \l_color_math_active_tl { ' }
```

(End definition for `\l_color_math_active_tl`. This function is documented on page 295.)

`\g_color_math_seq` Not all engines have multiple color stacks, and at the same time we are not expecting breaking within a colored math fragment. So we track the color stack ourselves.

```
35397 \seq_new:N \g_color_math_seq
```

(End definition for `\g_color_math_seq`.)

`\color_math:nn` `\color_math:nnn` `\_color_math:nn` The basic set up here is relatively simple: store the current color, parse the new color as-normal, then switch color before inserting the tokens we are asked to change. The tricky part is right at the end, handling the reset.

```
35398 \cs_new_protected:Npn \color_math:nn #1#2
35399 {
35400     \_color_math:nn {#2}
35401     { \_color_parse:nN {#1} \l_color_current_tl }
35402 }
35403 \cs_new_protected:Npn \color_math:nnn #1#2#3
35404 {
35405     \_color_math:nn {#3}
35406     {
35407         \_color_select_main:Nw \l_color_current_tl
35408         #1 / / \s_color_mark #2 / / \s_color_stop
35409     }
35410 }
35411 \cs_new_protected:Npn \_color_math:nn #1#2
35412 {
35413     \seq_gpush:NV \g_color_math_seq \l_color_current_tl
35414     #2
35415     \_color_select_math:N \l_color_current_tl
35416     #1
35417     \_color_math_scan:w
35418 }
```

(End definition for `\color_math:nn`, `\color_math:nnn`, and `\_color_math:nn`. These functions are documented on page 295.)

`\_color_math_scan:w` `\_color_math_scan_auxi:` `\_color_math_scan_auxii:` `\_color_math_scan_end:` The complication when changing the color back is due to the fact that the `\color_math:nn(n)` may be followed by `^` or `_` or the hidden superscript (for example `'`) and its argument may end in a `\mathop` in which case the sub- and superscripts may be attached as `\limits` instead of after the material. All cases need separate treatment. To avoid repeatedly collecting the same token, we first check for an alignment tab: assuming we don't have one of those, we can "recycle" `\l_peek_token` safely. As we have an explicit `\c_alignment_token`, there needs to be an align-safe group present.

```
35419 \cs_new_protected:Npn \_color_math_scan:w
35420 {
```

```

35421 \peek_remove_filler:n
35422 {
35423     \group_align_safe_begin:
35424     \peek_catcode:NTF \c_alignment_token
35425     {
35426         \group_align_safe_end:
35427         \__color_math_scan_end:
35428     }
35429     {
35430         \group_align_safe_end:
35431         \__color_math_scan_auxi:
35432     }
35433 }
35434 }
```

Dealing with literal `_` and `^` is easy, and as we have exactly two cases, we can hard-code this. We use a hard-coded list for limits: these are all primitives. The `\use_none:n` here also removes the test token so it is left just in the right place.

```

35435 \cs_new_protected:Npn \__color_math_scan_auxi:
35436 {
35437     \token_case_catcode:NnTF \l_peek_token
35438     {
35439         \c_math_subscript_token { }
35440         \c_math_superscript_token { }
35441     }
35442     { \__color_math_scripts:Nw }
35443     {
35444         \token_case_meaning:NnTF \l_peek_token
35445         {
35446             \tex_limits:D { \tex_limits:D }
35447             \tex_nolimits:D { \tex_nolimits:D }
35448             \tex_displaylimits:D { \tex_displaylimits:D }
35449         }
35450         { \__color_math_scan:w \use_none:n }
35451         { \__color_math_scan_auxii: }
35452     }
35453 }
```

The one final case to handle is math-active tokens, most obviously `,`, as these won't be covered earlier.

```

35454 \cs_new_protected:Npn \__color_math_scan_auxii:
35455 {
35456     \tl_map_inline:Nn \l_color_math_active_tl
35457     {
35458         \token_if_eq_meaning:NNT \l_peek_token ##1
35459         {
35460             \tl_map_break:n
35461             {
35462                 \use_i:nn
35463                 { \__color_math_scan_auxiii:N ##1 }
35464             }
35465         }
35466         \__color_math_scan_end:
35467     }
35468 }
```

```

35469 \cs_new_protected:Npn \__color_math_scan_auxiii:N #1
35470  {
35471      \exp_after:wN \exp_after:wN \exp_after:wN \__color_math_scan:w
35472      \char_generate:nn { '#1 } { 13 }
35473  }
35474 \cs_new_protected:Npn \__color_math_scan_end:
35475  {
35476      \__color_backend_reset:
35477      \seq_gpop:NN \g_color_math_seq \l__color_current_tl
35478  }

```

(End definition for `\__color_math_scan:w` and others.)

`\__color_math_scripts:Nw`  
`\__color_math_script_aux:N`

The tricky part of handling sub and superscripts is that we have to reset color to the one that is on the stack but reset it back to what it was before to allow for cases like

```
\[ \color{red} { a + \sum } _{ i = 1 } ^{ n } ]
```

Here, TeX constructs a `\vbox` stacking subscript, summation sign, and superscript. So technically the superscript comes first and the `\sum` that should get colored red is the middle.

The approach here is to set up a brace group immediately after the script token, then to set the color appropriately in that argument. We need an extra group to keep the color contained, and as we need to allow for an explicit closing brace in the source, the inner group also is a brace one rather than `\group_begin:-based`. At the end of the outer group we need to insert `\__color_math_scan:w` to continue the search for a second script token.

Notice that here we *don't* need to use the math-specific color selector as we can allow the `\group_insert_after:N \@@backend_reset:` to operate normally.

```

35479 \cs_new_protected:Npn \__color_math_scripts:Nw #1
35480  {
35481      #1
35482      \c_group_begin_token
35483      \c_group_begin_token
35484      \seq_get:NN \g_color_math_seq \l__color_current_tl
35485      \__color_select:N \l__color_current_tl
35486      \group_insert_after:N \c_group_end_token
35487      \group_insert_after:N \__color_math_scan:w
35488      \peek_remove_filler:n
35489      {
35490          \peek_catcode_remove:NF \c_group_begin_token
35491          { \__color_math_script_aux:N }
35492      }
35493  }

```

Deal with the case where we do not have an explicit brace pair in the source.

```
35494 \cs_new_protected:Npn \__color_math_script_aux:N #1 { #1 \c_group_end_token }
```

(End definition for `\__color_math_scripts:Nw` and `\__color_math_script_aux:N`.)

## 87.9 Fill and stroke color

```
\color_fill:n
\color_stroke:n
\color_fill:nn
\color_stroke:nn
\__color_draw:nnn
35495 \cs_new_protected:Npn \color_fill:n #1
35496 {
35497     \__color_parse:nN {#1} \l__color_current_tl
35498     \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
35499 }
35500 \cs_new_protected:Npn \color_stroke:n #1
35501 {
35502     \__color_parse:nN {#1} \l__color_current_tl
35503     \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
35504 }
35505 \cs_new_protected:Npn \color_fill:nn #1#2
35506 {
35507     \__color_select_main:Nw \l__color_current_tl
35508     #1 / / \s__color_mark #2 / / \s__color_stop
35509     \exp_after:wN \__color_draw:nnn \l__color_current_tl { fill }
35510 }
35511 \cs_new_protected:Npn \color_stroke:nn #1#2
35512 {
35513     \__color_select_main:Nw \l__color_current_tl
35514     #1 / / \s__color_mark #2 / / \s__color_stop
35515     \exp_after:wN \__color_draw:nnn \l__color_current_tl { stroke }
35516 }
35517 \cs_new_protected:Npn \__color_draw:nnn #1#2#3
35518 {
35519     \use:c { __color_backend_ #3 _ #1 :n } {#2}
35520     \exp_args:Nc \group_insert_after:N { __color_backend_ #3 _ reset: }
35521 }
```

(End definition for `\color_fill:n` and others. These functions are documented on page 295.)

## 87.10 Defining named colors

`\l__color_named_tl` Space to store the detail of the named color.

```
35522 \tl_new:N \l__color_named_tl
```

(End definition for `\l__color_named_tl`.)

`\color_set:nn` Defining named colors means working through the model list and saving both the “main” color and any equivalents in other models. Even if there is only one model, we store a `prop` as well as a `tl`, as there could be grouping weirdness, etc. When setting using an expression, we need to avoid any fixed model issues, which is done without a group as in `l3keys`.

```
\__color_set_aux:nnn
\__color_set_colon:nw
\__color_set_loop:nw
\color_set_eq:nn
35523 \cs_new_protected:Npn \color_set:nn #1#2
35524 {
35525     \exp_args:NV \__color_set:nnn
35526     \l_color_fixed_model_tl {#1} {#2}
35527 }
35528 \cs_new_protected:Npn \__color_set:nnn #1#2#3
35529 {
35530     \tl_clear:N \l_color_fixed_model_tl
```

```

35531     \__color_set:nn {#2} {#3}
35532     \tl_set:Nn \l_color_fixed_model_tl {#1}
35533   }
35534 \cs_new_protected:Npn \__color_set:nn #1#2
35535   {
35536     \str_if_eq:nnF {#1} { . }
35537   {
35538     \__color_parse:nN {#2} \l__color_named_tl
35539     \tl_clear_new:c { l__color_named_ #1 _tl }
35540     \tl_set:cx { l__color_named_ #1 _tl }
35541       { \__color_model:N \l__color_named_tl }
35542     \prop_clear_new:c { l__color_named_ #1 _prop }
35543     \prop_put:cxx { l__color_named_ #1 _prop } { l__color_named_ #1 _tl }
35544       { \__color_values:N \l__color_named_tl }
35545     \__color_set:nnw {#1} {#2} #2 ! \s__color_stop
35546   }
35547 }

```

When setting an expression-based color, there could be multiple model data available for one or more of the input colors. Where that is true for the *first* named color in an expression, we re-parse the expression when they are also parameter-based: only `cmyk`, `gray` and `rgb` make any sense here. There is a bit of a performance hit but this should be rare and taking place during set-up.

```

35548 \cs_new_protected:Npn \__color_set:nnw #1#2#3 ! #4 \s__color_stop
35549   {
35550     \clist_map_inline:nn { cmyk , gray , rgb }
35551   {
35552     \prop_get:cnNT { l__color_named_ #3 _prop } {##1} \l__color_internal_tl
35553   {
35554     \prop_if_in:cnF { l__color_named_ #1 _prop } {##1}
35555   {
35556     \group_begin:
35557       \tl_set:cn { l__color_named_ #3 _tl } {##1}
35558         \__color_parse:nN {#2} \l__color_internal_tl
35559       \exp_args:NNNV \group_end:
35560       \tl_set:Nn \l__color_internal_tl \l__color_internal_tl
35561       \prop_put:cxx { l__color_named_ #1 _prop }
35562         { \__color_model:N \l__color_internal_tl }
35563         { \__color_values:N \l__color_internal_tl }
35564   }
35565   }
35566 }
35567 }
35568 \cs_new_protected:Npn \color_set:nnn #1#2#3
35569   {
35570     \str_if_eq:nnF {#1} { . }
35571   {
35572     \tl_clear_new:c { l__color_named_ #1 _tl }
35573     \prop_clear_new:c { l__color_named_ #1 _prop }
35574     \exp_args:Ne \__color_set_aux:nnn { \tl_to_str:n {#2} }
35575       {#1} {#3}
35576   }
35577 }
35578 \cs_new_protected:Npx \__color_set_aux:nnn #1#2#3

```

```

35579    {
35580      \exp_not:N \__color_set_colon:nw {#2} {#3}
35581        #1 \cColonStr \cColonStr \exp_not:N \sColorStop
35582    }
35583 \use:x
35584  {
35585    \cs_new_protected:Npn \exp_not:N \__color_set_colon:nw
35586      ##1##2 ##3 \cColonStr ##4 \cColonStr
35587      ##5 \exp_not:N \sColorStop
35588  }
35589  {
35590    \tl_if_blank:nTF {#4}
35591      { \__color_set_loop:nw {#1} #3 }
35592      { \__color_set_loop:nw {#1} #4 }
35593      / / \sColorMark #2 / / \sColorStop
35594  }
35595 \cs_new_protected:Npn \__color_set_loop:nw
35596  #1#2 / #3 \sColorMark #4 / #5 \sColorStop
35597  {
35598    \tl_if_blank:nF {#2}
35599    {
35600      \__color_select:nnN {#2} {#4} \lColorNamedTl
35601      \tl_set:Nx \lColorInternalTl { \__color_model:N \lColorNamedTl }
35602      \tl_if_empty:cT { \lColorNamed_ #1 _tl }
35603        { \tl_set_eq:cN { \lColorNamed_ #1 _tl } \lColorInternalTl }
35604        \prop_put:cVx { \lColorNamed_ #1 _prop } \lColorInternalTl
35605        { \__color_values:N \lColorNamedTl }
35606        \__color_set_loop:nw {#1} #3 \sColorMark #5 \sColorStop
35607    }
35608  }
35609 \cs_new_protected:Npn \color_set_eq:nn #1#2
35610  {
35611    \color_if_exist:nTF {#2}
35612    {
35613      \tl_clear_new:c { \lColorNamed_ #1 _tl }
35614      \prop_clear_new:c { \lColorNamed_ #1 _prop }
35615      \str_if_eq:nnTF {#2} { . }
35616      {
35617        \tl_set:cx { \lColorNamed_ #1 _tl }
35618          { \__color_model:N \lColorCurrentTl }
35619        \prop_put:cVx { \lColorNamed_ #1 _prop } { \lColorNamed_ #1 _tl }
35620          { \__color_values:N \lColorCurrentTl }
35621      }
35622      {
35623        \tl_set_eq:cc { \lColorNamed_ #1 _tl } { \lColorNamed_ #2 _tl }
35624        \prop_set_eq:cc { \lColorNamed_ #1 _prop } { \lColorNamed_ #2 _prop }
35625      }
35626    }
35627    {
35628      \msg_error:nnn { color } { unknown-color } {#2}
35629    }
35630  }

```

(End definition for `\color_set:nn` and others. These functions are documented on page 294.)

A small set of colors are always defined.

```
35631 \color_set:nnn { black } { gray } { 0 }
35632 \color_set:nnn { white } { gray } { 1 }
35633 \color_set:nnn { cyan } { cmyk } { 1 , 0 , 0 , 0 }
35634 \color_set:nnn { magenta } { cmyk } { 0 , 1 , 0 , 0 }
35635 \color_set:nnn { yellow } { cmyk } { 0 , 0 , 1 , 0 }
35636 \color_set:nnn { red } { rgb } { 1 , 0 , 0 }
35637 \color_set:nnn { green } { rgb } { 0 , 1 , 0 }
35638 \color_set:nnn { blue } { rgb } { 0 , 0 , 1 }
```

\l\_color\_named\_.prop A special named color: this is always defined though not fixed in definition.

```
\l_color_named_.tl
35639 \prop_new:c { l_color_named_.prop }
35640 \tl_new:c { l_color_named_.tl }
35641 \tl_set:cx { l_color_named_.tl } { \__color_model:N \l_color_current_tl }
```

(End definition for \l\_color\_named\_.prop and \l\_color\_named\_.tl.)

## 87.11 Exporting colors

```
\color_export:nnN
\color_export:nnnN
\__color_export:nN
\__color_export:nnnN
35642 \cs_new_protected:Npn \color_export:nnN #1#2#3
35643 {
35644     \group_begin:
35645         \tl_if_exist:cT { c_color_export_ #2 _tl }
35646             { \tl_set_eq:Nc \l_color_fixed_model_tl { c_color_export_ #2 _tl } }
35647             \__color_parse:nN {#1} #3
35648             \__color_export:nN {#2} #3
35649         \exp_args:NNNV \group_end:
35650         \tl_set:Nn #3 #3
35651     }
35652 \cs_new_protected:Npn \color_export:nnnN #1#2#3#4
35653 {
35654     \__color_select_main:Nw #4
35655         #1 / / \s_color_mark #2 / / \s_color_stop
35656         \__color_export:nN {#3} #4
35657     }
35658 \cs_new_protected:Npn \__color_export:nN #1#2
35659     { \exp_after:wN \__color_export:nnnN #2 {#1} #2 }
35660 \cs_new:Npn \__color_export:nnnN #1#2#3#4
35661 {
35662     \cs_if_exist_use:cF { __color_export_format_ #3 :nnN }
35663     {
35664         \msg_error:nnn { color } { unknown-export-format } {#3}
35665         \use_none:nnn
35666     }
35667     {#1} {#2} #4
35668 }
```

(End definition for \color\_export:nnN and others. These functions are documented on page 296.)

\\_\_color\_export\_format\_backend:nnN Simple.

```
35669 \cs_new_protected:Npn \__color_export_format_backend:nnN #1#2#3
35670     { \tl_set:Nn #3 { {#1} {#2} } }
```

(End definition for `\__color_export_format_backend:nnN`.)

`\__color_export:nnNN` A generic auxiliary for cases where only one model is appropriate.

```
35671 \cs_new_protected:Npn \__color_export:nnNN #1#2#3#4#5
35672 {
35673     \str_if_eq:nnTF {#2} {#1}
35674         { #5 #4 #3 \s__color_stop }
35675     {
35676         \__color_convert:nnnN {#2} {#1} {#3} #4
35677         \exp_after:wN #5 \exp_after:wN #4
35678             #4 \s__color_stop
35679     }
35680 }
```

(End definition for `\__color_export:nnNN`.)

`\c_color_export_comma-sep-cmyk_tl`

`\c_color_export_comma-sep-rgb_tl`

`\c_color_export_HTML_tl`

`\c_color_export_space-sep-cmyk_tl`

`\c_color_export_space-sep-rgb_tl`

```
35681 \tl_const:cn { c_color_export_comma-sep-cmyk_tl } { cmyk }
35682 \tl_const:cn { c_color_export_comma-sep-rgb_tl } { rgb }
35683 \tl_const:Nn \c_color_export_HTML_tl { rgb }
35684 \tl_const:cn { c_color_export_space-sep-cmyk_tl } { cmyk }
35685 \tl_const:cn { c_color_export_space-sep-rgb_tl } { rgb }
```

(End definition for `\c_color_export_comma-sep-cmyk_tl` and others.)

`\__color_export_format_comma-sep-cmyk:nnN`

`\__color_export_format_comma-sep-rgb:nnN`

`\__color_export_format_space-sep-cmyk:nnN`

`\__color_export_format_space-sep-rgb:nnN`

```
35686 \group_begin:
35687     \cs_set_protected:Npn \__color_tmp:w #1#2
35688     {
35689         \cs_new_protected:cpn { __color_export_format_ #1 :nnN } ##1##2##3
35690         {
35691             \exp_not:N \__color_export:nnnN {#2} {##1} {##2} ##3
35692                 \exp_not:c { __color_export_ #1 :Nw }
35693         }
35694     }
35695     \__color_tmp:w { comma-sep-cmyk } { cmyk }
35696     \__color_tmp:w { comma-sep-rgb } { rgb }
35697     \__color_tmp:w { HTML } { rgb }
35698     \__color_tmp:w { space-sep-cmyk } { cmyk }
35699     \__color_tmp:w { space-sep-rgb } { rgb }
35700
35701 \group_end:
```

(End definition for `\__color_export_format_comma-sep-cmyk:nnN` and others.)

`\__color_export_space-sep-cmyk:Nw`

`\__color_export_comma-sep-cmyk:Nw`

```
35702 \cs_new_protected:cpn { __color_export_comma-sep-cmyk:Nw }
35703     #1#2 ~ #3 ~ #4 ~ #5 \s__color_stop
35704     { \tl_set:Nn #1 { #2 , #3 , #4 , #5 } }
35705 \cs_new_protected:cpn { __color_export_space-sep-cmyk:Nw } #1#2 \s__color_stop
35706     { \tl_set:Nn #1 {#2} }
```

(End definition for `\__color_export_space-sep-cmyk:Nw` and `\__color_export_comma-sep-cmyk:Nw`.)

```

\__color_export_comma-sep-rgb:Nw
\__color_export_HTML:Nw
\__color_export_space-sep-rgb:Nw
\__color_export_HTML:n

35707 \cs_new_protected:cpn { __color_export_comma-sep-rgb:Nw } #1#2 ~ #3 ~ #4 \s__color_stop
35708   { \tl_set:Nx #1 { #2 , #3 , #4 } }
35709   \cs_new_protected:Npn \__color_export_HTML:Nw #1#2 ~ #3 ~ #4 \s__color_stop
35710   {
35711     \tl_set:Nx #1
35712     {
35713       \__color_export_HTML:n {#2}
35714       \__color_export_HTML:n {#3}
35715       \__color_export_HTML:n {#4}
35716     }
35717   }
35718 \cs_new:Npn \__color_export_HTML:n #1
35719   {
35720     \fp_compare:nNnTF {#1} = { 0 }
35721     { 00 }
35722     {
35723       \fp_compare:nNnT { #1 * 255 } < { 16 } { 0 }
35724       \int_to_Hex:n { \fp_to_int:n { #1 * 255 } }
35725     }
35726   }
35727 \cs_new_protected:cpn { __color_export_space-sep-rgb:Nw } #1#2 \s__color_stop
35728   { \tl_set:Nn #1 {#2} }

(End definition for \__color_export_comma-sep-rgb:Nw and others.)

```

## 87.12 Additional color models

\l\_\_color\_internal\_prop

```
35729 \prop_new:N \l__color_internal_prop
```

(End definition for \l\_\_color\_internal\_prop.)

\g\_\_color\_model\_int

A tracker for the total number of new models.

```
35730 \int_new:N \g__color_model_int
```

(End definition for \g\_\_color\_model\_int.)

\c\_\_color\_fallback\_cmyk\_tl

For every colorspace, we define one of the base colorspaces as a fallback. The base colorspaces themselves are their own fallback.

```
35731 \tl_const:Nn \c__color_fallback_cmyk_tl { cmyk }
35732 \tl_const:Nn \c__color_fallback_gray_tl { gray }
35733 \tl_const:Nn \c__color_fallback_rgb_tl { rgb }
```

(End definition for \c\_\_color\_fallback\_cmyk\_tl, \c\_\_color\_fallback\_gray\_tl, and \c\_\_color\_fallback\_rgb\_tl.)

\g\_\_color\_colorants\_prop

Mapping from names to colorants.

```
35734 \prop_new:N \g__color_colorants_prop
35735 \prop_gput:Nnn \g__color_colorants_prop { black } { Black }
35736 \prop_gput:Nnn \g__color_colorants_prop { blue } { Blue }
35737 \prop_gput:Nnn \g__color_colorants_prop { cyan } { Cyan }
35738 \prop_gput:Nnn \g__color_colorants_prop { green } { Green }
```

```

35739 \prop_gput:Nnn \g_color_colorants_prop { magenta } { Magenta }
35740 \prop_gput:Nnn \g_color_colorants_prop { none } { None }
35741 \prop_gput:Nnn \g_color_colorants_prop { red } { Red }
35742 \prop_gput:Nnn \g_color_colorants_prop { yellow } { Yellow }

```

(End definition for `\g_color_colorants_prop`.)

Whitepoint data for the CIELAB profiles.

```

35743 \tl_const:Nn \c_color_model_whitepoint_CIELAB_a_tl { 1.0985 ~ 1 ~ 0.3558 }
35744 \tl_const:Nn \c_color_model_whitepoint_CIELAB_b_tl { 0.9807 ~ 1 ~ 1.1822 }
35745 \tl_const:Nn \c_color_model_whitepoint_CIELAB_e_tl { 1 ~ 1 ~ 1 }
35746 \tl_const:cn { c_color_model_whitepoint_CIELAB_d50_tl } { 0.9642 ~ 1 ~ 0.8251 }
35747 \tl_const:cn { c_color_model_whitepoint_CIELAB_d55_tl } { 0.9568 ~ 1 ~ 0.9214 }
35748 \tl_const:cn { c_color_model_whitepoint_CIELAB_d65_tl } { 0.9504 ~ 1 ~ 1.0888 }
35749 \tl_const:cn { c_color_model_whitepoint_CIELAB_d75_tl } { 0.9497 ~ 1 ~ 1.2261 }

```

(End definition for `\c_color_model_whitepoint_CIELAB_a_tl` and others.)

`\c_color_model_range_CIELAB_t1`

The range for CIELAB color spaces.

```
35750 \tl_const:Nn \c_color_model_range_CIELAB_t1 { 0 ~ 100 ~ -128 ~ 127 ~ -128 ~ 127 }
```

(End definition for `\c_color_model_range_CIELAB_t1`.)

`\g_color_alternative_model_prop`

For tracking the alternative model set up for separations, etc.

```

35751 \prop_new:N \g_color_alternative_model_prop
35752 \clist_map_inline:nn { cyan , magenta , yellow , black }
35753 { \prop_gput:Nnn \g_color_alternative_model_prop {#1} { cmyk } }
35754 \clist_map_inline:nn { red , green , blue }
35755 { \prop_gput:Nnn \g_color_alternative_model_prop {#1} { rgb } }

```

(End definition for `\g_color_alternative_model_prop`.)

`\g_color_alternative_values_prop`

Same for the values: a bit more involved.

```

35756 \prop_new:N \g_color_alternative_values_prop
35757 \prop_gput:Nnn \g_color_alternative_values_prop { cyan } { 1 , 0 , 0 , 0 }
35758 \prop_gput:Nnn \g_color_alternative_values_prop { magenta } { 0 , 1 , 0 , 0 }
35759 \prop_gput:Nnn \g_color_alternative_values_prop { yellow } { 0 , 0 , 1 , 0 }
35760 \prop_gput:Nnn \g_color_alternative_values_prop { black } { 0 , 0 , 0 , 1 }
35761 \prop_gput:Nnn \g_color_alternative_values_prop { red } { 1 , 0 , 0 }
35762 \prop_gput:Nnn \g_color_alternative_values_prop { green } { 0 , 1 , 0 }
35763 \prop_gput:Nnn \g_color_alternative_values_prop { blue } { 0 , 0 , 1 }

```

(End definition for `\g_color_alternative_values_prop`.)

`\color_model_new:nnn`  
`\__color_model_new:nnn`

Set up a new model: in general this has to be handled by a family-dependent function. To avoid some “interesting” questions with casing, we fold the case of the family name. The key-value list should always be present, so we convert it up-front to a `prop`, then deal with the detail on a per-family basis.

```

35764 \cs_new_protected:Npn \color_model_new:nnn #1#2#3
35765 {
35766 \exp_args:Nee \__color_model_new:nnn
35767 { \tl_to_str:n {#1} }
35768 { \str_casfold:n {#2} } {#3}
35769 }
35770 \cs_new_protected:Npn \__color_model_new:nnn #1#2#3

```

```

35771   {
35772     \cs_if_exist:cTF { __color_parse_model_ #1 :w }
35773     {
35774       \msg_error:nnn { color } { model-already-defined } {#1}
35775     }
35776     {
35777       \cs_if_exist:cTF { __color_model_ #2 :n }
35778         {
35779           \prop_set_from_keyval:Nn \l__color_internal_prop {#3}
35780           \use:c { __color_model_ #2 :n } {#1}
35781         }
35782         {
35783           \msg_error:nnn { color } { unknown-model-type } {#2}
35784         }
35785     }
35786   }

```

(End definition for `\color_model_new:nnn` and `\__color_model_new:nnn`. This function is documented on page 297.)

`\__color_model_init:nnn`  
`\__color_model_init:nnx`

A shared auxiliary to do the basics of setting up a new model: reserve a number, create a white-equivalent, set up links to the backend.

```

35787 \cs_new_protected:Npn \__color_model_init:nnn #1#2#3
35788   {
35789     \int_gincr:N \g__color_model_int
35790     \clist_map_inline:nn { fill , stroke , select }
35791       {
35792         \cs_new_protected:cp { __color_backend_ ##1 _ #1 :n } #####
35793           {
35794             \exp_not:c { __color_backend_ ##1 _ #2 :nn }
35795               { color \int_use:N \g__color_model_int } #####
35796           }
35797       }
35798     \cs_new_protected:cp { __color_model_ #1 _white: }
35799       {
35800         \prop_put:Nnn \exp_not:N \l__color_named_white_prop {#1}
35801           { \exp_not:n {#3} }
35802         \exp_not:N \int_compare:nNnF { \tex_currentgrouplevel:D } = 0
35803           { \group_insert_after:N \exp_not:c { __color_model_ #1 _white: } }
35804       }
35805     \use:c { __color_model_ #1 _white: }
35806   }
35807 \cs_generate_variant:Nn \__color_model_init:nnn { nnx }

(End definition for \__color_model_init:nnn.)

```

`\__color_model_separation:n`  
`\__color_model_separation:nn`  
`\__color_model_separation:nnn`

Separtions must have a “real” name, which is pretty easy to find.

```

35808 \cs_new_protected:Npn \__color_model_separation:n #1
35809   {
35810     \prop_get:NnNTF \l__color_internal_prop { name }
35811       \l__color_internal_tl
35812       {
35813         \exp_args:NV \__color_model_separation:nn
35814           \l__color_internal_tl {#1}
35815       }

```

```

35816   {
35817     \msg_error:nnn { color }
35818       { separation-requires-name } {#1}
35819   }
35820 }

```

We have two keys to find at this stage: the alternative space model and linked values.

```

35821 \cs_new_protected:Npn \__color_model_separation:nn #1#2
35822 {
35823   \prop_get:NnNTF \l__color_internal_prop { alternative-model }
35824     \l__color_internal_tl
35825   {
35826     \exp_args:NV \__color_model_separation:nnn
35827       \l__color_internal_tl {#2} {#1}
35828   }
35829   {
35830     \msg_error:nnn { color }
35831       { separation-alternative-model } {#2}
35832   }
35833 }
35834 \cs_new_protected:Npn \__color_model_separation:nnn #1#2#3
35835 {
35836   \cs_if_exist:cTF { __color_model_separation_ #1 :nnnnnn }
35837   {
35838     \prop_get:NnNTF \l__color_internal_prop { alternative-values }
35839       \l__color_internal_tl
35840     {
35841       \exp_after:wN \__color_model_separation:w \l__color_internal_tl
35842         , 0 , 0 , 0 , 0 \s__color_stop {#2} {#3} {#1}
35843     }
35844     {
35845       \msg_error:nnn { color }
35846         { separation-alternative-values } {#2}
35847     }
35848   }
35849   {
35850     \msg_error:nnn { color }
35851       { unknown-alternative-model } {#1}
35852   }
35853 }

```

As each alternative space leads to a different requirement for conversion, and as there are only a small number of choices, we manually split the data and then set up. Notice that mixing tints is really just the same as mixing `gray`. The `white` color is special, as it allows tints to be adjusted without an additional color space. To make sure the data is set for that at all group levels, we need to work on a per-level basis. Within the output, only the set-up needs the “real” name of the colorspace: we use a simple tracking number for general usage as this is a clear namespace without issues of escaping chars.

```

35854 \cs_new_protected:Npn \__color_model_separation:w
35855   #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7#8
35856   {
35857     \__color_model_init:nnn {#6} { separation } { 0 }
35858     \cs_new_eq:cN { __color_parse_mix_ #6 :nw } \__color_parse_mix_gray:nw
35859     \cs_new:cN { __color_parse_model_ #6 :w } ##1 , ##2 \s__color_stop

```

```

35860      { {#6} { \__color_parse_number:n {##1} } }
35861      \use:c { __color_model_separation_ #8 :nnnnnn }
35862          {#6} {#7} {#1} {#2} {#3} {#4}
35863      \prop_gput:Nnn \g__color_alternative_model_prop {#6} {#8}
35864      \prop_gput:Nnx \g__color_colorants_prop {#6}
35865          { \str_convert_pdfname:n {#7} }
35866      }
35867 \cs_new_protected:Npn \__color_model_separation_cmyk:nnnnnn #1#2#3#4#5#6
35868  {
35869      \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
35870      \cs_new:cpn { __color_convert_ #1 _cmyk:w } ##1 \s__color_stop
35871          {
35872              \fp_eval:n {##1 * #3} ~
35873              \fp_eval:n {##1 * #4} ~
35874              \fp_eval:n {##1 * #5} ~
35875              \fp_eval:n {##1 * #6}
35876          }
35877      \cs_new:cpn { __color_convert_cmyk_ #1 :w } ##1 \s__color_stop { 1 }
35878      \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 , #6 }
35879      \__color_backend_separation_init:nnnnn {#2} { /DeviceCMYK } { }
35880          { 0 ~ 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 ~ #6 }
35881  }
35882 \cs_new_protected:Npn \__color_model_separation_rgb:nnnnnn #1#2#3#4#5#6
35883  {
35884      \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
35885      \cs_new:cpn { __color_convert_ #1 _rgb:w } ##1 \s__color_stop
35886          {
35887              \fp_eval:n {##1 * #3} ~
35888              \fp_eval:n {##1 * #4} ~
35889              \fp_eval:n {##1 * #5}
35890          }
35891      \cs_new:cpn { __color_convert_rgb_ #1 :w } ##1 \s__color_stop { 1 }
35892      \prop_gput:Nnn \g__color_alternative_values_prop {#1} { #3 , #4 , #5 }
35893      \__color_backend_separation_init:nnnnn {#2} { /DeviceRGB } { }
35894          { 0 ~ 0 ~ 0 } { #3 ~ #4 ~ #5 }
35895  }
35896 \cs_new_protected:Npn \__color_model_separation_gray:nnnnnn #1#2#3#4#5#6
35897  {
35898      \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
35899      \cs_new:cpn { __color_convert_ #1 _gray:w } ##1 \s__color_stop
35900          { \fp_eval:n {##1 * #3} }
35901      \cs_new:cpn { __color_convert_gray_ #1 :w } ##1 \s__color_stop { 1 }
35902      \prop_gput:Nnn \g__color_alternative_values_prop {#1} {#3}
35903      \__color_backend_separation_init:nnnnn {#2} { /DeviceGray } { } { 0 } {#3}
35904  }

```

Generic model conversion *via* an alternative intermediate.

```

35905 \cs_new_protected:Npn \__color_model_convert:nnn #1#2#3
35906  {
35907      \cs_new:cpx { __color_convert_ #1 _ #3 :w } ##1 \s__color_stop
35908          {
35909              \exp_not:N \exp_args:NNe \exp_not:N \use:nn
35910              \exp_not:c { __color_convert_ #2 _ #3 :w }
35911                  { \exp_not:c { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop }
35912              \c_space_tl \exp_not:N \s__color_stop

```

```

35913     }
35914 }
```

Setting up for CIELAB needs a bit more work: there is the illuminant and the need for an appropriate object.

```

35915 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnn #1#2#3#4#5#6
35916 {
35917     \prop_get:NnNF \l__color_internal_prop { illuminant }
35918         \l__color_internal_tl
35919     {
35920         \msg_error:nnn { color }
35921             { CIELAB-requires-illuminant } {#1}
35922             \tl_set:Nn \l__color_internal_tl { d50 }
35923     }
35924     \exp_args:NV \__color_model_separation_CIELAB:nnnnnn
35925         \l__color_internal_tl {#1} {#2} {#3} {#4} {#5} {#6}
35926 }
```

If a CIELAB space is being set up, we need the illuminant, then create the appropriate set up. At present, this doesn't include `BlackPoint` or `Range` data, but that may be added later. As CIELAB colors cannot be converted to anything else, we fallback to producing black in the gray colorspace: the user should set up a second model for colors set up this way.

```

35927 \cs_new_protected:Npn \__color_model_separation_CIELAB:nnnnnn #1#2#3#4#5#6#7
35928 {
35929     \tl_if_exist:cTF { c__color_model_whitepoint_CIELAB_ #1 _tl }
35930     {
35931         \__color_backend_separation_init_CIELAB:nnn {#1} {#3} { #4 ~ #5 ~ #6 }
35932         \tl_const:cn { c__color_fallback_ #2 _tl } { gray }
35933         \cs_new:cpn { __color_convert_ #2 _gray:w } ##1 \s__color_stop
35934             { 0 }
35935         \cs_new:cpn { __color_convert_gray_ #2 :w } ##1 \s__color_stop
35936             { 1 }
35937     }
35938     {
35939         \msg_error:nnn { color }
35940             { unknown-CIELAB-illuminant } {#1}
35941     }
35942 }
```

(End definition for `\__color_model_separation:n` and others.)

We require a list of component names here: one might call them colorants, but it's convenient to use TeX names instead so we slightly adjust the terminology.

```

35943 \cs_new_protected:Npn \__color_model_devicen:n #1
35944 {
35945     \prop_get:NnTF \l__color_internal_prop { names }
35946         \l__color_internal_tl
35947     {
35948         \exp_args:NV \__color_model_devicen:nn
35949             \l__color_internal_tl {#1}
35950     }
35951     {
35952         \msg_error:nnn { color }
35953             { DeviceN-requires-names } {#1}
35954 }
```

```

35954     }
35955 }
```

All valid models will have an alternative listed, either hard-coded for the core device ones, or dynamically added for Separations, etc.

```

35956 \cs_new_protected:Npn \__color_model_devicen:nn #1#2
35957 {
35958     \tl_clear:N \l__color_model_tl
35959     \clist_map_inline:nn {#1}
35960     {
35961         \prop_get:NnNTF \g__color_alternative_model_prop {##1}
35962             \l__color_internal_tl
35963             {
35964                 \tl_if_empty:NTF \l__color_model_tl
35965                     { \tl_set_eq:NN \l__color_model_tl \l__color_internal_tl }
35966                     {
35967                         \str_if_eq:VVF \l__color_model_tl \l__color_internal_tl
35968                         {
35969                             \msg_error:nnn { color }
35970                                 { DeviceN-inconsistent-alternative }
35971                                 {##2}
35972                             \clist_map_break:n { \use_none:nnnn }
35973                         }
35974                     }
35975                 }
35976             {
35977                 \str_if_eq:nnF {##1} { none }
35978                 {
35979                     \msg_error:nnn { color }
35980                         { DeviceN-no-alternative }
35981                         {##2}
35982                     }
35983                 }
35984             }
35985         \tl_if_empty:NTF \l__color_model_tl
35986             {
35987                 \msg_error:nnn { color }
35988                     { DeviceN-no-alternative } {##2}
35989             }
35990             { \exp_args:NV \__color_model_devicen:nnn \l__color_model_tl {#1} {##2} }
35991         }
```

We now complete the data we require by first finding out how many colorants there are, then moving on to begin constructing the function required to map to the alternative color space.

```

35992 \cs_new_protected:Npn \__color_model_devicen:nnn #1#2#3
35993 {
35994     \exp_args:Nx \__color_model_devicen:nnnn
35995         { \clist_count:n {#2} } {#1} {#2} {#3}
35996 }
```

At this stage, we have checked everything is in place, so we can set up the T<sub>EX</sub> and backend data structures. As for separations, it's not really possible in general to have a fallback, so we simply provide "black" for each element.

```
35997 \cs_new_protected:Npn \__color_model_devicen:nnnn #1#2#3#4
```

```

35998  {
35999    \__color_model_init:nnx {#4} { devicen }
36000    {
36001      0 \prg_replicate:nn { #1 - 1 } { ~ 0 }
36002    }
36003    \cs_if_exist_use:cF { __color_model_devicen_parse_ #1 :nn }
36004    { \__color_model_devicen_parse_generic:nn }
36005    {#4} {#1}
36006    \__color_model_devicen_init:nnn {#1} {#2} {#3}
36007    \__color_model_devicen_convert:nnnx {#4} {#2} {#3}
36008    {
36009      1 \prg_replicate:nn { #1 - 1 } { ~ 1 }
36010    }
36011  }

```

For short lists of DeviceN colors, we can use hand-tuned parsing. This lines up with other models, where we allow for up to four components. For larger spaces, rather than limit artificially, we use a somewhat slow approach based on open-ended commas-lists.

```

36012 \cs_new_protected:cpn { __color_model_devicen_parse_1:nn } #1#2
36013  {
36014    \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s__color_stop
36015    { {#1} { \__color_parse_number:n {##1} } }
36016    \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_gray:nw
36017  }
36018 \cs_new_protected:cpn { __color_model_devicen_parse_2:nn } #1#2
36019  {
36020    \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 , ##3 \s__color_stop
36021    { {#1} { \__color_parse_number:n {##1} ~ \__color_parse_number:n {##2} } }
36022    \cs_new:cpn { __color_parse_mix_ #1 :nw }
36023    ##1##2 ~ ##3 \s__color_mark ##4 ~ ##5 \s__color_stop
36024    {
36025      \fp_eval:n { ##2 * ##1 + ##4 * ( 1 - ##1 ) } \c_space_tl
36026      \fp_eval:n { ##3 * ##1 + ##5 * ( 1 - ##1 ) }
36027    }
36028  }
36029 \cs_new_protected:cpn { __color_model_devicen_parse_3:nn } #1#2
36030  {
36031    \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 , ##3 , ##4 \s__color_stop
36032    {
36033      {#1}
36034      {
36035        \__color_parse_number:n {##1} ~
36036        \__color_parse_number:n {##2} ~
36037        \__color_parse_number:n {##3}
36038      }
36039    }
36040    \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_rgb:nw
36041  }
36042 \cs_new_protected:cpn { __color_model_devicen_parse_4:nn } #1#2
36043  {
36044    \cs_new:cpn { __color_parse_model_ #1 :w }
36045    ##1 , ##2 , ##3 , ##4 , ##5 \s__color_stop
36046    {
36047      {#1}

```

```

36048 {
36049     \__color_parse_number:n {##1} ~
36050     \__color_parse_number:n {##2} ~
36051     \__color_parse_number:n {##3} ~
36052     \__color_parse_number:n {##4}
36053 }
36054 }
36055 \cs_new_eq:cN { __color_parse_mix_ #1 :nw } \__color_parse_mix_cmyk:nw
36056 }
36057 \cs_new_protected:Npn \__color_model_devicen_parse_generic:nn #1#2
36058 {
36059     \cs_new:cpn { __color_parse_model_ #1 :w } ##1 , ##2 \s__color_stop
36060     {
36061         {#1}
36062         { \__color_model_devicen_parse:nw {#2} ##1 , ##2 , \q_nil , \s__color_stop }
36063     }
36064 \cs_new:cpx { __color_parse_mix_ #1 :nw }
36065     ##1 ##2 \s__color_mark ##3 \s__color_stop
36066     {
36067         \exp_not:N \__color_model_devicen_mix:nw {##1}
36068         ##2 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_mark
36069         ##3 \c_space_tl \exp_not:N \q_nil \c_space_tl \exp_not:N \s__color_stop
36070     }
36071 }
36072 \cs_new:Npn \__color_model_devicen_parse:nw #1#2 , #3 \s__color_stop
36073 {
36074     \int_compare:nNnT {#1} > 0
36075     {
36076         \quark_if_nil:nTF {#2}
36077             { \prg_replicate:nn {#1} { 0 ~ } }
36078         {
36079             \__color_parse_number:n {#2}
36080             \int_compare:nNnT {#1} > 1 { ~ }
36081             \exp_args:Nf \__color_model_devicen_parse:nw
36082                 { \int_eval:n { #1 - 1 } } #3 \s__color_stop
36083         }
36084     }
36085 }
36086 \cs_new:Npn \__color_model_devicen_mix:nw #1#2 ~ #3 \s__color_mark #4 ~ #5 \s__color_stop
36087 {
36088     \fp_eval:n { #2 * #1 + #4 * ( 1 - #1 ) }
36089     \quark_if_nil:oF { \tl_head:w #3 \q_stop }
36090     {
36091         \c_space_tl
36092         \__color_model_devicen_mix:nw {#1} #3 \s__color_mark #5 \s__color_stop
36093     }
36094 }

```

To construct the tint transformation, we have to use PostScript. The aim is to have the final tint for each device colorant as

$$1 - \prod_n (1 - X_n D_{X_n})$$

where  $X$  is a DeviceN colorant and  $D$  is the amount of device colorant that the DeviceN colorant maps to. At the start of the process, the PostScript stack will contain the

$X_n$  values, whilst we have the  $D$  values on a per-DeviceN colorant basis. The more convenient approach for us is therefore to take each DeviceN colorant in turn and find the value  $1 - X_n D_{X_n}$ , multiplying as we go, and finalise with the subtraction. That contrasts to `colorspace`: it splits the process up by process color, which works better when you have a fixed list of colorants. (`colorspace` only supports up to 4 DeviceN colors, and only `cmyk` as the alternative space.) To set this up, we first need to know the number of values in the target color space: this is easily handled as there are a very small range of possibles. Once we have that information, it's relatively easy to build the required PostScript using some generic code.

```

36095 \cs_new_protected:Npn \__color_model_devicen_init:nnn #1#2#3
36096 {
36097     \exp_args:Ne \__color_model_devicen_init:nnnn
36098     {
36099         \str_case:nn {#2}
36100         {
36101             { cmyk } { 4 }
36102             { gray } { 1 }
36103             { rgb } { 3 }
36104         }
36105     }
36106     {#1} {#2} {#3}
36107 }
```

As we always need to split the alternative values into parts, we use a shared auxiliary and only use a minimal difference between code paths. Construction of the tint transformation is as far as possible done using loops, which means there are some inefficiencies for device colors in the DeviceN space: we roll the stack one-at-a-time even if there is a potential shortcut. However, that way there is nothing to special-case. Once this is sorted, we can write the tint transform object, which will remain as the last object until we sort out the final step: the colorant list.

```

36108 \cs_new_protected:Npn \__color_model_devicen_init:nnnn #1#2#3#4
36109 {
36110     \tl_set:Nx \l__color_internal_tl
36111     { \prg_replicate:nn {#1} { 1.0 ~ } ~ }
36112     \int_zero:N \l__color_internal_int
36113     \clist_map_inline:nn {#4}
36114     {
36115         \int_incr:N \l__color_internal_int
36116         \prop_get:NnN \g__color_alternative_values_prop {##1}
36117             \l__color_value_tl
36118         \exp_after:wN \__color_model_devicen_transform:w
36119             \l__color_value_tl , 0 , 0 , 0 , \s__color_stop {#1} {#2}
36120     }
36121     \tl_put_right:Nx \l__color_internal_tl
36122     {
36123         \prg_replicate:nn {#1}
36124             { neg ~ 1.0 ~ add ~ #1 ~ -1 ~ roll ~ }
36125             \int_eval:n { #2 + #1 } ~ #1 ~ roll
36126             \prg_replicate:nn {#2} { ~ pop } ~
36127             #1 ~ 1 ~ roll
36128     }
36129     \use:x
36130     {
```

```

36131           \__color_backend_devicen_init:nnn
36132           {
36133               \clist_map_function:nN {#4}
36134                   \__color_model_devicen_colorant:n
36135           }
36136           {
36137               \str_case:nn {#3}
36138               {
36139                   { cmyk } { /DeviceCMYK }
36140                   { gray } { /DeviceGray }
36141                   { rgb } { /DeviceRGB }
36142               }
36143           }
36144           { \exp_not:V \l__color_internal_tl }
36145       }
36146   }
36147 \cs_new_protected:Npn \__color_model_devicen_transform:w
36148     #1 , #2 , #3 , #4 , #5 \s__color_stop #6#7
36149   {
36150       \use:c { __color_model_devicen_transform_ #6 :nnnnn }
36151           {#1} {#2} {#3} {#4} {#7}
36152   }
36153 \cs_new_protected:cpn { __color_model_devicen_transform_1:nnnnn } #1#2#3#4#5
36154     { \__color_model_devicen_transform:nnn {#5} { 1 } {#1} }
36155 \cs_new_protected:cpn { __color_model_devicen_transform_3:nnnnn } #1#2#3#4#5
36156   {
36157       \clist_map_inline:nn { #1 , #2 , #3 }
36158           { \__color_model_devicen_transform:nnn {#5} { 3 } {##1} }
36159   }
36160 \cs_new_protected:cpn { __color_model_devicen_transform_4:nnnnn } #1#2#3#4#5
36161   {
36162       \clist_map_inline:nn { #1 , #2 , #3 , #4 }
36163           { \__color_model_devicen_transform:nnn {#5} { 4 } {##1} }
36164   }
36165 \cs_new_protected:Npn \__color_model_devicen_transform:nnn #1#2#3
36166   {
36167       \tl_put_right:Nx \l__color_internal_tl
36168   {
36169       \fp_compare:nNnF {#3} = \c_zero_fp
36170   {
36171       \int_eval:n { #1 - \l__color_internal_int + #2 } ~ index ~
36172           -#3 ~ mul ~ 1.0 ~ add ~ mul ~
36173       }
36174       #2 ~ -1 ~ roll ~
36175   }
36176   }
36177 \cs_new:Npn \__color_model_devicen_colorant:n #1
36178   {
36179       / \prop_item:Nn \g__color_colorants_prop {#1} ~
36180   }

```

Here we need to set up conversion from the DeviceN space to the alternative at the TeX level. This also means supplying methods for inter-converting to other parameter-based spaces. Essentially the approach is exactly the same as the PostScript, just expressed in

T<sub>E</sub>X terms.

```

36181 \cs_new_protected:Npn \__color_model_devicen_convert:nnnn #1#2#3
36182   {
36183     \use:c { __color_model_devicen_convert_ #2 :nnn } {#1} {#3}
36184   }
36185 \cs_generate_variant:Nn \__color_model_devicen_convert:nnnn { nnx }
36186 \cs_new_protected:Npn \__color_model_devicen_convert_cmyk:nnn #1#2
36187   {
36188     \tl_const:cn { c__color_fallback_ #1 _tl } { cmyk }
36189     \__color_model_devicen_convert:nnnn {#1} { cmyk } { 4 } {#2}
36190   }
36191 \cs_new_protected:Npn \__color_model_devicen_convert_gray:nnn #1#2
36192   {
36193     \tl_const:cn { c__color_fallback_ #1 _tl } { gray }
36194     \__color_model_devicen_convert:nnnn {#1} { gray } { 1 } {#2}
36195   }
36196 \cs_new_protected:Npn \__color_model_devicen_convert_rgb:nnn #1#2
36197   {
36198     \tl_const:cn { c__color_fallback_ #1 _tl } { rgb }
36199     \__color_model_devicen_convert:nnnn {#1} { rgb } { 3 } {#2}
36200   }
36201 \cs_new_protected:Npn \__color_model_devicen_convert:nnnnn #1#2#3#4#5
36202   {
36203     \cs_new:cpn { __color_convert_ #2 _ #1 :w } ##1 \s__color_stop {#5}
36204     \cs_new:cpx { __color_convert_ #1 _ #2 :w } ##1 \s__color_stop
36205     {
36206       \exp_not:c { __color_convert_devicen_ #2 : \prg_replicate:nn {#3} { n } w }
36207       \prg_replicate:nn {#3} { { 1 } }
36208       ##1 ~ \exp_not:N \s__color_mark
36209       \clist_map_function:nN {#4} \__color_model_devicen_convert:n
36210       {}
36211       \exp_not:N \s__color_stop
36212     }
36213   }
36214 \cs_new:Npn \__color_model_devicen_convert:n #1
36215   {
36216     {
36217       \exp_args:Ne \__color_model_devicen_convert_aux:n
36218       { \prop_item:Nn \g__color_alternative_values_prop {#1} }
36219     }
36220   }
36221 \cs_new:Npn \__color_model_devicen_convert_aux:n #1
36222   { \__color_model_devicen_convert_aux:w #1 , , , , \s__color_stop }
36223 \cs_new:Npn \__color_model_devicen_convert_aux:w #1 , #2 , #3 , #4 , #5 \s__color_stop
36224   {
36225     {#1}
36226     \tl_if_blank:nF {#2}
36227     {
36228       {#2}
36229       \tl_if_blank:nF {#3}
36230       {
36231         {#3}
36232         \tl_if_blank:nF {#4} { {#4} }
36233       }
36234   }
```

```

36234     }
36235   }
36236 \cs_new:Npn \__color_convert_devicen_cmyk:nnnnw
36237   #1#2#3#4#5 ~ #6 \s__color_mark #7#8 \s__color_stop
36238   {
36239     \__color_convert_devicen_cmyk:nnnnnnnnn {#5} {#1} {#2} {#3} {#4} #7
36240     #6 \s__color_mark #8 \s__color_stop
36241   }
36242 \cs_new:Npn \__color_convert_devicen_cmyk:nnnnnnnnn #1#2#3#4#5#6#7#8#9
36243   {
36244     \use:e
36245     {
36246       \exp_not:N \__color_convert_devicen_cmyk_aux:nnnnw
36247       { \fp_eval:n { #2 * (1 - (#1 * #6)) } }
36248       { \fp_eval:n { #3 * (1 - (#1 * #7)) } }
36249       { \fp_eval:n { #4 * (1 - (#1 * #8)) } }
36250       { \fp_eval:n { #5 * (1 - (#1 * #9)) } }
36251     }
36252   }
36253 \cs_new:Npn \__color_convert_devicen_cmyk_aux:nnnnw
36254   #1#2#3#4 #5 \s__color_mark #6 \s__color_stop
36255   {
36256     \tl_if_blank:nTF {#5}
36257     {
36258       \fp_eval:n { 1 - #1 } ~
36259       \fp_eval:n { 1 - #2 } ~
36260       \fp_eval:n { 1 - #3 } ~
36261       \fp_eval:n { 1 - #4 }
36262     }
36263   {
36264     \__color_convert_devicen_cmyk:nnnnw {#1} {#2} {#3} {#4}
36265     #5 \s__color_mark #6 \s__color_stop
36266   }
36267 }
36268 \cs_new:Npn \__color_convert_devicen_gray:nw
36269   #1#2 ~ #3 \s__color_mark #4#5 \s__color_stop
36270   {
36271     \__color_convert_devicen_gray:nnn {#2} {#1} #4
36272     #3 \s__color_mark #5 \s__color_stop
36273   }
36274 \cs_new:Npn \__color_convert_devicen_gray:nnn #1#2#3
36275   {
36276     \exp_arsgs:Ne \__color_convert_devicen_gray_aux:nw
36277     { \fp_eval:n { #2 * (1 - (#1 * #3)) } }
36278   }
36279 \cs_new:Npn \__color_convert_devicen_gray_aux:nw
36280   #1 #2 \s__color_mark #3 \s__color_stop
36281   {
36282     \tl_if_blank:nTF {#2}
36283     { \fp_eval:n { 1 - #1 } }
36284   {
36285     \__color_convert_devicen_gray:nw {#1}
36286     #2 \s__color_mark #3 \s__color_stop
36287   }

```

```

36288     }
36289     \cs_new:Npn \__color_convert_devicen_rgb:nnnw
36290     #1#2#3#4 ~ #5 \s__color_mark #6#7 \s__color_stop
36291     {
36292         \__color_convert_devicen_rgb:nnnnnnn {#4} {#1} {#2} {#3} #6
36293         #5 \s__color_mark #7 \s__color_stop
36294     }
36295 \cs_new:Npn \__color_convert_devicen_rgb:nnnnnnn #1#2#3#4#5#6#7
36296     {
36297         \use:e
36298         {
36299             \exp_not:N \__color_convert_devicen_rgb_aux:nnnw
36300             { \fp_eval:n { #2 * (1 - (#1 * #5)) } }
36301             { \fp_eval:n { #3 * (1 - (#1 * #6)) } }
36302             { \fp_eval:n { #4 * (1 - (#1 * #7)) } }
36303         }
36304     }
36305 \cs_new:Npn \__color_convert_devicen_rgb_aux:nnnw
36306     #1#2#3 #4 \s__color_mark #5 \s__color_stop
36307     {
36308         \tl_if_blank:nTF {#4}
36309         {
36310             \fp_eval:n { 1 - #1 } ~
36311             \fp_eval:n { 1 - #2 } ~
36312             \fp_eval:n { 1 - #3 }
36313         }
36314     {
36315         \__color_convert_devicen_rgb:nnnw {#1} {#2} {#3}
36316         #4 \s__color_mark #5 \s__color_stop
36317     }
36318 }

```

(End definition for `\__color_model_devicen:n` and others.)

`\c_color icc colorspace signatures prop`

The signatures in the ICC file header indicating the underlying colorspace. We map it to three values: The number of components, the values corresponding to white, and the range.

```

36319 \prop_const_from_keyval:Nn \c_color_icc_colorspace_signatures_prop
36320     {
36321     % Gray
36322     47524159 = {1} {1} {0} {},
36323     % RGB
36324     52474220 = {3} {0~0~0} {1~1~1} {},
36325     % CMYK
36326     434D594B = {4} {0~0~0~1} {0~0~0~0} {},
36327     % Lab
36328     4C616220 = {3} {0~0~0} {100~0~0} {0~100~-128~127~-128~127}
36329 }

```

(End definition for `\c_color_icc_colorspace_signatures_prop`.)

`\__color_model_iccbased:n  
\__color_model_iccbased:nn  
\__color_model_iccbased:nnn  
\__color_model_iccbased_aux:nnn`

For an ICC profile, we need a file name and a number of components. The file name is processed here so the backend can treat it as a string.

```

36330 \cs_new_protected:Npn \__color_model_iccbased:n #1

```

```

36331  {
36332    \prop_get:NnNTF \l__color_internal_prop { file }
36333    \l__color_internal_tl
36334    {
36335      \exp_args:NV \__color_model_iccbased:nn
36336      \l__color_internal_tl {#1}
36337    }
36338    {
36339      \msg_error:nnn { color }
36340      { ICCBased-requires-file } {#1}
36341    }
36342  }
36343 \cs_new_protected:Npn \__color_model_iccbased:nn #1#2
36344  {
36345    \exp_args:NNx \prop_get:NnNTF \c__color_icc_colorspace_signatures_prop
36346    { \file_hex_dump:n {#1} { 17 } { 20 } } \l__color_internal_tl
36347    {
36348      \exp_last_unbraced:NV \__color_model_iccbased_aux:nnnnnn
36349      \l__color_internal_tl {#2} {#1}
36350    }
36351    {
36352      \msg_error:nnn { color }
36353      { ICCBased-unsupported-colorspace } {#2}
36354    }
36355  }

```

Here, we can use the same internals as for DeviceN approach as we know the number of components. No conversion is possible, so there is no need to worry about that at all.

```

36356 \cs_new_protected:Npn \__color_model_iccbased_aux:nnnnnn #1#2#3#4#5#6
36357  {
36358    \__color_model_init:nnn {#5} { iccbased } {#3}
36359    \tl_const:cn { c__color_fallback_ #5 _tl } { gray }
36360    \cs_new:cpn { __color_convert_ #5 _gray:w } ##1 \s__color_stop { 0 }
36361    \cs_new:cpn { __color_convert_gray_ #5 :w } ##1 \s__color_stop { #2 }
36362    \use:c { __color_model_device_n_parse_ #1 :nn } {#5} {#1}
36363    \exp_args:Nx \__color_backend_iccbased_init:nnn
36364      { \file_full_name:n {#6} } {#1} {#4}
36365  }

```

(End definition for `\__color_model_iccbased:n` and others.)

## 87.13 Applying profiles

`\color_profile_apply:nn`

```

\__color_profile_apply:nn
  \__color_profile_apply_gray:n
  \__color_profile_apply_rgb:n
  \__color_profile_apply_cmyk:n

```

```

36366 \cs_new_protected:Npn \color_profile_apply:nn #1#2
36367  {
36368    \exp_args:Ne \__color_profile_apply:nn
36369    { \file_full_name:n {#1} } {#2}
36370  }
36371 \cs_new_protected:Npn \__color_profile_apply:nn #1#2
36372  {
36373    \cs_if_exist_use:cF { __color_profile_apply_ \tl_to_str:n {#2} :n }
36374    {

```

```

36375           \msg_error:nnn { color } { ICC-Device-unknown } {#2}
36376           \use_none:n
36377       }
36378       {#1}
36379   }
36380 \cs_new_protected:Npn \__color_profile_apply_gray:n #1
36381   {
36382     \int_gincr:N \g__color_model_int
36383     \__color_backend_iccbased_device:nnn {#1} { Gray } { 1 }
36384   }
36385 \cs_new_protected:Npn \__color_profile_apply_rgb:n #1
36386   {
36387     \int_gincr:N \g__color_model_int
36388     \__color_backend_iccbased_device:nnn {#1} { RGB } { 3 }
36389   }
36390 \cs_new_protected:Npn \__color_profile_apply_cmyk:n #1
36391   {
36392     \int_gincr:N \g__color_model_int
36393     \__color_backend_iccbased_device:nnn {#1} { CMYK } { 4 }
36394   }

```

(End definition for `\color_profile_apply:nn` and others. This function is documented on page [298](#).)

## 87.14 Diagnostics

`\color_show:n` Extract the information about a color and format for the user: the approach is similar to the keys module here.

```

\__color_show:Nn
\__color_show:n
36395 \cs_new_protected:Npn \color_show:n
36396   { \__color_show:Nn \msg_show:nnxxxx }
36397 \cs_new_protected:Npn \color_log:n
36398   { \__color_show:Nn \msg_log:nnxxxx }
36399 \cs_new_protected:Npn \__color_show:Nn #1#2
36400   {
36401     #1 { color } { show }
36402     {#2}
36403     {
36404       \color_if_exist:nT {#2}
36405       {
36406         \exp_args:Nv \__color_show:n { l__color_named_ #2 _tl }
36407         \prop_map_function:cN
36408           { l__color_named_ #2 _prop }
36409           \msg_show_item_unbraced:nn
36410         }
36411       }
36412     { }
36413     { }
36414   }
36415 \cs_new:Npn \__color_show:n #1
36416   {
36417     \msg_show_item_unbraced:nn { model } {#1}
36418   }

```

(End definition for `\color_show:n` and others. These functions are documented on page [294](#).)

## 87.15 Messages

```
36419 \msg_new:nnnn { color } { CIELAB-requires-illuminant }
36420   { CIELAB-color-space-'#1'~require-an~illuminant. }
36421   {
36422     LaTeX-has-been-asked-to-create-a-separation-color-space-using-
36423     CIELAB-specifications,~but-no-\ \
36424     \iow_indent:n { illuminant==<basis> }
36425     \ \
36426     key-was-given-with-the-correct-information.~LaTeX-will-use-illuminant-
36427     'd50~for~recovery.
36428   }
36429 \msg_new:nnnn { color } { conversion-not-available }
36430   { No-model-conversion-available-from-'#1'~to-'#2'. }
36431   {
36432     LaTeX-has-been-asked-to-convert-a-color-from-model-'#1'~
36433     to-model'#2',~but~there~is~no~method~available~to~do~that.
36434   }
36435 \msg_new:nnnn { color } { DeviceN-inconsistent-alternative }
36436   { DeviceN-color-spaces-require-a-single-alternative-space. }
36437   {
36438     LaTeX-has-been-asked-to-create-a-DeviceN-color-space-'#1',~
36439     but~the~constituent~colors~do~not~have~a~common~alternative-
36440     color.
36441   }
36442 \msg_new:nnnn { color } { DeviceN-no-alternative }
36443   { DeviceN-color-spaces-require-an-alternative-space. }
36444   {
36445     LaTeX-has-been-asked-to-create-a-DeviceN-color-space-'#1',~
36446     but~the~constituent~colors~do~not~all~have~a~device-based-alternative.
36447   }
36448 \msg_new:nnnn { color } { DeviceN-requires-names }
36449   { DeviceN-color-space-'#1'~require-a-list-of-names. }
36450   {
36451     LaTeX-has-been-asked-to-create-a-DeviceN-color-space,~
36452     but-no-\ \
36453     \iow_indent:n { names==<names> }
36454     \ \
36455     key-was-given-with-the-correct-information.
36456   }
36457 \msg_new:nnnn { color } { ICC-Device-unknown }
36458   { Unknown-device-color-space-'#1'. }
36459   {
36460     LaTeX-has-been-asked-to-apply-an-ICC-profile~but~the~device~color~space~
36461     '#1'~is~unknown.
36462   }
36463 \msg_new:nnnn { color } { ICCBased-unsupported-colorspace }
36464   { ICCBased-color-space-'#1'~uses~an~unsupported~data~color~space. }
36465   {
36466     LaTeX-has-been-asked-to-create-a-ICCBased-colorspace,~but~the~
36467     used~data~colorspace~is~not~supported.~ICC~profiles~used~for~
36468     defining~a~ICCBased~colorspace~should~use~a~Lab,~RGB,~or~
36469     CMYK~data~colorspace.~LaTeX~will~ignore~this~request.
36470 }
```

```

36471 \msg_new:nnnn { color } { ICCBased-requires-file }
36472   { ICCBased-color-space-'#1'~require~an~file. }
36473   {
36474     LaTeX~has~been~asked~to~create~an~ICCBased~color~space,~but~no~\\ \\
36475     \iow_indent:n { file=-<name> }
36476     \\
36477     key~was~given~with~the~correct~information.~LaTeX~will~ignore~this~
36478     request.
36479   }
36480 \msg_new:nnnn { color } { model-already-defined }
36481   { Color~model-'#1'~already~defined. }
36482   {
36483     LaTeX~was~asked~to~define~a~new~color~model~called~'#1~,~but~
36484     this~color~model~already~exists.
36485   }
36486 \msg_new:nnnn { color } { separation-alternative-model }
36487   { Separation~color~space~'#1'~require~an~alternative~model. }
36488   {
36489     LaTeX~has~been~asked~to~create~a~separation~color~space,~
36490     but~no~\\ \\
36491     \iow_indent:n { alternative-model=~<model> }
36492     \\
36493     key~was~given~with~the~correct~information.
36494   }
36495 \msg_new:nnnn { color } { separation-alternative-values }
36496   { Separation~color~space~'#1'~require~values~for~the~alternative~space. }
36497   {
36498     LaTeX~has~been~asked~to~create~a~separation~color~space,~
36499     but~no~\\ \\
36500     \iow_indent:n { alternative-values=~<model> }
36501     \\
36502     key~was~given~with~the~correct~information.
36503   }
36504 \msg_new:nnnn { color } { separation-requires-name }
36505   { Separation~color~space~'#1'~require~a~formal~name. }
36506   {
36507     LaTeX~has~been~asked~to~create~a~separation~color~space,~
36508     but~no~\\ \\
36509     \iow_indent:n { name=-<formal-name> }
36510     \\
36511     key~was~given~with~the~correct~information.
36512   }
36513 \msg_new:nnn { color } { unhandled-model }
36514   {
36515     Unhandled~color~model~in~LaTeX2e~value~"#1":~\\ \\
36516       falling~back~on~grayscale.
36518   }
36519 \msg_new:nnnn { color } { unknown-color }
36520   { Unknown~color~'#1'. }
36521   {
36522     LaTeX~has~been~asked~to~use~a~color~named~'#1',~
36523     but~this~has~never~been~defined.
36524   }

```

```

36525 \msg_new:nnnn { color } { unknown-alternative-model }
36526   { Separation-color-space'#1'~require~an~valid~alternative~space. }
36527   {
36528     LaTeX~has~been~asked~to~create~a~separation~color~space,~
36529     but~the~model~given~as\\ \\
36530     \iow_indent:n { alternative-model==<model> }
36531     \\ \\
36532     is~unknown.
36533   }
36534 \msg_new:nnnn { color } { unknown-export-format }
36535   { Unknown~export~format~'#1'. }
36536   {
36537     LaTeX~has~been~asked~to~export~a~color~in~format~'#1',~
36538     but~this~has~never~been~defined.
36539   }
36540 \msg_new:nnnn { color } { unknown-CIELAB-illuminant }
36541   { Unknown~illuminant~model~'#1'. }
36542   {
36543     LaTeX~has~been~asked~to~use~create~a~color~space~using~CIELAB~
36544     illuminant~'#1',~but~this~does~not~exist.
36545   }
36546 \msg_new:nnnn { color } { unknown-model }
36547   { Unknown~color~model~'#1'. }
36548   {
36549     LaTeX~has~been~asked~to~use~a~color~model~called~'#1',~
36550     but~this~model~is~not~set~up.
36551   }
36552 \msg_new:nnnn { color } { unknown-model-type }
36553   { Unknown~color~model~type~'#1'. }
36554   {
36555     LaTeX~has~been~asked~to~create~a~new~color~model~called~'#1',~
36556     but~this~type~of~model~was~never~set~up.
36557   }
36558 \prop_gput:Nnn \g_msg_module_name_prop { color } { LaTeX }
36559 \prop_gput:Nnn \g_msg_module_type_prop { color } { }
36560 \msg_new:nnn { color } { show }
36561   {
36562     The~color~#1~
36563     \tl_if_empty:nTF {#2}
36564       { is~undefined. }
36565       { has~the~properties: #2 }
36566   }
36567 
```

  </package>

# Chapter 88

## Deprecated functions

---

`\pdf_object_new:nn \pdf_object_new:nn {⟨object⟩} {⟨type⟩}`

New: 2021-02-10 Declares ⟨object⟩ as a PDF object of ⟨type⟩, which should be one of

- `array`
- `dict`
- `fstream`
- `stream`

The object may be referenced from this point on, and written later using `\pdf_object_write:nn`.

Deprecated in favor of `\pdf_object_new:n`.

---

`\pdf_object_write:nn \pdf_object_write:nn {⟨object⟩} {⟨content⟩}`

`\pdf_object_write:nx` Writes the ⟨content⟩ as content of the ⟨object⟩. Depending on the ⟨type⟩ declared for the  
New: 2021-02-10 object, the format required for the ⟨data⟩ will vary

`array` A space-separated list of values

`dict` Key–value pairs in the form /⟨key⟩ ⟨value⟩

`fstream` Two brace groups: ⟨file name⟩ and ⟨file content⟩

`stream` Two brace groups: ⟨attributes (dictionary)⟩ and ⟨stream contents⟩

Deprecated in favor of `\pdf_object_write:nnn`.

# Chapter 89

## I3pdf implementation

```
36568  <*package>
36569  <@=pdf>
\s__pdf_stop Internal scan marks.
36570  \scan_new:N \s__pdf_stop
(End definition for \s__pdf_stop.)
```

\g\_\_pdf\_init\_bool A flag so we have some chance of avoiding setting things we are not allowed to. As we are potentially early in the format, we have to work a bit harder than ideal.

```
36571  \bool_new:N \g__pdf_init_bool
36572  \bool_lazy_and:nnT
36573  { \str_if_eq_p:Vn \fmtname { LaTeX2e } }
36574  { \tl_if_exist_p:N \expl@finalise@setup@@ }
36575  {
36576    \tl_gput_right:Nn \expl@finalise@setup@@
36577    {
36578      \tl_gput_right:Nn \kernel@after@begindocument
36579      { \bool_gset_true:N \g__pdf_init_bool }
36580    }
36581 }
```

(End definition for \g\_\_pdf\_init\_bool.)

### 89.1 Compression

\pdf\_uncompress: Simple to do.

```
36582  \cs_new_protected:Npn \pdf_uncompress:
36583  {
36584    \bool_if:NF \g__pdf_init_bool
36585    {
36586      \__pdf_backend_compresslevel:n { 0 }
36587      \__pdf_backend_compress_objects:n { \c_false_bool }
36588    }
36589 }
```

(End definition for \pdf\_uncompress:. This function is documented on page 301.)

## 89.2 Objects

```

\pdf_object_new:n
\pdf_object_write:nnn
\pdf_object_write:nnx
\pdf_object_ref:n
\pdf_object_unnamed_write:nn
\pdf_object_unnamed_write:nx
\pdf_object_ref_last:
\pdf_object_if_exist_p:n
\pdf_object_if_exist:nTF

```

Simple to do: all objects create a constant `int` so it is not a backend-specific name.

```

36590 \cs_new_protected:Npn \pdf_object_new:n #1
36591 {
36592     \__pdf_backend_object_new:n {#1}
36593     \cs_new_eq:cc
36594     { c__pdf_backend_object_ \tl_to_str:n {#1} _int }
36595     { c__pdf_object_ \tl_to_str:n {#1} _int }
36596 }
36597 \cs_new_protected:Npn \pdf_object_write:nnn #1#2#3
36598 {
36599     \__pdf_backend_object_write:nnn {#1} {#2} {#3}
36600     \bool_gset_true:N \g__pdf_init_bool
36601 }
36602 \cs_generate_variant:Nn \pdf_object_write:nnn { nnx }
36603 \cs_new:Npn \pdf_object_ref:n #1 { \__pdf_backend_object_ref:n {#1} }
36604 \cs_new_protected:Npn \pdf_object_unnamed_write:nn #1#2
36605 {
36606     \__pdf_backend_object_now:nn {#1} {#2}
36607     \bool_gset_true:N \g__pdf_init_bool
36608 }
36609 \cs_generate_variant:Nn \pdf_object_unnamed_write:nn { nx }
36610 \cs_new:Npn \pdf_object_ref_last: { \__pdf_backend_object_last: }
36611 \prg_new_conditional:Npnn \pdf_object_if_exist:n #1 { p , T , F , TF }
36612 {
36613     \int_if_exist:cTF { c__pdf_object_ \tl_to_str:n {#1} _int }
36614     \prg_return_true:
36615     \prg_return_false:
36616 }

```

(End definition for `\pdf_object_new:n` and others. These functions are documented on page 299.)

```
\pdf_pageobject_ref:n
```

```

36617 \cs_new:Npn \pdf_pageobject_ref:n #1
36618 { \__pdf_backend_pageobject_ref:n {#1} }

```

(End definition for `\pdf_pageobject_ref:n`. This function is documented on page 300.)

## 89.3 Version

`\pdf_version_compare:Nn` To compare version, we need to split the given value then deal with both major and minor version

```

36619 \prg_new_conditional:Npnn \pdf_version_compare:Nn #1#2 { p , T , F , TF }
36620 { \use:c { __pdf_version_compare_ #1 :w } #2 . . \s__pdf_stop }
36621 \cs_new:cpn { __pdf_version_compare_=:w } #1 . #2 . #3 \s__pdf_stop
36622 {
36623     \bool_lazy_and:nnTF
36624     { \int_compare_p:nNn \__pdf_backend_version_major: = {#1} }
36625     { \int_compare_p:nNn \__pdf_backend_version_minor: = {#2} }
36626     { \prg_return_true: }
36627     { \prg_return_false: }
36628 }

```

```

36629 \cs_new:cpn { __pdf_version_compare_<:w } #1 . #2 . #3 \s__pdf_stop
36630 {
36631     \bool_lazy_or:nnTF
36632     { \int_compare_p:nNn __pdf_backend_version_major: < {#1} }
36633     {
36634         \bool_lazy_and_p:nn
36635         { \int_compare_p:nNn __pdf_backend_version_major: = {#1} }
36636         { \int_compare_p:nNn __pdf_backend_version_minor: < {#2} }
36637     }
36638     { \prg_return_true: }
36639     { \prg_return_false: }
36640 }
36641 \cs_new:cpn { __pdf_version_compare_>:w } #1 . #2 . #3 \s__pdf_stop
36642 {
36643     \bool_lazy_or:nnTF
36644     { \int_compare_p:nNn __pdf_backend_version_major: > {#1} }
36645     {
36646         \bool_lazy_and_p:nn
36647         { \int_compare_p:nNn __pdf_backend_version_major: = {#1} }
36648         { \int_compare_p:nNn __pdf_backend_version_minor: > {#2} }
36649     }
36650     { \prg_return_true: }
36651     { \prg_return_false: }
36652 }

```

(End definition for `\pdf_version_compare:Nn` and others. This function is documented on page ??.)

`\pdf_version_gset:n`

`\pdf_version_min_gset:n`

`\__pdf_version_gset:w`

Split the version and set.

```

36653 \cs_new_protected:Npn \pdf_version_gset:n #1
36654     { \__pdf_version_gset:w #1 . . \s__pdf_stop }
36655 \cs_new_protected:Npn \pdf_version_min_gset:n #1
36656     {
36657         \pdf_version_compare:NnT < {#1}
36658         { \__pdf_version_gset:w #1 . . \s__pdf_stop }
36659     }
36660 \cs_new_protected:Npn \__pdf_version_gset:w #1 . #2 . #3 \s__pdf_stop
36661     {
36662         \bool_if:NF \g__pdf_init_bool
36663         {
36664             \__pdf_backend_version_major_gset:n {#1}
36665             \__pdf_backend_version_minor_gset:n {#2}
36666         }
36667     }

```

(End definition for `\pdf_version_gset:n`, `\pdf_version_min_gset:n`, and `\__pdf_version_gset:w`. These functions are documented on page 300.)

`\pdf_version:`

`\pdf_version_major:`

`\pdf_version_minor:`

Wrappers.

```

36668 \cs_new:Npn \pdf_version:
36669     { \__pdf_backend_version_major: . \__pdf_backend_version_minor: }
36670 \cs_new:Npn \pdf_version_major: { \__pdf_backend_version_major: }
36671 \cs_new:Npn \pdf_version_minor: { \__pdf_backend_version_minor: }

```

(End definition for `\pdf_version:`, `\pdf_version_major:`, and `\pdf_version_minor:`. These functions are documented on page 300.)

## 89.4 Page size

```
\pdf_pagesize_gset:nn
36672 \cs_new_protected:Npn \pdf_pagesize_gset:nn #1#2
36673 { \__pdf_pagesize_gset:nn {#1} {#2} }

(End definition for \pdf_pagesize_gset:nn. This function is documented on page 301.)
```

## 89.5 Destinations

```
\pdf_destination:nn
36674 \cs_new_protected:Npn \pdf_destination:nn #1#2
36675 { \__pdf_backend_destination:nn {#1} {#2} }

(End definition for \pdf_destination:nn. This function is documented on page 302.)
```

```
\pdf_destination:nnnn
36676 \cs_new_protected:Npn \pdf_destination:nnnn #1#2#3#4
36677 {
36678     \hbox_to_zero:n
36679     { \__pdf_backend_destination:nnnn {#1} {#2} {#3} {#4} }
36680 }

(End definition for \pdf_destination:nnnn. This function is documented on page 302.)
```

## 89.6 PDF Page size (media box)

Everything here is delayed to the start of the document so that the backend will definitely be loaded.

```
36681 \cs_if_exist:NT \@kernel@before@begindocument
36682 {
36683     \tl_gput_right:Nn \@kernel@before@begindocument
36684     {
36685         \bool_lazy_all:nT
36686         {
36687             { \cs_if_exist_p:N \stockheight }
36688             { \cs_if_exist_p:N \stockwidth }
36689             { \cs_if_exist_p:N \IfDocumentMetadataTF }
36690             { \IfDocumentMetadataTF { \c_true_bool } { \c_false_bool } }
36691             { \int_compare_p:nNn \tex_mag:D = { 1000 } }
36692         }
36693     {
36694         \bool_lazy_and:nnTF
36695             { \dim_compare_p:nNn \stockheight > { Opt } }
36696             { \dim_compare_p:nNn \stockwidth > { Opt } }
36697             {
36698                 \__pdf_backend_pagesize_gset:nn
36699                 \stockwidth \stockheight
36700             }
36701     {
36702         \bool_lazy_or:nnF
36703             { \dim_compare_p:nNn \stockheight < { Opt } }
```

```

36704 { \dim_compare_p:nNn \stockwidth < { 0pt } }
36705 {
36706     \bool_lazy_and:nnT
36707     { \dim_compare_p:nNn \paperheight > { 0pt } }
36708     { \dim_compare_p:nNn \paperwidth > { 0pt } }
36709     {
36710         \__pdf_backend_pagesize_gset:nn
36711             \paperwidth \paperheight
36712     }
36713 }
36714 }
36715 }
36716 }
36717 }

```

## 89.7 Deprecated functions

`\g__pdf_object_prop` For tracking objects.

```
36718 \prop_new:N \g__pdf_object_prop
```

(*End definition for `\g__pdf_object_prop`.*)

`\pdf_object_new:nn`

`\pdf_object_write:nn`

`\pdf_object_write:nx`

Wrap up the type data in a prop.

```

36719 \cs_new_protected:Npn \pdf_object_new:nn #1#2
36720 {
36721     \prop_gput:Nnn \g__pdf_object_prop {#1} {#2}
36722     \__pdf_backend_object_new:n {#1}
36723 }
36724 \cs_new_protected:Npn \pdf_object_write:nn #1#2
36725 {
36726     \exp_args:Nnx \__pdf_backend_object_write:nnn
36727         {#1} { \prop_item:Nn \g__pdf_object_prop {#1} } {#2}
36728     \bool_gset_true:N \g__pdf_init_bool
36729 }
36730 \cs_generate_variant:Nn \pdf_object_write:nn { nx }

```

(*End definition for `\pdf_object_new:nn` and `\pdf_object_write:nn`. These functions are documented on page 1384.*)

```
36731 </package>
```

# Chapter 90

## I3candidates Implementation

36732 ⟨\*package⟩

### 90.1 Additions to I3box

36733 ⟨@=box⟩

#### 90.1.1 Viewing part of a box

\box\_clip:N A wrapper around the driver-dependent code.  
\box\_clip:c  
\box\_gclip:N  
\box\_gclip:c

```
36734 \cs_new_protected:Npn \box_clip:N #1
36735 { \hbox_set:Nn #1 { \__box_backend_clip:N #1 } }
36736 \cs_generate_variant:Nn \box_clip:N { c }
36737 \cs_new_protected:Npn \box_gclip:N #1
36738 { \hbox_gset:Nn #1 { \__box_backend_clip:N #1 } }
36739 \cs_generate_variant:Nn \box_gclip:N { c }
```

(End definition for \box\_clip:N and \box\_gclip:N. These functions are documented on page 306.)

\box\_set\_trim:Nnnnn Trimming from the left- and right-hand edges of the box is easy: kern the appropriate parts off each side.

```
36740 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
36741 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
36742 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
36743 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
36744 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
36745 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
36746 \cs_new_protected:Npn \__box_set_trim:NnnnnN #1#2#3#4#5#6
36747 {
36748     \hbox_set:Nn \l__box_internal_box
36749     {
36750         \__kernel_kern:n { -#2 }
36751         \box_use:N #1
36752         \__kernel_kern:n { -#4 }
36753     }
```

For the height and depth, there is a need to watch the baseline is respected. Material always has to stay on the correct side, so trimming has to check that there is enough material to trim. First, the bottom edge. If there is enough depth, simply set the depth, or if not move down so the result is zero depth. \box\_move\_down:nn is used in both

cases so the resulting box always contains a `\lower` primitive. The internal box is used here as it allows safe use of `\box_set_dp:Nn`.

```

36754     \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
36755     {
36756         \hbox_set:Nn \l__box_internal_box
36757         {
36758             \box_move_down:nn \c_zero_dim
36759             { \box_use_drop:N \l__box_internal_box }
36760         }
36761         \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
36762     }
36763     {
36764         \hbox_set:Nn \l__box_internal_box
36765         {
36766             \box_move_down:nn { (#3) - \box_dp:N #1 }
36767             { \box_use_drop:N \l__box_internal_box }
36768         }
36769         \box_set_dp:Nn \l__box_internal_box \c_zero_dim
36770     }

```

Same thing, this time from the top of the box.

```

36771     \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
36772     {
36773         \hbox_set:Nn \l__box_internal_box
36774         {
36775             \box_move_up:nn \c_zero_dim
36776             { \box_use_drop:N \l__box_internal_box }
36777         }
36778         \box_set_ht:Nn \l__box_internal_box
36779         { \box_ht:N \l__box_internal_box - (#5) }
36780     }
36781     {
36782         \hbox_set:Nn \l__box_internal_box
36783         {
36784             \box_move_up:nn { (#5) - \box_ht:N \l__box_internal_box }
36785             { \box_use_drop:N \l__box_internal_box }
36786         }
36787         \box_set_ht:Nn \l__box_internal_box \c_zero_dim
36788     }
36789     #6 #1 \l__box_internal_box
36790 }

```

(End definition for `\box_set_trim:Nnnnn`, `\box_gset_trim:Nnnnn`, and `\_box_set_trim:NnnnnN`. These functions are documented on page [306](#).)

`\box_set_viewport:Nnnnn`  
`\box_set_viewport:cnnnn`

`\box_gset_viewport:Nnnnn`

`\box_gset_viewport:cnnnn`

`\_box_viewport:NnnnnN`

The same general logic as for the trim operation, but with absolute dimensions. As a result, there are some things to watch out for in the vertical direction.

```

36791 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
36792   { \_box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
36793 \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
36794 \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
36795   { \_box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
36796 \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
36797 \cs_new_protected:Npn \_box_set_viewport:NnnnnN #1#2#3#4#5#6

```

```

36798  {
36799    \hbox_set:Nn \l__box_internal_box
36800    {
36801      \__kernel_kern:n { -#2 }
36802      \box_use:N #1
36803      \__kernel_kern:n { #4 - \box_wd:N #1 }
36804    }
36805    \dim_compare:nNnTF {#3} < \c_zero_dim
36806    {
36807      \hbox_set:Nn \l__box_internal_box
36808      {
36809        \box_move_down:nn \c_zero_dim
36810        { \box_use_drop:N \l__box_internal_box }
36811      }
36812      \box_set_dp:Nn \l__box_internal_box { - \__box_dim_eval:n {#3} }
36813    }
36814    {
36815      \hbox_set:Nn \l__box_internal_box
36816      { \box_move_down:nn {#3} { \box_use_drop:N \l__box_internal_box } }
36817      \box_set_dp:Nn \l__box_internal_box \c_zero_dim
36818    }
36819    \dim_compare:nNnTF {#5} > \c_zero_dim
36820    {
36821      \hbox_set:Nn \l__box_internal_box
36822      {
36823        \box_move_up:nn \c_zero_dim
36824        { \box_use_drop:N \l__box_internal_box }
36825      }
36826      \box_set_ht:Nn \l__box_internal_box
36827      {
36828        (#5)
36829        \dim_compare:nNnT {#3} > \c_zero_dim
36830        { - (#3) }
36831      }
36832    }
36833    {
36834      \hbox_set:Nn \l__box_internal_box
36835      {
36836        \box_move_up:nn { - \__box_dim_eval:n {#5} }
36837        { \box_use_drop:N \l__box_internal_box }
36838      }
36839      \box_set_ht:Nn \l__box_internal_box \c_zero_dim
36840    }
36841    #6 #1 \l__box_internal_box
36842  }

```

(End definition for `\box_set_viewport:Nnnnn`, `\box_gset_viewport:Nnnnn`, and `\__box_viewport:NnnnnN`. These functions are documented on page 306.)

## 90.2 Additions to `I3flag`

36843 `<@=flag>`

\flag\_raise\_if\_clear:n It might be faster to just call the “trap” function in all cases but conceptually the function name suggests we should only run it if the flag is zero in case the “trap” made customizable in the future.

```

36844 \cs_new:Npn \flag_raise_if_clear:n #1
36845 {
36846     \if_cs_exist:w flag~#1~0 \cs_end:
36847     \else:
36848         \cs:w flag~#1 \cs_end: 0 ;
36849     \fi:
36850 }
```

(End definition for \flag\_raise\_if\_clear:n. This function is documented on page 307.)

### 90.3 Additions to l3msg

```
36851 ⟨@=msg⟩
```

\msg\_show\_eval:Nn \msg\_log\_eval:Nn \\_\_msg\_show\_eval:nnN A short-hand used for \int\_show:n and similar functions that passes to \tl\_show:n the result of applying #1 (a function such as \int\_eval:n) to the expression #2. The use of f-expansion ensures that #1 is expanded in the scope in which the show command is called, rather than in the group created by \iow\_wrap:nnnN. This is only important for expressions involving the \currentgrouplevel or \currentgrouptype. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

36852 \cs_new_protected:Npn \msg_show_eval:Nn #1#2
36853 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
36854 \cs_new_protected:Npn \msg_log_eval:Nn #1#2
36855 { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
36856 \cs_new_protected:Npn \__msg_show_eval:nnN #1#2#3 { #3 { #2 = #1 } }
```

(End definition for \msg\_show\_eval:Nn, \msg\_log\_eval:Nn, and \\_\_msg\_show\_eval:nnN. These functions are documented on page 308.)

\msg\_show\_item:n \msg\_show\_item\_unbraced:n \msg\_show\_item:nn \msg\_show\_item\_unbraced:nn Each item in the variable is formatted using one of the following functions. We cannot use \\ and so on because these short-hands cannot be used inside the arguments of messages, only when defining the messages.

```

36857 \cs_new:Npx \msg_show_item:n #1
36858 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { {#1} } }
36859 \cs_new:Npx \msg_show_item_unbraced:n #1
36860 { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n {#1} }
36861 \cs_new:Npx \msg_show_item:nn #1#2
36862 {
36863     \iow_newline: > \use:nn { ~ } { ~ }
36864     \exp_not:N \tl_to_str:n { {#1} }
36865     \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
36866     \exp_not:N \tl_to_str:n { {#2} }
36867 }
36868 \cs_new:Npx \msg_show_item_unbraced:nn #1#2
36869 {
36870     \iow_newline: > \use:nn { ~ } { ~ }
36871     \exp_not:N \tl_to_str:n {#1}
36872     \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
36873     \exp_not:N \tl_to_str:n {#2}
36874 }
```

(End definition for `\msg_show_item:n` and others. These functions are documented on page 308.)

## 90.4 Additions to l3prg

36875 `<@@=bool>`

`\bool_set_inverse:N` Set to false or true locally or globally.  
36876 `\cs_new_protected:Npn \bool_set_inverse:N #1`  
36877   `{ \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }`  
`\bool_gset_inverse:c` 36878 `\cs_generate_variant:Nn \bool_set_inverse:N { c }`  
36879 `\cs_new_protected:Npn \bool_gset_inverse:N #1`  
36880   `{ \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }`  
36881 `\cs_generate_variant:Nn \bool_gset_inverse:N { c }`

(End definition for `\bool_set_inverse:N` and `\bool_gset_inverse:N`. These functions are documented on page 308.)

`\s__bool_mark` Internal scan marks.

`\s__bool_stop` 36882 `\scan_new:N \s__bool_mark`  
36883 `\scan_new:N \s__bool_stop`

(End definition for `\s__bool_mark` and `\s__bool_stop`.)

`\bool_case_true:n` For boolean cases the overall idea is the same as for `\tl_case:nnTF` as described in l3tl.

`\bool_case_true:nTF` 36884 `\cs_new:Npn \bool_case_true:nTF`  
36885   `{ \exp:w \__bool_case:NnTF \c_true_bool }`  
`\bool_case_false:n` 36886 `\cs_new:Npn \bool_case_true:nT #1#2`  
36887   `{ \exp:w \__bool_case:NnTF \c_true_bool {#1} {#2} { } }`  
`\bool_case_false:nTF` 36888 `\cs_new:Npn \bool_case_true:nF #1`  
36889   `{ \exp:w \__bool_case:NnTF \c_true_bool {#1} { } }`  
`\__bool_case:NnTF` 36890 `\cs_new:Npn \bool_case_true:n1`  
36891   `{ \exp:w \__bool_case:NnTF \c_true_bool {#1} { } { } }`  
`\__bool_case_true:w` 36892 `\cs_new:Npn \bool_case_false:nTF`  
36893   `{ \exp:w \__bool_case:NnTF \c_false_bool }`  
`\__bool_case_false:w` 36894 `\cs_new:Npn \bool_case_false:nT #1#2`  
36895   `{ \exp:w \__bool_case:NnTF \c_false_bool {#1} {#2} { } }`  
`\__bool_case_end:nw` 36896 `\cs_new:Npn \bool_case_false:nF #1`  
36897   `{ \exp:w \__bool_case:NnTF \c_false_bool {#1} { } }`  
`\__bool_case_end:nw` 36898 `\cs_new:Npn \bool_case_false:n1`  
36899   `{ \exp:w \__bool_case:NnTF \c_false_bool {#1} { } { } }`  
`\__bool_case_end:nw` 36900 `\cs_new:Npn \__bool_case:NnTF #1#2#3#4`  
36901   `{`  
36902     `\bool_if:NTF #1 \__bool_case_true:w \__bool_case_false:w`  
36903     `#2 #1 { } \s__bool_mark {#3} \s__bool_mark {#4} \s__bool_stop`  
36904   `}`  
`\__bool_case_true:w` 36905 `\cs_new:Npn \__bool_case_true:w #1#2`  
36906   `{`  
36907     `\bool_if:nTF {#1}`  
36908     `{ \__bool_case_end:nw {#2} }`  
36909     `{ \__bool_case_true:w }`  
36910   `}`  
`\__bool_case_false:w` 36911 `\cs_new:Npn \__bool_case_false:w #1#2`  
36912   `{`  
36913     `\bool_if:nTF {#1}`

```

36914      { \__bool_case_false:w }
36915      { \__bool_case_end:nw {#2} }
36916    }
36917 \cs_new:Npn \__bool_case_end:nw #1#2#3 \s__bool_mark #4#5 \s__bool_stop
36918   { \exp_end: #1 #4 }

```

(End definition for `\bool_case_true:nTF` and others. These functions are documented on page 309.)

## 90.5 Additions to `I3prop`

```
36919 ⟨@=prop⟩
```

`\_prop_use_i_delimit_by_s_stop:nw` Functions to gobble up to a scan mark.

```
36920 \cs_new:Npn \_prop_use_i_delimit_by_s_stop:nw #1 #2 \s__prop_stop {#1}
```

(End definition for `\_prop_use_i_delimit_by_s_stop:nw`.)

`\prop_rand_key_value:N` Contrarily to `clist`, `seq` and `t1`, there is no function to get an item of a `prop` given an integer between 1 and the number of items, so we write the appropriate code. There is no bounds checking because `\int_rand:nn` is always within bounds. The initial `\int_value:w` is stopped by the first `\s__prop` in `#1`.

```

36921 \cs_new:Npn \prop_rand_key_value:N #1
36922  {
36923    \prop_if_empty:NF #1
36924     {
36925       \exp_after:wN \_prop_rand_item:w
36926       \int_value:w \int_rand:nn { 1 } { \prop_count:N #1 }
36927       #1 \s__prop_stop
36928     }
36929   }
36930 \cs_generate_variant:Nn \prop_rand_key_value:N { c }
36931 \cs_new:Npn \_prop_rand_item:w #1 \s__prop \_prop_pair:wn #2 \s__prop #3
36932  {
36933   \int_compare:nNnf {#1} > 1
36934   { \_prop_use_i_delimit_by_s_stop:nw { \exp_not:n { {#2} {#3} } } }
36935   \exp_after:wN \_prop_rand_item:w
36936   \int_value:w \int_eval:n { #1 - 1 } \s__prop
36937 }

```

(End definition for `\prop_rand_key_value:N` and `\_prop_rand_item:w`. This function is documented on page 309.)

## 90.6 Additions to `I3seq`

```
36938 ⟨@=seq⟩
```

`\seq_mapthread_function:NNN` The idea is to first expand both sequences, adding the usual `{ ? \prg_break: }` to the end of each one. This is most conveniently done in two steps using an auxiliary function. The mapping then throws away the first tokens of `#2` and `#5`, which for items in both sequences are `\s__seq \_seq_item:n`. The function to be mapped are then applied to the two entries. When the code hits the end of one of the sequences, the break material stops the entire loop and tidy up. This avoids needing to find the count of the two sequences, or worrying about which is longer.

```

36939 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
36940   { \exp_after:wN \__seq_mapthread_function:wNN #2 \s__seq_stop #1 #3 }
36941 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \s__seq_stop #2#3
36942   {
36943     \exp_after:wN \__seq_mapthread_function:wNw #2 \s__seq_stop #3
36944     #1 { ? \prg_break: } { }
36945     \prg_break_point:
36946   }
36947 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \s__seq_stop #2
36948   {
36949     \__seq_mapthread_function:Nnnwnn #2
36950     #1 { ? \prg_break: } { }
36951     \s__seq_stop
36952   }
36953 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \s__seq_stop #5#6
36954   {
36955     \use_none:n #2
36956     \use_none:n #5
36957     #1 {#3} {#6}
36958     \__seq_mapthread_function:Nnnwnn #1 #4 \s__seq_stop
36959   }
36960 \cs_generate_variant:Nn \seq_mapthread_function:NNN { Nc , c , cc }
```

(End definition for `\seq_mapthread_function:NNN` and others. This function is documented on page [310](#).)

`\seq_set_filter:NNn`  
`\seq_gset_filter:NNn`  
`\__seq_set_filter:NNNn`

Similar to `\seq_map_inline:Nn`, without a `\prg_break_point:` because the user's code is performed within the evaluation of a boolean expression, and skipping out of that would break horribly. The `\__seq_wrap_item:n` function inserts the relevant `\__seq_item:n` without expansion in the input stream, hence in the x-expanding assignment.

```

36961 \cs_new_protected:Npn \seq_set_filter:NNn
36962   { \__seq_set_filter:NNNn \__kernel_tl_set:Nx }
36963 \cs_new_protected:Npn \seq_gset_filter:NNn
36964   { \__seq_set_filter:NNNn \__kernel_tl_gset:Nx }
36965 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
36966   {
36967     \__seq_push_item_def:n { \bool_if:nT {#4} { \__seq_wrap_item:n {##1} } }
36968     #1 #2 { #3 }
36969     \__seq_pop_item_def:
36970   }
```

(End definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `\__seq_set_filter:NNNn`. These functions are documented on page [310](#).)

Set `\__seq_item:n` then map it using the loop code.

```

36971 \cs_new_protected:Npn \seq_set_from_inline_x:Nnn
36972   { \__seq_set_from_inline_x:NNnn \__kernel_tl_set:Nx }
36973 \cs_new_protected:Npn \seq_gset_from_inline_x:Nnn
36974   { \__seq_set_from_inline_x:NNnn \__kernel_tl_gset:Nx }
36975 \cs_new_protected:Npn \__seq_set_from_inline_x:NNnn #1#2#3#4
36976   {
36977     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {#4} }
36978     #1 #2 { \s__seq #3 \__seq_item:n }
36979     \__seq_pop_item_def:
36980   }
```

(End definition for `\seq_set_from_inline_x:Nnn`, `\seq_gset_from_inline_x:Nnn`, and `\_seq_set_from_inline_x:NNnn`. These functions are documented on page 310.)

`\seq_set_from_function:NnN`  
`\seq_gset_from_function:NnN`

Reuse `\seq_set_from_inline_x:Nnn`.

```
36981 \cs_new_protected:Npn \seq_set_from_function:NnN #1#2#3
36982   { \seq_set_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
36983 \cs_new_protected:Npn \seq_gset_from_function:NnN #1#2#3
36984   { \seq_gset_from_inline_x:Nnn #1 {#2} { #3 {##1} } }
```

(End definition for `\seq_set_from_function:NnN` and `\seq_gset_from_function:NnN`. These functions are documented on page 310.)

`\_seq_int_eval:w`

Useful to more quickly go through items.

```
36985 \cs_new_eq:NN \_seq_int_eval:w \tex_numexpr:D
```

(End definition for `\_seq_int_eval:w`.)

`\seq_set_item:Nnn`  
`\seq_set_item:cnn`

The conditionals are distinguished from the Nnn versions by the last argument `\use_ii:nn` vs `\use_i:nn`.

```
36986 \cs_new_protected:Npn \seq_set_item:Nnn #1#2#3
36987   { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_set:Nx \use_i:nn }
36988 \cs_new_protected:Npn \seq_gset_item:Nnn #1#2#3
36989   { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_gset:Nx \use_i:nn }
36990 \cs_generate_variant:Nn \seq_set_item:Nnn { c }
36991 \cs_generate_variant:Nn \seq_gset_item:Nnn { c }
36992 \prg_new_protected_conditional:Npnn \seq_set_item:Nnn #1#2#3 { TF , T , F }
36993   { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_set:Nx \use_ii:nn }
36994 \prg_new_protected_conditional:Npnn \seq_gset_item:Nnn #1#2#3 { TF , T , F }
36995   { \_seq_set_item:NnnNN #1 {#2} {#3} \_kernel_tl_gset:Nx \use_ii:nn }
36996 \prg_generate_conditional_variant:Nnn \seq_set_item:Nnn { c } { TF , T , F }
36997 \prg_generate_conditional_variant:Nnn \seq_gset_item:Nnn { c } { TF , T , F }
```

Save the item to be stored and evaluate the position and the sequence length only once. Then depending on the sign of the position, check that it is not bigger than the length (in absolute value) nor zero.

```
36998 \cs_new_protected:Npn \_seq_set_item:NnnNN #1#2#3
36999   {
37000     \tl_set:Nn \l__seq_internal_a_tl { \_seq_item:n {#3} }
37001     \exp_args:Nff \_seq_set_item:nnNNNN
37002       { \int_eval:n {#2} } { \seq_count:N #1 } #1 \use_none:nn
37003   }
37004 \cs_new_protected:Npn \_seq_set_item:nnNNNN #1#2
37005   {
37006     \int_compare:nNnTF {#1} > 0
37007       { \int_compare:nNnF {#1} > {#2} { \_seq_set_item:nNnnNNNN { #1 - 1 } } }
37008     {
37009       \int_compare:nNnF {#1} < {-#2}
37010         {
37011           \int_compare:nNnF {#1} = 0
37012             { \_seq_set_item:nNnnNNNN { #2 + #1 } }
37013         }
37014     }
37015     \_seq_set_item_false:nnNNNN {#1} {#2}
37016   }
```

If the position is not ok, `\__seq_set_item_false:nnNNNN` calls an error or returns `false` (depending on the `\use_i:nn` vs `\use_ii:nn` argument mentioned above).

```

37017 \cs_new_protected:Npn \__seq_set_item_false:nnNNNN #1#2#3#4#5#6
37018 {
37019     #6
37020     {
37021         \msg_error:nnxxx { seq } { item-too-large }
37022             { \token_to_str:N #3 } {#2} {#1}
37023     }
37024     { \prg_return_false: }
37025 }
37026 \msg_new:nnnn { seq } { item-too-large }
37027     { Sequence~'#1'~does~not~have~an~item~#3 }
37028     {
37029         An~attempt~was~made~to~push~or~pop~the~item~at~position~#3~
37030         of~'#1',~but~this~
37031         \int_compare:nTF { #3 = 0 }
37032             { position~does~not~exist. }
37033             { sequence~only~has~#2~item \int_compare:nF { #2 = 1 } {s}. }
37034 }
```

If the position is ok, `\__seq_set_item:nNnnNNNN` makes the assignment and returns `true` (in the case of conditionnals). Here `#1` is an integer expression (position minus one), it needs to be evaluated. The sequence `#5` starts with `\s__seq` (even if empty), which stops the integer expression and is absorbed by it. The `\if_meaning:w` test is slightly faster than an integer test (but only works when testing against zero, hence the offset we chose in the position). When we are done skipping items, insert the saved item `\l__seq_internal_a_t1`. For put functions the last argument of `\__seq_set_item_end:w` is `\use_none:nn` and it absorbs the item `#2` that we are removing: this is only useful for the pop functions.

```

37035 \cs_new_protected:Npn \__seq_set_item:nNnnNNNN #1#2#3#4#5#6#7#8
37036 {
37037     #7 #5
37038     {
37039         \s__seq
37040         \exp_after:wN \__seq_set_item:wn
37041         \int_value:w \__seq_int_eval:w #1
37042         #5 \s__seq_stop #6
37043     }
37044     #8 { } { \prg_return_true: }
37045 }
37046 \cs_new:Npn \__seq_set_item:wn #1 \__seq_item:n #2
37047 {
37048     \if_meaning:w 0 #1 \__seq_set_item_end:w \fi:
37049     \exp_not:n { \__seq_item:n {#2} }
37050     \exp_after:wN \__seq_set_item:wn
37051     \int_value:w \__seq_int_eval:w #1 - 1 \s__seq
37052 }
37053 \cs_new:Npn \__seq_set_item_end:w #1 \exp_not:n #2 #3 \s__seq #4 \s__seq_stop #5
37054 {
37055     #1
37056     \exp_not:o \l__seq_internal_a_t1
37057     \exp_not:n {#4}
```

```

37058      #5 #2
37059  }

```

(End definition for `\seq_set_item:NnnTF` and others. These functions are documented on page 311.)

`\seq_pop_item:NnN` The NnN versions simply call the conditionals, for which we will rely on the internals of `\seq_set_item:Nnn`.

`\seq_pop_item:NnNTF`

`\seq_pop_item:cnN`

`\seq_gpop_item:NnN`

`\seq_gpop_item:cnN`

`\seq_gpop_item:NnNTF`

`\seq_gpop_item:cnNTF`

`\__seq_pop_item:NnNNN`

Save in `\l__seq_internal_b_tl` the token list variable #3 in which we will store the item. The `\__seq_set_item:nnNNNN` auxiliary eventually inserts `\l__seq_internal_a_tl` in place of the item found in the sequence, so we empty that. Instead of the last argument `\use_i:nn` or `\use_ii:nn` used for put functions, we introduce `\__seq_pop_item:nn`, which stores `\q_no_value` before calling its second argument (`\prg_return_true:/false:`) to end the conditional. The item found is passed to `\__seq_pop_item_aux:w`, which interrupts the x-expanding sequence assignment and stores the item using the assignment function in `\l__seq_internal_b_tl`.

```

37072 \cs_new_protected:Npn \__seq_pop_item:NnNN #1#2#3#4
37073  {
37074    \tl_clear:N \l__seq_internal_a_tl
37075    \tl_set:Nn \l__seq_internal_b_tl { \__kernel_tl_set:Nx #3 }
37076    \exp_args:Nff \__seq_set_item:nnNNNN
37077      { \int_eval:n {#2} } { \seq_count:N #1 }
37078      #1 \__seq_pop_item_aux:w #4 \__seq_pop_item:nn
37079  }
37080 \cs_new_protected:Npn \__seq_pop_item:nn #1#2
37081  {
37082    \if_meaning:w \prg_return_false: #2
37083      \l__seq_internal_b_tl { \exp_not:N \q_no_value }
37084    \fi:
37085    #2
37086  }
37087 \cs_new:Npn \__seq_pop_item_aux:w \__seq_item:n #1
37088  {
37089    \if_false: { \fi: }
37090    \l__seq_internal_b_tl { \if_false: } \fi: \exp_not:n {#1}
37091  }

```

(End definition for `\seq_pop_item:NnNTF`, `\seq_gpop_item:NnNTF`, and `\__seq_pop_item:NnNNN`. These functions are documented on page 311.)

## 90.7 Additions to l3sys

37092 ⟨@@=sys⟩

\c\_sys\_engine\_version\_str Various different engines, various different ways to extract the data!

```
37093 \str_const:Nx \c_sys_engine_version_str
37094 {
37095   \str_case:on \c_sys_engine_str
37096   {
37097     { pdftex }
37098     {
37099       \fp_eval:n { round(\int_use:N \tex_pdftexversion:D / 100 , 2) }
37100       .
37101       \tex_pdftexrevision:D
37102     }
37103     { ptx }
37104     {
37105       \cs_if_exist:NT \tex_ptexversion:D
37106       {
37107         p
37108         \int_use:N \tex_ptexversion:D
37109         .
37110         \int_use:N \tex_ptexminorversion:D
37111         \tex_ptexrevision:D
37112         -
37113         \int_use:N \tex_epTeXversion:D
37114       }
37115     }
37116     { luatex }
37117     {
37118       \fp_eval:n { round(\int_use:N \tex_luatexversion:D / 100, 2) }
37119       .
37120       \tex_luatexrevision:D
37121     }
37122     { uptex }
37123     {
37124       \cs_if_exist:NT \tex_ptexversion:D
37125       {
37126         p
37127         \int_use:N \tex_ptexversion:D
37128         .
37129         \int_use:N \tex_ptexminorversion:D
37130         \tex_ptexrevision:D
37131         -
37132         u
37133         \int_use:N \tex_uptexversion:D
37134         \tex_uptexrevision:D
37135         -
37136         \int_use:N \tex_epTeXversion:D
37137       }
37138     }
37139     { xetex }
37140     {
37141       \int_use:N \tex_XeTeXversion:D
```

```

37142           \tex_XeTeXrevision:D
37143       }
37144   }
37145 }
```

(End definition for `\c_sys_engine_version_str`. This variable is documented on page 311.)

## 90.8 Additions to `I3file`

```
37146 <@=ior>
```

`\ior_shell_open:Nn`  
`\_ior_shell_open:nN`

Actually much easier than either the standard open or input versions! When calling `\_kernel_iор_open:Nn` the file the pipe is added to signal a shell command, but the quotes are not added yet—they are added later by `\_kernel_file_name_quote:n`.

```

37147 \cs_new_protected:Npn \ior_shell_open:Nn #1#2
37148 {
37149     \sys_if_shell:TF
37150     { \exp_args:No \_ior_shell_open:nN { \tl_to_str:n {#2} } #1 }
37151     { \msg_error:nn { ior } { pipe-failed } }
37152 }
37153 \cs_new_protected:Npn \_ior_shell_open:nN #1#2
37154 {
37155     \tl_if_in:nnTF {#1} { " }
37156     {
37157         \msg_error:nnx
37158         { kernel } { quote-in-shell } {#1}
37159     }
37160     { \_kernel_iор_open:Nn #2 { |#1 } }
37161 }
37162 \msg_new:nnnn { ior } { pipe-failed }
37163 { Cannot-run-piped-system-commands. }
37164 {
37165     LaTeX-tried-to-call-a-system-process-but-this-was-not-possible.\\
37166     Try-the--shell-escape-(or--enable-pipes)-option.
37167 }
```

(End definition for `\ior_shell_open:Nn` and `\_ior_shell_open:nN`. This function is documented on page 307.)

## 90.9 Additions to `I3tl`

### 90.9.1 Building a token list

```
37168 <@=tl>
```

Between `\tl_build_begin:N`  $\langle tl\ var\rangle$  and `\tl_build_end:N`  $\langle tl\ var\rangle$ , the  $\langle tl\ var\rangle$  has the structure

```
\exp_end: ... \exp_end: \_tl_build_last:NNn  $\langle assignment\rangle$   $\langle next\ tl\rangle$ 
{ $\langle left\rangle$ }  $\langle right\rangle$ 
```

where  $\langle right\rangle$  is not braced. The “data” it represents is  $\langle left\rangle$  followed by the “data” of  $\langle next\ tl\rangle$  followed by  $\langle right\rangle$ . The  $\langle next\ tl\rangle$  is a token list variable whose name is that of  $\langle tl\ var\rangle$  followed by ‘’. There are between 0 and 4 `\exp_end:` to keep track of when  $\langle left\rangle$

and  $\langle right \rangle$  should be put into the  $\langle next tl \rangle$ . The  $\langle assignment \rangle$  is  $\backslash cs\_set\_nopar:Npx$  if the variable is local, and  $\backslash cs\_gset\_nopar:Npx$  if it is global.

**$\backslash tl\_build\_begin:N$**   
 **$\backslash tl\_build\_gbegin:N$**   
 $\backslash \_tl\_build\_begin:NN$   
 $\backslash \_tl\_build\_begin:NNN$

First construct the  $\langle next tl \rangle$ : using a prime here conflicts with the usual  $\text{expl3}$  convention but we need a name that can be derived from #1 without any external data such as a counter. Empty that  $\langle next tl \rangle$  and setup the structure. The local and global versions only differ by a single function  $\backslash cs\_gset\_nopar:Npx$  used for all assignments: this is important because only that function is stored in the  $\langle tl var \rangle$  and  $\langle next tl \rangle$  for subsequent assignments. In principle  $\backslash \_tl\_build\_begin:NNN$  could use  $\backslash tl\_gclear:N$  to empty #1 and make sure it is defined, but logging the definition does not seem useful so we just do #3 #1 {} to clear it locally or globally as appropriate.

```

37169 \cs_new_protected:Npn \tl_build_begin:N #1
37170   { \_tl_build_begin:NN \cs_set_nopar:Npx #1 }
37171 \cs_new_protected:Npn \tl_build_gbegin:N #1
37172   { \_tl_build_begin:NN \cs_gset_nopar:Npx #1 }
37173 \cs_new_protected:Npn \_tl_build_begin:NN #1#2
37174   { \exp_args:Nc \_tl_build_begin:NNN { \cs_to_str:N #2 } #2 #1 }
37175 \cs_new_protected:Npn \_tl_build_begin:NNN #1#2#3
37176   {
37177     #3 #1 {}
37178     #3 #2
37179     {
37180       \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
37181       \exp_not:n { \_tl_build_last:NNn #3 #1 {} }
37182     }
37183   }

```

(End definition for  $\backslash tl\_build\_begin:N$  and others. These functions are documented on page 312.)

**$\backslash tl\_build\_clear:N$**   
 **$\backslash tl\_build\_gclear:N$**

The **begin** and **gbegin** functions already clear enough to make the token list variable effectively empty. Eventually the **begin** and **gbegin** functions should check that #1' is empty or undefined, while the **clear** and **gclear** functions ought to empty #1', #1'', and so on, similar to  $\backslash tl\_build\_end:N$ . This only affects memory usage.

```

37184 \cs_new_eq:NN \tl_build_clear:N \tl_build_begin:N
37185 \cs_new_eq:NN \tl_build_gclear:N \tl_build_gbegin:N

```

(End definition for  $\backslash tl\_build\_clear:N$  and  $\backslash tl\_build\_gclear:N$ . These functions are documented on page 312.)

**$\backslash tl\_build\_put\_right:Nn$**   
 **$\backslash tl\_build\_put\_right:Nx$**   
 **$\backslash tl\_build\_gput\_right:Nn$**   
 **$\backslash tl\_build\_gput\_right:Nx$**   
 $\backslash \_tl\_build\_last:NNn$   
 $\backslash \_tl\_build\_put:nn$   
 $\backslash \_tl\_build\_put:nw$

Similar to  $\backslash tl\_put\_right:Nn$ , but apply  $\exp:w$  to #1. Most of the time this just removes one  $\exp_end:$ . When there are none left,  $\backslash \_tl\_build\_last:NNn$  is expanded instead. It resets the definition of the  $\langle tl var \rangle$  by ending the  $\exp_not:n$  and the definition early. Then it makes sure the  $\langle next tl \rangle$  (its argument #1) is set-up and starts a new definition. Then  $\backslash \_tl\_build\_put:nn$  and  $\backslash \_tl\_build\_put:nw$  place the  $\langle left \rangle$  part of the original  $\langle tl var \rangle$  as appropriate for the definition of the  $\langle next tl \rangle$  (the  $\langle right \rangle$  part is left in the right place without ever becoming a macro argument). We use  $\exp_after:wN$  rather than some  $\exp_args:No$  to avoid reading arguments that are likely very long token lists. We use  $\backslash cs\_set\_nopar:Npx$  rather than  $\backslash tl\_gset:Nx$  partly for the same reason and partly because the assignments are interrupted by brace tricks, which implies that the assignment does not simply set the token list to an x-expansion of the second argument.

```

37186 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
37187   {
37188     \cs_set_nopar:Npx #1

```

```

37189      { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
37190    }
37191 \cs_new_protected:Npn \tl_build_put_right:Nx #1#2
37192  {
37193    \cs_set_nopar:Npx #1
37194    { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
37195  }
37196 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
37197  {
37198    \cs_gset_nopar:Npx #1
37199    { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
37200  }
37201 \cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
37202  {
37203    \cs_gset_nopar:Npx #1
37204    { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
37205  }
37206 \cs_new_protected:Npn \__tl_build_last>NNn #1#2
37207  {
37208    \if_false: { { \fi:
37209      \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
37210      \__tl_build_last>NNn #1 #2 { }
37211    }
37212  }
37213  \if_meaning:w \c_empty_tl #2
37214    \__tl_build_begin>NN #1 #2
37215  \fi:
37216  #1 #2
37217  {
37218    \exp_after:wN \exp_not:n \exp_after:wN
37219    {
37220      \exp:w \if_false: } } \fi:
37221      \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
37222    }
37223 \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
37224 \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last>NNn #3#4#5
37225  { #2 \__tl_build_last>NNn #3 #4 { #1 #5 } }

```

(End definition for `\tl_build_put_right:Nn` and others. These functions are documented on page 313.)

**\tl\_build\_put\_left:Nn**  
**\tl\_build\_put\_left:Nx**  
**\tl\_build\_gput\_left:Nn**

See `\tl_build_right:Nn` for all the machinery. We could easily provide `\tl_build_put_left_right:Nnn`, by just add the `\langle right\rangle` material after the `\{ \langle left\rangle\}` in the x-expanding assignment.

```

37226 \cs_new_protected:Npn \tl_build_put_left:Nn #1
37227  { \__tl_build_put_left>NNn \cs_set_nopar:Npx #1 }
37228 \cs_generate_variant:Nn \tl_build_put_left:Nn { Nx }
37229 \cs_new_protected:Npn \tl_build_gput_left:Nn #1
37230  { \__tl_build_put_left>NNn \cs_gset_nopar:Npx #1 }
37231 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Nx }
37232 \cs_new_protected:Npn \__tl_build_put_left>NNn #1#2#3
37233  {
37234    #1 #2
37235    {
37236      \exp_after:wN \exp_not:n \exp_after:wN

```

```

37237     {
37238         \exp:w \exp_after:wN \__tl_build_put:nn
37239             \exp_after:wN {#2} {#3}
37240     }
37241 }
37242 }
```

(End definition for `\tl_build_put_left:Nn`, `\tl_build_gput_left:Nn`, and `\__tl_build_put_left>NNn`. These functions are documented on page 313.)

```

\tl_build_get:NN
\__tl_build_get:NNN
\__tl_build_get:w
\__tl_build_get_end:w
```

The idea is to expand the  $\langle tl\ var \rangle$  then the  $\langle next\ tl \rangle$  and so on, all within an x-expanding assignment, and wrap as appropriate in `\exp_not:n`. The various  $\langle left \rangle$  parts are left in the assignment as we go, which enables us to expand the  $\langle next\ tl \rangle$  at the right place. The various  $\langle right \rangle$  parts are eventually picked up in one last `\exp_not:n`, with a brace trick to wrap all the  $\langle right \rangle$  parts together.

```

37243 \cs_new_protected:Npn \tl_build_get:NN
37244     { \__tl_build_get:NNN \__kernel_tl_set:Nx }
37245 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
37246     { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
37247 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last>NNN #2#3#4
37248     {
37249         \exp_not:n {#4}
37250         \if_meaning:w \c_empty_tl #3
37251             \exp_after:wN \__tl_build_get_end:w
37252             \fi:
37253             \exp_after:wN \__tl_build_get:w #3
37254     }
37255 \cs_new:Npn \__tl_build_get_end:w #1#2#3
37256     { \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi: }
```

(End definition for `\tl_build_get:NN` and others. This function is documented on page 313.)

```

\tl_build_end:N
\tl_build_gend:N
\__tl_build_end_loop>NN
```

Get the data then clear the  $\langle next\ tl \rangle$  recursively until finding an empty one. It is perhaps wasteful to repeatedly use `\cs_to_str:N`. The local/global scope is checked by `\tl_set:Nx` or `\tl_gset:Nx`.

```

37257 \cs_new_protected:Npn \tl_build_end:N #1
37258     {
37259         \__tl_build_get:NNN \__kernel_tl_set:Nx #1 #1
37260         \exp_args:Nc \__tl_build_end_loop>NN { \cs_to_str:N #1 ' } \tl_clear:N
37261     }
37262 \cs_new_protected:Npn \tl_build_gend:N #1
37263     {
37264         \__tl_build_get:NNN \__kernel_tl_gset:Nx #1 #1
37265         \exp_args:Nc \__tl_build_end_loop>NN { \cs_to_str:N #1 ' } \tl_gclear:N
37266     }
37267 \cs_new_protected:Npn \__tl_build_end_loop>NN #1#2
37268     {
37269         \if_meaning:w \c_empty_tl #1
37270             \exp_after:wN \use_none:nnnnnn
37271             \fi:
37272             #2 #1
37273             \exp_args:Nc \__tl_build_end_loop>NN { \cs_to_str:N #1 ' } #2
37274     }
```

(End definition for `\tl_build_end:N`, `\tl_build_gend:N`, and `\__tl_build_end_loop>NN`. These functions are documented on page 313.)

### 90.9.2 Other additions to `I3tl`

`\tl_range_braced:Nnn`  
`\tl_range_braced:cnn`  
`\tl_range_braced:nnn`  
`\tl_range_unbraced:Nnn`  
`\tl_range_unbraced:cnn`  
`\tl_range_unbraced:nnn`

`\_tl_range_braced:w`  
`\_tl_range_collect_braced:w`  
`\_tl_range_unbraced:w`  
`\_tl_range_collect_unbraced:w`

For the braced version `\_tl_range_braced:w` sets up `\_tl_range_collect_braced:w` which stores items one by one in an argument after the semicolon. The unbraced version is almost identical. The version preserving braces and spaces starts by deleting spaces before the argument to avoid collecting them, and sets up `\_tl_range_collect:nn` with a first argument of the form `{ {<collected>} <tokens> }`, whose head is the collected tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the `<collected>` tokens.

```

37275 \cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:nnn }
37276 \cs_generate_variant:Nn \tl_range_braced:Nnn { c }
37277 \cs_new:Npn \tl_range_braced:nnn { \_tl_range:Nnnn \_tl_range_braced:w }
37278 \cs_new:Npn \tl_range_unbraced:Nnn
37279 { \exp_args:No \tl_range_unbraced:nnn }
37280 \cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
37281 \cs_new:Npn \tl_range_unbraced:nnn
37282 { \_tl_range:Nnnn \_tl_range_unbraced:w }
37283 \cs_new:Npn \_tl_range_braced:w #1 ; #2
37284 { \_tl_range_collect_braced:w #1 ; { } #2 }
37285 \cs_new:Npn \_tl_range_unbraced:w #1 ; #2
37286 { \_tl_range_collect_unbraced:w #1 ; { } #2 }
37287 \cs_new:Npn \_tl_range_collect_braced:w #1 ; #2#3
37288 {
37289     \if_int_compare:w #1 > \c_one_int
37290         \exp_after:wN \_tl_range_collect_braced:w
37291         \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
37292     \fi:
37293     { #2 {#3} }
37294 }
37295 \cs_new:Npn \_tl_range_collect_unbraced:w #1 ; #2#3
37296 {
37297     \if_int_compare:w #1 > \c_one_int
37298         \exp_after:wN \_tl_range_collect_unbraced:w
37299         \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
37300     \fi:
37301     { #2 #3 }
37302 }
```

(End definition for `\tl_range_braced:Nnn` and others. These functions are documented on page 312.)

## 90.10 Additions to `I3token`

`\c_catcode_active_space_tl`

While `\char_generate:nn` can produce active characters in some engines it cannot in general. It would be possible to simply change the catcode of space but then the code would need to avoid all spaces, making it quite unreadable. Instead we use the primitive `\tex_lowercase:D` trick.

```

37303 \group_begin:
37304     \char_set_catcode_active:N *
37305     \char_set_lccode:nn { '*' } { '\ }
37306     \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
37307 \group_end:
```

(End definition for `\c_catcode_active_space_tl`. This variable is documented on page 313.)

37308 ⟨`\@=peek`⟩

`\l_peek_collect_tl`

37309 `\tl_new:N \l_peek_collect_tl`

(End definition for `\l_peek_collect_tl`.)

```
\peek_catcode_collect_inline:Nn
\peekCharCode_collect_inline:Nn
\peek_meaning_collect_inline:Nn
  \_peek_collect:NNn
  \_peek_collect_true:w
\_peek_collect_remove:nw
  \_peek_collect:N
```

Most of the work is done by `\_peek_execute_branches_...`, which calls either `\_peek_true:w` or `\_peek_false:w` according to whether the next token `\l_peek_token` matches the search token (stored in `\l_peek_search_token` and `\l_peek_search_tl`). Here, in the `true` case we run `\_peek_collect_true:w`, which generally calls `\_peek_collect:N` to store the peeked token into `\l_peek_collect_tl`, except in special non-N-type cases (begin-group, end-group, or space), where a frozen token is stored. The `true` branch calls `\_peek_execute_branches_...` to fetch more matching tokens. Once there are no more, `\_peek_false_aux:n` closes the safe-align group and runs the user's inline code.

```
37310 \cs_new_protected:Npn \peek_catcode_collect_inline:Nn
  { \_peek_collect>NNn \_peek_execute_branches_catcode: }
37312 \cs_new_protected:Npn \peekCharCode_collect_inline:Nn
  { \_peek_collect>NNn \_peek_execute_branches_charcode: }
37314 \cs_new_protected:Npn \peek_meaning_collect_inline:Nn
  { \_peek_collect>NNn \_peek_execute_branches_meaning: }
37316 \cs_new_protected:Npn \_peek_collect>NNn #1#2#3
  {
    \group_align_safe_begin:
    \cs_set_eq:NN \l_peek_search_token #2
    \tl_set:Nn \l_peek_search_tl {#2}
    \tl_clear:N \l_peek_collect_tl
    \cs_set:Npn \_peek_false:w
      { \exp_args:No \_peek_false_aux:n \l_peek_collect_tl }
    \cs_set:Npn \_peek_false_aux:n ##1
      {
        \group_align_safe_end:
        #3
      }
    \cs_set_eq:NN \_peek_true:w \_peek_collect_true:w
    \cs_set:Npn \_peek_true_aux:w { \peek_after:Nw #1 }
    \_peek_true_aux:w
  }
37332 \cs_new_protected:Npn \_peek_collect_true:w
  {
    \if_case:w
      \if_catcode:w \exp_not:N \l_peek_token { 1 \exp_stop_f: \fi:
      \if_catcode:w \exp_not:N \l_peek_token } 2 \exp_stop_f: \fi:
      \if_meaning:w \l_peek_token \c_space_token 3 \exp_stop_f: \fi:
      0 \exp_stop_f:
    \exp_after:wN \_peek_collect:N
    \or: \_peek_collect_remove:nw { \c_group_begin_token }
    \or: \_peek_collect_remove:nw { \c_group_end_token }
    \or: \_peek_collect_remove:nw { ~ }
    \fi:
  }
```

```
37346 \cs_new_protected:Npn \__peek_collect:N #1
37347  {
37348      \tl_put_right:Nn \l__peek_collect_tl {#1}
37349      \__peek_true_aux:w
37350  }
37351 \cs_new_protected:Npn \__peek_collect_remove:nw #1
37352  {
37353      \tl_put_right:Nn \l__peek_collect_tl {#1}
37354      \exp_after:wN \__peek_true_remove:w
37355  }
```

(End definition for `\peek_catcode_collect_inline:Nn` and others. These functions are documented on page [314](#).)

```
37356 </package>
```

# Chapter 91

## I3deprecation implementation

```
37357  <*package>
37358  <@=deprecation>
```

### 91.1 Patching definitions to deprecate

```
\_\_kernel\_patch_deprecation:nnNpn {\langle date\rangle} {\langle replacement\rangle} {\langle definition\rangle}
{\langle function\rangle} {\langle parameters\rangle} {\langle code\rangle}
```

defines the *⟨function⟩* to produce a warning and run its *⟨code⟩*, or to produce an error and not run any *⟨code⟩*, depending on the `expl3` date.

- If the `expl3` date is less than the *⟨date⟩* (plus 6 months in case `undo-recent-deprecations` is used) then we define the *⟨function⟩* to produce a warning and run its code. The warning is actually suppressed in two cases:
  - if neither `undo-recent-deprecations` nor `enable-debug` are in effect we may be in an end-user’s document so it is suppressed;
  - if the command is expandable then we cannot produce a warning.
- Otherwise, we define the *⟨function⟩* to produce an error.

In both cases we additionally make `\debug_on:n {deprecation}` turn the *⟨function⟩* into an `\outer` error, and `\debug_off:n {deprecation}` restore whatever the behaviour was without `\debug_on:n {deprecation}`.

In later sections we use the `I3doc` key `deprecated` with a date equal to that *⟨date⟩* plus 6 months, so that `I3doc` will complain if we forget to remove the stale *⟨parameters⟩* and *⟨code⟩*.

In the explanations below, *⟨definition⟩ ⟨function⟩ ⟨parameters⟩ ⟨code⟩* or assignments that only differ in the scope of the *⟨definition⟩* will be called “the standard definition”.

```
\_\_kernel_patch_deprecation:nnNpn
  \_\_deprecation_patch_aux:nnNnn
    \_\_deprecation_warn_once:nnNnn
\_\_deprecation_patch_aux:Nn
  \_\_deprecation_just_error:nnNN
```

(The parameter text is grabbed using `#5#.`) The arguments of `\_\_kernel_deprecation_code:nn` are run upon `\debug_on:n {deprecation}` and `\debug_off:n {deprecation}`, respectively. In both scenarios we the *⟨function⟩* may be `\outer` so we undefine it with `\tex_let:D` before redefining it, with `\_\_kernel_deprecation_error:Nnn` or with some code added shortly.

```

37359 \cs_new_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
37360   { \__deprecation_patch_aux:nnNNnn {#1} {#2} #3 #4 {#5} }
37361 \cs_new_protected:Npn \__deprecation_patch_aux:nnNNnn #1#2#3#4#5#6
37362   {
37363     \__kernel_deprecation_code:nn
37364     {
37365       \tex_let:D #4 \scan_stop:
37366       \__kernel_deprecation_error:Nnn #4 {#2} {#1}
37367     }
37368     { \tex_let:D #4 \scan_stop: }
37369     \cs_if_eq:NNTF #3 \cs_gset_protected:Npn
37370       { \__deprecation_warn_once:nnNnn {#1} {#2} #4 {#5} {#6} }
37371       { \__deprecation_patch_aux:Nn #3 { #4 #5 {#6} } }
37372   }

```

In case we want a warning, the *<function>* is defined to produce such a warning without grabbing any argument, then redefine itself to the standard definition that the *<function>* should have, with arguments, and call that definition. The x-type expansion and \exp-not:n avoid needing to double the #, which we could not do anyways. We then deal with the code for \debug\_off:n {deprecation}: presumably someone doing that does not need the warning so we simply do the standard definition.

```

37373 \cs_new_protected:Npn \__deprecation_warn_once:nnNnn #1#2#3#4#5
37374   {
37375     \cs_gset_protected:Npx #3
37376     {
37377       \__kernel_if_debug:TF
37378       {
37379         \exp_not:N \msg_warning:nnxxx
37380           { deprecation } { deprecated-command }
37381           {#1}
37382           { \token_to_str:N #3 }
37383           { \tl_to_str:n {#2} }
37384       }
37385       { }
37386     \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
37387     \exp_not:N #3
37388   }
37389   \__kernel_deprecation_code:nn { }
37390   { \cs_set_protected:Npn #3 #4 {#5} }
37391 }

```

In case we want neither warning nor error, the *<function>* is given its standard definition. Here #1 is \cs\_new:Npn or \cs\_new\_protected:Npn and #2 is *<function> <parameters> {<code>}*, so #1#2 performs the assignment. For \debug\_off:n {deprecation} we want to use the same assignment but with a different scope, hence the \cs\_if\_eq:NNTF test.

```

37392 \cs_new_protected:Npn \__deprecation_patch_aux:Nn #1#2
37393   {
37394     #1 #2
37395     \cs_if_eq:NNTF #1 \cs_gset_protected:Npn
37396       { \__kernel_deprecation_code:nn { } { \cs_set_protected:Npn #2 } }
37397       { \__kernel_deprecation_code:nn { } { \cs_set:Npn #2 } }
37398   }

```

(End definition for \\_\_kernel\_patch\_deprecation:nnNNpn and others.)

```
\_kernel_deprecation_error:Nnn
```

The `\outer` definition here ensures the command cannot appear in an argument. Use this auxiliary on all commands that have been removed since 2015.

```
37399 \cs_new_protected:Npn \_kernel_deprecation_error:Nnn #1#2#3
37400 {
37401     \tex_protected:D \tex_outer:D \tex_eodef:D #1
37402     {
37403         \exp_not:N \msg_expandable_error:nnnnn
37404             { deprecation } { deprecated-command }
37405             { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
37406         \exp_not:N \msg_error:nnxxx
37407             { deprecation } { deprecated-command }
37408             { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
37409     }
37410 }
```

(End definition for `\_kernel_deprecation_error:Nnn`.)

```
37411 \msg_new:nnn { deprecation } { deprecated-command }
37412 {
37413     \tl_if_blank:nF {#3} { Use~ \tl_trim_spaces:n {#3} ~not~ }
37414     #2-deprecated-on~#1.
37415 }
```

## 91.2 Removed functions

```
\_deprecation_old_protected:Nnn  
\_deprecation_old:Nnn
```

Short-hands for old commands whose definition does not matter any more as they were removed.

```
37416 \cs_new_protected:Npn \_deprecation_old_protected:Nnn #1#2#3
37417 {
37418     \_kernel_patch_deprecation:nnNNpn {#3} {#2}
37419     \cs_gset_protected:Npn #1 { }
37420 }
37421 \cs_new_protected:Npn \_deprecation_old:Nnn #1#2#3
37422 {
37423     \_kernel_patch_deprecation:nnNNpn {#3} {#2}
37424     \cs_gset:Npn #1 { }
37425 }
37426 \_deprecation_old_protected:Nnn \box_gset_eq_clear:NN
37427     { \box_gset_eq_drop:NN } { 2021-07-01 }
37428 \_deprecation_old_protected:Nnn \box_set_eq_clear:NN
37429     { \box_set_eq_drop:NN } { 2021-07-01 }
37430 \_deprecation_old_protected:Nnn \box_resize:Nnn
37431     { \box_resize_to_wd_and_ht_plus_dp:Nnn } { 2019-01-01 }
37432 \_deprecation_old_protected:Nnn \box_use_clear:N
37433     { \box_use_drop:N } { 2019-01-01 }
37434 \_deprecation_old:Nnn \c_job_name_tl
37435     { \c_sys_jobname_str } { 2017-01-01 }
37436 \_deprecation_old:Nnn \c_minus_one
37437     { -1 } { 2019-01-01 }
37438 \_deprecation_old:Nnn \c_zero
37439     { 0 } { 2020-01-01 }
37440 \_deprecation_old:Nnn \c_one
37441     { 1 } { 2020-01-01 }
```

```

37442 \__deprecation_old:Nnn \c_two
37443   { 2 } { 2020-01-01 }
37444 \__deprecation_old:Nnn \c_three
37445   { 3 } { 2020-01-01 }
37446 \__deprecation_old:Nnn \c_four
37447   { 4 } { 2020-01-01 }
37448 \__deprecation_old:Nnn \c_five
37449   { 5 } { 2020-01-01 }
37450 \__deprecation_old:Nnn \c_six
37451   { 6 } { 2020-01-01 }
37452 \__deprecation_old:Nnn \c_seven
37453   { 7 } { 2020-01-01 }
37454 \__deprecation_old:Nnn \c_eight
37455   { 8 } { 2020-01-01 }
37456 \__deprecation_old:Nnn \c_nine
37457   { 9 } { 2020-01-01 }
37458 \__deprecation_old:Nnn \c_ten
37459   { 10 } { 2020-01-01 }
37460 \__deprecation_old:Nnn \c_eleven
37461   { 11 } { 2020-01-01 }
37462 \__deprecation_old:Nnn \c_twelve
37463   { 12 } { 2020-01-01 }
37464 \__deprecation_old:Nnn \c_thirteen
37465   { 13 } { 2020-01-01 }
37466 \__deprecation_old:Nnn \c_fourteen
37467   { 14 } { 2020-01-01 }
37468 \__deprecation_old:Nnn \c_fifteen
37469   { 15 } { 2020-01-01 }
37470 \__deprecation_old:Nnn \c_sixteen
37471   { 16 } { 2020-01-01 }
37472 \__deprecation_old:Nnn \c_thirty_two
37473   { 32 } { 2020-01-01 }
37474 \__deprecation_old:Nnn \c_one_hundred
37475   { 100 } { 2020-01-01 }
37476 \__deprecation_old:Nnn \c_two_hundred_fifty_five
37477   { 255 } { 2020-01-01 }
37478 \__deprecation_old:Nnn \c_two_hundred_fifty_six
37479   { 256 } { 2020-01-01 }
37480 \__deprecation_old:Nnn \c_one_thousand
37481   { 1000 } { 2020-01-01 }
37482 \__deprecation_old:Nnn \c_ten_thousand
37483   { 10000 } { 2020-01-01 }
37484 \__deprecation_old:Nnn \c_term_ior
37485   { -1 } { 2021-07-01 }
37486 \__deprecation_old:Nnn \dim_case:nnn
37487   { \dim_case:nmF } { 2015-07-14 }
37488 \__deprecation_old_protected:Nnn \file_add_path:nN
37489   { \file_get_full_name:nN } { 2019-01-01 }
37490 \__deprecation_old_protected:Nnn \file_if_exist_input:nT
37491   { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
37492 \__deprecation_old_protected:Nnn \file_if_exist_input:nTF
37493   { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
37494 \__deprecation_old_protected:Nnn \file_list:
37495   { \file_log_list: } { 2019-01-01 }

```

```

37496 \__deprecation_old:Nnn \file_path_include:n
37497   { \seq_put_right:Nn \l_file_search_path_seq } { 2019-01-01 }
37498 \__deprecation_old_protected:Nnn \file_path_remove:n
37499   { \seq_remove_all:Nn \l_file_search_path_seq } { 2019-01-01 }
37500 \__deprecation_old:Nnn \g_file_current_name_tl
37501   { \g_file_curr_name_str } { 2019-01-01 }
37502 \__deprecation_old_protected:Nnn \hbox_unpack_clear:N
37503   { \hbox_unpack_drop:N } { 2021-07-01 }
37504 \__deprecation_old:Nnn \int_case:nnn
37505   { \int_case:nnF } { 2015-07-14 }
37506 \__deprecation_old:Nnn \int_from_binary:n
37507   { \int_from_bin:n } { 2016-01-05 }
37508 \__deprecation_old:Nnn \int_from_hexadecimal:n
37509   { \int_from_hex:n } { 2016-01-05 }
37510 \__deprecation_old:Nnn \int_from_octal:n
37511   { \int_from_oct:n } { 2016-01-05 }
37512 \__deprecation_old:Nnn \int_to_binary:n
37513   { \int_to_bin:n } { 2016-01-05 }
37514 \__deprecation_old:Nnn \int_to_hexadecimal:n
37515   { \int_to_hex:n } { 2016-01-05 }
37516 \__deprecation_old:Nnn \int_to_octal:n
37517   { \int_to_oct:n } { 2016-01-05 }
37518 \__deprecation_old_protected:Nnn \ior_get_str:NN
37519   { \ior_str_get:NN } { 2018-03-05 }
37520 \__deprecation_old_protected:Nnn \ior_list_streams:
37521   { \ior_show_list: } { 2019-01-01 }
37522 \__deprecation_old_protected:Nnn \ior_log_streams:
37523   { \ior_log_list: } { 2019-01-01 }
37524 \__deprecation_old_protected:Nnn \iow_list_streams:
37525   { \iow_show_list: } { 2019-01-01 }
37526 \__deprecation_old_protected:Nnn \iow_log_streams:
37527   { \iow_log_list: } { 2019-01-01 }
37528 \__deprecation_old:Nnn \lua_escape_x:n
37529   { \lua_escape:e } { 2020-01-01 }
37530 \__deprecation_old:Nnn \lua_now_x:n
37531   { \lua_now:e } { 2020-01-01 }
37532 \__deprecation_old_protected:Nnn \lua_shipout_x:n
37533   { \lua_shipout_e:n } { 2020-01-01 }
37534 \__deprecation_old:Nnn \luatex_if_engine_p:
37535   { \sys_if_engine_luatex_p: } { 2017-01-01 }
37536 \__deprecation_old:Nnn \luatex_if_engine:F
37537   { \sys_if_engine_luatex:F } { 2017-01-01 }
37538 \__deprecation_old:Nnn \luatex_if_engine:T
37539   { \sys_if_engine_luatex:T } { 2017-01-01 }
37540 \__deprecation_old:Nnn \luatex_if_engine:TF
37541   { \sys_if_engine_luatex:TF } { 2017-01-01 }
37542 \__deprecation_old_protected:Nnn \msg_interrupt:nnn
37543   { [Defined~error~message] } { 2020-01-01 }
37544 \__deprecation_old_protected:Nnn \msg_log:n
37545   { \iow_log:n } { 2020-01-01 }
37546 \__deprecation_old_protected:Nnn \msg_term:n
37547   { \iow_term:n } { 2020-01-01 }
37548 \__deprecation_old:Nnn \pdftex_if_engine_p:
37549   { \sys_if_engine_pdftex_p: } { 2017-01-01 }

```

```

37550 \__deprecation_old:Nnn \pdftex_if_engine:F
37551   { \sys_if_engine_pdftex:F } { 2017-01-01 }
37552 \__deprecation_old:Nnn \pdftex_if_engine:T
37553   { \sys_if_engine_pdftex:T } { 2017-01-01 }
37554 \__deprecation_old:Nnn \pdftex_if_engine:TF
37555   { \sys_if_engine_pdftex:TF } { 2017-01-01 }
37556 \__deprecation_old:Nnn \prop_get:cN
37557   { \prop_item:cN } { 2016-01-05 }
37558 \__deprecation_old:Nnn \prop_get:Nn
37559   { \prop_item:Nn } { 2016-01-05 }
37560 \__deprecation_old:Nnn \quark_if_recursion_tail_break:N
37561   { } { 2015-07-14 }
37562 \__deprecation_old:Nnn \quark_if_recursion_tail_break:n
37563   { } { 2015-07-14 }
37564 \__deprecation_old:Nnn \scan_align_safe_stop:
37565   { protected~commands } { 2017-01-01 }
37566 \__deprecation_old:Nnn \sort_ordered:
37567   { \sort_return_same: } { 2019-01-01 }
37568 \__deprecation_old:Nnn \sort_reversed:
37569   { \sort_return_swapped: } { 2019-01-01 }
37570 \__deprecation_old:Nnn \str_case:nnn
37571   { \str_case:nnF } { 2015-07-14 }
37572 \__deprecation_old:Nnn \str_case:onn
37573   { \str_case:onF } { 2015-07-14 }
37574 \__deprecation_old:Nnn \str_case_x:nn
37575   { \str_case_e:nn } { 2020-01-01 }
37576 \__deprecation_old:Nnn \str_case_x:nnn
37577   { \str_case_e:nnF } { 2015-07-14 }
37578 \__deprecation_old:Nnn \str_case_x:nnT
37579   { \str_case_e:nnT } { 2020-01-01 }
37580 \__deprecation_old:Nnn \str_case_x:nnTF
37581   { \str_case_e:nnTF } { 2020-01-01 }
37582 \__deprecation_old:Nnn \str_case_x:nnF
37583   { \str_case_e:nnF } { 2020-01-01 }
37584 \__deprecation_old:Nnn \str_if_eq_x:p:nn
37585   { \str_if_eq_p:ee } { 2020-01-01 }
37586 \__deprecation_old:Nnn \str_if_eq_x:nnT
37587   { \str_if_eq:eeT } { 2020-01-01 }
37588 \__deprecation_old:Nnn \str_if_eq_x:nnF
37589   { \str_if_eq:eeF } { 2020-01-01 }
37590 \__deprecation_old:Nnn \str_if_eq_x:nnTF
37591   { \str_if_eq:eeTF } { 2020-01-01 }
37592 \__deprecation_old_protected:Nnn \tl_show_analysis:N
37593   { \tl_analysis_show:N } { 2020-01-01 }
37594 \__deprecation_old_protected:Nnn \tl_show_analysis:n
37595   { \tl_analysis_show:n } { 2020-01-01 }
37596 \__deprecation_old:Nnn \tl_case:cnn
37597   { \tl_case:cNF } { 2015-07-14 }
37598 \__deprecation_old:Nnn \tl_case:Nnn
37599   { \tl_case:NnF } { 2015-07-14 }
37600 \__deprecation_old_protected:Nnn \tl_gset_from_file:Nnn
37601   { \file_get:nnN } { 2021-07-01 }
37602 \__deprecation_old_protected:Nnn \tl_gset_from_file_x:Nnn
37603   { \file_get:nnN } { 2021-07-01 }

```

```

37604 \__deprecation_old_protected:Nnn \tl_set_from_file:Nnn
37605   { \file_get:nnN } { 2021-07-01 }
37606 \__deprecation_old_protected:Nnn \tl_set_from_file_x:Nnn
37607   { \file_get:nnN } { 2021-07-01 }
37608 \__deprecation_old_protected:Nnn \tl_to_lowercase:n
37609   { \tex_lowercase:D } { 2018-03-05 }
37610 \__deprecation_old_protected:Nnn \tl_to_uppercase:n
37611   { \tex_uppercase:D } { 2018-03-05 }
37612 \__deprecation_old:Nnn \token_get_arg_spec:N
37613   { \cs_argument_spec:N } { 2021-07-01 }
37614 \__deprecation_old:Nnn \token_get_prefix_spec:N
37615   { \cs_prefix_spec:N } { 2021-07-01 }
37616 \__deprecation_old:Nnn \token_get_replacement_spec:N
37617   { \cs_replacement_spec:N } { 2021-07-01 }
37618 \__deprecation_old_protected:Nnn \token_new:Nn
37619   { \cs_new_eq:NN } { 2019-01-01 }
37620 \__deprecation_old_protected:Nnn \vbox_unpack_clear:N
37621   { \vbox_unpack_drop:N } { 2021-07-01 }
37622 \__deprecation_old:Nnn \xetex_if_engine_p:
37623   { \sys_if_engine_xetex_p: } { 2017-01-01 }
37624 \__deprecation_old:Nnn \xetex_if_engine:F
37625   { \sys_if_engine_xetex:F } { 2017-01-01 }
37626 \__deprecation_old:Nnn \xetex_if_engine:T
37627   { \sys_if_engine_xetex:T } { 2017-01-01 }
37628 \__deprecation_old:Nnn \xetex_if_engine:TF
37629   { \sys_if_engine_xetex:TF } { 2017-01-01 }

```

(End definition for `\__deprecation_old_protected:Nnn` and `\__deprecation_old:Nnn`.)

### 91.3 Deprecated l3basics functions

37630 `<@=cs>`

`\cs_argument_spec:N` For the present, do not deprecate fully as L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\epsilon$</sub>  will need to catch up: one for Fall 2022.

```

37631 \%__kernel_patch_deprecation:nnNNpn { 2022-06-24 } { \cs_parameter_spec:N }
37632 \cs_gset:Npn \cs_argument_spec:N { \cs_parameter_spec:N }

```

(End definition for `\cs_argument_spec:N`. This function is documented on page ??.)

### 91.4 Deprecated l3str functions

37633 `<@=str>`

```

\str_lower_case:n
\str_lower_case:f
\str_upper_case:n
\str_upper_case:f
\str_fold_case:n
\str_fold_case:v
37634 __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }
37635 \cs_gset:Npn \str_lower_case:n { \str_lowercase:n }
37636 __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:f }
37637 \cs_gset:Npn \str_lower_case:f { \str_lowercase:f }
37638 __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
37639 \cs_gset:Npn \str_upper_case:n { \str_uppercase:n }
37640 __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:f }
37641 \cs_gset:Npn \str_upper_case:f { \str_uppercase:f }

```

```

37642  \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:n }
37643  \cs_gset:Npn \str_fold_case:n { \str_casefold:n }
37644  \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casefold:V }
37645  \cs_gset:Npn \str_fold_case:V { \str_casefold:V }

(End definition for \str_lower_case:n, \str_upper_case:n, and \str_fold_case:n. These functions
are documented on page ??.)
```

\str\_foldcase:n  
\str\_foldcase:V

```

37646  \__kernel_patch_deprecation:nnNNpn { 2020-10-17 } { \str_casefold:n }
37647  \cs_gset:Npn \str_foldcase:n { \str_casefold:n }
37648  \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casefold:V }
37649  \cs_gset:Npn \str_foldcase:V { \str_casefold:V }
```

(End definition for \str\_foldcase:n. This function is documented on page 133.)

\str\_declare\_eight\_bit\_encoding:nnn

This command was made internal, with one more argument. There is no easy way to compute a reasonable value for that extra argument so we take a value that is big enough to accommodate all of Unicode.

```

37650  \__kernel_patch_deprecation:nnNNpn { 2020-08-20 } { }
37651  \cs_gset_protected:Npn \str_declare_eight_bit_encoding:nnn #1
37652    { \__str_declare_eight_bit_encoding:nnnn {#1} { 1114112 } }
```

(End definition for \str\_declare\_eight\_bit\_encoding:nnn. This function is documented on page ??.)

## 91.5 Deprecated l3seq functions

\seq\_indexed\_map\_inline:Nn  
\seq\_indexed\_map\_function:NN

```

37653  \__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_inline:Nn }
37654  \cs_gset_protected:Npn \seq_indexed_map_inline:Nn { \seq_map_indexed_inline:Nn }
37655  \__kernel_patch_deprecation:nnNNpn { 2020-06-18 } { \seq_map_indexed_function:NN }
37656  \cs_gset:Npn \seq_indexed_map_function:NN { \seq_map_indexed_function:NN }
```

(End definition for \seq\_indexed\_map\_inline:Nn and \seq\_indexed\_map\_function:NN. These functions
are documented on page ??.)

## 91.6 Deprecated l3sys functions

37657 <@=sys>

\sys\_load\_deprecation:

```

37658  \__kernel_patch_deprecation:nnNNpn { 2021-01-11 } { (no-longer-required) }
37659  \cs_gset_protected:Npn \sys_load_deprecation: { }
```

(End definition for \sys\_load\_deprecation:. This function is documented on page ??.)

## 91.7 Deprecated **I3tl** functions

```
37660 〈@0=t1〉  
  
\tl_lower_case:n  
\tl_lower_case:nn  
\tl_upper_case:n  
\tl_upper_case:nn  
\tl_mixed_case:n  
\tl_mixed_case:nn  
  
37661  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }  
37662  \cs_gset:Npn \tl_lower_case:n #1  
37663  { \text_lowercase:n {#1} }  
37664  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:nn }  
37665  \cs_gset:Npn \tl_lower_case:nn #1#2  
37666  { \text_lowercase:nn {#1} {#2} }  
37667  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }  
37668  \cs_gset:Npn \tl_upper_case:n #1  
37669  { \text_uppercase:n {#1} }  
37670  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:nn }  
37671  \cs_gset:Npn \tl_upper_case:nn #1#2  
37672  { \text_uppercase:nn {#1} {#2} }  
37673  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase:n }  
37674  \cs_gset:Npn \tl_mixed_case:n #1  
37675  { \text_titlecase:n {#1} }  
37676  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase:nn }  
37677  \cs_gset:Npn \tl_mixed_case:nn #1#2  
37678  { \text_titlecase:nn {#1} {#2} }  
  
(End definition for \tl_lower_case:n and others. These functions are documented on page ??.)
```

## 91.8 Deprecated **I3token** functions

```
\char_to_utfviii_bytes:n  
  
37679  __kernel_patch_deprecation:nnNNpn { 2022-10-09 } { [ \codepoint_generate:n ] }  
37680  \cs_gset:Npn \char_to_utfviii_bytes:n { __kernel_codepoint_to_bytes:n }  
  
(End definition for \char_to_utfviii_bytes:n. This function is documented on page ??.)  
  
\char_to_nfd:Nm  
\char_to_nfd:n  
  
37681  __kernel_patch_deprecation:nnNNpn { 2022-10-09 } { \codepoint_to_nfd:n }  
37682  \cs_gset:Npn \char_to_nfd:N #1 { \codepoint_to_nfd:n {'#1} }  
37683  __kernel_patch_deprecation:nnNNpn { 2022-10-09 } { \codepoint_to_nfd:n }  
37684  \cs_gset:Npn \char_to_nfd:n { \codepoint_to_nfd:n }  
  
(End definition for \char_to_nfd:Nm and \char_to_nfd:n. These functions are documented on page ??.)  
  
\char_lower_case:N  
\char_upper_case:N  
\char_mixed_case:Nn  
\char_fold_case:N  
\char_str_lower_case:N  
\char_str_upper_case:N  
\char_str_mixed_case:Nn  
\char_str_fold_case:N  
  
37685  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_lowercase:n }  
37686  \cs_gset:Npn \char_lower_case:N { \text_lowercase:n }  
37687  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_uppercase:n }  
37688  \cs_gset:Npn \char_upper_case:N { \text_uppercase:n }  
37689  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \text_titlecase:n }  
37690  \cs_gset:Npn \char_mixed_case:N { \text_titlecase:n }  
37691  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casfold:n }  
37692  \cs_gset:Npn \char_fold_case:N { \str_casfold:n }  
37693  __kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_lowercase:n }  
37694  \cs_gset:Npn \char_str_lower_case:N { \str_lowercase:n }
```

```

37695 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_uppercase:n }
37696 \cs_gset:Npn \char_str_upper_case:N { \str_uppercase:n }
37697 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_titlecase:n }
37698 \cs_gset:Npn \char_str_mixed_case:N { \str_titlecase:n }
37699 \__kernel_patch_deprecation:nnNNpn { 2020-01-03 } { \str_casfold:n }
37700 \cs_gset:Npn \char_str_fold_case:N { \str_casfold:n }

```

(End definition for `\char_lowercase:N` and others. These functions are documented on page ??.)

```

\char_lowercase:N
\char_uppercase:N
\char_foldcase:N
\char_str_lowercase:N
\char_str_uppercase:N
\char_str_foldcase:N
37701 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_lowercase:n }
37702 \cs_gset:Npn \char_lowercase:N { \text_lowercase:n }
37703 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \text_uppercase:n }
37704 \cs_gset:Npn \char_uppercase:N { \text_uppercase:n }
37705 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casfold:n }
37706 \cs_gset:Npn \char_foldcase:N { \str_casfold:n }
37707 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_lowercase:n }
37708 \cs_gset:Npn \char_str_lowercase:N { \str_lowercase:n }
37709 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_uppercase:n }
37710 \cs_gset:Npn \char_str_uppercase:N { \str_uppercase:n }
37711 \__kernel_patch_deprecation:nnNNpn { 2022-10-17 } { \str_casfold:n }
37712 \cs_gset:Npn \char_str_foldcase:N { \str_casfold:n }

```

(End definition for `\char_lowercase:N` and others. These functions are documented on page 198.)

A little extra fun here to deal with the expansion.

```

\peek_catcode_ignore_spaces:NTF
\peek_catcode_remove_ignore_spaces:NTF
\peekCharCode_ignore_spaces:NTF
\peekCharCode_remove_ignore_spaces:NTF
\peek_meaning_ignore_spaces:NTF
\peek_meaning_remove_ignore_spaces:NTF
37713 \tl_map_inline:nn
37714 {
37715     { catcode } { catcode_remove }
37716     { charcode } { charcode_remove }
37717     { meaning } { meaning_remove }
37718 }
37719 {
37720     \use:x
37721     {
37722         \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
37723         \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NTF } #####1#####2#####3
37724         {
37725             \peek_remove_spaces:n
37726             { \exp_not:c { peek_ #1 :NTF } #####1{#####2} {#####3} }
37727         }
37728         \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
37729         \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NT } #####1#####2
37730         {
37731             \peek_remove_spaces:n
37732             { \exp_not:c { peek_ #1 :NT } #####1{#####2} }
37733         }
37734         \__kernel_patch_deprecation:nnNNpn { 2022-01-11 } { \peek_remove_spaces:n }
37735         \cs_gset_protected:Npn \exp_not:c { peek_ #1 _ignore_spaces:NF } #####1#####2
37736         {
37737             \peek_remove_spaces:n
37738             { \exp_not:c { peek_ #1 :NF } #####1{#####2} }
37739         }
37740     }
37741 }

```

*(End definition for \peek\_catcode\_ignore\_spaces:NTF and others. These functions are documented on page ??.)*

37742 ⟨/package⟩

# Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
!	253
\"	18846, 18849, 30140, 32662, 32686, 32692, 32696, 32702, 32706, 32712, 32718, 32725, 32726, 32732, 32736, 32738, 32847
\#	10610, 13966, 18846, 32531
\\$	5460, 13965, 18846, 18849, 32531
\%	10612, 13967, 18846, 32531
\&	9405, 13958, 18846, 18849
\&&	252
\'	30140, 32656, 32683, 32690, 32694, 32699, 32704, 32707, 32709, 32716, 32721, 32722, 32729, 32734, 32737, 32745, 32746, 32793, 32794, 32801, 32802, 32813, 32814, 32819, 32820, 32848, 32849, 32871, 32872, 32875, 32876, 32877, 32878
\(	11112, 11298, 11373, 30166
\)	30166
\*	13571, 13594, 19112, 19114, 19118, 19126
*	253
**	253
+	252, 253
\,	20733, 32543
\-	189
\-	252, 253
\.	30140, 32661, 32749, 32750, 32759, 32760, 32769, 32770, 32787, 32799, 32800, 32850, 32851, 32881, 32882, 32885, 32886
\/	188, 4334
/	253
\:	13964
\:::	42, 387, 414, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2296, 2300, 2301, 2304, 2307, 2314, 2317, 2320, 2326, 2413, 2489, 2491, 2496, 2503, 2507, 2510, 2515, 2530, 2571, 2572, 2573, 2574, 2585
\::N	42, 2292, 2413, 2574
\::V	42, 2320
\::V_unbraced	42, 2488
\::c	42, 2294
\::e	42, 2298, 2413
\::e_unbraced	42, 2488, 2530
\::error	83
\::f	42, 2307, 2573
\::f_unbraced	42, 2488
\::n	42, 793, 2291, 2571, 2574
\::o	42, 2296, 2572
\::o_unbraced	42, 2488, 2571, 2572, 2573, 2574
\::p	42, 387, 2293
\::v	42, 2320
\::v_unbraced	42, 2488
\::x	42, 2314
\::x_unbraced	42, 2488, 2585
<	252
\=	20734, 30140, 32659, 32739, 32740, 32755, 32756, 32778, 32779, 32780, 32807, 32808, 32833, 32834, 32887, 32888
=	252
>	252
?	252
?:	252
\???	619
© commands:	
\@case:nn	29790
\`	2206, 3705, 3708, 3709, 3733, 3734, 3741, 3742, 4232, 4474, 4798, 4799, 6194, 6201, 6202, 6203, 6327, 8145, 8149, 8154, 8188, 8197, 8201, 8206, 8226, 8228, 8229, 8231, 8234, 8236, 8241, 8243, 8245, 8250, 8254, 8257, 8261, 8263, 8267, 8269, 8275, 8277, 8281, 8283, 8287, 8292, 8294, 8336, 8338, 8343, 8345, 8351, 8356, 8357, 8361, 8365, 8375, 8378, 8382, 8383, 8387, 8395, 8421, 8466, 9308, 9326, 9328, 9333, 9334, 9358, 9368, 9375, 9390, 9796, 9804, 9811, 9823, 9824, 9839, 9840, 9847, 9867, 9870, 9871, 9903, 9931, 9962, 9963, 9976, 10024, 10025, 10033, 10040, 10041, 10063, 10079, 10083, 10088, 10095, 10615, 11614, 11618, 11619, 11621, 11627, 11629, 11634, 11635, 11637, 11638, 11640, 11642, 11654, 11664, 11666, 11667, 11668, 11767, 11768, 13960, 14514, 14515, 14518,

14840, 14843, 14844, 14845, 14846,	32775, 32776, 32777, 32831, 32832
14851, 14857, 14862, 14869, 15028,	
15031, 15032, 15033, 15035, 15041,	
15046, 15051, 15202, 15209, 18846,	
21985, 21997, 22003, 23059, 23062,	
23063, 23064, 23071, 23074, 23075,	
29259, 29261, 29262, 29265, 29267,	
29268, 29271, 29273, 29274, 29275,	
29279, 29286, 32529, 34833, 36423,	
36425, 36452, 36454, 36474, 36476,	
36490, 36492, 36499, 36501, 36508,	
36510, 36516, 36529, 36531, 37165	
\{ . . . . . 4747, 8149,	
8154, 8201, 8243, 8245, 8257, 8294,	
8383, 8387, 10609, 13961, 14519,	
18846, 30206, 30207, 30208, 32531	
\} . . . 85, 8148, 8154, 8258, 8294, 8383,	
8387, 10611, 13962, 14519, 18846,	
30206, 30207, 30208, 30209, 32531	
<i>&lt;scope&gt;</i> commands:	
\<scope>_tmpa_<type> . . . . . 5	
\<scope>_tmpb_<type> . . . . . 5	
\_ . . . . . 85, 87, 90, 92, 187, 1797, 3756,	
3923, 4295, 4692, 4697, 4741, 4751,	
4935, 7418, 9841, 10010, 10616,	
11638, 13571, 13594, 14519, 14843,	
14844, 14845, 18846, 18994, 19078,	
19103, 29304, 29310, 29321, 29347,	
29685, 30204, 30205, 32542, 37305	
\^ . . . . . 80, 1893,	
2615, 3806, 3879, 3882, 4693, 4698,	
4699, 4700, 4701, 4704, 4715, 4752,	
4806, 4808, 4810, 4812, 4814, 4816,	
5459, 7369, 7372, 7386, 7389, 7398,	
7401, 7404, 7407, 7421, 7424, 8774,	
9412, 10527, 10563, 13963, 14633,	
14634, 14969, 14970, 15151, 15152,	
15153, 18846, 18849, 18851, 18857,	
18928, 26945, 30140, 32657, 32684,	
32691, 32695, 32700, 32705, 32710,	
32717, 32723, 32724, 32730, 32735,	
32747, 32748, 32765, 32766, 32773,	
32774, 32788, 32789, 32790, 32821,	
32822, 32843, 32844, 32845, 32846	
\~ . . . . . 253	
\_ . . . . . 13969, 18846, 18849, 32531	
\` . . . . . 30140, 32655, 32682, 32689, 32693,	
32698, 32703, 32708, 32715, 32719,	
32720, 32728, 32733, 32873, 32874	
\  . . . . . 252	
\~ . . . . . 85, 4737, 4741,	
4747, 10613, 12330, 13968, 18846,	
18849, 30140, 32535, 32658, 32685,	
32697, 32701, 32711, 32727, 32731,	
	<b>A</b>
\A . . . . . 13572, 13595	
\AA . . . . . 30144, 31869, 32580	
\aa . . . . . 30144, 31869, 32590	
\above . . . . . 190	
\abovedisplayshortskip . . . . . 191	
\abovedisplayskip . . . . . 192	
\abovewithdelims . . . . . 193	
abs . . . . . 253	
\accent . . . . . 194	
acos . . . . . 255	
acosd . . . . . 255	
acot . . . . . 256	
acotd . . . . . 256	
acsc . . . . . 255	
acscd . . . . . 255	
\adjdemerits . . . . . 195	
\adjustspacing . . . . . 912	
\advance . . . . . 196	
\AE . . . . . 30145, 31870, 32581, 32875	
\ae . . . . . 30145, 31870, 32591, 32876	
\afterassignment . . . . . 197	
\aftergroup . . . . . 198	
\alignmark . . . . . 783	
\aligntab . . . . . 784	
asec . . . . . 255	
asecd . . . . . 255	
asin . . . . . 255	
asind . . . . . 255	
atan . . . . . 256	
atand . . . . . 256	
\AtBeginDocument . . . . . 659, 11522	
\atop . . . . . 199	
\atopwithdelims . . . . . 200	
\attribute . . . . . 785	
\attributedef . . . . . 786	
\automaticdiscretionary . . . . . 787	
\automatichyphenmode . . . . . 789	
\automatichyphenpenalty . . . . . 790	
\autospacing . . . . . 1111	
\autoxspacing . . . . . 1112	
	<b>B</b>
\b . . . . . 30140, 32669	
\babelshorthand . . . . . 30161	
\badness . . . . . 201	
\baselineskip . . . . . 202	
\batchmode . . . . . 203	
\begin . . . . . 11608, 11611, 18938,	
25029, 30159, 30169, 31255, 32526	
\beginincsname . . . . . 792	

\begingroup . . . . . 3,  
   20, 24, 29, 33, 52, 84, 89, 103, 181, 204  
 \beginL . . . . . 512  
 \beginR . . . . . 513  
 \belowdisplayshortskip . . . . . 205  
 \belowdisplayskip . . . . . 206  
 \bfseries . . . . . 32505  
 \binoppenalty . . . . . 207  
 \bodydir . . . . . 793  
 \bodydirection . . . . . 794  
 bool commands:  
   \bool\_case\_false:n . . . . . 309, 36884, 36898  
   \bool\_case\_false:nTF . . . . .  
     . . . . . 309, 36884, 36892, 36894, 36896  
   \bool\_case\_true:n . . . . . 309, 36884, 36890  
   \bool\_case\_true:nTF . . . . .  
     . . . . . 309, 36884, 36884, 36886, 36888  
   \bool\_const:Nn . . . . . 65, 8475, 8475, 8480  
   \bool\_do\_until:Nn . . . . .  
     . . . . . 68, 8684, 8686, 8687, 8689  
   \bool\_do\_until:nn . . . . . 69, 8690, 8711, 8714  
   \bool\_do\_while:Nn . . . . .  
     . . . . . 69, 8684, 8684, 8685, 8688  
   \bool\_do\_while:nn . . . . . 69, 8690, 8698, 8701  
 .\bool\_gset:N . . . . . 224, 21259  
 \bool\_gset:Nn . . . . . 65, 8497, 8502, 8508  
 \bool\_gset\_eq:NN . . . . .  
   . . . . . 65, 4683, 6852, 8493, 8494, 8496  
 \bool\_gset\_false:N . . . . . 65, 6799,  
   8481, 8487, 8492, 14173, 14182, 36880  
 .\bool\_gset\_inverse:N . . . . . 224, 21267  
 \bool\_gset\_inverse:N . . . . .  
   . . . . . 308, 36876, 36879, 36881  
 \bool\_gset\_true:N . . . . . 65,  
   6865, 8481, 8485, 8491, 8952, 14163,  
   36579, 36600, 36607, 36728, 36880  
 \bool\_if:N . . . . . 8514, 8522  
 \bool\_if:n . . . . . 8557  
 \bool\_if:NTF . . . . . 65, 145, 2074, 5575,  
   5584, 6025, 6198, 6284, 6302, 6320,  
   6474, 6693, 6701, 6934, 7537, 7560,  
   7633, 7855, 8025, 8031, 8072, 8439,  
   8444, 8514, 8523, 8576, 8679, 8681,  
   8685, 8687, 8950, 10830, 10837,  
   14177, 14186, 19756, 19764, 20770,  
   20863, 20942, 21180, 21189, 21235,  
   21451, 21453, 21455, 21501, 21503,  
   21505, 21543, 21545, 21547, 21563,  
   21565, 21567, 21614, 21655, 21674,  
   21676, 21681, 21688, 21752, 21781,  
   21791, 21819, 30967, 34005, 34700,  
   36584, 36662, 36877, 36880, 36902  
 \bool\_if:nTF . . . . .  
   . . . . . 65, 67, 69, 890, 6287, 8525,

  8557, 8629, 8636, 8655, 8662, 8671,  
   8692, 8701, 8705, 8714, 8787, 11725,  
   12069, 12074, 36907, 36913, 36967  
 \bool\_if\_exist:N . . . . . 8553, 8555  
 \bool\_if\_exist:NTF . . . . . 66, 8553, 20965  
 \bool\_if\_exist\_p:N . . . . . 66, 8553  
 \bool\_if\_p:N . . . . . 65, 8514  
 \bool\_if\_p:n . . . . .  
   . . . . . 67, 580, 8478, 8500, 8505, 8557,  
   8565, 8565, 8636, 8662, 8668, 8672  
 \bool\_lazy\_all:n . . . . . 8618  
 \bool\_lazy\_all:nTF . . . . .  
   . . . . . 67, 68, 5858, 8616, 36685  
 \bool\_lazy\_all\_p:n . . . . . 68, 8616  
 \bool\_lazy\_and:nn . . . . . 8633  
 \bool\_lazy\_and:nnTF . . . . .  
   . . . . . 67, 68, 8633, 8859, 9080,  
   10403, 11595, 29085, 29879, 30170,  
   30376, 30482, 30544, 31003, 31664,  
   31727, 31765, 32008, 32415, 33729,  
   34613, 36572, 36623, 36694, 36706  
 \bool\_lazy\_and\_p:nn . . . . .  
   . . . . . 67, 68, 8633, 30956,  
   30974, 31522, 31819, 36634, 36646  
 \bool\_lazy\_any:n . . . . . 8644  
 \bool\_lazy\_any:nTF . . . . . 67, 68,  
   8642, 11210, 14015, 14287, 14311,  
   14333, 14503, 30006, 30184, 30268  
 \bool\_lazy\_any\_p:n . . . . .  
   . . . . . 67, 68, 8642, 30379, 31671  
 \bool\_lazy\_or:nn . . . . . 8659  
 \bool\_lazy\_or:nnTF . . . . . 67,  
   68, 3812, 8659, 8842, 8915, 15287,  
   19022, 29210, 29236, 29298, 29914,  
   30017, 30062, 30285, 30293, 30535,  
   30953, 30971, 31058, 31210, 31270,  
   31331, 31495, 31519, 31746, 31817,  
   32062, 32230, 32450, 32544, 32629,  
   35024, 35364, 36631, 36643, 36702  
 \bool\_lazy\_or\_p:nn . . . . . 68, 8659,  
   30547, 31666, 31729, 31768, 32418  
 \bool\_log:N . . . . . 66, 8530, 8532, 8533  
 \bool\_log:n . . . . . 66, 8526, 8528  
 \bool\_new:N . . . . . 65, 4540,  
   4970, 6768, 6769, 6771, 6772, 6773,  
   8473, 8473, 8474, 8549, 8550, 8551,  
   8552, 8947, 10558, 14026, 20597,  
   20836, 20837, 20844, 20845, 20849,  
   20852, 20965, 30607, 33602, 36571  
 \bool\_not\_p:n . . . . . 68, 8668, 8668  
 .\bool\_set:N . . . . . 224, 21259  
 \bool\_set:Nn . . . . .  
   . . . . . 65, 572, 576, 8497, 8497, 8507

\bool\_set\_eq:NN . . . . . 65, 4677,  
   7013, 8493, 8493, 8495, 19774, 19776  
\bool\_set\_false:N . . . . .  
   .. 65, 160, 5549, 5754, 6743, 6814,  
   6828, 6891, 6933, 8481, 8483, 8490,  
   10664, 10806, 10814, 10822, 10832,  
   10839, 20884, 21445, 21446, 21447,  
   21497, 21498, 21502, 21538, 21546,  
   21548, 21557, 21558, 21568, 21587,  
   21597, 21662, 34001, 34698, 36877  
.bool\_set\_inverse:N . . . . . 224, 21267  
\bool\_set\_inverse:N . . . . .  
   ..... 308, 36876, 36876, 36878  
\bool\_set\_true:N . . . . . 65,  
   174, 5554, 5758, 6737, 6931, 7012,  
   8481, 8481, 8489, 10792, 20879,  
   21452, 21454, 21456, 21496, 21504,  
   21506, 21539, 21540, 21544, 21559,  
   21564, 21566, 21584, 21592, 21669,  
   30608, 34019, 34041, 34072, 36877  
\bool\_show:N . . . . . 66, 8530, 8530, 8531  
\bool\_show:n . . . . . 66, 8526, 8526  
\bool\_to\_str:N . . 65, 8523, 8523, 8524  
\bool\_to\_str:n . . . . .  
   ..... 65, 8523, 8525, 8527, 8529  
\bool\_until\_do:Nn . . . . .  
   ..... 69, 8678, 8680, 8681, 8683  
\bool\_until\_do:nn . . 69, 8690, 8703, 8708  
\bool\_while\_do:Nn . . . . .  
   ..... 69, 8678, 8678, 8679, 8682  
\bool\_while\_do:nn . . 69, 8690, 8690, 8695  
\bool\_xor:nn . . . . . 8669  
\bool\_xor:nnTF . . . . . 68, 8669  
\bool\_xor\_p:nn . . . . . 68, 8669  
\c\_false\_bool . . . . . 26,  
   64, 372, 407, 558, 573, 576–578,  
   1648, 1700, 1701, 1732, 1754, 1759,  
   1791, 1810, 2011, 2018, 2948, 3222,  
   5235, 5253, 5447, 5494, 5793, 5995,  
   6012, 6025, 6221, 6357, 6862, 7960,  
   7969, 7978, 7988, 8050, 8058, 8473,  
   8484, 8488, 8541, 8576, 8607, 8630,  
   8636, 8654, 8798, 19758, 19766,  
   21203, 21205, 21212, 21217, 36587,  
   36690, 36893, 36895, 36897, 36899  
\g\_tmpa\_bool . . . . . 66, 8549  
\l\_tmpa\_bool . . . . . 66, 8549  
\g\_tmpb\_bool . . . . . 66, 8549  
\l\_tmpb\_bool . . . . . 66, 8549  
\c\_true\_bool . . . . . 26, 64,  
   372, 573, 576–578, 689, 1700, 1732,  
   1791, 1809, 2032, 4547, 4680, 5118,  
   5192, 5249, 5437, 5439, 5441, 5443,  
   5445, 5455, 5493, 5500, 5993, 6003,  
   6025, 6026, 6219, 6340, 6342, 6365,  
   6460, 6631, 6642, 6657, 6818, 7584,  
   7701, 7814, 8482, 8486, 8540, 8576,  
   8608, 8609, 8628, 8656, 8662, 8792,  
   19757, 19765, 19774, 21210, 21219,  
   36690, 36885, 36887, 36889, 36891  
bool internal commands:  
\\_\_bool\_!:\_Nw . . . . . 8587  
\\_\_bool\_&\_0: . . . . . 8599  
\\_\_bool\_&\_1: . . . . . 8599  
\\_\_bool\_&\_2: . . . . . 8599  
\\_\_bool\_(:\_Nw . . . . . 8592  
\\_\_bool\_)\_0: . . . . . 8599  
\\_\_bool\_)\_1: . . . . . 8599  
\\_\_bool\_)\_2: . . . . . 8599  
\\_\_bool\_case:NnTF . . . . .  
   36884, 36885, 36887, 36889, 36891,  
   36893, 36895, 36897, 36899, 36900  
\\_\_bool\_case\_end:nw . . . . .  
   ..... 36884, 36908, 36915, 36917  
\\_\_bool\_case\_false:w . . . . .  
   ..... 36884, 36902, 36911, 36914  
\\_\_bool\_case\_true:w . . . . .  
   ..... 36884, 36902, 36905, 36909  
\\_\_bool\_choose:NNN . . . . .  
   ..... 8594, 8598, 8599, 8599  
\\_\_bool\_get\_next>NN . . . . . 576,  
   577, 8573, 8577, 8577, 8589, 8595,  
   8610, 8611, 8612, 8613, 8614, 8615  
\\_\_bool\_if\_p:n . . . . . 8565, 8565, 8566  
\\_\_bool\_if\_p\_aux:w . . . . .  
   ..... 576, 8565, 8568, 8575  
\\_\_bool\_if\_recursion\_tail\_stop\_-  
   do:nn . . . . . 8513, 8513, 8628, 8654  
\\_\_bool\_lazy\_all:n . . . . .  
   ..... 8616, 8617, 8626, 8631  
\\_\_bool\_lazy\_any:n . . . . .  
   ..... 8642, 8643, 8652, 8657  
\\_\_bool\_p:Nw . . . . . 8597  
\\_\_bool\_show>NN . . 8530, 8530, 8532, 8534  
\\_\_bool\_use\_i\_delimit\_by\_q\_-  
   recursion\_stop:nw . . . . .  
   ..... 8511, 8511, 8630, 8656  
\\_\_bool\_|\_0: . . . . . 8599  
\\_\_bool\_|\_1: . . . . . 8599  
\\_\_bool\_|\_2: . . . . . 8599  
\botmark . . . . . 208  
\botmarks . . . . . 514  
\box . . . . . 209  
box commands:  
\box\_autosize\_to\_wd\_and\_ht:Nnn ..  
   ... 282, 33508, 33508, 33510, 33513  
\box\_autosize\_to\_wd\_and\_ht\_plus\_-  
   dp:Nnn . . . 282, 33508, 33514, 33519

\box\_clear:N 273, 274, 32906, 32906,  
   32910, 32913, 33637, 33724, 33801  
 \box\_clear\_new:N . . . . .  
   . . . . . 274, 32912, 32912, 32916  
 \box\_clip:N . . 306, 36734, 36734, 36736  
 \box\_dp:N . . . . . 275, 1290,  
   23547, 32934, 32935, 32938, 32941,  
   32946, 32950, 33253, 33382, 33497,  
   33516, 33522, 33838, 33839, 33945,  
   33950, 33978, 33992, 34163, 34441,  
   34462, 34765, 36754, 36761, 36766  
 \box\_gautosize\_to\_wd\_and\_ht:Nnn .  
   . . . . . 282, 33508, 33511  
 \box\_gautosize\_to\_wd\_and\_ht\_-  
   plus\_dp:Nnn 282, 33508, 33520, 33525  
 \box\_gclear:N . . . . .  
   273, 32906, 32908, 32911, 32915, 33646  
 \box\_gclear\_new:N . . . . .  
   . . . . . 274, 32912, 32914, 32917  
 \box\_gclip:N . . 306, 36734, 36737, 36739  
 \box\_gresize\_to\_ht:Nn . . . . .  
   . . . . . 282, 33401, 33404, 33406  
 \box\_gresize\_to\_ht\_plus\_dp:Nn .  
   . . . . . 283, 33401, 33424, 33426  
 \box\_gresize\_to\_wd:Nn . . . . .  
   . . . . . 283, 33401, 33444, 33446  
 \box\_gresize\_to\_wd\_and\_ht:Nnn . .  
   . . . . . 283, 33401, 33461, 33463  
 \box\_gresize\_to\_wd\_and\_ht\_plus\_-  
   dp:Nnn . . . . .  
   . . . . . 283, 33352, 33358, 33363, 34277  
 \box\_grotate:Nn . . . . .  
   . . . . . 284, 33234, 33237, 33239, 34112  
 \box\_gscales:Nnn . . . . .  
   . . . . . 284, 33479, 33482, 33484, 34319  
 \box\_gset\_dp:Nn . . . . .  
   . . . . . 275, 32943, 32949, 32951  
 \box\_gset\_eq:NN 274, 32909, 32918,  
   32920, 32923, 33823, 36744, 36795  
 \box\_gset\_eq\_clear:NN . . . . . 37426  
 \box\_gset\_eq\_drop:NN . . . . .  
   . . . . . 281, 32924, 32926, 32929, 37427  
 \box\_gset\_ht:Nn . . . . .  
   . . . . . 275, 32943, 32958, 32960  
 \box\_gset\_to\_last:N . . . . .  
   . . . . . 276, 32997, 32999, 33002  
 \box\_gset\_trim:Nnnnn . . . . .  
   . . . . . 306, 36740, 36743, 36745  
 \box\_gset\_viewport:Nnnnn . . . . .  
   . . . . . 306, 36791, 36794, 36796  
 \box\_gset\_wd:Nn . . . . .  
   . . . . . 275, 32943, 32967, 32969  
 \box\_ht:N . . . . . 275, 1290,  
   23546, 32934, 32934, 32937, 32941,  
   32955, 32959, 33252, 33381, 33496,  
   33509, 33512, 33516, 33522, 33719,  
   33796, 33840, 33841, 33936, 33941,  
   33978, 33985, 34157, 34161, 34440,  
   34461, 34763, 36771, 36779, 36784  
 \box\_ht\_plus\_dp:N . . . . .  
   . . . . . 275, 32940, 32940, 32942  
 \box\_if\_empty:N . . . . . 32993, 32995  
 \box\_if\_empty:NTF . . . . . 276, 32993  
 \box\_if\_empty\_p:N . . . . . 276, 32993  
 \box\_if\_exist:N . . . . . 32930, 32932  
 \box\_if\_exist:NTF . . . . .  
   . . . . . 274, 32913, 32915, 32930, 33028  
 \box\_if\_exist\_p:N . . . . . 274, 32930  
 \box\_if\_horizontal:N . . 32985, 32989  
 \box\_if\_horizontal:NTF . . 276, 32985  
 \box\_if\_horizontal\_p:N . . 276, 32985  
 \box\_if\_vertical:N . . 32987, 32991  
 \box\_if\_vertical:NTF . . 276, 32985  
 \box\_if\_vertical\_p:N . . 276, 32985  
 \box\_log:N . . . . . 277, 33014, 33014, 33016  
 \box\_log:Nnn . . . . .  
   . . . . . 277, 33014, 33015, 33017, 33025  
 \box\_move\_down:nn . . . . .  
   . . . . . 274, 1390, 32974, 32980,  
   34136, 36758, 36766, 36809, 36816  
 \box\_move\_left:nn . . 274, 32974, 32974  
 \box\_move\_right:nn . . 274, 32974, 32976  
 \box\_move\_up:nn . . . . .  
   . . . . . 274, 32974, 32978, 34481,  
   34760, 36775, 36784, 36823, 36836  
 \box\_new:N . . . . .  
   . . . . . 273, 274, 32900, 32900, 32905,  
   32913, 32915, 33003, 33004, 33005,  
   33006, 33007, 33233, 33578, 33653  
 \box\_resize:Nnn . . . . . 37430  
 \box\_resize\_to\_ht:Nn . . . . .  
   . . . . . 282, 33401, 33401, 33403  
 \box\_resize\_to\_ht\_plus\_dp:Nn . . . . .  
   . . . . . 283, 33401, 33421, 33423  
 \box\_resize\_to\_wd:Nn . . . . .  
   . . . . . 283, 33401, 33441, 33443  
 \box\_resize\_to\_wd\_and\_ht:Nnn . . . . .  
   . . . . . 283, 33401, 33458, 33460  
 \box\_resize\_to\_wd\_and\_ht\_plus\_-  
   dp:Nnn . . . . . 283,  
   33352, 33352, 33357, 34270, 37431  
 \box\_rotate:Nn . . . . .  
   . . . . . 284, 33234, 33234, 33236, 34109  
 \box\_scale:Nnn . . . . .  
   . . . . . 284, 33479, 33479, 33481, 34316  
 \box\_set\_dp:Nn . . . . . 275,  
   1391, 32943, 32943, 32948, 33279,

33550, 33553, 34141, 34441, 34462,  
 34764, 36761, 36769, 36812, 36817  
 $\backslash\text{box\_set\_eq:NN}$  ..... . 274, 32907, 32918, 32918, 32922,  
 33811, 34464, 34768, 36741, 36792  
 $\backslash\text{box\_set\_eq\_clear:NN}$  ..... 37428  
 $\backslash\text{box\_set\_eq\_drop:NN}$  ..... . 281, 32924, 32924, 32928, 37429  
 $\backslash\text{box\_set\_ht:Nn}$  ..... . 275, 32943, 32952, 32957, 33278,  
 33549, 33554, 34139, 34440, 34461,  
 34762, 36778, 36787, 36826, 36839  
 $\backslash\text{box\_set\_to\_last:N}$  ..... . 276, 32997, 32997, 33001  
 $\backslash\text{box\_set\_trim:Nnnnn}$  ..... . 306, 36740, 36740, 36742  
 $\backslash\text{box\_set\_viewport:Nnnnn}$  ..... . 306, 36791, 36791, 36793  
 $\backslash\text{box\_set\_wd:Nn}$  ..... . 275, 32943, 32961, 32966, 33280,  
 33566, 34142, 34442, 34463, 34766  
 $\backslash\text{box\_show:N}$  ..... . 277, 280, 289, 33008, 33008, 33010  
 $\backslash\text{box\_show:Nnn}$  277, 289, 1340, 33008,  
 33009, 33011, 33013, 34802, 34805  
 $\backslash\text{box\_use:N}$  ..... . 274, 32970,  
 32971, 32973, 33267, 34137, 34478,  
 34481, 34757, 34760, 36751, 36802  
 $\backslash\text{box\_use\_clear:N}$  ..... 37432  
 $\backslash\text{box\_use\_drop:N}$  ..... . 281, 32970, 32970, 32972,  
 33282, 33561, 33570, 34144, 34564,  
 34692, 36759, 36767, 36776, 36785,  
 36810, 36816, 36824, 36837, 37433  
 $\backslash\text{box\_wd:N}$  ..... . 275, 23545, 32934,  
 32936, 32939, 32964, 32968, 33254,  
 33383, 33498, 33530, 33842, 33843,  
 33940, 33949, 33967, 33972, 34160,  
 34168, 34362, 34369, 34395, 34442,  
 34463, 34479, 34758, 34767, 36803  
 $\backslash\text{c\_empty\_box}$  ..... . 273, 276, 32907, 32909, 33003  
 $\backslash\text{g\_tmpa\_box}$  ..... 276, 33004  
 $\backslash\text{l\_tmpa\_box}$  ..... 276, 33004  
 $\backslash\text{g\_tmpb\_box}$  ..... 276, 33004  
 $\backslash\text{l\_tmpb\_box}$  ..... 276, 33004  
 box internal commands:  
 $\backslash\text{l\_box\_angle_fp}$  ..... . 33222, 33244, 33245, 33246, 33275  
 $\backslash\text{l\_box\_autosize:NnnnN}$  ... 33508,  
 33509, 33512, 33516, 33522, 33526  
 $\backslash\text{l\_box\_backend\_clip:N}$  .. 36735, 36738  
 $\backslash\text{l\_box\_backend\_rotate:Nn}$  .... 33273  
 $\backslash\text{l\_box\_backend\_scale:Nnn}$  .... 33542  
 $\backslash\text{l\_box\_bottom\_dim}$  ..... 33225,  
 33253, 33310, 33314, 33319, 33325,  
 33330, 33334, 33343, 33345, 33374,  
 33382, 33391, 33435, 33497, 33503  
 $\backslash\text{l\_box\_bottom\_new\_dim}$  ..... . 33229, 33279, 33311, 33322, 33333,  
 33344, 33390, 33502, 33550, 33554  
 $\backslash\text{l\_box\_cos_fp}$  ..... 33223,  
 33246, 33258, 33263, 33290, 33302  
 $\backslash\text{l\_box\_dim\_eval:n}$  ..... . 1290, 32895, 32896, 32899,  
 32941, 32946, 32950, 32955, 32959,  
 32964, 32968, 32975, 32977, 32979,  
 32981, 33060, 33065, 33092, 33098,  
 33106, 33130, 33164, 33169, 33197,  
 33203, 33214, 33219, 36812, 36836  
 $\backslash\text{l\_box\_dim\_eval:w}$  32895, 32895, 32897  
 $\backslash\text{l\_box\_internal\_box}$  33233, 33267,  
 33268, 33274, 33278, 33279, 33280,  
 33282, 33540, 33549, 33550, 33553,  
 33554, 33561, 33566, 33570, 36748,  
 36756, 36759, 36761, 36764, 36767,  
 36769, 36771, 36773, 36776, 36778,  
 36779, 36782, 36784, 36785, 36787,  
 36789, 36799, 36807, 36810, 36812,  
 36815, 36816, 36817, 36821, 36824,  
 36826, 36834, 36837, 36839, 36841  
 $\backslash\text{l\_box\_left\_dim}$  ... 33225, 33255,  
 33310, 33312, 33321, 33325, 33330,  
 33336, 33341, 33345, 33384, 33499  
 $\backslash\text{l\_box\_left\_new\_dim}$  33229, 33270,  
 33281, 33313, 33324, 33335, 33346  
 $\backslash\text{l\_box\_log:nNnn}$  . 33014, 33018, 33019  
 $\backslash\text{l\_box\_resize:N}$  ... 33352, 33376,  
 33386, 33418, 33438, 33455, 33476  
 $\backslash\text{l\_box\_resize:NNN}$  ..... . 33352, 33388, 33390, 33392, 33396  
 $\backslash\text{l\_box\_resize\_common:N}$  ..... .  
 ... 33394, 33506, 33538, 33538  
 $\backslash\text{l\_box\_resize\_set\_corners:N}$  ..... .  
 ... 33352, 33368,  
 33379, 33411, 33431, 33451, 33468  
 $\backslash\text{l\_box\_resize\_to\_ht:Nnn}$  ..... .  
 ... 33401, 33402, 33405, 33407  
 $\backslash\text{l\_box\_resize\_to\_ht\_plus\_dp:NnN}$  .  
 ... 33401, 33422, 33425, 33427  
 $\backslash\text{l\_box\_resize\_to\_wd:Nnn}$  ..... .  
 ... 33401, 33442, 33445, 33447  
 $\backslash\text{l\_box\_resize\_to\_wd\_and\_ht:NnnN}$  .  
 ... 33459, 33462, 33464  
 $\backslash\text{l\_box\_resize\_to\_wd\_and\_ht\_plus\_dp:NnnN}$  . 33352, 33354, 33360, 33364  
 $\backslash\text{l\_box\_resize\_to\_wd\_ht:NnnN}$  .. 33401

\l_box_right_dim ..	33225, 33254, 33308, 33314, 33319, 33323, 33332, 33334, 33343, 33347, 33370, 33383, 33389, 33453, 33470, 33498, 33505	796
\l_box_right_new_dim ..	33229, 33281, 33315, 33326, 33337, 33348, 33388, 33504, 33558, 33560, 33566	210
\_box_rotate:N ..	33234, 33247, 33250	258
\_box_rotate:NnN .....	..... 33234, 33235, 33238, 33240	797
\_box_rotate_quadrant_four: ..	..... 33234, 33265, 33339	211
\_box_rotate_quadrant_one: ..	..... 33234, 33259, 33306	C
\_box_rotate_quadrant_three: ..	..... 33234, 33264, 33328	798
\_box_rotate_quadrant_two: ..	..... 33234, 33260, 33317	258
\_box_rotate_xdir:nnN .....	33234, 33284, 33312, 33314, 33323, 33325, 33334, 33336, 33345, 33347	258
\_box_rotate_ydir:nnN .....	33234, 33295, 33308, 33310, 33319, 33321, 33330, 33332, 33341, 33343	258
\_box_scale:N .....	..... 33479, 33491, 33494, 33535	258
\_box_scale:NnnN .....	..... 33479, 33480, 33483, 33485	258
\l_box_scale_x_fp .....	33350, 33369, 33389, 33417, 33437, 33452, 33454, 33469, 33489, 33505, 33530, 33532, 33533, 33534, 33544, 33556	258
\l_box_scale_y_fp .....	..... 33350, 33371, 33391, 33393, 33412, 33417, 33432, 33437, 33454, 33471, 33490, 33501, 33503, 33531, 33532, 33533, 33534, 33545, 33547	258
\_box_set_trim:NnnnnN .....	..... 36740, 36741, 36744, 36746	258
\_box_set_viewport:NnnnnN .....	..... 36792, 36795, 36797	258
\_box_show>NNnn .....	.. 33012, 33022, 33026, 33026, 33043	258
\l_box_sin_fp .....	.. 33223, 33245, 33256, 33291, 33301	258
\l_box_top_dim	33225, 33252, 33308, 33312, 33321, 33323, 33332, 33336, 33341, 33347, 33374, 33381, 33393, 33415, 33435, 33474, 33496, 33501	258
\l_box_top_new_dim .....	33229, 33278, 33309, 33320, 33331, 33342, 33392, 33500, 33549, 33553	258
\_box_viewport:NnnnnN .....	36791	258
\boxdir .....	..... 795	258

\g\_\_cctab\_group\_seq . . . . .  
     . . . . . 28911, 29078, 29084  
\\_\_cctab\_gset:n . . . . . 28954,  
     28956, 28970, 28988, 29060, 29239  
\\_\_cctab\_gset\_aux:n . . . . .  
     . . . . . 28954, 28957, 28958  
\\_\_cctab\_gstore:Nnn . . . . .  
     28916, 28932, 28941, 28942, 28943,  
     28944, 28946, 28947, 28949, 28950  
\l\_\_cctab\_internal\_a\_t1 . . . . .  
     . 1197, 1198, 28913, 29011, 29012,  
     29037, 29047, 29055, 29058, 29059,  
     29060, 29067, 29069, 29071, 29072  
\l\_\_cctab\_internal\_b\_t1 . . . . .  
     . . . . . 28913, 29084, 29088, 29095  
\g\_\_cctab\_internal\_cctab . . . . . 28993  
\\_\_cctab\_internal\_cctab\_name: . . . . .  
     . . . . . 28993,  
     28996, 29014, 29015, 29016, 29017  
\\_\_cctab\_item:nN . 29123, 29126, 29130  
\\_\_cctab\_nesting\_number:N . . . . .  
     . . . . . 29058, 29071, 29100, 29101, 29103  
\\_\_cctab\_nesting\_number:w . . . . .  
     . . . . . 29100, 29105, 29110  
\\_\_cctab\_new:N . . . . . 1194, 1197,  
     28916, 28921, 28923, 28930, 28937,  
     28995, 29015, 29036, 29045, 29206  
\g\_\_cctab\_next\_cctab . . . . . 29032  
\\_\_cctab\_select:N . . . . . 1197, 29003,  
     29004, 29008, 29021, 29061, 29072  
\g\_\_cctab\_stack\_seq . . . . .  
     . . . . . 1193, 28909, 29059, 29067, 29118  
\g\_\_cctab\_unused\_seq . . . . .  
     . . . . . 1193, 1198, 28909, 29055, 29069  
ceil . . . . . 254  
\char . . . . . 213, 19291  
char commands:  
   \l\_char\_active\_seq . . . . . 85, 186, 18844  
   \char\_fold\_case:N . . . . . 37685, 37692  
   \char\_foldcase:N . . . . .  
     . . . . . 198, 18984, 18990, 37701, 37706  
   \char\_generate:nn . . . . . 118,  
     183, 434, 454, 455, 541, 683, 743,  
     1405, 3759, 3760, 3761, 3762, 3764,  
     3765, 3766, 4299, 4385, 4401, 4413,  
     4831, 5868, 6242, 12311, 12327,  
     14087, 14342, 14358, 18871, 18871,  
     19019, 19030, 19102, 19103, 29306,  
     29314, 29333, 29336, 29339, 29341,  
     29353, 29384, 29918, 29956, 31579,  
     31590, 31751, 31774, 32553, 32574,  
     32576, 32635, 32649, 32651, 35472  
   \char\_gset\_active\_eq>NN . . . . .  
     . . . . . 183, 18850, 18867  
   \char\_gset\_active\_eq:nN . . . . .  
     . . . . . 183, 18850, 18869  
\char\_lower\_case:N . . . . . 37685, 37686  
\char\_lowercase:N . . . . .  
     . . . . . 198, 18984, 18984, 37701, 37702  
\char\_mixed\_case:N . . . . . 37690  
\char\_mixed\_case:Nn . . . . . 37685  
\char\_set\_active\_eq>NN . . . . .  
     . . . . . 183, 3756, 4295, 18850, 18866  
\char\_set\_active\_eq:nN . . . . .  
     . . . . . 183, 4336, 4337, 18850, 18868  
\char\_set\_catcode:nn . 185, 149, 150,  
     151, 152, 153, 154, 155, 156, 157,  
     18750, 18750, 18757, 18759, 18761,  
     18763, 18765, 18767, 18769, 18771,  
     18773, 18775, 18777, 18779, 18781,  
     18783, 18785, 18787, 18789, 18791,  
     18793, 18795, 18797, 18799, 18801,  
     18803, 18805, 18807, 18809, 18811,  
     18813, 18815, 18817, 18819, 18825  
\char\_set\_catcode\_active:N . 184,  
     3806, 7369, 9405, 18756, 18782,  
     18851, 18928, 19126, 32535, 37304  
\char\_set\_catcode\_active:n . 184,  
     18788, 18814, 18930, 20733, 20734,  
     29213, 29220, 29238, 29249, 29886  
\char\_set\_catcode\_alignment:N . . . . .  
     . . . . . 184, 7421, 18756, 18764, 19114  
\char\_set\_catcode\_alignment:n . . . . .  
     . . . . . 184, 167, 18788, 18796, 18948, 29226  
\char\_set\_catcode\_comment:N . . . . .  
     . . . . . 184, 18756, 18784  
\char\_set\_catcode\_comment:n . . . . .  
     . . . . . 184, 18788, 18816, 29225  
\char\_set\_catcode\_end\_line:N . . . . .  
     . . . . . 184, 18756, 18766  
\char\_set\_catcode\_end\_line:n . . . . .  
     . . . . . 184, 18788, 18798, 29221  
\char\_set\_catcode\_escape:N . . . . .  
     . . . . . 184, 18756, 18756  
\char\_set\_catcode\_escape:n . . . . .  
     . . . . . 184, 18788, 18788, 29228  
\char\_set\_catcode\_group\_begin:N . . . . .  
     . . . . . 184, 3879, 7372, 18756, 18758  
\char\_set\_catcode\_group\_begin:n . . . . .  
     . . . . . 184, 18788, 18790, 18954, 29231  
\char\_set\_catcode\_group\_end:N . . . . .  
     . . . . . 184, 3882, 7389, 18756, 18760  
\char\_set\_catcode\_group\_end:n . . . . .  
     . . . . . 184, 18788, 18792, 18952, 29233  
\char\_set\_catcode\_ignore:N . . . . .  
     . . . . . 184, 18756, 18774  
\char\_set\_catcode\_ignore:n . 184,  
     164, 165, 18788, 18806, 29218, 29222

```

\char_set_catcode_invalid:N . . .
    ..... 184, 18756, 18786
\char_set_catcode_invalid:n . . .
    184, 18788, 18818, 29209, 29212, 29235
\char_set_catcode_letter:N . . .
    184, 7398, 18756, 18778, 25170, 25171
\char_set_catcode_letter:n . . .
    ..... 184, 168, 170, 18788, 18810,
        18934, 29215, 29217, 29227, 29230
\char_set_catcode_math_subscript:N
    ..... 184, 7386, 18756, 18772, 19118
\char_set_catcode_math_subscript:n
    ..... 184, 18788, 18804, 18941, 29248
\char_set_catcode_math_superscript:N
    ..... 184, 7424, 18756, 18770
\char_set_catcode_math_superscript:n
    ..... 184, 169, 18788, 18802, 18943, 29229
\char_set_catcode_math_toggle:N .
    ..... 184, 7401, 18756, 18762, 19112
\char_set_catcode_math_toggle:n .
    ..... 184, 18788, 18794, 18950, 29224
\char_set_catcode_other:N . . .
    ..... 184, 1196, 4044,
        7404, 14633, 14634, 14969, 14970,
        15151, 15152, 15153, 18756, 18780
\char_set_catcode_other:n . . .
    ..... 184, 166,
        171, 18788, 18812, 18932, 29195,
        29214, 29216, 29219, 29232, 29247
\char_set_catcode_parameter:N . . .
    ..... 184, 7407, 18756, 18768
\char_set_catcode_parameter:n . . .
    ..... 184, 18788, 18800, 18945, 29223
\char_set_catcode_space:N . . .
    ..... 184, 18756, 18776
\char_set_catcode_space:n . . .
    ..... 184, 172, 11540, 18788, 18808,
        29200, 29234, 29245, 29246, 29685
\char_set_lccode:nn . . .
    ..... 185, 9401, 9402, 9403, 9404, 18820,
        18826, 18857, 18958, 18959, 37305
\char_set_mathcode:nn . . .
    ..... 186, 18820, 18820
\char_set_sfcode:nn 186, 18820, 18838
\char_set_uccode:nn 185, 18820, 18832
\char_show_value_catcode:n . . .
    ..... 185, 18750, 18754
\char_show_value_lccode:n . . .
    ..... 185, 18820, 18830
\char_show_value_mathcode:n . . .
    ..... 186, 18820, 18824
\char_show_value_sfcode:n . . .
    ..... 186, 18820, 18842
\char_show_value_uccode:n . . .
    ..... 186, 18820, 18836
\l_char_special_seq . . .
    ..... 186, 18844
\char_str_fold_case:N . 37685, 37700
\char_str_foldcase:N . . .
    ..... 198, 18984, 19074, 37701, 37712
\char_str_lower_case:N . 37685, 37694
\char_str_lowercase:N . . .
    ..... 198, 18984, 19068, 37701, 37708
\char_str_mixed_case:N . . .
    ..... 37698
\char_str_mixed_case:Nn . . .
    ..... 37685
\char_str_titlecase:N . . .
    ..... 198, 18984, 19072
\char_str_upper_case:N . 37685, 37696
\char_str_uppercase:N . . .
    ..... 198, 18984, 19070, 37701, 37710
\char_titlecase:N . 198, 18984, 18988
\char_to_nfd:N . . .
    ..... 37682
\char_to_nfd:n . . .
    ..... 37681, 37684
\char_to_nfd:Nm . . .
    ..... 37681
\char_to_utfviii_bytes:n 37679, 37680
\char_upper_case:N . . .
    ..... 37685, 37688
\char_uppercase:N . . .
    ..... 198, 18984, 18986, 37701, 37704
\char_value_catcode:n . . .
    ..... 185, 1201, 149, 150,
        151, 152, 153, 154, 155, 156, 157,
        12323, 12327, 18750, 18752, 18755,
        28975, 29134, 29466, 29919, 31049,
        31053, 31072, 31141, 32554, 32636
\char_value_lccode:n . . .
    ..... 185, 18820, 18828, 18831
\char_value_mathcode:n . . .
    ..... 186, 18820, 18822, 18825
\char_value_sfcode:n . . .
    ..... 186, 18820, 18840, 18843
\char_value_uccode:n . . .
    ..... 186, 18820, 18834, 18837
char internal commands:
\__char_change_case:nN . . .
    ..... 18984,
        18985, 18987, 18989, 18991, 18992
\__char_change_case:nnnN . . .
    ..... 18984, 19002, 19003
\__char_change_case_auxi:nN . . .
    ..... 18984, 18997, 19001
\__char_change_case_auxii:nN . . .
    ..... 18984,
        19008, 19011, 19013, 19017, 19026
\__char_change_case_catcode:N . . .
    ..... 18984, 19020, 19031, 19036
\__char_generate_aux:nn . . .
    ..... 18871
\__char_generate_aux:nnw . 18871,
    18896, 18907, 18915, 18916, 18968
\__char_generate_aux:w . 18873, 18877

```

\\_\_char\_generate\_auxii:nnw .....  
     ..... 18871, 18915, 18922  
 \\_\_char\_generate\_invalid\_-  
     catcode: ..... 18871  
 \\_\_char\_int\_to\_roman:w .....  
     ..... 18870, 18870, 18963, 18978  
 \\_\_char\_quark\_if\_no\_value:N .. 18749  
 \\_\_char\_quark\_if\_no\_value:NTF . 18749  
 \\_\_char\_quark\_if\_no\_value\_p:N . 18749  
 \\_\_char\_str\_change\_case:n ..  
     .. 18984, 19092, 19095, 19097, 19101  
 \\_\_char\_str\_change\_case:nN 18984,  
     19069, 19071, 19073, 19075, 19076  
 \\_\_char\_str\_change\_case:nnN ...  
     ..... 18984, 19086, 19087  
 \\_\_char\_str\_change\_case\_aux:nN ..  
     ..... 18984, 19081, 19085  
 \\_\_char\_tmp:n ..... 18956, 18967  
 \\_\_char\_tmp:nN .. 18852, 18863, 18864  
   \l\_\_char\_tmp\_tl ..... 18871  
 \chardef ..... 1201, 130, 142, 214  
 choice commands:  
   .choice: ..... 224, 21275  
 choices commands:  
   .choices:nn ..... 224, 21277  
 \cite ..... 1227, 30159, 30169  
 \cleaders ..... 215  
 \clearmarks ..... 799  
 clist commands:  
   \clist\_clear:N .....  
     ..... 173, 18119, 18119, 18120,  
           18136, 18299, 21484, 21526, 29641  
   \clist\_clear\_new:N .....  
     ..... 173, 18123, 18123, 18124  
   \clist\_concat:NNN .....  
     174, 18162, 18162, 18175, 18188, 18201  
   \clist\_const:Nn ..  
     .... 173, 18116, 18116, 18118, 29834  
   \clist\_count:N ..... 178, 180,  
     18543, 18543, 18551, 18576, 18641,  
     18707, 18718, 29555, 29584, 29593  
   \clist\_count:n ..... 178, 18543,  
     18555, 18672, 18698, 18719, 35995  
   \clist\_gclear:N .....  
     .... 173, 18119, 18121, 18122, 18138  
   \clist\_gclear\_new:N .....  
     .... 173, 18123, 18125, 18126  
   \clist\_gconcat:NNN .....  
     174, 18162, 18164, 18176, 18190, 18203  
   \clist\_get:NN .....  
     179, 18213, 18213, 18223, 18250, 18259  
   \clist\_get:MNTF ..... 179, 18250  
   \clist\_gpop>NN .....  
     180, 18224, 18226, 18249, 18262, 18274  
   \clist\_gpop:NNTF ..... 180, 18250  
   \clist\_gpush:Nn .....  
     . 180, 18275, 18283, 18284, 18285,  
       18286, 18287, 18288, 18289, 18290  
   \clist\_gput\_left:Nn .....  
     ..... 174, 18187, 18189,  
       18198, 18199, 18283, 18284, 18285,  
       18286, 18287, 18288, 18289, 18290  
   \clist\_gput\_right:Nn .....  
     .... 174, 18200, 18202, 18211, 18212  
   \clist\_gremove\_all:Nn .....  
     ..... 175, 18315, 18317, 18353  
   \clist\_gremove\_duplicates:N ....  
     ..... 175, 18293, 18295, 18314  
   \clist\_greverse:N .....  
     ..... 175, 18354, 18356, 18359  
   \clist\_gset:N ..... 224, 21287  
   \clist\_gset:Nn .....  
     .... 174, 14371, 18181, 18183, 18186  
   \clist\_gset\_eq:NN ..... 173, 18127,  
     18131, 18132, 18133, 18134, 18296  
   \clist\_gset\_from\_seq:NN 173, 3471,  
     18135, 18137, 18160, 18161, 18318  
   \clist\_gsort:Nn .....  
     .... 176, 3456, 3468, 3473, 18372  
   \clist\_if\_empty:N ..... 18372, 18374  
   \clist\_if\_empty:n ..... 18376  
   \clist\_if\_empty:NTF .....  
     . 176, 18171, 18306, 18339, 18372,  
       18429, 18468, 18500, 18706, 21064  
   \clist\_if\_empty:nTF ..... 176, 18376  
   \clist\_if\_empty\_p:N ..... 176, 18372  
   \clist\_if\_empty\_p:n ..... 176, 18376  
   \clist\_if\_exist:N ..... 18177, 18179  
   \clist\_if\_exist:NTF .....  
     .... 174, 11408, 11508, 18177, 18574  
   \clist\_if\_exist\_p:N ..... 174, 18177  
   \clist\_if\_in:Nn ..... 18390, 18423  
   \clist\_if\_in:nn ..... 18394, 18425  
   \clist\_if\_in:NnTF .....  
     .... 173, 176, 953, 18302, 18390  
   \clist\_if\_in:nnTF .. 176, 18390, 22932  
   \clist\_item:Nn .....  
     180, 862, 18638, 18638, 18668, 18707  
   \clist\_item:nn .....  
     .... 180, 862, 18669, 18669, 18702  
   \clist\_log:N . 181, 18710, 18712, 18713  
   \clist\_log:n ..... 181, 18732, 18733  
   \clist\_map\_break: .....  
     . 177, 18434, 18446, 18455, 18456,  
       18478, 18505, 18518, 18526, 18527,  
       18539, 18539, 18540, 18542, 21716

```

\clist_map_break:n ..... 178, 3464, 3470, 18410,
..... 18539, 18541, 21670, 21745, 35972
\clist_map_function:NN ..... 177, 857, 16510, 16520, 18413,
..... 18427, 18427, 18450, 18548, 18723
\clist_map_function:nN ..... 177, 310, 858,
..... 14374, 16515, 16525, 16536, 18451,
..... 18451, 18737, 21847, 36133, 36209
\clist_map_inline:Nn .. 177, 3464,
..... 3470, 9235, 18300, 18466, 18466,
..... 18485, 18487, 21665, 21707, 21736
\clist_map_inline:nn .. 177, 3260,
..... 10160, 11700, 11733, 11745, 18466,
..... 18482, 21018, 21142, 22192, 22378,
..... 29486, 29645, 35550, 35752, 35754,
..... 35790, 35959, 36113, 36157, 36162
\clist_map_tokens:Nn ..... 177, 857, 18489, 18498, 18498, 18522
\clist_map_tokens:nn 177, 18523, 18523
\clist_map_variable:NNn ..... 177, 18488, 18488, 18490, 18496
\clist_map_variable:nNn ..... 177, 18488, 18493
\clist_new:N ..... 173,
..... 844, 18114, 18114, 18115, 18291,
..... 18739, 18740, 18741, 18742, 20832
\clist_pop:NN ..... 179, 18224, 18224, 18248, 18260, 18273
\clist_pop:NNTF ..... 180, 18250
\clist_push:Nn ..... 180, 18275, 18275, 18276, 18277,
..... 18278, 18279, 18280, 18281, 18282
\clist_put_left:Nn ..... 174, 18187, 18187,
..... 18196, 18197, 18275, 18276, 18277,
..... 18278, 18279, 18280, 18281, 18282
\clist_put_right:Nn .. 174, 18200,
..... 18200, 18209, 18210, 21236, 21778,
..... 21788, 21816, 29554, 29592, 29609
\clist_rand_item:N ..... 181, 18697, 18704, 18709
\clist_rand_item:n ..... 74, 181, 18697, 18697
\clist_remove_all:Nn .. 175, 9250, 18315, 18315, 18352, 21237
\clist_remove_duplicates:N .. 173, 175, 18293, 18293, 18313
\clist_reverse:N ..... 175, 18354, 18354, 18358
\clist_reverse:n ..... 175, 853, 18355, 18357, 18360, 18360
\clist_set:N ..... 224, 21287
\clist_set:Nn .. 174, 179, 18181,
..... 18181, 18185, 18188, 18190, 18201,
..... 18203, 18396, 18484, 18495, 21063
\clist_set_eq:NN ..... 173, 18127, 18127, 18128,
..... 18129, 18130, 18294, 21650, 29623
\clist_set_from_seq:NN .. 173, 3465,
..... 18135, 18135, 18158, 18159, 18316
\clist_show:N 181, 18710, 18710, 18711
\clist_show:n ..... 181, 18732, 18732
\clist_sort:Nn ..... 176, 3456, 3462, 3467, 18372
\clist_use:Nn 179, 18572, 18602, 18604
\clist_use:nn .. 179, 18605, 18637
\clist_use:Nnnn ..... 178,
..... 179, 813, 18572, 18572, 18595, 18603
\clist_use:nnnn ..... 179, 18605, 18605, 18637
\c_empty_clist ..... 181, 18061, 18215, 18230, 18252, 18266
\g_tmptpa_clist ..... 181, 18739
\l_tmptpa_clist ..... 181, 18739
\g_tmppb_clist ..... 181, 18739
\l_tmppb_clist ..... 181, 18739
clist internal commands:
\__clist_concat:NNNN ..... 18162, 18163, 18165, 18166
\__clist_count:n .. 18543, 18548, 18552
\__clist_count:w ..... 18543, 18560, 18564, 18568
\__clist_get:wN ..... 18213, 18218, 18221, 18255
\__clist_if_empty_n:w ..... 18376, 18378, 18383, 18386
\__clist_if_empty_n:wNw ..... 18376, 18387, 18389
\__clist_if_in_return:nnN ..... 18390, 18392, 18397, 18400
\__clist_if_wrap:n ..... 18088
\__clist_if_wrap:nTF .. 845, 18088,
..... 18113, 18154, 18307, 18321, 18402
\__clist_if_wrap:w ..... 845, 18088, 18092, 18111
\l__clist_internal_clist ..... 848, 18062, 18193,
..... 18194, 18206, 18207, 18396, 18397,
..... 18398, 18484, 18485, 18495, 18496
\l__clist_internal_remove_clist .
..... 18291, 18299, 18302, 18304, 18306, 18311
\l__clist_internal_remove_seq .. 18291, 18323, 18324, 18325
\__clist_item:nnnN ..... 18638, 18640, 18646, 18661, 18671

```

\\_\_clist\_item\_n:nw 18669, 18675, 18677  
 \\_\_clist\_item\_n\_end:n .....  
     ..... 18669, 18685, 18693  
 \\_\_clist\_item\_N\_loop:nw .....  
     ..... 18638, 18644, 18662, 18666  
 \\_\_clist\_item\_n\_loop:nw .....  
     .. 18669, 18678, 18679, 18682, 18687  
 \\_\_clist\_item\_n\_strip:n .....  
     ..... 18669, 18694, 18695  
 \\_\_clist\_item\_n\_strip:w .....  
     ..... 18669, 18695, 18696  
 \\_\_clist\_map\_function:Nw .....  
     855, 18427, 18431, 18437, 18442, 18473  
 \\_\_clist\_map\_function\_end:w .....  
     .... 855, 18427, 18440, 18444, 18448  
 \\_\_clist\_map\_function\_n:Nn .....  
     .... 856, 18451, 18453, 18458, 18462  
 \\_\_clist\_map\_tokens:nw .....  
     ..... 18498, 18502, 18508, 18514  
 \\_\_clist\_map\_tokens\_end:w .....  
     ..... 18498, 18511, 18516, 18520  
 \\_\_clist\_map\_tokens\_n:nw .....  
     ..... 18523, 18525, 18529, 18537  
 \\_\_clist\_map\_unbrace:wn .....  
     856, 18451, 18461, 18465, 18535, 18622  
 \\_\_clist\_map\_variable:Nnn .....  
     ..... 857, 18488, 18489, 18491  
 \\_\_clist\_pop:NNN .....  
     ..... 18224, 18225, 18227, 18228  
 \\_\_clist\_pop:wN . 18224, 18241, 18247  
 \\_\_clist\_pop:wwNNN .....  
     .... 849, 18224, 18233, 18236, 18269  
 \\_\_clist\_pop\_TF:NNN .....  
     ..... 18250, 18261, 18263, 18264  
 \\_\_clist\_put\_left>NNNn .....  
     ..... 18187, 18188, 18190, 18191  
 \\_\_clist\_put\_right>NNNn .....  
     ..... 18200, 18201, 18203, 18204  
 \\_\_clist\_rand\_item:nn .....  
     ..... 18697, 18698, 18699  
 \\_\_clist\_remove\_all: .....  
     ..... 18315, 18332, 18336, 18349  
 \\_\_clist\_remove\_all>NNNn .....  
     ..... 18315, 18316, 18318, 18319  
 \\_\_clist\_remove\_all:w .....  
     .... 852, 18315, 18350, 18351  
 \\_\_clist\_remove\_duplicates:NN ...  
     ..... 18293, 18294, 18296, 18297  
 \\_\_clist\_reverse:wwNww .....  
     .... 853, 18360, 18362, 18363, 18367  
 \\_\_clist\_reverse\_end:ww .....  
     ..... 853, 18360, 18364, 18370  
 \\_\_clist\_sanitize:n .....  
   ... 18075, 18075, 18117, 18182, 18184  
 \\_\_clist\_sanitize:Nn .....  
     .... 845, 18075, 18077, 18081, 18085  
 \\_\_clist\_set\_from\_seq:n .....  
     ..... 18135, 18147, 18151  
 \\_\_clist\_set\_from\_seq>NNNN .....  
     ..... 18135, 18136, 18138, 18139  
 \\_\_clist\_show:NN .....  
     ..... 18710, 18710, 18712, 18714  
 \\_\_clist\_show:Nn .....  
     ..... 18732, 18732, 18733, 18734  
 \\_\_clist\_tmp:w .....  
     ..... 852,  
         18068, 18068, 18328, 18350, 18404,  
         18413, 18417, 18419, 18553, 18571  
 \\_\_clist\_trim\_next:w .....  
     ..... 845, 856, 18069, 18069,  
         18072, 18078, 18086, 18454, 18463  
 \\_\_clist\_use:Nw 860, 18605, 18607,  
     18608, 18609, 18615, 18618, 18634  
 \\_\_clist\_use:nwwn 18572, 18586, 18600  
 \\_\_clist\_use:nwwwnwn .....  
     .... 859, 18572, 18583, 18585, 18597  
 \\_\_clist\_use:wwn .....  
     ..... 18572, 18579, 18580, 18596  
 \\_\_clist\_use\_end:w .....  
     .... 860, 18605, 18609, 18628, 18634  
 \\_\_clist\_use\_i\_delimit\_by\_s\_-  
     stop:nw ..... 18065, 18067, 18665  
 \\_\_clist\_use\_more:w .....  
     .... 860, 18605, 18610, 18631, 18634  
 \\_\_clist\_use\_none\_delimit\_by\_s\_-  
     mark:w ..... 18065, 18065, 18620  
 \\_\_clist\_use\_none\_delimit\_by\_s\_-  
     stop:w .....  
     .... 852, 18065, 18066, 18083, 18331,  
         18439, 18446, 18460, 18510, 18518,  
         18533, 18566, 18607, 18651, 18656  
 \\_\_clist\_use\_one:w 18605, 18608, 18626  
 \\_\_clist\_wrap\_item:w .....  
     .... 845, 18084, 18112, 18112  
 \closein ..... 216  
 \closeout ..... 217  
 \clubpenalties ..... 515  
 \clubpenalty ..... 218  
 cm ..... 258  
 code commands:  
     .code:n ..... 225, 21285  
 codepoint commands:  
     \codepoint\_generate:n ..... 37679  
     \codepoint\_generate:nn 266, 29298,  
         29308, 29345, 29476, 30944, 30960,  
         30961, 31044, 31048, 31052, 31104,  
         31140, 31274, 31278, 31509, 31530,  
         31544, 31546, 31560, 31562, 31627,

31629, 31631, 31644, 31681, 31707,  
 31788, 31803, 31827, 31835, 31847  
`\codepoint_str_generate:n . . . . .`  
 ..... 266, 13949, 13952, 13954,  
29298, 29302, 29319, 29510, 29546,  
 29730, 29741, 29769, 29793, 29815  
`\codepoint_to_nfd:n . . . . .`  
 ..... 266, 29465, 29465,  
 31120, 37681, 37682, 37683, 37684  
 codepoint internal commands:  
`\__codepoint_add:nn . . . . .`  
 ..... 29541, 29542, 29552  
`\c__codepoint_block_size_int . . .`  
 ..... 29483, 29493, 29556, 29583,  
 29594, 29597, 29602, 29605, 29608,  
 29656, 29670, 29702, 29707, 29719  
`\__codepoint_case:nn . . . . .`  
 .. 29804, 29805, 29806, 29807, 29808  
`\__codepoint_case:nnn . . . . .`  
 ..... 29790, 29792, 29795  
`\__codepoint_casfold:n 29790, 29807`  
`\__codepoint_data:nnn . . . . .`  
 ..... 29696, 29698, 29716  
`\__codepoint_data_auxi:w . 29497,`  
 29691, 29723, 29751, 29756, 29786  
`\__codepoint_data_auxii:w . . . . .`  
 ..... 29503, 29507, 29735,  
 29739, 29759, 29760, 29762, 29764  
`\__codepoint_data_auxiii:w . . . . .`  
 ..... 29505, 29516  
`\__codepoint_data_auxiv:w . . . . .`  
 ..... 29521, 29534  
`\__codepoint_data_auxvi:nnnw . . .`  
 ..... 29538, 29559  
`\g__codepoint_data_ior . . . . .`  
 ..... 29484, 29683, 29686, 29722,  
 29748, 29754, 29755, 29777, 29788  
`\__codepoint_data_offset:nn . . .`  
 ..... 29523, 29524, 29528, 29543  
`\__codepoint_finalise_blocks: . . .`  
 ..... 29643, 29693  
`\__codepoint_finalise_blocks:n . . .`  
 ..... 29648, 29651  
`\__codepoint_finalise_blocks:nnn . . .`  
 ..... 29659, 29667  
`\__codepoint_finalise_blocks:nnnw . . .`  
 ..... 29669, 29674, 29680  
`\__codepoint_generate:n .. 29298,`  
 29370, 29371, 29374, 29376, 29381  
`\__codepoint_generate:nnnn . . .`  
 ..... 29298, 29358, 29364  
`\__codepoint_lowercase:n 29790, 29805`  
`\l__codepoint_matched_block_t1 . . .`  
 .. 29496, 29613, 29618, 29621, 29639  
`\l__codepoint_next_codepoint_--fint_t1 . 29495, 29536, 29549, 29574`  
`\__codepoint_nfd:n 29470, 29814, 29814`  
`\__codepoint_nfd:nn . . . . .`  
 ..... 29814, 29815, 29816  
`\__codepoint_range:nnn . . . . .`  
 ..... 29563, 29564, 29567, 29568, 29571, 29647  
`\__codepoint_range:nnnn . . . . .`  
 ..... 29578, 29589  
`\__codepoint_range_aux:nnn . . . . .`  
 ..... 29573, 29576  
`\__codepoint_save_blocks:nn . . . . .`  
 ..... 29557, 29595, 29604, 29611  
`\__codepoint_str_generate:nnnn . . . . .`  
 ..... 29298, 29326, 29331  
`\__codepoint_titlecase:n 29790, 29806`  
`\l__codepoint_tmpl_t1 . . . . .`  
 ..... 29686, 29688, 29691  
`\__codepoint_to_bytes_auxi:n . . . . .`  
 ..... 29387, 29389, 29392  
`\__codepoint_to_bytes_auxii:Nnn . . . . .`  
 .. 29387, 29397, 29403, 29414, 29436  
`\__codepoint_to_bytes_auxiii:n . . . . .`  
 ..... 29387, 29399, 29406,  
 29410, 29419, 29424, 29428, 29438  
`\__codepoint_to_bytes_end: . . . . .`  
 ..... 29387, 29434, 29441,  
 29444, 29447, 29453, 29461, 29464  
`\__codepoint_to_bytes_output:nnn . . . . .`  
 ..... 29387, 29442,  
 29445, 29449, 29455, 29458, 29463  
`\__codepoint_to_bytes_outputi:nw . . . . .`  
 ..... 29387,  
 29396, 29402, 29412, 29432, 29440  
`\__codepoint_to_bytes_outputii:nw . . . . .`  
 .. 29387, 29398, 29404, 29417, 29443  
`\__codepoint_to_bytes_outputiii:nw . . . . .`  
 ..... 29387, 29409, 29422, 29446  
`\__codepoint_to_bytes_outputiv:nw . . . . .`  
 ..... 29387, 29427, 29452  
`\__codepoint_to_nfd:nn . . . . .`  
 .. 29465, 29466, 29467, 29478, 29480  
`\__codepoint_to_nfd:nnn . . . . .`  
 ..... 29465, 29469, 29472  
`\__codepoint_to_nfd:nnnn . . . . .`  
 ..... 29465, 29472, 29473  
`\__codepoint_to_nfd_generate:n 29465`  
`\__codepoint_to_nfd_generate:nn . . . . .`  
 ..... 29465  
`\__codepoint_to_nfd_generate:nnnn . . . . .`  
 ..... 29465  
`\__codepoint_uppercase:n 29790, 29804`  
coffin commands:  
`\coffin_attach:NnnNnnnn . . . . .`  
 .. 287, 1339, 34424, 34424, 34429

```

\coffin_clear:N ..... 285, 33633, 33633, 33641
\coffin_display_handles:Nn ..... 288, 34670, 34670, 34745
\coffin_dp:N ..... 288, 33838, 33838, 33839, 34288, 34327, 34784
\coffin_gattach:NnnNnnnn ..... 287, 34424, 34430, 34435
\coffin_gclear:N ..... 285, 33633, 33642, 33650
\coffin_gjoin:NnnNnnnn ..... 288, 34373, 34379, 34384
\coffin_gresize:Nnn ..... 287, 34267, 34274, 34280
\coffin_grotate:Nn ..... 287, 34108, 34111, 34113
\coffin_gsacle:Nnn ..... 287, 34315, 34318, 34320
\coffin_gset_eq:NN ..... 285, 33807, 33819, 33830, 34382, 34433
\coffin_gset_horizontal_pole:Nnn ..... 286, 33868, 33871, 33873
\coffin_gset_vertical_pole:Nnn ..... 287, 33868, 33889, 33891
\coffin_ht:N ..... 288, 33838, 33840, 33841, 34288, 34327, 34783
\coffin_if_exist:N ..... 33612, 33622
\coffin_if_exist:NTF 285, 33612, 33626
\coffin_if_exist_p:N ..... 285, 33612
\coffin_join:NnnNnnnn ..... 288, 34373, 34373, 34378
\coffin_log:N 289, 34795, 34798, 34800
\coffin_log:Nnn ..... 289, 34795, 34799, 34804, 34806
\coffin_log_structure:N ..... 289, 34770, 34773, 34775
\coffin_mark_handle:Nnnm ..... 289, 34625, 34625, 34669
\coffin_new:N ..... 285, 1315, 33651, 33651, 33663, 33831, 33832, 33833, 33834, 33835, 33836, 33837, 34557, 34567, 34568, 34569
\coffin_resize:Nnn ..... 287, 34267, 34267, 34273
\coffin_rotate:Nn ..... 287, 34108, 34108, 34110
\coffin_scale:Nnn ..... 287, 34315, 34315, 34317
\coffin_scale:NnnNN ..... 34315
\coffin_set_eq:NN 285, 33807, 33807, 33818, 34376, 34427, 34483, 34686
\coffin_set_horizontal_pole:Nnn ..... 286, 33868, 33868, 33870
\coffin_set_vertical_pole:Nnn ..... 287, 33868, 33886, 33888
\coffin_show:N 289, 34795, 34795, 34797
\coffin_show:Nnn ..... 289, 34795, 34796, 34801, 34803
\coffin_show_structure:N ..... 289, 1340, 34770, 34770, 34772
\coffin_typeset:Nnnnn ..... 288, 34559, 34559, 34566
\coffin_wd:N ..... 288, 33838, 33842, 33843, 34284, 34331, 34785
\c_empty_coffin ..... 289, 33831
\g_tmpa_coffin ..... 290, 33834
\l_tmpa_coffin ..... 289, 33834
\g_tmpb_coffin ..... 290, 33834
\l_tmpb_coffin ..... 289, 33834
coffin internal commands:
\__coffin_align:NnnNnnnnN 34387, 34438, 34459, 34466, 34466, 34562
\l__coffin_aligned_coffin ..... 33831, 34388, 34389, 34393, 34399, 34402, 34405, 34421, 34422, 34439, 34440, 34441, 34442, 34443, 34446, 34450, 34454, 34455, 34460, 34461, 34462, 34463, 34464, 34497, 34513, 34563, 34564, 34755, 34762, 34764, 34766, 34768
\l__coffin_aligned_internal_coffin ..... 33831, 34476, 34483
\__coffin_attach:NnnNnnnnN ..... 34424, 34426, 34432, 34436
\__coffin_attach_mark:NnnNnnnn .. 34424, 34457, 34632, 34648, 34664
\l__coffin_bottom_corner_dim ... 34104, 34136, 34140, 34219, 34230, 34231, 34251, 34259
\l__coffin_bounding_prop ..... 34100, 34127, 34156, 34158, 34164, 34166, 34175, 34238
\l__coffin_bounding_shift_dim ... 34103, 34135, 34237, 34243, 34244
\__coffin_calculate_intersection:Nnn .. 33997, 33997, 34468, 34471, 34748
\__coffin_calculate_intersection:nnnnnn .. 33997, 34061, 34069
\__coffin_calculate_intersection:nnnnnnnn .. 33997, 34003, 34012, 34699
\c__coffin_corners_prop ..... 33581, 33658, 33857, 33864
\l__coffin_corners_prop ..... 34101, 34118, 34122, 34145, 34150, 34181, 34221, 34248, 34295, 34299, 34305, 34311, 34346, 34360

```

```

\l__coffin_cos_fp ..... 1323, 1325, 34098, 34117, 34202, 34211
\l__coffin_display_attach:Nnnnn .. 34670, 34704, 34721, 34740, 34746
\l__coffin_display_coffin .... 34567, 34686, 34692, 34757, 34758, 34763, 34765, 34767, 34768
\l__coffin_display_coord_coffin .. 34567, 34634, 34649, 34665, 34707, 34722, 34741
\l__coffin_display_font_tl .... 34612, 34637, 34710
\l__coffin_display_handles_- aux:nnnn 34670, 34727, 34732, 34738
\l__coffin_display_handles_- aux:nnnnnn ... 34670, 34690, 34694
\l__coffin_display_handles_prop .. 34570, 34640, 34644, 34713, 34717
\l__coffin_display_offset_dim .. 34607, 34666, 34667, 34742, 34743
\l__coffin_display_pole_coffin .. 34567, 34627, 34633, 34672, 34705
\l__coffin_display_poles_prop ... 34611, 34677, 34682, 34685, 34687, 34689, 34696
\l__coffin_display_x_dim ..... 34609, 34702, 34752
\l__coffin_display_y_dim ..... 34609, 34703, 34754
\c__coffin_empty_coffin 34557, 34562
\l__coffin_error_bool ..... 33602, 34001, 34005, 34019, 34041, 34072, 34698, 34700
\l__coffin_find_bounding_shift: .. 34130, 34235, 34235
\l__coffin_find_bounding_shift_- aux:nn ..... 34235, 34239, 34241
\l__coffin_find_corner_maxima:N .. 34129, 34215, 34215
\l__coffin_find_corner_maxima_- aux:nn ..... 34215, 34222, 34224
\l__coffin_get_pole:NnN ... 33844, 33844, 33999, 34000, 34524, 34525, 34528, 34529, 34679, 34680, 34683
\l__coffin_greset_structure:N ... 33647, 33854, 33861, 33918
\l__coffin_gupdate:N ..... 33686, 33699, 33758, 33776, 33910, 33916
\l__coffin_gupdate_corners:N ... 33919, 33922, 33924
\l__coffin_gupdate_poles:N ..... 33920, 33953, 33955
\l__coffin_if_exist:NTF ... 33624, 33624, 33635, 33644, 33666, 33679, 33704, 33739, 33752, 33781, 33809, 33821, 33876, 33894, 34778, 34809
\l__coffin_internal_box ..... 33578, 33713, 33719, 33724, 33790, 33796, 33801, 34132, 34139, 34141, 34142, 34144
\l__coffin_internal_dim ..... 33578, 34163, 34165, 34169, 34326, 34329, 34394, 34396, 34397
\l__coffin_internal_t1 ... 33578, 34495, 34496, 34498, 34641, 34642, 34645, 34646, 34654, 34659, 34714, 34715, 34718, 34719, 34728, 34733
\l__coffin_join:NnnNnnnnN ..... 34373, 34375, 34381, 34385
\l__coffin_left_corner_dim ..... 34104, 34135, 34143, 34220, 34226, 34227, 34250, 34258
\l__coffin_mark_handle_aux:nnnnNn .. 34625, 34653, 34658, 34662
\l__coffin_offset_corner:Nnnnn .. 34504, 34507, 34509
\l__coffin_offset_corners:Nnn ... 34410, 34411, 34417, 34418, 34504, 34504
\l__coffin_offset_pole:Nnnnnnn .. 34485, 34488, 34490
\l__coffin_offset_poles:Nnn ..... 34408, 34409, 34414, 34415, 34451, 34452, 34485, 34485
\l__coffin_offset_x_dim ..... 33603, 34391, 34392, 34395, 34406, 34408, 34410, 34416, 34419, 34453, 34472, 34480, 34751, 34759
\l__coffin_offset_y_dim ..... 33603, 34409, 34411, 34416, 34419, 34453, 34474, 34481, 34753, 34760
\l__coffin_pole_a_t1 ..... 33605, 33999, 34004, 34524, 34527, 34528, 34531, 34679, 34681, 34684
\l__coffin_pole_b_t1 ..... 33605, 34000, 34004, 34525, 34527, 34529, 34531, 34680, 34681, 34683, 34684
\c__coffin_poles_prop ..... 33588, 33660, 33859, 33866
\l__coffin_poles_prop ..... 34101, 34120, 34124, 34147, 34152, 34189, 34256, 34297, 34301, 34307, 34313, 34352, 34367
\l__coffin_reset_structure:N 33638, 33854, 33854, 33912, 34399, 34443
\l__coffin_resize:NnnNN ..... 34267, 34269, 34276, 34281

```

```

\__coffin_resize_common:NnnN . . .
    ..... 34291, 34293, 34293, 34332
\l__coffin_right_corner_dim . . .
    .. 34104, 34143, 34218, 34228, 34229
\__coffin_rotate:NnNNN . . .
    ..... 34108, 34109, 34112, 34114
\__coffin_rotate_bounding:nnn . .
    ..... 34128, 34172, 34172
\__coffin_rotate_corner:Nnnn . .
    ..... 34123, 34172, 34178
\__coffin_rotate_pole:Nnnnnn . .
    ..... 34125, 34184, 34184
\__coffin_rotate_vector:nnNN . .
    ..... 34174,
    34180, 34186, 34187, 34196, 34196
\__coffin_rule:nn . . .
    ..... 34620, 34620, 34630, 34675
\__coffin_scale:NnnNN . . .
    ..... 34316, 34319, 34321
\__coffin_scale_corner:Nnnn . .
    ..... 34300, 34343, 34343
\__coffin_scale_pole:Nnnnnn . .
    ..... 34302, 34343, 34349
\__coffin_scale_vector:nnNN . .
    ..... 34336, 34336, 34345, 34351
\l__coffin_scale_x_fp 34263, 34283,
    34303, 34323, 34325, 34331, 34339
\l__coffin_scale_y_fp ... 34263,
    34285, 34324, 34325, 34329, 34341
\l__coffin_scaled_total_height_-
    dim ..... 34265, 34328, 34333
\l__coffin_scaled_width_dim . .
    ..... 34265, 34330, 34333
\__coffin_set_bounding:N . . .
    ..... 34126, 34154, 34154
\__coffin_set_horizontal_-
    pole:NnnN 33868, 33869, 33872, 33874
\__coffin_set_pole:Nnn . . .
    ..... 33714, 33791, 33868, 33904, 33909,
    34497, 34537, 34541, 34549, 34553
\__coffin_set_vertical:NnnNN . .
    ..... 33690, 33692, 33698, 33702
\__coffin_set_vertical:NnNNNNw . .
    ..... 33765, 33767, 33774, 33779
\__coffin_set_vertical_aux: . .
    ..... 33690, 33709, 33727, 33785
\__coffin_set_vertical_pole:NnnN
    ..... 33868, 33887, 33890, 33892
\__coffin_shift_corner:Nnnn . .
    ..... 34146, 34246, 34246
\__coffin_shift_pole:Nnnnnn . .
    ..... 34148, 34246, 34254
\__coffin_show:NNNnnn . . .
    ..... 34795, 34802, 34805, 34807
\__coffin_show_structure:NN . . .
    .. 34770, 34771, 34774, 34776, 34811
\l__coffin_sin_fp . . .
    1323, 1325, 34098, 34116, 34203, 34210
\l__coffin_slope_A_fp . . .
    ..... 33600
\l__coffin_slope_B_fp . . .
    ..... 33600
\__coffin_to_value:N . . .
    ..... 33611,
    33611, 33616, 33655, 33656, 33657,
    33659, 33812, 33813, 33814, 33815,
    33824, 33825, 33826, 33827, 33847,
    33856, 33858, 33863, 33865, 33878,
    33896, 33906, 33929, 33960, 34119,
    34121, 34149, 34151, 34296, 34298,
    34310, 34312, 34402, 34446, 34449,
    34487, 34506, 34513, 34678, 34789
\l__coffin_top_corner_dim . . .
    .. 34104, 34140, 34217, 34232, 34233
\__coffin_update:N . . .
    ..... 33673,
    33693, 33745, 33769, 33910, 33910
\__coffin_update_B:nnnnnnnnN . .
    ..... 34522, 34530, 34545
\__coffin_update_corners:N . . .
    ..... 33913, 33922, 33922
\__coffin_update_corners:NN . .
    ..... 33922, 33923, 33925, 33926
\__coffin_update_corners:NNN . .
    ..... 33922, 33928, 33932
\__coffin_update_poles:N . . .
    .. 33914, 33953, 33953, 34405, 34450
\__coffin_update_poles:NN . . .
    ..... 33953, 33954, 33956, 33957
\__coffin_update_poles:NNN . . .
    ..... 33953, 33959, 33963
\__coffin_update_T:nnnnnnnnN . .
    ..... 34522, 34526, 34533
\__coffin_update_vertical_-
    poles:NNN 34421, 34454, 34522, 34522
\l__coffin_x_dim . . .
    ..... 33607, 34008,
    34017, 34043, 34074, 34092, 34174,
    34176, 34180, 34182, 34186, 34191,
    34345, 34347, 34351, 34354, 34469,
    34473, 34492, 34500, 34702, 34749
\l__coffin_x_prime_dim . . .
    ..... 33607, 34188,
    34192, 34469, 34473, 34749, 34752
\__coffin_x_shift_corner:Nnnn . .
    ..... 34306, 34358, 34358
\__coffin_x_shift_pole:Nnnnnn . .
    ..... 34308, 34358, 34365
\l__coffin_y_dim . . .
    ..... 33607,
    34009, 34021, 34039, 34088, 34174,
    34176, 34180, 34182, 34186, 34191,
    34345, 34347, 34351, 34354, 34470,
    34475, 34493, 34500, 34703, 34750

```

\l\_coffin\_y\_prime\_dim .....  
     ..... [33607](#), [34188](#),  
[34193](#), [34470](#), [34475](#), [34750](#), [34754](#)  
 color commands:  
 color.sc ..... [295](#)  
 \color\_ensure\_current: .....  
     ..... [291](#), [1312](#), [33670](#),  
[33683](#), [33741](#), [33754](#), [34841](#), [34841](#)  
 \color\_export:nnN ... [296](#), [35642](#), [35642](#)  
 \color\_export:nnnN ... [296](#), [35642](#), [35652](#)  
 \color\_fill:n .... [295](#), [35495](#), [35495](#)  
 \color\_fill:nn .... [295](#), [35495](#), [35505](#)  
 \l\_color\_fixed\_model:t1 [294](#), [34975](#),  
     [34977](#), [35318](#), [35321](#), [35324](#), [35326](#),  
     [35330](#), [35365](#), [35366](#), [35372](#), [35391](#),  
     [35393](#), [35526](#), [35530](#), [35532](#), [35646](#)  
 \color\_group\_begin: .....  
     ..... [291](#), [33045](#), [33049](#),  
[33054](#), [33061](#), [33066](#), [33074](#), [33080](#),  
[33094](#), [33100](#), [33107](#), [33112](#), [33125](#),  
[33127](#), [33131](#), [33136](#), [33141](#), [33146](#),  
[33153](#), [33158](#), [33165](#), [33170](#), [33178](#),  
[33184](#), [33199](#), [33205](#), [34839](#), [34839](#)  
 \color\_group\_end: .... [291](#), [33045](#),  
     [33049](#), [33054](#), [33061](#), [33066](#), [33086](#),  
     [33107](#), [33112](#), [33125](#), [33127](#), [33131](#),  
     [33136](#), [33141](#), [33146](#), [33153](#), [33158](#),  
     [33165](#), [33170](#), [33191](#), [34839](#), [34840](#)  
 \color\_if\_exist:n ..... [294](#), [34858](#)  
 \color\_if\_exist:nTF ... [294](#), [34858](#),  
     [34968](#), [35001](#), [35049](#), [35611](#), [36404](#)  
 \color\_if\_exist\_p:n ..... [294](#), [34858](#)  
 \color\_log:n ..... [294](#), [36395](#), [36397](#)  
 \color\_math:nn .... [295](#), [35398](#), [35398](#)  
 \color\_math:nn(n) ..... [1357](#)  
 \color\_math:nnn ... [295](#), [35398](#), [35403](#)  
 \l\_color\_math\_active:t1 .....  
     ..... [295](#), [35395](#), [35456](#)  
 \color\_model\_new:nnn [297](#), [35764](#), [35764](#)  
 \color\_profile\_apply:nn .....  
     ..... [298](#), [36366](#), [36366](#)  
 \color\_select:n .... [294](#), [34629](#),  
     [34636](#), [34674](#), [34709](#), [35347](#), [35347](#)  
 \color\_select:n(n) ..... [295](#)  
 \color\_select:nn ... [294](#), [35347](#), [35353](#)  
 \color\_select:nn(n) ..... [295](#)  
 \color\_set:nn .... [294](#), [35523](#), [35523](#)  
 \color\_set:nnn ..... [294](#),  
     [35523](#), [35568](#), [35631](#), [35632](#), [35633](#),  
     [35634](#), [35635](#), [35636](#), [35637](#), [35638](#)  
 \color\_set\_eq:nn ... [294](#), [35523](#), [35609](#)  
 \color\_show:n .... [294](#), [36395](#), [36395](#)  
 \color\_stroke:n ... [295](#), [35495](#), [35500](#)  
 \color\_stroke:nn ... [295](#), [35495](#), [35511](#)

color internal commands:

\g\_color\_alternative\_model\_prop  
     ..... [35751](#), [35863](#), [35961](#)  
 \g\_color\_alternative\_values\_-  
     prop ..... [35756](#),  
[35878](#), [35892](#), [35902](#), [36116](#), [36218](#)  
 \\_\_color\_backend\_devicen\_-  
     init:nnm ..... [36131](#)  
 \\_\_color\_backend\_iccbased\_-  
     device:nnn ... [36383](#), [36388](#), [36393](#)  
 \\_\_color\_backend\_iccbased\_-  
     init:nnm ..... [36363](#)  
 \\_\_color\_backend\_reset: [34847](#), [35476](#)  
 \\_\_color\_backend\_separation\_-  
     init:nnnnn ... [35879](#), [35893](#), [35903](#)  
 \\_\_color\_backend\_separation\_-  
     init\_CIELAB:nnn ..... [35931](#)  
 \\_\_color\_check\_model:N .....  
     ..... [34971](#), [35319](#), [35319](#)  
 \\_\_color\_check\_model:nn .....  
     ..... [35319](#), [35323](#), [35333](#)  
 \g\_color\_colorants\_prop .....  
     ..... [35734](#), [35864](#), [36179](#)  
 \\_\_color\_convert:nnN [34876](#), [34876](#),  
     [34878](#), [34990](#), [35074](#), [35085](#), [35326](#)  
 \\_\_color\_convert:nnnN ... [34876](#),  
     [34877](#), [34880](#), [34912](#), [35391](#), [35676](#)  
 \\_\_color\_convert\_cmyk\_cmyk:w . [34950](#)  
 \\_\_color\_convert\_cmyk\_gray:w . [34942](#)  
 \\_\_color\_convert\_cmyk\_rgb:w .. [34944](#)  
 \\_\_color\_convert\_devicen\_-  
     cmyk:nnnnnnnnn [35943](#), [36239](#), [36242](#)  
 \\_\_color\_convert\_devicen\_-  
     cmyk:nnnnw ... [35943](#), [36236](#), [36264](#)  
 \\_\_color\_convert\_devicen\_cmyk\_-  
     aux:nnnnw ... [35943](#), [36246](#), [36253](#)  
 \\_\_color\_convert\_devicen\_-  
     gray:nnn ... [35943](#), [36271](#), [36274](#)  
 \\_\_color\_convert\_devicen\_gray:nw  
     ..... [35943](#), [36268](#), [36285](#)  
 \\_\_color\_convert\_devicen\_gray\_-  
     aux:nw ... [35943](#), [36276](#), [36279](#)  
 \\_\_color\_convert\_devicen\_-  
     rgb:nnnnnnn ... [35943](#), [36292](#), [36295](#)  
 \\_\_color\_convert\_devicen\_-  
     rgb:nnnw ... [35943](#), [36289](#), [36315](#)  
 \\_\_color\_convert\_devicen\_rgb\_-  
     aux:nnnw ... [35943](#), [36299](#), [36305](#)  
 \\_\_color\_convert\_gray\_cmyk:w . [34917](#)  
 \\_\_color\_convert\_gray\_gray:w . [34913](#)  
 \\_\_color\_convert\_gray\_gray:w\uuuuu\u\_-  
     \_color\_convert\_gray\_rgb:w\uuuuu\u\_-  
     \_color\_convert\_gray\_cmyk:w\uuuuu\u\_-  
     \_color\_convert\_cmyk\_gray:w\uuuuu\u\_-

```

    _color_convert_cmyk_rgb:wuuuuuu\_-
    _color_convert_cmyk_cmyk:wuuuuuu\_-
    _color_convert_rgb_gray:wuuuuuu\_-
    _color_convert_rgb_rgb:wuuuuuu\_-
    _color_convert_rgb_cmyk:w . 34876
\__color_convert_gray_rgb:w . 34915
\__color_convert_rgb_cmyk:nnn ...
... 1346, 34876, 34925, 34930, 35286
\__color_convert_rgb_cmyk:nnn ...
... 34876, 34932, 34935
\__color_convert_rgb_cmyk:w . 34923
\__color_convert_rgb_gray:w . 34919
\__color_convert_rgb_rgb:w . 34921
\l__color_current_tl .... 1342,
  34839, 34842, 34853, 34959, 34962,
  35009, 35341, 35345, 35349, 35351,
  35355, 35358, 35401, 35407, 35413,
  35415, 35477, 35484, 35485, 35497,
  35498, 35502, 35503, 35507, 35509,
  35513, 35515, 35618, 35620, 35641
\__color_draw:nnn .... 35495,
  35498, 35503, 35509, 35515, 35517
\__color_export:nN .....
... 35642, 35648, 35656, 35658
\__color_export:nnnN .....
... 35642, 35659, 35660
\__color_export:nnnNN .....
... 35671, 35671, 35691
\__color_export_comma_sep_cmyk:Nw
... 35702
\c__color_export_comma_sep_cmyk_-
  tl .... 35681
\__color_export_comma_sep_rgb:Nw
... 35707
\c__color_export_comma_sep_rgb_-
  tl .... 35681
\__color_export_format_backend:nnN
... 35669, 35669
\__color_export_format_comma_sep_cmyk:nnN
... 35686
\__color_export_format_comma_sep_rgb:nnN
... 35686
\__color_export_format_space_sep_cmyk:nnN
... 35686
\__color_export_format_space_sep_rgb:nnN
... 35686
\__color_export_HTML:n .....
... 35707, 35713, 35714, 35715, 35718
\__color_export_HTML:Nw 35707, 35709
\c__color_export_HTML_tl .... 35681
\__color_export_space_sep_cmyk:Nw
... 35702
\c__color_export_space_sep_cmyk_-
  tl .... 35681
\__color_export_space_sep-rgb:Nw
... 35707
\c__color_export_space_sep-rgb_-
  tl .... 35681
\__color_extract:nNN .....
... 34870, 34870, 34875,
  35015, 35031, 35032, 35051, 35066
\c__color_fallback_cmyk_t1 ... 35731
\c__color_fallback_gray_t1 ... 35731
\c__color_fallback_rgb_t1 ... 35731
\__color_finalise_current: .....
... 35338, 35338, 35350, 35357
\c__color_icc_colorspace_-
  signatures_prop .... 36319, 36345
\l__color_internal_int .....
... 34855, 36112, 36115, 36171
\l__color_internal_prop .....
... 35729, 35779, 35810,
  35823, 35838, 35917, 35945, 36332
\l__color_internal_t1 .....
... 34855, 35017, 35020,
  35552, 35558, 35560, 35562, 35563,
  35601, 35603, 35604, 35811, 35814,
  35824, 35827, 35839, 35841, 35918,
  35922, 35925, 35946, 35949, 35962,
  35965, 35967, 36110, 36121, 36144,
  36167, 36333, 36336, 36346, 36349
\__color_math:nn .....
... 35398, 35400, 35405, 35411
\__color_math_scan:w 1359, 35417,
  35419, 35419, 35450, 35471, 35487
\__color_math_scan_auxi: .....
... 35419, 35431, 35435
\__color_math_scan_auxii: .....
... 35419, 35451, 35454
\__color_math_scan_auxiii:N .....
... 35463, 35469
\__color_math_scan_end: .....
... 35419, 35427, 35466, 35474
\__color_math_script_aux:N .....
... 35479, 35491, 35494
\__color_math_scripts:Nw .....
... 35442, 35479, 35479
\g__color_math_seq .....
... 35397, 35413, 35477, 35484
\__color_model:N .....
... 34868, 34868, 34959, 35341,
  35541, 35562, 35601, 35618, 35641
\__color_model_convert:nnn .....
... 35808, 35905
\__color_model_devicen:n 35943, 35943
\__color_model_devicen:nn .....
... 35943, 35948, 35956

```

\\_\_color\_model\_devicen:nnn . . . . .  
..... 35943, 35990, 35992  
\\_\_color\_model\_devicen:nnnn . . . . .  
..... 35943, 35994, 35997  
\\_\_color\_model\_devicen\_colorant:n  
..... 35943, 36134, 36177  
\\_\_color\_model\_devicen\_convert:n  
..... 35943, 36209, 36214  
\\_\_color\_model\_devicen\_convert:nnn  
..... 35943  
\\_\_color\_model\_devicen\_convert:nnnn  
..... 35943, 36007, 36181, 36185  
\\_\_color\_model\_devicen\_convert:nnnnn  
..... 36189, 36194, 36199, 36201  
\\_\_color\_model\_devicen\_convert:w  
..... 35943  
\\_\_color\_model\_devicen\_convert\_-  
aux:n ..... 35943, 36217, 36221  
\\_\_color\_model\_devicen\_convert\_-  
aux:w ..... 36222, 36223  
\\_\_color\_model\_devicen\_convert\_-  
cmyk:n ..... 35943  
\\_\_color\_model\_devicen\_convert\_-  
cmyk:nnn ..... 36186  
\\_\_color\_model\_devicen\_convert\_-  
gray:n ..... 35943  
\\_\_color\_model\_devicen\_convert\_-  
gray:nnn ..... 36191  
\\_\_color\_model\_devicen\_convert\_-  
rgb:n ..... 35943  
\\_\_color\_model\_devicen\_convert\_-  
rgb:nnn ..... 36196  
\\_\_color\_model\_devicen\_init:nnn .  
..... 35943, 36006, 36095  
\\_\_color\_model\_devicen\_init:nnnn  
..... 35943, 36097, 36108  
\\_\_color\_model\_devicen\_mix:nw .  
..... 35943, 36067, 36086, 36092  
\\_\_color\_model\_devicen\_parse:nw .  
..... 35943, 36062, 36072, 36081  
\\_\_color\_model\_devicen\_parse\_-  
1:nn ..... 35943  
\\_\_color\_model\_devicen\_parse\_-  
2:nn ..... 35943  
\\_\_color\_model\_devicen\_parse\_-  
3:nn ..... 35943  
\\_\_color\_model\_devicen\_parse\_-  
4:nn ..... 35943  
\\_\_color\_model\_devicen\_parse\_-  
generic:nn ... 35943, 36004, 36057  
\\_\_color\_model\_devicen\_transform:nnn  
..... 35943  
\\_\_color\_model\_devicen\_transform:w  
..... 35943  
\\_\_color\_model\_devicen\_transform\_-  
1:nnnnn ..... 35943  
\\_\_color\_model\_devicen\_transform\_-  
3:nnnnn ..... 35943  
\\_\_color\_model\_devicen\_transform\_-  
4:nnnnn ..... 35943  
\\_\_color\_model\_devicen\_transform:nnn  
..... 36154, 36158, 36163, 36165  
\\_\_color\_model\_devicen\_transform:w  
..... 36118, 36147  
\\_\_color\_model\_iccbased:n . . . . .  
..... 36330, 36330  
\\_\_color\_model\_iccbased:nn . . . . .  
..... 36330, 36335, 36343  
\\_\_color\_model\_iccbased:nnn .. 36330  
\\_\_color\_model\_iccbased\_aux:nnn .  
..... 36330  
\\_\_color\_model\_iccbased\_aux:nnnnn  
..... 36348, 36356  
\\_\_color\_model\_init:nnn .. 35787,  
35787, 35807, 35857, 35999, 36358  
\g\_color\_model\_int ..... 35730,  
35789, 35795, 36382, 36387, 36392  
\\_\_color\_model\_new:nnn . . . . .  
..... 35764, 35766, 35770  
\c\_color\_model\_range\_CIELAB\_t1 .  
..... 35750  
\\_\_color\_model\_separation:n . . . . .  
..... 35808, 35808  
\\_\_color\_model\_separation:nn . . . . .  
..... 35808, 35813, 35821  
\\_\_color\_model\_separation:nnn . . . . .  
..... 35808, 35826, 35834  
\\_\_color\_model\_separation:w . . . . .  
..... 35808, 35841, 35854  
\\_\_color\_model\_separation\_-  
CIELAB:nnnnnn ..... 35808, 35915  
\\_\_color\_model\_separation\_-  
CIELAB:nnnnnnn 35808, 35924, 35927  
\\_\_color\_model\_separation\_-  
cmyk:nnnnnn ..... 35808, 35867  
\\_\_color\_model\_separation\_-  
gray:nnnnnn ..... 35808, 35896  
\\_\_color\_model\_separation\_-  
rgb:nnnnnn ..... 35808, 35882  
\l\_color\_model\_t1 . . . . .  
34952, 34987, 34988, 34991, 35015,  
35018, 35033, 35052, 35054, 35061,  
35066, 35072, 35074, 35076, 35082,  
35087, 35324, 35326, 35335, 35958,  
35964, 35965, 35967, 35985, 35990  
\c\_color\_model\_whitepoint\_-  
CIELAB\_a\_t1 ..... 35743

```

\c__color_model_whitepoint_-
    CIELAB_b_t1 ..... 35743
\c__color_model_whitepoint_-
    CIELAB_d50_t1 ..... 35743
\c__color_model_whitepoint_-
    CIELAB_d55_t1 ..... 35743
\c__color_model_whitepoint_-
    CIELAB_d65_t1 ..... 35743
\c__color_model_whitepoint_-
    CIELAB_d75_t1 ..... 35743
\c__color_model_whitepoint_-
    CIELAB_e_t1 ..... 35743
\l__color_named_.prop ..... 35639
\l__color_named_.tl ..... 35639
\l__color_named_tl . 35522, 35538,
    35541, 35544, 35600, 35601, 35605
\l__color_named_white_prop ... 35800
\l__color_next_model_t1 .. 34952,
    35051, 35052, 35072, 35073, 35086
\l__color_next_value_t1 .. 34952,
    35051, 35061, 35077, 35083, 35088
\__color_parse:nN .....
    ... 34956, 34956, 35349, 35401,
        35497, 35502, 35538, 35558, 35647
\__color_parse:Nw 34956, 34970, 34999
\__color_parse_aux:nN .....
    ... 34956, 34963, 34966
\__color_parse_break:w .....
    ... 34956, 35067, 35091
\__color_parse_end: .....
    ... 34956, 35045, 35091, 35092
\__color_parse_eq:Nn ..... 34956
\__color_parse_eq:nNn ..... 34956
\__color_parse_gray:n .....
    ... 34956, 35055, 35070
\__color_parse_loop:nn .....
    ... 34956, 35038, 35047
\__color_parse_loop:w .....
    ... 34956, 35016, 35022, 35044
\__color_parse_loop_init:Nnn ...
    ... 34956, 35005, 35012
\__color_parse_mix:Nnnm .....
    ... 34956, 35060, 35093, 35099
\__color_parse_mix:nNnn .....
    ... 34956, 35095, 35100
\__color_parse_mix_cmyk:nw .....
    ... 34956, 35114, 36055
\__color_parse_mix_gray:nw .....
    ... 34956, 35105, 35858, 36016
\__color_parse_mix_rgb:nw .....
    ... 34956, 35107, 36040
\__color_parse_model_&spot:w . 35316
\__color_parse_model_cmy:w .....
    ... 35283, 35283
\__color_parse_model_cmyk:w .....
    ... 35122, 35133
\__color_parse_model_Gray:w .....
    ... 35147, 35147
\__color_parse_model_gray:w .....
    ... 35122, 35122
\__color_parse_model_hsb:nnn .....
    ... 35147,
        35150, 35153, 35156, 35193, 35290
\__color_parse_model_hsb:nnnn . 35147
\__color_parse_model_hsb:nnnnn 35147
\__color_parse_model_HSB:w .....
    ... 35147, 35191
\__color_parse_model_Hsb:w .....
    ... 35147, 35151
\__color_parse_model_hsb:w .....
    ... 35147, 35149
\__color_parse_model_hsb_0:nnnn .....
    ... 35147
\__color_parse_model_hsb_1:nnnn .....
    ... 35147
\__color_parse_model_hsb_2:nnnn .....
    ... 35147
\__color_parse_model_hsb_3:nnnn .....
    ... 35147
\__color_parse_model_hsb_4:nnnn .....
    ... 35147
\__color_parse_model_hsb_5:nnnn .....
    ... 35147
\__color_parse_model_hsb_aux:nnn .....
    ... 35147, 35160, 35164, 35276
\__color_parse_model_hsb_-
    aux:nnnn ..... 35166, 35170
\__color_parse_model_hsb_-
    aux:nnnnn ..... 35174, 35182
\__color_parse_model_HTML:w .....
    ... 35147, 35198
\__color_parse_model_HTML_aux:w .
    ... 35199, 35200
\__color_parse_model_RGB:w .....
    ... 35147, 35209
\__color_parse_model_rgb:w .....
    ... 35122, 35124
\__color_parse_model_tHsb:n .....
    ... 35288, 35291, 35293
\__color_parse_model_tHsb:nw .....
    ... 35288, 35295, 35306, 35310
\__color_parse_model_tHsb:w .....
    ... 35288, 35288
\__color_parse_model_wave:w .....
    ... 35147, 35218
\__color_parse_model_wave_-
    auxi:nn ..... 35147, 35223, 35227, 35228, 35232

```

```

\__color_parse_model_wave_-
    auxii:nn .. 35147, 35236, 35243,
    35250, 35257, 35264, 35268, 35274
\__color_parse_model_wave_rho:n .
    ..... 35147, 35237, 35244,
    35251, 35258, 35265, 35279, 35281
\__color_parse_number:n .....
    ..... 35122, 35123,
    35128, 35129, 35130, 35137, 35138,
    35139, 35140, 35143, 35175, 35860,
    36015, 36021, 36035, 36036, 36037,
    36049, 36050, 36051, 36052, 36079
\__color_parse_number:w .....
    ..... 35122, 35144, 35145
\__color_parse_set_eq:Nn .....
    ..... 34969, 34973, 35004
\__color_parse_set_eq:nNn .....
    ..... 34976, 34977, 34980
\__color_parse_std:n .....
    ..... 34956, 35056, 35079
\__color_profile_apply:nn .....
    ..... 36366, 36368, 36371
\__color_profile_apply_cmyk:n ...
    ..... 36366, 36390
\__color_profile_apply_gray:n ...
    ..... 36366, 36380
\__color_profile_apply_rgb:n ...
    ..... 36366, 36385
\__color_select:N .....
    34842,
    34844, 34844, 35351, 35358, 35485
\__color_select:nn .....
    ..... 34844, 34846, 34850, 34851
\__color_select:nnN .....
    .. 35347, 35363, 35373, 35380, 35600
\__color_select_loop:Nw .....
    ..... 35347, 35367, 35369, 35377
\__color_select_main:Nw .....
    ..... 35347, 35355,
    35360, 35407, 35507, 35513, 35654
\__color_select_math:N .....
    ..... 34844, 34849, 35415
\__color_select_swap:Nnn .....
    ..... 35347, 35376, 35389
\__color_set:nn . 35523, 35531, 35534
\__color_set:nnn . 35523, 35525, 35528
\__color_set:nnw . 35523, 35545, 35548
\__color_set_aux:nnn .....
    ..... 35523, 35574, 35578
\__color_set_colon:nnw .....
    ..... 35523, 35580, 35585
\__color_set_loop:nw .....
    .. 35523, 35591, 35592, 35595, 35606
\__color_show:n . 36395, 36406, 36415
\__color_show:Nn .....
    ..... 36395, 36396, 36398, 36399
\__color_tmp:w .....
    35687,
    35695, 35696, 35697, 35698, 35699
\l__color_value_tl .....
    34952, 34984, 34985, 34989, 34991,
    34995, 35015, 35018, 35033, 35058,
    35061, 35066, 35075, 35327, 35330,
    35336, 35391, 35393, 36117, 36119
\__color_values:N .....
    ..... 34868, 34869, 34962,
    35345, 35544, 35563, 35605, 35620
\columnwidth .....
    ..... 33734
\contextversion 10184, 10214, 10415, 10435
\copy ..... 219
\copyfont ..... 913
\cos ..... 255
\cosd ..... 255
\cot ..... 255
\cotd ..... 255
\count ..... 220, 19300
\countdef ..... 221
\cr ..... 222
\crampeddisplaystyle ..... 800
\crampedscriptscriptstyle ..... 801
\crampedscriptstyle ..... 803
\crampedtextstyle ..... 804
\crcr ..... 223
\creationdate ..... 773
\cs ..... 25024
cs commands:
    \cs:w 21, 539, 563, 644, 1444, 1446,
        1466, 1468, 1521, 1839, 1867, 2060,
        2125, 2295, 2344, 2353, 2355, 2359,
        2360, 2361, 2423, 2429, 2435, 2441,
        2468, 2470, 2475, 2482, 2483, 2548,
        2552, 2591, 3209, 4443, 7299, 7302,
        8725, 8727, 10918, 12053, 14098,
        14104, 17279, 17366, 18003, 18055,
        18978, 20091, 20387, 20434, 20525,
        21109, 21110, 21770, 22706, 22725,
        22792, 23603, 23792, 23824, 24238,
        24264, 24277, 24311, 24353, 24869,
        26574, 27665, 32528, 32903, 36848
    \cs_argument_spec:N .....
        ..... 37613, 37631, 37632
\cs_end: ..... 21, 413, 563, 645,
    1444, 1447, 1466, 1468, 1472, 1521,
    1833, 1839, 1861, 1867, 1987, 2060,
    2125, 2295, 2344, 2353, 2355, 2359,
    2360, 2361, 2423, 2429, 2435, 2441,
    2468, 2470, 2475, 2482, 2483, 2548,
    2552, 2591, 3209, 4443, 7108, 7315,
    8722, 8728, 8730, 8732, 8734, 8736,
```

8738, 8740, 8742, 8744, 8746, 8748,  
10918, 10933, 10936, 10937, 12053,  
14104, 14107, 17279, 17366, 18003,  
18008, 18036, 18045, 18055, 18978,  
20091, 20387, 20434, 20525, 21109,  
21110, 21770, 22709, 22725, 22800,  
23606, 23796, 23828, 24244, 24270,  
24283, 24314, 24356, 24875, 26574,  
27668, 32528, 32903, 36846, 36848  
\cs\_generate\_from\_arg\_count:Nn  
..... 19, 2040,  
2040, 2050, 2051, 2052, 2053, 2083  
\cs\_generate\_variant:Nn ..  
..... 15, 31–33, 64, 306, 407,  
2908, 2908, 2921, 2922, 3251, 3253,  
3255, 3257, 3434, 3436, 3458, 3461,  
3467, 3473, 4195, 5215, 6368, 7123,  
7164, 8474, 8480, 8489, 8490, 8491,  
8492, 8495, 8496, 8507, 8508, 8524,  
8531, 8533, 8682, 8683, 8688, 8689,  
9027, 9059, 9768, 10154, 10155,  
10156, 10157, 10195, 10200, 10234,  
10255, 10257, 10259, 10430, 10458,  
10466, 10478, 10480, 10482, 10509,  
10512, 10530, 10638, 11126, 11135,  
11136, 11137, 11418, 11502, 11722,  
12087, 12098, 12099, 12104, 12105,  
12110, 12111, 12116, 12117, 12134,  
12135, 12156, 12157, 12158, 12159,  
12160, 12161, 12162, 12163, 12204,  
12205, 12206, 12207, 12208, 12209,  
12210, 12211, 12246, 12247, 12248,  
12249, 12250, 12251, 12252, 12253,  
12301, 12302, 12303, 12304, 12368,  
12369, 12370, 12371, 12432, 12433,  
12438, 12439, 12609, 12623, 12653,  
12663, 12684, 12689, 12691, 12700,  
12712, 12713, 12750, 12753, 12758,  
12759, 12835, 12846, 13044, 13055,  
13056, 13079, 13086, 13088, 13165,  
13186, 13188, 13269, 13284, 13285,  
13288, 13289, 13300, 13322, 13323,  
13324, 13325, 13358, 13359, 13364,  
13365, 13444, 13506, 13524, 13550,  
13564, 13611, 13672, 13750, 13769,  
13807, 13822, 13839, 13840, 13841,  
13854, 13939, 13981, 13988, 16210,  
16211, 16281, 16288, 16312, 16486,  
16489, 16492, 16495, 16498, 16527,  
16528, 16529, 16530, 16531, 16532,  
16538, 16578, 16579, 16580, 16581,  
16586, 16587, 16609, 16610, 16611,  
16612, 16617, 16618, 16619, 16620,  
16637, 16638, 16663, 16664, 16681,  
16682, 16738, 16739, 16789, 16802,  
16803, 16821, 16847, 16848, 16900,  
16906, 16934, 16963, 16973, 16996,  
16997, 17074, 17097, 17111, 17145,  
17147, 17281, 17304, 17319, 17320,  
17325, 17326, 17328, 17330, 17343,  
17344, 17345, 17346, 17355, 17356,  
17357, 17358, 17363, 17364, 17972,  
17976, 18118, 18158, 18159, 18160,  
18161, 18175, 18176, 18185, 18186,  
18196, 18197, 18198, 18199, 18209,  
18210, 18211, 18212, 18223, 18248,  
18249, 18313, 18314, 18352, 18353,  
18358, 18359, 18450, 18487, 18490,  
18522, 18551, 18595, 18604, 18661,  
18668, 18709, 18711, 18713, 18866,  
18867, 18868, 18869, 19691, 19697,  
19700, 19703, 19706, 19709, 19725,  
19728, 19740, 19746, 19753, 19761,  
19769, 19801, 19802, 19803, 19804,  
19811, 19812, 19831, 19832, 19833,  
19834, 19845, 19855, 19927, 19929,  
19931, 19933, 19950, 19951, 20011,  
20026, 20049, 20055, 20057, 20093,  
20099, 20103, 20104, 20109, 20110,  
20119, 20120, 20123, 20126, 20134,  
20135, 20143, 20144, 20416, 20436,  
20442, 20445, 20446, 20451, 20452,  
20461, 20462, 20464, 20466, 20471,  
20472, 20477, 20478, 20506, 20507,  
20509, 20527, 20533, 20538, 20539,  
20544, 20545, 20554, 20555, 20557,  
20559, 20564, 20565, 20570, 20571,  
20577, 20876, 20959, 20962, 20978,  
21033, 21045, 21139, 21252, 21258,  
21461, 21475, 21481, 21491, 21517,  
21523, 21533, 21573, 21619, 22050,  
22102, 22135, 22146, 22184, 22187,  
22197, 22199, 22281, 22283, 22313,  
22355, 22364, 22373, 22382, 22390,  
22443, 22445, 22459, 22480, 22523,  
23049, 23052, 24744, 24751, 24752,  
24753, 24756, 24757, 24760, 24761,  
24766, 24767, 24774, 24775, 24776,  
24777, 24779, 24781, 25019, 25081,  
28163, 28217, 28295, 28340, 28355,  
28409, 28748, 28768, 28797, 28851,  
28859, 28867, 28953, 28992, 29005,  
29064, 29137, 29188, 29463, 29863,  
30227, 30603, 30665, 31023, 32470,  
32905, 32910, 32911, 32916, 32917,  
32922, 32923, 32928, 32929, 32937,  
32938, 32939, 32942, 32948, 32951,  
32957, 32960, 32966, 32969, 32972,

32973, 33001, 33002, 33010, 33013,  
 33016, 33025, 33043, 33056, 33057,  
 33068, 33069, 33082, 33083, 33102,  
 33103, 33122, 33123, 33148, 33149,  
 33160, 33161, 33172, 33173, 33186,  
 33187, 33207, 33208, 33211, 33212,  
 33215, 33221, 33236, 33239, 33357,  
 33363, 33403, 33406, 33423, 33426,  
 33443, 33446, 33460, 33463, 33481,  
 33484, 33510, 33513, 33519, 33525,  
 33641, 33650, 33663, 33676, 33689,  
 33695, 33701, 33749, 33762, 33771,  
 33778, 33818, 33830, 33870, 33873,  
 33888, 33891, 33909, 34110, 34113,  
 34273, 34280, 34317, 34320, 34378,  
 34384, 34429, 34435, 34566, 34669,  
 34745, 34772, 34775, 34797, 34800,  
 34803, 34806, 34875, 34878, 34879,  
 34912, 35099, 35807, 36185, 36602,  
 36609, 36730, 36736, 36739, 36742,  
 36745, 36793, 36796, 36878, 36881,  
 36930, 36960, 36990, 36991, 37064,  
 37065, 37228, 37231, 37276, 37280  
 $\backslash\text{cs\_gset:Nn}$  . . . . . 18, 2055, 2120  
 $\cdot\text{cs\_gset:Np}$  . . . . . 225, 21295  
 $\backslash\text{cs\_gset:Npn}$  . . . . . 14, 16, 1505,  
 1507, 1595, 1935, 1949, 1951, 7140,  
 9298, 9300, 9338, 11108, 16362,  
 16363, 16400, 16938, 18916, 19026,  
 21304, 21306, 23483, 30043, 30110,  
 31028, 37424, 37632, 37635, 37637,  
 37639, 37641, 37643, 37645, 37647,  
 37649, 37656, 37662, 37665, 37668,  
 37671, 37674, 37677, 37680, 37682,  
 37684, 37686, 37688, 37690, 37692,  
 37694, 37696, 37698, 37700, 37702,  
 37704, 37706, 37708, 37710, 37712  
 $\backslash\text{cs\_gset:Npx}$  . . . . . 16,  
 1505, 1509, 1936, 1949, 1952, 16943  
 $\backslash\text{cs\_gset_eq:NN}$  . . . . . 19, 1967,  
 1971, 1972, 1973, 1974, 1984, 1992,  
 8486, 8488, 9187, 10252, 10475,  
 12064, 12066, 12085, 14255, 14259,  
 16484, 16709, 16948, 16953, 18864,  
 19695, 20014, 20022, 29653, 29663  
 $\backslash\text{cs\_gset\_nopar:Nn}$  . . . . . 18, 2055, 2120  
 $\backslash\text{cs\_gset\_nopar:Npn}$  . . . . . 16, 1505,  
 1505, 1933, 1941, 1945, 16170, 16455  
 $\backslash\text{cs\_gset\_nopar:Npx}$  . . . . .  
 . . . . . 16, 676, 1402, 1505, 1506, 1934,  
 1941, 1946, 12081, 12091, 12096,  
 22038, 37172, 37198, 37203, 37230  
 $\backslash\text{cs\_gset\_protected:Nn}$  . . . . . 18, 2055, 2120  
 $\cdot\text{cs\_gset\_protected:Np}$  . . . . . 225, 21295  
 $\backslash\text{cs\_gset\_protected:Npn}$  . . . . . 16,  
 1505, 1515, 1601, 1622, 1939, 1961,  
 1963, 3385, 3393, 3406, 3816, 4089,  
 10217, 10366, 10438, 12641, 13510,  
 17006, 17590, 18471, 20017, 20345,  
 21308, 21310, 25086, 29169, 32265,  
 37369, 37386, 37395, 37419, 37651,  
 37654, 37659, 37723, 37729, 37735  
 $\backslash\text{cs\_gset\_protected:Npx}$  . . . . .  
 . . . . . 16, 1505, 1517, 1940,  
 1961, 1964, 17601, 20352, 25093, 37375  
 $\backslash\text{cs\_gset\_protected\_nopar:Nn}$  . . . . .  
 . . . . . 19, 2055, 2120  
 $\backslash\text{cs\_gset\_protected\_nopar:Npn}$  . . . . .  
 . . . . . 17, 1505, 1511, 1937, 1955, 1957  
 $\backslash\text{cs\_gset\_protected\_nopar:Npx}$  . . . . .  
 . . . . . 17, 1505, 1513, 1938, 1955, 1958  
 $\backslash\text{cs\_if\_eq:NN}$  . . . . . 2152  
 $\backslash\text{cs\_if\_eq:NNTF}$  . . . . . 27, 1409,  
 2152, 2158, 2159, 2160, 2162, 2163,  
 2164, 2166, 2167, 2168, 5843, 9233,  
 9348, 21162, 23248, 23258, 23284,  
 23286, 23288, 23488, 30398, 30447,  
 30467, 30866, 32404, 37369, 37395  
 $\backslash\text{cs\_if\_eq\_p:NN}$  . . . . .  
 . . . . . 27, 2152, 2157, 2161, 2165, 5860,  
 29880, 30536, 30537, 32451, 32452  
 $\backslash\text{cs\_if\_exist}$  . . . . . 60  
 $\backslash\text{cs\_if\_exist:N}$  . . . . . 1819,  
 1831, 8553, 8555, 12136, 12137,  
 16588, 16590, 17331, 17333, 18177,  
 18179, 19952, 19954, 20111, 20113,  
 20453, 20455, 20546, 20548, 24820,  
 24821, 29138, 29140, 32930, 32932  
 $\backslash\text{cs\_if\_exist:NTF}$  . . . . . 21, 27, 349, 404,  
 608, 740, 879, 1819, 1876, 1878,  
 1880, 1882, 1884, 1886, 1888, 1890,  
 2172, 2298, 2368, 2404, 2501, 2525,  
 2593, 2624, 2875, 3141, 3383, 3401,  
 3404, 5219, 5571, 7285, 8803, 8804,  
 8805, 8807, 8811, 8840, 9078, 9148,  
 9231, 9269, 9857, 10070, 10184,  
 10188, 10214, 10415, 10419, 10435,  
 10886, 11047, 11106, 11138, 11147,  
 11290, 11366, 11520, 11573, 11698,  
 12800, 13877, 13878, 14226, 14235,  
 14239, 14253, 16693, 17306, 17307,  
 17308, 17309, 17984, 17985, 18031,  
 18913, 19264, 19283, 19856, 20601,  
 20759, 20890, 20985, 21081, 21086,  
 21107, 21648, 21690, 21698, 21710,  
 21722, 21728, 21739, 21755, 21842,  
 21903, 21912, 22022, 22323, 23481,  
 23655, 24734, 29014, 29114, 29167,

29202, 29797, 30515, 30651, 30855,  
30901, 30909, 31897, 31904, 32608,  
33614, 33616, 34887, 35382, 35772,  
35777, 35836, 36681, 37105, 37124  
\cs\_if\_exist\_p:N 27, 349, 1819, 8871,  
9164, 29090, 30483, 30545, 32416,  
33730, 34614, 36687, 36688, 36689  
\cs\_if\_exist\_use:N . . . . . 21, 374,  
1875, 1881, 1889, 6232, 9648, 9666,  
10607, 21724, 21743, 30658, 30734  
\cs\_if\_exist\_use:NTF . . . . .  
. . . . . 21, 1875, 1875, 1877, 1879,  
1883, 1885, 1887, 3171, 3240, 4784,  
4791, 5178, 5183, 5227, 5639, 5725,  
7221, 9202, 16315, 22930, 23613,  
23615, 30921, 30927, 30994, 30996,  
34884, 34898, 35662, 36003, 36373  
\cs\_if\_free:N . . . . . 1847, 1859  
\cs\_if\_free:NTF . . . . . 27,  
62, 608, 1847, 1916, 3083, 3110, 9638  
\cs\_if\_free\_p:N . . . . . 26, 27, 62, 1847  
\cs\_log:N 20, 383, 2197, 2200, 2201, 2202  
\cs\_meaning:N . . . . .  
. . . . . 20, 360, 1453, 1454, 1469,  
1470, 1477, 1480, 2209, 9075, 29179  
\cs\_new:Nn . . . . . 17, 63, 2055, 2120  
\cs\_new:Npn . . . . . 14,  
15, 19, 62, 63, 411, 426, 1409, 1597,  
1618, 1925, 1935, 1949, 1953, 2026,  
2028, 2030, 2038, 2091, 2157, 2158,  
2159, 2160, 2161, 2162, 2163, 2164,  
2165, 2166, 2167, 2168, 2234, 2238,  
2247, 2256, 2265, 2268, 2277, 2278,  
2288, 2289, 2290, 2291, 2292, 2293,  
2294, 2296, 2300, 2304, 2307, 2320,  
2326, 2332, 2343, 2345, 2352, 2354,  
2356, 2363, 2364, 2366, 2370, 2374,  
2380, 2382, 2387, 2392, 2398, 2406,  
2413, 2414, 2420, 2426, 2432, 2438,  
2444, 2451, 2458, 2465, 2472, 2479,  
2488, 2489, 2491, 2496, 2503, 2507,  
2510, 2520, 2521, 2523, 2527, 2530,  
2531, 2533, 2535, 2541, 2547, 2549,  
2555, 2557, 2564, 2571, 2572, 2573,  
2574, 2575, 2577, 2586, 2588, 2591,  
2592, 2595, 2599, 2602, 2604, 2609,  
2619, 2622, 2626, 2644, 2645, 2647,  
2653, 2658, 2660, 2666, 2686, 2688,  
2690, 2703, 2710, 2725, 2731, 2737,  
2742, 2743, 2766, 2796, 2803, 2809,  
2837, 2843, 2848, 2854, 2903, 2904,  
2905, 2906, 2975, 2996, 3018, 3021,  
3029, 3042, 3057, 3068, 3100, 3205,  
3207, 3448, 3603, 3616, 3621, 3627,  
3628, 3635, 3642, 3649, 3656, 3663,  
3664, 3666, 3673, 3679, 3774, 3779,  
3780, 3788, 3794, 3971, 3976, 3983,  
4019, 4025, 4030, 4045, 4068, 4073,  
4125, 4131, 4149, 4154, 4169, 4171,  
4173, 4180, 4196, 4198, 4201, 4207,  
4475, 4504, 4509, 4511, 4520, 4522,  
4523, 4528, 4534, 4556, 4558, 4560,  
4782, 4788, 4799, 4804, 4817, 4822,  
4834, 4849, 4859, 4871, 4894, 5005,  
5023, 5031, 5043, 5055, 5559, 5841,  
5856, 5890, 5906, 5953, 5991, 6022,  
6028, 6034, 6042, 6047, 6053, 6058,  
6072, 6087, 6096, 6104, 6106, 6158,  
6167, 6238, 6484, 6899, 7003, 7006,  
7027, 7029, 7035, 7037, 7044, 7054,  
7254, 7636, 7739, 7745, 7784, 7791,  
7799, 7805, 7807, 8432, 8511, 8523,  
8525, 8565, 8566, 8575, 8577, 8587,  
8592, 8597, 8599, 8607, 8608, 8609,  
8610, 8611, 8612, 8613, 8614, 8615,  
8616, 8626, 8642, 8652, 8668, 8678,  
8680, 8684, 8686, 8690, 8698, 8703,  
8711, 8717, 8724, 8726, 8728, 8729,  
8731, 8733, 8735, 8737, 8739, 8741,  
8743, 8745, 8747, 8749, 8754, 8755,  
8756, 8757, 8758, 8759, 8760, 8761,  
8762, 8763, 8775, 8778, 9105, 9107,  
9144, 9150, 9154, 9264, 9337, 9372,  
9434, 9439, 9444, 9446, 9448, 9450,  
9456, 9464, 9470, 9476, 9617, 9619,  
9636, 9769, 9786, 9788, 9790, 10123,  
10124, 10133, 10146, 10148, 10150,  
10152, 10350, 10352, 10427, 10535,  
10543, 10581, 10598, 10653, 10704,  
10713, 10732, 10733, 10741, 10747,  
10755, 10765, 10770, 10776, 10782,  
10855, 10857, 10859, 10907, 10915,  
10921, 10928, 10929, 10935, 10937,  
10944, 10949, 10957, 10967, 10969,  
10978, 10980, 10981, 10983, 11031,  
11036, 11041, 11056, 11058, 11060,  
11071, 11072, 11073, 11075, 11093,  
11095, 11183, 11185, 11187, 11192,  
11197, 11199, 11201, 11208, 11222,  
11232, 11242, 11249, 11253, 11260,  
11333, 11446, 11451, 11456, 11461,  
11467, 11481, 11519, 11587, 11714,  
11715, 11719, 11720, 12296, 12357,  
12425, 12460, 12549, 12552, 12553,  
12554, 12555, 12566, 12581, 12586,  
12591, 12596, 12601, 12603, 12612,  
12614, 12621, 12624, 12631, 12637,  
12654, 12661, 12664, 12672, 12685,

12687, 12690, 12692, 12701, 12706,  
12711, 12714, 12725, 12726, 12727,  
12728, 12735, 12742, 12744, 12751,  
12762, 12774, 12783, 12789, 12795,  
12797, 12802, 12807, 12814, 12819,  
12825, 12836, 12837, 12838, 12839,  
12847, 12887, 12896, 12915, 12917,  
12930, 12937, 12953, 12964, 12972,  
12978, 12981, 12986, 12998, 13004,  
13005, 13007, 13015, 13021, 13028,  
13030, 13032, 13045, 13047, 13049,  
13057, 13065, 13071, 13078, 13080,  
13085, 13087, 13089, 13090, 13098,  
13110, 13119, 13128, 13133, 13139,  
13162, 13163, 13164, 13166, 13219,  
13257, 13258, 13422, 13427, 13432,  
13437, 13442, 13451, 13457, 13462,  
13467, 13472, 13477, 13479, 13485,  
13487, 13495, 13497, 13499, 13525,  
13551, 13553, 13555, 13563, 13565,  
13576, 13585, 13588, 13599, 13608,  
13610, 13612, 13620, 13622, 13629,  
13650, 13660, 13665, 13670, 13671,  
13673, 13681, 13683, 13691, 13697,  
13703, 13722, 13724, 13733, 13739,  
13746, 13748, 13751, 13761, 13768,  
13770, 13778, 13783, 13788, 13799,  
13806, 13808, 13814, 13816, 13821,  
13823, 13829, 13830, 13835, 13836,  
13837, 13838, 13842, 13847, 13852,  
13855, 13857, 13865, 13870, 13880,  
13883, 13898, 13900, 13906, 13915,  
13930, 13940, 13944, 14036, 14044,  
14051, 14092, 14094, 14100, 14106,  
14108, 14113, 14118, 14134, 14150,  
14164, 14261, 14267, 14293, 14299,  
14331, 14341, 14352, 14378, 14385,  
14445, 14452, 14474, 14484, 14553,  
14563, 14600, 14662, 14703, 14705,  
14722, 14728, 14751, 14781, 14802,  
14811, 14890, 14910, 14930, 14953,  
14960, 14987, 15090, 15104, 15131,  
15140, 15142, 15163, 15168, 15174,  
15179, 15247, 15270, 15282, 15291,  
15297, 15302, 16180, 16186, 16194,  
16201, 16208, 16212, 16218, 16251,  
16261, 16264, 16368, 16377, 16410,  
16415, 16417, 16426, 16432, 16437,  
16465, 16472, 16570, 16576, 16608,  
16621, 16676, 16757, 16787, 16815,  
16820, 16842, 16877, 16879, 16887,  
16893, 16901, 16907, 16909, 16911,  
16922, 16964, 16974, 16998, 17013,  
17023, 17051, 17069, 17073, 17075,  
17098, 17099, 17100, 17107, 17109,  
17163, 17183, 17188, 17190, 17191,  
17197, 17205, 17211, 17229, 17237,  
17245, 17258, 17260, 17267, 17269,  
17366, 17373, 17387, 17392, 17398,  
17409, 17414, 17421, 17423, 17425,  
17427, 17429, 17431, 17433, 17443,  
17448, 17453, 17458, 17463, 17465,  
17471, 17489, 17497, 17505, 17511,  
17517, 17525, 17533, 17539, 17545,  
17552, 17568, 17578, 17580, 17616,  
17630, 17636, 17668, 17700, 17702,  
17704, 17710, 17716, 17728, 17736,  
17748, 17756, 17789, 17822, 17824,  
17826, 17828, 17830, 17835, 17840,  
17845, 17850, 17851, 17852, 17853,  
17854, 17855, 17856, 17857, 17858,  
17859, 17860, 17861, 17862, 17863,  
17864, 17865, 17866, 17875, 17876,  
17885, 17891, 17893, 17902, 17909,  
17915, 17917, 17919, 17935, 17946,  
17969, 18002, 18042, 18043, 18052,  
18053, 18065, 18066, 18067, 18069,  
18075, 18081, 18111, 18112, 18151,  
18247, 18349, 18351, 18360, 18367,  
18370, 18383, 18389, 18427, 18437,  
18444, 18451, 18458, 18465, 18498,  
18508, 18516, 18523, 18529, 18539,  
18541, 18543, 18552, 18555, 18564,  
18572, 18596, 18597, 18600, 18602,  
18605, 18615, 18626, 18628, 18631,  
18637, 18638, 18646, 18662, 18669,  
18677, 18679, 18693, 18695, 18696,  
18697, 18699, 18704, 18752, 18822,  
18828, 18834, 18840, 18871, 18877,  
18907, 18968, 18984, 18986, 18988,  
18990, 18992, 19001, 19003, 19017,  
19036, 19068, 19070, 19072, 19074,  
19076, 19085, 19087, 19101, 19212,  
19242, 19389, 19401, 19402, 19410,  
19419, 19428, 19437, 19439, 19441,  
19443, 19445, 19447, 19449, 19451,  
19453, 19455, 19457, 19459, 19461,  
19468, 19474, 19481, 19482, 19483,  
19484, 19487, 19581, 19589, 19591,  
19593, 19603, 19613, 19682, 19788,  
19835, 19840, 19846, 19854, 19858,  
19874, 19878, 19889, 19969, 19987,  
19999, 20027, 20037, 20050, 20052,  
20073, 20087, 20145, 20150, 20152,  
20160, 20168, 20176, 20178, 20190,  
20196, 20209, 20211, 20213, 20215,  
20217, 20225, 20230, 20235, 20240,  
20245, 20247, 20253, 20255, 20263,

20271, 20277, 20283, 20291, 20299,  
20305, 20311, 20318, 20332, 20366,  
20368, 20374, 20387, 20388, 20395,  
20403, 20405, 20407, 20493, 20496,  
20501, 20504, 20572, 20603, 20615,  
20624, 20630, 20632, 20634, 20644,  
20650, 20655, 20662, 20670, 20674,  
20681, 20683, 20691, 20695, 20702,  
20712, 20720, 20728, 20740, 20749,  
20763, 20764, 20765, 20766, 20768,  
20784, 20795, 20803, 20808, 20814,  
20955, 21241, 21764, 21766, 21831,  
21840, 21846, 21848, 21853, 21857,  
21861, 21865, 21871, 21883, 21887,  
21891, 21954, 21966, 22173, 22175,  
22185, 22198, 22210, 22282, 22284,  
22286, 22297, 22356, 22358, 22365,  
22371, 22389, 22391, 22399, 22408,  
22418, 22531, 22532, 22533, 22534,  
22535, 22536, 22537, 22538, 22539,  
22540, 22550, 22574, 22576, 22578,  
22587, 22589, 22596, 22608, 22609,  
22611, 22621, 22631, 22641, 22651,  
22659, 22661, 22668, 22670, 22671,  
22676, 22683, 22697, 22699, 22715,  
22716, 22724, 22726, 22735, 22737,  
22749, 22754, 22758, 22763, 22765,  
22767, 22769, 22771, 22778, 22780,  
22788, 22790, 22802, 22804, 22806,  
22808, 22832, 22834, 22836, 22837,  
22838, 22840, 22842, 22844, 22846,  
22864, 22879, 22880, 22886, 22902,  
22908, 23036, 23037, 23038, 23039,  
23040, 23041, 23042, 23047, 23050,  
23096, 23098, 23100, 23102, 23108,  
23112, 23114, 23123, 23124, 23133,  
23146, 23159, 23166, 23180, 23196,  
23208, 23219, 23229, 23235, 23246,  
23256, 23282, 23293, 23310, 23321,  
23326, 23346, 23348, 23359, 23364,  
23377, 23400, 23401, 23405, 23422,  
23423, 23447, 23455, 23473, 23502,  
23528, 23532, 23535, 23537, 23543,  
23555, 23567, 23574, 23580, 23588,  
23611, 23626, 23645, 23653, 23668,  
23683, 23694, 23704, 23714, 23719,  
23728, 23745, 23758, 23763, 23769,  
23771, 23778, 23808, 23836, 23852,  
23863, 23868, 23886, 23904, 23915,  
23930, 23935, 23946, 23956, 23966,  
23982, 24026, 24031, 24038, 24046,  
24052, 24057, 24061, 24078, 24086,  
24118, 24135, 24149, 24168, 24176,  
24185, 24194, 24205, 24207, 24221,  
24231, 24232, 24249, 24256, 24261,  
24274, 24287, 24292, 24320, 24334,  
24362, 24363, 24367, 24384, 24406,  
24408, 24419, 24451, 24455, 24470,  
24487, 24511, 24513, 24515, 24517,  
24527, 24532, 24543, 24555, 24566,  
24579, 24599, 24617, 24619, 24631,  
24637, 24645, 24659, 24666, 24677,  
24684, 24698, 24794, 24816, 24818,  
24835, 24857, 24862, 24879, 24906,  
24907, 24908, 24909, 24925, 24936,  
24944, 24956, 24962, 24968, 24976,  
24984, 24990, 24996, 25004, 25012,  
25030, 25043, 25065, 25113, 25119,  
25130, 25154, 25156, 25158, 25160,  
25168, 25172, 25179, 25186, 25187,  
25188, 25189, 25190, 25191, 25194,  
25196, 25225, 25233, 25244, 25246,  
25248, 25250, 25257, 25281, 25283,  
25293, 25308, 25317, 25331, 25339,  
25347, 25354, 25361, 25369, 25379,  
25393, 25404, 25405, 25411, 25428,  
25435, 25437, 25444, 25449, 25466,  
25467, 25468, 25487, 25493, 25503,  
25515, 25522, 25536, 25544, 25582,  
25591, 25612, 25614, 25616, 25625,  
25636, 25648, 25663, 25676, 25689,  
25697, 25715, 25733, 25740, 25748,  
25758, 25759, 25768, 25769, 25778,  
25788, 25802, 25812, 25823, 25831,  
25833, 25844, 25850, 25885, 25906,  
25908, 25910, 25912, 25919, 25928,  
25933, 25940, 25947, 25967, 25972,  
25989, 26000, 26005, 26015, 26017,  
26027, 26034, 26036, 26042, 26044,  
26046, 26050, 26069, 26070, 26075,  
26083, 26084, 26107, 26120, 26127,  
26135, 26136, 26137, 26138, 26139,  
26140, 26148, 26154, 26156, 26158,  
26180, 26185, 26195, 26205, 26216,  
26229, 26240, 26245, 26252, 26261,  
26263, 26272, 26281, 26295, 26297,  
26299, 26312, 26322, 26327, 26336,  
26344, 26351, 26357, 26366, 26368,  
26380, 26385, 26393, 26398, 26408,  
26414, 26420, 26427, 26434, 26436,  
26441, 26443, 26448, 26450, 26464,  
26474, 26486, 26491, 26498, 26508,  
26510, 26512, 26523, 26537, 26551,  
26571, 26584, 26586, 26591, 26604,  
26609, 26617, 26622, 26632, 26644,  
26674, 26675, 26676, 26678, 26680,  
26682, 26696, 26702, 26711, 26730,  
26736, 26746, 26765, 26773, 26806,

26812, 26821, 26823, 26837, 26896,  
26904, 26922, 26939, 26940, 26945,  
26970, 26993, 27021, 27037, 27047,  
27058, 27079, 27094, 27099, 27104,  
27106, 27120, 27126, 27141, 27149,  
27159, 27169, 27182, 27200, 27206,  
27220, 27235, 27273, 27275, 27277,  
27279, 27281, 27296, 27311, 27326,  
27341, 27356, 27371, 27379, 27393,  
27395, 27401, 27413, 27421, 27428,  
27654, 27661, 27698, 27706, 27707,  
27718, 27725, 27727, 27733, 27744,  
27754, 27761, 27768, 27783, 27822,  
27835, 27866, 27872, 27879, 27899,  
27901, 27918, 27933, 27946, 27953,  
27958, 27960, 27969, 27982, 27985,  
28006, 28019, 28034, 28052, 28067,  
28077, 28086, 28099, 28115, 28132,  
28145, 28151, 28153, 28158, 28159,  
28160, 28161, 28164, 28169, 28175,  
28180, 28182, 28205, 28213, 28215,  
28218, 28223, 28229, 28234, 28236,  
28259, 28284, 28293, 28294, 28296,  
28301, 28303, 28308, 28310, 28320,  
28328, 28336, 28338, 28341, 28346,  
28351, 28353, 28354, 28356, 28361,  
28366, 28368, 28373, 28380, 28394,  
28399, 28401, 28411, 28413, 28415,  
28417, 28419, 28430, 28440, 28442,  
28448, 28455, 28461, 28471, 28476,  
28478, 28486, 28487, 28501, 28508,  
28514, 28515, 28528, 28543, 28549,  
28571, 28586, 28596, 28617, 28626,  
28649, 28667, 28678, 28683, 28694,  
28711, 28716, 28763, 28769, 28783,  
28852, 28860, 28868, 28874, 28881,  
28886, 28892, 28904, 28996, 29101,  
29103, 29110, 29122, 29126, 29130,  
29302, 29308, 29319, 29331, 29345,  
29364, 29381, 29387, 29392, 29436,  
29438, 29440, 29443, 29446, 29452,  
29458, 29464, 29465, 29467, 29472,  
29473, 29696, 29716, 29790, 29795,  
29804, 29805, 29806, 29807, 29808,  
29814, 29816, 29869, 29875, 29877,  
29887, 29902, 29912, 29923, 29937,  
29954, 29967, 29969, 29970, 30021,  
30024, 30039, 30056, 30058, 30072,  
30082, 30104, 30105, 30127, 30129,  
30210, 30218, 30225, 30228, 30230,  
30236, 30247, 30259, 30283, 30303,  
30319, 30326, 30340, 30351, 30360,  
30365, 30371, 30396, 30409, 30414,  
30419, 30430, 30432, 30437, 30443,  
30457, 30463, 30489, 30499, 30509,  
30511, 30522, 30528, 30533, 30541,  
30542, 30557, 30558, 30571, 30576,  
30586, 30593, 30609, 30611, 30613,  
30615, 30617, 30619, 30621, 30623,  
30625, 30635, 30644, 30649, 30655,  
30663, 30666, 30668, 30674, 30685,  
30690, 30704, 30716, 30730, 30737,  
30743, 30748, 30754, 30768, 30779,  
30788, 30795, 30802, 30812, 30821,  
30826, 30837, 30839, 30853, 30862,  
30864, 30871, 30876, 30881, 30886,  
30891, 30895, 30897, 30899, 30919,  
30925, 30931, 30938, 30949, 30965,  
30988, 30990, 30992, 31001, 31013,  
31018, 31035, 31039, 31062, 31066,  
31077, 31085, 31092, 31097, 31098,  
31114, 31138, 31143, 31145, 31150,  
31161, 31178, 31189, 31202, 31220,  
31225, 31256, 31266, 31283, 31293,  
31302, 31311, 31319, 31329, 31342,  
31371, 31405, 31434, 31459, 31475,  
31483, 31493, 31501, 31516, 31536,  
31538, 31554, 31570, 31574, 31585,  
31596, 31609, 31637, 31650, 31657,  
31662, 31687, 31700, 31713, 31720,  
31725, 31737, 31744, 31756, 31763,  
31779, 31796, 31810, 31815, 31841,  
31924, 31926, 31932, 31943, 31954,  
31960, 31978, 31991, 32001, 32017,  
32022, 32027, 32045, 32046, 32052,  
32055, 32060, 32075, 32080, 32098,  
32100, 32102, 32104, 32106, 32108,  
32110, 32112, 32114, 32116, 32121,  
32127, 32134, 32141, 32151, 32168,  
32173, 32180, 32185, 32203, 32205,  
32206, 32211, 32217, 32223, 32228,  
32238, 32245, 32256, 32258, 32260,  
32276, 32285, 32292, 32294, 32296,  
32302, 32313, 32314, 32319, 32324,  
32331, 32342, 32347, 32349, 32351,  
32356, 32361, 32372, 32383, 32388,  
32395, 32400, 32411, 32413, 32436,  
32437, 32443, 32448, 32460, 32528,  
32606, 32896, 34868, 34869, 34913,  
34915, 34917, 34919, 34921, 34923,  
34930, 34935, 34942, 34944, 34950,  
35093, 35100, 35105, 35107, 35114,  
35122, 35124, 35133, 35143, 35145,  
35147, 35149, 35151, 35156, 35164,  
35170, 35182, 35184, 35185, 35186,  
35187, 35188, 35189, 35190, 35191,  
35198, 35200, 35209, 35218, 35232,  
35274, 35281, 35283, 35288, 35293,

35306, 35316, 35660, 35718, 35859,  
35870, 35877, 35885, 35891, 35899,  
35901, 35933, 35935, 36014, 36020,  
36022, 36031, 36044, 36059, 36072,  
36086, 36177, 36203, 36214, 36221,  
36223, 36236, 36242, 36253, 36268,  
36274, 36279, 36289, 36295, 36305,  
36360, 36361, 36415, 36603, 36610,  
36617, 36621, 36629, 36641, 36668,  
36670, 36671, 36844, 36884, 36886,  
36888, 36890, 36892, 36894, 36896,  
36898, 36900, 36905, 36911, 36917,  
36920, 36921, 36931, 36939, 36941,  
36947, 36953, 37046, 37053, 37087,  
37247, 37255, 37275, 37277, 37278,  
37281, 37283, 37285, 37287, 37295  
\cs\_new:Npx . . . . . 15, 38, 39,  
405, 406, 1925, 1936, 1949, 1954,  
2934, 3997, 4278, 4805, 4807, 4809,  
4811, 4813, 4815, 10580, 10592,  
11084, 15304, 21943, 22690, 23514,  
24098, 25252, 27260, 27266, 27878,  
30264, 30477, 30637, 35907, 36064,  
36204, 36857, 36859, 36861, 36868  
\cs\_new\_eq:NN . . . 19, 64, 375, 377,  
874, 1739, 1967, 1975, 1980, 1981,  
1982, 2267, 2276, 2313, 2590, 2884,  
2885, 2886, 2887, 2888, 2889, 2890,  
2891, 2892, 2893, 2894, 2895, 2896,  
2897, 2898, 2901, 2902, 3143, 3751,  
3752, 3753, 4472, 4478, 4626, 4627,  
4628, 4727, 4735, 4756, 4795, 4796,  
4797, 4798, 4984, 6770, 7036, 7434,  
7956, 8471, 8473, 8493, 8494, 8789,  
8790, 8791, 8792, 8795, 8796, 8797,  
8798, 9069, 10194, 10216, 10284,  
10429, 10437, 10536, 11030, 11324,  
11365, 11433, 11439, 11709, 11710,  
11711, 12080, 12081, 12426, 12427,  
13268, 13282, 13283, 13286, 13287,  
13353, 13376, 13447, 13448, 13449,  
13450, 13975, 13982, 14310, 14326,  
14328, 15318, 16178, 16451, 16454,  
16479, 16499, 16500, 16501, 16502,  
16503, 16504, 16505, 16506, 16736,  
17112, 17113, 17114, 17115, 17116,  
17117, 17118, 17119, 17120, 17121,  
17122, 17123, 17124, 17125, 17126,  
17127, 17128, 17129, 17130, 17131,  
17132, 17133, 17134, 17135, 17136,  
17137, 17176, 17177, 17178, 17179,  
17180, 17310, 17311, 17314, 17365,  
17615, 17971, 17975, 18061, 18114,  
18115, 18119, 18120, 18121, 18122,  
18123, 18124, 18125, 18126, 18127,  
18128, 18129, 18130, 18131, 18132,  
18133, 18134, 18275, 18276, 18277,  
18278, 18279, 18280, 18281, 18282,  
18283, 18284, 18285, 18286, 18287,  
18288, 18289, 18290, 18870, 18915,  
19113, 19115, 19116, 19117, 19119,  
19122, 19123, 19477, 19478, 19479,  
19710, 19711, 19712, 19713, 19714,  
19715, 19716, 19717, 19888, 20082,  
20083, 20084, 20386, 20415, 20419,  
20420, 20443, 20444, 20498, 20499,  
20500, 20503, 20508, 20512, 20513,  
20574, 20575, 20576, 20580, 20581,  
20623, 22256, 22257, 22528, 22529,  
22530, 22734, 22901, 23179, 23207,  
23215, 23216, 23217, 23226, 23228,  
24025, 24144, 24145, 24146, 24743,  
24754, 24755, 28408, 28410, 28454,  
30702, 30893, 30986, 31026, 31079,  
31081, 31083, 31112, 31136, 31572,  
31854, 31856, 32054, 32133, 32140,  
32216, 32222, 32895, 32934, 32935,  
32936, 32970, 32971, 32982, 32983,  
32984, 33089, 33120, 33121, 33194,  
33209, 33210, 33611, 33838, 33839,  
33840, 33841, 33842, 33843, 34839,  
34840, 35858, 36016, 36040, 36055,  
36593, 36985, 37184, 37185, 37619  
\cs\_new\_nopar:Nn . . . . . 17, 2055, 2120  
\cs\_new\_nopar:Npn . . . . .  
15, 375, 376, 1925, 1933, 1941, 1947  
\cs\_new\_nopar:Npx . . . . .  
15, 1925, 1934, 1941, 1948  
\cs\_new\_protected:Nn . . 17, 2055, 2120  
\cs\_new\_protected:Npn . . . . . 15,  
411, 1409, 1603, 1624, 1925, 1939,  
1961, 1965, 1967, 1968, 1969, 1970,  
1971, 1972, 1973, 1974, 1975, 1980,  
1981, 1982, 1983, 1985, 2040, 2050,  
2052, 2063, 2072, 2170, 2179, 2181,  
2183, 2185, 2187, 2195, 2197, 2198,  
2200, 2201, 2203, 2211, 2213, 2215,  
2279, 2314, 2486, 2515, 2585, 2616,  
2908, 2921, 2939, 2943, 2946, 2955,  
3079, 3096, 3106, 3115, 3139, 3147,  
3159, 3167, 3178, 3180, 3182, 3184,  
3186, 3197, 3199, 3212, 3220, 3231,  
3250, 3252, 3254, 3256, 3258, 3343,  
3362, 3369, 3381, 3414, 3433, 3435,  
3437, 3456, 3459, 3462, 3468, 3474,  
3489, 3498, 3518, 3528, 3539, 3549,  
3559, 3560, 3567, 3573, 3583, 3593,  
3689, 3695, 3697, 3712, 3796, 3807,

3825, 3835, 3837, 3861, 3868, 3880,  
3883, 3886, 3896, 3904, 3911, 3920,  
3935, 3952, 3963, 4078, 4080, 4087,  
4096, 4102, 4104, 4106, 4116, 4118,  
4120, 4208, 4224, 4235, 4254, 4280,  
4292, 4317, 4328, 4342, 4350, 4357,  
4362, 4369, 4371, 4381, 4391, 4422,  
4428, 4430, 4435, 4440, 4449, 4473,  
4476, 4479, 4481, 4490, 4496, 4502,  
4562, 4564, 4565, 4570, 4576, 4584,  
4594, 4608, 4629, 4639, 4647, 4649,  
4657, 4669, 4688, 4690, 4695, 4703,  
4705, 4712, 4717, 4719, 4721, 4728,  
4736, 4738, 4740, 4742, 4749, 4754,  
4757, 4763, 4999, 5063, 5076, 5087,  
5100, 5133, 5162, 5171, 5176, 5181,  
5186, 5207, 5216, 5223, 5232, 5237,  
5244, 5257, 5259, 5261, 5263, 5269,  
5291, 5304, 5331, 5336, 5370, 5405,  
5426, 5428, 5436, 5438, 5440, 5442,  
5444, 5446, 5450, 5461, 5477, 5490,  
5496, 5507, 5520, 5526, 5545, 5565,  
5596, 5607, 5622, 5635, 5653, 5661,  
5666, 5668, 5670, 5687, 5706, 5708,  
5731, 5743, 5763, 5784, 5791, 5798,  
5809, 5816, 5822, 5880, 5932, 5941,  
5954, 5969, 5978, 5997, 6016, 6173,  
6244, 6254, 6256, 6258, 6265, 6310,  
6323, 6339, 6341, 6343, 6348, 6363,  
6369, 6392, 6415, 6430, 6437, 6444,  
6446, 6448, 6455, 6469, 6485, 6494,  
6508, 6520, 6537, 6546, 6548, 6560,  
6569, 6581, 6594, 6601, 6621, 6652,  
6686, 6704, 6713, 6719, 6725, 6731,  
6774, 6783, 6797, 6816, 6843, 6848,  
6858, 6870, 6908, 6917, 6929, 6936,  
6938, 6940, 6960, 6965, 6971, 6982,  
6987, 6992, 7008, 7060, 7079, 7081,  
7124, 7148, 7165, 7171, 7173, 7193,  
7219, 7230, 7239, 7248, 7281, 7295,  
7306, 7312, 7321, 7329, 7364, 7370,  
7373, 7381, 7387, 7390, 7399, 7402,  
7405, 7408, 7413, 7422, 7425, 7428,  
7433, 7439, 7444, 7449, 7454, 7455,  
7456, 7464, 7465, 7466, 7487, 7489,  
7491, 7499, 7501, 7503, 7507, 7508,  
7523, 7540, 7542, 7544, 7546, 7563,  
7565, 7567, 7582, 7590, 7600, 7611,  
7620, 7637, 7649, 7659, 7668, 7708,  
7763, 7768, 7812, 7834, 7847, 7849,  
7880, 7882, 7911, 7927, 7938, 7943,  
7957, 7963, 7965, 7966, 7972, 7974,  
7975, 7981, 7983, 7985, 7991, 7993,  
7995, 8003, 8023, 8029, 8035, 8041,  
8049, 8051, 8053, 8055, 8057, 8059,  
8061, 8063, 8065, 8070, 8099, 8109,  
8114, 8123, 8125, 8135, 8448, 8450,  
8452, 8459, 8473, 8475, 8481, 8483,  
8485, 8487, 8497, 8502, 8526, 8528,  
8530, 8532, 8534, 8785, 8872, 8889,  
8942, 8948, 8956, 8967, 8990, 9020,  
9024, 9052, 9056, 9060, 9065, 9120,  
9124, 9188, 9194, 9272, 9280, 9285,  
9287, 9294, 9296, 9303, 9344, 9373,  
9393, 9398, 9409, 9485, 9487, 9528,  
9551, 9607, 9621, 9646, 9668, 9669,  
9682, 9687, 9713, 9722, 9724, 9726,  
9743, 9770, 9772, 9774, 9776, 9778,  
9780, 9782, 9784, 10194, 10198,  
10212, 10223, 10244, 10256, 10258,  
10260, 10272, 10273, 10274, 10301,  
10303, 10314, 10316, 10335, 10337,  
10339, 10354, 10356, 10358, 10364,  
10371, 10380, 10382, 10384, 10389,  
10429, 10433, 10445, 10459, 10467,  
10479, 10481, 10483, 10495, 10496,  
10497, 10507, 10510, 10513, 10519,  
10525, 10532, 10534, 10544, 10575,  
10586, 10604, 10639, 10662, 10674,  
10693, 10697, 10786, 10802, 10811,  
10819, 10828, 10835, 10841, 10990,  
11024, 11121, 11175, 11266, 11268,  
11270, 11272, 11282, 11310, 11383,  
11388, 11394, 11395, 11400, 11419,  
11434, 11440, 11491, 11496, 11503,  
11504, 11505, 11532, 11545, 11557,  
11567, 11569, 11571, 11580, 11588,  
11716, 11717, 11723, 12059, 12061,  
12063, 12065, 12067, 12072, 12082,  
12088, 12093, 12100, 12102, 12106,  
12108, 12112, 12114, 12118, 12126,  
12144, 12146, 12148, 12150, 12152,  
12154, 12164, 12169, 12174, 12182,  
12184, 12189, 12194, 12202, 12212,  
12214, 12219, 12227, 12229, 12231,  
12236, 12244, 12262, 12268, 12270,  
12272, 12274, 12286, 12305, 12320,  
12338, 12360, 12362, 12364, 12366,  
12372, 12394, 12400, 12428, 12430,  
12434, 12436, 12522, 12523, 12524,  
12638, 12651, 12678, 12680, 12682,  
12754, 12756, 13051, 13053, 13185,  
13187, 13189, 13205, 13207, 13220,  
13222, 13314, 13316, 13318, 13320,  
13326, 13341, 13354, 13356, 13360,  
13362, 13507, 13522, 13531, 13541,  
13543, 13976, 13983, 13992, 14124,  
14140, 14156, 14162, 14166, 14168,

14188, 14206, 14214, 14224, 14233,  
14317, 14325, 14327, 14329, 14339,  
14345, 14369, 14380, 14386, 14389,  
14391, 14396, 14422, 14428, 14434,  
14462, 14536, 14584, 14637, 14720,  
14726, 14749, 14779, 14800, 14875,  
14971, 14976, 14978, 14980, 15056,  
15058, 15060, 15065, 15075, 15154,  
15159, 15161, 15214, 15216, 15218,  
15223, 15233, 16167, 16269, 16271,  
16282, 16289, 16295, 16313, 16322,  
16327, 16332, 16337, 16342, 16348,  
16353, 16360, 16366, 16375, 16385,  
16387, 16389, 16391, 16393, 16398,  
16442, 16481, 16487, 16490, 16493,  
16496, 16507, 16512, 16517, 16522,  
16533, 16539, 16541, 16543, 16545,  
16547, 16582, 16584, 16592, 16600,  
16613, 16615, 16623, 16625, 16627,  
16639, 16641, 16643, 16665, 16667,  
16669, 16696, 16697, 16698, 16720,  
16731, 16761, 16769, 16779, 16790,  
16792, 16794, 16796, 16804, 16822,  
16824, 16826, 16935, 16940, 16945,  
16951, 16957, 16986, 17003, 17031,  
17033, 17035, 17041, 17043, 17045,  
17144, 17146, 17148, 17276, 17282,  
17284, 17317, 17318, 17321, 17323,  
17327, 17329, 17335, 17337, 17339,  
17341, 17347, 17349, 17351, 17353,  
17359, 17361, 17367, 17582, 17584,  
17586, 17593, 17595, 17597, 17609,  
17973, 17977, 18000, 18005, 18006,  
18017, 18019, 18020, 18021, 18068,  
18116, 18135, 18137, 18139, 18162,  
18164, 18166, 18181, 18183, 18187,  
18189, 18191, 18200, 18202, 18204,  
18213, 18221, 18224, 18226, 18228,  
18236, 18264, 18293, 18295, 18297,  
18315, 18317, 18319, 18354, 18356,  
18400, 18466, 18482, 18488, 18491,  
18493, 18710, 18712, 18714, 18732,  
18733, 18734, 18750, 18754, 18756,  
18758, 18760, 18762, 18764, 18766,  
18768, 18770, 18772, 18774, 18776,  
18778, 18780, 18782, 18784, 18786,  
18788, 18790, 18792, 18794, 18796,  
18798, 18800, 18802, 18804, 18806,  
18808, 18810, 18812, 18814, 18816,  
18818, 18820, 18824, 18826, 18830,  
18832, 18836, 18838, 18842, 18854,  
19488, 19490, 19492, 19497, 19504,  
19513, 19524, 19529, 19543, 19551,  
19569, 19571, 19573, 19575, 19577,  
19579, 19639, 19656, 19661, 19668,  
19673, 19675, 19692, 19698, 19701,  
19704, 19707, 19723, 19726, 19729,  
19735, 19741, 19747, 19754, 19762,  
19770, 19772, 19778, 19780, 19789,  
19795, 19805, 19813, 19822, 19914,  
19915, 19916, 19935, 19937, 19939,  
20012, 20054, 20056, 20058, 20088,  
20094, 20100, 20101, 20105, 20107,  
20115, 20117, 20121, 20124, 20127,  
20129, 20136, 20138, 20219, 20341,  
20348, 20360, 20417, 20421, 20431,  
20437, 20447, 20449, 20457, 20459,  
20463, 20465, 20467, 20469, 20473,  
20475, 20510, 20514, 20522, 20528,  
20534, 20536, 20540, 20542, 20550,  
20552, 20556, 20558, 20560, 20562,  
20566, 20568, 20578, 20582, 20861,  
20868, 20870, 20877, 20882, 20887,  
20900, 20907, 20911, 20920, 20925,  
20934, 20940, 20957, 20960, 20963,  
20979, 20981, 20983, 20999, 21010,  
21012, 21014, 21031, 21034, 21036,  
21046, 21061, 21074, 21079, 21090,  
21092, 21094, 21113, 21121, 21126,  
21140, 21150, 21178, 21187, 21196,  
21229, 21242, 21253, 21259, 21261,  
21263, 21265, 21267, 21269, 21271,  
21273, 21275, 21277, 21279, 21281,  
21283, 21285, 21287, 21289, 21291,  
21293, 21295, 21297, 21299, 21301,  
21303, 21305, 21307, 21309, 21311,  
21313, 21315, 21317, 21319, 21321,  
21323, 21325, 21327, 21329, 21331,  
21333, 21335, 21337, 21339, 21341,  
21343, 21345, 21347, 21349, 21351,  
21353, 21355, 21357, 21359, 21361,  
21363, 21365, 21367, 21369, 21371,  
21373, 21375, 21377, 21379, 21381,  
21383, 21385, 21387, 21389, 21391,  
21393, 21395, 21397, 21399, 21401,  
21403, 21405, 21407, 21409, 21411,  
21413, 21415, 21417, 21419, 21421,  
21423, 21425, 21427, 21429, 21431,  
21433, 21435, 21437, 21439, 21441,  
21462, 21464, 21470, 21476, 21482,  
21489, 21492, 21511, 21518, 21524,  
21531, 21534, 21553, 21574, 21576,  
21582, 21590, 21595, 21600, 21620,  
21625, 21629, 21635, 21640, 21646,  
21660, 21686, 21705, 21720, 21734,  
21750, 21774, 21798, 21830, 21917,  
21919, 21921, 22034, 22040, 22125,  
22127, 22188, 22225, 22251, 22260,

22269, 22304, 22306, 22314, 22325,  
22333, 22340, 22346, 22374, 22383,  
22427, 22432, 22442, 22444, 22446,  
22457, 22462, 22467, 22481, 22486,  
22491, 22506, 22517, 22541, 22544,  
22655, 22928, 22945, 22947, 22949,  
22951, 22979, 22981, 22983, 22985,  
23005, 23007, 23009, 23011, 23013,  
23015, 23017, 23019, 23021, 24742,  
24745, 24747, 24749, 24758, 24759,  
24762, 24764, 24768, 24769, 24770,  
24771, 24772, 24778, 24780, 24782,  
24801, 24803, 25082, 25089, 25101,  
27911, 28736, 28749, 28788, 28798,  
28810, 28815, 28825, 28833, 28839,  
28918, 28923, 28930, 28932, 28934,  
28956, 28958, 28970, 28981, 29003,  
29008, 29021, 29034, 29042, 29051,  
29065, 29076, 29082, 29163, 29176,  
29183, 30598, 31087, 32262, 32465,  
32898, 32900, 32906, 32908, 32912,  
32914, 32918, 32920, 32924, 32926,  
32940, 32943, 32949, 32952, 32958,  
32961, 32967, 32974, 32976, 32978,  
32980, 32997, 32999, 33008, 33011,  
33014, 33017, 33019, 33026, 33044,  
33046, 33051, 33058, 33063, 33070,  
33076, 33084, 33090, 33096, 33104,  
33109, 33114, 33116, 33118, 33124,  
33126, 33128, 33133, 33138, 33143,  
33150, 33155, 33162, 33167, 33174,  
33180, 33188, 33195, 33201, 33213,  
33216, 33234, 33237, 33240, 33250,  
33284, 33295, 33306, 33317, 33328,  
33339, 33352, 33358, 33364, 33379,  
33386, 33396, 33401, 33404, 33407,  
33421, 33424, 33427, 33441, 33444,  
33447, 33458, 33461, 33464, 33479,  
33482, 33485, 33494, 33508, 33511,  
33514, 33520, 33526, 33538, 33624,  
33633, 33642, 33651, 33664, 33677,  
33690, 33696, 33702, 33737, 33750,  
33763, 33764, 33765, 33772, 33779,  
33805, 33806, 33807, 33819, 33844,  
33854, 33861, 33868, 33871, 33874,  
33886, 33889, 33892, 33904, 33910,  
33916, 33922, 33924, 33926, 33932,  
33953, 33955, 33957, 33963, 33997,  
34012, 34108, 34111, 34114, 34154,  
34172, 34178, 34184, 34196, 34215,  
34224, 34235, 34241, 34246, 34254,  
34267, 34274, 34281, 34293, 34315,  
34318, 34321, 34336, 34343, 34349,  
34358, 34365, 34373, 34379, 34385,  
34424, 34430, 34436, 34457, 34466,  
34485, 34490, 34504, 34509, 34522,  
34533, 34545, 34559, 34620, 34625,  
34662, 34670, 34694, 34738, 34746,  
34770, 34773, 34776, 34795, 34798,  
34801, 34804, 34807, 34841, 34844,  
34849, 34851, 34870, 34876, 34880,  
34966, 34973, 34980, 34999, 35012,  
35022, 35047, 35070, 35079, 35091,  
35092, 35319, 35333, 35347, 35353,  
35360, 35369, 35380, 35389, 35398,  
35403, 35411, 35419, 35435, 35454,  
35469, 35474, 35479, 35494, 35495,  
35500, 35505, 35511, 35517, 35523,  
35528, 35534, 35548, 35568, 35585,  
35595, 35609, 35642, 35652, 35658,  
35669, 35671, 35702, 35705, 35707,  
35709, 35727, 35764, 35770, 35787,  
35808, 35821, 35834, 35854, 35867,  
35882, 35896, 35905, 35915, 35927,  
35943, 35956, 35992, 35997, 36012,  
36018, 36029, 36042, 36057, 36095,  
36108, 36147, 36153, 36155, 36160,  
36165, 36181, 36186, 36191, 36196,  
36201, 36330, 36343, 36356, 36366,  
36371, 36380, 36385, 36390, 36395,  
36397, 36399, 36582, 36590, 36597,  
36604, 36653, 36655, 36660, 36672,  
36674, 36676, 36719, 36724, 36734,  
36737, 36740, 36743, 36746, 36791,  
36794, 36797, 36852, 36854, 36856,  
36876, 36879, 36961, 36963, 36965,  
36971, 36973, 36975, 36981, 36983,  
36986, 36988, 36998, 37004, 37017,  
37035, 37060, 37062, 37072, 37080,  
37147, 37153, 37169, 37171, 37173,  
37175, 37186, 37191, 37196, 37201,  
37206, 37223, 37224, 37226, 37229,  
37232, 37243, 37245, 37257, 37262,  
37267, 37310, 37312, 37314, 37316,  
37333, 37346, 37351, 37359, 37361,  
37373, 37392, 37399, 37416, 37421  
\cs\_new\_protected:Npx . . . . . 15,  
405, 406, 411, 1925, 1940, 1961,  
1966, 2057, 2122, 2923, 2927, 2932,  
3091, 3095, 4296, 5399, 5413, 5415,  
9496, 9498, 9500, 9502, 9504, 9513,  
9515, 9517, 10235, 11006, 11406,  
13295, 18860, 19628, 33727, 34956,  
35338, 35578, 35689, 35792, 35798  
\cs\_new\_protected\_nopar:Nn . . . . .  
17, 2055, 2120  
\cs\_new\_protected\_nopar:Npn . . . . .  
15, 1925, 1937, 1942, 1955, 1959

\cs\_new\_protected\_nopar:Npx ....  
..... 15, 1925, 1938, 1955, 1960  
\cs\_parameter\_spec:N ..... 22,  
2232, 2247, 13194, 13227, 37631, 37632  
\cs\_prefix\_spec:N .....  
. 22, 2232, 2238, 13194, 13227, 37615  
\cs\_replacement\_spec:N .....  
. 23, 2232, 2256, 21932, 30190, 37617  
\cs\_set:Nn ..... 17, 380, 2055, 2120  
.cs\_set:Np ..... 225, 21295  
\cs\_set:Npn 14, 16, 62, 63, 367, 376,  
380, 896, 1491, 1493, 1521, 1528,  
1530, 1533, 1534, 1535, 1536, 1537,  
1538, 1539, 1540, 1541, 1542, 1543,  
1544, 1545, 1546, 1547, 1548, 1549,  
1550, 1551, 1552, 1553, 1555, 1556,  
1557, 1558, 1559, 1560, 1561, 1562,  
1563, 1586, 1588, 1590, 1593, 1614,  
1616, 1667, 1670, 1732, 1733, 1734,  
1735, 1783, 1785, 1787, 1789, 1794,  
1800, 1801, 1805, 1812, 1815, 1875,  
1877, 1879, 1881, 1883, 1885, 1887,  
1889, 1908, 1925, 1941, 1949, 1949,  
2055, 2120, 3429, 4767, 4768, 4769,  
5095, 5096, 5674, 5676, 5693, 5695,  
5935, 5936, 6200, 6201, 6202, 6203,  
6229, 6274, 6819, 7126, 9073, 9289,  
9291, 10551, 10873, 11739, 12346,  
12403, 12531, 13343, 15081, 15238,  
16408, 16430, 17213, 17221, 18328,  
18417, 18929, 19501, 19515, 19782,  
21296, 21298, 21869, 22954, 22962,  
22971, 22988, 22996, 23024, 29528,  
32570, 32571, 32633, 32640, 32645,  
32646, 37322, 37324, 37330, 37397  
\cs\_set:Npx 16, 387, 1491, 1495, 1949,  
1950, 6471, 10609, 10610, 10611,  
10612, 10613, 12405, 14436, 14464,  
18404, 19499, 19516, 19556, 19562  
\cs\_set\_eq:NN ..... 19, 64,  
377, 573, 1737, 1967, 1967, 1968,  
1969, 1970, 1971, 1978, 2927, 2945,  
3095, 3691, 3692, 3696, 3864, 3907,  
3932, 4298, 4338, 4624, 6192, 6226,  
6825, 6875, 7712, 7735, 8038, 8076,  
8077, 8079, 8080, 8081, 8103, 8482,  
8484, 8819, 10615, 10616, 10617,  
10618, 10620, 10622, 10623, 12060,  
12062, 14175, 14184, 14983, 16671,  
16672, 16674, 16828, 16829, 16840,  
18009, 18863, 19150, 19495, 19554,  
19561, 21050, 21066, 21070, 21145,  
21156, 21166, 29080, 37319, 37329  
\cs\_set\_nopar:Nn ..... 18, 2055, 2120  
\cs\_set\_nopar:Npn ..... 14, 16,  
187, 376, 1491, 1491, 1520, 1578,  
1579, 1941, 1943, 21001, 21077,  
21644, 29489, 29490, 29491, 29495,  
29496, 29826, 29827, 29836, 29838  
\cs\_set\_nopar:Npx 16, 451, 934, 938,  
952, 1402, 1491, 1492, 1524, 1941,  
1944, 2076, 2316, 2517, 4256, 4492,  
4498, 12080, 20902, 20915, 20922,  
20929, 20930, 21055, 21632, 21637,  
29844, 37170, 37188, 37193, 37227  
\cs\_set\_protected:Nn .. 18, 2055, 2120  
.cs\_set\_protected:Np .... 225, 21295  
\cs\_set\_protected:Npn .....  
. .... 15, 16, 377, 161, 1491, 1501,  
1507, 1509, 1511, 1513, 1515, 1517,  
1522, 1565, 1566, 1571, 1576, 1577,  
1580, 1592, 1594, 1596, 1598, 1599,  
1600, 1602, 1604, 1613, 1615, 1617,  
1619, 1620, 1621, 1623, 1625, 1634,  
1646, 1672, 1689, 1708, 1716, 1724,  
1736, 1738, 1740, 1752, 1766, 1803,  
1891, 1904, 1906, 1910, 1912, 1914,  
1922, 1927, 1961, 1961, 1995, 2016,  
3113, 3274, 3693, 4212, 5411, 5448,  
6177, 6186, 6188, 6190, 6193, 6195,  
6204, 6206, 6211, 6213, 6218, 6220,  
6222, 6224, 6227, 6984, 6985, 7505,  
9417, 9482, 10121, 10531, 10533,  
10672, 10753, 10853, 10869, 11292,  
11748, 12538, 12760, 12970, 13264,  
13291, 14582, 14635, 16745, 18553,  
18852, 18956, 19238, 19257, 19637,  
20485, 20599, 20738, 20782, 21035,  
21300, 21302, 21804, 22907, 23403,  
23479, 24059, 24133, 24147, 24365,  
24382, 24417, 24453, 24468, 24485,  
26048, 29497, 29507, 29516, 29534,  
29552, 29559, 29571, 29576, 29589,  
29611, 29643, 29651, 29667, 29674,  
29723, 29739, 29756, 29764, 29828,  
29842, 30179, 31859, 31892, 32548,  
32560, 32622, 32673, 33742, 33755,  
33786, 34069, 35687, 37390, 37396  
\cs\_set\_protected:Npx .. 16, 147,  
1491, 1503, 1961, 1962, 9654, 21038  
\cs\_set\_protected\_nopar:Nn .....  
. .... 18, 2055, 2120  
\cs\_set\_protected\_nopar:Npn .....  
. .... 16, 377, 1491, 1497, 1955, 1955  
\cs\_set\_protected\_nopar:Npx .....  
. .... 16, 1491, 1499, 1955, 1956  
\cs\_show:N .....  
. 20, 27, 383, 2197, 2197, 2198, 2199

```

\cs_split_function:N ..... 22,
1609, 1630, 1745, 1746, 1803, 1805,
2027, 2068, 2914, 3217, 16285, 16306
\cs_to_str:N ..... 6,
21, 108, 121, 371, 372, 404, 730,
1404, 1794, 1794, 1809, 2712, 2878,
4506, 6007, 6143, 10536, 13958,
13959, 13960, 13961, 13962, 13963,
13964, 13965, 13966, 13967, 13968,
13969, 16413, 16435, 22905, 30507,
32532, 37174, 37260, 37265, 37273
\cs_undefine:N ..... 20, 841, 940, 1983,
1983, 1985, 9541, 9542, 9543, 10219,
10440, 11712, 11713, 12997, 13251,
28754, 29038, 29098, 29654, 29665
cs internal commands:
\__cs_count_signature:N ..... 370, 2026, 2026, 2038, 2039
\__cs_count_signature:n ..... 2026, 2027, 2028
\__cs_count_signature:nnN ..... 2026, 2029, 2030
\__cs_generate_from_signature:n .. 2077, 2091
\__cs_generate_from_signature:nnn .. 2059, 2063
\__cs_generate_from_signature:nnNNn .. 2067, 2072
\__cs_generate_internal_c:NN .. 3180
\__cs_generate_internal_end:w .. 3163, 3197
\__cs_generate_internal_long:nnnNNn .. 3201, 3205
\__cs_generate_internal_long:w .. 3164, 3199
\__cs_generate_internal_loop:nwnnw .. 3161,
3167, 3179, 3181, 3183, 3185, 3188
\__cs_generate_internal_N:NN .. 3178
\__cs_generate_internal_n:NN .. 3182
\__cs_generate_internal_one_-
go>NNn ..... 411, 3136, 3159
\__cs_generate_internal_other:NN .. 3172, 3186
\__cs_generate_internal_test:Nw .. 3121, 3143, 3147
\__cs_generate_internal_test_-
aux:w .. 3123, 3139, 3144, 3150, 3153
\__cs_generate_internal_variant:n ...
415, 3086, 3091, 3091, 3271, 3277
\__cs_generate_internal_variant:NNn ..
411, 3111, 3115
\__cs_generate_internal_variant:wwnNwn .. 3093, 3106
\__cs_generate_internal_variant_-
loop:n .. 3091, 3131, 3202, 3207, 3210
\__cs_generate_internal_x:NN .. 3184
\__cs_generate_variant:N .. 2910, 2923, 2923
\__cs_generate_variant:n ..... 3212
\__cs_generate_variant:nnNN .. 2913, 2946, 2946
\__cs_generate_variant:nnNnn .. 3212, 3216, 3220
\__cs_generate_variant:Nnnw .. 2953, 2955, 2955, 2973
\__cs_generate_variant:w .. 3212, 3227, 3231, 3248
\__cs_generate_variant:ww .. 2923, 2929, 2939
\__cs_generate_variant:wwNN .. 407-409, 2962, 3079, 3079
\__cs_generate_variant:wwNw .. 2923, 2941, 2943
\__cs_generate_variant_F_-
form:nnn ..... 3212, 3254
\__cs_generate_variant_loop:nNw .. 407, 408, 2963, 2975, 2975, 2994
\__cs_generate_variant_loop_-
base:N .. 2975, 2980, 2983, 2996
\__cs_generate_variant_loop_-
end:nwwwNNnn .. 407-409, 2965, 2975, 3021
\__cs_generate_variant_loop_-
invalid>NNwNNnn .. 408, 2975, 2987, 3042
\__cs_generate_variant_loop_-
long:wNNnn .. 408, 2968, 2975, 3029
\__cs_generate_variant_loop_-
same:w .. 408, 2975, 2978, 3018
\__cs_generate_variant_loop_-
special>NNwNNnn .. 2975, 2985, 3057, 3074
\__cs_generate_variant_p_-
form:nnn ..... 3212, 3250
\__cs_generate_variant_same:N .. 408, 3020, 3068, 3068
\__cs_generate_variant_T_-
form:nmm ..... 3212, 3252
\__cs_generate_variant_TF_-
form:nnn ..... 3212, 3256
\__cs_get_function_name:N ..... 370
\__cs_get_function_signature:N .. 371
\__cs_parm_from_arg_count_-
test:nnTF ..... 1995, 1997, 2016
\__cs_split_function_auxi:w .. 1803, 1808, 1812

```

\\_\_cs\_split\_function\_auxii:w . . . . .  
     . . . . . 1803, 1814, 1815  
 \\_\_cs\_tmp:w . . . . . 371,  
     405, 411, 415, 1803, 1818, 1925,  
     1925, 1933, 1934, 1935, 1936, 1937,  
     1938, 1939, 1940, 1941, 1943, 1944,  
     1945, 1946, 1947, 1948, 1949, 1950,  
     1951, 1952, 1953, 1954, 1955, 1956,  
     1957, 1958, 1959, 1960, 1961, 1962,  
     1963, 1964, 1965, 1966, 2055, 2076,  
     2078, 2081, 2096, 2097, 2098, 2099,  
     2100, 2101, 2102, 2103, 2104, 2105,  
     2106, 2107, 2108, 2109, 2110, 2111,  
     2112, 2113, 2114, 2115, 2116, 2117,  
     2118, 2119, 2120, 2128, 2129, 2130,  
     2131, 2132, 2133, 2134, 2135, 2136,  
     2137, 2138, 2139, 2140, 2141, 2142,  
     2143, 2144, 2145, 2146, 2147, 2148,  
     2149, 2150, 2151, 2927, 2945, 3087,  
     3095, 3113, 3158, 3274, 3281, 3282,  
     3283, 3284, 3285, 3286, 3287, 3288,  
     3289, 3290, 3291, 3292, 3293, 3294,  
     3295, 3296, 3297, 3298, 3299, 3300,  
     3301, 3302, 3303, 3304, 3305, 3306,  
     3307, 3308, 3309, 3310, 3311, 3312,  
     3313, 3314, 3315, 3316, 3317, 3318,  
     3319, 3320, 3321, 3322, 3323, 3324  
 \\_\_cs\_to\_str:N . . . . .  
     . . . . . 371, 1794, 1798, 1800, 1801  
 \\_\_cs\_to\_str:w . 371, 1794, 1797, 1801  
 \\_\_cs\_use\_i\_delimit\_by\_s\_stop:nw  
     . . . . . 2904, 2905, 3225  
 \\_\_cs\_use\_none\_delimit\_by\_q\_-  
     recursion\_stop:w . . . . .  
     . . . . . 2904, 2906, 2951, 2958, 3238  
 \\_\_cs\_use\_none\_delimit\_by\_s\_-  
     stop:w . . . . . 2904, 2904, 3229  
 csc . . . . . 255  
 cscd . . . . . 255  
 \csname . . . . . 645, 4, 21, 25, 30,  
     34, 47, 70, 71, 73, 82, 107, 111, 130, 224  
 \csstring . . . . . 805  
 \currentcjktoken . . . . . 1113, 1174  
 \currentgrouplevel . . . . . 516  
 \currentgrouptype . . . . . 517  
 \currentifbranch . . . . . 518  
 \currentiflevel . . . . . 519  
 \currentiftype . . . . . 520  
 \currentspacingmode . . . . . 1114  
 \currentxspacingmode . . . . . 1115

D

\d . . . . . 30140, 32666  
 \date . . . . . 350

\day . . . . . 225, 1322, 9097  
 dd . . . . . 258  
 \deadcycles . . . . . 226  
 debug commands:  
     \debug\_off: . . . . . 350  
     \debug\_off:n . . . . .  
         . . . . . 29, 1408, 1409, 1566, 1571, 1574  
     \debug\_on: . . . . . 350  
     \debug\_on:n . 29, 1408, 1566, 1566, 1569  
     \debug\_resume: . . . . .  
         . . . . . 29, 1312, 1576, 1577, 33661  
     \debug\_suspend: . . . . .  
         . . . . . 29, 1312, 1576, 1576, 33654  
 debug internal commands:  
     \g\_\_debug\_deprecation\_off\_tl . 1578  
     \g\_\_debug\_deprecation\_on\_tl .. 1578  
 \def . . . . . 53, 54, 55, 81, 83, 88,  
     89, 91, 104, 105, 108, 119, 143, 182, 227  
 default commands:  
     .default:n . . . . . 225, 21311  
 \defaulthyphenchar . . . . . 228  
 \defaultskewchar . . . . . 229  
 deg . . . . . 257  
 \delcode . . . . . 230  
 \delimiter . . . . . 231  
 \delimiterfactor . . . . . 232  
 \delimitershortfall . . . . . 233  
 deprecation internal commands:  
     \\_\_deprecation\_just\_error:nnNN 37359  
     \\_\_deprecation\_old:Nnn . . . . .  
         37416, 37421, 37434, 37436, 37438,  
         37440, 37442, 37444, 37446, 37448,  
         37450, 37452, 37454, 37456, 37458,  
         37460, 37462, 37464, 37466, 37468,  
         37470, 37472, 37474, 37476, 37478,  
         37480, 37482, 37484, 37486, 37496,  
         37500, 37504, 37506, 37508, 37510,  
         37512, 37514, 37516, 37528, 37530,  
         37534, 37536, 37538, 37540, 37548,  
         37550, 37552, 37554, 37556, 37558,  
         37560, 37562, 37564, 37566, 37568,  
         37570, 37572, 37574, 37576, 37578,  
         37580, 37582, 37584, 37586, 37588,  
         37590, 37596, 37598, 37612, 37614,  
         37616, 37622, 37624, 37626, 37628  
     \\_\_deprecation\_old\_protected:Nnn  
         . . . . . 37416,  
         37416, 37426, 37428, 37430, 37432,  
         37488, 37490, 37492, 37494, 37498,  
         37502, 37518, 37520, 37522, 37524,  
         37526, 37532, 37542, 37544, 37546,  
         37592, 37594, 37600, 37602, 37604,  
         37606, 37608, 37610, 37618, 37620

```

\__deprecation_patch_aux:Nn . . . . .
. . . . . 37359, 37371, 37392
\__deprecation_patch_aux:nnNNnn . . .
. . . . . 37359, 37360, 37361
\__deprecation_warn_once:nnNnn . . .
. . . . . 37359, 37370, 37373
\detokenize . . . . . 47, 130, 521
\DH . . . . . 30146, 31871, 32582
\dh . . . . . 30146, 31871, 32592
dim commands:
  \dim_abs:n . . . . . 208, 914, 20145, 20145
  \dim_add:Nn . . . . . 208, 20127, 20127, 20134
  \dim_case:nn . . . . . 211, 20225, 20240
  \dim_case:nnn . . . . . 37486
  \dim_case:nnTF . . . . . 211, 20225, 20225, 20230, 20235, 37487
  \dim_compare:n . . . . . 20185
  \dim_compare:nNn . . . . . 20180
  \dim_compare:nNnTF . . . . . 209–212, 244,
. . . . . 20180, 20249, 20285, 20293, 20302,
. . . . . 20308, 20320, 20323, 20334, 34015,
. . . . . 34018, 34023, 34037, 34040, 34045,
. . . . . 34391, 34396, 34406, 34535, 34547,
. . . . . 36754, 36771, 36805, 36819, 36829
  \dim_compare:nTF . . . . . 209, 210,
. . . . . 212, 20185, 20257, 20265, 20274, 20280
  \dim_compare_p:n . . . . . 210, 20185
  \dim_compare_p:nNn . . . . .
. . . . . 209, 20180, 36695,
. . . . . 36696, 36703, 36704, 36707, 36708
  \dim_const:Nn . . . . . 207, 907, 917, 20094,
. . . . . 20094, 20099, 20423, 20424, 22258
  \dim_do_until:nn . . . . .
. . . . . 212, 20255, 20277, 20281
  \dim_do_until:nNnn . . . . .
. . . . . 211, 20283, 20305, 20309
  \dim_do_while:nn . . . . .
. . . . . 212, 20255, 20271, 20275
  \dim_do_while:nNnn . . . . .
. . . . . 211, 20283, 20299, 20303
  \dim_eval:n . . . . .
. . . . . 209, 210, 213, 907, 1289, 1290,
. . . . . 20097, 20228, 20233, 20238, 20243,
. . . . . 20338, 20366, 20366, 20418, 20422,
. . . . . 33718, 33795, 33881, 33899, 33936,
. . . . . 33940, 33941, 33945, 33949, 33950,
. . . . . 33967, 33972, 33978, 33985, 33992,
. . . . . 34157, 34160, 34161, 34168, 34250,
. . . . . 34251, 34258, 34259, 34362, 34369,
. . . . . 34518, 34519, 34783, 34784, 34785
  \dim_gadd:Nn . . . . . 208, 20127, 20129, 20135
  .dim_gset:N . . . . . 225, 21319
  \dim_gset:Nn . . . . .
. . . . . 208, 907, 20115, 20117, 20120
  \dim_gset_eq:NN . . . . .
. . . . . 208, 20121, 20121, 20124, 20126
  \dim_gsub:Nn . . . . . 208, 20127, 20138, 20144
  \dim_gzero:N . . . . .
. . . . . 207, 20100, 20101, 20104, 20108, 20444
  \dim_gzero_new:N . . . . .
. . . . . 207, 20105, 20107, 20110
  \dim_if_exist:N . . . . . 20111, 20113
  \dim_if_exist:NTF . . . . .
. . . . . 208, 20106, 20108, 20111
  \dim_if_exist_p:N . . . . . 208, 20111
  \dim_log:N . . . . . 215, 20419, 20419, 20420
  \dim_log:n . . . . . 215, 20419, 20421
  \dim_max:nn . . . . .
. . . . . 208, 20145, 20152, 34229, 34233
  \dim_min:nn . . . . .
. . . . . 208, 20145, 20160, 34227, 34231, 34244
  \dim_new:N . . . . . 207,
. . . . . 20088, 20088, 20093, 20096, 20106,
. . . . . 20108, 20425, 20426, 20427, 20428,
. . . . . 33225, 33226, 33227, 33228, 33229,
. . . . . 33230, 33231, 33232, 33579, 33603,
. . . . . 33604, 33607, 33608, 33609, 33610,
. . . . . 34103, 34104, 34105, 34106, 34107,
. . . . . 34265, 34266, 34607, 34609, 34610
  \dim_ratio:nn . . . . .
. . . . . 209, 915, 20176, 20176, 20412
  .dim_set:N . . . . . 225, 21319
  \dim_set:Nn . . . . . 208, 20115, 20115, 20119,
. . . . . 33252, 33253, 33254, 33286, 33297,
. . . . . 33381, 33382, 33383, 33398, 33496,
. . . . . 33497, 33498, 33500, 33502, 33504,
. . . . . 33708, 33784, 34017, 34021, 34039,
. . . . . 34043, 34074, 34088, 34163, 34198,
. . . . . 34206, 34217, 34218, 34219, 34220,
. . . . . 34226, 34228, 34230, 34232, 34237,
. . . . . 34243, 34326, 34328, 34330, 34338,
. . . . . 34340, 34394, 34469, 34470, 34472,
. . . . . 34474, 34492, 34493, 34608, 34702,
. . . . . 34703, 34749, 34750, 34751, 34753
  \dim_set_eq:NN . . . . .
. . . . . 208, 20121, 20121, 20123, 33733, 33734
  \dim_show:N . . . . . 214, 20415, 20415, 20416
  \dim_show:n . . . . . 215, 916, 20417, 20417
  \dim_sign:n . . . . . 213, 20368, 20368
  \dim_step_function:nnN . . . . .
. . . . . 212, 913, 20311, 20311, 20363
  \dim_step_inline:nnnn . . . . .
. . . . . 212, 20341, 20341
  \dim_step_variable:nnNn . . . . .
. . . . . 213, 20341, 20348
  \dim_sub:Nn . . . . . 208, 20127, 20136, 20143
  \dim_to_decimal:n . . . . .
. . . . . 213, 20388, 20388, 20404, 20409

```

```

\dim_to_decimal_in_bp:n ..... 214, 20403, 20403
\dim_to_decimal_in_sp:n 214, 1022, 20405, 20405, 23541, 23578, 24172
\dim_to_decimal_in_unit:nn ..... 214, 20407, 20407
\dim_to_fp:n ..... 214, 1022, 1041, 20415, 28373, 28373, 33290, 33291, 33301, 33302, 33370, 33373, 33374, 33399, 33414, 33415, 33434, 33435, 33453, 33470, 33473, 33474, 34028, 34029, 34030, 34050, 34051, 34052, 34062, 34063, 34079, 34080, 34081, 34082, 34092, 34093, 34202, 34203, 34210, 34211, 34284, 34287, 34288, 34339, 34341
\dim_until_do:nn ..... 212, 20255, 20263, 20268
\dim_until_do:nNnn ..... 211, 20283, 20291, 20296
\dim_use:N ..... 213, 1289, 20148, 20154, 20155, 20156, 20162, 20163, 20164, 20188, 20207, 20367, 20371, 20386, 20386, 20387, 20391, 20498, 20499, 20574, 20575, 34165, 34169, 34176, 34182, 34191, 34192, 34193, 34347, 34354, 34500
\dim_while_do:nn ..... 212, 20255, 20255, 20260
\dim_while_do:nNnn ..... 212, 20283, 20283, 20288
\dim_zero:N ..... 207, 20100, 20100, 20103, 20106, 20443, 33255, 33384, 33499, 34008, 34009
\dim_zero_new:N ..... 207, 20105, 20105, 20109
\c_max_dim ..... 215, 218, 970, 20423, 20517, 22285, 22327, 22335, 34217, 34218, 34219, 34220, 34237
\g_tmpa_dim ..... 215, 20425
\l_tmpa_dim ..... 215, 20425
\g_tmpb_dim ..... 215, 20425
\l_tmpb_dim ..... 215, 20425
\c_zero_dim 215, 20320, 20323, 20376, 20423, 20516, 22352, 33111, 33135, 34015, 34018, 34023, 34037, 34040, 34045, 34391, 34396, 34406, 36758, 36769, 36775, 36787, 36805, 36809, 36817, 36819, 36823, 36829, 36839
dim internal commands:
  \__dim_abs:N .... 20145, 20147, 20150
  \__dim_case:nnTF ..... 20225, 20228, 20233, 20238, 20243, 20245
  \__dim_case:nw ..... 20225, 20246, 20247, 20251
  \__dim_case_end:nw 20225, 20250, 20253
  \__dim_compare:w . 20185, 20187, 20190
  \__dim_compare:wNN ..... 909, 20185, 20193, 20196, 20206
  \__dim_compare_!w ..... 20185
  \__dim_compare_<:w ..... 20185
  \__dim_compare_=:w ..... 20185
  \__dim_compare_>:w ..... 20185
  \__dim_compare_end:w .. 20193, 20217
  \__dim_compare_error: ..... 909, 20185, 20188, 20190, 20219, 20223
  \__dim_eval:w .. 915, 20082, 20083, 20116, 20118, 20128, 20132, 20137, 20141, 20148, 20154, 20155, 20156, 20162, 20163, 20164, 20179, 20182, 20188, 20207, 20212, 20314, 20315, 20316, 20367, 20371, 20391, 20406
  \__dim_eval_end: ... 20082, 20084, 20116, 20118, 20128, 20132, 20137, 20141, 20148, 20158, 20166, 20179, 20182, 20367, 20371, 20391, 20406
  \__dim_maxmin:wwN ..... 20145, 20154, 20162, 20168
  \__dim_ratio:n .. 20176, 20177, 20178
  \__dim_sign:Nw .. 20368, 20370, 20374
  \__dim_step:NnnnN ..... 20311, 20321, 20328, 20332, 20337
  \__dim_step:NNnnn ..... 20341, 20344, 20351, 20360
  \__dim_step:wwwN .. 20311, 20313, 20318
  \__dim_to_decimal:w ..... 20388, 20391, 20395
  \__dim_use_none_delimit_by_s_- stop:w ..... 20087, 20087, 20203
\dimen ..... 234, 19299
\dimendef ..... 235
\dimexpr ..... 522
\directlua ..... 6, 38, 40, 808
\disablecjktoken ..... 1175
\discretionary ..... 236
\discretionaryligaturemode ..... 806
\disinhibitglue ..... 1116
\displayindent ..... 237
\displaylimits ..... 238
\displaystyle ..... 239
\displaywidowpenalties ..... 523
\displaywidowpenalty ..... 240
\displaywidth ..... 241
\divide ..... 242
\DJ ..... 30147, 31872, 32583
\dj ..... 30147, 31872, 32593
\do ..... 1228

```

\doublehyphendemerits . . . . .	243	12444, 12454, 12469, 12478, 12488,
\dp . . . . .	244	12504, 12518, 12545, 12562, 12577,
\draftmode . . . . .	914	12856, 12874, 12892, 12900, 12910,
draw commands:		12926, 12949, 12960, 12966, 13112,
\draw_begin: . . . . .	295	13124, 13173, 13176, 13179, 13382,
\draw_end: . . . . .	295	13387, 13392, 13399, 13404, 13654,
\dtou . . . . .	1117	13710, 13713, 13716, 13728, 13743,
\dump . . . . .	245	13882, 14057, 14065, 14073, 14220,
\dviextension . . . . .	809	14271, 14272, 14276, 14281, 14304,
\dvifeedback . . . . .	810	14357, 14457, 14708, 14738, 14741,
\dvivariable . . . . .	811	14771, 14774, 14791, 14794, 14897,
<b>E</b>		
\edef . . . . .	94, 117, 246	14902, 14920, 14939, 14942, 14991,
\efcode . . . . .	685	14996, 14999, 15114, 15126, 15135,
\c_eight . . . . .	37454	15257, 15262, 16190, 16228, 16236,
\elapsedtime . . . . .	774	16247, 16257, 16276, 16300, 16304,
\c_eleven . . . . .	37460	16346, 16405, 16422, 16687, 16765,
\else . . . . .	5, 26, 28, 35, 71, 72, 75, 76, 247	16774, 17201, 17212, 17233, 17249,
else commands:		17252, 17273, 17313, 17412, 17439,
\else: . . . . .		17477, 17485, 17786, 17819, 17870,
27, 64, 93, 168, 169, 221, 284,		17987, 18012, 18038, 18047, 18217,
364, 366, 372, 406, 579, 701, 1067,		18232, 18254, 18268, 18886, 18892,
1429, 1432, 1474, 1660, 1668, 1694,		18895, 18921, 18927, 19040, 19043,
1823, 1826, 1835, 1841, 1851, 1854,		19046, 19049, 19052, 19055, 19058,
1863, 1869, 1989, 2011, 2020, 2034,		19132, 19137, 19142, 19147, 19154,
2093, 2094, 2155, 2338, 2618, 2770,		19161, 19166, 19171, 19176, 19181,
2798, 2813, 2821, 2858, 2928, 2979,		19186, 19191, 19196, 19201, 19223,
2980, 2982, 2986, 2998, 2999, 3000,		19229, 19232, 19268, 19271, 19406,
3001, 3002, 3003, 3004, 3005, 3006,		19415, 19423, 19432, 19508, 19533,
3070, 3071, 3073, 3122, 3152, 3239,		19537, 19547, 19585, 19599, 19608,
3374, 3375, 3842, 3845, 3848, 3858,		19618, 19652, 20151, 20172, 20183,
3873, 3900, 3915, 3942, 3958, 3992,		20193, 20218, 20378, 20381, 22290,
4000, 4002, 4004, 4006, 4008, 4010,		22582, 22599, 22600, 22615, 22625,
4012, 4014, 4036, 4057, 4061, 4137,		22720, 22796, 22858, 22861, 22875,
4141, 4250, 4269, 4287, 4320, 4322,		22893, 22897, 23137, 23150, 23170,
4346, 4397, 4408, 4515, 4612, 4613,		23198, 23199, 23221, 23242, 23265,
4618, 4619, 4636, 4643, 4843, 4853,		23266, 23299, 23316, 23334, 23369,
4903, 4912, 4924, 4925, 4927, 4929,		23373, 23409, 23426, 23432, 23436,
4932, 4933, 4936, 4937, 4946, 4948,		23440, 23599, 23632, 23640, 23673,
4950, 4953, 4954, 4956, 4992, 4995,		23677, 23689, 23699, 23709, 23740,
5016, 5019, 5027, 5035, 5038, 5047,		23753, 23788, 23798, 23817, 23830,
5050, 5059, 5067, 5070, 5080, 5200,		23843, 23847, 23858, 23881, 23898,
5314, 5358, 5362, 5365, 5376, 5381,		23910, 23924, 23937, 23941, 23949,
5471, 5617, 5630, 5719, 5748, 5787,		23951, 23961, 23972, 23988, 24002,
5805, 5918, 5974, 6008, 6038, 6463,		24008, 24011, 24018, 24040, 24070,
6481, 6500, 6534, 6587, 6634, 6638,		24093, 24121, 24124, 24298, 24302,
6645, 6666, 6677, 6824, 6937, 7047,		24309, 24328, 24340, 24344, 24351,
7091, 7094, 7214, 7225, 7234, 7262,		24373, 24390, 24396, 24428, 24460,
7274, 7300, 7317, 7325, 7586, 7840,		24476, 24496, 24537, 24552, 24585,
8119, 8130, 8518, 8561, 8582, 8604,		24587, 24593, 24608, 24661, 24826,
8622, 8638, 8648, 8664, 8674, 8766,		24842, 24853, 24891, 24894, 24897,
8768, 8770, 8772, 10291, 10294,		24900, 24931, 24940, 24949, 24952,
10297, 11339, 11348, 11359, 12055,		25123, 25136, 25139, 25146, 25164,
		25188, 25189, 25204, 25214, 25263,
		25266, 25275, 25287, 25298, 25312,

25325, 25365, 25399, 25419, 25456,	89, 257
25474, 25477, 25483, 25497, 25532,	258
25550, 25553, 25556, 25559, 25620,	259
25693, 25763, 25764, 25773, 25808,	<u>11845</u>
25891, 25895, 25899, 25961, 25996,	4461
26011, 26276, 26305, 26309, 26469,	812
26478, 26532, 26543, 26559, 26567,	813
26626, 26706, 26717, 26722, 26756,	526
26769, 26781, 26787, 26908, 26916,	527
26955, 26962, 26984, 27011, 27026,	814
27030, 27052, 27083, 27086, 27111,	815
27114, 27155, 27163, 27174, 27177,	1120
27292, 27307, 27322, 27337, 27352,	260
27367, 27388, 27433, 27739, 27777,	261
27778, 27787, 27831, 27886, 27887,	528
27888, 27992, 28014, 28029, 28047,	262
28095, 28111, 28317, 28384, 28389,	45, 46, 263
28553, 28589, 28602, 28632, 28636,	264
28644, 28671, 28697, 28705, 28722,	265
28725, 28774, 28778, 28830, 28889,	266
28901, 29400, 29411, 29431, 29733,	258
29892, 29896, 29907, 29928, 29932,	816
29941, 29942, 29943, 29944, 29945,	267
29946, 29947, 29948, 29949, 29960,	253
29974, 29977, 29980, 29983, 29986,	exp commands:
29989, 29992, 30088, 30093, 30099,	\exp:w 40, 41, 364, 371, 389, 390, 397,
30116, 30120, 30589, 31346, 31350,	398, 560, 562–564, 581, 638, 704,
31354, 31357, 31361, 31375, 31378,	722, 845, 1012, 1014, 1015, 1018,
31381, 31384, 31387, 31390, 31409,	1019, 1037, 1042, 1402, 1451, 1587,
31412, 31415, 31418, 31421, 31438,	1589, 2310, 2323, 2329, 2377, 2381,
31441, 31444, 31447, 31463, 31466,	2385, 2390, 2396, 2402, 2418, 2430,
31469, 32986, 32988, 32994, 36847	2436, 2442, 2447, 2449, 2456, 2463,
\em .....	2494, 2499, 2508, 2513, 2522, 2524,
em .....	2532, 2539, 2545, 2553, 2562, 2569,
\emergencystretch .....	2583, 2603, 2607, 2612, 2614, 2651,
\emph .....	2833, 3192, 8039, 8047, 8085, 8124,
\enablecjktoken .....	8132, 8140, 8572, 8719, 8776, 10718,
\end .....	12343, 12574, 12583, 12588, 12593,
245, 249, 25020, 30159, 30169, 31251, 32527	12598, 13036, 13117, 13424, 13429,
\endcsname .....	13434, 13439, 13459, 13464, 13469,
645, 4, 21, 25, 30, 34, 47, 70, 71, 73, 82, 107, 111, 130, 250	13474, 13637, 13646, 13701, 17445,
\endgroup .....	17450, 17455, 17460, 18078, 18086,
19, 20, 24, 29, 33, 53, 88, 92, 98, 115, 251	18146, 18454, 18463, 18873, 19438,
\endinput .....	19440, 19442, 19444, 19446, 19448,
\endL .....	19450, 19452, 19454, 19456, 19458,
\endlinechar .....	19460, 19527, 20192, 20227, 20232,
\endR .....	20237, 20242, 22628, 22743, 22747,
\enquote .....	23113, 23239, 23240, 23241, 23242,
\ensuremath .....	23361, 23379, 23408, 23452, 23464,
\epTeXinputencoding .....	23469, 23477, 23485, 23506, 23512,
\epTeXversion .....	23584, 23597, 23598, 23607, 23620,
\eqno .....	23638, 23639, 23659, 23672, 23676,
\errhelp .....	23698, 23726, 23739, 23752, 23776,
\errmessage .....	23787, 23797, 23816, 23829, 23842,

23845, 23857, 23880, 23909, 23923,  
23940, 23960, 23971, 23977, 23987,  
24029, 24036, 24067, 24082, 24090,  
24107, 24123, 24127, 24136, 24173,  
24182, 24191, 24196, 24198, 24209,  
24211, 24226, 24229, 24236, 24247,  
24331, 24377, 24395, 24398, 24412,  
24425, 24475, 24493, 24564, 24576,  
24605, 24607, 24611, 24613, 24671,  
24681, 24691, 24703, 24833, 24850,  
24860, 25015, 25016, 25017, 25208,  
25211, 25219, 25229, 25237, 26249,  
26780, 26802, 26957, 27133, 27409,  
28152, 28167, 28184, 28221, 28238,  
28280, 28299, 28312, 28344, 28359,  
28370, 28492, 28539, 28579, 28615,  
28795, 28895, 30214, 30253, 30574,  
30629, 30696, 30710, 30722, 30847,  
32280, 36885, 36887, 36889, 36891,  
36893, 36895, 36897, 36899, 37189,  
37194, 37199, 37204, 37220, 37238  
\exp\_after:wN 38, 40, 41, 194, 364,  
368, 387, 390, 434, 451, 541, 560,  
562, 563, 607, 697, 710, 833, 858,  
871, 899, 987, 1011, 1012, 1014,  
1015, 1081, 1082, 1146, 1402, 1448,  
1448, 1466, 1468, 1473, 1475, 1587,  
1589, 1651, 1675, 1693, 1695, 1757,  
1762, 1769, 1798, 1802, 1807, 1818,  
1834, 1836, 1839, 1862, 1864, 1867,  
1988, 1990, 1999, 2019, 2021, 2060,  
2125, 2222, 2242, 2251, 2260, 2272,  
2282, 2288, 2295, 2297, 2309, 2310,  
2322, 2323, 2328, 2329, 2334, 2339,  
2341, 2344, 2353, 2355, 2358, 2359,  
2360, 2363, 2365, 2367, 2371, 2376,  
2381, 2384, 2389, 2394, 2395, 2396,  
2400, 2401, 2402, 2408, 2409, 2416,  
2417, 2418, 2422, 2423, 2424, 2428,  
2429, 2430, 2434, 2435, 2436, 2440,  
2441, 2442, 2446, 2447, 2448, 2449,  
2453, 2454, 2455, 2456, 2460, 2461,  
2462, 2463, 2467, 2468, 2469, 2474,  
2475, 2476, 2477, 2481, 2482, 2483,  
2484, 2490, 2493, 2494, 2498, 2499,  
2512, 2513, 2520, 2522, 2524, 2528,  
2532, 2534, 2537, 2538, 2543, 2544,  
2548, 2551, 2552, 2556, 2559, 2560,  
2561, 2566, 2567, 2568, 2576, 2579,  
2580, 2581, 2582, 2587, 2589, 2591,  
2592, 2603, 2606, 2611, 2636, 2637,  
2638, 2649, 2650, 2662, 2663, 2664,  
2669, 2677, 2678, 2679, 2680, 2681,  
2682, 2705, 2706, 2707, 2708, 2768,  
2769, 2771, 2793, 2798, 2799, 2811,  
2812, 2814, 2818, 2819, 2820, 2823,  
2825, 2828, 2832, 2833, 2834, 2839,  
2840, 2851, 2857, 2859, 2863, 2864,  
2867, 2868, 2878, 2925, 2929, 2951,  
2958, 2978, 3121, 3123, 3150, 3151,  
3153, 3190, 3192, 3209, 3227, 3238,  
3366, 3452, 3453, 3495, 3514, 3515,  
3530, 3531, 3532, 3776, 3782, 3784,  
3795, 3855, 3856, 3857, 3858, 3864,  
3865, 3881, 3899, 3901, 3907, 3908,  
3939, 3941, 3943, 3973, 3988, 3990,  
3991, 3993, 4022, 4032, 4040, 4050,  
4060, 4062, 4064, 4090, 4127, 4136,  
4139, 4140, 4142, 4143, 4151, 4152,  
4166, 4204, 4241, 4245, 4246, 4247,  
4249, 4251, 4261, 4265, 4267, 4268,  
4271, 4272, 4284, 4286, 4288, 4309,  
4313, 4345, 4347, 4364, 4365, 4366,  
4400, 4443, 4480, 4493, 4494, 4499,  
4500, 4503, 4506, 4514, 4516, 4573,  
4580, 4587, 4591, 4598, 4604, 4642,  
4644, 4653, 4775, 4778, 4819, 4837,  
4842, 4844, 4845, 4852, 4855, 4856,  
4862, 4874, 4886, 4905, 4914, 5026,  
5028, 5034, 5037, 5039, 5046, 5049,  
5051, 5058, 5060, 5066, 5069, 5072,  
5166, 5408, 5472, 5484, 5616, 5619,  
5629, 5631, 5813, 5820, 5828, 5913,  
6007, 6039, 6426, 6716, 6722, 6728,  
6839, 6863, 6894, 6925, 6951, 6989,  
7039, 7040, 7048, 7051, 7169, 7198,  
7260, 7261, 7264, 7265, 7273, 7275,  
7276, 7299, 7302, 7379, 7742, 7765,  
7766, 7777, 7801, 7802, 7809, 7829,  
7940, 7941, 8039, 8043, 8044, 8045,  
8085, 8124, 8137, 8138, 8139, 8570,  
8589, 8594, 8598, 8720, 8981, 8982,  
8983, 9084, 9426, 9630, 9631, 10126,  
10127, 10213, 10375, 10393, 10434,  
10648, 10651, 10701, 10710, 10713,  
10716, 10717, 10719, 10759, 10825,  
10856, 10877, 10889, 10917, 10925,  
11014, 11015, 11016, 11437, 11541,  
12052, 12122, 12123, 12130, 12131,  
12147, 12153, 12167, 12172, 12178,  
12179, 12183, 12187, 12192, 12198,  
12199, 12203, 12213, 12217, 12223,  
12224, 12228, 12230, 12234, 12240,  
12241, 12245, 12265, 12288, 12289,  
12290, 12291, 12292, 12351, 12352,  
12353, 12404, 12414, 12419, 12462,  
12500, 12514, 12558, 12560, 12690,  
12743, 12748, 12810, 12821, 12828,

12841, 12844, 12889, 12899, 12923,  
12933, 12934, 12935, 12938, 12942,  
12943, 12967, 13034, 13113, 13115,  
13116, 13117, 13122, 13123, 13125,  
13197, 13214, 13215, 13489, 13490,  
13502, 13567, 13590, 13624, 13625,  
13636, 13637, 13645, 13653, 13655,  
13662, 13667, 13685, 13686, 13687,  
13699, 13700, 13727, 13729, 13735,  
13741, 13755, 13775, 13786, 13802,  
13810, 13818, 13825, 13832, 13844,  
14031, 14047, 14066, 14075, 14096,  
14097, 14102, 14103, 14128, 14129,  
14144, 14145, 14193, 14198, 14263,  
14592, 14628, 14630, 14645, 14651,  
14815, 14817, 14884, 14903, 14904,  
14918, 14919, 14946, 14947, 15062,  
15084, 15097, 15098, 15125, 15220,  
15241, 15250, 16183, 16189, 16191,  
16215, 16266, 16267, 16346, 16380,  
16421, 16429, 16440, 16583, 16585,  
16597, 16605, 16749, 16783, 16795,  
16808, 16809, 16810, 16832, 16833,  
16878, 16913, 16914, 16915, 17015,  
17016, 17018, 17019, 17027, 17028,  
17055, 17056, 17082, 17083, 17086,  
17153, 17193, 17207, 17212, 17215,  
17216, 17223, 17224, 17240, 17241,  
17262, 17263, 17272, 17384, 17389,  
17394, 17417, 17419, 17547, 17548,  
17549, 17574, 17575, 17758, 17786,  
17791, 17819, 17832, 17842, 17869,  
17871, 17872, 17880, 17897, 17941,  
18003, 18010, 18046, 18048, 18055,  
18077, 18085, 18146, 18218, 18233,  
18255, 18269, 18336, 18344, 18350,  
18431, 18453, 18462, 18579, 18580,  
18583, 18584, 18873, 18874, 18910,  
18972, 18973, 18975, 18976, 18977,  
19208, 19227, 19275, 19384, 19413,  
19414, 19416, 19422, 19425, 19507,  
19510, 19526, 19532, 19535, 19538,  
19545, 19546, 19548, 19584, 19586,  
19596, 19597, 19598, 19600, 19606,  
19607, 19609, 19616, 19617, 19619,  
19646, 19651, 19653, 19659, 19785,  
19838, 19864, 19888, 19966, 19989,  
19990, 19991, 20063, 20147, 20151,  
20154, 20155, 20162, 20163, 20187,  
20192, 20203, 20206, 20313, 20314,  
20315, 20370, 20390, 20489, 20903,  
20931, 20944, 21109, 21117, 21129,  
21234, 21610, 21770, 21771, 21850,  
21873, 22308, 22309, 22310, 22328,  
22336, 22360, 22361, 22405, 22412,  
22413, 22424, 22581, 22583, 22584,  
22602, 22603, 22604, 22614, 22616,  
22624, 22626, 22633, 22634, 22635,  
22636, 22637, 22638, 22643, 22644,  
22645, 22646, 22647, 22648, 22649,  
22692, 22705, 22708, 22719, 22721,  
22736, 22740, 22741, 22742, 22745,  
22746, 22810, 22812, 22839, 22843,  
22868, 22872, 22889, 22896, 22898,  
22968, 22976, 22993, 23002, 23048,  
23113, 23182, 23183, 23184, 23244,  
23254, 23273, 23279, 23298, 23300,  
23302, 23313, 23314, 23317, 23328,  
23332, 23339, 23340, 23351, 23352,  
23361, 23368, 23370, 23371, 23379,  
23408, 23426, 23427, 23430, 23431,  
23433, 23434, 23438, 23439, 23441,  
23442, 23451, 23452, 23457, 23463,  
23469, 23477, 23485, 23504, 23505,  
23508, 23509, 23511, 23518, 23519,  
23521, 23539, 23540, 23568, 23571,  
23576, 23577, 23582, 23583, 23585,  
23594, 23595, 23596, 23597, 23600,  
23601, 23602, 23605, 23620, 23637,  
23638, 23648, 23649, 23659, 23671,  
23675, 23688, 23690, 23698, 23708,  
23710, 23716, 23721, 23723, 23725,  
23731, 23732, 23736, 23738, 23750,  
23751, 23773, 23775, 23781, 23784,  
23786, 23790, 23795, 23800, 23801,  
23811, 23812, 23814, 23815, 23818,  
23822, 23827, 23841, 23844, 23856,  
23865, 23872, 23873, 23874, 23875,  
23877, 23879, 23890, 23891, 23892,  
23893, 23895, 23897, 23899, 23900,  
23901, 23907, 23908, 23918, 23922,  
23923, 23925, 23926, 23927, 23932,  
23938, 23939, 23950, 23952, 23959,  
23960, 23962, 23963, 23970, 23976,  
23986, 24050, 24063, 24064, 24065,  
24066, 24080, 24081, 24083, 24088,  
24089, 24104, 24106, 24123, 24127,  
24136, 24170, 24171, 24172, 24178,  
24179, 24180, 24181, 24187, 24188,  
24189, 24190, 24197, 24210, 24218,  
24224, 24225, 24227, 24228, 24234,  
24235, 24237, 24263, 24276, 24296,  
24297, 24299, 24300, 24307, 24308,  
24310, 24313, 24325, 24326, 24327,  
24329, 24330, 24331, 24338, 24339,  
24341, 24342, 24349, 24350, 24352,  
24355, 24370, 24371, 24372, 24375,  
24376, 24377, 24387, 24388, 24389,

24392, 24393, 24394, 24397, 24401,  
24410, 24411, 24422, 24423, 24424,  
24427, 24429, 24430, 24431, 24458,  
24459, 24461, 24462, 24463, 24473,  
24474, 24475, 24477, 24478, 24479,  
24491, 24492, 24495, 24497, 24498,  
24499, 24519, 24520, 24521, 24522,  
24523, 24524, 24525, 24535, 24536,  
24538, 24539, 24540, 24546, 24557,  
24558, 24559, 24560, 24561, 24562,  
24563, 24564, 24569, 24570, 24571,  
24572, 24573, 24574, 24575, 24591,  
24592, 24594, 24595, 24602, 24603,  
24604, 24609, 24610, 24612, 24628,  
24643, 24652, 24662, 24668, 24669,  
24670, 24675, 24688, 24689, 24690,  
24696, 24788, 24832, 24849, 24859,  
24884, 24885, 24932, 25014, 25122,  
25124, 25163, 25165, 25168, 25203,  
25205, 25207, 25210, 25217, 25218,  
25221, 25222, 25227, 25228, 25235,  
25236, 25271, 25272, 25273, 25275,  
25286, 25311, 25313, 25319, 25320,  
25324, 25327, 25349, 25351, 25364,  
25366, 25372, 25374, 25377, 25383,  
25385, 25387, 25388, 25389, 25391,  
25396, 25398, 25400, 25404, 25407,  
25413, 25414, 25418, 25420, 25421,  
25422, 25430, 25432, 25433, 25440,  
25446, 25453, 25454, 25459, 25460,  
25461, 25462, 25481, 25482, 25483,  
25489, 25490, 25491, 25496, 25498,  
25506, 25508, 25510, 25511, 25513,  
25524, 25526, 25528, 25529, 25534,  
25585, 25586, 25593, 25594, 25596,  
25598, 25600, 25603, 25606, 25608,  
25610, 25619, 25621, 25627, 25629,  
25631, 25632, 25633, 25639, 25641,  
25643, 25644, 25645, 25666, 25667,  
25670, 25678, 25680, 25684, 25685,  
25686, 25687, 25692, 25694, 25701,  
25704, 25707, 25710, 25719, 25722,  
25725, 25728, 25735, 25737, 25743,  
25751, 25753, 25755, 25772, 25774,  
25781, 25783, 25786, 25792, 25794,  
25796, 25797, 25798, 25800, 25814,  
25815, 25818, 25836, 25838, 25840,  
25852, 25855, 25858, 25861, 25864,  
25867, 25870, 25873, 25877, 25889,  
25893, 25897, 25900, 25915, 25921,  
25923, 25925, 25935, 25959, 25962,  
25974, 25976, 25980, 25981, 25982,  
25984, 25985, 25987, 25994, 26002,  
26003, 26009, 26010, 26016, 26019,  
26020, 26021, 26022, 26030, 26072,  
26077, 26079, 26086, 26089, 26092,  
26095, 26098, 26101, 26109, 26110,  
26122, 26130, 26132, 26142, 26144,  
26151, 26160, 26162, 26165, 26168,  
26171, 26174, 26187, 26189, 26197,  
26199, 26207, 26209, 26219, 26222,  
26225, 26232, 26247, 26248, 26265,  
26267, 26268, 26325, 26338, 26340,  
26346, 26359, 26361, 26363, 26387,  
26401, 26403, 26410, 26412, 26453,  
26454, 26455, 26457, 26458, 26459,  
26461, 26462, 26468, 26470, 26471,  
26477, 26479, 26480, 26481, 26482,  
26494, 26500, 26502, 26539, 26546,  
26553, 26573, 26574, 26576, 26578,  
26580, 26593, 26598, 26599, 26600,  
26601, 26602, 26606, 26611, 26613,  
26619, 26625, 26627, 26628, 26634,  
26635, 26636, 26637, 26638, 26639,  
26640, 26641, 26646, 26648, 26650,  
26652, 26654, 26659, 26661, 26663,  
26665, 26667, 26669, 26687, 26691,  
26699, 26700, 26705, 26707, 26716,  
26719, 26720, 26721, 26723, 26724,  
26725, 26733, 26739, 26751, 26754,  
26755, 26757, 26758, 26782, 26783,  
26786, 26788, 26804, 26808, 26809,  
26810, 26826, 26832, 26898, 26899,  
26900, 26907, 26909, 26910, 26915,  
26917, 26918, 26927, 26928, 26930,  
26933, 26936, 26952, 26956, 26957,  
26961, 26963, 26998, 27004, 27005,  
27007, 27009, 27010, 27012, 27013,  
27023, 27024, 27027, 27028, 27029,  
27031, 27032, 27033, 27050, 27051,  
27053, 27054, 27060, 27062, 27065,  
27068, 27071, 27074, 27082, 27085,  
27087, 27090, 27097, 27101, 27109,  
27110, 27113, 27115, 27117, 27122,  
27123, 27129, 27134, 27135, 27143,  
27144, 27145, 27146, 27189, 27211,  
27212, 27215, 27216, 27225, 27226,  
27227, 27231, 27238, 27239, 27240,  
27381, 27382, 27383, 27385, 27403,  
27404, 27405, 27406, 27407, 27408,  
27415, 27424, 27431, 27432, 27656,  
27657, 27663, 27664, 27667, 27672,  
27675, 27678, 27681, 27684, 27687,  
27690, 27693, 27709, 27710, 27720,  
27729, 27737, 27738, 27740, 27741,  
27746, 27747, 27756, 27763, 27772,  
27773, 27786, 27788, 27816, 27817,  
27826, 27829, 27854, 27860, 27861,

27903, 27904, 27906, 27920, 27921,  
 27929, 27940, 27974, 27977, 27987,  
 27988, 27991, 27993, 27999, 28013,  
 28015, 28056, 28059, 28079, 28152,  
 28162, 28166, 28184, 28187, 28208,  
 28209, 28216, 28220, 28238, 28241,  
 28270, 28271, 28277, 28278, 28279,  
 28286, 28294, 28298, 28312, 28315,  
 28331, 28332, 28339, 28343, 28354,  
 28358, 28370, 28375, 28376, 28377,  
 28383, 28385, 28388, 28390, 28452,  
 28481, 28491, 28498, 28503, 28504,  
 28514, 28541, 28552, 28554, 28556,  
 28558, 28563, 28564, 28566, 28577,  
 28578, 28598, 28604, 28605, 28607,  
 28610, 28615, 28621, 28622, 28652,  
 28654, 28657, 28660, 28662, 28670,  
 28672, 28676, 28680, 28685, 28690,  
 28701, 28765, 28766, 28790, 28791,  
 28792, 28793, 28794, 28802, 28813,  
 28819, 28845, 28846, 28847, 28854,  
 28855, 28862, 28863, 28871, 28872,  
 28876, 28877, 28878, 28883, 28888,  
 28894, 28897, 28898, 28899, 28900,  
 28901, 29105, 29106, 29313, 29352,  
 29366, 29383, 29689, 29691, 29835,  
 29895, 29897, 29904, 29905, 29908,  
 29931, 29933, 29939, 29951, 29959,  
 29961, 30086, 30090, 30091, 30092,  
 30094, 30095, 30096, 30100, 30114,  
 30117, 30121, 30212, 30251, 30321,  
 30322, 30365, 30366, 30389, 30439,  
 30459, 30538, 30564, 30573, 30588,  
 30590, 30627, 30694, 30708, 30720,  
 30846, 31914, 32278, 32326, 32327,  
 32388, 32389, 32390, 32397, 32456,  
 32573, 32575, 32648, 32650, 34846,  
 34850, 34868, 34869, 35323, 35376,  
 35471, 35498, 35503, 35509, 35515,  
 35659, 35677, 35841, 36118, 36925,  
 36935, 36940, 36943, 37040, 37050,  
 37189, 37194, 37199, 37204, 37218,  
 37221, 37236, 37238, 37239, 37246,  
 37251, 37253, 37256, 37270, 37290,  
 37291, 37298, 37299, 37340, 37354  
**\exp\_args:cc** . . . . . 34, 1465, 1467, 2352  
**\exp\_args:Nc** . . . . . 32, 34, 381,  
1465, 1465, 1469, 1477, 1699, 1712,  
 1720, 1728, 1923, 1942, 1968, 1973,  
 1980, 2039, 2051, 2124, 2157, 2158,  
 2159, 2160, 2182, 2186, 2352, 2922,  
 4084, 5769, 10131, 10137, 10361,  
 12383, 12643, 13269, 18023, 21932,  
 23262, 23501, 24381, 24405, 24435,  
 24437, 24439, 24441, 24443, 24445,  
 24447, 24449, 24467, 24483, 24484,  
 24503, 24505, 29015, 29016, 29017,  
 29045, 29636, 32524, 33928, 33959,  
 35520, 37174, 37260, 37265, 37273  
**\exp\_args:Ncc** . . . . . 36, 1970,  
 1974, 1982, 2165, 2166, 2167, 2168,  
2352, 2354, 14211, 14393, 16355, 16395  
**\exp\_args:Nccc** . . . . . 36, 2352, 2356  
**\exp\_args:Ncco** . . . . . 36, 2451, 2479  
**\exp\_args:Ncxx** . . . . . 37, 3300  
**\exp\_args:Ncf** . . . . . 36, 2392, 2438  
**\exp\_args:NcNc** . . . . . 36, 2451, 2465  
**\exp\_args:NcNo** . . . . . 36, 2451, 2472  
**\exp\_args:Ncno** . . . . . 37, 3300  
**\exp\_args:NcnV** . . . . . 37, 3300  
**\exp\_args:Ncnx** . . . . . 37, 3300  
**\exp\_args:Nco** . . . . . 36, 392, 2392, 2420  
**\exp\_args:Ncoo** . . . . . 37, 3300  
**\exp\_args:NcV** . . . . . 36, 2392, 2426  
**\exp\_args:Ncv** . . . . . 36, 2392, 2432  
**\exp\_args:NcVV** . . . . . 37, 3300  
**\exp\_args:Ncx** . . . . . 36, 3274  
**\exp\_args:Ne** . . . . . 35, 388,  
 1530, 2305, 2368, 2370, 2374, 2600,  
 10909, 10911, 11033, 11057, 11189,  
 11198, 11217, 11227, 11237, 11250,  
 11453, 13196, 13934, 15284, 15299,  
 18997, 19081, 21930, 29469, 29792,  
 29815, 30401, 30506, 30516, 30630,  
 30814, 31020, 31118, 31598, 31613,  
 31689, 31925, 32281, 32430, 32565,  
 32642, 34932, 34963, 35153, 35160,  
 35290, 35574, 36097, 36217, 36368  
**\exp\_args:Nee** . . . . . 36, 3274,  
 10135, 11328, 11483, 35166, 35766  
**\exp\_args:Neee** . . . . . . 37, 3300, 11203, 34925, 35193, 35276  
**\exp\_args:Nf** . . . . . . 35, 2027, 2380, 2380, 2646, 2712,  
2747, 2761, 4510, 6140, 6142, 6565,  
 8565, 9452, 9453, 10751, 10809,  
 12332, 13059, 13060, 13076, 13094,  
 13102, 13106, 13130, 13136, 13146,  
 13193, 13226, 13614, 13616, 13675,  
 13677, 13693, 14110, 14115, 14448,  
 14476, 15000, 15001, 15165, 15176,  
 16881, 16882, 16898, 17446, 17451,  
 17456, 17461, 17632, 17701, 17703,  
 17721, 17730, 17741, 17750, 17888,  
 17905, 18652, 18666, 18687, 18698,  
 18755, 18825, 18831, 18837, 18843,  
 20228, 20233, 20238, 20243, 22379,  
 25075, 28148, 28714, 28957, 29123,

29178, 29389, 29573, 29578, 29680,  
 29698, 31339, 31368, 31402, 31431,  
 31456, 35095, 36081, 36853, 36855  
`\exp_args:Nff` . . . . . 36, 3274, 13100, 22909, 37001, 37076  
`\exp_args:Nffo` . . . . . 37, 3300  
`\exp_args:Nfo` . . . . . 36, 3274  
`\exp_args:NNc` . . . . . 36,  
 360, 1969, 1972, 1981, 2053, 2161,  
 2162, 2163, 2164, 2199, 2202, 2352,  
 2352, 3192, 10213, 10344, 10345,  
 10434, 17008, 17589, 17600, 20344,  
 20351, 25085, 25092, 28742, 29047  
`\exp_args:Nnc` . . . . . 36, 3274, 32267  
`\exp_args:NNcf` . . . . . 37, 3300  
`\exp_args:NNe` . . . . .  
 . . . 36, 2392, 2406, 2413, 29601, 35909  
`\exp_args:Nne` . . . . .  
 . . . . . 36, 3274, 11039, 11059, 11094  
`\exp_args:NNf` . . . . . 36, 2392,  
 2414, 6837, 10212, 10433, 10635,  
 20337, 22488, 22493, 27375, 27376  
`\exp_args:Nnf` . . . . .  
 . . . . . 36, 3274, 12310, 21928, 29669  
`\exp_args:Nnff` . . . . . 37, 3300  
`\exp_args:Nnnc` . . . . . 37, 3300  
`\exp_args:Nnnf` . . . . . 37, 3300  
`\exp_args:NNNo` . . . . . 36, 2363,  
 2366, 4323, 5121, 6234, 6297, 7645  
`\exp_args:NNno` . . . . . 37, 3300  
`\exp_args:Nnno` . . . . . 37, 3300  
`\exp_args:NNNV` . . . . . 36,  
 2451, 2451, 30198, 35019, 35559, 35649  
`\exp_args:NNNv` . . . . . 36, 2451, 2458, 10347  
`\exp_args:NNnV` . . . . . 37, 3300  
`\exp_args:NNNx` . . . . . 37, 417, 3300, 5129, 5612  
`\exp_args:NNnx` . . . . . 37, 3300  
`\exp_args:Nnnx` . . . . . 37, 3300  
`\exp_args:NNo` . . . . . 30, 36,  
 2363, 2364, 2655, 7119, 8032, 12268,  
 17620, 19779, 19837, 19965, 21768  
`\exp_args:Nno` . . . . .  
 . . . . . 36, 3274, 7139, 8989, 10318,  
 11023, 12295, 12356, 12464, 12473,  
 12824, 12839, 12930, 12964, 13557,  
 16401, 20195, 22953, 22961, 22970,  
 22987, 22995, 23023, 23527, 23531  
`\exp_args:NNoo` . . . . . 37, 3300  
`\exp_args:NNox` . . . . . 37, 3300  
`\exp_args:Nnox` . . . . . 37, 3300  
`\exp_args:NNV` . . . . . 36, 2392, 2392  
`\exp_args:NNv` . . . . . 36, 2392, 2398  
`\exp_args:NnV` . . . . . 36, 3274, 35074  
`\exp_args:Nnv` . . . . . 36, 3274

\exp\_args:Nxo . . . . . 36, [3274](#)  
 \exp\_args:Nxx . . . . . 36, [3274](#)  
 \exp\_args\_generate:n . . . . .  
     306, [3258](#), 3258, 10132  
 \exp\_arssgs:Nn . . . . . 36276  
 \exp\_end: . . . . . 40, 41, [364](#),  
     367, [371](#), [389](#), [390](#), [397](#), [398](#), [560](#),  
     562, [563](#), [583](#), [696](#), [704](#), [722](#), [1012](#),  
     1042, [1401](#), [1402](#), [1452](#), [1700](#), [1713](#),  
     1721, [1729](#), [2341](#), [2350](#), [2614](#), [2644](#),  
     2834, [3192](#), [8039](#), [8044](#), [8094](#), [8124](#),  
     8132, [8138](#), [8751](#), [8754](#), [8755](#), [8756](#),  
     8757, [8758](#), [8759](#), [8760](#), [8761](#), [8762](#),  
     8764, [12359](#), [12613](#), [13006](#), [13125](#),  
     13486, [13670](#), [17472](#), [18073](#), [18900](#),  
     18907, [18910](#), [18916](#), [18924](#), [18968](#),  
     18972, [19475](#), [20254](#), [23244](#), [24203](#),  
     26804, [28541](#), [28543](#), [28615](#), [30233](#),  
     30671, [32299](#), [36918](#), [37180](#), [37209](#)  
 \exp\_end\_continue:f:nw 41, [2614](#), 2622  
 \exp\_end\_continue:f:w . . . . .  
     41, [389](#), [1014](#), [1015](#), [2310](#), [2381](#),  
     2418, [2442](#), [2513](#), [2532](#), [2545](#), [2569](#),  
     2583, [2603](#), [2614](#), [2616](#), [8572](#), [10055](#),  
     10718, [12574](#), [18146](#), [19527](#), [20192](#),  
     22628, [22743](#), [22747](#), [23361](#), [23379](#),  
     23400, [23464](#), [23469](#), [23477](#), [23485](#),  
     23506, [23584](#), [23620](#), [23628](#), [23659](#),  
     24029, [24036](#), [24082](#), [24129](#), [24136](#),  
     24173, [24217](#), [24223](#), [24226](#), [24236](#),  
     24247, [24412](#), [24605](#), [24607](#), [24611](#),  
     24613, [24671](#), [24681](#), [24691](#), [24703](#),  
     24833, [24850](#), [24860](#), [25015](#), [25016](#),  
     25017, [25208](#), [25219](#), [25229](#), [25237](#),  
     26249, [26957](#), [27133](#), [27409](#), [28152](#),  
     28167, [28184](#), [28221](#), [28238](#), [28280](#),  
     28299, [28312](#), [28344](#), [28359](#), [28370](#),  
     28492, [28579](#), [28795](#), [28895](#), [30574](#)  
 \exp\_last\_two\_unbraced:Nnn . . . . .  
     38, [2586](#), [2586](#), [34002](#), [34526](#), [34530](#)  
 \exp\_last\_unbraced:cf . . . . .  
     34889, [34895](#), [34901](#)  
 \exp\_last\_unbraced:Nco . . . . .  
     37, [2520](#), [2547](#), [18473](#)  
 \exp\_last\_unbraced:NcV 37, [2520](#), [2549](#)  
 \exp\_last\_unbraced:Ne . . . . .  
     37, [2520](#), [2527](#), [2530](#), [28423](#)  
 \exp\_last\_unbraced:Nf . . . . .  
     37, [2520](#), [2531](#), [4830](#), [6160](#), [14544](#),  
     14829, [15018](#), [15190](#), [16284](#), [16305](#),  
     16412, [16434](#), [17719](#), [17739](#), [22395](#),  
     22410, [22904](#), [24824](#), [25301](#), [34879](#)  
 \exp\_last\_unbraced:Nfo . . . . .  
     37, [2520](#), [2573](#), [28745](#)  
 \exp\_last\_unbraced:NNf 37, [2520](#), [2541](#)  
 \exp\_last\_unbraced:NNNf . . . . .  
     37, [2520](#), [2564](#), [8499](#)  
 \exp\_last\_unbraced:NNNNf . . . . .  
     37, [2520](#), [2577](#), [8504](#)  
 \exp\_last\_unbraced:NNNNo . . . . .  
     37, [2520](#), [2575](#),  
     2938, [2942](#), [3105](#), [10927](#), [13864](#),  
     14661, [20954](#), [22696](#), [22714](#), [29967](#)  
 \exp\_last\_unbraced:NNNo 37, [2520](#), [2555](#)  
 \exp\_last\_unbraced:NnNo 37, [2520](#), [2574](#)  
 \exp\_last\_unbraced:NNNV 37, [2520](#), [2557](#)  
 \exp\_last\_unbraced:NNo . . . . .  
     37, [2520](#), [2533](#),  
     10550, [13014](#), [30259](#), [32314](#), [34497](#)  
 \exp\_last\_unbraced:Nno . . . . .  
     37, [2520](#), [2571](#), [16966](#), [18502](#), [20029](#)  
 \exp\_last\_unbraced:Nv 37, [2520](#), [2535](#)  
 \exp\_last\_unbraced:Nx 38, [2520](#), [2585](#)  
 \exp\_not:N . . . . . 38,  
     92, [157](#), [258](#), [390](#), [396](#), [401](#), [434](#),  
     443, [451](#), [453](#), [532](#), [537](#), [560](#), [700](#)–  
     702, [870](#), [871](#), [878](#), [888](#), [1020](#), [1448](#),  
     1449, [1655](#), [1744](#), [1747](#), [2059](#), [2060](#),  
     2124, [2125](#), [2222](#), [2288](#), [2334](#), [2487](#),  
     2591, [2591](#), [2669](#), [2673](#), [2715](#), [2768](#),  
     2788, [2798](#), [2811](#), [2915](#), [2917](#), [2918](#),  
     2925, [2926](#), [2927](#), [2928](#), [2929](#), [2930](#),  
     2936, [2962](#), [2971](#), [3026](#), [3027](#), [3087](#),  
     3093, [3095](#), [3101](#), [3162](#), [3179](#), [3181](#),  
     3209, [3843](#), [3846](#), [3999](#), [4000](#), [4001](#),  
     4002, [4003](#), [4004](#), [4005](#), [4006](#), [4007](#),  
     4008, [4009](#), [4010](#), [4011](#), [4012](#), [4013](#),  
     4014, [4229](#), [4238](#), [4239](#), [4241](#), [4247](#),  
     4258, [4262](#), [4274](#), [4279](#), [4300](#), [4304](#),  
     4400, [4402](#), [4404](#), [4410](#), [4412](#), [4454](#),  
     4806, [4808](#), [4810](#), [4812](#), [4814](#), [4816](#),  
     5199, [5201](#), [5401](#), [5403](#), [5414](#), [5418](#),  
     5576, [6249](#), [6956](#), [7371](#), [7384](#), [8118](#),  
     9497, [9499](#), [9501](#), [9503](#), [9508](#), [9509](#),  
     9514, [9516](#), [9518](#), [10237](#), [10238](#),  
     10242, [11008](#), [11011](#), [11012](#), [11014](#),  
     11015, [11016](#), [11017](#), [11020](#), [11021](#),  
     11086, [11088](#), [11408](#), [11409](#), [11410](#),  
     11411, [11412](#), [11415](#), [11416](#), [12407](#),  
     12854, [12872](#), [12898](#), [12905](#), [12916](#),  
     12917, [12989](#), [12992](#), [12993](#), [13297](#),  
     13298, [13514](#), [13515](#), [14438](#), [14439](#),

14466, 14467, 14468, 15306, 15307,  
15308, 15310, 15311, 15313, 16572,  
16574, 16990, 17037, 17605, 17868,  
18406, 18861, 18931, 18933, 18935,  
18939, 18940, 18942, 18944, 18946,  
18947, 18949, 18951, 18953, 18955,  
18963, 18977, 19038, 19041, 19044,  
19047, 19050, 19053, 19056, 19127,  
19131, 19136, 19141, 19146, 19153,  
19160, 19165, 19170, 19175, 19180,  
19185, 19195, 19200, 19205, 19208,  
19209, 19212, 19222, 19227, 19242,  
19261, 19266, 19267, 19268, 19269,  
19270, 19271, 19273, 19275, 19276,  
19277, 19281, 19282, 19285, 19286,  
19378, 19381, 19382, 19384, 19385,  
19389, 19392, 19393, 19395, 19398,  
19518, 19531, 19545, 19558, 19564,  
19595, 19598, 19605, 19606, 19615,  
19616, 19630, 19631, 19642, 19643,  
19920, 19943, 20076, 20356, 20395,  
20968, 20970, 21024, 21025, 21041,  
21098, 21100, 21133, 21134, 21247,  
21248, 21445, 21446, 21447, 21448,  
21449, 21452, 21454, 21456, 21457,  
21496, 21497, 21498, 21499, 21502,  
21504, 21506, 21507, 21538, 21539,  
21540, 21541, 21544, 21546, 21548,  
21549, 21557, 21558, 21559, 21560,  
21561, 21564, 21566, 21568, 21569,  
21945, 21950, 21954, 21957, 21966,  
21967, 22692, 22693, 23425, 23426,  
23521, 23522, 23523, 23524, 23630,  
23670, 23674, 23696, 23789, 23821,  
23906, 23920, 23937, 23948, 23958,  
23995, 23997, 24100, 24101, 24103,  
24104, 24105, 24106, 24107, 24108,  
24111, 24113, 24115, 24294, 24295,  
24336, 24337, 24457, 24472, 25097,  
25254, 27262, 27263, 27264, 27268,  
27269, 27270, 29110, 29326, 29358,  
29889, 29890, 29893, 29904, 29972,  
29975, 29978, 29981, 29984, 29987,  
29990, 30160, 30266, 30267, 30268,  
30270, 30272, 30276, 30280, 30281,  
30479, 30480, 30485, 30486, 30516,  
30588, 30639, 30640, 30644, 30646,  
30730, 30795, 31954, 32574, 32576,  
32649, 32651, 33733, 33734, 34958,  
34959, 34960, 34961, 34962, 34963,  
34964, 35174, 35340, 35341, 35342,  
35343, 35344, 35345, 35580, 35581,  
35585, 35587, 35691, 35692, 35794,  
35800, 35802, 35803, 35909, 35910,  
35911, 35912, 36067, 36068, 36069,  
36206, 36208, 36211, 36246, 36299,  
36858, 36860, 36864, 36866, 36871,  
36873, 36977, 37083, 37336, 37337,  
37379, 37387, 37403, 37406, 37723,  
37726, 37729, 37732, 37735, 37738  
\exp\_not:n . 38, 39, 51, 92, 113–117,  
146, 151, 152, 157, 178–181, 194,  
202, 235, 236, 266, 309, 312, 350,  
428, 433, 434, 440, 443, 451, 452,  
460, 532, 539, 543, 554, 678, 686,  
687, 707, 803, 806, 846, 847, 849,  
853, 863, 929, 1236, 1239, 1402,  
1404, 1409, 1448, 1450, 1656, 1662,  
1664, 1670, 1671, 1749, 1999, 2234,  
2235, 2288, 2301, 2317, 2504, 2517,  
2591, 2592, 2595, 2599, 2602, 2604,  
2609, 2672, 2714, 3031, 3046, 3061,  
3136, 3183, 3206, 3605, 3871, 3985,  
4016, 4259, 4261, 4293, 4305, 4307,  
4313, 4326, 4454, 4531, 5199, 5201,  
5834, 6306, 6473, 6691, 6880, 6945,  
6957, 7035, 7036, 7299, 7302, 7371,  
7379, 7396, 7411, 7452, 7636, 7737,  
7751, 7759, 7787, 7792, 7794, 8085,  
9021, 9053, 9491, 9508, 9656, 10511,  
10528, 11715, 11718, 11720, 12172,  
12192, 12217, 12234, 12408, 12409,  
12410, 12988, 12991, 12995, 13075,  
13335, 13380, 13391, 13535, 16544,  
16546, 16596, 16597, 16604, 16605,  
16621, 16653, 16673, 16676, 16679,  
16788, 16820, 16844, 16897, 16991,  
17047, 17098, 17108, 17606, 18113,  
18155, 18156, 18170, 18172, 18242,  
18307, 18343, 18351, 18371, 18407,  
18596, 18601, 18627, 18630, 18633,  
18665, 18696, 18716, 18964, 19006,  
19033, 19280, 19499, 19519, 19559,  
19565, 19750, 19843, 19921, 19924,  
19925, 19944, 19948, 20076, 20357,  
20745, 20754, 21027, 21041, 21057,  
21098, 21100, 21136, 21249, 21450,  
21458, 21486, 21499, 21500, 21508,  
21528, 21541, 21542, 21550, 21562,  
21570, 21782, 21792, 21818, 21820,  
23516, 24746, 24748, 24750, 24786,  
25098, 25255, 28434, 29908, 29931,  
30159, 30161, 30191, 30403, 30404,  
30541, 30562, 30816, 30817, 31031,  
31042, 31144, 32611, 32615, 32616,  
33850, 35801, 36144, 36934, 37049,  
37053, 37056, 37057, 37090, 37180,  
37181, 37189, 37194, 37199, 37204,

37218, 37236, 37249, 37256, 37386  
`\exp_stop_f:` ..... . . . . .  
 .. 39, 41, 168, 389, 428, 434, 638,  
 706, 802, 818, 978, 991, 1061, 1153,  
 1182, 2307, 2313, 2775, 2778, 2793,  
 2801, 2856, 3365, 3624, 3809, 3818,  
 3819, 3853, 3923, 3956, 3957, 3961,  
 4037, 4053, 4059, 4138, 4301, 4611,  
 4612, 4613, 4619, 4641, 4901, 4921,  
 4922, 4926, 4930, 4931, 4934, 4935,  
 4943, 4944, 4947, 4951, 4952, 4955,  
 5014, 5374, 5379, 5393, 5394, 5407,  
 5469, 5470, 5509, 5786, 6478, 6531,  
 7046, 7050, 7199, 7223, 7258, 7263,  
 7269, 7606, 8774, 8776, 9121, 10213,  
 10434, 10706, 10720, 10732, 10942,  
 13112, 13171, 13177, 13652, 13668,  
 13708, 13714, 13726, 13743, 13879,  
 14055, 14063, 14270, 14271, 14272,  
 14277, 14278, 14302, 14673, 14735,  
 14739, 14769, 14772, 14788, 14792,  
 14813, 14893, 14895, 14915, 14916,  
 14933, 14935, 14989, 14992, 14993,  
 15112, 15117, 15258, 16608, 17195,  
 17209, 17219, 17227, 17411, 17416,  
 17570, 18751, 18753, 18821, 18823,  
 18827, 18829, 18833, 18835, 18839,  
 18841, 18880, 18881, 18882, 18883,  
 18889, 18893, 18906, 18911, 18919,  
 20201, 20372, 22288, 22291, 22420,  
 22497, 22501, 22531, 22736, 22851,  
 22866, 23113, 23139, 23188, 23200,  
 23266, 23407, 23437, 23593, 23636,  
 23687, 23707, 23734, 23748, 23783,  
 23810, 23819, 23838, 23854, 23870,  
 23888, 23949, 23968, 23984, 24218,  
 24306, 24348, 24586, 24590, 24904,  
 24906, 24921, 24938, 24946, 24947,  
 25144, 25264, 25270, 25285, 25322,  
 25395, 25417, 25471, 25472, 25480,  
 25817, 25835, 25979, 25991, 26007,  
 26024, 26302, 26303, 26400, 26493,  
 26528, 26546, 26555, 26557, 26713,  
 26748, 26901, 26951, 26997, 27002,  
 27084, 27151, 27157, 27172, 27184,  
 27222, 27243, 27283, 27298, 27313,  
 27328, 27343, 27358, 27386, 27430,  
 27696, 27706, 27736, 27888, 27890,  
 27939, 28012, 28021, 28036, 28088,  
 28101, 28185, 28239, 28290, 28313,  
 28589, 28590, 28591, 28600, 28610,  
 28628, 28697, 28700, 28703, 28771,  
 28775, 28817, 28889, 28896, 29394,  
 29395, 29401, 29958, 30084, 30085,  
 30089, 30112, 30113, 31344, 31347,  
 31348, 31351, 31352, 31373, 31376,  
 31379, 31382, 31385, 31388, 31407,  
 31410, 31413, 31416, 31419, 31436,  
 31439, 31442, 31445, 31461, 31464,  
 31467, 37336, 37337, 37338, 37339  
`\exp` internal commands:  
`\__exp_arg_last_unbraced:Nnn` .....  
 .. 2488, 2488, 2490, 2493, 2498, 2512  
`\__exp_arg_next:Nnn` .. 2288, 2289, 2295  
`\__exp_arg_next:nnn` ... 389, 2288,  
 2288, 2297, 2305, 2309, 2322, 2328  
`\__exp_e:N` ..... 2630, 2660, 2660  
`\__exp_e:nn` .. 391, 398, 2377, 2508,  
 2626, 2626, 2646, 2651, 2659, 2687,  
 2689, 2734, 2735, 2740, 2807, 2825  
`\__exp_e:Nnn` ... 399, 2660, 2662, 2666  
`\__exp_e_end:nn` .....  
 .. 398, 2626, 2636, 2644, 2759  
`\__exp_e_expandable:Nnn` .....  
 .. 399, 2660, 2677, 2688  
`\__exp_e_group:n` .... 2633, 2647, 2647  
`\__exp_e_if_toks_register:N` .. 2871  
`\__exp_e_if_toks_register:NTF` ..  
 .. 2822, 2871  
`\__exp_e_noexpand:Nnn` .....  
 .. 2680, 2715, 2737, 2737  
`\__exp_e_primitive:Nnn` .....  
 .. 2682, 2690, 2690, 2721  
`\__exp_e_primitive_aux:NNnn` ..  
 .. 2690, 2705, 2710  
`\__exp_e_primitive_aux:NNw` ..  
 .. 2690, 2694, 2698, 2703  
`\__exp_e_primitive_other:NNnn` ..  
 .. 2690, 2723, 2725  
`\__exp_e_primitive_other_-aux:nNNnn` .. 2690, 2727, 2731  
`\__exp_e_protected:Nnn` .....  
 .. 399, 2660, 2678, 2686  
`\__exp_e_put:nn` ..... 399, 401,  
 402, 2647, 2649, 2653, 2740, 2752, 2839  
`\__exp_e_put:nnn` .....  
 .. 403, 2647, 2655, 2658, 2845  
`\__exp_e_space:nn` ... 2637, 2645, 2645  
`\__exp_e_the:N` ..... 2803, 2806, 2809  
`\__exp_e_the:Nnn` .....  
 .. 2681, 2716, 2803, 2803, 2832  
`\__exp_e_the_errhelp:` ... 2871, 2885  
`\__exp_e_the_everycr:` ... 2871, 2886  
`\__exp_e_the_everydisplay:` 2871, 2887  
`\__exp_e_the_everyeof:` ... 2871, 2888  
`\__exp_e_the_everyhbox:` ... 2871, 2889  
`\__exp_e_the_everyjob:` ... 2871, 2890  
`\__exp_e_the_everymath:` ... 2871, 2891

	<b>F</b>
\fact	253
\false	258
\fam	269
\fi	18, 23, 32, 37, 49, 50, 51, 71, 76, 77, 78, 93, 101, 113, 114, 270
fi commands:	
\fi:	27, 64, 70, 93, 168, 169, 194, 221, 284, 364, 366, 368, 371, 372, 398, 434, 455, 456, 553, 554, 557, 583, 646, 682, 687, 722, 724, 727, 824, 833, 871, 903, 904, 991, 1019, 1034, 1067, 1429, 1433, 1476, 1652, 1660, 1668, 1676, 1696, 1701, 1714, 1722, 1730, 1732, 1733, 1734, 1735, 1758, 1763, 1770, 1797, 1802, 1828, 1829, 1837, 1843, 1856, 1857, 1865, 1871, 1991, 2012, 2022, 2036, 2094, 2155, 2273, 2283, 2337, 2340, 2347, 2348, 2621, 2628, 2638, 2651, 2664, 2669, 2672, 2673, 2674, 2675, 2683, 2692, 2708, 2772, 2800, 2806, 2815, 2820, 2826, 2829, 2833, 2841, 2851, 2860, 2864, 2868, 2936, 2952, 2959, 2968, 2982, 2983, 2988, 2989, 2990, 3008, 3009, 3010, 3011, 3012, 3013, 3014, 3015, 3016, 3024, 3043, 3045, 3075, 3076, 3077, 3124, 3154, 3226, 3237, 3247, 3367, 3378, 3379, 3423, 3454, 3496, 3507, 3516, 3525, 3580, 3590, 3600, 3712, 3714, 3786, 3790, 3794, 3821, 3832, 3850, 3851, 3852, 3859, 3875, 3881, 3884, 3892, 3902, 3917, 3925, 3933, 3944, 3960, 3980, 3994, 4016, 4038, 4046, 4048, 4051, 4058, 4063, 4144, 4145, 4205, 4238, 4239, 4240, 4243, 4252, 4273, 4289, 4324, 4325, 4335, 4348, 4403, 4404, 4411, 4412, 4417, 4418, 4445, 4446, 4517, 4574, 4581, 4582, 4588, 4592, 4599, 4600, 4605, 4606, 4616, 4617, 4622, 4623, 4637, 4645, 4654, 4655, 4682, 4838, 4846, 4857, 4863, 4875, 4915, 4917, 4924, 4927, 4928, 4932, 4936, 4937, 4938, 4939, 4948, 4949, 4953, 4956, 4957, 4958, 4994, 4997, 5018, 5021, 5029, 5040, 5041, 5052, 5053, 5061, 5073, 5074, 5084, 5085, 5098, 5117, 5118, 5126, 5127, 5192, 5202, 5235, 5249, 5253, 5316, 5364, 5367, 5368, 5383, 5386, 5409, 5467, 5468, 5473, 5500, 5501, 5512, 5516, 5550, 5555, 5563, 5598, 5605, 5610, 5620, 5632, 5658, 5721, 5750, 5789, 5796,

5807, 5895, 5914, 5920, 5925, 5948,  
5960, 5961, 5964, 5976, 6010, 6040,  
6427, 6466, 6482, 6506, 6526, 6535,  
6592, 6599, 6619, 6637, 6648, 6650,  
6680, 6683, 6717, 6723, 6729, 6826,  
6864, 6895, 6896, 6926, 6952, 6997,  
7049, 7096, 7097, 7109, 7160, 7162,  
7215, 7227, 7237, 7246, 7266, 7277,  
7303, 7319, 7327, 7377, 7379, 7394,  
7396, 7417, 7588, 7609, 7687, 7704,  
7705, 7725, 7751, 7753, 7759, 7761,  
7766, 7796, 7832, 7843, 7932, 7934,  
7935, 7941, 8121, 8133, 8520, 8563,  
8582, 8604, 8624, 8640, 8650, 8666,  
8676, 8766, 8768, 8770, 8772, 8776,  
8779, 8977, 8985, 10293, 10296,  
10299, 10376, 10394, 10658, 10699,  
10708, 10729, 10739, 10743, 10750,  
10758, 10953, 10957, 10960, 11008,  
11021, 11341, 11350, 11361, 12057,  
12277, 12284, 12418, 12419, 12446,  
12456, 12471, 12480, 12490, 12506,  
12520, 12530, 12534, 12547, 12564,  
12579, 12805, 12810, 12817, 12822,  
12829, 12832, 12858, 12876, 12894,  
12902, 12912, 12922, 12928, 12935,  
12946, 12951, 12953, 12957, 12962,  
12966, 12967, 13114, 13126, 13175,  
13181, 13182, 13382, 13387, 13392,  
13399, 13404, 13503, 13581, 13585,  
13586, 13604, 13657, 13670, 13712,  
13718, 13719, 13730, 13743, 13744,  
13765, 13803, 13829, 13929, 14032,  
14040, 14048, 14059, 14076, 14078,  
14222, 14274, 14275, 14280, 14283,  
14284, 14306, 14359, 14459, 14675,  
14708, 14744, 14745, 14776, 14777,  
14797, 14798, 14816, 14901, 14905,  
14915, 14925, 14941, 14945, 14948,  
14953, 14955, 14998, 15002, 15003,  
15094, 15099, 15110, 15116, 15128,  
15131, 15133, 15137, 15251, 15265,  
15266, 16184, 16192, 16216, 16230,  
16238, 16249, 16259, 16279, 16309,  
16310, 16382, 16405, 16424, 16427,  
16649, 16652, 16689, 16750, 16767,  
16777, 16831, 16836, 17202, 17203,  
17212, 17235, 17252, 17253, 17255,  
17272, 17273, 17316, 17369, 17377,  
17404, 17412, 17418, 17441, 17479,  
17487, 17572, 17787, 17820, 17868,  
17873, 17989, 18014, 18040, 18049,  
18219, 18234, 18257, 18271, 18880,  
18881, 18882, 18883, 18888, 18889,  
18897, 18898, 18899, 18923, 18947,  
18953, 18971, 18980, 18982, 19060,  
19061, 19062, 19063, 19064, 19065,  
19066, 19132, 19137, 19142, 19147,  
19154, 19161, 19166, 19171, 19176,  
19181, 19186, 19191, 19196, 19201,  
19223, 19234, 19235, 19285, 19286,  
19408, 19417, 19426, 19434, 19511,  
19540, 19541, 19549, 19587, 19601,  
19610, 19620, 19642, 19643, 19644,  
19654, 19661, 19663, 19993, 19994,  
19995, 19996, 20005, 20006, 20007,  
20008, 20031, 20032, 20033, 20034,  
20043, 20044, 20045, 20046, 20151,  
20174, 20183, 20200, 20204, 20218,  
20221, 20383, 20384, 22293, 22294,  
22329, 22337, 22403, 22422, 22436,  
22585, 22601, 22605, 22617, 22627,  
22722, 22775, 22778, 22779, 22784,  
22798, 22836, 22837, 22838, 22839,  
22840, 22841, 22842, 22843, 22844,  
22845, 22846, 22847, 22860, 22862,  
22873, 22876, 22890, 22895, 22899,  
23028, 23119, 23120, 23129, 23130,  
23141, 23142, 23143, 23154, 23155,  
23156, 23163, 23174, 23175, 23176,  
23186, 23187, 23191, 23192, 23200,  
23203, 23204, 23212, 23223, 23243,  
23266, 23301, 23318, 23337, 23338,  
23347, 23353, 23373, 23374, 23402,  
23411, 23428, 23435, 23443, 23444,  
23545, 23546, 23547, 23550, 23553,  
23592, 23608, 23634, 23635, 23642,  
23650, 23679, 23680, 23683, 23685,  
23686, 23691, 23701, 23704, 23706,  
23711, 23742, 23755, 23760, 23766,  
23769, 23770, 23804, 23805, 23832,  
23833, 23846, 23849, 23860, 23883,  
23902, 23912, 23928, 23937, 23943,  
23949, 23953, 23958, 23964, 23979,  
23990, 24007, 24015, 24017, 24023,  
24044, 24072, 24095, 24128, 24130,  
24253, 24301, 24305, 24315, 24316,  
24332, 24343, 24347, 24357, 24358,  
24378, 24399, 24402, 24432, 24464,  
24480, 24500, 24541, 24553, 24566,  
24568, 24588, 24589, 24596, 24614,  
24653, 24663, 24828, 24844, 24855,  
24884, 24885, 24886, 24893, 24895,  
24896, 24902, 24903, 24906, 24933,  
24941, 24942, 24950, 24951, 24953,  
24954, 25125, 25138, 25148, 25149,  
25154, 25155, 25156, 25157, 25158,  
25159, 25166, 25176, 25183, 25194,

25195, 25206, 25223, 25268, 25269,  
25276, 25289, 25304, 25314, 25328,  
25358, 25367, 25401, 25423, 25441,  
25458, 25475, 25476, 25478, 25479,  
25484, 25499, 25532, 25561, 25562,  
25563, 25564, 25565, 25578, 25622,  
25695, 25762, 25764, 25765, 25775,  
25804, 25807, 25808, 25819, 25839,  
25902, 25903, 25904, 25916, 25955,  
25956, 25957, 25958, 25964, 25967,  
25969, 25979, 25997, 26012, 26024,  
26031, 26278, 26282, 26284, 26288,  
26295, 26296, 26306, 26307, 26310,  
26402, 26472, 26483, 26495, 26527,  
26534, 26545, 26561, 26568, 26629,  
26686, 26696, 26698, 26708, 26726,  
26727, 26759, 26762, 26771, 26773,  
26775, 26789, 26803, 26827, 26911,  
26919, 26950, 26958, 26964, 26975,  
26978, 26981, 26990, 26999, 27001,  
27007, 27014, 27017, 27026, 27034,  
27055, 27088, 27089, 27116, 27118,  
27136, 27137, 27156, 27167, 27176,  
27179, 27190, 27193, 27196, 27214,  
27224, 27235, 27237, 27246, 27293,  
27308, 27323, 27338, 27353, 27368,  
27371, 27373, 27390, 27435, 27742,  
27778, 27779, 27789, 27830, 27831,  
27855, 27882, 27883, 27886, 27888,  
27889, 27894, 27906, 27925, 27930,  
27938, 27941, 27973, 27983, 27984,  
27994, 28016, 28031, 28049, 28057,  
28060, 28088, 28096, 28112, 28184,  
28202, 28238, 28256, 28289, 28312,  
28318, 28391, 28392, 28523, 28524,  
28533, 28540, 28545, 28555, 28565,  
28590, 28593, 28606, 28638, 28646,  
28647, 28675, 28697, 28698, 28699,  
28702, 28707, 28727, 28728, 28780,  
28781, 28822, 28830, 28883, 28889,  
28902, 29429, 29430, 29433, 29690,  
29732, 29736, 29737, 29752, 29894,  
29898, 29909, 29930, 29934, 29950,  
29962, 29994, 29995, 29996, 29997,  
29998, 29999, 30000, 30097, 30098,  
30101, 30119, 30122, 30307, 30313,  
30591, 31356, 31359, 31360, 31363,  
31364, 31392, 31393, 31394, 31395,  
31396, 31397, 31423, 31424, 31425,  
31426, 31427, 31449, 31450, 31451,  
31452, 31471, 31472, 31473, 32986,  
32988, 32994, 36849, 37048, 37084,  
37089, 37090, 37208, 37215, 37220,  
37246, 37252, 37256, 37271, 37292,  
37300, 37336, 37337, 37338, 37344  
\c\_fifteen ..... 37468  
file commands:  
  \file\_add\_path:nN ..... 37488  
  \file\_compare\_timestamp:nNn .....  
    ..... 97, 11325, 11368  
  \file\_compare\_timestamp:nNnTF .....  
    ..... 97, 11325  
  \file\_compare\_timestamp\_p:nNn .....  
    ..... 97, 11325  
  \g\_file\_curr\_dir\_str .....  
    ..... 93, 10864, 11423, 11429, 11442  
  \g\_file\_curr\_ext\_str .....  
    ..... 93, 10864, 11425, 11431, 11444  
  \g\_file\_curr\_name\_str .....  
    ..... 93, 9186, 9313,  
      10864, 11424, 11430, 11443, 37501  
  \g\_file\_current\_name\_tl ..... 37500  
  \file\_full\_name:n 94, 11031, 11031,  
    11108, 11130, 11190, 11198, 11204,  
    11250, 11329, 11330, 36364, 36369  
  \file\_get:nnN .. 94, 10990, 10990,  
    10995, 37601, 37603, 37605, 37607  
  \file\_get:nnNTF .... 94, 10990, 10992  
  \file\_get\_full\_name:nN .....  
    ..... 94, 352, 11121,  
    11121, 11126, 11127, 11140, 37489  
  \file\_get\_full\_name:nNTF 94, 10204,  
    10997, 11121, 11123, 11135, 11136,  
    11137, 11379, 11385, 11390, 11402  
  \file\_get\_hex\_dump:nN .....  
    ..... 95, 11266, 11266, 11274  
  \file\_get\_hex\_dump:nnN .....  
    ..... 95, 11310, 11310, 11315  
  \file\_get\_hex\_dump:nnnNTF .....  
    ..... 95, 11310, 11312  
  \file\_get\_hex\_dump:nNTF .....  
    ..... 95, 11266, 11267  
  \file\_get\_md5\_hash:nN .....  
    ..... 96, 11268, 11276  
  \file\_get\_md5\_hash:nN\file\_-  
    get\_size:nN ..... 11266  
  \file\_get\_md5\_hash:nN\file\_-  
    get\_size:nNTF ..... 11266  
  \file\_get\_md5\_hash:nNTF 96, 11269  
  \file\_get\_size:nN .. 96, 11270, 11278  
  \file\_get\_size:nNTF ..... 96, 11271  
  \file\_get\_timestamp:nN .....  
    ..... 96, 11266, 11272, 11280  
  \file\_get\_timestamp:nNTF .....  
    ..... 96, 11266, 11273  
  \file\_hex\_dump:n .... 95, 11201, 11249  
  \file\_hex\_dump:nnn .....  
    ..... 95, 11201, 11201, 11319, 36346

```

\file_if_exist:n ..... 11377
\file_if_exist:nTF .....
..... 94, 97, 11377, 11688,
11690, 11694, 14241, 37491, 37493
\file_if_exist_input:n .....
..... 97, 11383, 11383
\file_if_exist_input:nTF .....
..... 97, 11383, 11388, 37490, 37492
\file_input:n .....
..... 97, 11400, 11400, 14245, 37491, 37493
\file_input_stop: ... 97, 11394, 11394
\file_list: ..... 37494
\file_log_list: 97, 11503, 11504, 37495
\file_mdfive_hash:n . 96, 11183, 11197
\file_parse_full_name:n .....
..... 95, 657, 11446, 11446
\file_parse_full_name:nNNN .....
..... 95, 11160, 11427, 11491, 11491, 11502
\file_parse_full_name_apply:nN ..
..... 95, 657, 11446, 11448, 11451, 11493
\file_path_include:n ..... 37496
\file_path_remove:n ..... 37498
\l_file_search_path_seq .....
..... 94-
97, 10898, 11045, 11145, 37497, 37499
\file_show_list: ... 97, 11503, 11503
\file_size:n ..... 96, 11183, 11183
\file_timestamp:n ... 96, 11183, 11185
file internal commands:
\l_file_base_name_tl .....
..... 10893, 11142, 11178
\file_compare_timestamp:nnN ...
..... 11325, 11328, 11333
\file_const:nn ..... 11702
\file_details:nn .....
..... 11183, 11184, 11186, 11187
\file_details_aux:nn .....
..... 11183, 11189, 11192, 11218
\l_file_dir_str .....
..... 10895, 11161, 11428, 11429
\file_ext_check:nn .....
..... 11054, 11066, 11073
\file_ext_check:nnn . 11088, 11093
\file_ext_check:nnnn . 11094, 11095
\file_ext_check:nnnw . 11079, 11084
\file_ext_check:nnw .....
..... 11074, 11075, 11082
\l_file_ext_str .....
.. 10895, 11161, 11162, 11428, 11431
\file_full_name:n .....
..... 11031, 11033, 11036
\file_full_name_assign:nnnNNN .
..... 11494, 11496
\file_full_name_aux:nN .....
..... 11031, 11057, 11058
\__file_full_name_aux:nn .....
..... 11031, 11039, 11041
\__file_full_name_aux:Nnn .....
..... 11031, 11046, 11050, 11056
\__file_full_name_aux:nnN .....
..... 11031, 11059, 11060
\l__file_full_name_tl .....
..... 10893, 10997, 11000,
11152, 11154, 11160, 11165, 11167,
11170, 11177, 11179, 11379, 11385,
11386, 11390, 11391, 11402, 11403
\__file_get_aux:nnN .....
..... 10990, 10999, 11006
\__file_get_details:nnN .....
..... 11266, 11275,
11277, 11279, 11281, 11282, 11292
\__file_get_do:Nw 10990, 11014, 11024
\__file_get_full_name_search:nN .
..... 11121, 11144, 11146, 11150, 11175
\__file_hex_dump:n .....
..... 11201, 11250, 11253, 11260
\__file_hex_dump_auxi:nnn .....
..... 11201, 11203, 11208
\__file_hex_dump_auxii:nnnn .....
..... 11201, 11217, 11222
\__file_hex_dump_auxiii:nnnn ...
..... 11201, 11225, 11227, 11232
\__file_hex_dump_auxiv:nnn .. 11201
\__file_hex_dump_auxiv:nnn .....
..... 11235, 11237, 11242
\__file_id_info_auxi:w .....
..... 11532, 11543, 11545
\__file_id_info_auxii:w .....
..... 660, 11532, 11555, 11557
\__file_id_info_auxiii:w .....
..... 11532, 11565, 11567
\__file_if_recursion_tail_-
break>NN ..... 10905
\__file_if_recursion_tail_stop:N
..... 10905
\__file_if_recursion_tail_stop_-
do:Nn ..... 10905
\__file_if_recursion_tail_stop_-
do:nn ..... 10906
\__file_input:n ..... 11386,
11391, 11400, 11403, 11406, 11418
\__file_input_pop: .....
..... 11400, 11416, 11434, 11439
\__file_input_pop:nnn .....
..... 11400, 11437, 11440
\__file_input_push:n .....
..... 11400, 11411, 11419, 11433

```

\g\_\_file\_internal\_ior . . . . .  
     11156, 11164, 11166, 11169, 11179,  
     11180, 11182, 11590, 11601, 11603  
\l\_\_file\_internal\_tl . . . . .  
     . . . . . 10863, 11436, 11437  
\\_\_file\_kernel\_dependency\_-  
     compare:nmn . . . . .  
     . . . . . 11569, 11575, 11578, 11580  
\\_\_file\_list:N . . . . .  
     . . . . . 11503, 11503, 11504, 11505  
\\_\_file\_list\_aux:n 11503, 11516, 11519  
\c\_\_file\_marker\_tl . . . . .  
     . . . . . 646, 10989, 11012, 11025  
\\_\_file\_mdfive\_hash:n . . . . .  
     . . . . . 11183, 11198, 11199  
\\_\_file\_mismatched\_dependency\_-  
     error:nn . . . . . 11585, 11588, 11588  
\\_\_file\_name\_cleanup:w . . . . .  
     . . . . . 11031, 11067, 11071  
\\_\_file\_name\_end: . . . . .  
     . . . . . 11031, 11052, 11071, 11072  
\\_\_file\_name\_expand:n . . . . .  
     . . . . . 10907, 10912, 10915  
\\_\_file\_name\_expand\_cleanup:Nw . .  
     . . . . . 645, 10907, 10917, 10921  
\\_\_file\_name\_expand\_cleanup:w . .  
     . . . . . 645, 10907, 10925, 10928  
\\_\_file\_name\_expand\_end: . . . . .  
     . . . . . 645, 10907, 10919, 10921, 10924,  
     10929, 10933, 10935, 10936, 10938  
\\_\_file\_name\_expand\_error:Nw . . .  
     . . . . . 645, 10907, 10924, 10935  
\\_\_file\_name\_expand\_error\_aux:Nw . .  
     . . . . . 645, 10907, 10936, 10937  
\\_\_file\_name\_ext\_check:nn . . . . . 11031  
\\_\_file\_name\_ext\_check:nnn . . . . . 11031  
\\_\_file\_name\_ext\_check:nnnn . . . . . 11031  
\\_\_file\_name\_ext\_check:nnnw . . . . . 11031  
\\_\_file\_name\_ext\_check:nww . . . . . 11031  
\\_\_file\_name\_quote:nw . . . . .  
     . . . . . 10981, 10982, 10983  
\l\_\_file\_name\_str . . . . .  
     . . . . . 10895, 11161, 11428, 11430  
\\_\_file\_name\_strip\_quotes:n . . . . .  
     . . . . . 10907, 10911, 10944  
\\_\_file\_name\_strip\_quotes:nnn . . . . . 10907  
\\_\_file\_name\_strip\_quotes:nnnw . . . . . 10907  
\\_\_file\_name\_strip\_quotes:nw . . . . .  
     . . . . . 10946, 10949, 10955, 10958  
\\_\_file\_name\_strip\_quotes\_-  
     end:wnwn . . . . . 10952, 10957  
\\_\_file\_name\_trim\_spaces:n . . . . .  
     . . . . . 10907, 10909, 10967  
  
\\_\_file\_name\_trim\_spaces:nw . . . . .  
     . . . . . 10907, 10968, 10969  
\\_\_file\_name\_trim\_spaces\_aux:n . .  
     . . . . . 10907, 10974, 10978  
\\_\_file\_name\_trim\_spaces\_aux:w . .  
     . . . . . 10907, 10979, 10980  
\\_\_file\_parse\_full\_name\_area:nw . .  
     . . . . . 657, 11456, 11458, 11461, 11465  
\\_\_file\_parse\_full\_name\_auxi:nN . .  
     . . . . . 11453, 11456, 11456  
\\_\_file\_parse\_full\_name\_base:nw . .  
     . . . . . 658, 11464, 11467, 11467, 11479  
\\_\_file\_parse\_full\_name\_tidy:nnNN . .  
     658, 11474, 11475, 11477, 11481, 11481  
\\_\_file\_parse\_version:w . . . . .  
     . . . . . 11569, 11583, 11584, 11587  
\\_\_file\_quark\_if\_nil:n . . . . . 10902  
\\_\_file\_quark\_if\_nil:nTF . . . . .  
     . . . . . 10902, 10971, 10985, 11077, 11086  
\\_\_file\_quark\_if\_nil\_p:n . . . . . 10902  
\g\_\_file\_record\_seq . . . . . 656,  
     659, 10892, 11410, 11513, 11527, 11528  
\\_\_file\_size:n . . . . .  
     . . . . . 11030, 11030, 11039, 11059, 11094  
\g\_\_file\_stack\_seq . . . . .  
     . . . . . 656, 10867, 11421, 11436  
\\_\_file\_str\_cmp:nn 11324, 11324, 11354  
\\_\_file\_timestamp:n . . . . .  
     . . . . . 11325, 11355, 11356, 11365, 11366  
\\_\_file\_tmp:w . . . . .  
     . . . . . 10869, 10873, 10877, 10883, 10889  
\l\_\_file\_tmp\_seq . . . . . 10899, 11507,  
     11510, 11513, 11514, 11516, 11524,  
     11529, 11600, 11602, 11625, 11629  
\filedump . . . . . 775  
\filemoddate . . . . . 776  
\filesize . . . . . 777  
\finalhyphendemerits . . . . . 271  
\firstrmark . . . . . 272  
\firstrmarks . . . . . 529  
\firstrvalidlanguage . . . . . 820  
\c\_five . . . . . 37448  
flag commands:  
     \flag\_clear:n . . . . .  
         . . . . . 170, 171, 5825, 7710, 7711,  
         14319, 14347, 14441, 14470, 14539,  
         14587, 14588, 14640, 14641, 14877,  
         14878, 14879, 14880, 14881, 14982,  
         15077, 15078, 15079, 15080, 15235,  
         15236, 15237, 18005, 18005, 18018  
     \flag\_clear\_new:n 171, 756, 14822,  
         14823, 14824, 14825, 15005, 15006,  
         15007, 15185, 15186, 18017, 18017

\flag\_height:n . . . . . 171, 7720, 7722,  
     14165, 18026, 18042, 18042, 18056  
 \flag\_if\_exist:n . . . . . 171, 18029  
 \flag\_if\_exist:nTF . 171, 18018, 18029  
 \flag\_if\_exist\_p:n . . . . . 171, 18029  
 \flag\_if\_raised:n . . . . . 171, 18034  
 \flag\_if\_raised:nTF . . . . .  
     171, 5833, 14158, 14163, 14165,  
     14849, 14855, 14860, 14867, 15039,  
     15044, 15049, 15200, 15207, 18034  
 \flag\_if\_raised\_p:n . . . . . 171, 18034  
 \flag\_log:n . . . . . 171, 18019, 18020  
 \flag\_new:n . . . . .  
     . . . . . 170, 171, 756, 841, 5815, 7570,  
     7571, 14027, 14028, 18000, 18000,  
     18018, 22924, 22925, 22926, 22927  
 \flag\_raise:n . . . . . 171, 7750, 7758,  
     14305, 14355, 14455, 14488, 14559,  
     14572, 14610, 14615, 14696, 14898,  
     14899, 14922, 14923, 14936, 14937,  
     14956, 14957, 14963, 14964, 14994,  
     15144, 15145, 15254, 15255, 15259,  
     15260, 15274, 15275, 18053, 18053  
 \flag\_raise\_if\_clear:n . . . . . 307,  
     5852, 5874, 22958, 22967, 22975,  
     22992, 23001, 23032, 36844, 36844  
 \flag\_show:n . . . . . 171, 18019, 18019  
 flag internal commands:  
   \\_\\_flag\_clear:wn . . . . .  
     . . . . . 18005, 18005, 18006, 18010  
   \\_\\_flag\_height\_end:wn . . . . .  
     . . . . . 18042, 18048, 18052  
   \\_\\_flag\_height\_loop:wn . . . . .  
     . . . . . 18042, 18042, 18043, 18046  
   \\_\\_flag\_show:Nn . . . . .  
     . . . . . 18019, 18019, 18020, 18021  
 \floatingpenalty . . . . . 273  
 floor . . . . . 254  
 \fmtname . . . . . 106, 8840, 8843, 8844,  
     8856, 8866, 30171, 30483, 30484,  
     33730, 33731, 34614, 34615, 36573  
 \font . . . . . 274  
 \fontchardp . . . . . 530  
 \fontcharht . . . . . 531  
 \fontcharic . . . . . 532  
 \fontcharwd . . . . . 533  
 \fontdimen . . . . . . . . . . . 962, 275  
 \fontencoding . . . . . 32473  
 \fontfamily . . . . . 32474  
 \fontid . . . . . . . . . . . 821  
 \fontname . . . . . 276  
 \fontseries . . . . . 32475  
 \fontshape . . . . . 32476  
 \fontsize . . . . . 32479  
 \footnotesize . . . . . 32515  
 \forcecjktoken . . . . . 1177  
 \formatname . . . . . 822  
 \c\_four . . . . . 37446  
 \c\_fourteen . . . . . 37466  
 fp commands:  
   \c\_e\_fp . . . . . 248, 250, 24805  
   flag\_fp\_division\_by\_zero . 249, 22924  
   flag\_fp\_invalid\_operation 249, 22924  
   flag\_fp\_overflow . . . . . 249, 22924  
   flag\_fp\_underflow . . . . . 249, 22924  
   \fp\_abs:n . . . . . 253,  
     258, 1176, 28413, 28413, 33399,  
     33501, 33503, 33505, 34329, 34331  
   \fp\_add:Nn . . . . .  
     . . . . . 241, 1176, 24768, 24768, 24774  
   \fp\_compare:n . . . . . 24830  
   \fp\_compare:nNn . . . . . 24846  
   \fp\_compare:nNnTF 244–246, 24846,  
     24987, 24993, 24998, 25006, 25067,  
     25073, 33256, 33258, 33263, 33532,  
     33547, 33556, 34071, 34303, 35222,  
     35226, 35234, 35241, 35248, 35255,  
     35262, 35309, 35720, 35723, 36169  
   \fp\_compare:nTF . . . . . 244–246,  
     252, 24830, 24959, 24965, 24970, 24978  
   \fp\_compare\_p:n . . . . . 245, 24830  
   \fp\_compare\_p:nNn . . . . . 244, 24846  
   \fp\_const:Nn . . . . . 241, 24745, 24749,  
     24753, 24805, 24806, 24807, 24808  
   \fp\_do\_until:nn . . . . .  
     . . . . . 246, 24956, 24956, 24960  
   \fp\_do\_until:nNnn . . . . .  
     . . . . . 245, 24984, 24984, 24988  
   \fp\_do\_while:nn . . . . .  
     . . . . . 246, 24956, 24962, 24966  
   \fp\_do\_while:nNnn . . . . .  
     . . . . . 245, 24984, 24990, 24994  
   \fp\_eval:n . . . . . 242,  
     245, 252–258, 1056, 28408, 28410,  
     34918, 34920, 34926, 34927, 34928,  
     34933, 34937, 34938, 34939, 34943,  
     34946, 34947, 34948, 35096, 35106,  
     35110, 35111, 35112, 35117, 35118,  
     35119, 35120, 35148, 35153, 35161,  
     35167, 35176, 35177, 35178, 35194,  
     35195, 35196, 35204, 35205, 35206,  
     35213, 35214, 35215, 35277, 35282,  
     35313, 35872, 35873, 35874, 35875,  
     35887, 35888, 35889, 35900, 36025,  
     36026, 36088, 36247, 36248, 36249,  
     36250, 36258, 36259, 36260, 36261,  
     36277, 36283, 36300, 36301, 36302,  
     36310, 36311, 36312, 37099, 37118

```

\fp_format:nn ..... 259
\fp_gadd:Nn .. 241, 24768, 24769, 24775
\fp_gset:N ..... 225, 21327
\fp_gset:Nn ..... 241, 24745, 24747, 24752, 24769, 24771
\fp_gset_eq:NN ..... 241, 24754, 24755, 24757, 24759
\fp_gsub:Nn .. 242, 24768, 24771, 24777
\fp_gzero:N ..... 241, 24758, 24759, 24761, 24765
\fp_gzero_new:N ..... 241, 24762, 24764, 24767
\fp_if_exist:N ..... 24820, 24821
\fp_if_exist:NTF ..... 243, 24763, 24765, 24820
\fp_if_exist_p:N ..... 243, 24820
\fp_if_nan:n ..... 306, 24822
\fp_if_nan:nTF ..... 259, 306, 24822
\fp_if_nan_p:n ..... 306, 24822
\fp_log:N .... 249, 24778, 24780, 24781
\fp_log:n ..... 249, 24801, 24803
\fp_max:nn ..... 258, 28415, 28415
\fp_min:nn ..... 258, 28415, 28417
\fp_new:N ..... 241, 24742, 24744, 24763, 24765,
24809, 24810, 24811, 24812, 33222,
33223, 33224, 33350, 33351, 33600,
33601, 34098, 34099, 34263, 34264
\fp_set:N ..... 225, 21327
\fp_set:Nn ..... 241, 24745, 24751, 24768, 24770,
33244, 33245, 33246, 33369, 33371,
33412, 33432, 33452, 33469, 33471,
33489, 33490, 33530, 33531, 34116,
34117, 34283, 34285, 34323, 34324
\fp_set_eq:NN ..... 241, 24754, 24754, 24756, 24758,
33417, 33437, 33454, 33533, 33534
\fp_show:N ... 249, 24778, 24778, 24779
\fp_show:n ..... 249, 24801, 24801
\fp_sign:n ..... 242, 28411, 28411
\fp_step_function:nnnN ..... 247, 25012, 25012, 25019, 25104
\fp_step_inline:nnnn 247, 25082, 25082
\fp_step_variable:nnnNn ..... 247, 25082, 25089
\fp_sub:Nn .. 242, 24768, 24770, 24776
\fp_to_decimal:N .. 242, 243, 22915,
28215, 28215, 28217, 28246, 28408
\fp_to_decimal:n ..... 242, 243, 28215, 28218,
28410, 28412, 28414, 28416, 28418
\fp_to_dim:N ..... 242, 1174, 28338, 28338, 28340
\fp_to_dim:N ..... 242, 248, 28338, 28341, 33288,
33299, 33399, 34026, 34048, 34076,
34090, 34200, 34208, 34339, 34341
\fp_to_int:N .. 242, 28354, 28354, 28355
\fp_to_int:n .. 242, 28354, 28356, 35724
\fp_to_scientific:N ..... 243, 28161, 28161, 28163, 28192, 28199
\fp_to_scientific:n 243, 28161, 28164
\fp_to_tl:N ..... 243, 261, 22916, 24792, 28294, 28294, 28295
\fp_to_tl:n ..... 243, 22542, 22957, 22966, 22991, 23000, 23029,
24626, 24641, 24802, 24804, 25039,
25040, 25059, 25070, 28294, 28296
\fp_trap:nn ..... 249, 996, 22928,
22928, 23043, 23044, 23045, 23046
\fp_until_do:nn ..... 246, 24956, 24968, 24973
\fp_until_do:nNnn ..... 246, 24984, 24996, 25001
\fp_use:N 243, 261, 28408, 28408, 28409
\fp_while_do:nn ..... 246, 24956, 24976, 24981
\fp_while_do:nNnn ..... 246, 24984, 25004, 25009
\fp_zero:N ..... 241, 24758, 24758, 24760, 24763
\fp_zero_new:N 241, 24762, 24762, 24766
\c_inf_fp ..... 248, 257, 22556, 24138, 25575, 25657,
25995, 26755, 26778, 26980, 26983,
26987, 27010, 27212, 27375, 28900
\c_minus_inf_fp ..... 248, 257, 22556, 25576,
25660, 25993, 26530, 27376, 28901
\c_minus_zero_fp ..... 247, 22556, 25572, 28095, 28899
\c_nan_fp ... 257, 999, 1024, 22556,
22968, 22976, 23048, 23254, 23273,
23279, 23302, 23469, 23477, 23485,
23563, 23620, 23659, 24050, 24127,
24139, 24628, 24643, 25063, 26954,
28452, 28498, 28813, 28872, 28898
\c_one_degree_fp 248, 257, 24141, 24807
\c_one_fp ..... 247, 1052, 1160, 24142, 24571, 24592, 24805,
25163, 26016, 26749, 26949, 27000,
27185, 27299, 27329, 27878, 28514
\c_pi_fp . 248, 257, 1034, 24140, 24807
\g_tmpa_fp ..... 248, 24809
\l_tmpa_fp ..... 248, 24809
\g_tmpb_fp ..... 248, 24809
\l_tmpb_fp ..... 248, 24809

```

\c\_zero\_fp . . . . . 247,  
   1055, 1072, 1188, 22556, 22610,  
   24143, 24583, 24595, 24743, 24758,  
   24759, 25165, 25168, 25404, 25571,  
   26758, 26779, 26977, 27013, 28093,  
   28199, 28383, 28897, 33256, 33258,  
   33263, 33547, 33556, 34303, 36169  
 fp internal commands:  
   \\_\_fp\_ . . . . . 25172, 25179, 25188, 25189  
   \\_\_fp\_&\_o:ww . . . . . 1058, 1067, 25169  
   \\_\_fp\_&\_tuple\_o:ww . . . . . 25169  
   \\_\_fp\_\*\_o:ww . . . . . 25536  
   \\_\_fp\_\*\_tuple\_o:ww . . . . . 26042  
   \\_\_fp+\_o:ww . . . . . 1070, 1071, 1100, 25257  
   \\_\_fp-\_o:ww . . . . . 1070, 1071, 25252  
   \\_\_fp/\_o:ww . . . . . 1079, 1080, 1123, 25648  
   \\_\_fp^\_o:ww . . . . . 26945  
   \\_\_fp\_acos\_o:w 1164, 1167, 28034, 28034  
   \\_\_fp\_acot\_o:Nw . . . . .  
     . . . . . 27274, 27276, 27866, 27872  
   \\_\_fp\_acotii\_o:Nww 27876, 27879, 27899  
   \\_\_fp\_acotii\_o:ww . . . . . 1160  
   \\_\_fp\_acsc\_normal\_o:NnwNnw . . . . .  
     . . . . . 1166, 28092, 28107, 28115, 28115  
   \\_\_fp\_acsc\_o:w . . . . . 28086, 28086  
   \\_\_fp\_add:NNNn . . . . . 24768,  
     24768, 24769, 24770, 24771, 24772  
   \\_\_fp\_add\_big\_i:wNww . . . . . 1073  
   \\_\_fp\_add\_big\_i\_o:wNww . . . . .  
     . . . . . 1070, 1073, 25324, 25331, 25331  
   \\_\_fp\_add\_big\_ii:wNww . . . . . 1073  
   \\_\_fp\_add\_big\_ii\_o:wNww . . . . .  
     . . . . . 25327, 25331, 25339  
   \\_\_fp\_add\_inf\_o:Nww . . . . .  
     . . . . . 25273, 25293, 25293  
   \\_\_fp\_add\_normal\_o:Nww . . . . .  
     . . . . . 1073, 25272, 25308, 25308  
   \\_\_fp\_add\_npos\_o:NnwNnw . . . . .  
     . . . . . 1073, 25311, 25317, 25317  
   \\_\_fp\_add\_return\_ii\_o:Nww . . . . .  
     . . . . . 25275, 25281, 25281, 25286  
   \\_\_fp\_add\_significand\_carry\_-  
     o:wwwNN . . . . . 1075, 25364, 25379, 25379  
   \\_\_fp\_add\_significand\_no\_carry\_-  
     o:wwwNN . . . . . 1074, 25366, 25369, 25369  
   \\_\_fp\_add\_significand\_o:NnnnnnnnN  
     1073, 1074, 25334, 25342, 25347, 25347  
   \\_\_fp\_add\_significand\_pack:NNNNNNN  
     . . . . . 25347, 25351, 25354  
   \\_\_fp\_add\_significand\_test\_o:N ..  
     . . . . . 25347, 25349, 25361  
   \\_\_fp\_add\_zeros\_o:Nww . . . . .  
     . . . . . 25271, 25283, 25283  
  
   \\_\_fp\_and\_return:wNw . . . . .  
     . . . . . 25169, 25175, 25182, 25194  
   \\_\_fp\_array\_bounds:NNnTF . . . . .  
     . . . . . 28769, 28769, 28800, 28870  
   \\_\_fp\_array\_bounds\_error>NNn . . . . .  
     . . . . . 28769, 28772, 28776, 28783  
   \\_\_fp\_array\_count:n 22659, 22659,  
     23238, 24913, 24914, 26055, 28134  
   \\_\_fp\_array\_gset:NNNNww . . . . .  
     . . . . . 28788, 28791, 28798  
   \\_\_fp\_array\_gset:w 28788, 28804, 28815  
   \\_\_fp\_array\_gset\_normal:w . . . . .  
     . . . . . 28788, 28819, 28825  
   \\_\_fp\_array\_gset\_recover:Nw . . . . .  
     . . . . . 28788, 28805, 28810  
   \\_\_fp\_array\_gset\_special:nnNNN . . . . .  
     . . . . . 28788,  
       28818, 28820, 28821, 28833, 28845  
   \\_\_fp\_array\_gzero:N . . . . . 1188  
   \\_\_fp\_array\_if\_all\_fp:nTF . . . . .  
     . . . . . 22671, 22671, 24621  
   \\_\_fp\_array\_if\_all\_fp\_loop:w . . . . .  
     . . . . . 22671, 22673, 22676, 22679  
   \g\_fp\_array\_int . . . . .  
     . . . . . 28734, 28741, 28743, 28755  
   \\_\_fp\_array\_item:N 28852, 28876, 28881  
   \\_\_fp\_array\_item:NNNnN . . . . .  
     . . . . . 28852, 28871, 28874  
   \\_\_fp\_array\_item:NwN . . . . .  
     . . . . . 28852, 28862, 28868  
   \\_\_fp\_array\_item:w 28852, 28884, 28886  
   \\_\_fp\_array\_item\_normal:w . . . . .  
     . . . . . 28852, 28888, 28904  
   \\_\_fp\_array\_item\_special:w . . . . .  
     . . . . . 28852, 28883, 28892  
   \l\_fp\_array\_loop\_int . . . . .  
     . . . . . 28735, 28841, 28844, 28847  
   \\_\_fp\_array\_new:nNNN . . . . . 28736  
   \\_\_fp\_array\_new:nNNNN . . . . . 28745, 28749  
   \\_\_fp\_array\_to\_clist:n . . . . .  
     . . . . . 23306, 28419, 28419, 28538  
   \fp\_array\_to\_clist\_loop:Nw . . . . .  
     . . . . . 28419, 28425, 28430, 28435  
   \fp\_asec\_o:w . . . . . 28099, 28099  
   \fp\_asin\_auxi\_o:NnwN . . . . .  
     1165, 1167, 28064, 28067, 28067, 28126  
   \fp\_asin\_isqrt:wn . . . . .  
     . . . . . 28067, 28070, 28077  
   \fp\_asin\_normal\_o:NnwNnnnnw . . . . .  
     . . . . . 28025, 28041, 28052, 28052  
   \fp\_asin\_o:w . . . . . 28019, 28019  
   \fp\_atan\_auxi:ww . . . . .  
     . . . . . 1162, 27944, 27958, 27958  
   \fp\_atan\_auxii:w 27958, 27959, 27960

`\_fp_atan_combine_aux:ww . . . . .`  
`..... 27985, 27999, 28006`  
`\_fp_atan_combine_o:NwwwN . . . . .`  
`... 1161, 27903, 27920, 27985, 27985`  
`\_fp_atan_default:w . . . . .`  
`1052, 1160, 27866, 27870, 27876, 27878`  
`\_fp_atan_div:wnwwn . . . . .`  
`..... 1161, 27931, 27933, 27933`  
`\_fp_atan_inf_o:NNNw . . . . .`  
`..... 1160, 27891, 27892,`  
`27893, 27901, 27901, 28037, 28110`  
`\_fp_atan_near:wwn . . . . .`  
`..... 27933, 27940, 27946`  
`\_fp_atan_near_aux:wwn . . . . .`  
`..... 27933, 27951, 27953`  
`\_fp_atan_normal_o:NNnwNw . . . . .`  
`..... 1160, 27895, 27911, 27911`  
`\_fp_atan_o:Nw . . . . .`  
`..... 27278, 27280, 27866, 27866`  
`\_fp_atan_Taylor_break:w . . . . .`  
`..... 27969, 27972, 27982`  
`\_fp_atan_Taylor_loop:ww . . . . .`  
`... 1162, 27964, 27969, 27969, 27977`  
`\_fp_atan_test_o:NwwNwwN . . . . .`  
`... 1166, 27914, 27918, 27918, 28074`  
`\_fp_atanii_o:Nww . . . . .`  
`..... 27870, 27879, 27879, 27900`  
`\_fp_basics_pack_high:NNNNNw . . . . .`  
`..... 1074,`  
`1091, 22769, 22771, 25372, 25524,`  
`25627, 25639, 25781, 25974, 26500`  
`\_fp_basics_pack_high_carry:w . . . . .`  
`..... 988, 22769, 22774, 22778`  
`\_fp_basics_pack_low:NNNNNw . . . . .`  
`..... 1081, 1091, 22769,`  
`22769, 25374, 25526, 25629, 25641,`  
`25783, 25923, 25925, 25976, 26502`  
`\_fp_basics_pack_weird_high:NNNNNNNNw . . . . .`  
`..... 22780, 22788, 25383, 25792`  
`\_fp_basics_pack_weird_low:NNNNw . . . . .`  
`..... 22780, 22780, 25385, 25794`  
`\c_fp_big_leading_shift_int . . . . .`  
`.. 22755, 25853, 26188, 26198, 26208`  
`\c_fp_big_middle_shift_int . . . . .`  
`... 22755, 25856, 25859, 25862,`  
`25865, 25868, 25871, 25875, 26190,`  
`26200, 26210, 26220, 26223, 26226`  
`\c_fp_big_trailing_shift_int . . . . .`  
`..... 22755, 25879, 26233`  
`\c_fp_Bigg_leading_shift_int . . . . .`  
`..... 22760, 25702, 25720`  
`\c_fp_Bigg_middle_shift_int . . . . .`  
`.. 22760, 25705, 25708, 25723, 25726`  
  
`\c_fp_Bigg_trailing_shift_int . . . . .`  
`..... 22760, 25711, 25729`  
`\_fp_binary_rev_type_o:Nww . . . . .`  
`..... 24261, 24274, 26045, 26047`  
`\_fp_binary_type_o:Nww . . . . .`  
`..... 24261, 24261, 26043, 26056`  
`\c_fp_block_int . . . . .`  
`..... 22561, 26452`  
`\_fp_case_return:nw . . . . .`  
`991, 22837, 22837, 22867, 22870, 22875,`  
`23367, 26714, 27209, 27891, 27892,`  
`27893, 28186, 28240, 28314, 28316,`  
`28317, 28383, 28818, 28820, 28821`  
`\_fp_case_return_i_o:ww . . . . .`  
`..... 22844, 22844,`  
`25274, 25288, 25297, 25569, 27882`  
`\_fp_case_return_ii_o:ww . . . . .`  
`22844, 22846, 25570, 26998, 27016, 27883`  
`\_fp_case_return_o:Nw . . . . .`  
`..... 991, 992, 22838,`  
`22838, 25995, 26749, 26754, 26757,`  
`26949, 26954, 26977, 26980, 26983,`  
`27185, 27299, 27329, 28093, 28095`  
`\_fp_case_return_o:Nww . . . . .`  
`..... 22842, 22842, 25571, 25572,`  
`25575, 25576, 27000, 27009, 27012`  
`\_fp_case_return_same_o:w . . . . .`  
`..... 991, 992,`  
`22840, 22840, 25804, 25808, 25996,`  
`26008, 26011, 26533, 26761, 26974,`  
`27189, 27192, 27284, 27292, 27307,`  
`27322, 27337, 27344, 27352, 27367,`  
`28022, 28030, 28048, 28094, 28111`  
`\_fp_case_use:nw . . . . .`  
`991, 22836,`  
`22836, 25299, 25567, 25568, 25573,`  
`25574, 25656, 25659, 25806, 25992,`  
`26526, 26529, 26985, 27195, 27285,`  
`27290, 27300, 27305, 27315, 27320,`  
`27330, 27335, 27345, 27350, 27360,`  
`27365, 28024, 28027, 28037, 28039,`  
`28045, 28089, 28091, 28102, 28105,`  
`28110, 28189, 28196, 28243, 28250`  
`\_fp_change_func_type:NNN . . . . .`  
`22699, 22699, 24054, 26038, 28171,`  
`28225, 28302, 28348, 28363, 28802`  
`\_fp_change_func_type_aux:w . . . . .`  
`..... 22699, 22708, 22715`  
`\_fp_change_func_type_chk:NNN . . . . .`  
`..... 22699, 22705, 22716`  
`\_fp_chk:w . . . . .`  
`977–979,`  
`1034, 1071, 1073, 1075, 1081, 1084,`  
`22543, 22544, 22545, 22556, 22557,`  
`22558, 22559, 22560, 22570, 22575,`  
`22577, 22578, 22606, 22609, 22611,`  
`22621, 22634, 22653, 22848, 22864,`

23024, 23029, 23256, 23310, 23319,  
23321, 24152, 24789, 24795, 24798,  
24799, 24880, 24881, 25043, 25059,  
25063, 25127, 25128, 25131, 25142,  
25143, 25151, 25152, 25160, 25172,  
25175, 25179, 25182, 25258, 25278,  
25279, 25281, 25282, 25283, 25291,  
25294, 25305, 25306, 25308, 25317,  
25393, 25545, 25579, 25580, 25583,  
25664, 25802, 25810, 25812, 25989,  
25998, 26000, 26005, 26013, 26015,  
26017, 26021, 26523, 26535, 26537,  
26746, 26763, 26765, 26946, 26965,  
26967, 26968, 26971, 26988, 26991,  
26994, 27018, 27019, 27021, 27037,  
27126, 27139, 27141, 27145, 27149,  
27182, 27198, 27281, 27294, 27296,  
27309, 27311, 27324, 27326, 27339,  
27341, 27354, 27356, 27369, 27379,  
27880, 27896, 27897, 27901, 27912,  
28019, 28032, 28034, 28050, 28053,  
28063, 28086, 28097, 28099, 28113,  
28115, 28120, 28182, 28203, 28206,  
28236, 28257, 28260, 28310, 28326,  
28329, 28404, 28405, 28515, 28517,  
28549, 28815, 28823, 28826, 28905  
\\_fp\_compare:wNNNNw . . . . . [24511](#)  
\\_fp\_compare\_aux:wn . . . . .  
..... [24846](#), 24849, 24857  
\\_fp\_compare\_back:ww . . . . . [1182](#),  
[24862](#), [24867](#), [24879](#), [25141](#), [28533](#)  
\\_fp\_compare\_back\_any:ww . . . . .  
..... [1059](#)–[1061](#),  
24586, 24859, [24862](#), 24862, 24930  
\\_fp\_compare\_back\_tuple:ww . . . . .  
..... [24907](#), [24907](#)  
\\_fp\_compare\_nan:w . . . . .  
... [1060](#), [24862](#), [24884](#), [24885](#), [24906](#)  
\\_fp\_compare\_npos:nwnw . . . . .  
..... [1058](#), [1060](#), [1061](#),  
24890, [24936](#), [24936](#), [25395](#), [26302](#)  
\\_fp\_compare\_return:w . . . . .  
..... [24830](#), [24832](#), [24835](#)  
\\_fp\_compare\_significand:n>NNNNNN  
..... [24936](#), [24939](#), [24944](#)  
\\_fp\_cos\_o:w . . . . . [27296](#), [27296](#)  
\\_fp\_cot\_o:w . . . . . [1145](#), [27356](#), [27356](#)  
\\_fp\_cot\_zero\_o:Nnw . . . . .  
1144, [1145](#), [27314](#), [27356](#), [27359](#), [27371](#)  
\\_fp\_csc\_o:w . . . . . [27311](#), [27311](#)  
\\_fp\_decimate:nNnnn . . . . .  
... [989](#), [992](#), [1138](#), [22790](#), [22790](#),  
[22855](#), [22882](#), [23323](#), [25333](#), [25341](#),  
[25420](#), [26792](#), [26796](#), [27164](#), [28266](#)  
\\_fp\_decimate:\_Nnnn . . . . . [22802](#), [22802](#)  
\\_fp\_decimate\_auxi:Nnnnn . . . . . [990](#), [22806](#)  
\\_fp\_decimate\_auxii:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxiii:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxiv:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxix:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxv:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxvi:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxvii:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxviii:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxx:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxi:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxii:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxiii:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxiv:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxv:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_auxxvi:Nnnnn . . . . . [22806](#)  
\\_fp\_decimate\_pack:nNNNNNNNNNw . . . . .  
..... [990](#), [22813](#), [22832](#), [22832](#)  
\\_fp\_decimate\_pack:nNNNNNNNw . . . . .  
..... [22833](#), [22834](#)  
\\_fp\_decimate\_tiny:Nnnnn . . . . .  
..... [22802](#), [22804](#)  
\\_fp\_div\_npos\_o:Nnw . . . . .  
.... [1083](#), [1084](#), [25653](#), [25663](#), [25663](#)  
\\_fp\_div\_significand\_calc:wwwNNNNNN  
..... [1087](#), [25680](#),  
[25689](#), [25689](#), [25737](#), [26606](#), [26613](#)  
\\_fp\_div\_significand\_calc\_-  
i:wwwNNNNNN . . . . . [25689](#), [25692](#), [25697](#)  
\\_fp\_div\_significand\_calc\_-  
ii:wwwNNNNNN . . . . . [25689](#), [25694](#), [25715](#)  
\\_fp\_div\_significand\_i\_o:wnw ..  
... [1084](#), [1087](#), [25670](#), [25676](#), [25676](#)  
\\_fp\_div\_significand\_ii:wnw . . .  
..... [1089](#),  
[25684](#), [25685](#), [25686](#), [25733](#), [25733](#)  
\\_fp\_div\_significand\_iii:wwwNNNN  
..... [1089](#), [25687](#), [25740](#), [25740](#)  
\\_fp\_div\_significand\_iv:wwwNNNNNN  
..... [1089](#), [25743](#), [25748](#), [25748](#)  
\\_fp\_div\_significand\_large\_-  
o:wwwNNNNwN . . . . .  
..... [1091](#), [25774](#), [25788](#), [25788](#)  
\\_fp\_div\_significand\_pack:NNN ..  
..... [1091](#), [1124](#), [25735](#),  
[25768](#), [25768](#), [26593](#), [26611](#), [26619](#)  
\\_fp\_div\_significand\_small\_-  
o:wwwNNNNwN . . . . .  
..... [1091](#), [25772](#), [25778](#), [25778](#)  
\\_fp\_div\_significand\_test\_o:w ..  
..... [1091](#), [25678](#), [25769](#), [25769](#)  
\\_fp\_div\_significand\_v:NN . . . . .  
..... [25753](#), [25755](#), [25758](#)

$\text{\_fp\_div\_significand\_v:Nw}$  . . . . . [25748](#)  
 $\text{\_fp\_div\_significand\_vi:Nw}$  . . . . . [1090](#), [25748](#), [25751](#), [25759](#)  
 $\text{\_fp\_division\_by\_zero\_o:Nw}$  . . . . . [996](#), [22988](#), [23036](#),  
[23039](#), [25993](#), [26530](#), [27375](#), [27376](#)  
 $\text{\_fp\_division\_by\_zero\_o:Nw}$  . . . . . [996](#), [22996](#),  
[23036](#), [23040](#), [25657](#), [25660](#), [26987](#)  
 $\text{\_c\_fp\_empty\_tuple\_fp}$  . . . . . [22654](#), [23463](#), [24113](#), [24123](#)  
 $\text{\_fp\_ep\_compare:www}$  . . . . . [26297](#), [26297](#), [27927](#)  
 $\text{\_fp\_ep\_compare\_aux:www}$  . . . . . [26297](#), [26298](#), [26299](#)  
 $\text{\_fp\_ep\_div:wwwn}$  . . . . . [1157](#), [26327](#), [26327](#), [26438](#),  
[27856](#), [27943](#), [27947](#), [27956](#), [28123](#)  
 $\text{\_fp\_ep\_div\_eps\_pack:NNNNw}$  . . . . . [26357](#), [26361](#), [26363](#), [26366](#)  
 $\text{\_fp\_ep\_div\_epsi:wnNNNNn}$  . . . . . [1114](#)  
 $\text{\_fp\_ep\_div\_epsi:wnNNNNN}$  . . . . . [26354](#), [26357](#), [26357](#)  
 $\text{\_fp\_ep\_div\_epsii:wnNNNNN}$  . . . . . [26357](#), [26359](#), [26368](#)  
 $\text{\_fp\_ep\_div\_esti:www}$  . . . . . [1114](#), [26333](#), [26336](#), [26336](#)  
 $\text{\_fp\_ep\_div\_estii:wwnnwn}$  . . . . . [26336](#), [26338](#), [26344](#)  
 $\text{\_fp\_ep\_div\_estiii:NNNNNwwn}$  . . . . . [26336](#), [26346](#), [26351](#)  
 $\text{\_fp\_ep\_inv\_to\_float\_o:wN}$  . . . . . [1146](#)  
 $\text{\_fp\_ep\_inv\_to\_float\_o:wwN}$  [1156](#),  
[26434](#), [26436](#), [26442](#), [27318](#), [27333](#)  
 $\text{\_fp\_ep\_isqrt:wwn}$  [26380](#), [26380](#), [28084](#)  
 $\text{\_fp\_ep\_isqrt\_aux:wwn}$  . . . . . [26380](#)  
 $\text{\_fp\_ep\_isqrt\_auxi:wwn}$  [26383](#), [26385](#)  
 $\text{\_fp\_ep\_isqrt\_auxii:wwnnwn}$  . . . . . [26380](#), [26387](#), [26393](#)  
 $\text{\_fp\_ep\_isqrt\_epsi:wN}$  . . . . . [1117](#), [26417](#), [26420](#), [26420](#)  
 $\text{\_fp\_ep\_isqrt\_epsii:wwN}$  . . . . . [26420](#), [26423](#), [26424](#), [26425](#), [26427](#)  
 $\text{\_fp\_ep\_isqrt\_esti:wwnnwn}$  . . . . . [26395](#), [26398](#), [26398](#), [26403](#)  
 $\text{\_fp\_ep\_isqrt\_estii:wwnnwn}$  . . . . . [26398](#), [26401](#), [26408](#)  
 $\text{\_fp\_ep\_isqrt\_estiii:NNNNNwwn}$  . . . . . [26398](#), [26410](#), [26414](#)  
 $\text{\_fp\_ep\_mul:wwwn}$  . . . . . [1140](#), [26312](#), [26312](#), [27225](#),  
[27238](#), [27813](#), [27843](#), [28071](#), [28082](#)  
 $\text{\_fp\_ep\_mul\_raw:wwwN}$  . . . . . [26312](#), [26318](#), [26322](#), [27397](#), [27763](#)  
 $\text{\_fp\_ep\_to\_ep:wwN}$  . . . . . [26263](#), [26263](#), [26314](#),  
[26317](#), [26329](#), [26332](#), [26382](#), [28072](#)  
 $\text{\_fp\_ep\_to\_ep\_end:www}$  . . . . . [26263](#), [26277](#), [26281](#)  
 $\text{\_fp\_ep\_to\_ep\_loop:N}$  [1155](#), [26263](#),  
[26268](#), [26272](#), [26279](#), [26282](#), [27764](#)  
 $\text{\_fp\_ep\_to\_ep\_zero:ww}$  . . . . . [26263](#), [26287](#), [26295](#)  
 $\text{\_fp\_ep\_to\_fixed:www}$  [26245](#), [26245](#),  
[27394](#), [27950](#), [27959](#), [28069](#), [28558](#)  
 $\text{\_fp\_ep\_to\_fixed\_auxi:www}$  . . . . . [26245](#), [26247](#), [26252](#)  
 $\text{\_fp\_ep\_to\_fixed\_auxii:nnnnnnwn}$  . . . . . [26245](#), [26258](#), [26261](#)  
 $\text{\_fp\_ep\_to\_float\_o:wN}$  . . . . . [1146](#)  
 $\text{\_fp\_ep\_to\_float\_o:wwN}$  . . . . . [1143](#), [1156](#), [26434](#), [26434](#), [26439](#),  
[26446](#), [27249](#), [27288](#), [27303](#), [27862](#)  
 $\text{\_fp\_error:nnnn}$  . . . . . [22957](#), [22965](#), [22974](#),  
[22991](#), [22999](#), [23027](#), [23050](#), [23050](#),  
[23052](#), [23249](#), [23251](#), [23272](#), [23277](#),  
[24049](#), [24624](#), [24639](#), [25039](#), [25058](#),  
[25069](#), [28177](#), [28231](#), [28305](#), [28812](#)  
 $\text{\_fp\_exp\_after\_?\_f:nw}$  . . . . . [985](#), [1020](#), [23447](#)  
 $\text{\_fp\_exp\_after\_any\_f:Nw}$  . . . . . [22724](#), [22724](#), [22730](#)  
 $\text{\_fp\_exp\_after\_any\_f:nw}$  . . . . . [986](#), [22724](#), [22726](#), [22750](#), [23449](#), [24218](#)  
 $\text{\_fp\_exp\_after\_array\_f:w}$  . . . . . [986](#), [22735](#), [22744](#), [22749](#), [22750](#),  
[24103](#), [25209](#), [25220](#), [25230](#), [25238](#)  
 $\text{\_fp\_exp\_after\_expr\_mark\_f:nw}$  . . . . . [1020](#), [23447](#), [23455](#)  
 $\text{\_fp\_exp\_after\_expr\_stop\_f:nw}$  . . . . . [22724](#), [22734](#)  
 $\text{\_fp\_exp\_after\_f:nw}$  . . . . . [982](#), [1020](#),  
[22611](#), [22621](#), [22729](#), [24151](#), [24289](#)  
 $\text{\_fp\_exp\_after\_normal:nNw}$  . . . . . [22614](#), [22624](#), [22641](#), [22641](#)  
 $\text{\_fp\_exp\_after\_normal:Nwww}$  . . . . . [22643](#), [22651](#)  
 $\text{\_fp\_exp\_after\_o:w}$  . . . . . [982](#), [22611](#), [22611](#),  
[22841](#), [22845](#), [22847](#), [23317](#), [23361](#),  
[23379](#), [24606](#), [25159](#), [25177](#), [25186](#),  
[25195](#), [25282](#), [26019](#), [27138](#), [27143](#)  
 $\text{\_fp\_exp\_after\_special:nNw}$  . . . . . [983](#), [22616](#), [22626](#), [22631](#), [22631](#)  
 $\text{\_fp\_exp\_after\_tuple\_f:nw}$  . . . . . [22735](#), [22736](#), [22737](#), [24413](#)

$\backslash\text{--fp\_exp\_after\_tuple\_o:w}$  . . . . .  
 . . . . . 22735, 25184, 25187, 25190, 25192  
 $\backslash\text{c\_fp\_exp\_intarray}$  . . . . .  
 . . . . . 26839, 26925, 26932, 26935, 26937  
 $\backslash\text{--fp\_exp\_intarray:w}$  . . . . .  
 . . . . . 26896, 26909, 26922  
 $\backslash\text{--fp\_exp\_intarray\_aux:w}$  . . . . .  
 . . . . . 26896, 26930, 26933, 26936, 26939  
 $\backslash\text{--fp\_exp\_large:NwN}$  . . . . . 1132,  
 . . . . . 26896, 26898, 26904, 26917, 27122  
 $\backslash\text{--fp\_exp\_large\_after:wwn}$  . . . . .  
 . . . . . 1132, 26896, 26915, 26940  
 $\backslash\text{--fp\_exp\_normal\_o:w}$  . . . . .  
 . . . . . 26751, 26765, 26765  
 $\backslash\text{--fp\_exp\_o:w}$  . . . . . 26509, 26746, 26746  
 $\backslash\text{--fp\_exp\_overflow>NN}$  . . . . .  
 . . . . . 26765, 26778, 26779, 26806  
 $\backslash\text{--fp\_exp\_pos\_large:NnnNwN}$  . . . . .  
 . . . . . 26797, 26896, 26896  
 $\backslash\text{--fp\_exp\_pos\_o:NNwnw}$  . . . . .  
 . . . . . 26768, 26770, 26773  
 $\backslash\text{--fp\_exp\_pos\_o:NnwNw}$  . . . . . 26765  
 $\backslash\text{--fp\_exp\_Taylor:Nnnwn}$  . . . . .  
 . . . . . 26793, 26812, 26812, 26942  
 $\backslash\text{--fp\_exp\_Taylor\_break:Nww}$  . . . . .  
 . . . . . 26812, 26826, 26837  
 $\backslash\text{--fp\_exp\_Taylor\_ii:ww}$  . . . . . 26818, 26821  
 $\backslash\text{--fp\_exp\_Taylor\_loop:www}$  . . . . .  
 . . . . . 26812, 26822, 26823, 26832  
 $\backslash\text{--fp\_expand:n}$  . . . . . 1176  
 $\backslash\text{--fp\_exponent:w}$  . . . . . 22578, 22578  
 $\backslash\text{--fp\_facorial\_int\_o:n}$  . . . . . 1140  
 $\backslash\text{--fp\_fact\_int\_o:n}$  . . . . . 27203, 27206  
 $\backslash\text{--fp\_fact\_int\_o:w}$  . . . . . 27200  
 $\backslash\text{--fp\_fact\_loop\_o:w}$  . . . . .  
 . . . . . 27218, 27220, 27220, 27231  
 $\backslash\text{c\_fp\_fact\_max\_arg\_int}$  . . . . . 27181, 27208  
 $\backslash\text{--fp\_fact\_o:w}$  . . . . . 26513, 27182, 27182  
 $\backslash\text{--fp\_fact\_pos\_o:w}$  . . . . . 27197, 27200, 27200  
 $\backslash\text{--fp\_fact\_small\_o:w}$  . . . . . 27223, 27235  
 $\backslash\text{c\_fp\_five\_int}$  . . . . . 23111,  
 . . . . . 23135, 23148, 23161, 23168, 23221  
 $\backslash\text{--fp\_fixed\_<calculation>:wwn}$  . . . . . 1102  
 $\backslash\text{--fp\_fixed\_add:nnNnnwn}$  . . . . .  
 . . . . . 26138, 26146, 26148  
 $\backslash\text{--fp\_fixed\_add:Nnnnnwnn}$  . . . . .  
 . . . . . 26138, 26138, 26139, 26140  
 $\backslash\text{--fp\_fixed\_add:wwn}$  . . . . . 1102, 1105,  
 . . . . . 26138, 26138, 26378, 26688, 26696,  
 . . . . . 26707, 26725, 27955, 28015, 28573  
 $\backslash\text{--fp\_fixed\_add\_after:NNNNNwn}$  . . . . .  
 . . . . . 26138, 26142, 26156  
 $\backslash\text{--fp\_fixed\_add\_one:wN}$  . . . . .  
 . . . . . 1103, 26070, 26070,  
 . . . . . 26371, 26829, 26838, 28081, 28564  
 $\backslash\text{--fp\_fixed\_add\_pack:NNNNNwn}$  . . . . .  
 . . . . . 26138, 26144, 26151, 26154  
 $\backslash\text{--fp\_fixed\_continue:wn}$  . . . . .  
 . . . . . 26069, 26069, 26315,  
 . . . . . 26320, 26330, 26907, 27097, 27432,  
 . . . . . 27801, 28073, 28082, 28556, 28568  
 $\backslash\text{--fp\_fixed\_div\_int:wnN}$  . . . . .  
 . . . . . 26107, 26112, 26120, 26132  
 $\backslash\text{--fp\_fixed\_div\_int:wwN}$  . . . . . 1104,  
 . . . . . 26107, 26107, 26687, 26828, 27974  
 $\backslash\text{--fp\_fixed\_div\_int\_after:Nw}$  . . . . .  
 . . . . . 1105, 26107, 26109, 26137  
 $\backslash\text{--fp\_fixed\_div\_int\_auxi:wnn}$  . . . . .  
 . . . . . 26107, 26113,  
 . . . . . 26114, 26115, 26116, 26117, 26127  
 $\backslash\text{--fp\_fixed\_div\_int\_auxii:wnn}$  . . . . .  
 . . . . . 1105, 26107, 26118, 26135  
 $\backslash\text{--fp\_fixed\_div\_int\_pack:Nw}$  . . . . .  
 . . . . . 1105, 26107, 26130, 26136  
 $\backslash\text{--fp\_fixed\_div\_myriad:wn}$  . . . . .  
 . . . . . 26075, 26075, 26375  
 $\backslash\text{--fp\_fixed\_inv\_to\_float\_o:wN}$  . . . . .  
 . . . . . 26441, 26441, 26770, 27033  
 $\backslash\text{--fp\_fixed\_mul:mnnnnnnw}$  . . . . .  
 . . . . . 26158, 26178, 26180  
 $\backslash\text{--fp\_fixed\_mul:wwn}$  . . . . . 1102, 1104,  
 . . . . . 1106, 1154, 1156, 26158, 26158,  
 . . . . . 26324, 26355, 26370, 26372, 26376,  
 . . . . . 26429, 26432, 26445, 26689, 26699,  
 . . . . . 26739, 26830, 26928, 26943, 27043,  
 . . . . . 27770, 27824, 27962, 27995, 27997  
 $\backslash\text{--fp\_fixed\_mul\_add:nnnnwnnn}$  . . . . .  
 . . . . . 1109, 26227, 26229, 26229  
 $\backslash\text{--fp\_fixed\_mul\_add:nnnnwnnwN}$  . . . . .  
 . . . . . 1110, 26234, 26240, 26240  
 $\backslash\text{--fp\_fixed\_mul\_add:Nwnnnwnnn}$  . . . . .  
 . . . . . 1109,  
 . . . . . 26191, 26201, 26212, 26216, 26216  
 $\backslash\text{--fp\_fixed\_mul\_add:wwnn}$  . . . . .  
 . . . . . 1107, 26185, 26185, 28578  
 $\backslash\text{--fp\_fixed\_mul\_after:wwn}$  . . . . .  
 . . . . . 1107, 26077, 26083, 26083, 26086,  
 . . . . . 26160, 26187, 26197, 26207, 27060  
 $\backslash\text{--fp\_fixed\_mul\_one\_minus\_}$   
 $\text{mul:wwn}$  . . . . . 26185  
 $\backslash\text{--fp\_fixed\_mul\_short:wwn}$  . . . . .  
 . . . . . 1104, 26084, 26084,  
 . . . . . 26353, 26374, 26416, 26418, 28008  
 $\backslash\text{--fp\_fixed\_mul\_sub\_back:wwn}$  . . . . .  
 . . . . . 1107, 26185, 26195,  
 . . . . . 26430, 27791, 27793, 27794, 27795,

27796, 27797, 27798, 27799, 27800,  
 27804, 27806, 27807, 27808, 27809,  
 27810, 27811, 27812, 27837, 27839,  
 27840, 27841, 27842, 27845, 27847,  
 27848, 27849, 27850, 27975, 27983  
 $\backslash\text{--fp\_fixed\_one\_minus\_mul:wwn}$  ...  
                        1107–1109, 26205  
 $\backslash\text{--fp\_fixed\_sub:wwn}$  ...  
                        26138, 26139, 26422,  
 26705, 26721, 26733, 27436, 27956,  
 28013, 28079, 28566, 28575, 28607  
 $\backslash\text{--fp\_fixed\_to\_float\_o:Nw}$  ...  
                        26448, 26448, 26714  
 $\backslash\text{--fp\_fixed\_to\_float\_o:wN}$  ...  
                        1103, 1118, 1163, 26435,  
 26448, 26449, 26450, 26734, 26744,  
 26768, 27029, 28003, 28506, 28612  
 $\backslash\text{--fp\_fixed\_to\_float\_pack:ww}$  ...  
                        26481, 26491  
 $\backslash\text{--fp\_fixed\_to\_float\_rad\_o:wn}$  ...  
                        26443, 26443, 28003  
 $\backslash\text{--fp\_fixed\_to\_float\_round\_up:wnnnnw}$  ...  
                        26494, 26498  
 $\backslash\text{--fp\_fixed\_to\_float\_zero:w}$  ...  
                        26477, 26486  
 $\backslash\text{--fp\_fixed\_to\_loop:N}$  ...  
                        26454, 26464, 26468  
 $\backslash\text{--fp\_fixed\_to\_loop\_end:w}$  ...  
                        26470, 26474  
 $\backslash\text{--fp\_from\_dim:wNNnnnnnn}$  ...  
                        28373, 28396, 28399  
 $\backslash\text{--fp\_from\_dim:wnnnnwNn}$  28400, 28401  
 $\backslash\text{--fp\_from\_dim:wnnnnwNw}$  ... 28373  
 $\backslash\text{--fp\_from\_dim:wNw}$  28373, 28385, 28394  
 $\backslash\text{--fp\_from\_dim\_test:ww}$  1175, 23539,  
 23576, 24170, 28373, 28375, 28380  
 $\backslash\text{--fp\_func\_to\_name:N}$  ...  
                        22902, 22902, 24049, 24058  
 $\backslash\text{--fp\_func\_to\_name\_aux:w}$  ...  
                        22902, 22905, 22908  
 $\backslash\text{c\_fp\_half\_prec\_int}$  ...  
                        22561, 23780, 23812  
 $\backslash\text{--fp\_if\_type_fp:NTwFw}$  ...  
                        984, 1052, 22591,  
 22670, 22670, 22678, 22685, 22701,  
 22728, 24633, 24647, 24838, 24864,  
 24865, 25032, 25033, 25034, 25200  
 $\backslash\text{--fp\_inf\_fp:N}$  ... 22574, 22576, 23012  
 $\backslash\text{--fp\_int:w}$  ... 22848  
 $\backslash\text{--fp\_int:wTF}$  ... 22848, 28517  
 $\backslash\text{--fp\_int\_eval:w}$  ... 987,  
 1002, 1004, 1019, 1034, 1073, 1081,  
 1085, 1089, 1118, 22528, 22528,  
 22588, 22663, 22794, 22797, 23185,  
 23189, 23201, 23202, 23238, 23329,  
 23333, 23372, 23586, 23591, 23633,  
 23722, 23733, 23782, 23813, 23819,  
 23820, 23866, 23876, 23878, 23894,  
 23896, 23919, 23921, 24084, 24304,  
 24346, 24546, 24851, 25321, 25329,  
 25350, 25352, 25373, 25375, 25384,  
 25386, 25415, 25421, 25431, 25433,  
 25507, 25509, 25525, 25527, 25531,  
 25547, 25587, 25595, 25597, 25599,  
 25601, 25604, 25607, 25609, 25628,  
 25630, 25640, 25642, 25668, 25671,  
 25679, 25681, 25702, 25705, 25708,  
 25711, 25720, 25723, 25726, 25729,  
 25736, 25738, 25744, 25752, 25754,  
 25756, 25762, 25782, 25784, 25793,  
 25795, 25816, 25837, 25841, 25853,  
 25856, 25859, 25862, 25865, 25868,  
 25871, 25874, 25878, 25890, 25894,  
 25898, 25901, 25922, 25924, 25926,  
 25936, 25975, 25977, 25986, 26073,  
 26078, 26080, 26087, 26090, 26093,  
 26096, 26099, 26102, 26111, 26123,  
 26131, 26133, 26143, 26145, 26152,  
 26161, 26163, 26166, 26169, 26172,  
 26175, 26188, 26190, 26198, 26200,  
 26208, 26210, 26220, 26223, 26226,  
 26233, 26248, 26266, 26269, 26325,  
 26339, 26341, 26347, 26360, 26362,  
 26364, 26388, 26404, 26411, 26412,  
 26435, 26452, 26456, 26501, 26503,  
 26547, 26558, 26577, 26579, 26581,  
 26594, 26607, 26612, 26614, 26620,  
 26637, 26638, 26639, 26640, 26641,  
 26642, 26647, 26649, 26651, 26653,  
 26655, 26660, 26662, 26664, 26666,  
 26668, 26670, 26692, 26700, 26784,  
 26833, 26910, 26918, 26926, 26932,  
 26935, 27040, 27061, 27063, 27066,  
 27069, 27072, 27075, 27091, 27117,  
 27131, 27147, 27217, 27227, 27232,  
 27384, 27416, 27425, 27657, 27671,  
 27674, 27677, 27680, 27683, 27686,  
 27689, 27692, 27695, 27711, 27721,  
 27730, 27748, 27757, 27764, 27775,  
 27785, 27818, 27828, 27853, 27862,  
 27905, 27922, 27924, 27936, 27937,  
 27978, 27989, 28000, 28058, 28210,  
 28333, 28386, 28482, 28505, 28559,  
 28611, 28633, 28635, 28637, 28642,  
 28661, 28673, 28681, 28686, 28691  
 $\backslash\text{--fp\_int\_eval\_end:}$  ... 22528,  
 22529, 22588, 22666, 22785, 23238,  
 23343, 23347, 24547, 24851, 25531,

25566, 25758, 26133, 26269, 27091,  
 27147, 27417, 27426, 27775, 27785,  
 27828, 27853, 27937, 28640, 28642  
 $\backslash\text{--fp\_int\_p:w}$  ..... 22848  
 $\backslash\text{--fp\_int\_to\_roman:w}$  22528, 22530,  
 22797, 23794, 23826, 26574, 28743  
 $\backslash\text{--fp\_invalid\_operation:nnw}$  ....  
 ..... 995, 996,  
 22954, 23036, 23036, 23048, 28191,  
 28198, 28245, 28252, 28352, 28367  
 $\backslash\text{--fp\_invalid\_operation\_o:nw}$  ....  
 ..... 996, 23047,  
 23047, 23049, 24058, 25806, 26032,  
 26526, 27195, 27204, 27291, 27306,  
 27321, 27336, 27351, 27366, 28028,  
 28046, 28062, 28090, 28103, 28119  
 $\backslash\text{--fp\_invalid\_operation\_o:Nww}$  ....  
 ..... 996, 22962, 23036, 23037,  
 24259, 25301, 25573, 25574, 27132  
 $\backslash\text{--fp\_invalid\_operation\_o:nww}$  . 26057  
 $\backslash\text{--fp\_invalid\_operation\_tl\_o:nn}$  ..  
 ..... 996, 22971, 23036, 23038, 23304, 28537  
 $\backslash\text{--fp\_kind:w}$  22589, 22589, 23297, 24824  
 $\backslash\text{c\_--fp\_leading\_shift\_int}$  ....  
 ..... 22751, 26078,  
 26087, 26161, 27061, 27711, 27748  
 $\backslash\text{--fp\_ln\_c:NwNw}$  .....  
 ..... 1126, 1127, 26671, 26702, 26702  
 $\backslash\text{--fp\_ln\_div\_after:Nw}$  ....  
 ..... 1124, 26573, 26622  
 $\backslash\text{--fp\_ln\_div\_i:w}$  ..... 26595, 26604  
 $\backslash\text{--fp\_ln\_div\_ii:wwn}$  .....  
 .. 26598, 26599, 26600, 26601, 26609  
 $\backslash\text{--fp\_ln\_div\_vi:wwn}$  ... 26602, 26617  
 $\backslash\text{--fp\_ln\_exponent:wn}$  ....  
 ..... 1127, 26549, 26711, 26711  
 $\backslash\text{--fp\_ln\_exponent\_one:ww}$  26716, 26730  
 $\backslash\text{--fp\_ln\_exponent\_small:NNww}$  ....  
 ..... 26719, 26723, 26736  
 $\backslash\text{c\_--fp\_ln\_i\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_ii\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_iii\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_iv\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_ix\_fixed\_tl}$  ..... 26514  
 $\backslash\text{--fp\_ln\_npos\_o:w}$  ....  
 ..... 1120, 1121, 26535, 26537, 26537  
 $\backslash\text{--fp\_ln\_o:w}$  .....  
 ..... 1120, 1135, 26511, 26523, 26523  
 $\backslash\text{--fp\_ln\_significand:NNNNnnnnN}$  ...  
 ... 1122, 26548, 26551, 26551, 27041  
 $\backslash\text{--fp\_ln\_square\_t\_after:w}$  ....  
 ..... 26646, 26678  
 $\backslash\text{--fp\_ln\_square\_t\_pack:NNNNNw}$  ...  
 ... 26648, 26650, 26652, 26654, 26676  
 $\backslash\text{--fp\_ln\_t\_large:NNw}$  .....  
 ..... 1125, 26627, 26634, 26644  
 $\backslash\text{--fp\_ln\_t\_small:Nw}$  ... 26625, 26632  
 $\backslash\text{--fp\_ln\_t\_small:w}$  ..... 1125  
 $\backslash\text{--fp\_ln\_Taylor:wwNw}$  ....  
 ..... 1126, 26679, 26680, 26680  
 $\backslash\text{--fp\_ln\_Taylor\_break:w}$  26685, 26696  
 $\backslash\text{--fp\_ln\_Taylor\_loop:ww}$  ....  
 ..... 26681, 26682, 26691  
 $\backslash\text{--fp\_ln\_twice\_t\_after:w}$  26659, 26675  
 $\backslash\text{--fp\_ln\_twice\_t\_pack:Nw}$  . 26661,  
 26663, 26665, 26667, 26669, 26674  
 $\backslash\text{c\_--fp\_ln\_vi\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_vii\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_viii\_fixed\_tl}$  ..... 26514  
 $\backslash\text{c\_--fp\_ln\_x\_fixed\_tl}$  ....  
 ..... 26514, 26733, 26740  
 $\backslash\text{--fp\_ln\_x\_ii:wnnnn}$  .....  
 ..... 26553, 26571, 26571  
 $\backslash\text{--fp\_ln\_x\_iii:NNNNNNw}$  . 26580, 26584  
 $\backslash\text{--fp\_ln\_x\_iii\_var:NNNNNw}$  ....  
 ..... 26578, 26586  
 $\backslash\text{--fp\_ln\_x\_iv:wnnnnnnnn}$  ....  
 ..... 1124, 26576, 26591  
 $\backslash\text{--fp\_logb\_aux\_o:w}$  25989, 25994, 26000  
 $\backslash\text{--fp\_logb\_o:w}$  .. 25247, 25989, 25989  
 $\backslash\text{c\_--fp\_max\_exp\_exponent\_int}$  ....  
 ..... 22567, 26776  
 $\backslash\text{c\_--fp\_max\_exponent\_int}$  .. 22565,  
 22571, 22599, 26286, 26488, 27096  
 $\backslash\text{c\_--fp\_middle\_shift\_int}$  ....  
 ..... 22751, 26090,  
 26093, 26096, 26099, 26163, 26166,  
 26169, 26172, 27063, 27066, 27069,  
 27072, 27714, 27721, 27751, 27757  
 $\backslash\text{--fp\_minmax\_aux\_o:Nw}$  .....  
 ..... 25113, 25117, 25119  
 $\backslash\text{--fp\_minmax\_auxi:ww}$  .....  
 ..... 25135, 25147, 25154, 25154  
 $\backslash\text{--fp\_minmax\_auxii:ww}$  .....  
 ..... 25137, 25145, 25154, 25156  
 $\backslash\text{--fp\_minmax\_break\_o:w}$  .....  
 ..... 25128, 25158, 25158  
 $\backslash\text{--fp\_minmax\_loop:Nww}$  .... 1065,  
 25122, 25124, 25130, 25130, 25150  
 $\backslash\text{--fp\_minmax\_o:Nw}$  .....  
 ... 1058, 24817, 24819, 25113, 25113  
 $\backslash\text{c\_--fp\_minus\_min\_exponent\_int}$  ....  
 ..... 22565, 22600  
 $\backslash\text{--fp\_misused:n}$  .....  
 ..... 22541, 22541, 22545, 22656  
 $\backslash\text{--fp\_mul\_cases\_o:NnNnww}$  .....  
 ... 1083, 25538, 25544, 25650  
 $\backslash\text{--fp\_mul\_cases\_o:nNnnww}$  .... 25544

\\_\_fp\_mul\_npos\_o:Nww . . . . .  
     . . . . . 1080, 1081, 1083,  
     1174, 1175, 25541, 25582, 25582, 28403  
 \\_\_fp\_mul\_significand\_drop:NNNNNw . . . . .  
     . . . . . 1081, 25591,  
     25600, 25603, 25606, 25608, 25612  
 \\_\_fp\_mul\_significand\_keep:NNNNNw . . . . .  
     . . . . . 25591, 25596, 25598, 25614  
 \\_\_fp\_mul\_significand\_large\_-\_f:NwwNNNN . . . . .  
     . . . . . 25621, 25625, 25625  
 \\_\_fp\_mul\_significand\_o:nnnnNnnnn . . . . .  
     . . . . . 1081, 25589, 25591, 25591  
 \\_\_fp\_mul\_significand\_small\_-\_f:NNwwwN . . . . .  
     . . . . . 25619, 25636, 25636  
 \\_\_fp\_mul\_significand\_test\_f:NNN . . . . .  
     . . . . . 1082, 25593, 25616, 25616  
 \c\_fp\_myriad\_int . . . . . 22564,  
     26073, 26104, 26105, 26182, 26243  
 \\_\_fp\_neg\_sign:N . . . . .  
     . . . . . 1071, 22587, 22587, 25255, 25408  
 \\_\_fp\_not\_o:w 1058, 24077, 25160, 25160  
 \c\_fp\_one\_fixed\_t1 . . . . . 26067,  
     26687, 26900, 27097, 27124, 27907,  
     27974, 28079, 28556, 28566, 28607  
 \\_\_fp\_overflow:w . . . . . 982, 996,  
     998, 22602, 23036, 23041, 26778, 27211  
 \c\_fp\_overflowing\_fp . . . . .  
     . . . . . 22568, 28192, 28246  
 \\_\_fp\_pack:NNNNNw . . . . .  
     . . . . . 22751, 22754, 26079,  
     26089, 26092, 26095, 26098, 26101,  
     26162, 26165, 26168, 26171, 26174,  
     27062, 27065, 27068, 27071, 27074  
 \\_\_fp\_pack\_big:NNNNNNw . . . . .  
     . . . . . 22755, 22758,  
     25855, 25858, 25861, 25864, 25867,  
     25870, 25873, 25877, 26189, 26199,  
     26209, 26219, 26222, 26225, 26232  
 \\_\_fp\_pack\_Bigg:NNNNNNNw . . . . .  
     . . . . . 22760, 22763, 25704,  
     25707, 25710, 25722, 25725, 25728  
 \\_\_fp\_pack\_eight:wNNNNNNNN . . . . .  
     . . . . . 988, 1077, 22767, 22767,  
     25517, 25826, 26254, 27403, 27404  
 \\_\_fp\_pack\_twice\_four:wNNNNNNNN . . . . .  
     . . . . . 988, 22765, 22765, 23354, 23355,  
     25459, 25460, 26255, 26256, 26257,  
     26289, 26290, 26291, 26479, 26480,  
     26815, 26816, 26817, 27405, 27406,  
     27700, 27701, 27702, 27703, 28396  
 \\_\_fp\_parse:n . . . 1010, 1022, 1034,  
     1042, 1055, 1056, 1063, 1176, 1188,  
     23385, 23536, 24194, 24194, 24746,  
     24748, 24750, 24773, 24824, 24833,  
     . . . . . 24850, 24860, 25017, 25077, 26002,  
     28167, 28221, 28299, 28344, 28359,  
     28412, 28414, 28416, 28418, 28795  
 \\_\_fp\_parse\_after:ww . . . . .  
     . . . . . 24194, 24197, 24205, 24210  
 \\_\_fp\_parse\_apply\_binary:NwNwN . . . . .  
     . . . . . 1014,  
     1015, 1018, 1046, 24232, 24232, 24423  
 \\_\_fp\_parse\_apply\_binary\_chk:NN . . . . .  
     . . . . . 24232, 24237, 24249, 24263, 24276  
 \\_\_fp\_parse\_apply\_binary\_error:NNN . . . . .  
     . . . . . 24232, 24252, 24256  
 \\_\_fp\_parse\_apply\_comma:NwNwN . . . . .  
     . . . . . 1046, 24382, 24393, 24408  
 \\_\_fp\_parse\_apply\_compare:NwNNNNNwN . . . . .  
     . . . . . 24570, 24579  
 \\_\_fp\_parse\_apply\_compare\_error:NNNwN . . . . .  
     . . . . . 24591, 24594, 24599  
 \\_\_fp\_parse\_apply\_function:NNNwN . . . . .  
     . . . . . 1037, 24026, 24026, 24187  
 \\_\_fp\_parse\_apply Unary:NNNwN . . . . .  
     . . . . . 24031, 24031, 24063, 24178  
 \\_\_fp\_parse\_apply Unary\_chk:nNNNNw . . . . .  
     . . . . . 24042, 24043, 24046  
 \\_\_fp\_parse\_apply Unary\_chk:nNNNw . . . . .  
     . . . . . 24031  
 \\_\_fp\_parse\_apply Unary\_chk:NwNw . . . . .  
     . . . . . 24031, 24033, 24038  
 \\_\_fp\_parse\_apply Unary\_error:NNw . . . . .  
     . . . . . 24031, 24054, 24057, 26039  
 \\_\_fp\_parse\_apply Unary\_type:NNN . . . . .  
     . . . . . 24031, 24034, 24052  
 \\_\_fp\_parse\_caseless\_inf:N . . . . .  
     . . . . . 24144, 24144  
 \\_\_fp\_parse\_caseless\_infinity:N . . . . .  
     . . . . . 24144, 24145  
 \\_\_fp\_parse\_caseless\_nan:N . . . . .  
     . . . . . 24144, 24146  
 \\_\_fp\_parse\_compare:NNNNNN . . . . .  
     . . . . . 24511, 24512, 24514,  
     24516, 24519, 24532, 24540, 24601  
 \\_\_fp\_parse\_compare\_auxi:NNNNNNN . . . . .  
     . . . . . 24511, 24535, 24543, 24557  
 \\_\_fp\_parse\_compare\_auxii:NNNNN . . . . .  
     . . . . . 24511,  
     24548, 24549, 24550, 24551, 24555  
 \\_\_fp\_parse\_compare\_end:NNNNw . . . . .  
     . . . . . 24511, 24552, 24566  
 \\_\_fp\_parse\_continue:NwN . . . . .  
     . . . . . 1014, 1015,  
     1042, 24221, 24224, 24231, 24234,  
     24410, 24609, 25217, 25227, 25235  
 \\_\_fp\_parse\_continue\_compare:NNwNN . . . . .  
     . . . . . 24602, 24617

`\_fp_parse_digits_-:N . . . . .` [23403](#), [23421](#), [23422](#)  
`\_fp_parse_digits_i:N .` [23403](#), [23420](#)  
`\_fp_parse_digits_ii:N` [23403](#), [23419](#)  
`\_fp_parse_digits_iii:N` [23403](#), [23418](#)  
`\_fp_parse_digits_iv:N` [23403](#), [23417](#)  
`\_fp_parse_digits_v:N .` [23403](#), [23416](#)  
`\_fp_parse_digits_vi:N . . . . .` [23403](#), [23415](#), [23738](#), [23786](#)  
`\_fp_parse_digits_vii:N . . . . .` [1027](#), [23403](#), [23725](#), [23775](#)  
`\_fp_parse_excl_error: . . . . .` [24511](#), [24527](#), [24536](#)  
`\_fp_parse_expand:w . . . . .` [1018](#), [1019](#), [23400](#), [23400](#), [23402](#),  
[23412](#), [23452](#), [23512](#), [23556](#), [23565](#),  
[23568](#), [23572](#), [23609](#), [23643](#), [23681](#),  
[23683](#), [23702](#), [23704](#), [23726](#), [23743](#),  
[23756](#), [23776](#), [23806](#), [23834](#), [23850](#),  
[23861](#), [23884](#), [23913](#), [23923](#), [23930](#),  
[23944](#), [23960](#), [23980](#), [23991](#), [24073](#),  
[24096](#), [24108](#), [24183](#), [24192](#), [24200](#),  
[24213](#), [24331](#), [24377](#), [24401](#), [24427](#),  
[24475](#), [24495](#), [24564](#), [24577](#), [25213](#)  
`\_fp_parse_exponent:N` [1032](#), [23511](#),  
[23717](#), [23866](#), [23933](#), [23935](#), [23935](#)  
`\_fp_parse_exponent:Nw . . . . .` [23741](#), [23754](#), [23803](#),  
[23831](#), [23882](#), [23911](#), [23930](#), [23930](#)  
`\_fp_parse_exponent_aux:NN . . . . .` [23935](#), [23938](#), [23946](#)  
`\_fp_parse_exponent_body:N . . . . .` [23962](#), [23966](#), [23966](#)  
`\_fp_parse_exponent_digits:N . . . . .` [23970](#), [23982](#), [23982](#), [23986](#)  
`\_fp_parse_exponent_keep:N . . . . .` [23993](#)  
`\_fp_parse_exponent_keep:NTF . . . . .` [23973](#), [23993](#)  
`\_fp_parse_exponent_sign:N . . . . .` [23952](#), [23956](#), [23956](#), [23959](#)  
`\_fp_parse_function:NNN . . . . .` [23097](#), [23099](#), [23101](#), [23104](#),  
[24176](#), [24185](#), [24817](#), [24819](#), [27274](#),  
[27276](#), [27278](#), [27280](#), [28441](#), [28443](#)  
`\_fp_parse_function_all_fp_-o:nw . . .` [23231](#), [24619](#), [24619](#), [25115](#)  
`\_fp_parse_function_one_two:nnw . . . . .` [1160](#),  
[24631](#), [24631](#), [27868](#), [27874](#), [28510](#)  
`\_fp_parse_function_one_two_-aux:nnw . . . . .` [24631](#), [24635](#), [24645](#)  
`\_fp_parse_function_one_two_-auxii:nnw . . . . .` [24631](#), [24657](#), [24659](#)  
`\_fp_parse_function_one_two_-error_o:w . . . . .` [24631](#), [24634](#), [24637](#), [24654](#), [24662](#)  
`\_fp_parse_infix:NN . . . . .` [1020](#), [1024](#), [1040](#),  
[1045](#), [23451](#), [23621](#), [23660](#), [24136](#),  
[24151](#), [24173](#), [24289](#), [24292](#), [24375](#)  
`\_fp_parse_infix_!:_N . . . . .` [24511](#)  
`\_fp_parse_infix_&:Nw . . . . .` [24468](#)  
`\_fp_parse_infix_(:_N . . . . .` [24451](#)  
`\_fp_parse_infix_):N . . . . .` [24365](#)  
`\_fp_parse_infix_*:N . . . . .` [24453](#)  
`\_fp_parse_infix_+:N . . . . .` [1018](#), [23400](#), [24417](#)  
`\_fp_parse_infix_,:N . . . . .` [24382](#)  
`\_fp_parse_infix_-=N . . . . .` [24417](#)  
`\_fp_parse_infix_/_:N . . . . .` [24417](#)  
`\_fp_parse_infix_::N . . . . .` [24485](#), [25198](#)  
`\_fp_parse_infix_<:_N . . . . .` [24511](#)  
`\_fp_parse_infix_=:_N . . . . .` [24511](#)  
`\_fp_parse_infix_>:_N . . . . .` [24511](#)  
`\_fp_parse_infix_?:N . . . . .` [24485](#)  
`\_fp_parse_infix_(operation2):N . . . . .` [1018](#)  
`\_fp_parse_infix_~:_N . . . . .` [24417](#)  
`\_fp_parse_infix_after_operand:NwN . . . . .` [1024](#),  
[23504](#), [23582](#), [24080](#), [24287](#), [24287](#)  
`\_fp_parse_infix_after_paren:NN . . . . .` [24105](#), [24131](#), [24334](#), [24334](#)  
`\_fp_parse_infix_and:N . . . . .` [24417](#), [24484](#)  
`\_fp_parse_infix_check:NNN . . . . .` [24310](#), [24320](#), [24352](#)  
`\_fp_parse_infix_comma:w . . . . .` [1046](#), [24382](#), [24397](#), [24406](#)  
`\_fp_parse_infix_end:N . . . . .` [1042](#), [1046](#), [24201](#),  
[24206](#), [24214](#), [24363](#), [24363](#), [24364](#)  
`\_fp_parse_infix_juxt:N . . . . .` [1045](#), [24300](#), [24308](#), [24417](#)  
`\_fp_parse_infix_mark:NNN . . . . .` [24297](#), [24339](#), [24362](#), [24362](#)  
`\_fp_parse_infix_mul:N . . . . .` [1045](#), [1048](#), [24325](#),  
[24342](#), [24350](#), [24417](#), [24452](#), [24461](#)  
`\_fp_parse_infix_or:N . . . . .` [24417](#), [24483](#)  
`\_fp_parse_infix_|:Nw . . . . .` [24468](#)  
`\_fp_parse_large:N . . . . .` [1026](#), [23688](#), [23771](#), [23771](#)  
`\_fp_parse_large_leading:wwNN . . . . .` [1030](#), [23773](#), [23778](#), [23778](#)  
`\_fp_parse_large_round:NN . . . . .` [1030](#), [23814](#), [23886](#), [23886](#)  
`\_fp_parse_large_round_aux:wNN . . . . .` [23886](#), [23895](#), [23915](#)

```

\__fp_parse_large_round_test:Nn . . . . . 23886, 23899, 23904
\__fp_parse_large_trailing:wwNn . . . . . 1030, 23784, 23808, 23808
\__fp_parse_letters:N . . . . . 1024, 23597, 23611, 23626, 23638
\__fp_parse_lparen_after:NwN . . . . . 24086, 24088, 24098
\__fp_parse_o:n . . . . . 1010, 24194, 24207, 25015, 25016
\__fp_parse_one:Nw . . . . . 1013–
    1018, 1025, 1040, 1042, 23400,
    23423, 23423, 23665, 24025, 24227
\__fp_parse_one_digit:Nn . . . . . 1038, 23439, 23580, 23580
\__fp_parse_one_fp:Nn . . . . . 1020, 23431, 23447, 23447
\__fp_parse_one_other:Nn . . . . . 23442, 23588, 23588
\__fp_parse_one_register:Nn . . . . . 23434, 23502, 23502
\__fp_parse_one_register_aux:Nw . . . . . 23502, 23508, 23514
\__fp_parse_one_register_–
    auxii:wwwNw . . . . . 23502, 23519, 23528
\__fp_parse_one_register_dim:ww . . . . . 23502, 23522, 23534, 23537
\__fp_parse_one_register_int:www . . . . . 23502, 23524, 23535
\__fp_parse_one_register_–
    math>NNw . . . . . 23543, 23549, 23552, 23555
\__fp_parse_one_register_mu:www . . . . . 23502, 23523, 23532
\__fp_parse_one_register_–
    special:N . . . . . 23507, 23543, 23543
\__fp_parse_one_register_wd:Nw . . . . . 23543, 23571, 23574
\__fp_parse_one_register_wd:w . . . . . 23543, 23545, 23546, 23547, 23567
\__fp_parse_operand:Nw . . . . . 1013–
    1016, 1018, 1042, 1046, 23400,
    24069, 24071, 24092, 24094, 24183,
    24192, 24199, 24212, 24221, 24221,
    24400, 24426, 24494, 24577, 25212
\__fp_parse_pack_carry:w . . . . . 1029, 23758, 23766, 23769
\__fp_parse_pack_leading:NNNNNww . . . . . 23721, 23758, 23763, 23781
\__fp_parse_pack_trailing:NNNNNNww . . . . . 23731,
    23758, 23758, 23800, 23811, 23818
\__fp_parse_prefix:NNN . . . . . 23600, 23645, 23645
\__fp_parse_prefix_!Nw . . . . . 24059
\__fp_parse_prefix_(:Nw . . . . . 24086
\__fp_parse_prefix_):Nw . . . . . 24118
\__fp_parse_prefix_+:Nw . . . . . 24025
\__fp_parse_prefix_-:Nw . . . . . 24059
\__fp_parse_prefix_.:Nw . . . . . 24078
\__fp_parse_prefix_unknown:NNN . . . . .
    . . . . . 23645, 23648, 23653
\__fp_parse_return_semicolon:w . . . .
    . . . . . 23401, 23401, 23410, 23641,
    23848, 23859, 23942, 23974, 23989
\__fp_parse_round:Nw . . . . . 23102, 23108
\__fp_parse_round_after:wN . . . . . 1032,
    23863, 23863, 23868, 23877, 23918
\__fp_parse_round_loop:N . . . . .
    . . . . . 1032, 1033, 23836,
    23836, 23841, 23879, 23897, 23922
\__fp_parse_round_up:N . . . . .
    . . . . . 23836, 23844, 23852, 23856
\__fp_parse_small:N . . . . .
    . . . . . 1027, 23708, 23719, 23719
\__fp_parse_small_leading:wwNn . . . .
    . . . . . 1028, 23723, 23728, 23728, 23790
\__fp_parse_small_round:NN . . . . .
    . . . . . 23750, 23868, 23868, 23907
\__fp_parse_small_trailing:wwNn . . . .
    . . . . . 1028, 23736, 23745, 23745, 23822
\__fp_parse_strim_end:w . . . . .
    . . . . . 23694, 23700, 23704
\__fp_parse_strim_zeros:N . . . . .
    . . . . . 1026, 1038,
    23675, 23694, 23694, 23698, 24084
\__fp_parse_trim_end:w . . . . .
    . . . . . 23668, 23678, 23683
\__fp_parse_trim_zeros:N . . . . .
    . . . . . 23586, 23668, 23668, 23671
\__fp_parse_unary_function:NNN . . .
    . . . . . 24176,
    24176, 25245, 25247, 25249, 25251,
    26509, 26511, 26513, 27262, 27268
\__fp_parse_word:Nw . . . . .
    . . . . . 1024, 23594, 23611, 23611
\__fp_parse_word_abs:N . . . . . 25244, 25244
\__fp_parse_word_acos:N . . . . . 27254
\__fp_parse_word_acosd:N . . . . . 27254
\__fp_parse_word_acot:N . . . . . 27273, 27273
\__fp_parse_word_acotd:N . . . . . 27273, 27275
\__fp_parse_word_acsc:N . . . . . 27254
\__fp_parse_word_acscd:N . . . . . 27254
\__fp_parse_word_asec:N . . . . . 27254
\__fp_parse_word_asecd:N . . . . . 27254
\__fp_parse_word_asin:N . . . . . 27254
\__fp_parse_word_asind:N . . . . . 27254
\__fp_parse_word_atan:N . . . . . 27273, 27277
\__fp_parse_word_atand:N . . . . . 27273, 27279

```

`\_fp_parse_word_bp:N` ..... 24147  
`\_fp_parse_word_cc:N` ..... 24147  
`\_fp_parse_word_ceil:N` 23096, 23100  
`\_fp_parse_word_cm:N` ..... 24147  
`\_fp_parse_word_cos:N` ..... 27254  
`\_fp_parse_word_cosd:N` ..... 27254  
`\_fp_parse_word_cot:N` ..... 27254  
`\_fp_parse_word_cotd:N` ..... 27254  
`\_fp_parse_word_csc:N` ..... 27254  
`\_fp_parse_word_cscd:N` ..... 27254  
`\_fp_parse_word_dd:N` ..... 24147  
`\_fp_parse_word_deg:N` ..... 24133  
`\_fp_parse_word_em:N` ..... 24166  
`\_fp_parse_word_ex:N` ..... 24166  
`\_fp_parse_word_exp:N` . 26508, 26508  
`\_fp_parse_word_fact:N` 26508, 26512  
`\_fp_parse_word_false:N` ..... 24133  
`\_fp_parse_word_floor:N` 23096, 23098  
`\_fp_parse_word_in:N` ..... 24147  
`\_fp_parse_word_inf:N` .....  
..... 24133, 24144, 24145  
`\_fp_parse_word_ln:N` . 26508, 26510  
`\_fp_parse_word_logb:N` 25244, 25246  
`\_fp_parse_word_max:N` . 24816, 24816  
`\_fp_parse_word_min:N` . 24816, 24818  
`\_fp_parse_word_mm:N` ..... 24147  
`\_fp_parse_word_nan:N` . 24133, 24146  
`\_fp_parse_word_nc:N` ..... 24147  
`\_fp_parse_word_nd:N` ..... 24147  
`\_fp_parse_word_pc:N` ..... 24147  
`\_fp_parse_word_pi:N` ..... 24133  
`\_fp_parse_word_pt:N` ..... 24147  
`\_fp_parse_word_rand:N` 28440, 28440  
`\_fp_parse_word_randint:N` .....  
..... 28440, 28442  
`\_fp_parse_word_round:N` 23102, 23102  
`\_fp_parse_word_sec:N` ..... 27254  
`\_fp_parse_word_secd:N` ..... 27254  
`\_fp_parse_word_sign:N` 25244, 25248  
`\_fp_parse_word_sin:N` ..... 27254  
`\_fp_parse_word_sind:N` ..... 27254  
`\_fp_parse_word_sp:N` ..... 24147  
`\_fp_parse_word_sqrt:N` 25244, 25250  
`\_fp_parse_word_tan:N` ..... 27254  
`\_fp_parse_word_tand:N` ..... 27254  
`\_fp_parse_word_true:N` ..... 24133  
`\_fp_parse_word_trunc:N` 23096, 23096  
`\_fp_parse_zero:` .....  
... 1026, 23690, 23710, 23714, 23714  
`\_fp_pow_B:wwN` ..... 27044, 27079  
`\_fp_pow_C_neg:w` ..... 27082, 27099  
`\_fp_pow_C_overflow:w` .....  
..... 27087, 27094, 27115  
`\_fp_pow_C_pack:w` 27101, 27109, 27120  
`\_fp_pow_C_pos:w` ..... 27085, 27104  
`\_fp_pow_C_pos_loop:wnN` .....  
..... 27105, 27106, 27113  
`\_fp_pow_exponent:Nwnnnnnw` .....  
..... 27050, 27053, 27058  
`\_fp_pow_exponent:wnN` . 27042, 27047  
`\_fp_pow_neg:www` .....  
..... 1138, 26956, 27126, 27126  
`\_fp_pow_neg_aux:wNN` .....  
..... 1137, 27126, 27129, 27141  
`\_fp_pow_neg_case:w` .....  
..... 27128, 27149, 27149  
`\_fp_pow_neg_case_aux:nnnnn` .....  
..... 27149, 27153, 27159  
`\_fp_pow_neg_case_aux:Nnnw` .....  
..... 1138, 27149, 27165, 27169  
`\_fp_pow_normal_o:ww` .....  
..... 1133, 26961, 26993, 26993  
`\_fp_pow_npos_aux:NNnww` .....  
..... 27027, 27031, 27037, 27037  
`\_fp_pow_npos_o:Nww` .....  
..... 1135, 27004, 27021, 27021  
`\_fp_pow_zero_or_inf:ww` .....  
..... 1133, 26963, 26970, 26970  
`\c_fp_prec_and_int` ... 23385, 24448  
`\c_fp_prec_colon_int` .....  
..... 23385, 24506, 25212  
`\c_fp_prec_comma_int` .....  
..... 1039, 23385, 23459,  
24092, 24120, 24386, 24391, 24400  
`\c_fp_prec_comp_int` .....  
..... 23385, 24534, 24577  
`\c_fp_prec_end_int` ... 1042, 1046,  
23385, 23461, 24199, 24212, 24369  
`\c_fp_prec_func_int` .....  
... 1039, 23385, 24091, 24183, 24192  
`\c_fp_prec_hat_int` ... 23385, 24436  
`\c_fp_prec_hatii_int` . 23385, 24436  
`\c_fp_prec_int` .....  
22561, 22794, 22855, 22882, 23323,  
26796, 27161, 27164, 28264, 28266,  
28272, 28323, 28521, 28560, 28611  
`\c_fp_prec_juxt_int` .. 23385, 24438  
`\c_fp_prec_not_int` .....  
..... 1038, 23385, 24076, 24077  
`\c_fp_prec_or_int` ... 23385, 24450  
`\c_fp_prec_plus_int` .....  
..... 1013, 23385, 24444, 24446  
`\c_fp_prec_quest_int` .....  
..... 23385, 24489, 24504  
`\c_fp_prec_times_int` .....  
..... 23385, 24440, 24442  
`\c_fp_prec_tuple_int` .....  
... 1039, 23385, 23460, 24094, 24122

`\_fp_rand_myriads:n . . . . .`  
`1181, 1182, 28476, 28476, 28493, 28579`  
`\_fp_rand_myriads_get:w . . . . .`  
`28476, 28481, 28486`  
`\_fp_rand_myriads_loop:w . . . . .`  
`28476, 28477, 28478, 28484`  
`\_fp_rand_o:Nw . . . . .`  
`28441, 28448, 28454, 28487, 28487`  
`\_fp_rand_o:w . . 28487, 28491, 28501`  
`\_fp_randinat_wide_aux:w . . 28649`  
`\_fp_randinat_wide_auxii:w . . 28649`  
`\_fp_randint:n . 28711, 28714, 28716`  
`\_fp_randint:ww . . . . .`  
`28617, 28621, 28626, 28631, 28721`  
`\_fp_randint_auxi_o:ww . . . . .`  
`28508, 28535, 28543`  
`\_fp_randint_auxii:wn . . . . .`  
`28508, 28546, 28547, 28549`  
`\_fp_randint_auxiii_o:ww . . . . .`  
`28508, 28547, 28571`  
`\_fp_randint_auxiv_o:ww . . . . .`  
`28508, 28582, 28586`  
`\_fp_randint_auxvx_o:w . . . . .`  
`28508, 28584, 28594, 28596`  
`\_fp_randint_badarg:w . . . . .`  
`1182, 28508, 28515, 28531, 28532`  
`\_fp_randint_default:w . . . . .`  
`28508, 28512, 28514`  
`\_fp_randint_o:Nw . . . . .`  
`28443, 28454, 28508, 28508`  
`\_fp_randint_o:w 28508, 28512, 28528`  
`\_fp_randint_split_aux:w . . . . .`  
`28649, 28672, 28678`  
`\_fp_randint_split_o:Nw . . . . .`  
`1185, 28649,`  
`28654, 28657, 28660, 28662, 28667`  
`\_fp_randint_wide_aux:w . . . . .`  
`1185, 28652, 28683`  
`\_fp_randint_wide_auxii:w . . . . .`  
`28685, 28694`  
`\_fp_reverse_args:Nww . . . . .`  
`1166, 1167, 22537, 22537,`  
`27854, 27929, 28042, 28108, 28605`  
`\_fp_round:NNN . . . . .`  
`1002, 1004, 1082, 1098,`  
`23112, 23179, 23182, 25376, 25387,`  
`25631, 25643, 25785, 25796, 25980`  
`\_fp_round:Nwn . . . . .`  
`23240, 23293, 23295, 23310, 28371`  
`\_fp_round:Nww . . . . .`  
`23241, 23262, 23293, 23293`  
`\_fp_round:Nwww . 23242, 23256, 23256`  
`\_fp_round_aux_o:Nw . . . . .`  
`23229, 23233, 23235`  
`\_fp_round_digit:Nw . . . . .`  
`990, 1004, 1081, 1082,`  
`1098, 22812, 23196, 23196, 25390,`  
`25533, 25634, 25646, 25799, 25985`  
`\_fp_round_name_from_cs:N 23232,`  
`23252, 23278, 23282, 23282, 23305`  
`\_fp_round_neg:NNN . . . . .`  
`1002, 1005, 1078,`  
`23207, 23228, 25495, 25510, 25528`  
`\_fp_round_no_arg_o:Nw . . . . .`  
`23239, 23246, 23246`  
`\_fp_round_normal:NnnwNNnn . . . . .`  
`23293, 23324, 23326`  
`\_fp_round_normal:NNwNnn . . . . .`  
`23293, 23328, 23348`  
`\_fp_round_normal:NwNNnw . . . . .`  
`23293, 23313, 23321`  
`\_fp_round_normal_end:wwNnn . . . . .`  
`23293, 23356, 23359`  
`\_fp_round_o:Nw . . . . .`  
`23097, 23099, 23101, 23105, 23229,`  
`\_fp_round_pack:Nw . . . . .`  
`23293, 23332, 23346`  
`\_fp_round_return_one: . . . . .`  
`1002, 23112, 23118,`  
`23128, 23136, 23140, 23149, 23153,`  
`23162, 23169, 23173, 23211, 23222`  
`\_fp_round_s:NNNw . . . . .`  
`1002, 1032, 23180, 23180, 23872, 23890`  
`\_fp_round_special:NwwNnn . . . . .`  
`23293, 23351, 23364`  
`\_fp_round_special_aux:Nw . . . . .`  
`23293, 23370, 23377`  
`\_fp_round_to_nearest:NNN 1005,`  
`1006, 23105, 23108, 23112, 23133,`  
`23179, 23216, 23248, 23258, 28371`  
`\_fp_round_to_nearest_neg:NNN . . . . .`  
`23207, 23216, 23228`  
`\_fp_round_to_nearest_ninf:NNN . . . . .`  
`1006, 23112, 23146, 23227`  
`\_fp_round_to_nearest_ninf_-`  
`neg:NNN . . . . .`  
`23207, 23217`  
`\_fp_round_to_nearest_pinf:NNN . . . . .`  
`1006, 23112, 23166, 23218`  
`\_fp_round_to_nearest_pinf_-`  
`neg:NNN . . . . .`  
`23207, 23226`  
`\_fp_round_to_nearest_zero:NNN . . . . .`  
`1006, 23112, 23159`  
`\_fp_round_to_nearest_zero_-`  
`neg:NNN . . . . .`  
`23207, 23219`  
`\_fp_round_to_ninf:NNN . . . . .`  
`23099, 23112, 23114, 23215, 23286`  
`\_fp_round_to_ninf_neg:NNN . . . . .`  
`23207, 23207`

\\_fp\_round\_to\_pinf:NNN . . . . .  
.. 23101, 23112, 23124, 23207, 23288  
\\_fp\_round\_to\_pinf\_neg:NNN . . . . .  
..... 23207, 23215  
\\_fp\_round\_to\_zero:NNN . . . . .  
..... 23097, 23112, 23123, 23284  
\\_fp\_round\_to\_zero\_neg:NNN . . . . .  
..... 23207, 23208  
\\_fp\_rrrot:www .. 22538, 22538, 27975  
\\_fp\_sanitize:Nw . . . . . 1073, 1076,  
1081, 1084, 1092, 1140, 1156, 1163,  
1182, 22596, 22596, 22608, 23362,  
23380, 25319, 25413, 25585, 25666,  
25814, 26539, 26782, 27023, 27215,  
27816, 27860, 27987, 28503, 28598  
\\_fp\_sanitize:wN . . . . .  
1023, 1027, 22596, 22608, 23585, 24083  
\\_fp\_sanitize\_zero:w . . . . .  
..... 22596, 22604, 22609  
\\_fp\_sec\_o:w . . . . . 27326, 27326  
\\_fp\_set\_sign\_o:w . . . . . 24076,  
25245, 26016, 26017, 26017, 26038  
\\_fp\_show>NN . . . . .  
..... 24778, 24778, 24780, 24782  
\\_fp\_show\_validate:w . . . . .  
..... 24778, 24788, 24794  
\\_fp\_sign\_aux\_o:w . . . . .  
..... 26005, 26009, 26010, 26015  
\\_fp\_sign\_o:w .. 25249, 26005, 26005  
\\_fp\_sin\_o:w . . . . .  
..... 994, 1037, 1165, 27281, 27281  
\\_fp\_sin\_series\_aux\_o:NNnwww . . . . .  
..... 27768, 27772, 27783  
\\_fp\_sin\_series\_o:NNwww . . . . .  
..... 1143, 1157, 27287,  
27302, 27317, 27332, 27768, 27768  
\\_fp\_small\_int:wTF . . . . .  
... 1139, 22864, 22864, 23295, 27202  
\\_fp\_small\_int\_normal:NnwTF . . . . .  
..... 22864, 22868, 22880  
\\_fp\_small\_int\_test:NnnwNTF . 22864  
\\_fp\_small\_int\_test:NnnwNw . . . . .  
..... 22883, 22886  
\\_fp\_small\_int\_true:wTF . . . . .  
.. 22864, 22867, 22872, 22879, 22889  
\\_fp\_sqrt\_auxi\_o:NNNNwnnN . . . . .  
..... 25836, 25844, 25844  
\\_fp\_sqrt\_auxii\_o:NnnnnnnmN . . . . .  
..... 1094, 1096,  
25846, 25850, 25850, 25930, 25942  
\\_fp\_sqrt\_auxiii\_o:nnnnnnnn . . . . .  
..... 25847, 25885, 25885, 25931  
\\_fp\_sqrt\_auxiv\_o:NNNNNw . . . . .  
..... 25885, 25889, 25906  
\\_fp\_sqrt\_auxix\_o:wnwnw . . . . .  
..... 25919, 25921, 25928  
\\_fp\_sqrt\_auxv\_o:NNNNNw . . . . .  
..... 25885, 25893, 25908  
\\_fp\_sqrt\_auxvi\_o:NNNNNw . . . . .  
..... 25885, 25897, 25910  
\\_fp\_sqrt\_auxvii\_o:NNNNNw . . . . .  
..... 25885, 25900, 25912  
\\_fp\_sqrt\_auxviii\_o:nnnnnnn . . . . .  
..... 25907,  
25909, 25911, 25917, 25919, 25919  
\\_fp\_sqrt\_auxxx\_o:Nnnnnnnn . . . . .  
..... 25915, 25933, 25933  
\\_fp\_sqrt\_auxxi\_o:wwnnN . . . . .  
..... 25933, 25935, 25940  
\\_fp\_sqrt\_auxxii\_o:nnnnnnnnw . . . . .  
..... 25943, 25947, 25947  
\\_fp\_sqrt\_auxxiii\_o:w . . . . .  
..... 25947, 25954, 25967  
\\_fp\_sqrt\_auxxiv\_o:wnnnnnnnN . . . . .  
.. 25959, 25962, 25970, 25972, 25972  
\\_fp\_sqrt\_Newton\_o:wwn . . . . . 1093,  
25821, 25832, 25833, 25833, 25840  
\\_fp\_sqrt\_npos\_auxi\_o:wwnnN . . . . .  
..... 25812, 25818, 25823  
\\_fp\_sqrt\_npos\_auxii\_o:wNNNNNNNN . . . . .  
..... 25812, 25827, 25831  
\\_fp\_sqrt\_npos\_o:w . . . . .  
..... 25809, 25812, 25812  
\\_fp\_sqrt\_o:w .. 25251, 25802, 25802  
\\_fp\_step>NNnnnn . . . . .  
..... 25082, 25085, 25092, 25101  
\\_fp\_step:NnnnnN . . . . . 25012,  
25048, 25049, 25065, 25076, 25081  
\\_fp\_step:wwwN . 25012, 25014, 25030  
\\_fp\_step\_fp:wwwN 25012, 25035, 25043  
\\_fp\_str\_if\_eq:nn . 22901, 22901,  
23997, 24009, 24295, 24337, 26996  
\\_fp\_sub\_back\_far\_o:NnnnnnnnN . . . . .  
..... 1077, 25422, 25468, 25468  
\\_fp\_sub\_back\_near\_after:wNNNNw . . . . .  
..... 25428, 25430, 25437, 25506  
\\_fp\_sub\_back\_near\_o:nnnnnnnnN . . . . .  
..... 1076, 25418, 25428, 25428  
\\_fp\_sub\_back\_near\_pack:NNNNNNw . . . . .  
..... 25428, 25432, 25435, 25508  
\\_fp\_sub\_back\_not\_far\_o:wwwNN . . . . .  
..... 25483, 25503, 25503  
\\_fp\_sub\_back Quite\_far\_ii:NN . . . . .  
..... 25487, 25489, 25493  
\\_fp\_sub\_back Quite\_far\_o:wwNN . . . . .  
..... 25481, 25487, 25487  
\\_fp\_sub\_back\_shift:wnnnn . . . . .  
..... 1077, 25440, 25444, 25444

```

\__fp_sub_back_shift_ii:ww . . .
    ..... 25444, 25446, 25449
\__fp_sub_back_shift_iii:NNNNNNNNw
    ..... 25444, 25454, 25457, 25466
\__fp_sub_back_shift_iv:nnnw . .
    ..... 25444, 25461, 25467
\__fp_sub_back_very_far_ii-
    o:nnNwwNN . . 25515, 25518, 25522
\__fp_sub_back_very_far_o:wwwNN
    ..... 25482, 25515, 25515
\__fp_sub_eq_o:Nnwnw . . .
    ..... 25393, 25396, 25404
\__fp_sub_npos_i:o:Nnwnw . . .
    ... 1075, 25398, 25407, 25411, 25411
\__fp_sub_npos_ii:o:Nnwnw . . .
    ..... 25393, 25400, 25405
\__fp_sub_npos_o:NnwNnw . . .
    ..... 1075, 25313, 25393, 25393
\__fp_tan_o:w . . .
    ... 27341, 27341
\__fp_tan_series_aux_o:Nnwww . .
    ..... 27822, 27826, 27835
\__fp_tan_series_o:NNwww . .
    ... 1145, 27348, 27363, 27822
\__fp_ternary:NwwN . . .
    ... 1058, 24504, 25196, 25196
\__fp_ternary_auxi:NwwN . . .
    ... 1058, 1068, 25196, 25205, 25225
\__fp_ternary_auxii:NwwN . . .
    ... 1058, 1068, 24506, 25196, 25203, 25233
\__fp_tmp:w . . .
    ... 990, 1047, 22806, 22816, 22817, 22818, 22819, 22820, 22821, 22822, 22823, 22824, 22825, 22826, 22827, 22828, 22829, 22830, 22831, 22907, 22909, 23403, 23415, 23416, 23417, 23418, 23419, 23420, 23421, 23479, 23501, 24059, 24076, 24077, 24133, 24138, 24139, 24140, 24141, 24142, 24143, 24147, 24155, 24156, 24157, 24158, 24159, 24160, 24161, 24162, 24163, 24164, 24165, 24365, 24381, 24382, 24405, 24417, 24435, 24437, 24439, 24441, 24443, 24445, 24447, 24449, 24453, 24467, 24468, 24483, 24484, 24485, 24503, 24505, 26048, 26062, 26063
\__fp_to_decimal:w . . .
    ... 28226, 28236, 28236, 28353, 28370, 28857
\__fp_to_decimal_dispatch:w . .
    ... 1170, 1173, 1174, 25075, 28216, 28220, 28223, 28223, 28235
\__fp_to_decimal_huge:wnnnn . .
    ..... 28236, 28271, 28293
\__fp_to_decimal_large:Nnnw . .
    ..... 28236, 28267, 28284
\__fp_to_decimal_normal:wnnnnn . .
    ..... 28236, 28241, 28259, 28324
\__fp_to_decimal_recover:w . .
    ..... 28223, 28226, 28229
\__fp_to_dim:w . .
    ... 28338, 28348, 28353
\__fp_to_dim_dispatch:w . .
    ... 1173, 28338, 28339, 28343, 28346
\__fp_to_dim_recover:w . .
    ..... 28338, 28348, 28351
\__fp_to_int:w . .
    ... 1174, 28363, 28368
\__fp_to_int_dispatch:w . .
    ..... 28354, 28354, 28358, 28361
\__fp_to_int_recover:w . .
    ..... 28354, 28363, 28366
\__fp_to_scientific:w . .
    ... 1171, 28172, 28182, 28182
\__fp_to_scientific_dispatch:w . .
    ... 1169, 1173, 28162, 28166, 28169, 28169, 28181
\__fp_to_scientific_normal:wnnnnn . .
    ..... 28182, 28187, 28205
\__fp_to_scientific_normal:wNw . .
    ..... 28182, 28208, 28213
\__fp_to_scientific_recover:w . .
    ..... 28169, 28172, 28175
\__fp_to_tl:w . .
    ... 28302, 28310, 28310, 28865
\__fp_to_tl_dispatch:w . .
    ... 1168, 1172, 28294, 28298, 28301, 28301, 28309, 28434
\__fp_to_tl_normal:nnnnn . .
    ..... 28310, 28315, 28320
\__fp_to_tl_recover:w . .
    ... 28301, 28302, 28303
\__fp_to_tl_scientific:wnnnnn . .
    ..... 28310, 28325, 28328
\__fp_to_tl_scientific:wNw . .
    ..... 28310, 28331, 28336
\c_fp_trailing_shift_int . .
    ... 22751, 26080, 26102, 26175, 27075, 27714, 27751
\__fp_trap_division_by_zero-
    set:N . .
    ... 22979, 22980, 22982, 22984, 22985
\__fp_trap_division_by_zero_set-
    error: . .
    ... 22979, 22979
\__fp_trap_division_by_zero_set-
    flag: . .
    ... 22979, 22981
\__fp_trap_division_by_zero_set-
    none: . .
    ... 22979, 22983
\__fp_trap_invalid_operation-
    set:N . .
    ... 22945, 22946, 22948, 22950, 22951

```

\\_\_fp\_trap\_invalid\_operation\_-  
     set\_error: ..... [22945](#), 22945  
 \\_\_fp\_trap\_invalid\_operation\_-  
     set\_flag: ..... [22945](#), 22947  
 \\_\_fp\_trap\_invalid\_operation\_-  
     set\_none: ..... [22945](#), 22949  
 \\_\_fp\_trap\_overflow\_set:N .....  
     .. [23005](#), 23006, 23008, 23010, 23011  
 \\_\_fp\_trap\_overflow\_set:NnNn .....  
     .. [23005](#), 23012, 23020, 23021  
 \\_\_fp\_trap\_overflow\_set\_error: ..  
     ..... [23005](#), 23005  
 \\_\_fp\_trap\_overflow\_set\_flag: ..  
     ..... [23005](#), 23007  
 \\_\_fp\_trap\_overflow\_set\_none: ..  
     ..... [23005](#), 23009  
 \\_\_fp\_trap\_underflow\_set:N .....  
     .. [23005](#), 23014, 23016, 23018, 23019  
 \\_\_fp\_trap\_underflow\_set\_error: ..  
     ..... [23005](#), 23013  
 \\_\_fp\_trap\_underflow\_set\_flag: ..  
     ..... [23005](#), 23015  
 \\_\_fp\_trap\_underflow\_set\_none: ..  
     ..... [23005](#), 23017  
 \\_\_fp\_trig:NNNNNwn .....  
     ..... [27287](#), 27302, 27317,  
         27332, 27347, 27362, 27379, 27379  
 \c\_fp\_trig\_intarray ..... [1153](#),  
     [27440](#), 27670, 27673, 27676, 27679,  
     27682, 27685, 27688, 27691, 27694  
 \\_\_fp\_trig\_large:ww .....  
     ..... [27387](#), [27654](#), 27654  
 \\_\_fp\_trig\_large\_auxi:w .....  
     ..... [27654](#), 27656, 27661  
 \\_\_fp\_trig\_large\_auxii:w .....  
     ..... [1153](#), [27654](#), 27664, 27698  
 \\_\_fp\_trig\_large\_auxiii:w .. [1153](#),  
     [27654](#), 27672, 27675, 27678, 27681,  
     27684, 27687, 27690, 27693, 27706  
 \\_\_fp\_trig\_large\_auxix:Nw .....  
     ..... [27727](#), 27737, 27740, 27744  
 \\_\_fp\_trig\_large\_auxv:www .....  
     ..... [27704](#), [27707](#), 27707  
 \\_\_fp\_trig\_large\_auxvi:wnnnnnnnn .....  
     ..... [27707](#), 27713, 27718  
 \\_\_fp\_trig\_large\_auxvii:w .....  
     ..... [27710](#), [27727](#), 27727  
 \\_\_fp\_trig\_large\_auxviii:w ... [27727](#)  
 \\_\_fp\_trig\_large\_auxviii:ww .....  
     ..... [27729](#), 27733  
 \\_\_fp\_trig\_large\_auxx:wNNNNN ..  
     ..... [27727](#), 27750, 27754  
 \\_\_fp\_trig\_large\_auxxi:w .....  
     ..... [27727](#), 27747, 27761  
  
 \\_\_fp\_trig\_large\_pack:NNNNNw ...  
     ..... [27707](#), 27720, 27725, 27756  
 \\_\_fp\_trig\_small:ww .. [1147](#), 1155,  
     27389, [27393](#), 27393, 27399, 27766  
 \\_\_fp\_trigd\_large:ww .....  
     ..... [27387](#), [27401](#), 27401  
 \\_\_fp\_trigd\_large\_auxi:nnnnwNNNN .....  
     ..... [27401](#), 27407, 27413  
 \\_\_fp\_trigd\_large\_auxii:wNw .....  
     ..... [27401](#), 27415, 27421  
 \\_\_fp\_trigd\_large\_auxiii:www .....  
     ..... [27401](#), 27424, 27428  
 \\_\_fp\_trigd\_small:ww .....  
     ... [1147](#), 27389, [27395](#), 27395, 27438  
 \\_\_fp\_trim\_zeros:w .....  
     ... [28153](#), 28153, 28277, 28286, 28337  
 \\_\_fp\_trim\_zeros\_dot:w .....  
     ..... [28153](#), 28156, 28159  
 \\_\_fp\_trim\_zeros\_end:w .....  
     ..... [28153](#), 28159, 28160  
 \\_\_fp\_trim\_zeros\_loop:w .....  
     ..... [28153](#), 28155, 28156, 28158  
 \\_\_fp\_tuple\_ 25186, 25187, 25190, 25191  
 \\_\_fp\_tuple\_&\_o:ww ..... [25169](#)  
 \\_\_fp\_tuple\_&\_tuple\_o:ww ..... [25169](#)  
 \\_\_fp\_tuple\_\*\_o:ww ..... [26042](#)  
 \\_\_fp\_tuple\_+\_tuple\_o:ww ..... [26048](#)  
 \\_\_fp\_tuple\_-\_tuple\_o:ww ..... [26048](#)  
 \\_\_fp\_tuple\_/\_o:ww ..... [26042](#)  
 \\_\_fp\_tuple\_chk:w .....  
     ..... [983](#), [22654](#), 22655, 22656,  
         22658, 22660, 22661, 22738, 22741,  
         24414, 24626, 24641, 24666, 24669,  
         24685, 24686, 24689, 24910, 24911,  
         26051, 26052, 26058, 26059, 28132  
 \\_\_fp\_tuple\_compare\_back:ww ....  
     ..... [24907](#), 24908  
 \\_\_fp\_tuple\_compare\_back\_loop:w ..  
     ..... [24907](#), 24917, 24925, 24934  
 \\_\_fp\_tuple\_compare\_back\_-  
     tuple:ww ..... [24907](#), 24909  
 \\_\_fp\_tuple\_convert:Nw .....  
     ... [28132](#), 28132, 28181, 28235, 28309  
 \\_\_fp\_tuple\_convert\_end:w .....  
     ..... [28132](#), 28137, 28141, 28151  
 \\_\_fp\_tuple\_convert\_loop:nNw ...  
     ..... [28132](#), 28140, 28145, 28148  
 \\_\_fp\_tuple\_count:w .....  
     ..... [22659](#), 22660, 22661  
 \\_\_fp\_tuple\_count\_loop:Nw .....  
     ..... [22659](#), 22664, 22668, 22669  
 \\_\_fp\_tuple\_map\_loop\_o:nw .....  
     ..... [24666](#), 24672, 24677, 24682

```

\__fp_tuple_map_o:nw . . . . . 24666,
    24666, 26035, 26043, 26045, 26047
\__fp_tuple_mapthread_loop_o:nw . . . . .
    . . . . . 24684, 24692, 24698, 24704
\__fp_tuple_mapthread_o:nww . . . . .
    . . . . . 24684, 24684, 26056
\__fp_tuple_not_o:w . . . . . 25160, 25168
\__fp_tuple_set_sign_aux_o:Nnw . . . .
    . . . . . 26027, 26030, 26034
\__fp_tuple_set_sign_aux_o:w . . . .
    . . . . . 26027, 26035, 26036
\__fp_tuple_set_sign_o:w 26027, 26027
\__fp_tuple_to_decimal:w 28223, 28234
\__fp_tuple_to_scientific:w . . . .
    . . . . . 28169, 28180
\__fp_tuple_to_tl:w . . . . . 28301, 28308
\__fp_tuple_|_o:ww . . . . . 25169
\__fp_tuple_|_tuple_o:ww . . . . . 25169
\__fp_type_from_scan:N . . . . . 984,
    22683, 22683, 24240, 24242, 24266,
    24268, 24279, 24281, 24871, 24873
\__fp_type_from_scan:w . . . . .
    . . . . . 22683, 22692, 22697
\__fp_type_from_scan_other:N . . .
    . . . . . 22683, 22687, 22690, 22707, 22725
\__fp_underflow:w . . . . . 982,
    996, 998, 22603, 23036, 23042, 26779
\__fp_use_i:ww . . . . .
    . . . . . 1111, 1165, 22539, 22539, 26292, 28061
\__fp_use_i:www . . . . . 22539, 22540
\__fp_use_i_delimit_by_s_stop:nw . . .
    . . . . . 22550, 22550, 24839, 25201
\__fp_use_i_until_s:nw . . . .
    . . . . . 1155, 22534, 22535,
    22583, 22593, 22856, 27431, 27709,
    27715, 27746, 28521, 28592, 28803
\__fp_use_ii_until_s:nnw . . . .
    . . . . . 22534, 22536, 22581, 22592
\__fp_use_none_stop_f:n . . . .
    . . . . . 22531, 22531, 26457, 26458, 26459
\__fp_use_none_until_s:w . . . . . 22534,
    22534, 25838, 27135, 28056, 28059
\__fp_use_s:n . . . . . 22532, 22532
\__fp_use_s:nn . . . . . 22532, 22533
\__fp_zero_fp:N . . . . .
    . . . . . 22574, 22574, 23020, 23368
\__fp_|_o:ww . . . . . 1058, 25169
\__fp_|_tuple_o:ww . . . . . 25169
fpararray commands:
\fparray_count:N . . . . . 261, 28763,
    28763, 28768, 28775, 28786, 28842
\fparray_gset:Nnn . . . . .
    . . . . . 261, 1189, 28788, 28788, 28797
\fparray_gzero:N . . . . .
    . . . . . 261, 28839, 28839, 28851
\fparray_item:Nn . . . . .
    . . . . . 261, 1189, 28852, 28852, 28859
\fparray_item_to_tl:Nn . . . .
    . . . . . 261, 28852, 28860, 28867
\fparray_new:Nn . . . . .
    . . . . . 261, 28736, 28736, 28748
\futurelet . . . . . 277

```

**G**

```

\gdef . . . . . 278
get commands:
    get_luadata . . . . . 12033
\GetIdInfo . . . . . 10, 11532
\gleaders . . . . . 828
\global . . . . . 179, 279
\globaldefs . . . . . 280
\glueexpr . . . . . 534
\glueshrink . . . . . 535
\glueshrinkorder . . . . . 536
\gluestretch . . . . . 537
\gluestretchorder . . . . . 538
\gluetomu . . . . . 539
group commands:
\group_align_safe_begin/end: 437, 583
\group_align_safe_begin: . . . .
    . . . . . 71, 576, 686, 691, 3799, 4210,
    8569, 8773, 8775, 12402, 13023,
    19500, 19521, 19553, 20617, 20761,
    30220, 30657, 32287, 35423, 37318
\group_align_safe_end: . . . . . 71,
    686, 691, 3802, 4220, 8571, 8773,
    8778, 12423, 13006, 19509, 19518,
    19558, 19564, 20620, 20761, 30232,
    30670, 32298, 35426, 35430, 37326
\group_begin: . . . . . 13, 682, 1342,
    1359, 1458, 1459, 2199, 2202, 2205,
    2614, 3085, 3276, 3439, 3476, 3755,
    3798, 3805, 3878, 4043, 4226, 4294,
    4673, 4765, 5089, 5600, 5934, 6175,
    6276, 6706, 7083, 7363, 7613, 7639,
    7651, 7661, 7670, 7851, 7884, 8005,
    8773, 8818, 8976, 9072, 9376, 9400,
    9416, 9481, 10341, 10560, 10606,
    10868, 11009, 11539, 12139, 12276,
    12498, 12511, 13263, 13290, 13570,
    13593, 14080, 14190, 14243, 14538,
    14586, 14632, 14639, 14968, 15150,
    16706, 16743, 18850, 18856, 18903,
    19107, 19125, 19149, 19237, 19256,
    19636, 20598, 20737, 20781, 21868,
    25169, 28985, 29207, 29485, 29684,
    29721, 29824, 29885, 30178, 31858,
```

- 31891, 32534, 32621, 33030, 34839,  
 35014, 35556, 35644, 35686, 37303  
`\c_group_begin_token` . . . . .  
 . . . . . 107, 194, 314, 455, 701, 875,  
 3843, 4398, 12869, 12907, 19107,  
 19131, 29941, 30305, 33073, 33079,  
 33093, 33099, 33177, 33183, 33198,  
 33204, 35482, 35483, 35490, 37341  
`\group_end:` . . . . .  
 . . . . . 13, 14, 559, 804, 1196, 1199,  
 1342, 1458, 1460, 2199, 2202, 2208,  
 2623, 3088, 3279, 3443, 3484, 3693,  
 3768, 3803, 3824, 3885, 4067, 4216,  
 4316, 4685, 4779, 5122, 5130, 5613,  
 5938, 6235, 6283, 6290, 6298, 6710,  
 6711, 7120, 7427, 7618, 7646, 7734,  
 7878, 7924, 8006, 8007, 8777, 8837,  
 8993, 9100, 9395, 9408, 9427, 9635,  
 10347, 10564, 10635, 10891, 11027,  
 11542, 12142, 12298, 12348, 12501,  
 12515, 13281, 13313, 13575, 13598,  
 14090, 14203, 14246, 14551, 14598,  
 14657, 14719, 15149, 15281, 16715,  
 16753, 16758, 18858, 18865, 18983,  
 19124, 19128, 19156, 19255, 19304,  
 19660, 20736, 20762, 20821, 21882,  
 25193, 28989, 29240, 29694, 29695,  
 29789, 29859, 29901, 30198, 31884,  
 31917, 32538, 32891, 33036, 34840,  
 35019, 35559, 35649, 35701, 37307  
`\c_group_end_token` . . . . . 314, 875,  
 3846, 19107, 19136, 29942, 30311,  
 33087, 33192, 35486, 35494, 37342  
`\group_insert_after:N` . . . . .  
 . . . . . 14, 1464, 1464, 4681,  
 34847, 35486, 35487, 35520, 35803  
`\group_log_list:` . . . . . 14, 2211, 2213  
`\group_show_list:` . . . . . 14, 2211, 2211  
groups commands:  
`.groups:n` . . . . . 225, 21335
- H**
- `\H` . . . . . 86, 30140,  
 32664, 32811, 32812, 32839, 32840  
`\halign` . . . . . 281  
`\hangafter` . . . . . 282  
`\hangindent` . . . . . 283  
`\hbadness` . . . . . 284  
`\hbox` . . . . . 285  
hbox commands:  
`\hbox:n` . . . . . 273,  
 277, 33044, 33044, 33271, 33567, 34623  
`\hbox_gset:Nn` . . . . . 277,  
 33046, 33051, 33057, 33238, 33361,
- 33405, 33425, 33445, 33462, 33483,  
 33512, 33523, 33681, 34112, 36738  
`\hbox_gset:Nw` . . . . .  
 . . . . . 278, 33070, 33076, 33083, 33754  
`\hbox_gset_end:` . . . . .  
 . . . . . 278, 33070, 33089, 33757  
`\hbox_gset_to_wd:Nnn` . . . . .  
 . . . . . 278, 33058, 33063, 33069  
`\hbox_gset_to_wd:Nnw` . . . . .  
 . . . . . 278, 33090, 33096, 33103  
`\hbox_overlap_center:n` . . . . .  
 . . . . . 278, 33114, 33114  
`\hbox_overlap_left:n` 278, 33114, 33116  
`\hbox_overlap_right:n` . . . . .  
 . . . . . 278, 33114, 33118  
`\hbox_set:Nn` . . . . .  
 . . . . . 273, 277, 278, 291, 33046, 33046,  
 33056, 33235, 33267, 33268, 33355,  
 33402, 33422, 33442, 33459, 33480,  
 33509, 33517, 33540, 33668, 34109,  
 34132, 34389, 34476, 34755, 36735,  
 36748, 36756, 36764, 36773, 36782,  
 36799, 36807, 36815, 36821, 36834  
`\hbox_set:Nw` . . . . .  
 . . . . . 278, 33070, 33070, 33082, 33741  
`\hbox_set_end:` . . . . .  
 . . . . . 278, 33070, 33084, 33089, 33744  
`\hbox_set_to_wd:Nnn` . . . . .  
 . . . . . 278, 33058, 33058, 33068  
`\hbox_set_to_wd:Nnw` . . . . .  
 . . . . . 278, 33090, 33090, 33102  
`\hbox_to_wd:nn` 277, 33104, 33104, 33558  
`\hbox_to_zero:n` . . . . . 277, 33104,  
 33109, 33115, 33117, 33119, 36678  
`\hbox_unpack:N` . . . . .  
 . . . . . 278, 33120, 33120, 33122, 34393  
`\hbox_unpack_clear:N` . . . . . 37502  
`\hbox_unpack_drop:N` . . . . .  
 . . . . . 281, 33120, 33121, 33123, 37503
- hcoffin commands:
- `\hcoffin_gset:Nn` . . . . .  
 . . . . . 286, 33664, 33677, 33689  
`\hcoffin_gset:Nw` . . . . .  
 . . . . . 286, 33737, 33750, 33762  
`\hcoffin_gset_end:` . . . . .  
 . . . . . 286, 33737, 33755, 33764  
`\hcoffin_set:Nn` 286, 33664, 33664,  
 33676, 34627, 34634, 34672, 34707  
`\hcoffin_set:Nw` . . . . .  
 . . . . . 286, 33737, 33737, 33749  
`\hcoffin_set_end:` . . . . .  
 . . . . . 286, 33737, 33742, 33763
- `\hfi` . . . . . 1121  
`\hfil` . . . . . 286

\hfill ..... 287  
 \hfilneg ..... 288  
 \hfuzz ..... 289  
 \hjcode ..... 823  
 \hoffset ..... 290  
 \holdinginserts ..... 291  
 hook commands:  
     \hook\_gput\_code:nmm .... 29114, 29116  
 \hpack ..... 824  
 \hrule ..... 292  
 \hsize ..... 293  
 \hskip ..... 294  
 \hss ..... 295  
 \ht ..... 296  
 \Huge ..... 32512  
 \huge ..... 32516  
 \hyphenation ..... 297  
 \hyphenationbounds ..... 825  
 \hyphenationmin ..... 826  
 \hyphenchar ..... 298  
 \hyphenpenalty ..... 299  
 \hyphenpenaltymode ..... 827

**I**

\i ..... 31882,  
     32594, 32720, 32722, 32724, 32726,  
     32777, 32780, 32783, 32786, 32857  
 \if ..... 226, 300  
 if commands:  
     \if:w 28, 182, 370, 371, 408, 476, 674,  
         688, 689, 692, 702, 703, 716, 1429,  
         1435, 1797, 2093, 2094, 2977, 2980,  
         2981, 2982, 2983, 2998, 2999, 3000,  
         3001, 3002, 3003, 3004, 3005, 3006,  
         3070, 3071, 3073, 4264, 4990, 12452,  
         12462, 12558, 12921, 12941, 12956,  
         13391, 13398, 13403, 17868, 19421,  
         23297, 23670, 23674, 23696, 23789,  
         23821, 23840, 23906, 23920, 23937,  
         23958, 23997, 24009, 24295, 24337,  
         24457, 24472, 24824, 26996, 27026,  
         28533, 29725, 29734, 29750, 29925  
     \if\_bool:N ... 70, 572, 1439, 8471, 8516  
     \if\_box\_empty:N ..... 284, 32982, 32984, 32994  
     \if\_case:w .... 168, 722, 724, 763,  
         833, 991, 1083, 1138, 1182, 2000,  
         2668, 3854, 4053, 4237, 4632, 4904,  
         5717, 5746, 5803, 6478, 6531, 7152,  
         7199, 7297, 7606, 8116, 8127, 10720,  
         13668, 13742, 14066, 15095, 17176,  
         17180, 17759, 17792, 18974, 22598,  
         22851, 22866, 23237, 23266, 24545,  
         24586, 25260, 25395, 25470, 25495,

25547, 25991, 26007, 26024, 26301,  
     26528, 26555, 26713, 26748, 26906,  
     26951, 27002, 27128, 27151, 27184,  
     27243, 27283, 27298, 27313, 27328,  
     27343, 27358, 27884, 27937, 28021,  
     28036, 28088, 28101, 28185, 28239,  
     28313, 28530, 28817, 28896, 37335

\if\_catcode:w .....  
     .. 28, 701, 888, 1429, 1437, 3117,  
     3843, 3846, 4001, 4003, 4005, 4007,  
     4009, 4011, 4013, 4238, 4239, 4398,  
     12864, 12905, 19038, 19041, 19044,  
     19047, 19050, 19053, 19056, 19131,  
     19136, 19141, 19146, 19153, 19160,  
     19165, 19170, 19175, 19180, 19185,  
     19195, 19222, 19531, 19590, 19595,  
     19642, 19643, 22434, 23425, 23630,  
     23948, 23995, 24294, 24336, 29889,  
     29890, 29926, 29941, 29942, 29943,  
     29944, 29945, 29946, 29947, 29948,  
     29949, 29972, 29975, 29978, 29981,  
     29984, 29987, 29990, 37336, 37337

\if\_charcode:w .....  
     .. 28, 182, 442, 700, 701, 727,  
     888, 1429, 1436, 3913, 3937, 3986,  
     4282, 4319, 4321, 4841, 4851, 5360,  
     5916, 7089, 7092, 11337, 11346,  
     12850, 12898, 13826, 14046, 14708,  
     19200, 19592, 22854, 24837, 25199

\if\_cs\_exist:N ..... 28, 1444,  
     1444, 1824, 1852, 2617, 19230, 19430

\if\_cs\_exist:w .....  
     .. 28, 1444, 1445, 1472, 1833,  
     1861, 1987, 18008, 18036, 18045, 36846

\if\_dim:w ..... 221, 20082,  
     20082, 20170, 20182, 20205, 20376

\if\_eof:w ..... 93,  
     627, 10284, 10284, 10289, 10374, 10392

\if\_false: 27, 64, 194, 398, 434, 455,  
     541, 553, 554, 557, 583, 646, 682,  
     687, 691, 699, 807, 824, 870, 871,  
     903, 1429, 1430, 2628, 2638, 2651,  
     2664, 2692, 2708, 2806, 2820, 2826,  
     2833, 2841, 2851, 2864, 2868, 3790,  
     3832, 3881, 3884, 4402, 4403, 4410,  
     4411, 5098, 5117, 5118, 5127, 5192,  
     5235, 5249, 5253, 5467, 5500, 5512,  
     5516, 5550, 5555, 5563, 5598, 5605,  
     5610, 5658, 5895, 5914, 5925, 5948,  
     5960, 5961, 5964, 7379, 7396, 7725,  
     7751, 7759, 7766, 7796, 7932, 7934,  
     7935, 7941, 8776, 8977, 8985, 10699,  
     10739, 10743, 10750, 10758, 11008,  
     11021, 12062, 12066, 12277, 12284,

12418, 12419, 12530, 12534, 12573,  
12805, 12810, 12817, 12822, 12832,  
12922, 12935, 12953, 12957, 12967,  
16649, 16652, 16831, 16836, 17389,  
18947, 18953, 18971, 20005, 20006,  
20007, 20008, 20043, 20044, 20045,  
20046, 20192, 30307, 30313, 37089,  
37090, 37208, 37220, 37246, 37256  
\if\_hbox:N . . . . . 284, 32982, 32982, 32986  
\if\_int\_compare:w . . . . .  
.... 27, 168, 716, 824, 825, 1462,  
1462, 2856, 3364, 3421, 3450, 3491,  
3502, 3505, 3523, 3578, 3588, 3598,  
3818, 3890, 3923, 3931, 3954, 3978,  
4034, 4049, 4135, 4138, 4333, 4513,  
4572, 4578, 4579, 4586, 4590, 4596,  
4597, 4602, 4603, 4611, 4612, 4613,  
4619, 4651, 4652, 4901, 4921, 4922,  
4923, 4926, 4930, 4931, 4934, 4935,  
4943, 4944, 4947, 4951, 4952, 4955,  
5014, 5036, 5048, 5057, 5065, 5068,  
5078, 5081, 5109, 5196, 5308, 5374,  
5379, 5407, 5465, 5498, 5609, 5626,  
5972, 6005, 6036, 6425, 6496, 6522,  
6583, 6596, 6607, 6623, 6674, 6715,  
6721, 6727, 6892, 6893, 6920, 6947,  
7046, 7104, 7223, 7232, 7243, 7258,  
7314, 7323, 7375, 7392, 7415, 7673,  
7702, 7747, 7755, 7837, 8779, 10287,  
10288, 10706, 11353, 13112, 13121,  
13170, 13171, 13177, 13379, 13386,  
13652, 13707, 13708, 13714, 13726,  
13742, 13876, 14055, 14063, 14270,  
14271, 14272, 14277, 14278, 14302,  
14354, 14454, 14673, 14735, 14739,  
14769, 14772, 14788, 14792, 14813,  
14893, 14895, 14914, 14915, 14933,  
14935, 14989, 14992, 14993, 15111,  
15112, 15253, 15258, 17176, 17231,  
17272, 17273, 17369, 17422, 17424,  
17426, 17428, 17430, 17432, 17434,  
17437, 17570, 18880, 18881, 18882,  
18883, 18888, 18889, 18893, 18919,  
19412, 20221, 22288, 22291, 22335,  
22401, 22420, 22599, 22600, 22794,  
22891, 23117, 23127, 23135, 23148,  
23161, 23168, 23189, 23201, 23210,  
23221, 23330, 23335, 23407, 23437,  
23590, 23592, 23629, 23634, 23687,  
23707, 23734, 23748, 23783, 23810,  
23838, 23854, 23870, 23888, 23948,  
23968, 23984, 24068, 24091, 24120,  
24122, 24303, 24305, 24345, 24347,  
24369, 24386, 24391, 24421, 24489,  
24534, 24848, 24895, 24898, 24929,  
24938, 24941, 24946, 24947, 24950,  
24953, 25140, 25264, 25285, 25322,  
25417, 25471, 25472, 25475, 25478,  
25548, 25557, 25762, 25835, 25888,  
25892, 25896, 25914, 25949, 25950,  
25951, 25952, 25953, 25979, 26303,  
26306, 26400, 26493, 26541, 26557,  
26684, 26718, 26776, 26785, 26825,  
26997, 27008, 27026, 27049, 27081,  
27084, 27131, 27161, 27208, 27222,  
27386, 27430, 27888, 27926, 27935,  
27971, 28055, 28058, 28287, 28520,  
28588, 28589, 28590, 28600, 28628,  
28633, 28634, 28697, 28698, 28699,  
28703, 28718, 28723, 28771, 28775,  
29394, 29395, 29401, 29726, 29958,  
30084, 30085, 30089, 30112, 30113,  
31344, 31347, 31348, 31351, 31352,  
31373, 31376, 31379, 31382, 31385,  
31388, 31407, 31410, 31413, 31416,  
31419, 31436, 31439, 31442, 31445,  
31461, 31464, 31467, 37289, 37297  
\if\_int\_odd:w . . . . . 169, 1158,  
4059, 4641, 5025, 5033, 5045, 5471,  
5786, 17176, 17179, 17305, 17475,  
17483, 17983, 18879, 18887, 18904,  
19641, 23139, 23186, 23198, 24582,  
25531, 25817, 27172, 27736, 27775,  
27785, 27828, 27852, 28012, 28696  
\if\_meaning:w . . . . . 27, 451, 701,  
790, 1067, 1222, 1398, 1429, 1438,  
1648, 1674, 1692, 1754, 1759, 1768,  
1821, 1839, 1849, 1867, 2018, 2032,  
2154, 2271, 2334, 2335, 2669, 2672,  
2673, 2674, 2675, 2768, 2798, 2811,  
2817, 2925, 2948, 2957, 3149, 3222,  
3234, 3235, 3374, 3375, 3840, 3870,  
3898, 3999, 4203, 4240, 4241, 4344,  
4395, 4442, 4443, 4680, 4836, 4861,  
4873, 4989, 5013, 5356, 5359, 5793,  
6460, 6631, 6642, 6657, 6818, 6862,  
6997, 7270, 7584, 7701, 7814, 8582,  
8604, 10656, 10951, 12442, 12486,  
12502, 12516, 12889, 13501, 13579,  
13602, 13763, 13801, 14216, 14943,  
15092, 15107, 15134, 15249, 16182,  
16188, 16214, 16226, 16234, 16266,  
16273, 16297, 16301, 16379, 16404,  
16419, 16685, 16748, 16763, 16771,  
17199, 17202, 17212, 17247, 17252,  
17253, 17404, 18215, 18230, 18252,  
18266, 19190, 19227, 19266, 19269,  
19404, 19506, 19534, 19545, 19583,

19644, 20151, 20198, 20379, 22580,	\ifhmode . . . . .	307
22601, 22613, 22623, 22718, 22773,	\ifincname . . . . .	686
22782, 22873, 22888, 22890, 23028,	\ifinner . . . . .	308
23116, 23126, 23138, 23151, 23152,	\ifjfont . . . . .	1124
23171, 23172, 23186, 23187, 23198,	\ifmbox . . . . .	1125
23199, 23265, 23312, 23347, 23350,	\ifmdir . . . . .	1126
23366, 23373, 23426, 23429, 23545,	\ifmmode . . . . .	309
23546, 23547, 23548, 23551, 23647,	\ifnum . . . . .	27, 39, 69, 76, 310
23760, 23766, 23996, 24040, 24251,	\ifodd . . . . .	311
24322, 24583, 24601, 24651, 24661,	\ifpdfabsdim . . . . .	647
24884, 24885, 24886, 24887, 24888,	\ifpdfabsnum . . . . .	648
24889, 25121, 25133, 25134, 25162,	\ifpdfprimitive . . . . .	649
25174, 25181, 25198, 25261, 25296,	\ifprimitive . . . . .	779
25310, 25356, 25363, 25439, 25451,	\iftbox . . . . .	1127
25551, 25554, 25565, 25618, 25691,	\iftdir . . . . .	1129
25761, 25764, 25771, 25804, 25805,	\iftfont . . . . .	1128
25808, 26029, 26274, 26285, 26466,	\iftrue . . . . .	312
26476, 26525, 26624, 26704, 26753,	\ifvbox . . . . .	313
26767, 26914, 26948, 26960, 26973,	\ifvmode . . . . .	314
26976, 26979, 26982, 27007, 27108,	\ifvoid . . . . .	315
27112, 27171, 27188, 27194, 27778,	\ifx . . . . .	4, 21, 25, 30, 34, 70, 71, 73, 82, 106, 107, 316
27831, 27882, 27883, 27885, 27886,	\ifybox . . . . .	1130
27906, 27923, 27990, 28088, 28184,	\ifydir . . . . .	1131
28238, 28312, 28382, 28387, 28519,	\ignoreligaturesinfont . . . . .	918
28551, 28562, 28669, 28830, 28883,	\ignorespaces . . . . .	317
28889, 29688, 29904, 30588, 37048,	\IJ . . . . .	30148, 31873, 32584
37082, 37213, 37250, 37269, 37338	\ij . . . . .	30148, 31873, 32596
\if_mode_horizontal: . . . . .	\immediate . . . . .	89, 318
..... 28, 1440, 1441, 8768	\immediateassigned . . . . .	830
\if_mode_inner: . 28, 1440, 1443, 8770	\immediateassignment . . . . .	831
\if_mode_math: .. 28, 1440, 1440, 8772	in . . . . .	258
\if_mode_vertical: . . . . .	\indent . . . . .	319
..... 28, 1440, 1442, 2281, 8766	inf . . . . .	257
\if_predicate:w . . . . .	\infty . . . . .	23548, 23549
..... 62, 64, 70, 8471, 8471, 8559, 8620, 8635, 8646, 8661, 8672	inherit commands:	
\if_true: . . . . .	.inherit:n . . . . .	226, 21337
..... 27, 64, 1429, 1429, 12060, 12064, 12947, 12953	\inhibitglue . . . . .	1132
\if_vbox:N . . . . .	\inhibitxspcode . . . . .	1133
284, 32982, 32983, 32988	\initcatcodetable . . . . .	832
\ifabsdim . . . . .	initial commands:	
916	.initial:n . . . . .	226, 21339
\ifabsnum . . . . .	\input . . . . .	31, 320
917	\inputlineno . . . . .	321
\ifcase . . . . .	\insert . . . . .	322
301	\insertht . . . . .	919
\ifcat . . . . .	\insertpenalties . . . . .	323
302	int commands:	
\ifcondition . . . . .	\int_abs:n 157, 818, 17205, 17205, 22335	
829	\int_add:Nn . . . . .	
\ifcsname . . . . .	..... 159, 4620, 5788, 6613, 6614, 6872, 6944, 10815, 17335, 17335, 17343	
350, 644, 540		
\ifdbox . . . . .		
1122		
\ifddir . . . . .		
1123		
\ifdefined . . . . .		
541		
\ifdim . . . . .		
303		
\IfDocumentMetadataTF . . . . .		
36689, 36690		
\ifeof . . . . .		
304		
\iffalse . . . . .		
305		
\iffontchar . . . . .		
542		
\ifhbox . . . . .		
306		

```

\int_case:nn ..... 161, 833, 17443, 17458, 17622,
17628, 29280, 31600, 31615, 31691
\int_case:nnn ..... 37504
\int_case:nnTF ..... 161, 4373,
8437, 17079, 17443, 17443, 17448,
17453, 18576, 23457, 28134, 37505
\int_compare:n ..... 17382
\int_compare:nNn ..... 17435
\int_compare:nNnTF ..... 159–162,
244, 3351, 4182, 4659, 4671, 4824,
5151, 5153, 6018, 6821, 7175, 7334,
7726, 7918, 8454, 8847, 8853, 9201,
10515, 10628, 11100, 11224, 11234,
11582, 11625, 12279, 12307, 12322,
12330, 13067, 13074, 13141, 13631,
13633, 13642, 13885, 13887, 13890,
13946, 14410, 14486, 15293, 16700,
16889, 16896, 17286, 17292, 17435,
17467, 17519, 17527, 17536, 17542,
17554, 17557, 17618, 17706, 17712,
17718, 17738, 17892, 17911, 17913,
17955, 18648, 18650, 18655, 18664,
18684, 18701, 18718, 18994, 19005,
19028, 19078, 19089, 20399, 22044,
22273, 22278, 22285, 22393, 22469,
22495, 24913, 26054, 28117, 28262,
28264, 28751, 28961, 29132, 29165,
29304, 29310, 29321, 29347, 29350,
29475, 29536, 29543, 29555, 29593,
29596, 29620, 30026, 30028, 30031,
30045, 30047, 30069, 30078, 31030,
31071, 35028, 35802, 36074, 36080,
36933, 37006, 37007, 37009, 37011
\int_compare:nTF ..... 160, 162, 245, 909, 6241, 6281,
8173, 8402, 8403, 8408, 8410, 10246,
10469, 17382, 17491, 17499, 17508,
17514, 28322, 29171, 37031, 37033
\int_compare_p:n ..... 160, 6288, 17382
\int_compare_p:nNn ..... 27,
160, 5863, 5864, 8861, 9083, 9180,
9182, 9184, 10405, 11213, 11214,
17435, 29087, 30958, 30976, 31524,
31747, 31748, 31769, 31770, 32010,
35026, 36624, 36625, 36632, 36635,
36636, 36644, 36647, 36648, 36691
\int_const:Nn ..... 158, 4550,
4551, 4552, 4553, 4962, 4963, 4964,
4965, 4966, 4967, 4971, 4972, 4973,
4974, 4975, 4976, 4977, 4978, 4979,
4980, 4981, 4982, 4983, 8997, 9093,
9095, 9097, 9098, 9099, 9167, 10177,
10334, 10400, 10401, 13995, 13996,
17282, 17282, 17304, 17921, 17922,
17923, 17924, 17925, 17926, 17927,
17928, 17929, 17930, 17931, 17932,
17933, 17934, 17979, 17980, 17981,
22561, 22562, 22563, 22564, 22565,
22566, 22567, 22751, 22752, 22753,
22755, 22756, 22757, 22760, 22761,
22762, 23111, 23385, 23386, 23387,
23388, 23389, 23390, 23391, 23392,
23393, 23394, 23395, 23396, 23397,
23398, 23399, 27181, 28470, 29483
\int_decr:N ..... 159, 3511, 3512, 3513, 3576, 3577,
3586, 3587, 3596, 3597, 3833, 7316,
7393, 7608, 7703, 17347, 17349, 17356
\int_div_round:nn .. 157, 17237, 17258
\int_div_truncate:nn ..... 157, 158, 9096, 14111,
14116, 14762, 14763, 14818, 15000,
15166, 15177, 17237, 17237, 17633,
17731, 17751, 29407, 29420, 29425,
29437, 29493, 29597, 29605, 29706
\int_do_until:nn ..... 162, 17489, 17511, 17515
\int_do_until:nNn ..... 161, 17517, 17539, 17543
\int_do_while:nn ..... 162, 17489, 17505, 17509
\int_do_while:nNn ..... 161, 17517, 17533, 17537
\int_eval:n ..... 19, 33,
157–161, 168, 308, 349, 378, 459,
696, 820, 836, 965, 966, 970, 971,
977, 1011, 1060, 1085, 1087, 1393,
2000, 2029, 2045, 3453, 3717, 3718,
3956, 4037, 4041, 4064, 4510, 4641,
4905, 5786, 6032, 6240, 6445, 6447,
6461, 6462, 6464, 6465, 6607, 6697,
6740, 6915, 6963, 7062, 7236, 7242,
7245, 8129, 8132, 8177, 8222, 8223,
8443, 8721, 9121, 10008, 10230,
10454, 10751, 10809, 11205, 11206,
11228, 11238, 11245, 11246, 12333,
12703, 12708, 12716, 13060, 13068,
13076, 13103, 13107, 13116, 13123,
13158, 13168, 13625, 13638, 13663,
13687, 13688, 13700, 13705, 13736,
13753, 13790, 13881, 13899, 13903,
13910, 13919, 14066, 14086, 14104,
14379, 14904, 14919, 14947, 15096,
15101, 15119, 15263, 16882, 16890,
16898, 17053, 17188, 17188, 17283,
17446, 17451, 17456, 17461, 17615,
17701, 17703, 17833, 17843, 17878,
```

17889, 17895, 17906, 17937, 17974,  
 17978, 18545, 18557, 18642, 18652,  
 18666, 18673, 18688, 18751, 18753,  
 18821, 18823, 18827, 18829, 18833,  
 18835, 18839, 18841, 18874, 18875,  
 19848, 22043, 22129, 22177, 22195,  
 22213, 22272, 22310, 22311, 22362,  
 22379, 22465, 22489, 22498, 22502,  
 22571, 28459, 28619, 28622, 28623,  
 28713, 28714, 28746, 28794, 28856,  
 28864, 28933, 28957, 29123, 29390,  
 29442, 29445, 29450, 29456, 29532,  
 29550, 29574, 29630, 29635, 29670,  
 29681, 29700, 29727, 29810, 31016,  
 31340, 31369, 31403, 31432, 31457,  
 33012, 33022, 36082, 36125, 36171,  
 36936, 37002, 37077, 37291, 37299  
 $\backslash \text{int\_eval:w}$  . . . . . 157, 351, 354,  
 3783, 4023, 4033, 10702, 10711,  
 10736, 10748, 13656, 14098, 17029,  
 $\underline{17188}$ , 17190, 18011, 18046, 22227,  
 22406, 22413, 22414, 22425, 26003  
 $\backslash \text{int\_from\_alph:n}$  . . . . . 165,  $\underline{17876}$ , 17876  
 $\backslash \text{int\_from\_base:nn}$  . . . . . . . . .  
 166,  $\underline{17893}$ , 17893, 17916, 17918, 17920  
 $\backslash \text{int\_from\_bin:n}$  . . . . . . . . .  
 166,  $\underline{17915}$ , 17915, 37507  
 $\backslash \text{int\_from\_binary:n}$  . . . . . 37506  
 $\backslash \text{int\_from\_hex:n}$  . . . . . 166,  $\underline{17915}$ ,  
 17917, 35204, 35205, 35206, 37509  
 $\backslash \text{int\_from\_hexadecimal:n}$  . . . . . 37508  
 $\backslash \text{int\_from\_oct:n}$  . . . . . . . . .  
 166,  $\underline{17915}$ , 17919, 37511  
 $\backslash \text{int\_from\_octal:n}$  . . . . . 37510  
 $\backslash \text{int\_from\_roman:n}$  . . . . . 166,  $\underline{17935}$ , 17935  
 $\backslash \text{int\_gadd:Nn}$  . . . . . 159,  $\underline{17335}$ , 17339, 17344  
 $\backslash \text{int\_gdecr:N}$  . . . . . . . . . 159,  
 4094, 10369, 12649, 13520, 16955,  
 17011,  $\underline{17347}$ , 17353, 17358, 17613,  
 18479, 20021, 20364, 25105, 32270  
 $\backslash \text{int\_gincr:N}$  . . . . . 159, 4083, 4211, 6386,  
 10360, 12640, 13509, 16947, 17005,  
 $\underline{17347}$ , 17351, 17357, 17588, 17599,  
 18470, 20016, 20343, 20350, 22037,  
 22263, 25084, 25091, 28741, 29044,  
 32264, 35789, 36382, 36387, 36392  
 $\cdot \text{int\_gset:N}$  . . . . . . . . . 226,  $\underline{21347}$   
 $\backslash \text{int\_gset:Nn}$  . . . . . . . . . 159,  
 821, 6406, 9433,  $\underline{17359}$ , 17361, 17364  
 $\backslash \text{int\_gset_eq:NN}$  . . . . . . . . .  
 158,  $\underline{17327}$ , 17329, 17330  
 $\backslash \text{int\_gsub:Nn}$  . . . . . . . . .  
 159,  $\underline{17335}$ , 17341, 17346, 28755  
 $\backslash \text{int\_gzero:N}$  . . . . . . . . . 158,  
 6366, 6383,  $\underline{17317}$ , 17318, 17320, 17324  
 $\backslash \text{int\_gzero\_new:N}$  . . . . . . . . .  
 158,  $\underline{17321}$ , 17323, 17326  
 $\backslash \text{int\_if\_even:n}$  . . . . . . . . . 17481  
 $\backslash \text{int\_if\_even:nTF}$  . . . . . . . . . 161,  $\underline{17473}$   
 $\backslash \text{int\_if\_even_p:n}$  . . . . . . . . . 161,  $\underline{17473}$   
 $\backslash \text{int\_if\_exist:N}$  . . . . . . . . . 17331, 17333  
 $\backslash \text{int\_if\_exist:NTF}$  . . . . . . . . .  
 158, 5712, 5767, 17322,  
 $\underline{17324}$ ,  $\underline{17331}$ , 17949, 17953, 36613  
 $\backslash \text{int\_if\_exist_p:N}$  . . . . . . . . . 158,  $\underline{17331}$   
 $\backslash \text{int\_if\_odd:n}$  . . . . . . . . . 17473  
 $\backslash \text{int\_if\_odd:nTF}$  . . . . . . . . . 161,  
 7525, 7548, 7622, 10961,  $\underline{17473}$ , 26389  
 $\backslash \text{int\_if\_odd_p:n}$  . . . . . . . . . 161, 6314,  $\underline{17473}$   
 $\backslash \text{int\_incr:N}$  . . . . . . . . . 159, 3425, 3521,  
 3522, 3874, 3916, 3929, 3947, 4486,  
 4487, 5603, 6246, 6413, 6453, 6542,  
 6873, 6969, 7304, 7376, 7602, 7607,  
 7642, 7700, 7772, 7773, 7822, 7844,  
 7947, 7948, 8111, 16722,  $\underline{17347}$ ,  
 17347, 17355, 21020, 22194, 22351,  
 22385, 22438, 22511, 28844, 36115  
 $\backslash \text{int\_log:N}$  . . . . . 167,  $\underline{17975}$ , 17975, 17976  
 $\backslash \text{int\_log:n}$  . . . . . . . . . 167,  $\underline{17977}$ , 17977  
 $\backslash \text{int\_max:nn}$  . . . . . . . . . 158, 1176,  
 6091, 6092, 6099, 6100, 6398, 6566,  
 7914, 7916,  $\underline{17205}$ , 17213, 26250, 27410  
 $\backslash \text{int\_min:nn}$  . . . . . . . . .  
 158,  $\underline{1180}$ ,  $\underline{17205}$ , 17221, 29580  
 $\backslash \text{int\_mod:nn}$  . . . . . . . . .  
 158, 9094, 14415, 14479, 14763,  
 14764, 15001,  $\underline{17237}$ , 17260, 17623,  
 17722, 17742, 29439, 29608, 29719  
 $\backslash \text{int\_new:N}$  . . . . . . . . .  
 158, 3330, 3331, 3332, 3333, 3334,  
 3335, 3336, 3337, 3338, 3339, 3340,  
 3769, 3770, 3771, 3772, 4200, 4537,  
 4538, 4539, 4549, 4960, 4961, 4968,  
 4969, 4986, 6331, 6333, 6334, 6335,  
 6338, 6361, 6362, 6746, 6747, 6748,  
 6749, 6750, 6751, 6752, 6754, 6755,  
 6756, 6757, 6760, 6761, 6762, 7023,  
 7569, 7572, 7573, 7574, 7580, 7581,  
 8458, 8780, 10537, 10540, 10542,  
 10555, 14461,  $\underline{17276}$ , 17276, 17281,  
 17288, 17294, 17322, 17324, 17991,  
 17992, 17993, 17994, 17995, 17996,  
 20830, 22021, 22024, 22025, 22259,  
 28734, 28735, 28912, 34855, 35730  
 $\backslash \text{int\_rand:n}$  . . . . . 166, 307, 22186,  
 22372, 28461, 28464,  $\underline{28711}$ , 28711

```

\int_rand:nn ..... 74,
  166, 307, 1179, 1180, 1187, 1395,
  13083, 16904, 17979, 18702, 18707,
  28455, 28458, 28617, 28617, 36926
\int_range:nn ..... 1181
.int_set:N ..... 226, 21347
\int_set:Nn ..... 159,
  349, 2206, 2220, 2221, 2224, 2226,
  2228, 3345, 3347, 3349, 3371, 3372,
  3387, 3395, 3396, 3408, 3409, 3427,
  3430, 3828, 3891, 4232, 4331, 4334,
  4474, 5794, 6332, 6394, 6396, 6402,
  6440, 6442, 6511, 6562, 6563, 6573,
  6584, 6608, 6626, 6675, 6807, 6809,
  6812, 6833, 6877, 6878, 6919, 6954,
  7647, 7719, 7721, 7860, 7890, 7913,
  7915, 10320, 10322, 10521, 10523,
  10538, 10548, 10561, 10608, 10614,
  10626, 10631, 12280, 12315, 14540,
  14589, 14642, 16723, 17359, 17359,
  17363, 21025, 22429, 29012, 29013,
  29028, 29193, 29208, 29244, 33021,
  33023, 33031, 33032, 33033, 33034
\int_set_eq:NN 158, 3388, 3418, 4610,
  5079, 5083, 5092, 5094, 5137, 5204,
  5502, 5602, 5615, 5714, 6352, 6372,
  6389, 6417, 6451, 6452, 6502, 6605,
  6606, 6658, 6707, 6785, 6808, 6813,
  6827, 6831, 6835, 6874, 6884, 7015,
  7016, 7598, 7815, 8107, 8978, 11010,
  12278, 12281, 17327, 17327, 17328
\int_show:N .. 167, 17971, 17971, 17972
\int_show:n 167, 839, 1393, 17973, 17973
\int_sign:n ..... 157, 914, 17191, 17191, 29094
\int_step_... ..... 60
\int_step_function:nN ..... 163, 16713, 17545, 17578, 29212, 29213
\int_step_function:nnN . 163, 3480,
  6885, 7715, 17545, 17580, 18967,
  29214, 29215, 29216, 29217, 29238
\int_step_function:nnnN ..... 163, 310, 829, 1063, 7893, 7901,
  17545, 17545, 17579, 17581, 17612
\int_step_inline:nn ..... 163, 971, 17582, 17582, 22266,
  28938, 28972, 29023, 29614, 29657
\int_step_inline:nnn .. 163, 6800,
  8461, 10181, 10412, 14399, 14408,
  17582, 17584, 28945, 28948, 29194
\int_step_inline:nnnn ..... 163, 1065, 17582, 17583, 17585, 17586
\int_step_variable:nNn ..... 163, 17582, 17593
\int_step_variable:nnNn ..... 163, 17582, 17595
\int_step_variable:nnnNn ..... 163, 17582, 17594, 17596, 17597
\int_sub:Nn ..... 159, 4614, 5315, 6661, 6669,
  6678, 9377, 10823, 17335, 17337, 17345
\int_to_Alph:n 164, 165, 17636, 17668
\int_to_alph:n 164, 165, 17636, 17636
\int_to_arabic:n ... 164, 17615, 17615
\int_to_Base:n ..... 165
\int_to_base:n ..... 165
\int_to_Base:nn ..... 165, 166, 17700, 17702, 17827
\int_to_base:nn ..... 165,
  166, 17700, 17700, 17823, 17825, 17829
\int_to_bin:n ..... 165, 166, 17822, 17822, 37513
\int_to_binary:n ..... 37512
\int_to_Hex:n ..... 165,
  166, 4827, 17822, 17826, 29745, 35724
\int_to_hex:n ..... 165, 166, 17822, 17824, 37515
\int_to_hexadecimal:n ..... 37514
\int_to_oct:n ..... 165, 166, 17822, 17828, 37517
\int_to_octal:n ..... 37516
\int_to_Roman:n 165, 166, 17830, 17840
\int_to_roman:n 165, 166, 17830, 17830
\int_to_symbols:nnn ..... 164, 17616, 17616, 17632, 17638, 17670
\int_until_do:nn ..... 162, 17489, 17497, 17502
\int_until_do:nNn ..... 162, 17517, 17525, 17530
\int_use:N ..... 156,
  159, 1004, 1010, 2225, 2227, 2229,
  4085, 4213, 4230, 5111, 5198, 5276,
  5287, 5296, 5300, 5311, 5312, 5318,
  5319, 5325, 5326, 5485, 6314, 6407,
  6412, 6433, 6435, 6540, 6553, 6554,
  6955, 7007, 7106, 7117, 7272, 7647,
  7731, 7732, 7921, 7922, 8455, 9337,
  10010, 10323, 10362, 10517, 12642,
  12644, 13511, 13515, 14900, 14924,
  14938, 14958, 14965, 15147, 15256,
  15261, 15277, 16948, 16954, 17007,
  17009, 17365, 17365, 17366, 17591,
  17602, 18472, 18474, 20015, 20023,
  20346, 20353, 21026, 22038, 22132,
  22180, 25087, 25094, 29046, 29048,
  29079, 29127, 32266, 32268, 35795,
  37099, 37108, 37110, 37113, 37118,
  37127, 37129, 37133, 37136, 37141

```

```

\int_value:w ..... 157,
168, 354, 403, 434, 577, 818, 824,
909, 965, 966, 970, 971, 981, 987,
991, 1004, 1012, 1019, 1022, 1027,
1034, 1061, 1071, 1079, 1087, 1153,
1157, 1171, 1395, 1802, 2818, 2820,
3453, 3783, 3795, 3974, 4021, 4023,
4033, 4041, 4060, 4062, 4070, 4279,
4310, 4386, 4406, 4415, 4820, 5344,
5350, 5382, 5384, 5393, 5394, 5509,
5994, 6009, 7040, 7041, 7052, 7743,
8595, 8598, 8721, 9109, 9151, 9156,
10702, 10711, 13116, 13123, 13625,
13626, 13638, 13656, 13663, 13686,
13687, 13688, 13700, 13736, 14332,
14456, 14818, 14896, 14904, 14919,
14947, 17017, 17029, 17176, 17176,
17193, 17194, 17207, 17208, 17215,
17216, 17217, 17223, 17224, 17225,
17239, 17241, 17242, 17259, 17262,
17263, 17264, 17271, 17385, 17389,
17419, 17548, 17549, 17550, 17576,
17786, 17819, 18011, 18046, 18056,
18874, 18875, 20179, 20370, 20406,
22282, 22285, 22310, 22311, 22357,
22362, 22406, 22413, 22414, 22425,
22645, 22646, 22647, 22648, 22649,
22663, 22811, 22872, 22890, 23185,
23315, 23329, 23331, 23333, 23336,
23372, 23510, 23540, 23541, 23578,
23586, 23717, 23722, 23724, 23733,
23737, 23774, 23782, 23785, 23791,
23802, 23813, 23819, 23820, 23823,
23866, 23876, 23878, 23894, 23896,
23919, 23933, 24009, 24010, 24084,
24172, 24883, 24916, 25271, 25272,
25273, 25275, 25321, 25324, 25327,
25350, 25352, 25373, 25375, 25384,
25386, 25390, 25408, 25415, 25421,
25431, 25433, 25447, 25455, 25463,
25507, 25509, 25525, 25527, 25530,
25533, 25587, 25595, 25597, 25599,
25601, 25604, 25607, 25609, 25628,
25630, 25634, 25640, 25642, 25646,
25668, 25671, 25679, 25681, 25684,
25685, 25686, 25687, 25702, 25705,
25708, 25711, 25720, 25723, 25726,
25729, 25736, 25738, 25744, 25752,
25754, 25756, 25782, 25784, 25793,
25795, 25799, 25816, 25837, 25841,
25853, 25856, 25859, 25862, 25865,
25868, 25871, 25874, 25878, 25890,
25894, 25898, 25901, 25922, 25924,
25926, 25936, 25960, 25963, 25975,
25977, 25983, 25986, 26003, 26023,
26073, 26078, 26080, 26087, 26090,
26093, 26096, 26099, 26102, 26111,
26123, 26131, 26133, 26143, 26145,
26152, 26161, 26163, 26166, 26169,
26172, 26175, 26188, 26190, 26198,
26200, 26208, 26210, 26220, 26223,
26226, 26233, 26248, 26266, 26269,
26325, 26339, 26341, 26347, 26360,
26362, 26364, 26388, 26404, 26411,
26412, 26456, 26458, 26459, 26460,
26501, 26503, 26540, 26547, 26554,
26575, 26577, 26579, 26581, 26594,
26598, 26599, 26600, 26601, 26602,
26607, 26612, 26614, 26620, 26637,
26638, 26639, 26640, 26641, 26642,
26647, 26649, 26651, 26653, 26655,
26660, 26662, 26664, 26666, 26668,
26670, 26692, 26700, 26716, 26721,
26725, 26784, 26833, 26901, 26910,
26918, 26929, 26931, 26934, 26937,
27025, 27061, 27063, 27066, 27069,
27072, 27075, 27082, 27085, 27087,
27091, 27113, 27115, 27147, 27217,
27227, 27232, 27242, 27384, 27416,
27425, 27657, 27658, 27669, 27672,
27675, 27678, 27681, 27684, 27687,
27690, 27693, 27711, 27721, 27730,
27748, 27757, 27764, 27774, 27818,
27827, 27862, 27905, 27922, 27978,
27989, 28000, 28210, 28286, 28333,
28378, 28386, 28388, 28390, 28482,
28505, 28559, 28599, 28611, 28622,
28623, 28653, 28656, 28659, 28661,
28663, 28670, 28673, 28681, 28686,
28691, 28794, 28856, 28864, 28877,
28878, 28879, 28889, 29940, 36926,
36936, 37041, 37051, 37291, 37299
\int_while_do:nn ..... 162, 17489, 17489, 17494
\int_while_do:nNnn ..... 162, 17517, 17517, 17522
\int_zero:N ..... 158, 2214, 3829, 3830, 3831,
3930, 5091, 5313, 5751, 6278, 6351,
6382, 6806, 7085, 7592, 7593, 7641,
7828, 8102, 10668, 16707, 17317,
17317, 17319, 17322, 21017, 22191,
22348, 22377, 22508, 28841, 36112
\int_zero_new:N ..... 158, 17321, 17321, 17325
\c_max_char_int 167, 4824, 17981, 18889
\c_max_int ..... 167, 237, 523,
974, 1180, 4238, 4239, 4240, 17980,
```

28664, 33009, 33015, 34796, 34799  
`\c_max_register_int` .....  
     ..... 167, 419, 1482, 3347,  
     3372, 3409, 10008, 10010, 16700, 17176  
`\c_one_int` .. 167, 3931, 4242, 6596,  
     6607, 17348, 17350, 17352, 17354,  
     17979, 19645, 22406, 22425, 25888,  
     25892, 25896, 25950, 26541, 26684,  
     26825, 27131, 27971, 28055, 28522,  
     28526, 28533, 28718, 37289, 37297  
`\g_tmipa_int` ..... 167, 17991  
`\l_tmipa_int` ..... 4, 52, 167, 17991  
`\g_tmrb_int` ..... 167, 17991  
`\l_tmrb_int` ..... 4, 167, 17991  
`\c_zero_int` ... 167, 360, 371, 696,  
     1481, 1800, 1802, 3954, 3978, 4034,  
     4135, 4244, 4513, 5109, 5609, 6005,  
     6496, 6522, 6583, 6623, 6674, 7104,  
     7243, 7314, 7323, 7375, 7392, 7415,  
     7747, 7755, 8779, 8978, 9112, 9159,  
     11010, 11357, 12278, 13121, 13170,  
     13381, 13386, 13707, 13742, 14454,  
     15253, 17272, 17273, 17286, 17317,  
     17318, 17369, 17377, 17554, 17557,  
     17979, 18888, 19642, 19643, 19644,  
     20221, 20399, 22267, 22393, 22891,  
     23117, 23121, 23123, 23127, 23131,  
     23144, 23157, 23164, 23177, 23189,  
     23201, 23210, 23213, 23224, 23330,  
     23335, 24898, 24930, 25914, 25949,  
     25951, 25952, 25953, 26718, 26785,  
     27008, 27026, 27049, 27081, 27927,  
     28287, 28505, 28534, 28634, 28698  
 int internal commands:  
     `\__int_abs:N` .... 17205, 17207, 17211  
     `\__int_case:nnTF` ..... 17443,  
         17446, 17451, 17456, 17461, 17463  
     `\__int_case:nw` .....  
         ..... 17443, 17464, 17465, 17469  
     `\__int_case_end:nw` 17443, 17468, 17471  
     `\__int_compare:nnN` .....  
         . 825, 17382, 17414, 17422, 17424,  
             17426, 17428, 17430, 17432, 17434  
     `\__int_compare:NNw` .....  
         ..... 824, 825, 17382, 17394, 17398  
     `\__int_compare:Nw` .....  
         824, 825, 17382, 17390, 17392, 17419  
     `\__int_compare:w` .....  
         ..... 824, 17382, 17384, 17387  
     `\__int_compare_!=:NNw` ..... 17382  
     `\__int_compare_<:NNw` ..... 17382  
     `\__int_compare_<=:NNw` ..... 17382  
     `\__int_compare_=:NNw` ..... 17382  
     `\__int_compare_==:NNw` ..... 17382  
  
     `\__int_compare_>:NNw` ..... 17382  
     `\__int_compare_>=:NNw` ..... 17382  
     `\__int_compare_end_=:NNw` . 825, 17382  
     `\__int_compare_error:` ..... 823,  
         824, 17367, 17367, 17371, 17385, 17387  
     `\__int_compare_error:Nw` .....  
         ..... 823-825, 17367, 17373, 17407  
     `\__int_const:nN` . 17282, 17283, 17284  
     `\__int_constdef:Nw` ..... 17282,  
         17299, 17310, 17311, 17312, 17314  
     `\__int_div_truncate:NwNw` .....  
         ..... 17237, 17240, 17245, 17268  
     `\__int_eval:w` ..... 349,  
         818, 819, 824, 17176, 17177, 17189,  
         17190, 17194, 17208, 17216, 17217,  
         17224, 17225, 17239, 17241, 17242,  
         17259, 17262, 17263, 17264, 17271,  
         17302, 17336, 17338, 17340, 17342,  
         17360, 17362, 17385, 17419, 17437,  
         17475, 17483, 17548, 17549, 17550,  
         17576, 17759, 17786, 17792, 17819  
     `\__int_eval_end:` .....  
         ..... 17176, 17178, 17189,  
         17194, 17208, 17243, 17259, 17265,  
         17274, 17302, 17336, 17338, 17340,  
         17342, 17360, 17362, 17437, 17475,  
         17483, 17759, 17786, 17792, 17819  
     `\__int_from_alpha:N` .....  
         ..... 836, 17876, 17889, 17891  
     `\__int_from_alpha:nN` .....  
         .... 836, 17876, 17881, 17885, 17888  
     `\__int_from_base:N` .....  
         ..... 836, 17893, 17906, 17909  
     `\__int_from_base:nnN` .....  
         .... 836, 17893, 17898, 17902, 17905  
     `\__int_from_roman:NN` .....  
         ... 17935, 17941, 17946, 17962, 17966  
`\c_int_from_roman_C_int` ..... 17921  
`\c_int_from_roman_c_int` ..... 17921  
`\c_int_from_roman_D_int` ..... 17921  
`\c_int_from_roman_d_int` ..... 17921  
`\__int_from_roman_error:w` .....  
         ..... 17935, 17950, 17954, 17969  
`\c_int_from_roman_I_int` ..... 17921  
`\c_int_from_roman_i_int` ..... 17921  
`\c_int_from_roman_L_int` ..... 17921  
`\c_int_from_roman_l_int` ..... 17921  
`\c_int_from_roman_M_int` ..... 17921  
`\c_int_from_roman_m_int` ..... 17921  
`\c_int_from_roman_V_int` ..... 17921  
`\c_int_from_roman_v_int` ..... 17921  
`\c_int_from_roman_X_int` ..... 17921  
`\c_int_from_roman_x_int` ..... 17921

```

\__int_if_recursion_tail_stop:N . . . . . 17186, 17187, 17948
\__int_if_recursion_tail_stop_-do:Nn . . . . . 17186, 17186, 17887, 17904, 17951
\l__int_internal_a_int . . . . . 17995
\l__int_internal_b_int . . . . . 17995
\c__int_max_constdef_int . . . . . 17282
\__int_maxmin:wwN . . . . . 17205, 17215, 17223, 17229
\__int_mod:ww . . . . . 17237, 17262, 17267
\__int_pass_signs:wn . . . . . 835, 17866, 17866, 17869, 17880, 17897
\__int_pass_signs_end:wn . . . . . 17866, 17871, 17875
\__int_show:nN . . . . . 17971
\__int_sign:Nw . . . . . 17191, 17193, 17197
\__int_step:NNnnnn . . . . . 17582, 17589, 17600, 17609
\__int_step:NwnnN . . . . . 17545, 17555, 17563, 17568, 17574
\__int_step:wwwN . . . . . 17545, 17547, 17552
\__int_to_Base:nn . . . . . 17700, 17703, 17710
\__int_to_base:nn . . . . . 17700, 17701, 17704
\__int_to_Base:nnN . . . . . 17700, 17713, 17714, 17736, 17750
\__int_to_base:nnN . . . . . 17700, 17707, 17708, 17716, 17730
\__int_to_Base:nnnN . . . . . 17700, 17741, 17748
\__int_to_base:nnnN . . . . . 17700, 17721, 17728
\__int_to_Letter:n . . . . . 17700, 17739, 17742, 17789
\__int_to_letter:n . . . . . 17700, 17719, 17722, 17756
\__int_to_roman:N . . . . . 17830, 17832, 17835, 17838
\__int_to_roman:w . . . . . 824, 835, 1462, 1463, 17176, 17395, 17833, 17843
\__int_to_Roman_aux:N . . . . . 17842, 17845, 17848
\__int_to_Roman_c:w . . . . . 17830, 17862
\__int_to_roman_c:w . . . . . 17830, 17854
\__int_to_Roman_d:w . . . . . 17830, 17863
\__int_to_roman_d:w . . . . . 17830, 17855
\__int_to_Roman_i:w . . . . . 17830, 17858
\__int_to_roman_i:w . . . . . 17830, 17850
\__int_to_Roman_l:w . . . . . 17830, 17861
\__int_to_roman_l:w . . . . . 17830, 17853
\__int_to_Roman_m:w . . . . . 17830, 17864
\__int_to_roman_m:w . . . . . 17830, 17856
\__int_to_Roman_Q:w . . . . . 17830, 17865
\__int_to_roman_Q:w . . . . . 17830, 17857
\__int_to_Roman_v:w . . . . . 17830, 17859
\__int_to_roman_v:w . . . . . 17830, 17851
\__int_to_Roman_x:w . . . . . 17830, 17860
\__int_to_roman_x:w . . . . . 17830, 17852
\__int_to_symbols:nnnn . . . . . 17616, 17620, 17630
\__int_use_none_delimit_by_s_-stop:w . . . . . 17183, 17183, 17417
intarray commands:
\intarray_const_from_clist:Nn . . . . . 238, 22188, 22188, 22197, 22374, 22374, 22382, 26839, 27440
\intarray_count:N . . . . . 237, 238, 351, 14416, 14479, 22044, 22047, 22088, 22102, 22132, 22180, 22186, 22273, 22276, 22278, 22279, 22282, 22282, 22283, 22291, 22301, 22349, 22372, 22393, 22452, 22509, 28766
\intarray_gset:Nnn . . . . . 237, 351, 970, 972, 14400, 14412, 14425, 14431, 22124, 22127, 22135, 22304, 22306, 22313, 28933, 29677
\intarray_gset_rand:Nn . . . . . 307, 22457, 22457, 22459
\intarray_gset_rand:Nnn 307, 22457, 22458, 22462, 22517, 22520, 22523
\intarray_gzero:N . . . . . 238, 22137, 22146, 22346, 22346, 22355
\intarray_item:Nn . . . . . 238, 351, 966, 970, 972, 14410, 14415, 14449, 14478, 14486, 22148, 22175, 22184, 22186, 22356, 22358, 22364, 22372, 29133, 29704, 29718
\intarray_log:N . . . . . 238, 22442, 22444, 22445
\intarray_new:Nn 237, 963, 969, 972, 6763, 6764, 7575, 7576, 7577, 7578, 7579, 14398, 14407, 22033, 22040, 22050, 22260, 22269, 22281, 28758, 28759, 28760, 28931, 29492, 29655
\intarray_rand_item:N 238, 22185, 22185, 22187, 22371, 22371, 22373
\intarray_show:N . . . . . 238, 967, 972, 22442, 22442, 22443
\intarray_to_clist:N . . . . . 307, 22198, 22198, 22199, 22389, 22389, 22390
intarray internal commands:
\__intarray:w . . . . . 22027, 22038
\l__intarray_bad_index_int . . . . . 22024, 22132, 22180
\__intarray_bounds>NNnTF . . . . . 22286, 22286, 22316, 22367
\__intarray_bounds_error>NNnw . . . . . 22286, 22289, 22292, 22297

```

\\_\_intarray\_const\_from\_clist:nN . . . . .  
 ..... 22374, 22379, 22383  
 \\_\_intarray\_count:w . . . . . 22256,  
 22257, 22272, 22282, 22380, 22401  
 \\_\_intarray\_entry:w . . . . .  
 .. 22256, 22256, 22305, 22352, 22357  
 \g\_intarray\_font\_int . . . . .  
 ..... 22259, 22263, 22265  
 \\_\_intarray\_gset:Nnn . . . . . 22304  
 \\_\_intarray\_gset:Nnw . . . . . 22308, 22314  
 \\_\_intarray\_gset:w . . . . . 22104, 22126  
 \\_\_intarray\_gset:wTF . . . . . 22104, 22129  
 \\_\_intarray\_gset\_all\_same:Nn . . . . .  
 ..... 22457, 22493, 22506  
 \\_\_intarray\_gset\_count:Nw . . . . .  
 ..... 962, 22022, 22043, 22088  
 \\_\_intarray\_gset\_overflow:Nnn . . . . . 22304  
 \\_\_intarray\_gset\_overflow>NNnn . . . . .  
 ..... 22328, 22336, 22340  
 \\_\_intarray\_gset\_overflow\_-  
 test:nw . . . . . 968,  
 972, 974, 22250, 22251, 22318,  
 22325, 22333, 22386, 22476, 22483  
 \\_\_intarray\_gset\_rand:Nnn . . . . .  
 .. 22457, 22464, 22467, 22473, 22480  
 \\_\_intarray\_gset\_rand\_auxi:Nnnn . . . . .  
 ..... 22457, 22477, 22481  
 \\_\_intarray\_gset\_rand\_auxii:Nnnn  
 ..... 22457, 22484, 22486  
 \\_\_intarray\_gset\_rand\_auxiii:Nnnn  
 ..... 22457, 22488, 22491  
 \\_\_intarray\_gset\_range:nNw . . . . . 22224  
 \\_\_intarray\_gset\_range:Nw . . . . .  
 ..... 22427, 22430, 22432, 22439  
 \\_\_intarray\_gset\_range:w . . . . . 22227  
 \\_\_intarray\_item:Nn . . . . . 22356  
 \\_\_intarray\_item:Nw . . . . . 22360, 22365  
 \\_\_intarray\_item:w . . . . . 22148, 22174  
 \\_\_intarray\_item:wTF . . . . . 22148, 22177  
 \l\_intarray\_loop\_int 22021, 22191,  
 22194, 22195, 22348, 22351, 22352,  
 22377, 22380, 22385, 22387, 22429,  
 22437, 22438, 22508, 22511, 22512  
 \\_\_intarray\_new:N . . . . .  
 ..... 22033, 22034, 22042,  
 22190, 22260, 22260, 22271, 22376  
 \\_\_intarray\_range\_to\_clist:w . . . . .  
 ..... 22209, 22212  
 \\_\_intarray\_range\_to\_clist:ww . . . . .  
 ..... 22408, 22412, 22418, 22424  
 \\_\_intarray\_show>NN . . . . .  
 ..... 22442, 22444, 22446  
 \\_\_intarray\_signed\_max\_dim:n . . . . .  
 ..... 22284, 22284, 22343, 22344  
 \c\_intarray\_sp\_dim . . . . .  
 ..... 22258, 22265, 22305  
 \\_\_intarray\_table . . . . . 22062  
 \g\_intarray\_table\_int . . . . .  
 ..... 22024, 22037, 22038  
 \\_\_intarray\_to\_clist:Nn 967, 22198,  
 22198, 22389, 22389, 22391, 22453  
 \\_\_intarray\_to\_clist:w . . . . .  
 .. 22198, 22389, 22396, 22399, 22405  
 \interactionmode . . . . . 543  
 \interlinepenalties . . . . . 544  
 \interlinepenalty . . . . . 324  
 ior commands:  
 \ior\_close:N . . . . . 86, 307,  
 10225, 10244, 10244, 10255, 11156,  
 11169, 11603, 29754, 29788, 29858  
 \ior\_get:NN . . . . . 87,  
 88, 307, 10301, 10301, 10305, 10381  
 \ior\_get:NNTF . . . . . 87, 10301, 10302  
 \ior\_get\_str>NN . . . . . 37518  
 \ior\_get\_term:nN . . . . . 307, 10335, 10335  
 \ior\_if\_eof:N . . . . . 627, 10285  
 \ior\_if\_eof:NFT . . . . . 90, 10285, 10307,  
 10327, 10367, 10386, 11166, 11180  
 \ior\_if\_eof\_p:N . . . . . 90, 10285  
 \ior\_list\_streams: . . . . . 37520  
 \ior\_log:N . . . . . 86, 10256, 10258, 10259  
 \ior\_log\_list: . . . . . 87, 10272, 10273, 37523  
 \ior\_log\_streams: . . . . . 37522  
 \ior\_map\_break: . . . . . 89,  
 10350, 10350, 10351, 10353, 10368,  
 10375, 10387, 10393, 29689, 29784  
 \ior\_map\_break:n . . . . . 89, 10350, 10352  
 \ior\_map\_inline:Nn . . . . .  
 ..... 88, 10354, 10354, 11601  
 \ior\_map\_variable>NNn . . . . .  
 ..... 88, 10380, 10380, 29686  
 \ior\_new:N . . . . . 86, 10194, 10194, 10195,  
 10196, 10197, 11182, 29484, 29823  
 \ior\_open:Nn . . . . . 86,  
 656, 10198, 10198, 10200, 10202,  
 10211, 29683, 29722, 29755, 29825  
 \ior\_open:NNTF . . . . . 86, 10199, 10202  
 \ior\_shell\_open:Nn . . . . .  
 ..... 307, 11590, 37147, 37147  
 \ior\_show:N . . . . . 86, 10256, 10256, 10257  
 \ior\_show\_list: . . . . . 87, 10272, 10272, 37521  
 \ior\_str\_get>NN . . . . . 87, 88,  
 307, 10314, 10314, 10325, 10383, 37519  
 \ior\_str\_get:NNTF . . . . . 88, 10314, 10315  
 \ior\_str\_get\_term:nN . . . . . 307, 10335, 10337  
 \ior\_str\_map\_inline:Nn . . . . . 88,  
 89, 10354, 10356, 29748, 29777, 29850

\ior\_str\_map\_variable:N<sub>Nn</sub> ..... 89, [10380](#), 10382  
 ..... 37484  
 \g\_tmpa\_ior ..... 93, [10196](#)  
 \g\_tmpb\_ior ..... 93, [10196](#)

ior internal commands:

- \l\_iор\_file\_name\_t1 ..... 10201, 10204, 10206
- \\_\_ior\_get:NN ..... 10301, 10303, 10310, 10336, 10355
- \\_\_ior\_get\_term:N<sub>nN</sub> ..... 10335, 10336, 10338, 10339
- \l\_iор\_internal\_t1 ..... 10176, 10264, 10267, 10373, 10377
- \\_\_ior\_list:N ..... 10272, 10272, 10273, 10274
- \\_\_ior\_map\_inline:N<sub>Nn</sub> ..... 10354, 10355, 10357, 10358
- \\_\_ior\_map\_inline:NN<sub>Nn</sub> ..... 10354, 10361, 10364
- \\_\_ior\_map\_inline\_loop:NNN ..... 10354, 10367, 10371, 10378
- \\_\_ior\_map\_variable:NN<sub>Nn</sub> ..... 10380, 10381, 10383, 10384
- \\_\_ior\_map\_variable\_loop:NNN<sub>n</sub> ..... 10380, 10386, 10389, 10396
- \\_\_ior\_new:N ..... 623, [10212](#), 10212, 10216, 10217, 10229
- \\_\_ior\_new\_aux:N ..... 10216, 10220
- \\_\_ior\_open\_stream:N<sub>n</sub> ..... 10223, 10227, 10231, 10235
- \\_\_ior\_shell\_open:nN ..... 37147, 37150, 37153
- \\_\_ior\_show>NN ..... 10256, 10256, 10258, 10260
- \\_\_ior\_str\_get:NN ..... 10314, 10316, 10330, 10338, 10357
- \l\_iор\_stream\_t1 ..... 10179, 10226, 10230, 10237
- \g\_iор\_streams\_prop ..... 624, [10180](#), 10238, 10249, 10264, 10279
- \g\_iор\_streams\_seq ..... 10178, 10226, 10250, 10251
- \c\_iор\_term\_ior ..... 10177, 10194, 10246, 10252, 10288, 10345
- \c\_iор\_term\_noprompt\_ior ..... 10334, 10344

iow commands:

- \iow\_allow\_break: . 92, 307, [10575](#), 10575, 10578, 10584, 10617, 10622
- \iow\_allow\_break:n ..... 634
- \iow\_char:N 80, 91, 3705, 3708, 3709, 3733, 3734, 3741, 3742, 4798, 4799, 4806, 4808, 4810, 4812, 4814, 4816,
- 5459, 5460, 6194, 6201, 6202, 6203, 6327, 8145, 8148, 8149, 8154, 8188, 8197, 8201, 8206, 8226, 8228, 8229, 8231, 8234, 8236, 8241, 8243, 8245, 8250, 8254, 8257, 8258, 8261, 8263, 8267, 8269, 8275, 8277, 8281, 8283, 8287, 8292, 8294, 8336, 8338, 8343, 8345, 8351, 8356, 8361, 8365, 8375, 8378, 8382, 8383, 8387, 8395, 8466, 9867, 9870, 9871, 9903, 9931, 10063, 10536, 10536, 11664, 11666, 11667, 11668, 14518, 26945, 29259, 29261, 29262, 29265, 29267, 29268, 29271, 29273, 29274, 29275, 29279, 29286
- \iow\_close:N ..... 86, [10449](#), [10467](#), 10467, 10478
- \iow\_indent:n ..... 92, 635, 636, 8422, 9824, 9963, 10025, 10033, 10041, [10586](#), 10586, 10589, 10601, 10618, 10623, 14516, 14841, 15029, 23060, 23072, 36424, 36453, 36475, 36491, 36500, 36509, 36530
- \l\_iow\_line\_count\_int ..... 92, 448, 636, 4184, 4188, 9377, [10537](#), 10627, 10632, 10670
- \iow\_list\_streams: ..... 37524
- \iow\_log:N .. 86, [10479](#), 10481, 10482
- \iow\_log:n 90, [1910](#), 1910, 9585, 9592, [10531](#), 10531, 10532, 13221, 37545
- \iow\_log\_list: 87, [10495](#), 10496, 37527
- \iow\_log\_streams: ..... 37526
- \iow\_new:N ..... 86, [10429](#), 10429, 10430, 10431, 10432
- \iow\_newline: ..... 80, 90–92, 352, 602, 632, 710, 4134, 6249, 9399, [10535](#), 10535, 10615, 10624, 10630, 11519, 34783, 34784, 34785, 36858, 36860, 36863, 36870
- \iow\_now:N<sub>n</sub> ..... 90, 91, 9025, [10525](#), 10525, 10530, 10531, 10532, 10533, 10534
- \iow\_open:N<sub>n</sub> .. 86, [10445](#), 10445, 10458
- \iow\_shipout:N<sub>n</sub> ..... 90, 91, 632, 9057, [10510](#), 10510, 10512
- \iow\_shipout\_x:N<sub>n</sub> ..... 90, 91, 632, [10507](#), 10507, 10509
- \iow\_show:N .. 86, [10479](#), 10479, 10480
- \iow\_show\_list: 87, [10495](#), 10495, 37525
- \iow\_term:n ..... 90, 307, 1910, 1912, 8456, 9411, 9575, 9580, 9598, 9624, [10531](#), 10533, 10534, 37547
- \iow\_wrap:nnnN ..... 90, 92, 307, 636, 710, 1393, 9375, 9378, 9390, 9556, 9590, 9596,

9603, 10578, 10584, 10589, 10601,  
~~10604~~, 10604, 10638, 13206, 13221  
~~\c\_log\_iow~~ .....  
~~93, 628, 10400, 10469, 10531, 10532~~  
~~\c\_term\_iow~~ .....  
~~93, 628, 10400, 10429, 10469, 10475, 10533, 10534~~  
~~\g\_tmtpa\_iow~~ .....  
~~\g\_tmtpb\_iow~~ .....  
 iow internal commands:  
~~\\_\_iow\_allow\_break:~~ .....  
~~634, 10575, 10580, 10617~~  
~~\\_\_iow\_allow\_break\_error:~~ .....  
~~634, 10575, 10581, 10622~~  
~~\l\_\_iow\_file\_name\_tl~~ .....  
~~10444, 10447, 10451, 10455~~  
~~\\_\_iow\_indent:n~~ .....  
~~635, 10586, 10592, 10618~~  
~~\\_\_iow\_indent\_error:n~~ .....  
~~635, 10586, 10598, 10623~~  
~~\l\_\_iow\_indent\_int~~ .....  
~~10554, 10668, 10686, 10798, 10815, 10823~~  
~~\l\_\_iow\_indent\_tl~~ ..  
~~10554, 10669, 10685, 10797, 10816, 10824, 10825~~  
~~\l\_\_iow\_internal\_tl~~ .....  
~~10399, 10487, 10490~~  
~~\l\_\_iow\_line\_break\_bool~~ .....  
~~10558, 10664, 10792, 10806, 10814, 10822, 10830, 10832, 10837, 10839~~  
~~\l\_\_iow\_line\_part\_tl~~ .....  
~~638-640, 10556, 10666, 10678, 10699, 10757, 10760, 10791, 10805, 10807, 10813, 10821, 10844~~  
~~\l\_\_iow\_line\_target\_int~~ .....  
~~641, 10540, 10626, 10628, 10631, 10793, 10798, 10833~~  
~~\l\_\_iow\_line\_tl~~ ~~10556, 10665, 10682, 10772, 10788, 10804, 10805, 10813, 10821, 10843, 10844, 10849, 10851~~  
~~\\_\_iow\_list:N~~ .....  
~~10495, 10495, 10496, 10497~~  
~~\\_\_iow\_new:N~~ .....  
~~10433, 10433, 10437, 10438, 10453~~  
~~\\_\_iow\_new\_aux:N~~ .....  
~~10437, 10441~~  
~~\l\_\_iow\_newline\_tl~~ .....  
~~10539, 10624, 10625, 10627, 10630, 10848~~  
~~\l\_\_iow\_one\_indent\_int~~ .....  
~~10541, 10815, 10823~~  
~~\l\_\_iow\_one\_indent\_tl~~ .....  
~~633, 10541, 10816~~  
~~\\_\_iow\_open\_stream:Nn~~ .....  
~~10445, 10451, 10455, 10459, 10466~~  
~~\\_\_iow\_set\_indent:n~~ .....  
~~633, 10541, 10544, 10553~~  
~~\\_\_iow\_show>NN~~ .....  
~~10479, 10479, 10481, 10483~~  
~~\l\_\_iow\_stream\_tl~~ .....  
~~10410, 10450, 10454, 10461~~  
~~\g\_\_iow\_streams\_prop~~ .....  
~~631, 10411, 10462, 10472, 10487, 10502~~  
~~\g\_\_iow\_streams\_seq~~ .....  
~~10409, 10450, 10473, 10474~~  
~~\\_\_iow\_tmp:w~~ .....  
~~639, 10672, 10696, 10753, 10785, 10853, 10861~~  
~~\\_\_iow\_unindent:w~~ .....  
~~633, 10541, 10543, 10551, 10825~~  
~~\\_\_iow\_use\_i\_delimit\_by\_s\_-stop:nw~~ .....  
~~10427, 10427, 10657~~  
~~\\_\_iow\_with:nNnn~~ ..  
~~10513, 10517, 10519~~  
~~\\_\_iow\_wrap\_allow\_break:n~~ .....  
~~10802, 10802~~  
~~\c\_\_iow\_wrap\_allow\_break\_marker\_-tl~~ .....  
~~10560, 10580~~  
~~\\_\_iow\_wrap\_break:w~~ .....  
~~10739, 10753, 10755~~  
~~\\_\_iow\_wrap\_break\_end:w~~ .....  
~~639, 10753, 10762, 10782~~  
~~\\_\_iow\_wrap\_break\_first:w~~ .....  
~~10753, 10759, 10765~~  
~~\\_\_iow\_wrap\_break\_loop:w~~ .....  
~~10753, 10768, 10776, 10780~~  
~~\\_\_iow\_wrap\_break\_none:w~~ .....  
~~10753, 10767, 10770~~  
~~\\_\_iow\_wrap\_chunk:nw~~ .....  
~~10670, 10672, 10674, 10808, 10809, 10817, 10826, 10833~~  
~~\\_\_iow\_wrap\_do:~~ ..  
~~10634, 10639, 10639~~  
~~\\_\_iow\_wrap\_end:n~~ .....  
~~10828, 10835~~  
~~\\_\_iow\_wrap\_end\_chunk:w~~ .....  
~~637, 10690, 10697, 10747, 10789~~  
~~\c\_\_iow\_wrap\_end\_marker\_tl~~ .....  
~~10560, 10644~~  
~~\\_\_iow\_wrap\_fix\_newline:w~~ .....  
~~10639, 10648, 10653, 10660~~  
~~\\_\_iow\_wrap\_indent:n~~ ..  
~~10811, 10811~~  
~~\c\_\_iow\_wrap\_indent\_marker\_tl~~ ..  
~~10560, 10594~~  
~~\\_\_iow\_wrap\_line:nw~~ .....  
~~637, 640, 10684, 10688, 10697, 10697, 10796~~  
~~\\_\_iow\_wrap\_line\_aux:Nw~~ .....  
~~10697, 10707, 10713~~  
~~\\_\_iow\_wrap\_line\_end:NnnnnnnnN~~ ..  
~~10697, 10716, 10733~~  
~~\\_\_iow\_wrap\_line\_end:nw~~ ~~639, 10697, 10738, 10741, 10773, 10774, 10783~~  
~~\\_\_iow\_wrap\_line\_loop:w~~ .....  
~~10697, 10701, 10704, 10710~~

```

\__iow_wrap_line_seven:nnnnnn . . . . . 10697, 10728, 10732
\c__iow_wrap_marker_tl . . . . . 634, 637, 10560, 10696
\__iow_wrap_newline:n . . . . . 10828, 10828
\c__iow_wrap_newline_marker_tl . . . . . 636, 10560, 10659
\__iow_wrap_next:nw . . . . . 10672, 10679, 10693, 10751, 10793
\__iow_wrap_next_line:w . . . . . 10745, 10786, 10786
\__iow_wrap_start:w . . . . . 10639, 10651, 10662
\__iow_wrap_store_do:n . . . . . 10744, 10831, 10838, 10841, 10841
\l__iow_wrap_tl . . . . . 636, 641, 642, 10559, 10621, 10636, 10641, 10643, 10646, 10648, 10651, 10667, 10845, 10847
\__iow_wrap_trim:N . . . . . 642, 10774, 10805, 10831, 10838, 10853, 10855
\__iow_wrap_trim:w . . . . . 10853, 10856, 10857
\__iow_wrap_trim_aux:w . . . . . 10853, 10858, 10859
\__iow_wrap_unindent:n . . . . . 10811, 10819
\c__iow_wrap_unindent_marker_tl . . . . . 10560, 10596
\itshape . . . . . 32507

J
\j . . . . . 31883, 32595, 32790, 32869
\jcharwidowpenalty . . . . . 1134
\jfam . . . . . 1135
\jfont . . . . . 1136
\jis . . . . . 1137
\jobname . . . . . 325

K
\k . . . . . 30140, 32668, 32743, 32744, 32761, 32762, 32784, 32785, 32786, 32841, 32842, 32867, 32868
\kanjiskip . . . . . 1138
\kansuji . . . . . 1139
\kansujichar . . . . . 1140
\kcatcode . . . . . 1141
\kchar . . . . . 1178
\kchardef . . . . . 1179
\kern . . . . . 326
kernel internal commands:
    \__kernel_backend_align_begin: . . . . . 356
    \__kernel_backend_align_end: . . . . . 356
    \g__kernel_backend_header_bool . . . . . 356
    \__kernel_backend_literal:n . . . . . 356
    \__kernel_backend_literal_pdf:n . . . . . 356
    \__kernel_backend_literal_-
        postscript:n . . . . . 356
    \__kernel_backend_literal_svg:n . . . . . 356
    \__kernel_backend_matrix:n . . . . . 356
    \__kernel_backend_postscript:n . . . . . 356
    \__kernel_backend_scope_begin: . . . . . 356
    \__kernel_backend_scope_end: . . . . . 356
    \__kernel_chk_cs_exist:N . . . . . 349
    \__kernel_chk_defined:NTF . . . . .
        349, 2170, 2170, 2189, 8536, 10262, 10485, 13191, 13224, 18023, 22448
    \__kernel_chk_expr:nNnN . . . . . 349
    \__kernel_chk_if_free_cs:N . . . . .
        . . . . . 599, 874, 1914, 1914, 1922, 1923, 1929, 1977, 8477, 12084, 12090, 12095, 16169, 16483, 17278, 17298, 19108, 19110, 19120, 19694, 20090, 20433, 20524, 22036, 22262, 28920, 28936, 32902
    \__kernel_chk_tl_type:NnnTF . . . . .
        . . . . . 349, 815, 863, 904, 7468, 13222, 13222, 13978, 13985, 17150, 18716, 20060, 24784
    \__kernel_codepoint_case:nn . . . . .
        . . . . . 355, 13935, 18998, 19082, 29790, 29790, 31021
    \__kernel_codepoint_data:nn . . . . .
        . . . . . 355, 29696, 29696, 29727, 29810
    \__kernel_codepoint_to_bytes:n . . . . .
        . . . . . 350, 15300, 29327, 29359, 29387, 29387, 32566, 32643, 37680
    \l__kernel_color_stack_int . . . . . 357
    \__kernel_cs_parm_from_arg_-
        count:nnTF . . . . .
        . . . . . 350, 1636, 1995, 1995, 2042
    \__kernel_dependency_version_-
        check:Nn . . . . . 350, 11569, 11569
    \__kernel_dependency_version_-
        check:nn . . . . . 350, 11569, 11570, 11571
    \__kernel_deprecation_code:nn . . . .
        . . . . . 350, 1408, 1578, 1580, 37363, 37389, 37396, 37397
    \__kernel_deprecation_error:Nnn . . .
        . . . . . 1408, 37366, 37399, 37399
    \__kernel_exp_not:w . . . . .
        . . . . . 350, 396, 434, 451, 697, 899, 2590, 2590, 2592, 2596, 2600, 2603, 2606, 2611, 4261, 12091, 12122, 12123, 12130, 12131, 12145, 12147, 12151, 12153, 12167, 12172, 12178, 12179, 12183, 12187, 12192, 12198, 12199, 12203, 12213, 12217, 12223, 12224, 12228, 12230, 12234, 12240, 12241, 12245, 12404, 12743,

```

12748, 12804, 12809, 12816, 12841,  
 13034, 13197, 17400, 19860, 19868,  
 19875, 19880, 20605, 29313, 29352,  
 29366, 29383, 30212, 30627, 32278  
 \l\_kernel\_expl\_bool .....  
     ..... 142, 145, 160, 174, 1428  
 \c\_kernel\_expl\_date\_t1 .....  
     662, 1428, 11573, 11576, 11616, 11620  
 \kernel\_file\_input\_pop: .....  
     ..... 351, 11400, 11439  
 \kernel\_file\_input\_push:n ..  
     ..... 351, 11400, 11433  
 \kernel\_file\_missing:n ..  
     .... 350, 10199, 11395, 11395, 11404  
 \kernel\_file\_name\_quote:n ..  
     ..... 1401, 10242,  
         10464, 10981, 10981, 11020, 11415  
 \kernel\_file\_name\_sanitize:n ..  
     ..... 350, 657, 10448, 10907,  
         10907, 11034, 11143, 11398, 11454  
 \kernel\_group\_show:NN .....  
     ..... 2211, 2212, 2214, 2215  
 \kernel\_if\_debug:TF .....  
     ..... 1565, 1565, 37377  
 \kernel\_int\_add:nnn .....  
     ..... 351, 17269, 17269, 28664  
 \kernel\_intarray\_gset:Nnn ....  
     ..... 351, 965, 967, 970, 6803,  
         6910, 6913, 7604, 7676, 7678, 7684,  
         7692, 7694, 7697, 7818, 7820, 7824,  
         7826, 7838, 7841, 22124, 22125,  
         22195, 22267, 22279, 22304, 22304,  
         22319, 22387, 22437, 22512, 28828,  
         28829, 28831, 28835, 28836, 28837,  
         28939, 28940, 28974, 28977, 29636  
 \kernel\_intarray\_gset\_range\_-  
     from\_clist:Nnn ..... 351,  
         6973, 22224, 22225, 22427, 22427  
 \kernel\_intarray\_item:Nn .. 351,  
     966, 971, 1153, 4518, 6921, 6948,  
     7032, 7033, 7057, 7058, 7066, 7073,  
     7130, 7134, 7153, 7681, 7871, 8090,  
     22148, 22173, 22356, 22356, 22368,  
     22404, 22423, 26925, 26931, 26934,  
     26937, 27670, 27673, 27676, 27679,  
     27682, 27685, 27688, 27691, 27694,  
     28877, 28878, 28879, 29026, 29029  
 \kernel\_intarray\_range\_to\_-  
     clist:Nnn ..... 351,  
         6902, 22209, 22210, 22408, 22408  
 \kernel\_ior\_open:Nn .....  
     .... 352, 1401, 10206, 10223,  
         10223, 10234, 11164, 11179, 37160  
 \\_\_kernel\_iow\_with:Nnn . 352, 602,  
     632, 710, 9412, 9414, 9626, 9628,  
     10513, 10513, 10527, 13210, 13212  
 \\_\_kernel\_kern:n 352, 1428, 32898,  
     32898, 33270, 33560, 33569, 34134,  
     34392, 34397, 34479, 34480, 34758,  
     34759, 36750, 36752, 36801, 36803  
 \\_\_kernel\_keyval\_allow\_blank\_-  
     keys\_bool ..... 19756,  
         19764, 19774, 19776, 20597, 20770  
 \\_\_kernel\_msg\_error:nnn .. 9774, 9780  
 \\_\_kernel\_msg\_error:nnnn . 9774, 9782  
 \\_\_kernel\_msg\_error:nnnnn 9774, 9784  
 \\_\_kernel\_msg\_expandable\_-  
     error:nnn ..... 9786, 9786, 9788  
 \\_\_kernel\_msg\_expandable\_-  
     error:nnnn ..... 9786, 9790  
 \\_\_kernel\_msg\_info:nnnn .. 9774, 9774  
 \\_\_kernel\_msg\_new:nnn 9770, 9772, 9996  
 \\_\_kernel\_msg\_new:nnnn ... 9770, 9770  
 \\_\_kernel\_msg\_warning:nnn 9774, 9776  
 \\_\_kernel\_msg\_warning:nnnn 9774, 9778  
 \\_\_kernel\_patch\_deprecation:nnNpn  
     .... 1408, 37359, 37359, 37418,  
         37423, 37631, 37634, 37636, 37638,  
         37640, 37642, 37644, 37646, 37648,  
         37650, 37653, 37655, 37658, 37661,  
         37664, 37667, 37670, 37673, 37676,  
         37679, 37681, 37683, 37685, 37687,  
         37689, 37691, 37693, 37695, 37697,  
         37699, 37701, 37703, 37705, 37707,  
         37709, 37711, 37722, 37728, 37734  
 \\_\_kernel\_prefix\_arg\_replacement:wN  
     .... 2232, 2234, 2242, 2251, 2260  
 \g\_kernel\_prg\_map\_int . 352, 446,  
     694, 829, 913, 1428, 4083, 4085,  
     4094, 4211, 4213, 4230, 8780, 10360,  
     10362, 10369, 12640, 12642, 12644,  
     12649, 13509, 13511, 13515, 13520,  
     16947, 16948, 16954, 16955, 17005,  
     17007, 17009, 17011, 17588, 17591,  
     17599, 17602, 17613, 18470, 18472,  
     18474, 18479, 20015, 20016, 20021,  
     20023, 20343, 20346, 20350, 20353,  
     20364, 25084, 25087, 25091, 25094,  
     25105, 32264, 32266, 32268, 32270  
 \\_\_kernel\_primitive:NN .....  
     .... 322, 182, 182, 187, 188,  
         189, 190, 191, 192, 193, 194, 195,  
         196, 197, 198, 199, 200, 201, 202,  
         203, 204, 205, 206, 207, 208, 209,  
         210, 211, 212, 213, 214, 215, 216,  
         217, 218, 219, 220, 221, 222, 223,  
         224, 225, 226, 227, 228, 229, 230,

- 231, 232, 233, 234, 235, 236, 237,  
238, 239, 240, 241, 242, 243, 244,  
245, 246, 247, 248, 249, 250, 251,  
252, 253, 254, 255, 256, 257, 258,  
259, 260, 261, 262, 263, 264, 265,  
266, 267, 268, 269, 270, 271, 272,  
273, 274, 275, 276, 277, 278, 279,  
280, 281, 282, 283, 284, 285, 286,  
287, 288, 289, 290, 291, 292, 293,  
294, 295, 296, 297, 298, 299, 300,  
301, 302, 303, 304, 305, 306, 307,  
308, 309, 310, 311, 312, 313, 314,  
315, 316, 317, 318, 319, 320, 321,  
322, 323, 324, 325, 326, 327, 328,  
329, 330, 331, 332, 333, 334, 335,  
336, 337, 338, 339, 340, 341, 342,  
343, 344, 345, 346, 347, 348, 349,  
350, 351, 352, 353, 354, 355, 356,  
357, 358, 359, 360, 361, 362, 363,  
364, 365, 366, 367, 368, 369, 370,  
371, 372, 373, 374, 375, 376, 377,  
378, 379, 380, 381, 382, 383, 384,  
385, 386, 387, 388, 389, 390, 391,  
392, 393, 394, 395, 396, 397, 398,  
399, 400, 401, 402, 403, 404, 405,  
406, 407, 408, 409, 410, 411, 412,  
413, 414, 415, 416, 417, 418, 419,  
420, 421, 422, 423, 424, 425, 426,  
427, 428, 429, 430, 431, 432, 433,  
434, 435, 436, 437, 438, 439, 440,  
441, 442, 443, 444, 445, 446, 447,  
448, 449, 450, 451, 452, 453, 454,  
455, 456, 457, 458, 459, 460, 461,  
462, 463, 464, 465, 466, 467, 468,  
469, 470, 471, 472, 473, 474, 475,  
476, 477, 478, 479, 480, 481, 482,  
483, 484, 485, 486, 487, 488, 489,  
490, 491, 492, 493, 494, 495, 496,  
497, 498, 499, 500, 501, 502, 503,  
504, 505, 506, 507, 508, 509, 510,  
511, 512, 513, 514, 515, 516, 517,  
518, 519, 520, 521, 522, 523, 524,  
525, 526, 527, 528, 529, 530, 531,  
532, 533, 534, 535, 536, 537, 538,  
539, 540, 541, 542, 543, 544, 545,  
546, 547, 548, 549, 550, 551, 552,  
553, 554, 555, 556, 557, 558, 559,  
560, 561, 562, 563, 564, 565, 566,  
567, 568, 569, 570, 571, 572, 573,  
574, 575, 576, 577, 578, 579, 580,  
581, 582, 583, 584, 585, 586, 587,  
588, 589, 590, 591, 592, 593, 594,  
595, 596, 597, 598, 599, 600, 601,  
602, 604, 605, 606, 607, 608, 609,  
610, 612, 613, 614, 615, 616, 617,  
618, 619, 620, 621, 622, 623, 624,  
625, 626, 627, 628, 629, 630, 631,  
632, 633, 634, 635, 636, 637, 638,  
639, 640, 641, 642, 643, 644, 645,  
646, 647, 648, 649, 650, 651, 652,  
653, 654, 655, 656, 657, 658, 659,  
660, 661, 662, 663, 664, 665, 666,  
667, 668, 669, 670, 671, 672, 673,  
674, 675, 676, 677, 678, 679, 680,  
681, 682, 683, 684, 685, 686, 687,  
688, 689, 690, 691, 692, 693, 694,  
699, 708, 709, 710, 711, 712, 713,  
715, 716, 717, 718, 719, 720, 721,  
722, 723, 724, 725, 727, 729, 731,  
732, 733, 735, 736, 737, 738, 739,  
740, 742, 744, 745, 747, 749, 750,  
751, 752, 753, 754, 755, 756, 757,  
758, 759, 760, 761, 762, 763, 764,  
765, 766, 767, 768, 769, 770, 771,  
772, 773, 774, 775, 776, 777, 778,  
779, 780, 781, 782, 783, 784, 785,  
786, 787, 789, 790, 792, 793, 794,  
795, 796, 797, 798, 799, 800, 801,  
803, 804, 805, 806, 808, 809, 810,  
811, 812, 813, 814, 815, 816, 817,  
818, 819, 820, 821, 822, 823, 824,  
825, 826, 827, 828, 829, 830, 831,  
832, 833, 834, 835, 836, 837, 838,  
839, 840, 841, 842, 843, 844, 845,  
846, 847, 848, 849, 850, 851, 852,  
853, 854, 855, 856, 857, 858, 859,  
860, 861, 862, 863, 864, 865, 866,  
867, 868, 869, 870, 871, 872, 873,  
874, 875, 876, 877, 878, 879, 880,  
881, 882, 883, 884, 885, 886, 887,  
888, 889, 890, 891, 892, 893, 894,  
895, 896, 897, 898, 899, 900, 901,  
902, 903, 904, 906, 907, 908, 909,  
910, 911, 912, 913, 914, 915, 916,  
917, 918, 919, 920, 922, 924, 926,  
927, 928, 929, 930, 931, 932, 933,  
934, 935, 936, 937, 938, 939, 940,  
941, 942, 943, 944, 945, 946, 947,  
948, 949, 950, 951, 952, 953, 954,  
955, 956, 957, 958, 959, 960, 961,  
962, 963, 964, 965, 966, 967, 968,  
969, 971, 973, 974, 975, 976, 978,  
979, 980, 981, 983, 984, 986, 988,  
989, 990, 991, 992, 994, 996, 997,  
998, 999, 1001, 1002, 1003, 1004,  
1005, 1006, 1007, 1008, 1009, 1010,  
1011, 1012, 1013, 1014, 1015, 1016,  
1017, 1018, 1019, 1020, 1021, 1022,

```

1023, 1024, 1025, 1026, 1027, 1028,
1029, 1030, 1031, 1032, 1033, 1034,
1035, 1036, 1037, 1038, 1040, 1042,
1043, 1045, 1047, 1048, 1049, 1050,
1052, 1053, 1054, 1056, 1058, 1060,
1061, 1062, 1063, 1064, 1065, 1066,
1067, 1068, 1069, 1070, 1071, 1073,
1075, 1076, 1077, 1078, 1079, 1080,
1081, 1082, 1083, 1084, 1085, 1086,
1087, 1088, 1089, 1090, 1091, 1093,
1095, 1096, 1097, 1098, 1099, 1100,
1101, 1102, 1103, 1104, 1105, 1106,
1107, 1108, 1109, 1110, 1111, 1112,
1113, 1114, 1115, 1116, 1117, 1118,
1119, 1120, 1121, 1122, 1123, 1124,
1125, 1126, 1127, 1128, 1129, 1130,
1131, 1132, 1133, 1134, 1135, 1136,
1137, 1138, 1139, 1140, 1141, 1142,
1143, 1144, 1145, 1146, 1147, 1148,
1149, 1150, 1151, 1152, 1153, 1154,
1155, 1156, 1158, 1160, 1161, 1162,
1163, 1164, 1166, 1167, 1168, 1169,
1170, 1171, 1172, 1173, 1174, 1175,
1176, 1177, 1178, 1179, 1180, 1181,
1182, 1183, 1184, 1185, 1186, 1187,
1188, 1189, 1190, 1191, 1192, 1193
\__kernel_quark_new_conditional:Nn
..... 354, 4561, 10902, 12260,
16269, 16289, 18749, 20860, 29866
\__kernel_quark_new_test:N
..... 353, 789, 790, 792,
793, 8513, 10905, 10906, 12259,
13261, 13262, 16269, 16269, 17186,
17187, 19690, 29871, 29872, 32275
\__kernel_randint:n
.... 354, 974, 1181, 1185, 22502,
28471, 28471, 28483, 28641, 28726
\__kernel_randint:nn
..... 354,
974, 22498, 28645, 28649, 28649, 28724
\c_kernel_randint_max_int
..... 1184, 1428, 22495, 28470, 28639, 28723
\__kernel_register_log:N
..... 354, 2179,
2183, 2185, 2186, 17975, 20419,
20420, 20512, 20513, 20580, 20581
\__kernel_register_show:N
..... 354, 709, 2179, 2179,
2181, 2182, 17971, 20415, 20508, 20576
\__kernel_register_show_aux:NN ..
..... 2179, 2180, 2184, 2187
\__kernel_register_show_aux:nNN .
..... 2179, 2191, 2195
\__kernel_show:NN
..... 2197, 2197, 2200, 2203
\__kernel_str_to_other:n 355, 718,
721, 725, 13565, 13565, 13617, 13678
\__kernel_str_to_other_fast:n ...
... 355, 4772, 5982, 10547, 10643,
13516, 13536, 13588, 13588, 14192
\__kernel_str_to_other_fast_-
loop:w ..... 13588
\__kernel_sys_configuration_-
load:n ..... 8885, 8951
\__kernel_tl_gset:Nn
..... 355, 553, 676, 3441, 3965, 4771,
5980, 7713, 7724, 7775, 7930, 9214,
12080, 12081, 12128, 12151, 12153,
12155, 12186, 12191, 12196, 12203,
12230, 12233, 12238, 12245, 12363,
12367, 12757, 13054, 13317, 13321,
14126, 14142, 14192, 14320, 14372,
14383, 14541, 14590, 14643, 14649,
14882, 15082, 15239, 16519, 16524,
16542, 16546, 16602, 16642, 16668,
16825, 16868, 17034, 17044, 18138,
18165, 18184, 18227, 18263, 18318,
18357, 19915, 19938, 24748, 36964,
36974, 36989, 36995, 37069, 37264
\__kernel_tl_set:Nn 355, 4661, 5550,
5555, 5826, 5895, 7865, 7899, 10230,
10447, 10454, 10546, 10621, 10624,
10625, 10641, 10646, 10804, 10824,
10843, 10845, 11129, 11142, 11177,
11284, 11318, 12080, 12080, 12120,
12145, 12147, 12149, 12166, 12171,
12176, 12183, 12213, 12216, 12221,
12228, 12361, 12365, 12755, 13052,
13315, 13319, 14081, 16509, 16514,
16540, 16544, 16566, 16594, 16640,
16666, 16781, 16806, 16823, 16837,
16865, 17032, 17042, 18136, 18163,
18182, 18225, 18261, 18316, 18355,
19914, 19936, 21486, 21528, 21602,
21800, 24746, 36962, 36972, 36987,
36993, 37067, 37075, 37244, 37259
\__kernel_tl_to_str:w
..... 355, 691, 692, 1455, 1457, 12462,
12559, 12690, 13491, 13559, 16267
keys commands:
\l_keys_choice_int . 224, 227, 229,
230, 20830, 21017, 21020, 21025, 21026
\l_keys_choice_tl
.... 224, 227, 229, 230, 20830, 21024
\keys_define:nn 223, 9961, 20868, 20868
\keys_if_choice_exist:nnn .. 21909
\keys_if_choice_exist:nnnTF ..
..... 234, 21909

```

```

\keys_if_choice_exist_p:nnn . . .
    ..... 234, 21909
\keys_if_exist:nn . . . 21901, 21908
\keys_if_exist:nnTF . . .
    ..... 234, 959, 21901, 21926
\keys_if_exist_p:nn . . . 234, 21901
\l_keys_key_str 231, 20833, 20974,
    21104, 21611, 21612, 21711, 21715,
    21740, 21743, 21744, 21780, 21837
\l_keys_key_tl . . . 20833, 21612
\keys_log:nn . . . 234, 21917, 21919
\l_keys_path_str 231, 20838, 20896,
    20915, 20922, 20930, 20931, 20932,
    20949, 20967, 20969, 20971, 20986,
    20989, 20993, 21001, 21003, 21004,
    21007, 21022, 21038, 21051, 21056,
    21066, 21070, 21077, 21082, 21086,
    21087, 21097, 21099, 21101, 21115,
    21124, 21129, 21131, 21146, 21157,
    21163, 21167, 21183, 21192, 21234,
    21245, 21286, 21602, 21610, 21648,
    21651, 21690, 21694, 21699, 21708,
    21722, 21724, 21725, 21729, 21737,
    21760, 21790, 21813, 21825, 21834
\l_keys_path_tl . 20838, 20932, 20993
\keys_compile:nnN 233, 21582, 21582
\keys_set:nn . . .
    ..... 223, 226, 231–233, 21117,
    21124, 21441, 21441, 21461, 21586
\keys_set_filter:nnn . . .
    ..... 233, 21511, 21531, 21533
\keys_set_filter:nnnN . . .
    ..... 233, 21511, 21511, 21517
\keys_set_filter:nnnnN . . .
    ..... 233, 21511, 21518, 21523
\keys_set_groups:nnn . . .
    ..... 233, 21511, 21553, 21573
\keys_set_known:nn . . .
    ..... 232, 21470, 21489, 21491
\keys_set_known:nnN . . .
    ..... 232, 950, 21470, 21470, 21475
\keys_set_known:nnnN . . .
    ..... 232, 21470, 21476, 21481
\keys_show:nn . . . 234, 21917, 21917
\l_keys_usage_load_prop . . .
    ..... 231, 20854, 21202, 21209, 21216
\l_keys_usage_preamble_prop . . .
    ..... 231, 20854, 21204, 21211, 21218
\l_keys_value_tl 231, 20848, 21183,
    21693, 21697, 21703, 21714, 21725,
    21744, 21757, 21782, 21792, 21820
keys internal commands:
    \__keys_bool_set:Nn . . .
        ..... 20957, 20957, 20959,
    \__keys_bool_set:Nnn . . .
        ..... 20957, 20958, 20961, 20963
    \__keys_bool_set_inverse:Nn . . .
        ..... 20957, 20960,
        20962, 21268, 21270, 21272, 21274
    \__keys_check_forbidden: 21150, 21178
    \__keys_check_groups: . 21652, 21660
    \__keys_check_required: 21150, 21187
    \c__keys_check_root_str . 20823,
        21157, 21163, 21167, 21724, 21743
    \__keys_choice_find:n . . .
        ..... 20980, 21831, 21831, 21847
    \__keys_choice_find:nn . . .
        ..... 21831, 21834, 21836, 21840
    \__keys_choice_make: . . . 20966,
        20979, 20979, 21011, 21096, 21276
    \__keys_choice_make:N . . .
        ..... 20979, 20980, 20982, 20983
    \__keys_choice_make_aux:N . . .
        ..... 20979, 20995, 20997, 20999
    \__keys_choices_make:nn . . . 21010,
        21010, 21278, 21280, 21282, 21284
    \__keys_choices_make:Nnn . . .
        ..... 21010, 21011, 21013, 21014
    \__keys_cmd_set:nn . 20967, 20969,
        21021, 21031, 21031, 21033, 21097,
        21099, 21101, 21131, 21245, 21286
    \__keys_cmd_set_direct:nn . . .
        ..... 20971, 21003, 21004,
        21031, 21032, 21034, 21115, 21123
    \c__keys_code_root_str . . .
        ..... 956, 20823, 21035,
        21038, 21086, 21722, 21740, 21756,
        21770, 21842, 21904, 21913, 21934
    \__keys_cs_set:NNpn . . . 21036,
        21036, 21045, 21296, 21298, 21300,
        21302, 21304, 21306, 21308, 21310
    \__keys_default_inherit: . . .
        ..... 21686, 21700, 21705
    \c__keys_default_root_str . . .
        ..... 20823, 21051,
        21056, 21690, 21694, 21711, 21715
    \__keys_default_set:n . . .
        ..... 20976, 21046, 21046,
        21106, 21312, 21314, 21316, 21318
    \__keys_define:n . 20873, 20877, 20877
    \__keys_define:nn 20873, 20877, 20882
    \__keys_define:nnn . . .
        ..... 20868, 20869, 20870, 20876
    \__keys_define_aux:nn . . .
        ..... 20877, 20880, 20885, 20887
    \__keys_define_code:n . . .
        ..... 20891, 20940, 20940

```

\\_\_keys\_define\_code:w .....  
     ..... [20940](#), [20944](#), [20955](#)  
 \\_\_keys\_execute: ..... [21616](#),  
     [21656](#), [21678](#), [21682](#), [21720](#), [21720](#)  
 \\_\_keys\_execute:nn .....  
     [21087](#), [21720](#), [21725](#), [21744](#), [21757](#),  
     [21764](#), [21765](#), [21766](#), [21843](#), [21844](#)  
 \\_\_keys\_execute\_inherit: .....  
     ..... [21083](#), [21720](#), [21730](#), [21734](#)  
 \\_\_keys\_execute\_unknown: .....  
     .... [955](#), [21720](#), [21731](#), [21748](#), [21750](#)  
 \l\_\_keys\_filtered\_bool ... [20844](#),  
     [21446](#), [21453](#), [21454](#), [21497](#), [21503](#),  
     [21504](#), [21539](#), [21545](#), [21546](#), [21558](#),  
     [21565](#), [21566](#), [21655](#), [21676](#), [21681](#)  
 \\_\_keys\_find\_key\_module:wNN .....  
     ..... [21129](#), [21590](#), [21610](#), [21620](#)  
 \\_\_keys\_find\_key\_module\_auxi:Nw ..  
     [21590](#), [21622](#), [21625](#), [21633](#), [21638](#)  
 \\_\_keys\_find\_key\_module\_auxii:Nw ..  
     [21590](#), [21622](#), [21629](#), [21630](#)  
 \\_\_keys\_find\_key\_module\_auxiii:Nn ..  
     ..... [21590](#)  
 \\_\_keys\_find\_key\_module\_auxiii:Nw ..  
     ..... [21633](#), [21635](#)  
 \\_\_keys\_find\_key\_module\_auxiv:Nw ..  
     [21590](#), [21623](#), [21640](#), [21642](#)  
 \l\_\_keys\_groups\_clist ... [20832](#),  
     [21063](#), [21064](#), [21071](#), [21650](#), [21665](#)  
 \c\_\_keys\_groups\_root\_str .....  
     ... [20823](#), [21066](#), [21070](#), [21648](#), [21651](#)  
 \\_\_keys\_groups\_set:n .....  
     ..... [21061](#), [21061](#), [21336](#)  
 \\_\_keys\_inherit:n [21074](#), [21074](#), [21338](#)  
 \c\_\_keys\_inherit\_root\_str .....  
     ..... [20823](#), [21077](#),  
     [21082](#), [21699](#), [21708](#), [21729](#), [21737](#)  
 \l\_\_keys\_inherit\_str ..... [20840](#),  
     [21085](#), [21609](#), [21742](#), [21833](#), [21837](#)  
 \\_\_keys\_initialise:n ..... [21079](#),  
     [21079](#), [21340](#), [21342](#), [21344](#), [21346](#)  
 \\_\_keys\_legacy\_if\_inverse:nn . [21090](#)  
 \\_\_keys\_legacy\_if\_inverse:nnnn [21090](#)  
 \\_\_keys\_legacy\_if\_set:nn .....  
     ..... [21090](#), [21090](#), [21356](#), [21358](#)  
 \\_\_keys\_legacy\_if\_set:nnnn .....  
     ..... [21091](#), [21093](#), [21094](#)  
 \\_\_keys\_legacy\_if\_set\_inverse:nn ..  
     ..... [21092](#), [21360](#), [21362](#)  
 \\_\_keys\_meta\_make:n .....  
     ..... [21113](#), [21113](#), [21364](#)  
 \\_\_keys\_meta\_make:nn .....  
     ..... [21113](#), [21121](#), [21366](#)

\l\_\_keys\_module\_str .....  
     ..... [20835](#), [20869](#), [20872](#), [20874](#),  
     [20916](#), [21118](#), [21231](#), [21238](#), [21463](#),  
     [21466](#), [21468](#), [21593](#), [21598](#), [21608](#),  
     [21611](#), [21617](#), [21756](#), [21757](#), [21760](#)  
 \\_\_keys\_multichoice\_find:n .....  
     ..... [20982](#), [21831](#), [21846](#)  
 \\_\_keys\_multichoice\_make: .....  
     ..... [20979](#), [20981](#), [21013](#), [21368](#)  
 \\_\_keys\_multichoices\_make:nn ...  
     ..... [21010](#),  
     [21012](#), [21370](#), [21372](#), [21374](#), [21376](#)  
 \l\_\_keys\_no\_value\_bool .....  
     ..... [20836](#), [20879](#),  
     [20884](#), [20942](#), [21180](#), [21189](#), [21592](#),  
     [21597](#), [21688](#), [21781](#), [21791](#), [21819](#)  
 \l\_\_keys\_only\_known\_bool .....  
     .... [20837](#), [21445](#), [21451](#), [21452](#),  
     [21496](#), [21501](#), [21502](#), [21538](#), [21543](#),  
     [21544](#), [21557](#), [21563](#), [21564](#), [21752](#)  
 \\_\_keys\_parent:n .....  
     ... [20986](#), [20989](#), [20993](#), [21082](#), [21699](#),  
     [21708](#), [21729](#), [21737](#), [21848](#), [21848](#)  
 \\_\_keys\_parent\_auxi:w .....  
     ... [21848](#), [21850](#), [21853](#), [21859](#), [21863](#)  
 \\_\_keys\_parent\_auxii:w .....  
     ..... [21848](#), [21850](#), [21857](#)  
 \\_\_keys\_parent\_auxiii:n .....  
     ..... [21848](#), [21859](#), [21861](#)  
 \\_\_keys\_parent\_auxiv:w .....  
     ..... [21848](#), [21851](#), [21865](#)  
 \\_\_keys\_compile:n .....  
     ..... [20861](#), [20861](#), [21032](#),  
     [21040](#), [21947](#), [21949](#), [21955](#), [21956](#)  
 \l\_\_keys\_compile\_bool .....  
     ..... [20852](#), [20863](#), [21584](#), [21587](#)  
 \l\_\_keys\_compile\_t1 .....  
     ..... [20852](#), [20864](#), [21585](#), [21588](#)  
 \\_\_keys\_prop\_put:Nn [21126](#), [21126](#),  
     [21139](#), [21386](#), [21388](#), [21390](#), [21392](#)  
 \\_\_keys\_property\_find:n .....  
     ..... [20889](#), [20900](#), [20900](#)  
 \\_\_keys\_property\_find\_auxi:w ...  
     ... [20900](#), [20903](#), [20907](#), [20917](#), [20923](#)  
 \\_\_keys\_property\_find\_auxii:w ...  
     ..... [20900](#), [20904](#), [20911](#), [20912](#)  
 \\_\_keys\_property\_find\_auxiii:w ...  
     ..... [20900](#), [20917](#), [20920](#), [20926](#)  
 \\_\_keys\_property\_find\_auxiv:w ...  
     ..... [20900](#), [20918](#), [20925](#), [20927](#)  
 \\_\_keys\_property\_find\_err:w ...  
     ..... [20905](#), [20913](#), [20934](#), [20935](#)  
 \l\_\_keys\_property\_str [20843](#), [20890](#),  
     [20893](#), [20896](#), [20902](#), [20903](#), [20929](#),

20937, 20945, 20946, 20949, 20952  
`\c__keys_props_root_str` .....  
 20829, 20890, 20946, 20952, 21259,  
 21261, 21263, 21265, 21267, 21269,  
 21271, 21273, 21275, 21277, 21279,  
 21281, 21283, 21285, 21287, 21289,  
 21291, 21293, 21295, 21297, 21299,  
 21301, 21303, 21305, 21307, 21309,  
 21311, 21313, 21315, 21317, 21319,  
 21321, 21323, 21325, 21327, 21329,  
 21331, 21333, 21335, 21337, 21339,  
 21341, 21343, 21345, 21347, 21349,  
 21351, 21353, 21355, 21357, 21359,  
 21361, 21363, 21365, 21367, 21369,  
 21371, 21373, 21375, 21377, 21379,  
 21381, 21383, 21385, 21387, 21389,  
 21391, 21393, 21395, 21397, 21399,  
 21401, 21403, 21405, 21407, 21409,  
 21411, 21413, 21415, 21417, 21419,  
 21421, 21423, 21425, 21427, 21429,  
 21431, 21433, 21435, 21437, 21439  
`\_keys_quark_if_no_value:N` .. 20860  
`\_keys_quark_if_no_value:NTF` ...  
 ..... 20860, 21776  
`\_keys_quark_if_no_value:p:N` .. 20860  
`\l__keys_relative_tl` .... 20841,  
 21448, 21457, 21458, 21499, 21507,  
 21508, 21541, 21549, 21550, 21560,  
 21569, 21570, 21776, 21786, 21800,  
 21801, 21805, 21806, 21814, 21826  
`\l__keys_selective_bool` .....  
 .... 20844, 21447, 21455, 21456,  
 21498, 21505, 21506, 21540, 21547,  
 21548, 21559, 21567, 21568, 21614  
`\l__keys_selective_seq` .....  
 .. 20846, 21575, 21578, 21580, 21663  
`\_keys_set:nn` .....  
 .. 21441, 21450, 21462, 21500, 21579  
`\_keys_set:nnn` .. 21441, 21463, 21464  
`\_keys_set_filter:nnnm` .....  
 ..... 21511, 21527, 21532, 21534  
`\_keys_set_filter:nnnnN` .....  
 ..... 21511, 21513, 21520, 21524  
`\_keys_set_keyval:n` .....  
 ..... 21467, 21590, 21590  
`\_keys_set_keyval:nn` .....  
 ..... 21467, 21590, 21595  
`\_keys_set_keyval:nnn` .....  
 .. 21590, 21593, 21598, 21600, 21619  
`\_keys_set_known:nnn` .....  
 ..... 21470, 21485, 21490, 21492  
`\_keys_set_known:nnnnN` .....  
 ..... 21470, 21472, 21478, 21482  
`\_keys_set_selective:` .....  
 ..... 21590, 21615, 21646  
`\_keys_set_selective:nnn` .....  
 ..... 21511, 21542, 21562, 21574  
`\_keys_set_selective:nnnn` .....  
 ..... 21511, 21575, 21576  
`\_keys_show:n` .. 21917, 21930, 21943  
`\_keys_show:Nnn` .....  
 ..... 21917, 21918, 21920, 21921  
`\_keys_show:Nw` .. 21917, 21962, 21966  
`\_keys_show:w` .. 21917, 21945, 21954  
`\_keys_store_unused:` ... 21657,  
 21677, 21683, 21720, 21753, 21774  
`\_keys_store_unused:w` .....  
 ..... 21804, 21825, 21830  
`\_keys_store_unused_aux:` .....  
 ..... 21720, 21795, 21798  
`\_keys_tmp:w` ..... 21869, 21881  
`\l__keys_tmp_bool` .....  
 ..... 20849, 21662, 21669, 21674  
`\l__keys_tmpta_tl` ... 20849, 21130,  
 21231, 21232, 21236, 21237, 21239  
`\l__keys_tmpb_tl` ..... 20849,  
 21130, 21135, 21233, 21236, 21237  
`\_keys_trim_spaces:n` ..... 934,  
 20872, 20902, 20931, 21022, 21466,  
 21606, 21801, 21842, 21843, 21868,  
 21871, 21904, 21913, 21924, 21935  
`\_keys_trim_spaces_auxi:w` .....  
 .. 21868, 21873, 21874, 21883, 21893  
`\_keys_trim_spaces_auxii:w` 21868,  
 21875, 21877, 21887, 21894, 21896  
`\_keys_trim_spaces_auxiii:w` ...  
 ..... 21868, 21878, 21891, 21897  
`\c__keys_type_root_str` .....  
 ..... 20823, 20986, 20989, 21001  
`\_keys_undefine:` .....  
 ..... 21076, 21140, 21140, 21434  
`\l__keys_unused_clist` .....  
 . 949, 20847, 21473, 21479, 21484,  
 21486, 21487, 21514, 21521, 21526,  
 21528, 21529, 21778, 21788, 21816  
`\_keys_usage:n` .. 21196, 21196, 21436  
`\_keys_usage:NN` .....  
 ..... 21196, 21202, 21204,  
 21209, 21211, 21216, 21218, 21229  
`\_keys_usage:w` .. 21196, 21234, 21241  
`\_keys_value_or_default:n` .....  
 ..... 21613, 21686, 21686  
`\_keys_value_requirement:nn` ...  
 ..... 21058,  
 21150, 21150, 21256, 21438, 21440  
`\_keys_variable_set:NnnN` .....  
 .. 21242, 21242, 21252, 21255,

21288, 21290, 21292, 21294, 21402,  
 21404, 21406, 21408, 21410, 21412,  
 21414, 21416, 21418, 21420, 21422,  
 21424, 21426, 21428, 21430, 21432  
 $\backslash\_\_keyval\_variable\_set\_required:NnnN$   
     ..... 21242, 21253, 21258,  
 21320, 21322, 21324, 21326, 21328,  
 21330, 21332, 21334, 21348, 21350,  
 21352, 21354, 21378, 21380, 21382,  
 21384, 21394, 21396, 21398, 21400  
 keyval commands:  
 $\backslash\keyval\_parse:NNn$  .. . . . .  
     236, 930, 20601, 20623, 20873, 21467  
 $\backslash\keyval\_parse:nnn$  .. 235, 236, 924,  
     929, 19775, 20601, 20603, 20615, 20623  
 keyval internal commands:  
 $\backslash\_\_keyval\_blank\_key\_error:w$  .. .  
     ..... 20742, 20751, 20766, 20768  
 $\backslash\_\_keyval\_blank\_true:w$  .. . . . .  
     ..... 20698, 20766, 20766  
 $\backslash\_\_keyval\_clean\_up\_active:w$  .. .  
     ..... 927, 20640,  
     20653, 20674, 20674, 20706, 20726  
 $\backslash\_\_keyval\_clean\_up\_other:w$  .. . . .  
     .... 928, 20679, 20684, 20695, 20695  
 $\backslash\_\_keyval\_end\_loop\_active:w$  .. .  
     ..... 20627, 20720, 20728  
 $\backslash\_\_keyval\_end\_loop\_other:w$  .. . . .  
     .... 928, 20637, 20720, 20720  
 $\backslash\_\_keyval\_if\_blank:w$  .. . . . .  
     .. 20698, 20742, 20751, 20763, 20764  
 $\backslash\_\_keyval\_if\_empty:w$  .. . . . .  
     ..... 20763, 20763, 20764  
 $\backslash\_\_keyval\_if\_recursion\_tail:w$  .. .  
     ..... 20626, 20636, 20763, 20765  
 $\backslash\_\_keyval\_key:nn$  .. . . . .  
     .... 928, 20700, 20737, 20749, 20766  
 $\backslash\_\_keyval\_loop\_active:nnw$  .. . . .  
     .. 20608, 20618, 20624, 20624, 20727  
 $\backslash\_\_keyval\_loop\_other:nnw$  .. . . .  
     ..... 925, 20628,  
     20634, 20634, 20710, 20718, 20730,  
     20747, 20756, 20767, 20768, 20773  
 $\backslash\_\_keyval\_misplaced\_equal\_after\_-$   
 $\backslash\_\_keyval\_misplaced\_equal\_in\_-$   
 $\backslash\_\_keyval\_misplaced\_equal\_in\_split\_error:w$  .. . . .  
     ..... 20658, 20663, 20667,  
     20671, 20687, 20692, 20702, 20712  
 $\backslash\_\_keyval\_pair:nnnn$  .. . . . .  
     .... 927, 20673, 20694, 20737, 20740  
 $\backslash\_\_keyval\_split\_active:w$  ... 925,  
     926, 20630, 20632, 20638, 20657, 20722  
 $\backslash\_\_keyval\_split\_active\_auxi:w$  . . . .  
     .. 20639, 20644, 20644, 20675, 20725  
 $\backslash\_\_keyval\_split\_active\_auxii:w$  . . . . .  
     .... 926, 20644, 20648, 20650, 20704  
 $\backslash\_\_keyval\_split\_active\_auxiii:w$  . . . . .  
     .... 926, 20644, 20654, 20655  
 $\backslash\_\_keyval\_split\_active\_auxiv:w$  . . . . .  
     .... 926, 20644, 20659, 20662  
 $\backslash\_\_keyval\_split\_active\_auxv:w$  . . . . .  
     .... 20644, 20668, 20670  
 $\backslash\_\_keyval\_split\_other:w$  .. 20630,  
     20630, 20646, 20666, 20677, 20686  
 $\backslash\_\_keyval\_split\_other\_auxi:w$  . . . . .  
     .... 927, 20678, 20681, 20681, 20696  
 $\backslash\_\_keyval\_split\_other\_auxii:w$  . . . . .  
     .... 20681, 20682, 20683  
 $\backslash\_\_keyval\_split\_other\_auxiii:w$  . . . . .  
     .... 927, 20681, 20688, 20691  
 $\backslash\_\_keyval\_tmp:w$  929, 20599, 20735,  
     20738, 20760, 20761, 20782, 20820  
 $\backslash\_\_keyval\_trim:nN$  .. . . . .  
     ..... 20654, 20673, 20682,  
     20694, 20700, 20766, 20781, 20784  
 $\backslash\_\_keyval\_trim\_auxi:w$  .. . . . .  
     .. 20781, 20786, 20795, 20798, 20803  
 $\backslash\_\_keyval\_trim\_auxii:w$  .. . . . .  
     .... 20781, 20790, 20803  
 $\backslash\_\_keyval\_trim\_auxiii:w$  .. 20781,  
     20791, 20805, 20808, 20812, 20816  
 $\backslash\_\_keyval\_trim\_auxiv:w$  .. . . . .  
     .... 20781, 20793, 20814  
 $\backslash\kuten$  .. . . . . 1142, 1180

**L**

$\backslash L$  ..... 30149, 31874, 32585  
 $\backslash l$  ..... 30149, 31874, 32597  
 $\backslash label$  ..... 30159, 30169, 32533  
 $\backslash language$  ..... 327  
 $\backslash LARGE$  ..... 32513  
 $\backslash Large$  ..... 32514  
 $\backslash large$  ..... 32517  
 $\backslash lastallocatedtoks$  ..... 3402  
 $\backslash lastbox$  ..... 328  
 $\backslash lastkern$  ..... 329  
 $\backslash lastlinefit$  ..... 545  
 $\backslash lastnamedcs$  ..... 833  
 $\backslash lastnodechar$  ..... 1143  
 $\backslash lastnodefont$  ..... 1144  
 $\backslash lastnodesubtype$  ..... 1145  
 $\backslash lastnodetype$  ..... 546  
 $\backslash lastpenalty$  ..... 330  
 $\backslash lastsavedboxresourceindex$  ..... 920  
 $\backslash lastsavedimageresourceindex$  ..... 922  
 $\backslash lastsavedimageresourcepages$  ..... 924

\lastskip . . . . .	331	\looseness . . . . .	344
\lastxpos . . . . .	926	\lower . . . . .	345
\lastypos . . . . .	927	\lowercase . . . . .	88, 346
\latelua . . . . .	834	\lpcode . . . . .	689
\lateluafunction . . . . .	835	\ltx.utils . . . . .	99, 11776
\lccode . . . . .	85, 86, 332	\ltx.utils.filiedump . . . . .	99, 11849
\leaders . . . . .	333	\ltx.utils.filemd5sum . . . . .	99, 11870
\left . . . . .	334	\ltx.utils.filemoddate . . . . .	99, 11879
\leftghost . . . . .	836	\ltx.utils.filesize . . . . .	100, 11932
\lefthyphenmin . . . . .	335	lua commands:	
\leftmarginkern . . . . .	687	\lua_escape:n . . . . .	99, 11712, 11714, 11719, 11720, 11735, 37529
\leftskip . . . . .	336	\lua_escape_x:n . . . . .	37528
legacy commands:		\lua_load_module:n . . . . .	99, 11721, 11723, 11746
\legacy_if:n . . . . .	12050	\lua_now:n . . . . .	. 98, 99, 8819, 8828, 11713, 11714, 11714, 11715, 11736, 29127, 37531
\legacy_if_nTF . . . . .	101, 12050	\lua_now_x:n . . . . .	37530
\legacy_if_gset:n . . . . .	226, 21355	\lua_shipout:n . . . . .	98, 11714, 11717, 11746
\legacy_if_gset:nn . . . . .	101, 12067, 12072	\lua_shipout_e:n . . . . .	98, 11714, 11716, 11718, 11746, 37533
\legacy_if_gset_false:n . . . . .	101, 12059, 12065, 12074	\lua_shipout_x:n . . . . .	37532
\legacy_if_gset_inverse:n . . . . .	226, 21355	lua internal commands:	
\legacy_if_gset_true:n . . . . .	101, 12059, 12063, 12074	\l_lua_err_msg_str . . . . .	11721, 11728
\legacy_if_p:n . . . . .	101, 12050	\_\_lua_escape:n . . . . .	11709, 11709, 11719
\legacy_if_set:n . . . . .	226, 21355	\_\_lua_load_module_p:n . . . . .	11725, 11992
\legacy_if_set:nn . . . . .	101, 12067, 12067	\_\_lua_now:n . . . . .	11709, 11710, 11714
\legacy_if_set_false:n . . . . .	101, 12059, 12061, 12069	\_\_lua_shipout:n . . . . .	11709, 11711, 11716
\legacy_if_set_inverse:n . . . . .	226, 21355	\luabytecode . . . . .	842
\legacy_if_set_true:n . . . . .	101, 12059, 12059, 12069	\luabytocodecall . . . . .	843
\leqno . . . . .	337	\luacopyinputnodes . . . . .	844
\let . . . . .	22, 179, 180, 338	\luadef . . . . .	845
\letcharcode . . . . .	837	\luadef . . . . .	11942
\letterspacefont . . . . .	688	\luaescapestring . . . . .	848
\limits . . . . .	1357, 339	\luafunction . . . . .	849
\LineBreak . . . . .	58,	\luafunctioncall . . . . .	850
59, 60, 61, 62, 63, 64, 65, 66, 67, 81, 83		luatex commands:	
\linedir . . . . .	838	\luatex_if_engine:TF . . . . .	37536, 37538, 37540
\linedirection . . . . .	839	\luatex_if_engine_p: . . . . .	37534
\linepenalty . . . . .	340	\luatexalignmark . . . . .	1250
\lineskip . . . . .	341	\luatexaligntab . . . . .	1251
\lineskiplimit . . . . .	342	\luatexattribute . . . . .	1252
\linewidth . . . . .	33733	\luatexattributedef . . . . .	1253
\ln . . . . .	27050, 27053	\luatexbanner . . . . .	851
\ln . . . . .	253	\luatexbodydir . . . . .	1289
\localbrokenpenalty . . . . .	840	\luatexboxdir . . . . .	1290
\localinterlinepenalty . . . . .	841	\luatexcatcodetable . . . . .	1254
\localleftbox . . . . .	846	\luatexclearmarks . . . . .	1255
\localrightbox . . . . .	847	\luatexcrampeddisplaystyle . . . . .	1256
\loccount . . . . .	10188, 10419	\luatexcrampedscriptstyle . . . . .	1258
\loctoks . . . . .	3374, 3375, 3401	\luatexcrampedscriptstyle . . . . .	1259
\logb . . . . .	254	\luatexcrampedtextstyle . . . . .	1260
\long . . . . .	182, 343, 19293, 19297		
\LongText . . . . .	55, 97		

\luatexfontid .....	1261	\mathcode .....	355
\luatexformatname .....	1262	\mathcolor .....	1357
\luatexleaders .....	1263	\mathdefaultsmode .....	854
\luatexinitcatcodetable .....	1264	\mathdelimitersmode .....	855
\luatexlatelua .....	1265	\mathdir .....	856
\luatexleftghost .....	1291	\mathdirection .....	857
\luatexlocalbrokenpenalty .....	1292	\mathdisplayskipmode .....	858
\luatexlocalinterlinepenalty .....	1294	\matheqdirmode .....	859
\luatexlocalleftbox .....	1295	\matheqnogapstep .....	860
\luatexlocalrightbox .....	1296	\mathinner .....	356
\luatexluaescapestring .....	1266	\mathnolimitsmode .....	861
\luatexluafunction .....	1267	\mathop .....	357
\luatexmathdir .....	1297	\mathopen .....	358
\luatexmathstyle .....	1268	\mathoption .....	862
\luatexnokerns .....	1269	\mathord .....	359
\luatexnoligs .....	1270	\mathpenaltiesmode .....	863
\luatexoutputbox .....	1271	\mathpunct .....	360
\luatexpagebottomoffset .....	1298	\mathrel .....	361
\luatexpagedir .....	1299	\mathrulesfam .....	864
\luatexpageheight .....	1300	\mathscriptboxmode .....	866
\luatexpageleftoffset .....	1272	\mathscriptcharmode .....	867
\luatexpagerightoffset .....	1301	\mathscriptsmode .....	865
\luatexpagetopoffset .....	1273	\mathstyle .....	868
\luatexpagewidth .....	1302	\mathsurround .....	362
\luatexpardir .....	1303	\mathsurroundmode .....	869
\luatexpostexhyphenchar .....	1274	\mathsurroundskip .....	870
\luatexposthyphenchar .....	1275	max .....	254
\luatexpreeexhyphenchar .....	1276	\maxdeadcycles .....	363
\luatexprehyphenchar .....	1277	\maxdepth .....	364
\luatexrevision .....	852	md5.HEX .....	11862
\luatexrightghost .....	1304	\mdfivesum .....	778
\luatexsavecatcodetable .....	1278	\mdseries .....	32506
\luatextscantexttokens .....	1279	\meaning .....	365
\luatexsuppressfontnotfounderror .....	1249, 1288	\medmuskip .....	366
\luatexsuppressifcsnameerror .....	1281	\message .....	367
\luatexsuppresslongerror .....	1282	\MessageBreak .....	81
meta commands:			
.meta:n .....	226, 21363		
.meta:nn .....	226, 21365		
.middle .....	548		
min .....	254		
\c_minus_one .....	37436		
\mkern .....	368		
mm .....	258		
mode commands:			
\mode_if_horizontal: .....	8767		
\mode_if_horizontal:TF .....	70, 8767		
\mode_if_horizontal_p: .....	70, 8767		
\mode_if_inner: .....	8769		
\mode_if_inner:TF .....	70, 8769		
\mode_if_inner_p: .....	70, 8769		
\mode_if_math: .....	8771		
\mode_if_math:TF .....	70, 8771		
\mode_if_math_p: .....	70, 8771		

## M

\mag .....	347
\MARJ .....	31518
\mark .....	348
\marks .....	547
\mathaccent .....	349
\mathbin .....	350
\mathchar .....	351, 19292
\mathchardef .....	352
\mathchoice .....	353
\mathclose .....	354

\mode\_if\_vertical: ..... 8765  
 \mode\_if\_vertical:TF ..... 70, 8765  
 \mode\_if\_vertical\_p: ..... 70, 8765  
 \mode\_leave\_vertical: ..... 28, 2279, 2279, 34561, 34622  
 \month ..... 369, 1328, 9098  
 \moveleft ..... 370  
 \moveright ..... 371  
 msg commands:  
   \msg\_critical:nn ..... 81, 97, 9533  
   \msg\_critical:nnn ..... 81, 9533  
   \msg\_critical:nnnn ..... 81, 9533  
   \msg\_critical:nnnnn ..... 81, 9533  
   \msg\_critical:nnnnnn ..... 81, 9533  
   \msg\_critical\_text:n ..... 79, 9434, 9439, 9536  
 \msg\_error:nn ..... 81, 1891, 1906, 5071, 5104, 5152,  
   5155, 5628, 5899, 7332, 7416, 8878,  
   9541, 9543, 29074, 29119, 34007, 37151  
 \msg\_error:nnn ..... 81, 1568,  
   1573, 1649, 1704, 1755, 1760, 1891,  
   1904, 2087, 2175, 2949, 3223, 3699,  
   5110, 5333, 5727, 5740, 5779, 5812,  
   5926, 7105, 7112, 7324, 7430, 8971,  
   9126, 9541, 9542, 9639, 9781, 11295,  
   11397, 11750, 12376, 13330, 14159,  
   14218, 16274, 16298, 16302, 16446,  
   16702, 16733, 19771, 20938, 20973,  
   21103, 21173, 21191, 22046, 22275,  
   22519, 22542, 22940, 28753, 29092,  
   29150, 29156, 33039, 33629, 35008,  
   35065, 35387, 35628, 35664, 35774,  
   35783, 35817, 35830, 35845, 35850,  
   35920, 35939, 35952, 35969, 35979,  
   35987, 36339, 36352, 36375, 37157  
 \msg\_error:nnnn ..... 81,  
   1640, 1680, 1774, 1891, 1891, 1905,  
   1907, 1918, 2044, 3034, 3243, 3266,  
   3562, 3569, 5310, 5375, 5590, 7336,  
   7352, 8900, 8919, 8935, 9276, 9541,  
   9541, 9665, 9783, 10577, 11604,  
   11722, 11727, 14250, 16317, 18727,  
   20895, 20948, 20992, 21006, 21182,  
   21223, 21759, 21812, 22936, 33849  
 \msg\_error:nnnnn ..... 81, 2080, 3715, 7729, 7920, 8544,  
   9541, 9785, 10588, 13200, 13239,  
   22131, 22316, 28800, 37021, 37406  
 \msg\_error:nnnnnn 81, 83, 3049, 3063,  
   7527, 7550, 7624, 9541, 13233, 22342  
 \msg\_error\_text:n ..... 79,  
   9434, 9437, 9442, 9444, 9547, 10144  
 \msg\_expandable\_error:nn 83, 2620,  
   4801, 8752, 10132, 10152, 10931,  
   16474, 18884, 18890, 18894, 18920,  
   19683, 20708, 20716, 20772, 23468  
 \msg\_expandable\_error:nnn ..... 83, 2349,  
   2696, 2756, 2780, 4896, 5917, 9110,  
   9157, 9787, 9789, 10132, 10150,  
   10157, 10963, 11110, 11371, 11741,  
   12696, 17093, 17378, 17559, 18591,  
   20325, 23475, 23490, 23495, 23561,  
   23618, 23657, 23663, 23999, 24004,  
   24013, 24020, 24111, 24125, 24323,  
   24374, 25054, 28450, 28457, 28463  
 \msg\_expandable\_error:nnnn ..... 83, 4826, 7235, 9791, 10132,  
   10148, 10156, 10583, 10940, 22471,  
   24508, 24529, 25215, 28629, 28719  
 \msg\_expandable\_error:nnnnn . 83,  
   10132, 10146, 10155, 10600, 22179,  
   22367, 23051, 28496, 28870, 37403  
 \msg\_expandable\_error:nnnnnn ...  
   83, 10132, 10133,  
   10147, 10149, 10151, 10153, 10154  
 \msg\_fatal:nn ..... 80, 9520  
 \msg\_fatal:nnn ..... 80, 9520  
 \msg\_fatal:nnnn ..... 80, 9520  
 \msg\_fatal:nnnnn ..... 80, 9520  
 \msg\_fatal:nnnnnn ..... 80, 9520  
 \msg\_fatal\_text:n 79, 9434, 9434, 9523  
 \msg\_gset:nnn ..... 78, 9280, 9303  
 \msg\_gset:nnnn ..... 78, 9280, 9283, 9296, 9304  
 \msg\_if\_exist:nn ..... 9267  
 \msg\_if\_exist:nnTF 78, 9267, 9274, 9649  
 \msg\_if\_exist\_p:nn ..... 78, 9267  
 \msg\_info:nn ..... 82, 9551  
 \msg\_info:nnn ..... 82, 9551  
 \msg\_info:nnnn ..... 82, 9551, 9775  
 \msg\_info:nnnnn ..... 82, 9551  
 \msg\_info:nnnnnn ..... 82, 9551  
 \msg\_info\_text:n ..... 79, 9434, 9448, 9580, 9585  
 \msg\_interrupt:nnn ..... 37542  
 \msg\_line\_context: . 78, 600, 1908,  
   1908, 9337, 9338, 20776, 20778, 29259  
 \msg\_line\_number: 79, 9337, 9337, 9342  
 \msg\_log:n ..... 37544  
 \msg\_log:nn ..... 82, 9588  
 \msg\_log:nnn ..... 82, 9588  
 \msg\_log:nnnn ..... 82, 9588  
 \msg\_log:nnnnn ..... 82, 9588  
 \msg\_log:nnnnnn . 82, 4105, 4119,  
   7455, 7465, 9588, 10273, 10496,

11504, 17146, 18712, 18733, 20056,  
 21920, 22444, 34774, 34805, 36398  
`\msg_log_eval:Nn` ..... 308, 8529, 17978, 20422,  
 20515, 20583, 24804, 36852, 36854  
`\msg_module_name:n` ..... 78,  
 80, 9347, 9453, 9470, 9470, 9478, 9554  
`\g_msg_module_name_prop` ..... 78,  
 3745, 8430, 9461, 9472, 9473, 10158,  
 10166, 10169, 10171, 11772, 14873,  
 20779, 22013, 22919, 29293, 36558  
`\msg_module_type:n` ..... 78,  
 79, 9452, 9464, 9464  
`\g_msg_module_type_prop` ..... 78,  
 3746, 8431, 9461, 9466, 9467, 10159,  
 10167, 10170, 10172, 11773, 14874,  
 20780, 22014, 22920, 29294, 36559  
`\msg_new:nnn` ..... 78,  
 4462, 8144, 8146, 8151, 8412, 8424,  
 9280, 9285, 9773, 9902, 9935, 9946,  
 9948, 9950, 9952, 10054, 10056,  
 10058, 10060, 10062, 10064, 10066,  
 10068, 10072, 10075, 10082, 10084,  
 10091, 10098, 11651, 14490, 14492,  
 14501, 20775, 20777, 22006, 22019,  
 23079, 23081, 23083, 23085, 23087,  
 23089, 23091, 24706, 24708, 24710,  
 24712, 24714, 24716, 24718, 24720,  
 24722, 24724, 24726, 24728, 24730,  
 24732, 24736, 25107, 25109, 25111,  
 28446, 34831, 36513, 36560, 37411  
`\msg_new:nnnn` ..... 78, 599, 3704, 3721,  
 3728, 3737, 8157, 8164, 8170, 8180,  
 8186, 8210, 8217, 8225, 8233, 8240,  
 8247, 8253, 8260, 8266, 8274, 8280,  
 8286, 8296, 8303, 8312, 8315, 8323,  
 8329, 8335, 8342, 8349, 8359, 8370,  
 8380, 8390, 8399, 8405, 8414, 8417,  
 9280, 9280, 9286, 9771, 9792, 9800,  
 9808, 9815, 9826, 9834, 9843, 9850,  
 9859, 9863, 9873, 9882, 9889, 9895,  
 9904, 9911, 9919, 9927, 9954, 9957,  
 9966, 9972, 9979, 9986, 9998, 10005,  
 10013, 10020, 10036, 10045, 10109,  
 10115, 11115, 11612, 11645, 11657,  
 11663, 11670, 11675, 11755, 11762,  
 14361, 14494, 14509, 14523, 14529,  
 14576, 14621, 14711, 14826, 15008,  
 15015, 15187, 21970, 21973, 21976,  
 21982, 21988, 21994, 22000, 22911,  
 23053, 23068, 29252, 29258, 29264,  
 29270, 29277, 34815, 34822, 34825,  
 36419, 36429, 36435, 36442, 36448,  
 36457, 36463, 36471, 36480, 36486,  
 36495, 36504, 36519, 36525, 36534,  
 36540, 36546, 36552, 37026, 37162  
`\msg_none:nn` ..... 82, 9600  
`\msg_none:nnn` ..... 82, 9600  
`\msg_none:nnnn` ..... 82, 9600  
`\msg_none:nnnnn` ..... 82, 9600  
`\msg_note:nn` ..... 82, 9551  
`\msg_note:nnn` ..... 82, 9551  
`\msg_note:nnnn` ..... 82, 9551  
`\msg_note:nnnnn` ..... 82, 9551  
`\msg_note:nnnnnn` ..... 82, 9551  
`\msg_redirect_class:nn` 84, 9722, 9722  
`\msg_redirect_module:nnn` ..... 84, 9722, 9724  
`\msg_redirect_name:nnn` 84, 9713, 9713  
`\msg_see_documentation_text:n` ...  
 ..... 80, 9470, 9476  
`\msg_set:nnn` ..... 78, 9280, 9294  
`\msg_set:nnnn` ..... 78, 9280, 9287, 9295  
`\msg_show:nn` ..... 83, 9601  
`\msg_show:nnn` ..... 83, 9601  
`\msg_show:nnnn` ..... 83, 9601  
`\msg_show:nnnnn` ..... 83, 9601  
`\msg_show:nnnnnn` ..... 83, 308, 1340,  
 4103, 4117, 7454, 7464, 9601, 10272,  
 10495, 11503, 17144, 18710, 18732,  
 20054, 21918, 22442, 34771, 36396  
`\msg_show_eval:Nn` .....  
 ..... 308, 8527, 17974, 20418,  
 20511, 20579, 24802, 36852, 36852  
`\msg_show_item:n` ..... 83,  
 308, 17159, 18723, 18737, 36857, 36857  
`\msg_show_item:nn` .....  
 ..... 308, 904, 20069, 36857, 36861  
`\msg_show_item_unbraced:n` .....  
 ..... 308, 36857, 36859  
`\msg_show_item_unbraced:nn` .....  
 .. 308, 624, 10280, 10503, 21928,  
 34790, 36409, 36417, 36857, 36868  
`\msg_term:n` ..... 37546  
`\msg_term:nn` ..... 82, 9588  
`\msg_term:nnn` ..... 82, 9588  
`\msg_term:nnnn` ..... 82, 9588  
`\msg_term:nnnnn` ..... 82, 9588  
`\msg_term:nnnnnn` 82, 1340, 9588, 34802  
`\msg_warning:nn` ..... 81, 5618, 9551  
`\msg_warning:nnn` ..... 81, 5534, 5538,  
 5580, 5642, 5680, 5699, 9551, 9777  
`\msg_warning:nnnn` .....  
 ..... 81, 5240, 5389, 9551, 9779  
`\msg_warning:nnnnn` .. 81, 9551, 37379  
`\msg_warning:nnnnnn` .. 81, 9551, 9753

```

\msg_warning_text:n ..... 79, 9434, 9446, 9575
msg internal commands:
  \__msg_chk_free:nn ..... 9272, 9282
  \__msg_chk_if_free:nn ..... 9272
  \__msg_class_chk_exist:nTF .....
    .. 9636, 9636, 9651, 9718, 9728, 9733
  \l__msg_class_loop_seq . 611, 9645,
    9737, 9745, 9755, 9756, 9759, 9761
  \__msg_class_new:nn ..... 608,
    9481, 9482, 9520, 9533, 9544, 9573,
    9578, 9583, 9588, 9594, 9600, 9601
  \l__msg_class_tl ..... 609,
    611, 9641, 9658, 9671, 9692, 9696,
    9699, 9707, 9746, 9748, 9750, 9764
\c__msg_coding_error_text_tl ...
  ... 9305, 9795, 9803, 9829, 9837,
  9846, 9853, 9866, 9876, 9898, 9907,
  9914, 9922, 9930, 9960, 9969, 9975,
  9982, 9989, 10001, 10016, 10023, 10039
\c__msg_continue_text_tl . 9305, 9354
\c__msg_critical_text_tl . 9305, 9538
\l__msg_current_class_tl .....
  ..... 611, 9641, 9653,
  9691, 9696, 9699, 9707, 9736, 9750
\__msg_expandable_error:n ..... 619
\__msg_expandable_error:nn .....
  ..... 10121, 10124, 10135
\__msg_fatal_exit: .. 9520, 9526, 9528
\c__msg_fatal_text_tl ... 9305, 9525
\c__msg_help_text_tl ... 9305, 9364
\l__msg_hierarchy_seq .....
  ... 609, 610, 9644, 9674, 9684, 9689
\__msg_info_aux:NNnnnnnn .....
  ..... 9551, 9551, 9575, 9580, 9585
\l__msg_internal_tl .....
  .. 9259, 9390, 9396, 9531, 9625, 9631
\__msg_interrupt:n .. 9391, 9400, 9409
\__msg_interrupt:Nnnn ..... 9344
\__msg_interrupt:NnnnN .....
  ..... 9344, 9522, 9535, 9546
\__msg_interrupt_more_text:n ...
  ..... 601, 9373, 9375, 9398
\__msg_interrupt_text:n .....
  ..... 9373, 9389, 9393
\__msg_interrupt_wrap:nnn .....
  ..... 9352, 9362, 9373, 9373
\c__msg_more_text_prefix_tl ...
  ..... 9265, 9291, 9300, 9349, 9366
\l__msg_name_str ..... 9260,
  9347, 9380, 9384, 9554, 9562, 9566
\c__msg_no_info_text_tl .. 9305, 9356
\__msg_no_more_text:nnnn .....
  ..... 9344, 9350, 9372
\c__msg_on_line_text_t1 .. 9305, 9340
\__msg_redirect:nnn .....
  ..... 9722, 9723, 9725, 9726
\__msg_redirect_loop_chk:nnn .....
  ..... 9722, 9738, 9743, 9764, 9768
\__msg_redirect_loop_list:n .....
  ..... 9722, 9760, 9769
\l__msg_redirect_prop .....
  ..... 9643, 9671, 9716, 9719
\c__msg_return_text_t1 .....
  ..... 9305, 9798, 9806, 9813
\__msg_show:n .. 607, 9601, 9605, 9607
\__msg_show:nn .....
  ..... 9601, 9615, 9618, 9620, 9621
\__msg_show:w .. 9601, 9612, 9619
\__msg_show_dot:w ... 9601, 9612, 9617
\__msg_show_eval:nnN .....
  ..... 36852, 36853, 36855, 36856
\__msg_text:n .. 9434, 9452, 9453, 9456
\__msg_text:nn .....
  ..... 9434, 9445, 9447, 9449, 9450
\c__msg_text_prefix_t1 .....
  ... 619, 9265, 9269, 9289, 9298, 9353,
  9363, 9559, 9591, 9597, 9604, 10138
\l__msg_text_str ..... 9260,
  9346, 9378, 9383, 9553, 9558, 9565
\__msg_tmp:w ..... 10121, 10131
\c__msg_trouble_text_t1 ... 9305
\__msg_use:nnnnnn .. 9491, 9646, 9646
\__msg_use_code: .....
  ... 609, 9646, 9654, 9668, 9672, 9697, 9708
\__msg_use_hierarchy:nwwN .....
  ..... 9646, 9675, 9676, 9682
\__msg_use_none_delimit_by_s_-
  stop:w ... 9264, 9264, 9677, 10127
\__msg_use_redirect_module:n ...
  ... 610, 9646, 9679, 9687, 9700
\__msg_use_redirect_name:n .....
  ..... 9646, 9662, 9669
\mskip ..... 372
\muexpr ..... 549
multichoice commands:
  .multichoice: ..... 227, 21367
multicoices commands:
  .multicoices:nn ..... 227, 21367
\multiply ..... 373
\muskip ..... 374, 19301
muskip commands:
  \c_max_muskip ..... 221, 20584
  \muskip_add:Nn 219, 20560, 20560, 20564
  \muskip_const:Nn .. ...
    ... 219, 20528, 20528, 20533, 20584, 20585
  \muskip_eval:n .....
    ... 220, 20531, 20572, 20572, 20579, 20583

```

\muskip_gadd:Nn . . . . .	219, 20560, 20562, 20565	\newlinechar . . . . .	80, 376
.muskip_gset:N . . . . .	227, 21377	\newluabytecode . . . . .	36
\muskip_gset:Nn . . . . .	220, 20550, 20552, 20555	\next . . . . .	53, 94, 102, 105, 108, 116
\muskip_gset_eq:NN . . . . .	220, 20556, 20558, 20559	\NG . . . . .	30150, 31875, 32586
\muskip_gsub:Nn . . . . .	220, 20560, 20568, 20571	\ng . . . . .	30150, 31875, 32598
\muskip_gzero:N . . . . .	219, 20534, 20536, 20539, 20543	\c_nine . . . . .	37456
\muskip_gzero_new:N . . . . .	219, 20540, 20542, 20545	\noalign . . . . .	377
\muskip_if_exist:N . . . . .	20546, 20548	\noautospacing . . . . .	1146
\muskip_if_exist:NTF . . . . .	219, 20541, 20543, 20546	\noautoxspacing . . . . .	1147
\muskip_if_exist_p:N . . . . .	219, 20546	\noboundary . . . . .	378
\muskip_log:N . . . . .	221, 20580, 20580, 20581	\nobreakspace . . . . .	32541
\muskip_log:n . . . . .	221, 20580, 20582	\noexpand . . . . .	81, 96, 99, 119, 379
\muskip_new:N . . . . .	219, 20522, 20527, 20530, 20541, 20543, 20586, 20587, 20588, 20589	\nohrule . . . . .	871
.muskip_set:N . . . . .	227, 21377	\noindent . . . . .	380
\muskip_set:Nn . . . . .	220, 20550, 20550, 20554	\nokerns . . . . .	872
\muskip_set_eq:NN . . . . .	220, 20556, 20556, 20557	\noligs . . . . .	873
\muskip_show:N . . . . .	220, 20576, 20576, 20577	\nolimits . . . . .	381
\muskip_show:n . . . . .	221, 923, 20578, 20578	\nonscript . . . . .	382
\muskip_sub:Nn . . . . .	220, 20560, 20566, 20570	\nonstopmode . . . . .	383
\muskip_use:N . . . . .	220, 20573, 20574, 20574, 20575	\normaldeviate . . . . .	928
\muskip_zero:N . . . . .	219, 20534, 20534, 20538, 20541	\normalend . . . . .	1351, 1352
\muskip_zero_new:N . . . . .	219, 20540, 20540, 20544	\normaleveryjob . . . . .	1353
\g_tmpa_muskip . . . . .	221, 20586	\normalexpanded . . . . .	1362
\l_tmpa_muskip . . . . .	221, 20586	\normalfont . . . . .	32501
\g_tmpb_muskip . . . . .	221, 20586	\normalhoffset . . . . .	1365
\l_tmpb_muskip . . . . .	221, 20586	\normalinput . . . . .	1354
\c_zero_muskip . . . . .	221, 20535, 20537, 20584	\normalitaliccorrection . . . . .	1364, 1366
\muskipdef . . . . .	375	\normallanguage . . . . .	1355
\mutoglu . . . . .	550	\normalleft . . . . .	1372, 1373
my commands:		\normalmathop . . . . .	1356
\l_my_int . . . . .	157	\normalmiddle . . . . .	1374
		\normalmonth . . . . .	1357
<b>N</b>		\normalouter . . . . .	1358
\n . . . . .	9006, 9008, 9010, 11990	\normalover . . . . .	1359
nan . . . . .	257	\normalright . . . . .	1375
nc . . . . .	258	\normalshowtokens . . . . .	1368
nd . . . . .	258	\normalsize . . . . .	32518
\newbox . . . . .	820	\normalunexpanded . . . . .	1361
\newcatcodetable . . . . .	28925	\normalvcenter . . . . .	1360
\newcount . . . . .	820	\normalvoffset . . . . .	1367
\newdimen . . . . .	820	\nospaces . . . . .	874
		\notexpanded: <i>(token)</i> . . . . .	196
		\novrule . . . . .	875
		\nulldelimiterspace . . . . .	384
		\nullfont . . . . .	385
		\num . . . . .	240
		\number . . . . .	386
		\numexpr . . . . .	551
		<b>O</b>	
		\o . . . . .	30151, 31876, 32587, 32877
		\o . . . . .	30151, 31876, 32599, 32878
		\odelcode . . . . .	1183
		\odelimiter . . . . .	1184

\OE	30152, 31877, 32588	28104, 28110, 28187, 28188, 28195,
\oe	30152, 31877, 32600	28241, 28242, 28249, 28315, 28316,
\omathaccent	1185	28536, 28819, 28820, 28821, 28898,
\omathchar	1186	28899, 28900, 37341, 37342, 37343
\omathchardef	1187	\radical ..... 1189
\omathcode	1188	\orieveryjob ..... 1345, 1346
\omit	387	\oripdfoutput ..... 1348, 1349
\c_one	37440	\outer ..... 7, 820, 391
\c_one_hundred	37474	\output ..... 392
\openin	388	\outputbox ..... 876
\openout	389	\outputmode ..... 929
\or	390	\outputpenalty ..... 393
or commands:		\over ..... 394
\or: .	168, 722, 724, 871, 991, 1429, 1431, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2678, 2679, 2680, 2681, 2682, 3856, 3857, 4055, 4056, 4248, 4633, 4634, 4635, 4636, 4907, 4908, 4909, 4910, 4911, 6480, 6533, 7167, 7169, 7201, 7202, 7203, 7204, 7205, 7206, 7207, 7208, 7209, 7210, 7211, 7212, 7213, 7607, 7608, 10722, 10723, 10724, 10725, 10726, 10727, 10728, 13668, 13744, 14068, 14069, 14070, 14071, 14072, 15097, 15098, 17176, 17761, 17762, 17763, 17764, 17765, 17766, 17767, 17768, 17769, 17770, 17771, 17772, 17773, 17774, 17775, 17776, 17777, 17778, 17779, 17780, 17781, 17782, 17783, 17784, 17785, 17794, 17795, 17796, 17797, 17798, 17799, 17800, 17801, 17802, 17803, 17804, 17805, 17806, 17807, 17808, 17809, 17810, 17811, 17812, 17813, 17814, 17815, 17816, 17817, 17818, 18931, 18933, 18935, 18939, 18940, 18942, 18944, 18946, 18947, 18949, 18951, 18953, 18955, 22602, 22603, 22604, 22853, 22868, 22869, 23240, 23241, 23266, 24549, 24550, 24551, 24587, 25272, 25273, 25274, 25397, 25482, 25568, 25569, 25570, 25571, 25572, 25573, 25574, 25575, 25576, 25655, 25658, 25994, 25995, 26009, 26010, 26024, 26308, 26531, 26556, 26562, 26563, 26564, 26565, 26566, 26715, 26750, 26752, 26760, 26953, 27003, 27006, 27015, 27130, 27153, 27154, 27186, 27187, 27191, 27244, 27245, 27285, 27290, 27300, 27305, 27315, 27320, 27330, 27335, 27345, 27350, 27360, 27365, 27892, 27893, 27938, 28023, 28026, 28038, 28044, 28091, 28093, 28094,	28104, 28110, 28187, 28188, 28195, 28241, 28242, 28249, 28315, 28316, 28536, 28819, 28820, 28821, 28898, 28899, 28900, 37341, 37342, 37343
P		
\PackageError	88, 96	
\pagebottomoffset	877	
\pagedepth	398	
\pagedir	878	
\pagedirection	879	
\pagediscards	552	
\pagefullstretch	399	
\pagefillstretch	400	
\pagefilstretch	401	
\pagefistretch	1148	
\pagegoal	402	
\pageheight	930	
\pageleftoffset	880	
\pagerightoffset	881	
\pageshrink	403	
\pagestretch	404	
\pagetopoffset	882	
\pagetotal	405	
\pagewidth	931	
\paperheight	36707, 36711	
\paperwidth	36708, 36711	
\par	14–19, 87, 377, 1296, 406, 33125, 33127, 33131, 33136, 33141, 33146, 33153, 33158, 33165, 33170, 33190	
\pardir	883	
\pardirection	884	
\parfillskip	407	
\parindent	408	
\parshape	409	
\parshapedimen	553	
\parshapeindent	554	
\parshapelen	555	
\parskip	410	
\partokencontext	1190	
\partokenname	1191	
\patterns	411	

\pausing .....	412
pc .....	258
pdf commands:	
\pdf_destination:nn .....	302, 36674, 36674
\pdf_destination:nnnn .....	302, 36676, 36676
\pdf_object_if_exist:n .....	36611
\pdf_object_if_exist:nTF .....	300, 36590
\pdf_object_if_exist_p:n .....	300, 36590
\pdf_object_new:n .....	299, 303, 1384, 36590, 36590
\pdf_object_new:nn .....	303, 1384, 36719, 36719
\pdf_object_ref:n .....	299, 36590, 36603
\pdf_object_ref_last: .....	300, 36590, 36610
\pdf_object_unnamed_write:nn .....	300, 36590, 36604, 36609
\pdf_object_write:nn .....	299, 303, 1384, 36719, 36724, 36730
\pdf_object_write:nnn .....	299, 303, 1384, 36590, 36597, 36602
\pdf_pageobject_ref:n .....	300, 36617, 36617
\pdf_pagesize_gset:nn .....	301, 36672, 36672
\pdf_pagobject_ref:n .....	300
\pdf_uncompress: .....	301, 36582, 36582
\pdf_version: .....	300, 36668, 36668
\pdf_version_compare:Nn .....	300, 36619, 36619
\pdf_version_compare:NnTF .....	300, 36657
\pdf_version_compare_p:Nn .....	300
\pdf_version_gset:n .....	300, 36653, 36653
\pdf_version_major: .....	300, 36668, 36670
\pdf_version_min_gset:n .....	300, 36653, 36655
\pdf_version_minor: .....	300, 36668, 36671
pdf internal commands:	
\__pdf_backend_compress_objects:n .....	36587
\__pdf_backend_compresslevel:n .....	36586
\__pdf_backend_destination:nn .....	36675
\__pdf_backend_destination:nnnn .....	36679
\__pdf_backend_object_last: .....	36610
\__pdf_backend_object_new:n .....	36592, 36722
\__pdf_backend_object_now:nn .....	36606
\__pdf_backend_object_ref:n .....	36603
\__pdf_backend_object_write:nnn .....	36599, 36726
\__pdf_backend_pageobject_ref:n .....	36618
\__pdf_backend_pagesize_gset:nn .....	36698, 36710
\__pdf_backend_version_major: .....	36624, 36632, 36635, 36644, 36647, 36669, 36670
\__pdf_backend_version_major_-gset:n .....	36664
\__pdf_backend_version_minor: .....	36625, 36636, 36648, 36669, 36671
\__pdf_backend_version_minor_-gset:n .....	36665
\g__pdf_init_bool .....	36571, 36584, 36600, 36607, 36662, 36728
\g__pdf_object_prop .....	36718, 36721, 36727
\__pdf_pagesize_gset:nn .....	36673
\__pdf_version_compare_<:w .....	36619
\__pdf_version_compare_=:w .....	36619
\__pdf_version_compare_>:w .....	36619
\__pdf_version_gset:w .....	36653, 36654, 36658, 36660
\pdfadjustspacing .....	650
\pdfannot .....	578
\pdfcatalog .....	579
\pdfcolorstack .....	581
\pdfcolorstackinit .....	582
\pdfcompresslevel .....	580
\pdfcopyfont .....	651
\pdfcreationdate .....	583
\pdfdecimaldigits .....	584
\pdfdest .....	585
\pdfdestmargin .....	586
\pdfdraftmode .....	652
\pdfeachlinedepth .....	653
\pdfeachlineheight .....	654
\pdfelapsedtime .....	655
\pdfendlink .....	587
\pdfendthread .....	588
\pdfextension .....	885
\pdffeedback .....	886
\pdffiledump .....	712
\pdffilemoddate .....	711
\pdffilesize .....	709
\pdffirstlineheight .....	656
\pdffontattr .....	589
\pdffontexpand .....	657
\pdffontname .....	590
\pdffontobjnum .....	591
\pdffontsize .....	658
\pdfgamma .....	592
\pdfgentounicode .....	595
\pdfglyptounicode .....	596
\pdfhorigin .....	597
\pdfignoreddimen .....	659

\pdfimageapplygamma .....	593	\pdfsetmatrix .....	634
\pdfimagegamma .....	594	\pdfsetrandomseed .....	678
\pdfimagehicolor .....	598	\pdfshellescape .....	679
\pdfimageresolution .....	599	\pdfstartlink .....	635
\pdfincludechars .....	600	\pdfstartthread .....	636
\pdfinclusioncopyfonts .....	601	\pdfstrcmp .....	125, 715, 22, 708
\pdfinclusionerrorlevel .....	602	\pdfsuppressptexinfo .....	637
\pdfinfo .....	604	pdftex commands:	
\pdfinsertht .....	660	\pdftex_if_engine:TF .....	
\pdflastannot .....	605	..... 37550, 37552, 37554	
\pdflastlinedepth .....	661	\pdftex_if_engine_p: .....	37548
\pdflastlink .....	606	\pdftexbanner .....	682
\pdflastobj .....	607	\pdftexrevision .....	683
\pdflastxform .....	608	\pdftexversion .....	684
\pdflastximage .....	609	\pdfthread .....	638
\pdflastximagecolordepth .....	610	\pdfthreadmargin .....	639
\pdflastximagepages .....	612	\pdftracingfonts .....	680, 1240, 1241
\pdflastxpos .....	662	\pdftrailer .....	640
\pdflastypos .....	663	\pdfuniformdeviate .....	681
\pdflinkmargin .....	613	\pdfuniqueresource .....	641
\pdfliteral .....	614	\pdfvariable .....	887
\pdfmajorversion .....	615	\pdfvorigin .....	642
\pdfmapfile .....	664	\pdfxform .....	643
\pdfmapline .....	665	\pdfxformname .....	644
\pdfmdfivesum .....	710	\pdfximage .....	645
\pdfminorversion .....	616	\pdfximagebbox .....	646
\pdfnames .....	617	peek commands:	
\pdfnoligatures .....	666	\peek_after:Nw .....	
\pdfnormaldeviate .....	667	..... 71, 191, 192, 4233, 19488, 19488, 19501, 19526, 19567, 37330	
\pdfobj .....	618	\peek_analysis_map_break: .....	
\pdfobjcompresslevel .....	619	... 194, 4196, 4196, 4197, 4199, 4219	
\pdfoutline .....	620	\peek_analysis_map_break:n .....	
\pdfoutput .....	621	..... 194, 4196, 4198, 8020	
\pdfpageattr .....	622	\peek_analysis_map_inline:n .. 45, 191, 194, 434, 559, 4208, 4208, 8013	
\pdfpagebox .....	624	\peek_catcode:NTF .. 192, 19622, 35424	
\pdfpageheight .....	668	\peek_catcode_collect_inline:Nn .. 314, 37310, 37310	
\pdfpageref .....	625	\peek_catcode_ignore_spaces:NTF .. ..... 37713	
\pdfpageresources .....	626	\peek_catcode_remove:NTF .. 192, 19622, 35490	
\pdfpagesattr .....	623, 627	\peek_catcode_remove_ignore_- spaces:NTF .. 37713	
\pdfpagewidth .....	669	\peek_charcode:NTF 192, 194, 195, 19622	
\pdfpkmode .....	670	\peek_charcode_collect_inline:Nn .. 314, 37310, 37312	
\pdfpkresolution .....	671	\peek_charcode_ignore_spaces:NTF .. 37713	
\pdfprimitive .....	672	\peek_charcode_remove:NTF .. 192, 195, 19622	
\pdfprotrudechars .....	673	\peek_charcode_remove_ignore_- spaces:NTF .. 37713	
\pdfpxdimen .....	674		
\pdfrandomseed .....	675		
\pdfrefobj .....	628		
\pdfrefxform .....	629		
\pdfrefximage .....	630		
\pdfresettimer .....	676		
\pdfrestore .....	631		
\pdfretval .....	632		
\pdfsave .....	633		
\pdfsavepos .....	677		

```

\peek_gafter:Nw . . . 192, 19488, 19490
\peek_meaning:NTF . . . . . 192, 19622
\peek_meaning_collect_inline:Nn .
. . . . . 314, 37310, 37314
\peek_meaning_ignore_spaces:NTF .
. . . . . 37713
\peek_meaning_remove:NTF . 193, 19622
\peek_meaning_remove_ignore-
spaces:NTF . . . . . 37713
\peek_N_type:TF . . . . .
. . . . . 193, 19636, 19668, 19673, 19675
\peek_regex:NTF . . . . .
. . . . . 194, 7957, 7966, 7972, 7973, 7974
\peek_regex:nTF . . . 194, 510, 558,
560, 561, 7957, 7957, 7963, 7964, 7965
\peek_regex_remove_once:n . . . 194
\peek_regex_remove_once:NTF . .
195, 7957, 7985, 7991, 7992, 7993, 7994
\peek_regex_remove_once:nTF 195,
560, 7957, 7975, 7981, 7982, 7983, 7984
\peek_regex_replace_once:Nn . .
. . . . . 195, 8049, 8063
\peek_regex_replace_once:nn . .
. . . . . 195, 8049, 8055
\peek_regex_replace_once:NnTF . .
. . . . . 195, 8049,
8057, 8059, 8060, 8061, 8062, 8064
\peek_regex_replace_once:nnTF . .
. . . . . 195, 532, 536, 558, 563, 8049,
8049, 8051, 8052, 8053, 8054, 8056
\peek_remove_filler:n . .
. . . . . 193, 19513, 19513, 35421, 35488
\peek_remove_spaces:n . .
. . . . . 193, 19497, 19497, 37722,
37725, 37728, 37731, 37734, 37737
\g_peek_token . . . . . 192, 19477, 19491
\l_peek_token . . . . .
. . . . . 191, 192, 194, 453–455,
886, 888–890, 1357, 1406, 4238,
4239, 4240, 4241, 4330, 4395, 4398,
4443, 19477, 19489, 19506, 19531,
19534, 19545, 19583, 19595, 19615,
19642, 19643, 19644, 19647, 35437,
35444, 35458, 37336, 37337, 37338
peek internal commands:
  \__peek_collect:N . . . .
. . . . . 1406, 37310, 37340, 37346
  \__peek_collect>NNn . . . .
. . . . . 37310, 37311, 37313, 37315, 37316
  \__peek_collect_remove:nw . . .
. . . . . 37310, 37341, 37342, 37343, 37351
\l__peek_collect_t1 . . . . . 1406,
37309, 37321, 37323, 37348, 37353
  \__peek_collect_true:w . . . .
. . . . . 1406, 37310, 37329, 37333
  \__peek_execute_branches_...: . 1406
  \__peek_execute_branches_-
catcode: . 889, 19589, 19589, 37311
  \__peek_execute_branches_-
catcode_aux: . . . .
. . . . . 19589, 19590, 19592, 19593
  \__peek_execute_branches_-
catcode_auxii:N 19589, 19597, 19603
  \__peek_execute_branches_-
catcode_auxiii: 19589, 19600, 19613
  \__peek_execute_branches_-
charcode: 889, 19589, 19591, 37313
  \__peek_execute_branches_-
meaning: . 889, 19581, 19581, 37315
  \__peek_execute_branches_N_type:
. . . 19636, 19639, 19671, 19674, 19676
  \__peek_false:w . . . .
. . . . . 890, 1406, 19481, 19483, 19499,
19510, 19516, 19546, 19562, 19586,
19609, 19619, 19653, 19666, 37322
  \__peek_false_aux:n 1406, 37323, 37324
  \__peek_N_type:w . 19636, 19646, 19656
  \__peek_N_type_aux:nnw . .
. . . . . 19636, 19648, 19661
  \__peek_remove_filler: . . .
. . . . . 19513, 19526, 19529
  \__peek_remove_filler:w . . .
. . . . . 19513, 19515, 19522, 19524, 19548
  \__peek_remove_filler_expand:w . .
. . . . . 19513, 19539, 19543
  \__peek_remove_spaces: . . .
. . . . . 19497, 19501, 19504
\l__peek_search_t1 885, 888, 1406,
19480, 19555, 19606, 19616, 37320
\l__peek_search_token . . .
885, 1406, 19479, 19554, 19583, 37319
  \__peek_tmp:w . . .
. . . . . 19481, 19484, 19495, 19637, 19659
  \__peek_token_generic:NNTF . 889,
890, 19569, 19569, 19571, 19572,
19573, 19574, 19670, 19674, 19676
  \__peek_token_generic_aux:NNNTF . .
. . . . . 19551, 19551, 19570, 19576
  \__peek_token_remove_generic:NNTF .
. . . . . 889, 19569,
19575, 19577, 19578, 19579, 19580
  \__peek_true:w . . . . . 890, 1406,
19481, 19481, 19561, 19584, 19607,
19617, 19651, 19665, 19666, 37329
  \__peek_true_aux:w 886, 887, 19481,
19482, 19494, 19501, 19502, 19515,
19556, 19570, 37330, 37331, 37349

```

\\_\_peek\_true\_remove:w .....  
       ..... 886, 887, 19492, 19492,  
       19507, 19532, 19536, 19576, 37354  
 \\_\_peek\_use\_none\_delimit\_by\_s\_-  
       stop:w ... 890, 19487, 19487, 19649  
 \penalty ..... 413  
 \pi ..... 23551, 23552  
 pi ..... 257  
 \pm ..... 25022, 25023  
 \postbreakpenalty ..... 1149  
 \postdisplaypenalty ..... 414  
 \postexhyphenchar ..... 888  
 \posthyphenchar ..... 889  
 \prebinoppenalty ..... 890  
 \prebreakpenalty ..... 1150  
 \predisplaydirection ..... 556  
 \predisplaygapfactor ..... 891  
 \predisplaypenalty ..... 415  
 \predisplaysize ..... 416  
 \preexhyphenchar ..... 892  
 \prehyphenchar ..... 893  
 \prerelpenalty ..... 894  
 \pretolerance ..... 417  
 \prevdepth ..... 418  
 \prevgraf ..... 419  
 prg commands:  
     \prg\_break: 71, 531, 758, 808, 809,  
       2276, 2277, 3618, 3693, 4050, 4128,  
       4158, 4159, 4160, 4161, 4162, 4163,  
       4530, 4798, 4802, 6060, 6070, 6075,  
       6084, 6108, 6153, 7020, 7048, 7836,  
       8781, 11180, 13073, 14130, 14146,  
       14264, 14296, 14402, 14405, 14546,  
       14593, 14646, 14652, 14885, 14966,  
       15138, 15279, 16884, 16917, 16968,  
       17021, 17561, 18079, 18561, 22664,  
       22673, 24673, 24693, 24694, 24918,  
       24919, 24932, 25032, 25033, 25034,  
       28425, 28477, 28701, 36944, 36950  
     \prg\_break:n .....  
       71, 2276, 2278, 6556, 8781, 13075,  
       14032, 14040, 14052, 16758, 16897,  
       17571, 22402, 22421, 22435, 22680  
     \prg\_break\_point: ..... 71, 421,  
       428, 650, 1396, 2276, 2277,  
       2278, 3446, 3487, 3611, 3618, 3968,  
       4129, 4165, 4526, 4776, 6056, 6105,  
       6557, 6890, 7042, 7830, 7836, 8781,  
       11153, 13063, 14033, 14041, 14131,  
       14147, 14265, 14297, 14403, 14406,  
       14547, 14594, 14647, 14653, 14886,  
       15086, 15243, 16755, 16885, 16919,  
       16970, 17021, 17066, 17073, 17566,  
       18079, 18561, 22396, 22415, 22430,  
       22665, 22674, 24674, 24695, 24920,  
       25036, 28426, 28477, 28709, 36945  
 \prg\_break\_point:Nn .....  
       70, 71, 141, 386, 450, 461,  
       809, 829, 913, 2267, 2267, 2268,  
       4093, 4219, 6781, 6795, 6841, 8019,  
       8781, 10368, 10387, 12619, 12648,  
       12659, 13493, 13519, 13539, 13561,  
       16920, 16961, 16971, 16994, 17001,  
       17010, 17613, 18434, 18456, 18478,  
       18505, 18527, 19997, 20019, 20035,  
       20364, 25105, 31930, 31949, 32269  
 \prg\_do\_nothing: .....  
       13, 71, 480, 534, 554,  
       657, 681, 682, 739, 799, 845, 862,  
       895, 998, 1175, 2265, 2265, 2276,  
       2600, 2992, 3019, 3118, 3119, 3120,  
       3533, 3691, 3692, 3939, 3988, 4265,  
       4284, 4624, 5106, 5149, 5150, 5157,  
       5158, 7103, 7331, 7789, 7806, 7810,  
       8983, 10721, 11016, 11449, 11486,  
       11488, 12292, 12922, 12957, 13315,  
       13317, 14196, 15136, 16419, 16420,  
       16557, 16564, 16850, 16852, 18072,  
       18078, 18086, 18241, 18454, 18463,  
       18526, 18537, 18612, 18624, 18678,  
       18682, 18689, 21765, 22946, 22980,  
       23006, 23014, 24558, 28406, 29080  
 \prg\_generate\_conditional\_-\_  
       variant:Nnn .... 64, 3212, 3212,  
       8522, 10211, 12448, 12458, 12482,  
       12492, 12508, 12525, 12536, 12610,  
       12860, 12878, 13394, 13406, 13414,  
       13445, 16240, 16262, 16691, 16759,  
       16853, 16855, 16869, 16871, 16873,  
       16875, 18259, 18273, 18274, 18423,  
       18425, 19912, 19913, 19961, 19974,  
       19985, 21908, 32989, 32991, 32995,  
       33622, 36996, 36997, 37070, 37071  
 \prg\_gset\_conditional:Nnn .....  
       ... 62, 1613, 1615  
 \prg\_gset\_conditional:Npnn .....  
       ... 62, 1592, 1594  
 \prg\_gset\_protected\_conditional:Nnn .....  
       ... 63, 1613, 1621  
 \prg\_gset\_protected\_conditional:Npnn .....  
       ... 63, 1592, 1600  
 \prg\_map\_break:Nn .....  
       .. 70, 71, 386, 449, 695, 858, 904,  
       2267, 2268, 2274, 4197, 4199, 4521,  
       8781, 10351, 10353, 12686, 12688,  
       13552, 13554, 16908, 16910, 18540,  
       18542, 20051, 20053, 32259, 32261  
 \prg\_new\_conditional:Nnn .....

..... 62, [1613](#), [1617](#), [8472](#)  
 \prg\_new\_conditional:Npnn .....  
 ..... 62–64, [354](#), [701](#), [875](#),  
[889](#), [1592](#), [1596](#), [2152](#), [2871](#), [4899](#),  
[4919](#), [4941](#), [4987](#), [5011](#), [8472](#), [8514](#),  
[8557](#), [8618](#), [8633](#), [8644](#), [8659](#), [8669](#),  
[8765](#), [8767](#), [8769](#), [8771](#), [9267](#), [10285](#),  
[11325](#), [12050](#), [12440](#), [12450](#), [12465](#),  
[12474](#), [12484](#), [12540](#), [12556](#), [12567](#),  
[12848](#), [12862](#), [12880](#), [12919](#), [12939](#),  
[12954](#), [13377](#), [13384](#), [13389](#), [13396](#),  
[13401](#), [14029](#), [14038](#), [14053](#), [14061](#),  
[14733](#), [14767](#), [14786](#), [16224](#), [16232](#),  
[16242](#), [16252](#), [16401](#), [16683](#), [17382](#),  
[17435](#), [17473](#), [17481](#), [18029](#), [18034](#),  
[18088](#), [18376](#), [19129](#), [19134](#), [19139](#),  
[19144](#), [19151](#), [19157](#), [19163](#), [19168](#),  
[19173](#), [19178](#), [19183](#), [19188](#), [19193](#),  
[19198](#), [19205](#), [19220](#), [19225](#), [19261](#),  
[19369](#), [19378](#), [19956](#), [19963](#), [20180](#),  
[20185](#), [20479](#), [20487](#), [21901](#), [21909](#),  
[22848](#), [23993](#), [24822](#), [24830](#), [24846](#),  
[30002](#), [30066](#), [30075](#), [31337](#), [31366](#),  
[31399](#), [31429](#), [31454](#), [32985](#), [32987](#),  
[32993](#), [33612](#), [34858](#), [36611](#), [36619](#)  
\prg\_new\_eq\_conditional:Nnn ....  
. 64, [1736](#), [1738](#), [8472](#), [8553](#), [8555](#),  
[12136](#), [12137](#), [13366](#), [13368](#), [13370](#),  
[13372](#), [13374](#), [16588](#), [16590](#), [17138](#),  
[17139](#), [17140](#), [17141](#), [17142](#), [17143](#),  
[17331](#), [17333](#), [18177](#), [18179](#), [18372](#),  
[18374](#), [19952](#), [19954](#), [20111](#), [20113](#),  
[20453](#), [20455](#), [20546](#), [20548](#), [24820](#),  
[24821](#), [29138](#), [29140](#), [32930](#), [32932](#)  
\prg\_new\_protected\_conditional:Nnn ..... 63, [1613](#), [1623](#), [8472](#)  
\prg\_new\_protected\_conditional:Npnn ..... 63, [1592](#),  
1602, [5354](#), [7477](#), [7482](#), [7509](#), [7511](#),  
[8472](#), [8961](#), [10202](#), [10305](#), [10325](#),  
[10995](#), [11127](#), [11274](#), [11276](#), [11278](#),  
[11280](#), [11315](#), [11377](#), [12496](#), [12509](#),  
[12527](#), [13408](#), [13416](#), [14170](#), [14179](#),  
[16740](#), [16849](#), [16851](#), [16857](#), [16860](#),  
[16863](#), [16866](#), [18250](#), [18260](#), [18262](#),  
[18390](#), [18394](#), [19892](#), [19902](#), [19976](#),  
[29142](#), [36992](#), [36994](#), [37066](#), [37068](#)  
\prg\_replicate:nn ..... 69, [112](#), [149](#), [557](#), [581](#), [971](#), [4483](#),  
5112, [5867](#), [6487](#), [6513](#), [6659](#), [6667](#),  
[6830](#), [6996](#), [7108](#), [7748](#), [7756](#), [7816](#),  
[7932](#), [7934](#), [8717](#), [8717](#), [9381](#), [9563](#),  
[10553](#), [22349](#), [22509](#), [26250](#), [27102](#),  
[27410](#), [27666](#), [27712](#), [27749](#), [28272](#),  
28280, [28739](#), [28842](#), [29591](#), [29602](#),  
29607, [29632](#), [36001](#), [36009](#), [36077](#),  
36111, [36123](#), [36126](#), [36206](#), [36207](#)  
\prg\_return\_false: ..... 63, 64,  
366, [548](#), [804](#), [824](#), [854](#), [1586](#), [1588](#),  
1660, [1668](#), [1822](#), [1827](#), [1840](#), [1845](#),  
1853, [1870](#), [2155](#), [2881](#), [4913](#), [4924](#),  
[4927](#), [4932](#), [4936](#), [4937](#), [4945](#), [4948](#),  
[4953](#), [4956](#), [4993](#), [4996](#), [5017](#), [5020](#),  
[5361](#), [5366](#), [7587](#), [8472](#), [8519](#), [8562](#),  
[8623](#), [8639](#), [8649](#), [8665](#), [8675](#), [8766](#),  
[8768](#), [8770](#), [8772](#), [8965](#), [8973](#), [9270](#),  
10209, [10292](#), [10308](#), [10328](#), [11004](#),  
11132, [11157](#), [11287](#), [11307](#), [11321](#),  
11340, [11349](#), [11360](#), [11374](#), [11381](#),  
12054, [12445](#), [12455](#), [12470](#), [12479](#),  
12489, [12505](#), [12519](#), [12533](#), [12546](#),  
12563, [12578](#), [12857](#), [12875](#), [12893](#),  
12901, [12911](#), [12927](#), [12950](#), [12961](#),  
13382, [13387](#), [13392](#), [13399](#), [13404](#),  
13412, [13420](#), [14034](#), [14042](#), [14058](#),  
14074, [14177](#), [14186](#), [14737](#), [14740](#),  
14743, [14770](#), [14773](#), [14790](#), [14793](#),  
14796, [16229](#), [16237](#), [16248](#), [16258](#),  
16405, [16688](#), [16754](#), [16773](#), [17380](#),  
17412, [17417](#), [17440](#), [17478](#), [17486](#),  
18032, [18039](#), [18104](#), [18107](#), [18253](#),  
18267, [18379](#), [18414](#), [18420](#), [19132](#),  
19137, [19142](#), [19147](#), [19154](#), [19161](#),  
19166, [19171](#), [19176](#), [19181](#), [19186](#),  
19191, [19196](#), [19201](#), [19218](#), [19223](#),  
19228, [19233](#), [19267](#), [19270](#), [19282](#),  
19382, [19407](#), [19424](#), [19433](#), [19900](#),  
19910, [19959](#), [19967](#), [19983](#), [20183](#),  
20202, [20217](#), [20218](#), [20483](#), [20490](#),  
21906, [21915](#), [22859](#), [22861](#), [24006](#),  
24016, [24827](#), [24841](#), [24854](#), [29152](#),  
29158, [30012](#), [30015](#), [30070](#), [30080](#),  
31345, [31349](#), [31355](#), [31391](#), [31422](#),  
31448, [31470](#), [32986](#), [32988](#), [32994](#),  
33618, [33620](#), [34863](#), [34866](#), [36615](#),  
36627, [36639](#), [36651](#), [37024](#), [37082](#)  
\prg\_return\_true: ..... 63, 64, 366, 545, 548, 647, 688, 701, 804, 900, 1399, 1586, 1588, 1660, 1668, 1825, 1842, 1850, 1855, 1868, 1873, 2155, 2873, 2881, 4902, 4916, 4924, 4927, 4932, 4936, 4948, 4953, 4956, 4991, 5015, 5357, 5363, 7585, 8472, 8517, 8560, 8621, 8637, 8647, 8663, 8673, 8766, 8768, 8770, 8772, 8986, 9270, 10207, 10290, 10295, 10298, 10311, 10331, 11002, 11133, 11171, 11288, 11322, 11338,

11347, 11358, 11380, 12056, 12443,  
 12453, 12468, 12477, 12487, 12503,  
 12517, 12533, 12544, 12561, 12576,  
 12855, 12873, 12891, 12909, 12925,  
 12948, 12959, 13382, 13387, 13392,  
 13399, 13404, 13412, 13420, 14052,  
 14056, 14064, 14077, 14177, 14186,  
 14737, 14743, 14775, 14790, 14796,  
 16227, 16235, 16246, 16256, 16405,  
 16686, 16758, 16776, 17412, 17438,  
 17476, 17484, 18032, 18037, 18100,  
 18103, 18109, 18256, 18270, 18380,  
 18410, 18420, 19132, 19137, 19142,  
 19147, 19154, 19161, 19166, 19171,  
 19176, 19181, 19186, 19191, 19196,  
 19201, 19217, 19223, 19231, 19281,  
 19405, 19431, 19898, 19908, 19959,  
 19972, 19981, 20183, 20218, 20482,  
 20491, 21905, 21914, 22852, 22857,  
 24001, 24022, 24825, 24843, 24852,  
 29148, 30013, 30070, 30080, 31353,  
 31358, 31362, 31374, 31377, 31380,  
 31383, 31386, 31389, 31408, 31411,  
 31414, 31417, 31420, 31437, 31440,  
 31443, 31446, 31462, 31465, 31468,  
 32986, 32988, 32994, 33617, 34864,  
 36614, 36626, 36638, 36650, 37044  
 $\backslash$ prg\_set\_conditional:Nnn . . . . .  
     . . . . . 62, 1613, 1613, 8472  
 $\backslash$ prg\_set\_conditional:Npnn . . . . .  
     . . . . . 62-64, 1592, 1592,  
     1819, 1831, 1847, 1859, 8472, 11368  
 $\backslash$ prg\_set\_eq\_conditional:NNn . . . . .  
     . . . . . 64, 1736, 1736, 8472  
 $\backslash$ prg\_set\_protected\_conditional:Nnn . . . . .  
     . . . . . 63, 1613, 1619, 8472  
 $\backslash$ prg\_set\_protected\_conditional:Npnn . . . . .  
     . . . . . 63, 1592, 1598, 8472, 11140  
 prg internal commands:  
 $\backslash$ \_\_prg\_break\_point:Nn . . . . . 385  
 $\backslash$ \_\_prg\_F\_true:w . . . . . 1689, 1722, 1734  
 $\backslash$ \_\_prg\_generate\_conditional:nnNNNnnn . . . . .  
     . . . . . 1608, 1637, 1646, 1646  
 $\backslash$ \_\_prg\_generate\_conditional:NNnnnnNw . . . . .  
     . . . . . 1646, 1655, 1672, 1687  
 $\backslash$ \_\_prg\_generate\_conditional\_-  
     count:NNNnn . . . . . 1613, 1614,  
     1616, 1618, 1620, 1622, 1624, 1625  
 $\backslash$ \_\_prg\_generate\_conditional\_-  
     count:nnNNNnn . . . . . 1613, 1629, 1634  
 $\backslash$ \_\_prg\_generate\_conditional\_-  
     fast:nw . . . . . 366, 367, 1646, 1659, 1670  
 $\backslash$ \_\_prg\_generate\_conditional\_-  
     parm:NNNpnn . . . . . 1592, 1593,  
                        1595, 1597, 1599, 1601, 1603, 1604  
 $\backslash$ \_\_prg\_generate\_conditional\_-  
     test:w . . . . . 1646, 1657, 1667  
 $\backslash$ \_\_prg\_generate\_F\_form:wNNnnnnN . . . . .  
     . . . . . 1689, 1716  
 $\backslash$ \_\_prg\_generate\_p\_form:wNNnnnnN . . . . .  
     . . . . . 366, 1689, 1689  
 $\backslash$ \_\_prg\_generate\_T\_form:wNNnnnnN . . . . .  
     . . . . . 1689, 1708  
 $\backslash$ \_\_prg\_generate\_TF\_form:wNNnnnnN . . . . .  
     . . . . . 1689, 1724  
 $\backslash$ \_\_prg\_p\_true:w . . . . . 1689, 1701, 1732  
 $\backslash$ \_\_prg\_replicate:N . . . . .  
     . . . . . 8717, 8724, 8725, 8727  
 $\backslash$ \_\_prg\_replicate\_ . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_0:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_1:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_2:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_3:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_4:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_5:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_6:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_7:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_8:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_9:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first:N . . . . .  
     . . . . . 8717, 8720, 8726  
 $\backslash$ \_\_prg\_replicate\_first\_-:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_0:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_1:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_2:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_3:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_4:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_5:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_6:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_7:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_8:n . . . . . 8717  
 $\backslash$ \_\_prg\_replicate\_first\_9:n . . . . . 8717  
 $\backslash$ \_\_prg\_set\_eq\_conditional:NNNn . . . . .  
     . . . . . 1736, 1737, 1739, 1740  
 $\backslash$ \_\_prg\_set\_eq\_conditional:nnNnnNw . . . . .  
     . . . . . 1744, 1752, 1752  
 $\backslash$ \_\_prg\_set\_eq\_conditional\_F\_-  
     form:nnn . . . . . 1752  
 $\backslash$ \_\_prg\_set\_eq\_conditional\_F\_-  
     form:wNnnn . . . . . 1789  
 $\backslash$ \_\_prg\_set\_eq\_conditional\_-  
     loop:nnnnNw . . . . . 1752, 1764, 1766, 1781  
 $\backslash$ \_\_prg\_set\_eq\_conditional\_p\_-  
     form:nnn . . . . . 1752  
 $\backslash$ \_\_prg\_set\_eq\_conditional\_p\_-  
     form:wNnnn . . . . . 1783  
 $\backslash$ \_\_prg\_set\_eq\_conditional\_T\_-  
     form:nnn . . . . . 1752

\\_\_prg\_set\_eq\_conditional\_T\_-  
   form:wNnnnn ..... 1787  
 \\_\_prg\_set\_eq\_conditional\_TF\_-  
   form:nnn ..... 1752  
 \\_\_prg\_set\_eq\_conditional\_TF\_-  
   form:wNnnnn ..... 1785  
 \\_\_prg\_T\_true:w ..... 1689, 1714, 1733  
 \\_\_prg\_TF\_true:w 368, 1689, 1730, 1735  
 \\_\_prg\_use\_none\_delimit\_by\_q\_-  
   recursion\_stop:w .....  
   .. 1590, 1590, 1675, 1757, 1762, 1769  
\primitive ..... 780  
prop commands:  
  \c\_empty\_prop ..... 206,  
   894, 19685, 19695, 19699, 19702, 19958  
  \prop\_clear:N .....  
   ..... 199, 19698, 19698, 19700,  
   19705, 19737, 19751, 34400, 35342  
  \prop\_clear\_new:N .... 199, 19704,  
   19704, 19706, 35542, 35573, 35614  
  \prop\_concat:NNN .....  
   ..... 201, 894, 19723, 19723, 19725  
  \prop\_const\_from\_keyval:Nn .....  
   ..... 200, 19735,  
   19747, 19753, 33581, 33588, 36319  
  \prop\_count:N .....  
   .... 202, 19846, 19846, 19855, 36926  
  \prop\_gclear:N .....  
   199, 19698, 19701, 19703, 19708, 19743  
  \prop\_gclear\_new:N .... 199, 1312,  
   19704, 19707, 19709, 33655, 33656  
  \prop\_gconcat:NNN .....  
   ..... 201, 19723, 19726, 19728  
  \prop\_get:Nn ..... 71, 37556, 37558  
  \prop\_get:NnN ..... 138, 139, 201,  
   202, 19805, 19805, 19811, 19812,  
   19976, 19985, 34640, 34644, 34713,  
   34717, 34873, 34988, 35076, 36116  
  \prop\_get:NnNTF .. 201, 203, 9671,  
   9691, 9746, 10264, 10487, 14237,  
   19976, 21231, 29011, 33846, 34982,  
   35081, 35552, 35810, 35823, 35838,  
   35917, 35945, 35961, 36332, 36345  
  \prop\_gpop:NnN ..... 202, 19813,  
   19822, 19833, 19834, 19902, 19913  
  \prop\_gpop:NnNTF .... 202, 204, 19892  
  \prop\_gput:N ..... 227, 21385  
  \prop\_gput:Nnn ..... 200,  
   3745, 3746, 8430, 8431, 9463, 10158,  
   10159, 10166, 10167, 10169, 10170,  
   10171, 10172, 10192, 10238, 10423,  
   10462, 11772, 11773, 14000, 14001,  
   14002, 14003, 14004, 14005, 14006,  
   14007, 14008, 14009, 14010, 14011,  
   14012, 14013, 14014, 14021, 14024,  
   14873, 14874, 19767, 19914, 19915,  
   19931, 19933, 20779, 20780, 22013,  
   22014, 22919, 22920, 28964, 29293,  
   29294, 33872, 33890, 33925, 33956,  
   35735, 35736, 35737, 35738, 35739,  
   35740, 35741, 35742, 35753, 35755,  
   35757, 35758, 35759, 35760, 35761,  
   35762, 35763, 35863, 35864, 35878,  
   35892, 35902, 36558, 36559, 36721  
  \prop\_gput\_from\_keyval:Nn .....  
   ..... 201, 19735, 19744, 19762, 19769  
  \prop\_gput\_if\_new:Nnn .....  
   ..... 201, 19935, 19937, 19951  
  \prop\_gremove:Nn 202, 10249, 10472,  
   19789, 19795, 19803, 19804, 28962  
  \prop\_gset\_eq:NN .... 200, 19702,  
   19710, 19714, 19715, 19716, 19717,  
   19727, 33657, 33659, 33824, 33826,  
   33863, 33865, 34112, 34278, 34319  
  \prop\_gset\_from\_keyval:Nn .....  
   ..... 200, 19735, 19741, 19746  
  \prop\_hput:Nnn ..... 19914  
  \prop\_if\_empty:N ..... 19956, 19961  
  \prop\_if\_empty:NTF .....  
   203, 19861, 19881, 19956, 34862, 36923  
  \prop\_if\_empty\_p:N ..... 203, 19956  
  \prop\_if\_exist:N ..... 19952, 19954  
  \prop\_if\_exist:NTF .....  
   202, 19705, 19708, 19952, 21128, 34860  
  \prop\_if\_exist\_p:N ..... 202, 19952  
  \prop\_if\_in:Nn ..... 19963, 19974  
  \prop\_if\_in:NnTF .....  
   ..... 203, 9466, 9472, 19963, 35554  
  \prop\_if\_in\_p:Nn ..... 203, 19963  
  \prop\_item:Nn ..... 202, 204,  
   9467, 9473, 19835, 19835, 19845,  
   36179, 36218, 36727, 37557, 37559  
  \prop\_log:N .. 206, 20054, 20056, 20057  
  \prop\_map\_break: ... 205, 903, 904,  
   19993, 19994, 19995, 19996, 19997,  
   20019, 20031, 20032, 20033, 20034,  
   20035, 20050, 20050, 20051, 20053  
  \prop\_map\_break:n .....  
   ..... 205, 19843, 19972, 20050, 20052  
  \prop\_map\_function:NN .....  
   ..... 204, 308, 904, 10279,  
   10502, 19851, 19869, 19884, 19987,  
   19987, 20011, 20069, 34788, 36407  
  \prop\_map\_inline:Nn .....  
   . 204, 19732, 20012, 20012, 20026,  
   34122, 34124, 34127, 34145, 34147,  
   34221, 34238, 34299, 34301, 34305,  
   34307, 34487, 34506, 34687, 34696

```

\prop_map_tokens:Nn  204, 810, 898,
902, 19837, 19965, 20027, 20027, 20049
\prop_new:N ..... 199, 9461, 9462,
9484, 9643, 10180, 10411, 13999,
19692, 19692, 19697, 19705, 19708,
19718, 19719, 19720, 19721, 19722,
20854, 20855, 21128, 28915, 34100,
34101, 34102, 34570, 34611, 35639,
35729, 35734, 35751, 35756, 36718
\prop_pop:NnN ..... 201, 19813,
19813, 19831, 19832, 19892, 19912
\prop_pop:NnTF ..... 201, 203, 19892
\prop_put:N ..... 227, 21385
\prop_put:Nnn ..... 200, 201, 407,
892, 893, 9719, 9735, 9752, 19732,
19759, 19914, 19914, 19927, 19929,
21238, 33869, 33887, 33906, 33923,
33954, 34156, 34158, 34164, 34166,
34175, 34181, 34189, 34248, 34256,
34346, 34352, 34360, 34367, 34511,
34571, 34573, 34575, 34577, 34579,
34581, 34583, 34585, 34587, 34589,
34591, 34593, 34595, 34597, 34599,
34601, 34603, 34605, 34960, 35343,
35543, 35561, 35604, 35619, 35800
\prop_put_from_keyval:Nn .....
.... 201, 19735, 19738, 19754, 19761
\prop_put_if_new:Nnn .....
.... 201, 19935, 19935, 19950
\prop_rand_key_value:N .....
.... 309, 36921, 36921, 36930
\prop_remove:Nn .....
.... 202, 9716, 9731, 19789, 19789,
19801, 19802, 34682, 34685, 34689
\prop_set_eq:NN  200, 19699, 19710,
19710, 19711, 19712, 19713, 19724,
19731, 33812, 33814, 33856, 33858,
34109, 34118, 34120, 34271, 34295,
34297, 34316, 34444, 34677, 35624
\prop_set_from_keyval:Nn  200, 201,
894, 19735, 19735, 19740, 19749, 35779
\prop_show:N . 205, 20054, 20054, 20055
\prop_to_keyval:N .....
.... 202, 19856, 19858, 19878
\g_tmpa_prop ..... 206, 19718
\l_tmpa_prop ..... 206, 19718
\g_tmpb_prop ..... 206, 19718
\l_tmpb_prop ..... 206, 19718
prop internal commands:
\__prop_concat:NNNN .....
.... 19723, 19724, 19727, 19729
\__prop_count:nn . 19846, 19851, 19854
\__prop_from_keyval_key:w .... 895
\__prop_from_keyval_value:w ... 895
\__prop_if_in:nnn  19963, 19966, 19969
\__prop_if_recursion_tail_stop:n
..... 19690, 19690, 19691
\l__prop_internal_prop .....
.... 19722, 19731,
19732, 19733, 19749, 19750, 19751
\l__prop_internal_tl .....
.... 900, 19681, 19684,
19918, 19924, 19925, 19941, 19948
\__prop_item:nnn .....
.... 898, 19835, 19838, 19840
\__prop_keyval_parse:NNNn .....
... 19757, 19758, 19765, 19766, 19772
\__prop_map_function:Nw .....
.... 903, 19987, 19990, 19999, 20009
\__prop_map_tokens:nw .....
.... 20027, 20030, 20037, 20047
\__prop_missing_eq:n .....
.... 19735, 19770, 19775
\__prop_pair:wn  892, 896, 903, 904,
19681, 19682, 19682, 19783, 19786,
19920, 19943, 19993, 19994, 19995,
19996, 20000, 20001, 20002, 20003,
20015, 20017, 20022, 20031, 20032,
20033, 20034, 20038, 20039, 20040,
20041, 20064, 20073, 20076, 36931
\__prop_put:Nnn .....
.... 19914, 19914, 19915, 19916
\__prop_put_if_new:NNnn .....
.... 19935, 19936, 19938, 19939
\__prop_rand_item:w .....
.... 36921, 36925, 36931, 36935
\__prop_show>NN .....
.... 20054, 20054, 20056, 20058
\__prop_show_validate:w .....
.... 20054, 20063, 20073, 20077
\__prop_split:NnTF .....
.... 893, 900–902, 19778, 19778,
19791, 19797, 19807, 19815, 19824,
19894, 19904, 19923, 19946, 19978
\__prop_split_aux:NnTF .....
.... 19778, 19779, 19780
\__prop_split_aux:w .....
.... 896, 19778, 19782, 19785, 19788
\__prop_to_keyval:nn .....
.... 19856, 19869, 19874
\__prop_to_keyval:nnw .....
.... 19856, 19884, 19889
\__prop_to_keyval_exp_after:wN ..
.... 899, 19856, 19885, 19888, 19889, 19890
\__prop_use_i_delimit_by_s_-
stop:nw ..... 36920, 36920, 36934
\protect ..... 1234, 10620, 23486, 30191, 30479,
```

30494, 30503, 30517, 30519, 32439  
 \protected 117, 119, 143, 557, 19295, 19297  
 \protrudechars ..... 932  
 \ProvidesExplClass ..... 9  
 \ProvidesExplFile ..... 9  
 \ProvidesExplPackage ..... 9  
 pt ..... 258  
 \ptexlineendmode ..... 1151  
 \ptexminorversion ..... 1152  
 \ptexrevision ..... 1153  
 \ptexversion ..... 1154  
 \pxdimen ..... 933

## Q

quark commands:

\q\_mark ..... 139, 434, 16172  
 \q\_nil ..... 25,  
     119, 139, 362, 786, 788, 790, 1548,  
     1551, 16172, 16226, 16245, 16251,  
     16266, 16267, 16273, 16297, 16301,  
     21850, 21851, 21853, 21855, 21857,  
     21859, 21865, 36062, 36068, 36069  
 \q\_no\_value .....  
     74, 87, 88, 94–96, 138, 139,  
     145, 146, 152, 179, 201, 202, 311,  
     786, 788, 805, 806, 849, 897, 1399,  
     8959, 10302, 10315, 10993, 11124,  
     11152, 11267, 11269, 11271, 11273,  
     11313, 16172, 16234, 16255, 16261,  
     16764, 16772, 16784, 16810, 18216,  
     18231, 19809, 19820, 19829, 37083  
 \quark\_if\_nil:N ..... 16224  
 \quark\_if\_nil:n 788, 789, 16242, 16262  
 \quark\_if\_nil:NTF 139, 354, 790, 16224  
 \quark\_if\_nil:nTF ..... 139,  
     691, 787, 788, 790, 16242, 36076, 36089  
 \quark\_if\_nil\_p:N ..... 139, 16224  
 \quark\_if\_nil\_p:n ..... 139, 16242  
 \quark\_if\_no\_value:N .. 16232, 16240  
 \quark\_if\_no\_value:n ..... 16252  
 \quark\_if\_no\_value:NTF 139, 11154,  
     16224, 34642, 34646, 34715, 34719  
 \quark\_if\_no\_value:nTF ... 139, 16242  
 \quark\_if\_no\_value\_p:N ... 139, 16224  
 \quark\_if\_no\_value\_p:n ... 139, 16242  
 \quark\_if\_recursion\_tail\_break:N  
     ..... 37560  
 \quark\_if\_recursion\_tail\_break:n  
     ..... 37562  
 \quark\_if\_recursion\_tail\_-  
     break:NN ... 141, 789, 16212, 16212  
 \quark\_if\_recursion\_tail\_-  
     break:nN ... 141, 789, 16212, 16218

\quark\_if\_recursion\_tail\_stop:N .  
     ..... 140, 353,  
     789, 1231, 16180, 16180, 31861,  
     31894, 32550, 32562, 32624, 32675  
 \quark\_if\_recursion\_tail\_stop:n .  
     ..... 140,  
     353, 789, 6049, 6169, 16194,  
     16194, 16210, 17165, 20075, 29676  
 \quark\_if\_recursion\_tail\_stop\_-  
     do:Nn .. 140, 353, 789, 16180, 16186  
 \quark\_if\_recursion\_tail\_stop\_-  
     do:nn ..... 140, 353,  
     789, 16194, 16201, 16211, 32175, 35308  
 \quark\_new:N ..... 139, 353,  
     354, 792, 4554, 4555, 8509, 8510,  
     10428, 10901, 10903, 10904, 12254,  
     12255, 12256, 12257, 12258, 13259,  
     13260, 13998, 16167, 16167, 16172,  
     16173, 16174, 16175, 16176, 16177,  
     16179, 17184, 17185, 18748, 19688,  
     19689, 20859, 29865, 29867, 29868  
 \q\_recursion\_stop .....  
     25, 140, 141, 362, 786, 1550, 1554,  
     6045, 6164, 16176, 17154, 20064,  
     29672, 31881, 31916, 32171, 32180,  
     32204, 32604, 32672, 32890, 35304  
 \q\_recursion\_tail .....  
     ..... 140, 141, 786, 787,  
     6045, 6163, 16176, 16182, 16188,  
     16197, 16204, 16209, 16214, 16221,  
     17154, 20064, 29672, 31880, 31915,  
     32171, 32603, 32671, 32889, 35303  
 \q\_stop ..... 25, 37, 114, 138,  
     139, 362, 786, 1549, 1552, 12836,  
     16172, 29503, 29507, 29517, 29539,  
     29559, 29691, 29723, 29735, 29739,  
     29750, 29751, 29757, 29759, 29760,  
     29762, 29765, 29779, 29786, 29828,  
     29840, 29842, 29852, 29855, 36089  
 quark internal commands:  
 \\_\_bool\_recursion\_stop .....  
     ..... 8509, 8512, 8617, 8643  
 \\_\_bool\_recursion\_tail .....  
     ..... 8509, 8617, 8643  
 \\_\_char\_no\_value ..... 18748  
 \\_\_cs\_nil ..... 3235  
 \\_\_cs\_recursion\_stop .....  
     ..... 2903, 2907, 2918, 3228  
 \\_\_file\_nil .....  
     ... 10901, 10968, 10982, 11074, 11080  
 \\_\_file\_recursion\_stop .....  
     ..... 10903, 10947, 10958  
 \\_\_file\_recursion\_tail .....  
     ..... 10903, 10947, 10951

```

\q__int_recursion_stop ..... .
.. 17184, 17882, 17899, 17942, 17969
\q__int_recursion_tail ..... .
.. 17184, 17882, 17899, 17942
\q__iow_nil ..... 10428, 10649, 10656
\q__keys_no_value ..... .
.. 956, 20842, 20859, 21449,
21473, 21490, 21515, 21532, 21561
\q__prg_recursion_stop ..... .
.. 369, 1591, 1664, 1749
\q__prg_recursion_tail ..... .
.. 369, 1664, 1674, 1749, 1768
\q__prop_recursion_stop ..... 19688
\q__prop_recursion_tail ..... 19688
\__quark_if_empty_if:n ..... .
.. 16242, 16244, 16254, 16264, 16403
\__quark_if_nil:w ..... .
.. 788, 16242, 16245, 16251
\__quark_if_no_value:w ..... .
.. 16242, 16255, 16261
\__quark_if_recursion_tail:w 787,
792, 16194, 16197, 16204, 16208, 16221
\__quark_module_name:N ..... .
.. 793, 16270, 16293, 16408, 16410
\__quark_module_name:w ..... .
.. 16408, 16412, 16415
\__quark_module_name_end:w ..... .
.. 16408, 16423, 16426
\__quark_module_name_loop:w ..... .
.. 16408, 16416, 16417, 16421
\__quark_new_conditional:Nnnn .. .
.. 16269, 16291, 16295, 16312
\__quark_new_conditional_aux_- do:Nnnn ..... .
.. 793, 16390, 16392, 16393, 16393
\__quark_new_conditional_- define:NNNNn ..... .
.. 793, 16393, 16395, 16398
\__quark_new_conditional_N:Nnnn .. .
.. 16389, 16391
\__quark_new_conditional_n:Nnnn .. .
.. 16389, 16389
\__quark_new_test:NNNn ..... .
.. 16269, 16277, 16282, 16288
\__quark_new_test_aux:Nn ..... .
.. 16270, 16271, 16281
\__quark_new_test_aux:nnNNnnnn .. .
.. 16269, 16284, 16305, 16313
\__quark_new_test_aux_do:nNNnnnnNNn .. .
.. 791, 792, 16324, 16329, 16334,
16339, 16344, 16350, 16353, 16353
\__quark_new_test_define_break_ ifx:nNNNNn ... 16351, 16366, 16387
\__quark_new_test_define_break_- tl:nNNNNn ... 16335, 16366, 16385
\__quark_new_test_define_- ifx:nNnNNn ..... 791,
792, 16340, 16345, 16366, 16375, 16388
\__quark_new_test_define_- tl:nNnNNn ..... 791,
792, 16325, 16330, 16366, 16366, 16386
\__quark_new_test_N:Nnnn 16322, 16337
\__quark_new_test_n:Nnnn 16322, 16322
\__quark_new_test_NN:Nnnn ..... .
.. 16322, 16348
\__quark_new_test_Nn:Nnnn ..... .
.. 16322, 16342
\__quark_new_test_nN:Nnnn ..... 16332
\__quark_new_test_nn:Nnnn ..... .
.. 16322, 16327
\q__quark_nil ..... 16179
\__quark_quark_conditional_- name:N ... 794, 16292, 16430, 16432
\__quark_quark_conditional_- name:w ... 794, 16430, 16434, 16437
\__quark_test_define_aux:NNNNnnNNn .. .
.. 792, 16353, 16355, 16360
\__quark_tmp:w ..... .
.. 794, 16408, 16429, 16430, 16440
\q__regex_nil ..... .
.. 4525, 4530, 4555, 4560, 5167,
5171, 5829, 5847, 5848, 5943, 5953
\q__regex_recursion_stop ..... .
.. 4554, 4557, 4559, 5829, 5848, 7831
\q__str_nil ..... .
.. 764, 13998, 15085, 15092, 15107, 15134
\q__str_recursion_stop ..... .
.. 13259, 13849, 13857, 13862
\q__str_recursion_tail ..... .
.. 718, 13259, 13492,
13501, 13518, 13538, 13560, 13849
\q__text_nil .... 29865, 30510, 30511
\q__text_recursion_stop ..... .
.. 29867, 29870, 30324, 30417, 30441,
30461, 30660, 30674, 30683, 30685,
30752, 30768, 30777, 30824, 30938,
30947, 31150, 31159, 31179, 31187,
31284, 31291, 31312, 31317, 31476,
31481, 31502, 31514, 31650, 31655,
31713, 31718, 31756, 31761, 31796,
31808, 31929, 31932, 31941, 31948,
31991, 31999, 32025, 32045, 32078,
32141, 32149, 32183, 32205, 32238,
32243, 32289, 32302, 32311, 32329,
32342, 32344, 32361, 32370, 32398
\q__text_recursion_tail ..... .
.. 29867, 30010, 30323
```

30417, 30441, 30461, 30660, 30685,  
 30751, 30824, 31929, 31948, 32025,  
 32078, 32183, 32289, 32328, 32398  
`\q__text_stop` ..... 30640, 30646, 30648, 30649  
`\q__tl_mark` ..... 685, 12254, 12361, 12363, 12365, 12367  
`\q__tl_nil` ..... 686, 12254, 12387  
`\q__tl_recursion_stop` ..... 12257  
`\q__tl_recursion_tail` ..... 12257, 13062  
`\q__tl_stop` ..... 686, 12254, 12386  
`\quitvmode` ..... 690

**R**

`\r` 30140, 32663, 32687, 32713, 32837, 32838  
`\radical` ..... 420  
`\raise` ..... 421  
`rand` ..... 257  
`randint` ..... 257  
`\randomseed` ..... 934  
`\read` ..... 422  
`\readline` ..... 558  
`\readpapersizespecial` ..... 1155  
`\ref` ..... 30159, 30169  
 regex commands:  
   `\regex_case_replace_all:nN` ... 7546  
   `\regex_case_replace_all:nNTF` .. 7546  
   `\regex_const:Nn` ..... 54, 7439, 7449  
   `\regex_count:NnN` ..... 55, 7487, 7489  
   `\regex_count:nnN` .. 55, 547, 7487, 7487  
   `\regex_extract_all:NnN` 56, 7505, 7517  
   `\regex_extract_all:nnN` ..... 47, 56, 459, 7505, 7517  
   `\regex_extract_all:NnNTF` .. 56, 7505  
   `\regex_extract_all:nnNTF` .. 56, 7505  
   `\regex_extract_once:NnN` 56, 7505, 7515  
   `\regex_extract_once:nnN` 56, 7505, 7515  
   `\regex_extract_once:NnNTF` .. 56, 7505  
   `\regex_extract_once:nnNTF` 50, 56, 7505  
   `\regex_gset:Nn` ..... 54, 7439, 7444  
   `\regex_log:N` ..... 54, 502, 7454, 7465  
   `\regex_log:n` ..... 54, 7454, 7455  
   `\regex_match:Nn` ..... 7482  
   `\regex_match:nn` ..... 7477  
   `\regex_match:NnTF` ..... 55, 7477  
   `\regex_match:nnTF` .. 55, 549, 559, 7477  
   `\regex_match_case:nn` ..... 55, 58, 481, 511, 7491, 7499, 7625  
   `\regex_match_case:nnTF` .. 55, 7491, 7491, 7500, 7501, 7502, 7503, 7504  
   `\regex_new:N` ..... 54, 462, 7433, 7433, 7435, 7436, 7437, 7438  
   `\regex_replace:nnN` ..... 183  
   `\regex_replace_all:NnN` 57, 7505, 7521

`\regex_replace_all:nnN` ..... 47, 57, 547, 7505, 7521  
`\regex_replace_all:NnNTF` .. 57, 7505  
`\regex_replace_all:nnNTF` .. 57, 7505  
`\regex_replace_case_all:nN` ..... 58, 7551, 7563  
`\regex_replace_case_all:nNTF` .. 58, 7546, 7564, 7565, 7566, 7567, 7568  
`\regex_replace_case_once:nN` ..... 58, 7523, 7528, 7540  
`\regex_replace_case_once:nNTF` .. 58, 7523, 7541, 7542, 7543, 7544, 7545  
`\regex_replace_once:NnN` 57, 7505, 7519  
`\regex_replace_once:nnN` ..... 56–58, 195, 546, 7505, 7519  
`\regex_replace_once:NnNTF` .. 57, 7505  
`\regex_replace_once:nnNTF` ..... 57, 563, 7505  
`\regex_set:Nn` .. 46, 54, 55, 7439, 7439  
`\regex_show:N` 54, 490, 502, 7454, 7464  
`\regex_show:n` .. 47, 52, 54, 7454, 7454  
`\regex_split:NnN` ..... 57, 7505, 7522  
`\regex_split:nnN` ..... 57, 7505, 7522  
`\regex_split:NnNTF` ..... 57, 7505  
`\regex_split:nnNTF` ..... 57, 7505  
`\g_tmpa_regex` ..... 59, 7435  
`\l_tmpa_regex` ..... 59, 7435  
`\g_tmpb_regex` ..... 59, 7435  
`\l_tmpb_regex` ..... 59, 7435

regex internal commands:

- `\__regex_A_test:` .. 474, 5437, 5459, 6075, 6078, 6084, 6202, 6686, 6719
- `\__regex_action_cost:n` ..... 510, 514, 6475, 6476, 6484, 6934, 6960, 6960
- `\__regex_action_free:n` .. 510, 522, 6498, 6504, 6505, 6516, 6574, 6578, 6603, 6628, 6632, 6635, 6663, 6671, 6681, 6695, 6738, 6932, 6936, 6936
- `\__regex_action_free_aux:nn` ..... 6936, 6937, 6939, 6940
- `\__regex_action_free_group:n` .. 510, 522, 6524, 6643, 6646, 6936, 6938
- `\__regex_action_start_wildcard:N` ..... 510, 6356, 6376, 6929, 6929
- `\__regex_action_submatch:nN` ..... 510, 6380, 6405, 6597, 6598, 6736, 6985, 6987, 6987
- `\__regex_action_submatch_aux:w` ..... 6987, 6989, 6992
- `\__regex_action_submatch_auxii:w` ..... 6987, 6998, 7003
- `\__regex_action_submatch_-auxiii:w` 6987, 6999, 7004, 7005, 7006

\\_\\_regex\\_action\\_submatch\\_auxiv:w ..... 6987  
 \\_\\_regex\\_action\\_success: ..... 510, 6359, 6408, 6426, 7008, 7008  
 \\_\\_regex\\_action\\_wildcard: ..... 527  
 \l\\_\\_regex\\_added\\_begin\\_int ..... 7580, 7719, 7727, 7731, 7772, 7913, 7918, 7921, 7932, 7947  
 \l\\_\\_regex\\_added\\_end\\_int ..... 7580, 7721, 7727, 7732, 7773, 7915, 7918, 7922, 7934, 7948  
 \c\\_\\_regex\\_all\\_catcodes\\_int ..... 4971, 5093, 5197, 5795  
 \c\\_\\_regex\\_ascii\\_lower\\_int ..... 4553, 4615, 4621  
 \c\\_\\_regex\\_ascii\\_max\\_control\\_int ..... 4550, 4732  
 \c\\_\\_regex\\_ascii\\_max\\_int ..... 4550, 4725, 4733, 4923  
 \c\\_\\_regex\\_ascii\\_min\\_int ..... 4550, 4724, 4731  
 \\_\\_regex\\_assertion:Nn . 474, 488, 520, 5433, 5455, 6064, 6195, 6686, 6686  
 \\_\\_regex\\_b\\_test: ..... 474, 520, 5445, 5447, 6081, 6200, 6686, 6704  
 \l\\_\\_regex\\_balance\\_int ..... 462, 534, 557, 4549, 7085, 7117, 7376, 7393, 7592, 7605, 7607, 7608, 7860, 7890, 7914, 7916  
 \g\\_\\_regex\\_balance\\_intarray ..... 459, 548, 7064, 7071, 7579, 7604  
 \g\\_\\_regex\\_balance\\_tl ..... 534, 537, 7026, 7086, 7116, 7142, 7159, 7169, 7244  
 \\_\\_regex\\_begin ..... 7570  
 \\_\\_regex\\_branch:n ..... 474, 492, 516, 4546, 5098, 5173, 5605, 5658, 5843, 5953, 5961, 6045, 6047, 6050, 6177, 6569, 6569  
 \\_\\_regex\\_break\\_point:TF ..... 463, 487, 514, 4562, 4563, 4564, 4568, 6475, 6476, 6692, 6709  
 \\_\\_regex\\_break\\_true:w ..... 463, 464, 4562, 4562, 4568, 4573, 4580, 4587, 4591, 4598, 4604, 4653, 4665, 4681, 5408, 6716, 6722, 6728  
 \\_\\_regex\\_build:N ..... 546, 6339, 6341, 7484, 7490, 7508, 7512  
 \\_\\_regex\\_build:n ..... 511, 546, 6339, 6339, 7479, 7488, 7507, 7510  
 \\_\\_regex\\_build\\_aux:NN . 558, 6339, 6342, 6346, 6348, 7969, 7988, 8058  
 \\_\\_regex\\_build\\_aux:Nn ..... 558, 6339, 6340, 6343, 7960, 7978, 8050  
 \\_\\_regex\\_build\\_for\\_cs:n ..... 4676, 6415, 6415  
 \\_\\_regex\\_build\\_new\\_state: ..... 6353, 6354, 6373, 6374, 6378, 6418, 6419, 6448, 6448, 6457, 6489, 6523, 6527, 6571, 6586, 6591, 6630, 6649, 6684, 6688, 6733  
 \l\\_\\_regex\\_build\\_tl ..... 492, 563, 4543, 5090, 5097, 5115, 5120, 5123, 5124, 5127, 5128, 5131, 5191, 5194, 5234, 5248, 5252, 5377, 5391, 5432, 5454, 5467, 5499, 5512, 5516, 5598, 5601, 5604, 5610, 5611, 5614, 5657, 5947, 5951, 5958, 5964, 5985, 6001, 6019, 6176, 6233, 6236, 6247, 6277, 6292, 6296, 6299, 6305, 7084, 7107, 7118, 7121, 7172, 7241, 7298, 7301, 7315, 7383, 8117, 8120, 8128, 8131  
 \\_\\_regex\\_build\\_transition\\_left:NNN 6444, 6444, 6632, 6646, 6663  
 \\_\\_regex\\_build\\_transition\\_right:nNn ..... 6444, 6446, 6490, 6524, 6574, 6578, 6603, 6628, 6635, 6643, 6671, 6681  
 \\_\\_regex\\_build\\_transitions\\_lazy:NNNN ..... 6455, 6455, 6497, 6503, 6515  
 \l\\_\\_regex\\_capturing\\_group\\_int ..... 459, 509, 556, 6338, 6351, 6389, 6394, 6399, 6540, 6542, 6553, 6554, 6562, 6563, 6566, 6830, 6904, 6905, 6978, 6997, 7232, 7236, 7816, 7837, 7842, 7895, 7903  
 \g\\_\\_regex\\_case\\_balance\\_tl ..... 7147, 7150, 7156, 7160, 7168  
 \\_\\_regex\\_case\\_build:n ..... 549, 6363, 6363, 6368, 7534, 7557, 7631  
 \\_\\_regex\\_case\\_build\\_aux:Nn ..... 6363, 6365, 6369  
 \\_\\_regex\\_case\\_build\\_loop:n ..... 6363, 6387, 6392  
 \l\\_\\_regex\\_case\\_changed\\_char\\_int ..... 464, 4590, 4602, 4603, 4610, 4614, 4620, 6751  
 \g\\_\\_regex\\_case\\_int ..... 545, 546, 6361, 6366, 6383, 6386, 6406, 6407, 7495, 7535, 7827  
 \l\\_\\_regex\\_case\\_max\\_group\\_int ..... 6362, 6382, 6389, 6396, 6398  
 \\_\\_regex\\_case\\_replacement:n ..... 7146, 7148, 7164, 7558  
 \\_\\_regex\\_case\\_replacement\\_aux:n ..... 7158, 7165

```

\g__regex_case_replacement_tl . . .
    ..... 7146, 7156, 7162, 7167
\c__regex_catcode_A_int . . .
\c__regex_catcode_B_int . . .
\c__regex_catcode_C_int . . .
\c__regex_catcode_D_int . . .
\c__regex_catcode_E_int . . .
\c__regex_catcode_in_class_mode_-
    int 4961, 5082, 5466, 5627, 5720, 5749
\c__regex_catcode_L_int . . .
\c__regex_catcode_M_int . . .
\c__regex_catcode_mode_int . . .
    .. 4961, 5078, 5151, 5498, 5718, 5747
\c__regex_catcode_O_int . . .
\c__regex_catcode_P_int . . .
\c__regex_catcode_S_int . . .
\c__regex_catcode_T_int . . .
\c__regex_catcode_U_int . . .
\l__regex_catcodes_bool . . .
    ..... 4968, 5754, 5758, 5793
\l__regex_catcodes_int . . .
    ..... 475, 4968, 5094, 5196,
        5198, 5204, 5485, 5502, 5602, 5615,
        5714, 5751, 5786, 5788, 5794, 5795
\__regex_char_if_alphanumeric:N 4941
\__regex_char_if_alphanumeric:NTF
    ..... 4919, 5144, 7350
\__regex_char_if_special:N . . .
4919
\__regex_char_if_special:NTF . . .
    ..... 4919, 5140
\__regex_chk_c_allowed:TF . . .
    ..... 5063, 5063, 5707
\__regex_class:NnnnN . . .
    ..... 474, 482, 483, 489,
        4547, 5192, 5493, 5494, 5500, 5860,
        5993, 6003, 6065, 6192, 6469, 6469
\c__regex_class_mode_int . . .
    ..... 4961, 5068, 5083
\__regex_class_repeat:n . . .
    ... 515, 6479, 6485, 6485, 6501, 6510
\__regex_class_repeat:nN . . .
    ..... 6480, 6494, 6494
\__regex_class_repeat:nnN . . .
    ..... 6481, 6508, 6508
\__regex_clean_assertion:Nn . . .
    ..... 6022, 6064, 6072
\__regex_clean_bool:n . . .
    .. 6022, 6022, 6074, 6089, 6093, 6101
\__regex_clean_branch:n . . .
    ..... 6022, 6050, 6053
\__regex_clean_branch_loop:n 6022,
    6055, 6058, 6063, 6085, 6094, 6102
\__regex_clean_class:n . . .
    ..... 6022, 6090, 6104, 6115, 6136
\__regex_clean_class:NnnnN . . .
    ..... 6022, 6065, 6087
\__regex_clean_class_loop:nnn . . .
    ..... 6022,
        6105, 6106, 6117, 6127, 6137, 6151
\__regex_clean_exact_cs:n . . .
    ..... 6022, 6112, 6158
\__regex_clean_exact_cs:w . . .
    ..... 6022, 6162, 6167, 6171
\__regex_clean_group:nnnN . . .
    ..... 6022, 6066, 6067, 6068, 6096
\__regex_clean_int:n . . .
    .. 6022, 6028, 6031, 6091, 6092,
        6099, 6100, 6113, 6114, 6126, 6136
\__regex_clean_int_aux:N . . .
    ..... 6022, 6032, 6034
\__regex_clean_regex:n . . .
    ..... 6022, 6042, 6098, 6111, 7469
\__regex_clean_regex_loop:w . . .
    ..... 6022, 6044, 6047, 6051
\__regex_command_K: . . .
    ... 474, 6019, 6063, 6193, 6731, 6731
\__regex_compile:n . . .
    ... 5133, 5133,
        5169, 6345, 7441, 7446, 7451, 7458
\__regex_compile:w . . .
    ..... 480, 5087, 5087, 5135, 5800
\__regex_compile_$: . . .
    ..... 5428
\__regex_compile_(: . . .
    ..... 5622
\__regex_compile_): . . .
    ..... 5661
\__regex_compile_.: . . .
    ..... 5399
\__regex_compile/_A: . . .
    ..... 5428
\__regex_compile/_B: . . .
    ..... 5428
\__regex_compile/_b: . . .
    ..... 5428
\__regex_compile/_c: . . .
    ..... 5706
\__regex_compile/_D: . . .
    ..... 5411
\__regex_compile/_d: . . .
    ..... 5411
\__regex_compile/_G: . . .
    ..... 5428
\__regex_compile/_H: . . .
    ..... 5411
\__regex_compile/_h: . . .
    ..... 5411
\__regex_compile/_K: . . .
    ..... 6016
\__regex_compile/_N: . . .
    ..... 5411
\__regex_compile/_S: . . .
    ..... 5411
\__regex_compile/_s: . . .
    ..... 5411
\__regex_compile/_u: . . .
    ..... 5880
\__regex_compile/_V: . . .
    ..... 5411
\__regex_compile/_v: . . .
    ..... 5411
\__regex_compile/_W: . . .
    ..... 5411
\__regex_compile/_w: . . .
    ..... 5411
\__regex_compile/_Z: . . .
    ..... 5428
\__regex_compile/_z: . . .
    ..... 5428
\__regex_compile_[: . . .
    ..... 5477
\__regex_compile_]: . . .
    ..... 5461
\__regex_compile_^: . . .
    ..... 5428

```

\\_regex\_compile\_abort\_tokens:n .  
  .. [5207](#), [5207](#), [5215](#), [5241](#), [5582](#), [5592](#)  
\\_regex\_compile\_anchor\_letter:NNN  
  ..... [5428](#), [5428](#),  
  [5437](#), [5439](#), [5441](#), [5443](#), [5445](#), [5447](#)  
\\_regex\_compile\_c\_[:w ..... [5743](#)  
\\_regex\_compile\_c\_C:NN ..  
  ..... [5722](#), [5731](#), [5731](#)  
\\_regex\_compile\_c\_lbrace:\_add:N ..  
  ..... [5743](#), [5769](#), [5784](#)  
\\_regex\_compile\_c\_lbrace\_end: ..  
  ..... [5743](#), [5776](#), [5780](#), [5791](#)  
\\_regex\_compile\_c\_lbrace\_-  
  loop:NN [5743](#), [5755](#), [5759](#), [5763](#), [5771](#)  
\\_regex\_compile\_c\_test:NN ..  
  ..... [5706](#), [5707](#), [5708](#)  
\\_regex\_compile\_class:NN ..  
  ..... [5507](#), [5513](#), [5517](#), [5520](#)  
\\_regex\_compile\_class:TFNN ..  
  ..... [489](#), [5492](#), [5503](#), [5507](#), [5507](#)  
\\_regex\_compile\_class\_catcode:w ..  
  ..... [5484](#), [5496](#), [5496](#)  
\\_regex\_compile\_class\_normal:w ..  
  ..... [5487](#), [5490](#), [5490](#)  
\\_regex\_compile\_class\_posix:NNNNw ..  
  ..... [5526](#), [5532](#), [5545](#)  
\\_regex\_compile\_class\_posix\_-  
  end:w ..  
  ..... [5526](#), [5563](#), [5565](#)  
\\_regex\_compile\_class\_posix\_-  
  loop:w .. [5526](#), [5551](#), [5556](#), [5559](#), [5562](#)  
\\_regex\_compile\_class\_posix\_-  
  test:w ..  
  ..... [5480](#), [5526](#), [5526](#)  
\\_regex\_compile\_cs\_aux:Nn ..  
  ..... [5815](#), [5828](#), [5841](#), [5849](#)  
\\_regex\_compile\_cs\_aux:NNnnnN ..  
  ..... [5815](#), [5846](#), [5856](#), [5869](#)  
\\_regex\_compile\_end: ..  
  ..... [480](#), [5087](#), [5100](#), [5160](#), [5824](#)  
\\_regex\_compile\_end\_cs: ..  
  ..... [5156](#), [5815](#), [5819](#), [5822](#)  
\\_regex\_compile\_escaped:N ..  
  ..... [5145](#), [5176](#), [5181](#)  
\\_regex\_compile\_group\_begin:N ..  
  .. [5596](#), [5596](#), [5644](#), [5649](#), [5667](#), [5669](#)  
\\_regex\_compile\_group\_end: ..  
  ..... [5596](#), [5607](#), [5664](#)  
\\_regex\_compile\_if\_quantifier:TFw ..  
  ..... [5216](#), [5216](#), [5944](#), [5956](#)  
\\_regex\_compile\_lparen:w [5631](#), [5635](#)  
\\_regex\_compile\_one:n ..  
  ..... [5186](#), [5186](#), [5343](#), [5349](#),  
  [5403](#), [5414](#), [5417](#), [5427](#), [5573](#), [5831](#)  
\\_regex\_compile\_quantifier:w ..  
  ..... [5205](#),  
  [5223](#), [5223](#), [5472](#), [5616](#), [5949](#), [5965](#)  
\\_regex\_compile\_quantifier\_\*:w [5257](#)  
\\_regex\_compile\_quantifier\_+:w [5257](#)  
\\_regex\_compile\_quantifier\_?:w [5257](#)  
\\_regex\_compile\_quantifier\_-  
  abort:NNN ..  
  .. [5232](#), [5237](#), [5267](#), [5286](#), [5299](#), [5322](#)  
\\_regex\_compile\_quantifier\_-  
  braced\_auxi:w .. [5263](#), [5266](#), [5269](#)  
\\_regex\_compile\_quantifier\_-  
  braced\_auxii:w .. [5263](#), [5282](#), [5291](#)  
\\_regex\_compile\_quantifier\_-  
  braced\_auxiii:w .. [5263](#), [5281](#), [5304](#)  
\\_regex\_compile\_quantifier\_-  
  lazyness:nnNN .. [483](#), [5244](#), [5244](#),  
  [5258](#), [5260](#), [5262](#), [5275](#), [5295](#), [5317](#)  
\\_regex\_compile\_quantifier\_-  
  none: .. [5228](#), [5230](#), [5232](#), [5232](#), [5239](#)  
\\_regex\_compile\_range:Nw ..  
  ..... [5341](#), [5354](#), [5370](#)  
\\_regex\_compile\_raw:N [5013](#), [5141](#),  
  [5145](#), [5147](#), [5179](#), [5184](#), [5212](#), [5334](#),  
  [5336](#), [5336](#), [5356](#), [5402](#), [5452](#), [5475](#),  
  [5523](#), [5543](#), [5561](#), [5619](#), [5624](#), [5629](#),  
  [5645](#), [5655](#), [5663](#), [5681](#), [5682](#), [5683](#),  
  [5689](#), [5700](#), [5701](#), [5702](#), [5710](#), [5765](#),  
  [5813](#), [5820](#), [5885](#), [5901](#), [5902](#), [5908](#)  
\\_regex\_compile\_raw\_error:N ..  
  ..... [5331](#), [5331](#), [5430](#), [5883](#), [6020](#)  
\\_regex\_compile\_special:N .. [476](#),  
  [5141](#), [5176](#), [5176](#), [5218](#), [5225](#), [5246](#),  
  [5273](#), [5278](#), [5293](#), [5306](#), [5340](#), [5359](#),  
  [5510](#), [5528](#), [5547](#), [5567](#), [5568](#), [5637](#),  
  [5672](#), [5690](#), [5733](#), [5752](#), [5892](#), [5911](#)  
\\_regex\_compile\_special\_group\_-  
  -w ..  
  ..... [5670](#)  
\\_regex\_compile\_special\_group\_-  
  ::w ..  
  ..... [5666](#)  
\\_regex\_compile\_special\_group\_-  
  -i:w ..  
  ..... [5670](#), [5670](#)  
\\_regex\_compile\_special\_group\_-  
  |:w ..  
  ..... [5666](#)  
\\_regex\_compile\_u\_brace:NNN ..  
  ..... [5886](#), [5887](#), [5890](#), [5890](#)  
\\_regex\_compile\_u\_end: ..  
  ..... [5887](#), [5954](#), [5954](#)  
\\_regex\_compile\_u\_in\_cs: ..  
  ..... [5975](#), [5978](#), [5978](#)  
\\_regex\_compile\_u\_in\_cs\_aux:n ..  
  ..... [5988](#), [5991](#)  
\\_regex\_compile\_u\_loop>NN ..  
  ..... [5896](#), [5906](#), [5906](#), [5909](#), [5921](#)

\\_\\_regex\\_compile\_u\_not\_cs: .....  
     ..... 5973, 5997, 5997  
 \\_\\_regex\\_compile\_u\_payload: .....  
     ..... 500, 5954, 5963, 5967, 5969  
 \\_\\_regex\\_compile\_ur:n .....  
     ..... 500, 5932, 5939, 5941  
 \\_\\_regex\\_compile\_ur\_aux:w .....  
     ..... 5932, 5943, 5953  
 \\_\\_regex\\_compile\_ur\_end: .....  
     ..... 5886, 5900, 5932, 5932  
 \\_\\_regex\\_compile\_use:n .....  
     ..... 5162, 5162, 6395  
 \\_\\_regex\\_compile\_use\_aux:w 5166, 5171  
 \\_\\_regex\\_compile\_|: ..... 5653  
 \\_\\_regex\\_compute\_case\_changed\_-  
     char: ..... 4608, 4608, 4626, 6875  
 \\_\\_regex\\_count:nnN .....  
     ..... 7488, 7490, 7637, 7637  
 \c\\_regex\\_cs\_in\_class\_mode\_int ..  
     ..... 4961, 5806  
 \c\\_regex\\_cs\_mode\_int ... 4961, 5804  
 \l\\_regex\\_curr\\_analysis\_tl ....  
     524, 6765, 6811, 6839, 6846, 6880, 6881  
 \l\\_regex\\_curr\\_catcode\\_int ....  
     .. 4632, 4651, 4659, 4671, 6751, 6878  
 \l\\_regex\\_curr\\_char\\_int ....  
     ..... 526, 4572, 4578, 4579,  
         4586, 4596, 4597, 4610, 4611, 4612,  
         4613, 4619, 4652, 5407, 6425, 6707,  
         6715, 6751, 6835, 6874, 6877, 6893  
 \\_\\_regex\\_curr\\_cs\\_to\\_str: .....  
     ..... 4504, 4504, 4662, 4679  
 \l\\_regex\\_curr\\_pos\\_int .....  
     .. 460, 526, 6727, 6746, 6822, 6833,  
         6873, 7007, 7015, 7593, 7598, 7602,  
         7603, 7605, 8102, 8107, 8111, 8112  
 \l\\_regex\\_curr\\_state\\_int ... 523,  
     529, 6757, 6911, 6912, 6914, 6919,  
     6922, 6944, 6949, 6954, 6955, 6963  
 \l\\_regex\\_curr\\_submatches\_tl ...  
     ..... 6758, 6829, 6924,  
         6956, 6957, 6968, 6990, 6994, 7019  
 \l\\_regex\\_curr\\_token\_tl .....  
     ..... 4507, 6751, 6876  
 \l\\_regex\\_default\\_catcodes\\_int ...  
     ..... 475, 4968,  
         5092, 5094, 5204, 5502, 5602, 5615  
 \\_\\_regex\\_disable\\_submatches: 4675,  
     5801, 6982, 6982, 7614, 7640, 7999  
 \l\\_regex\\_empty\\_success\\_bool ...  
     ..... 6768, 6814, 6818, 7013, 7701  
 \\_\\_regex\\_end ..... 7570  
 \\_\\_regex\\_escape\_\w: ..... 4798  
 \\_\\_regex\\_escape\_/\scan\_stop::w 4798

\\_\\_regex\\_escape\_/a:w ..... 4798  
 \\_\\_regex\\_escape/\_e:w ..... 4798  
 \\_\\_regex\\_escape/\_f:w ..... 4798  
 \\_\\_regex\\_escape/\_n:w ..... 4798  
 \\_\\_regex\\_escape/\_r:w ..... 4798  
 \\_\\_regex\\_escape/\_t:w ..... 4798  
 \\_\\_regex\\_escape/\_x:w ..... 4817  
 \\_\\_regex\\_escape\_\w ..... 4782  
 \\_\\_regex\\_escape\_`\scan\_stop::w . 4798  
 \\_\\_regex\\_escape\\_escaped:N .....  
     ..... 4768, 4792, 4795, 4796  
 \\_\\_regex\\_escape\\_loop:N ..... 469,  
     4775, 4782, 4782, 4786, 4789, 4793,  
     4817, 4856, 4867, 4868, 4888, 4897  
 \\_\\_regex\\_escape\\_raw:N .....  
     .... 470, 4769, 4795, 4797, 4806,  
     4808, 4810, 4812, 4814, 4816, 4830  
 \\_\\_regex\\_escape\\_unescape:N ....  
     ..... 4767, 4785, 4795, 4795  
 \\_\\_regex\\_escape\\_use:nnnn .....  
     .... 468, 480, 4763, 4763, 5138, 7087  
 \\_\\_regex\\_escape\_x:N .....  
     ..... 470, 4855, 4859, 4859  
 \\_\\_regex\\_escape\_x\\_end:w .....  
     ..... 470, 4817, 4819, 4822  
 \\_\\_regex\\_escape\_x\\_large:n .... 4817  
 \\_\\_regex\\_escape\_x\\_loop:N .....  
     ... 470, 4852, 4871, 4871, 4880, 4883  
 \\_\\_regex\\_escape\_x\\_loop\\_error: . 4871  
 \\_\\_regex\\_escape\_x\\_loop\\_error:n ..  
     ..... 4877, 4889, 4894  
 \\_\\_regex\\_escape\_x\\_test:N .....  
     .... 470, 4820, 4834, 4834, 4842  
 \\_\\_regex\\_escape\_x\\_testii:N .....  
     ..... 4834, 4844, 4849  
 \l\\_regex\\_every\\_match\_tl .....  
     ..... 6767, 6850, 6860, 6897  
 \\_\\_regex\\_extract: .....  
     ..... 550, 561, 7655, 7662,  
     7675, 7812, 7812, 7857, 7885, 8074  
 \\_\\_regex\\_extract\\_all:nnN .....  
     ..... 7516, 7649, 7659  
 \\_\\_regex\\_extract\\_aux:w .....  
     ..... 7812, 7829, 7834, 7845  
 \\_\\_regex\\_extract\\_check:n .....  
     ..... 7763, 7765, 7768  
 \\_\\_regex\\_extract\\_check:w .....  
     .... 552, 553, 7723, 7763, 7763, 7774  
 \\_\\_regex\\_extract\\_check\\_end:w ...  
     ..... 554, 7763, 7779, 7791  
 \\_\\_regex\\_extract\\_check\\_loop:w ...  
     .... 7763, 7777, 7784, 7789, 7792  
 \\_\\_regex\\_extract\\_map:N .....  
     ..... 7737, 7799, 7799

\\_\\_regex\\_extract\\_map\\_aux:NNn . . .  
     . . . . . 7799, 7801, 7805  
 \\_\\_regex\\_extract\\_map\\_loop:w . . .  
     . . . . . 554, 7735, 7799, 7807, 7810  
 \\_\\_regex\\_extract\\_once:nnN . . . . .  
     . . . . . 7514, 7649, 7649  
 \\_\\_regex\\_extract\\_seq\\_aux:n . . . . .  
     . . . . . 7716, 7739, 7739  
 \\_\\_regex\\_extract\\_seq\\_aux:ww . . . . .  
     . . . . . 7739, 7742, 7745  
 \l\\_\\_regex\\_fresh\\_thread\\_bool . . . . .  
     . . . . . 524, 529, 6737,  
     6743, 6768, 6891, 6931, 6933, 7014  
 \\_\\_regex\\_G\\_test: . . . . .  
     . . . . . 474, 5439, 6079, 6203, 6686, 6725  
 \\_\\_regex\\_get\\_digits:Ntfw . . . . .  
     . . . . . 4999, 4999, 5265, 5280  
 \\_\\_regex\\_get\\_digits\\_loop:nw . . . . .  
     . . . . . 5002, 5005, 5008  
 \\_\\_regex\\_get\\_digits\\_loop:w . . . . . 4999  
 \\_\\_regex\\_group:nnnN . . . . .  
     . . . . . 474, 492, 5644, 5649,  
     5935, 6066, 6186, 6357, 6537, 6537  
 \\_\\_regex\\_group\\_aux:nnnnN . . . . .  
     . . . . . 516, 6520, 6520, 6539, 6547, 6550  
 \\_\\_regex\\_group\\_aux:nnnnnN . . . . . 516  
 \\_\\_regex\\_group\\_end\\_extract\\_seq:N . . . . .  
     . . . . . 553, 7657, 7666, 7706, 7708, 7708  
 \\_\\_regex\\_group\\_end\\_replace:N . . . . .  
     . . . . . 7876, 7909, 7911, 7911  
 \\_\\_regex\\_group\\_end\\_replace\_-  
     check:n . . . . . 557, 7911, 7940, 7943  
 \\_\\_regex\\_group\\_end\\_replace\_-  
     check:w . . . . . 557, 7911, 7929, 7938  
 \\_\\_regex\\_group\\_end\\_replace\\_try: . . . . .  
     . . . . . 557, 7911, 7917, 7927, 7949  
 \l\\_\\_regex\\_group\\_level\\_int . . . . . 4960,  
     5091, 5109, 5111, 5113, 5603, 5609  
 \\_\\_regex\\_group\\_no\\_capture:nnnN . . . . .  
     . . . . . 474, 5667, 5935, 5936,  
     5948, 5960, 6067, 6188, 6537, 6546  
 \\_\\_regex\\_group\\_repeat:nn . . . . .  
     . . . . . 6532, 6581, 6581  
 \\_\\_regex\\_group\\_repeat:nnN . . . . .  
     . . . . . 6533, 6621, 6621  
 \\_\\_regex\\_group\\_repeat:nnnN . . . . .  
     . . . . . 6534, 6652, 6652  
 \\_\\_regex\\_group\\_repeat\\_aux:n . . . . .  
     . . . . . 517, 519, 6588, 6601, 6601, 6639, 6656  
 \\_\\_regex\\_group\\_resetting:nnnN . . . . .  
     . . . . . 474, 5669, 5936, 6068, 6190, 6548, 6548  
 \\_\\_regex\\_group\\_resetting\_-  
     loop:nnNn . . . . . 6548, 6552, 6560, 6565  
 \\_\\_regex\\_group\\_submatches:nNN . . . . .  
     . . . . . 6589, 6594, 6594, 6624, 6640, 6654  
 \\_\\_regex\\_hexadecimal\\_use:N . . . . . 4899  
 \\_\\_regex\\_hexadecimal\\_use:NTF . . . . .  
     . . . . . 4854, 4866, 4879, 4899  
 \\_\\_regex\\_if\\_end\\_range:NN . . . . . 5354  
 \\_\\_regex\\_if\\_end\\_range:NNTF 5354, 5372  
 \\_\\_regex\\_if\\_in\\_class:TF 5023, 5023,  
     5102, 5189, 5205, 5338, 5401, 5463,  
     5479, 5624, 5655, 5663, 8191, 8204  
 \\_\\_regex\\_if\\_in\\_class\\_or\\_catcode:TF . . . . .  
     . . . . . 5043, 5043, 5430, 5452, 5882  
 \\_\\_regex\\_if\\_in\\_cs:TF . . . . .  
     . . . . . 5031, 5031, 5811, 5818, 8189, 8198  
 \\_\\_regex\\_if\\_match:nn . . . . .  
     . . . . . 7479, 7484, 7611, 7611, 7630  
 \\_\\_regex\\_if\\_raw\\_digit:NN . . . . . 5011  
 \\_\\_regex\\_if\\_raw\\_digit:NNTF . . . . .  
     . . . . . 5001, 5007, 5011  
 \\_\\_regex\\_if\\_two\\_empty\\_matches:TF . . . . .  
     . . . . . 524, 6768, 6770, 6819, 6825, 7010  
 \\_\\_regex\\_if\\_within\\_catcode:TF . . . . .  
     . . . . . 5055, 5055, 5482  
 \\_\\_regex\\_input\\_item:n . . . . .  
     . . . . . 558, 562, 563, 7955,  
     7956, 8016, 8038, 8079, 8103, 8112  
 \l\\_\\_regex\\_input\\_tl . . . . .  
     . . . . . 559, 560, 562, 7955,  
     8011, 8015, 8037, 8039, 8101, 8105  
 \\_\\_regex\\_int\\_eval:w . . . . . 4472, 4472,  
     6976, 7040, 7041, 7052, 7839, 7842  
 \\_\\_regex\\_intarray\\_item:NnTF . . . . .  
     . . . . . 4509, 4509, 7064, 7071  
 \\_\\_regex\\_intarray\\_item\\_aux:nNtf . . . . .  
     . . . . . 4509, 4510, 4511  
 \l\\_\\_regex\\_internal\\_a\\_int . . . . . 484, 538,  
     4535, 5265, 5276, 5287, 5296, 5300,  
     5308, 5311, 5315, 5318, 5325, 6502,  
     6505, 6511, 6516, 6590, 6605, 6611,  
     6617, 6626, 6629, 6633, 6636, 6641,  
     6644, 6647, 6662, 6670, 6679, 7251,  
     7272, 7828, 7837, 7839, 7842, 7844  
 \l\\_\\_regex\\_internal\\_a\\_tl . . . . .  
     468, 500, 501, 505, 557, 4535, 4661,  
     4664, 4766, 4773, 4780, 5550, 5555,  
     5571, 5576, 5581, 5585, 5591, 5592,  
     5826, 5837, 5895, 5939, 5971, 5983,  
     5999, 6180, 6183, 6236, 6257, 6299,  
     6306, 6401, 6402, 6439, 6440, 6441,  
     6442, 6572, 6573, 6577, 6579, 6836,  
     6839, 7462, 7474, 7865, 7899, 7933  
 \l\\_\\_regex\\_internal\\_b\\_int . . . . .  
     . . . . . 4535, 5280,

5309, 5312, 5313, 5315, 5319, 5326,  
 6606, 6611, 6616, 6662, 6670, 6679  
`\l__regex_internal_b_tl` .....  
 ..... 4535, 5894, 5914, 5927  
`\l__regex_internal_bool` .....  
 ..... 4535, 5549, 5554, 5575, 5584  
`\l__regex_internal_c_int` .....  
 ..... 4535, 6608, 6613, 6614, 6618  
`\l__regex_internal_regex` .....  
 .... 479, 4984, 5131, 5169, 5828,  
 5834, 6346, 7442, 7447, 7452, 7459  
`\l__regex_internal_seq` 4535, 6312,  
 6313, 6318, 6325, 6326, 6327, 6329  
`\g__regex_internal_t1` .....  
 ..... 552–554, 4535,  
 4771, 4775, 5980, 5987, 7713, 7724,  
 7725, 7775, 7778, 7803, 7925, 7930  
`\_regex_item_caseful_equal:n` ...  
 ..... 474, 4570,  
 4570, 4692, 4693, 4697, 4698, 4699,  
 4700, 4701, 4710, 4715, 4733, 4751,  
 5095, 5694, 5862, 5994, 6113, 6204  
`\_regex_item_caseful_range:nn` ...  
 ..... 474,  
 4570, 4576, 4689, 4704, 4707, 4708,  
 4709, 4723, 4730, 4737, 4739, 4741,  
 4744, 4745, 4746, 4747, 4752, 4755,  
 4760, 4761, 5096, 5696, 6121, 6206  
`\_regex_item_caseless_equal:n` ...  
 ... 474, 4584, 4584, 5675, 6114, 6211  
`\_regex_item_caseless_range:nn` ...  
 ... 474, 4584, 4594, 5677, 6122, 6213  
`\_regex_item_catcode`: .....  
 ..... 4629, 4629, 4641  
`\_regex_item_catcode:nTF` .....  
 ..... 474, 489, 4629,  
 4639, 4648, 5198, 5504, 6132, 6218  
`\_regex_item_catcode_reverse:nTF` ...  
 ... 474, 4629, 4647, 5505, 6133, 6220  
`\_regex_item_cs:n` .....  
 ... 474, 4669, 4669, 5834, 6111, 6227  
`\_regex_item_equal:n` .....  
 ..... 4627, 4627, 5095, 5344,  
 5350, 5380, 5393, 5394, 5674, 5693  
`\_regex_item_exact:nn` .....  
 ... 474, 501, 4649, 4649, 6009, 6123, 6224  
`\_regex_item_exact_cs:n` ... 474,  
 497, 4649, 4657, 5836, 6006, 6112, 6226  
`\_regex_item_range:nn` .....  
 .. 4627, 4628, 5096, 5382, 5676, 5695  
`\_regex_item_reverse:n` .....  
 .... 474, 490, 4565, 4565, 4648,  
 4714, 5418, 5575, 6115, 6222, 6710  
`\l__regex_last_char_int` .....  
 ..... 6707, 6721, 6751, 6874, 7016  
`\l__regex_last_char_success_int` .  
 ..... 6751, 6809, 6835, 7016  
`\l__regex_left_state_int` .. 6334,  
 6355, 6375, 6379, 6433, 6440, 6451,  
 6458, 6461, 6462, 6464, 6465, 6491,  
 6499, 6502, 6525, 6573, 6575, 6585,  
 6605, 6625, 6627, 6655, 6658, 6661,  
 6664, 6676, 6689, 6698, 6734, 6741  
`\l__regex_left_state_seq` .....  
 ..... 6334, 6432, 6439, 6572  
`\_regex_maplike_break:` .....  
 .... 461, 559, 4520, 4520, 4521,  
 6781, 6795, 6841, 6855, 6863, 8019  
`\_regex_match:n` . 6774, 6774, 7617,  
 7644, 7654, 7664, 7690, 7854, 7887  
`\_regex_match_case:nnTF` .....  
 ..... 7493, 7620, 7620  
`\_regex_match_case_aux:nn` 7620, 7636  
`\l__regex_match_count_int` .....  
 .... 547, 550, 7569, 7641, 7642, 7647  
`\_regex_match_cs:n` . 4679, 6774, 6783  
`\_regex_match_init`: .....  
 ..... 6774, 6776, 6786, 6797, 8010  
`\_regex_match_once_init`: .....  
 ... 6777, 6787, 6816, 6816, 6867, 8012  
`\_regex_match_once_init_aux`: ...  
 ..... 6837, 6843  
`\_regex_match_one_active:n` ...  
 ..... 6870, 6888, 6899  
`\_regex_match_one_token:nnN` ...  
 . 526, 529, 559, 6779, 6780, 6791,  
 6792, 6794, 6840, 6870, 6870, 8017  
`\l__regex_match_success_bool` ...  
 ... 524, 6771, 6828, 6854, 6862, 7012  
`\l__regex_matched_analysis_t1` ...  
 524, 6765, 6810, 6836, 6845, 6879, 7017  
`\l__regex_max_pos_int` .....  
 ..... 533, 6746, 7598,  
 7696, 7702, 7874, 7907, 8093, 8107  
`\l__regex_max_state_int` .....  
 . 509, 512, 571, 6331, 6352, 6372,  
 6410, 6412, 6413, 6417, 6450, 6452,  
 6453, 6512, 6584, 6604, 6606, 6614,  
 6658, 6664, 6672, 6682, 6801, 8463  
`\l__regex_max_thread_int` .....  
 ..... 6761, 6785,  
 6831, 6884, 6887, 6892, 6969, 6977  
`\_regex_maybe_compute_ccc`: ...  
 .... 4589, 4601, 4624, 4626, 6875  
`\l__regex_min_pos_int` .....  
 ..... 533, 6746, 6807, 6808

\l\_\_regex\_min\_state\_int 512, 6331,  
6352, 6372, 6417, 6801, 6832, 8462  
\l\_\_regex\_min\_submatch\_int .....  
..... 548, 552,  
556, 6812, 6813, 7572, 7715, 7894, 7902  
\l\_\_regex\_min\_thread\_int .....  
.. 6761, 6785, 6831, 6884, 6886, 6892  
\l\_\_regex\_mode\_int ..... 4961,  
5025, 5033, 5036, 5045, 5048, 5057,  
5065, 5068, 5078, 5079, 5081, 5083,  
5137, 5151, 5153, 5465, 5469, 5470,  
5471, 5498, 5509, 5626, 5716, 5717,  
5745, 5746, 5802, 5803, 5972, 6018  
\\_\_regex\_mode\_quit\_c: .....  
..... 5076, 5076, 5188, 5599  
\\_\_regex\_msg\_repeated:nnN .....  
..... 6272, 6293, 6303, 8432, 8432  
\\_\_regex\_multi\_match:n .....  
524, 6848, 6858, 7642, 7662, 7671, 7885  
\c\_\_regex\_no\_match\_regex .....  
..... 4544, 4984, 7434  
\c\_\_regex\_outer\_mode\_int .....  
..... 4961, 5036, 5048, 5057,  
5065, 5079, 5137, 5153, 5972, 6018  
\\_\_regex\_peek:nnTF .....  
561, 7959, 7968, 7977, 7987, 7995, 7995  
\\_\_regex\_peek\_aux:nnTF .....  
..... 7995, 7997, 8003, 8068  
\\_\_regex\_peek\_end: .....  
.... 558, 560, 7961, 7970, 8023, 8023  
\l\_\_regex\_peek\_false\_t1 .....  
..... 7952, 8007, 8027, 8033, 8097  
\\_\_regex\_peek\_reinsert:N ... 560,  
561, 8026, 8027, 8033, 8035, 8035, 8097  
\\_\_regex\_peek\_remove\_end:n .....  
.... 558, 560, 7979, 7989, 8023, 8029  
\\_\_regex\_peek\_replace:nnTF .....  
..... 8050, 8058, 8065, 8065  
\\_\_regex\_peek\_replace\_end: .....  
..... 8068, 8070, 8070  
\\_\_regex\_peek\_replacement\_put:n ..  
..... 8076, 8114, 8114  
\\_\_regex\_peek\_replacement\_put\_-  
submatch\_aux:n .. 8078, 8125, 8125  
\\_\_regex\_peek\_replacement\_-  
token:n .... 563, 8080, 8123, 8123  
\\_\_regex\_peek\_replacement\_var:N ..  
..... 8081, 8135, 8135  
\l\_\_regex\_peek\_true\_t1 .....  
560, 562, 7952, 8006, 8026, 8032, 8085  
\\_\_regex\_pop\_lr\_states: .....  
..... 6390, 6422, 6430, 6437, 6530  
\\_\_regex\_posix\_alnum: ... 4717, 4717  
\\_\_regex\_posix\_alpha: .....  
..... 504, 4717, 4718, 4719  
\\_\_regex\_posix\_ascii: ... 4717, 4721  
\\_\_regex\_posix\_blank: ... 4717, 4727  
\\_\_regex\_posix\_cntrl: ... 4717, 4728  
\\_\_regex\_posix\_digit: .....  
..... 4717, 4718, 4735, 4759  
\\_\_regex\_posix\_graph: ... 4717, 4736  
\\_\_regex\_posix\_lower: 4717, 4720, 4738  
\\_\_regex\_posix\_print: ... 4717, 4740  
\\_\_regex\_posix\_punct: ... 4717, 4742  
\\_\_regex\_posix\_space: ... 4717, 4749  
\\_\_regex\_posix\_upper: 4717, 4720, 4754  
\\_\_regex\_posix\_word: ... 4717, 4756  
\\_\_regex\_posix\_xdigit: ... 4717, 4757  
\\_\_regex\_prop\_: ..... 487, 5399  
\\_\_regex\_prop\_d: .....  
..... 487, 504, 4688, 4688, 4735  
\\_\_regex\_prop\_h: ... 4688, 4690, 4727  
\\_\_regex\_prop\_N: ... 4688, 4712, 5427  
\\_\_regex\_prop\_s: ..... 4688, 4695  
\\_\_regex\_prop\_v: ..... 4688, 4703  
\\_\_regex\_prop\_w: .....  
... 4688, 4705, 4756, 6708, 6710, 6711  
\\_\_regex\_push\_lr\_states: .....  
..... 6381, 6420, 6430, 6430, 6528  
\\_\_regex\_quark\_if\_nil:N ..... 4561  
\\_\_regex\_quark\_if\_nil:NTF 5852, 5872  
\\_\_regex\_quark\_if\_nil:nTF .... 4561  
\\_\_regex\_quark\_if\_nil\_p:n .... 4561  
\\_\_regex\_query\_range:nn .....  
..... 533, 562, 7031, 7037,  
7037, 7056, 7128, 7869, 7906, 8088  
\\_\_regex\_query\_range\_loop:ww ...  
..... 7037, 7039, 7044, 7051  
\\_\_regex\_query\_set:n ..... 7590,  
7590, 7656, 7665, 7691, 7858, 7888  
\\_\_regex\_query\_set\_aux:nn .....  
..... 7590, 7594, 7596, 7597, 7600  
\\_\_regex\_query\_set\_from\_input\_-  
tl: ..... 8075, 8099, 8099  
\\_\_regex\_query\_set\_item:n .....  
..... 8099, 8103, 8104, 8106, 8109  
\\_\_regex\_query\_submatch:n .....  
... 7054, 7054, 7242, 7754, 8129, 8132  
\\_\_regex\_reinsert\_item:n .....  
560, 562, 8035, 8038, 8041, 8079, 8118  
\\_\_regex\_replace\_all:nnN .....  
..... 7520, 7880, 7880  
\\_\_regex\_replace\_all\_aux:nnN ...  
..... 7556, 7881, 7882  
\\_\_regex\_replace\_once:nnN .....  
..... 7518, 7847, 7847

\\_\\_regex\\_replace\\_once\\_aux:nN . . . . .  
 ..... 7533, 7847, 7848, 7849  
 \\_\\_regex\\_replacement:n . 561, 7079,  
 7079, 7123, 7535, 7848, 7881, 8082  
 \\_\\_regex\\_replacement\\_apply:Nn . . . . .  
 ..... 7079, 7080, 7081, 7158  
 \\_\\_regex\\_replacement\\_balance\_-  
 one\\_match:n . . . . .  
 ... 532, 7027, 7027, 7140, 7862, 7897  
 \\_\\_regex\\_replacement\_c:w . 7281, 7281  
 \\_\\_regex\\_replacement\_c\_A:w . . . . .  
 ..... 536, 7213, 7369, 7370  
 \\_\\_regex\\_replacement\_c\_B:w . . . . .  
 ..... 7201, 7372, 7373  
 \\_\\_regex\\_replacement\_c\_C:w 7381, 7381  
 \\_\\_regex\\_replacement\_c\_D:w . . . . .  
 ..... 7208, 7386, 7387  
 \\_\\_regex\\_replacement\_c\_E:w . . . . .  
 ..... 7202, 7389, 7390  
 \\_\\_regex\\_replacement\_c\_L:w . . . . .  
 ..... 7211, 7398, 7399  
 \\_\\_regex\\_replacement\_c\_M:w . . . . .  
 ..... 7203, 7401, 7402  
 \\_\\_regex\\_replacement\_c\_O:w 7200,  
 7205, 7209, 7212, 7214, 7404, 7405  
 \\_\\_regex\\_replacement\_c\_P:w . . . . .  
 ..... 7206, 7407, 7408  
 \\_\\_regex\\_replacement\_c\_S:w . . . . .  
 ..... 7196, 7210, 7413, 7413  
 \\_\\_regex\\_replacement\_c\_T:w . . . . .  
 ..... 7204, 7421, 7422  
 \\_\\_regex\\_replacement\_c\_U:w . . . . .  
 ..... 7207, 7424, 7425  
 \\_\\_regex\\_replacement\_cat:NNN . . . . .  
 ..... 7286, 7329, 7329  
 \l\\_\\_regex\\_replacement\\_category\_-  
 seq 7024, 7110, 7113, 7114, 7183, 7343  
 \l\\_\\_regex\\_replacement\\_category\_-  
 tl . . . . . 536,  
 7024, 7178, 7184, 7187, 7344, 7345  
 \\_\\_regex\\_replacement\\_char:nNN . . . . .  
 ..... 543,  
 7364, 7364, 7371, 7378, 7388, 7395,  
 7400, 7403, 7406, 7410, 7423, 7426  
 \l\\_\\_regex\\_replacement\\_csnames\_-  
 int 531, 7023, 7104, 7106, 7108,  
 7175, 7243, 7297, 7304, 7314, 7316,  
 7323, 7334, 7375, 7392, 8116, 8127  
 \\_\\_regex\\_replacement\_cu\_aux:Nw . . . . .  
 ..... 7291, 7295, 7295, 7309  
 \\_\\_regex\\_replacement\\_do\\_one\_-  
 match:n . . . . . 561,  
 563, 7029, 7029, 7126, 7867, 7905, 8086  
 \\_\\_regex\\_replacement\_error:NNN . . . . .  
 ..... 7252, 7264,  
 7275, 7287, 7292, 7310, 7428, 7428  
 \\_\\_regex\\_replacement\\_escaped:N . . . . .  
 ..... 7100, 7219, 7219, 7348  
 \\_\\_regex\\_replacement\\_exp\\_not:N . . . . .  
 ... 539, 7035, 7035, 7291, 7384, 8080  
 \\_\\_regex\\_replacement\\_exp\\_not:n . . . . .  
 ..... 7036, 7036, 7309, 8081  
 \\_\\_regex\\_replacement\_g:w . 7248, 7248  
 \\_\\_regex\\_replacement\_g\_digits>NN  
 ..... 7248, 7251, 7254, 7261  
 \\_\\_regex\\_replacement\_lbrace:N . . . . .  
 .. 7093, 7250, 7290, 7308, 7321, 7321  
 \\_\\_regex\\_replacement\\_normal:n . . . . .  
 ..... 7095, 7101, 7173, 7173,  
 7226, 7256, 7283, 7318, 7326, 7341  
 \\_\\_regex\\_replacement\\_normal\_-  
 aux:N . . . . . 7173, 7179, 7193  
 \\_\\_regex\\_replacement\\_put:n . . . . .  
 .. 7171, 7171, 7176, 7367, 7419, 8076  
 \\_\\_regex\\_replacement\\_put\_-  
 submatch:n . 7224, 7230, 7230, 7271  
 \\_\\_regex\\_replacement\\_put\_-  
 submatch\\_aux:n . . . . .  
 ..... 7230, 7233, 7239, 8077  
 \\_\\_regex\\_replacement\_rbrace:N . . . . .  
 ..... 7090, 7270, 7312, 7312  
 \\_\\_regex\\_replacement\\_set:n . . . . .  
 ..... 7079, 7080, 7124, 7161  
 \l\\_\\_regex\\_replacement\\_tl . . . . .  
 ..... 7954, 8067, 8082  
 \\_\\_regex\\_replacement\_u:w . 7306, 7306  
 \\_\\_regex\\_return: . . . . .  
 546, 7480, 7485, 7510, 7512, 7582, 7582  
 \l\\_\\_regex\\_right\\_state\\_int . . . . .  
 ..... 6334, 6358, 6402, 6403,  
 6423, 6435, 6442, 6451, 6452, 6491,  
 6498, 6504, 6517, 6525, 6575, 6579,  
 6590, 6604, 6613, 6625, 6629, 6633,  
 6636, 6641, 6644, 6647, 6655, 6669,  
 6672, 6675, 6678, 6682, 6698, 6741  
 \l\\_\\_regex\\_right\\_state\\_seq . . . . .  
 .. 6334, 6401, 6411, 6434, 6441, 6577  
 \l\\_\\_regex\\_saved\\_success\\_bool . . . . .  
 ..... 524, 4677, 4684, 6771  
 \\_\\_regex\\_show:N . . . . .  
 ..... 544, 6173, 6173, 7459, 7471  
 \\_\\_regex\\_show>NN 7454, 7464, 7465, 7466  
 \\_\\_regex\\_show:Nn 7454, 7454, 7455, 7456  
 \\_\\_regex\\_show\\_char:n . . . . . 6205,  
 6209, 6212, 6216, 6225, 6238, 6238  
 \\_\\_regex\\_show\\_class:NnnnN . . . . .  
 ..... 6192, 6274, 6274

\\_\_regex\_show\_group\_aux:nnnn . . . . .  
     ..... 6187, 6189, 6191, 6265, 6265  
 \\_\_regex\_show\_item\_catcode:NnTF . . . . .  
     ..... 6219, 6221, 6310, 6310  
 \\_\_regex\_show\_item\_exact\_cs:n . . . . .  
     ..... 6226, 6323, 6323  
 \l\_\_regex\_show\_lines\_int . . . . .  
     ..... 4986, 6246, 6278, 6281, 6288  
 \\_\_regex\_show\_one:n . . . . .  
     ... 6181, 6194, 6197, 6205, 6208,  
         6212, 6215, 6225, 6229, 6244, 6244,  
         6260, 6267, 6271, 6284, 6300, 6328  
 \\_\_regex\_show\_pop: . . . . .  
     ..... 6254, 6256, 6263, 6270  
 \l\_\_regex\_show\_prefix\_seq . 4985,  
     6179, 6182, 6230, 6250, 6255, 6257  
 \\_\_regex\_show\_push:n . . . . .  
     .. 6231, 6254, 6254, 6261, 6268, 6279  
 \\_\_regex\_show\_scope:nn . . . . .  
     ..... 6223, 6228, 6254, 6258, 6315  
 \\_\_regex\_single\_match: . 524, 4674,  
     6848, 6848, 7615, 7652, 7852, 8008  
 \\_\_regex\_split:nnN . 7522, 7668, 7668  
 \\_\_regex\_standard\_escapechar: . . . . .  
     .. 4473, 4473, 4770, 5136, 6350, 6371  
 \l\_\_regex\_start\_pos\_int . . . . .  
     ... 6727, 6746, 6822, 6827, 6834,  
         7674, 7686, 7699, 7702, 7825, 7907  
 \g\_\_regex\_state\_active\_intarray . . . . .  
     ..... 459, 512, 523-  
         525, 6763, 6804, 6910, 6913, 6921, 6948  
 \l\_\_regex\_step\_int 459, 6760, 6806,  
     6872, 6911, 6915, 6923, 6937, 6939  
 \\_\_regex\_store\_state:n . . . . .  
     ..... 523, 6832, 6962, 6965, 6965  
 \\_\_regex\_store\_submatches: . . . . . 6965  
 \\_\_regex\_store\_submatches:n . . . . . 6984  
 \\_\_regex\_store\_submatches:nn . . . . .  
     ..... 6967, 6971  
 \\_\_regex\_submatch\_balance:n . . . . .  
     .. 7028, 7060, 7060, 7143, 7245, 7743  
 \g\_\_regex\_submatch\_begin\_-  
     intarray . . . . .  
         . 459, 532, 555, 7033, 7057, 7074,  
             7135, 7575, 7681, 7684, 7697, 7838  
 \g\_\_regex\_submatch\_case\_intarray . . . . .  
     ..... 7154, 7575, 7820, 7826  
 \g\_\_regex\_submatch\_end\_intarray . . . . .  
     ..... 459, 555, 7058, 7067,  
         7575, 7678, 7694, 7841, 7871, 8090  
 \l\_\_regex\_submatch\_int . . . . .  
     459, 548, 551, 552, 556, 6813, 7572,  
         7693, 7695, 7698, 7700, 7703, 7716,  
         7815, 7819, 7821, 7822, 7896, 7904  
 \g\_\_regex\_submatch\_prev\_intarray  
     ..... 459, 548, 554, 7032,  
         7131, 7575, 7676, 7692, 7818, 7824  
 \g\_\_regex\_success\_bool . . . . .  
     .... 524, 4678, 4680, 4683, 6771,  
         6799, 6853, 6865, 7537, 7560, 7584,  
         7633, 7814, 7855, 8025, 8031, 8072  
 \l\_\_regex\_success\_pos\_int . . . . .  
     ..... 6746, 6808, 6827, 7015, 7674  
 \l\_\_regex\_success\_submatches\_t1 . . . . .  
     .... 523, 554, 6758, 7018, 7829  
 \\_\_regex\_tests\_action\_cost:n . . . . .  
     .. 6469, 6471, 6484, 6490, 6499, 6517  
 \g\_\_regex\_thread\_info\_intarray . . . . .  
     . 459, 522-524, 530, 6763, 6903, 6974  
 \\_\_regex\_tl\_even\_items:n . . . . .  
     ..... 4522, 4522, 4523, 7558  
 \\_\_regex\_tl\_even\_items\_loop:nn . . . . .  
     ..... 4522, 4525, 4528, 4532  
 \\_\_regex\_tl\_odd\_items:n . . . . .  
     ..... 4522, 4522, 7534, 7557, 7631  
 \\_\_regex\_tmp:w . . . . . 552, 554, 4492,  
     4494, 4498, 4500, 4534, 4534, 5411,  
         5421, 5422, 5423, 5424, 5425, 5448,  
         5459, 5460, 7505, 7514, 7516, 7518,  
         7520, 7522, 7712, 7717, 7735, 7741,  
         7779, 7784, 7788, 7792, 7797, 7807  
 \\_\_regex\_toks\_clear:N . . . . .  
     ..... 4476, 4476, 6410, 6450  
 \\_\_regex\_toks\_memcpy:NNn . . . . .  
     ..... 4481, 4481, 6615  
 \\_\_regex\_toks\_put\_left:Nn . 4490,  
     4490, 6379, 6403, 6445, 6597, 6598  
 \\_\_regex\_toks\_put\_right:Nn . . . . .  
     460, 4490, 4496, 4502, 6355, 6358,  
         6375, 6423, 6447, 6458, 6689, 6734  
 \\_\_regex\_toks\_set:Nn . . . . .  
     .. 4476, 4477, 4478, 4479, 7603, 8112  
 \\_\_regex\_toks\_use:w . . . . .  
     ..... 4475, 4475, 6912, 7050, 8466  
 \\_\_regex\_trace:nnn . . . . .  
     ..... 8448, 8449, 8451, 8452, 8465  
 \\_\_regex\_trace\_pop:nnN . . . . . 8448, 8450  
 \\_\_regex\_trace\_push:nnN . . . . . 8448, 8448  
 \g\_\_regex\_trace\_regex\_int . . . . . 8458  
 \\_\_regex\_trace\_states:n . . . . . 8459, 8459  
 \\_\_regex\_two\_if\_eq:NNNN . . . . . 4987  
 \\_\_regex\_two\_if\_eq>NNNNTF . . . . .  
     ... 4987, 5246, 5293, 5306, 5340,  
         5510, 5547, 5567, 5568, 5637, 5672,  
         5689, 5690, 5752, 5885, 5892, 7341  
 \\_\_regex\_use\_i\_delimit\_by\_q\_-  
     recursion\_stop:nw 4556, 4558, 5875

\\_\_regex\_use\_none\_delimit\_by\_q\_-  
 nil:w ..... 4530, 4556, 4560  
 \\_\_regex\_use\_none\_delimit\_by\_q\_-  
 recursion\_stop:w .....  
 ..... 4556, 4556, 5853, 5877, 7830  
 \\_\_regex\_use\_state: .....  
 ..... 6908, 6908, 6925, 6951  
 \\_\_regex\_use\_state\_and\_submatches:w  
 ..... 527, 6901, 6917, 6917  
 \\_\_regex\_Z\_test: ..... 474, 5441,  
 5443, 5460, 6080, 6201, 6686, 6713  
 \l\\_\_regex\_zeroth\_submatch\_int ...  
 ..... 548, 554, 7572, 7677, 7679,  
 7682, 7685, 7815, 7825, 7827, 7839,  
 7842, 7863, 7868, 7872, 8087, 8091  
 register commands:  
 register\_luadata ..... 12013  
 \relax ..... 4, 21,  
 25, 30, 34, 70, 71, 73, 82, 107, 120,  
 121, 122, 123, 124, 125, 126, 127,  
 128, 129, 130, 132, 133, 134, 135,  
 136, 137, 138, 139, 140, 141, 142, 423  
 \relpenalty ..... 424  
 \RequirePackage ..... 110  
 \resettimer ..... 781  
 reverse commands:  
 \reverse\_if:N .....  
 27, 674, 727, 824, 825, 1020, 1429,  
 1434, 4578, 4579, 4596, 4597, 4602,  
 4603, 8671, 12052, 13826, 17272,  
 17422, 17424, 17426, 17428, 17483,  
 20205, 20210, 20214, 20216, 23427,  
 27007, 27829, 27852, 29726, 29750  
 \right ..... 425  
 \rightghost ..... 895  
 \righthyphenmin ..... 426  
 \rightmarginkern ..... 691  
 \rights skip ..... 427  
 \rmfamily ..... 32502  
 \romannumeral ..... 428  
 round ..... 254  
 \rpcode ..... 692

**S**

\saveboxresource ..... 938  
 \savecatcodetable ..... 896  
 \saveimageresource ..... 939  
 \savepos ..... 937  
 \savinghyphcodes ..... 559  
 \savingvdiscards ..... 560  
 scan commands:  
 \scan\_align\_safe\_stop: ..... 37564  
 \scan\_new:N ..... 142, 711, 795,  
 3341, 3342, 3750, 9262, 9263, 10425,  
 10426, 10900, 12969, 13244, 13245,  
 13246, 13255, 13256, 13997, 16442,  
 16442, 16469, 16470, 16471, 17181,  
 17182, 18063, 18064, 18747, 19105,  
 19106, 19485, 19486, 19681, 19686,  
 19687, 20085, 20086, 20430, 20593,  
 20594, 20595, 20596, 20856, 20857,  
 20858, 22543, 22546, 22547, 22548,  
 22549, 22551, 22552, 22553, 22554,  
 22555, 22654, 29864, 29873, 29874,  
 34843, 34857, 36570, 36882, 36883  
 \scan\_stop: ..... 13, 22,  
 23, 142, 157, 193, 351, 354, 372,  
 373, 375, 384, 390, 401, 451, 466,  
 474, 497, 573, 677, 681, 688, 689,  
 691, 696, 702, 727, 794, 824, 829,  
 878, 886, 888, 890, 892, 903, 907,  
 908, 913, 1016, 1020–1022, 1025,  
 1236, 1237, 159, 173, 1458, 1458,  
 1821, 1839, 1849, 1867, 1893, 2222,  
 2245, 2254, 2263, 2335, 2777, 2901,  
 2902, 2917, 2957, 2983, 3007, 3024,  
 3228, 3234, 3375, 3756, 3888, 3928,  
 3932, 3938, 3940, 3987, 3989, 4258,  
 4264, 4266, 4283, 4285, 4295, 4336,  
 4337, 4338, 4344, 4353, 4662, 4663,  
 4776, 4836, 4861, 4873, 5009, 5845,  
 6163, 6167, 6170, 6325, 6927, 7366,  
 7418, 7712, 8980, 8984, 9233, 10237,  
 10242, 10342, 10461, 10464, 11013,  
 11020, 11415, 12282, 12452, 12462,  
 12529, 12558, 12560, 12924, 12944,  
 12958, 13827, 14107, 15096, 16178,  
 16451, 16454, 16829, 17613, 18976,  
 19150, 19222, 19534, 19595, 19671,  
 19674, 19676, 20097, 20116, 20118,  
 20122, 20125, 20128, 20132, 20137,  
 20141, 20364, 20440, 20458, 20460,  
 20468, 20470, 20474, 20476, 20497,  
 20502, 20505, 20531, 20551, 20553,  
 20561, 20563, 20567, 20569, 20573,  
 22043, 22126, 22227, 22265, 22272,  
 22434, 22529, 22718, 23425, 23429,  
 23630, 23647, 23948, 23995, 23996,  
 24251, 24294, 24322, 24336, 25105,  
 26918, 26926, 27671, 27674, 27677,  
 27680, 27683, 27686, 27689, 27692,  
 27695, 28960, 28987, 29890, 29891,  
 32897, 33045, 34623, 37365, 37368  
 \s\_stop ..... 142, 795, 16454, 16465  
 scan internal commands:  
 \s\_\_bool\_mark ..... 36882, 36903, 36917  
 \s\_\_bool\_stop ..... 36882, 36903, 36917  
 \s\_\_char\_stop ..... 18747

\s\_\_clist\_mark . . . . . 849, 852–854,  
   859, 18063, 18065, 18093, 18094,  
   18111, 18233, 18243, 18247, 18269,  
   18331, 18337, 18351, 18363, 18364,  
   18365, 18368, 18369, 18370, 18379,  
   18380, 18389, 18585, 18586, 18598,  
   18599, 18612, 18620, 18626, 18629  
 \s\_\_clist\_stop . . . . . 853, 855, 859, 18063, 18066, 18067,  
   18079, 18083, 18218, 18221, 18233,  
   18236, 18244, 18247, 18255, 18269,  
   18337, 18365, 18368, 18369, 18381,  
   18389, 18432, 18433, 18440, 18444,  
   18446, 18448, 18455, 18460, 18476,  
   18477, 18503, 18504, 18511, 18516,  
   18518, 18520, 18526, 18533, 18561,  
   18566, 18587, 18598, 18599, 18600,  
   18613, 18626, 18629, 18659, 18693  
 \s\_\_color\_mark . . . . . 34857, 35103, 35105, 35108, 35115,  
   35356, 35361, 35367, 35370, 35377,  
   35408, 35508, 35514, 35593, 35596,  
   35606, 35655, 36023, 36065, 36068,  
   36086, 36092, 36208, 36237, 36240,  
   36254, 36265, 36269, 36272, 36280,  
   36286, 36290, 36293, 36306, 36316  
 \s\_\_color\_stop . . . . . 1344,  
   34843, 34885, 34891, 34892, 34899,  
   34903, 34904, 34907, 34913, 34915,  
   34917, 34919, 34921, 34923, 34942,  
   34944, 34950, 34970, 34999, 35016,  
   35022, 35044, 35103, 35105, 35108,  
   35115, 35122, 35124, 35133, 35144,  
   35145, 35147, 35149, 35151, 35191,  
   35198, 35199, 35200, 35209, 35218,  
   35283, 35288, 35316, 35356, 35361,  
   35367, 35370, 35377, 35385, 35408,  
   35508, 35514, 35545, 35548, 35581,  
   35587, 35593, 35596, 35606, 35655,  
   35674, 35678, 35703, 35705, 35707,  
   35709, 35727, 35842, 35855, 35859,  
   35870, 35877, 35885, 35891, 35899,  
   35901, 35907, 35911, 35912, 35933,  
   35935, 36014, 36020, 36023, 36031,  
   36045, 36059, 36062, 36065, 36069,  
   36072, 36082, 36086, 36092, 36119,  
   36148, 36203, 36204, 36211, 36222,  
   36223, 36237, 36240, 36254, 36265,  
   36269, 36272, 36280, 36286, 36290,  
   36293, 36306, 36316, 36360, 36361  
 \s\_\_cs\_mark . . . . . 372, 406,  
   407, 1809, 1810, 1813, 1814, 1815,  
   2901, 2931, 2932, 2934, 2940, 2944,  
   2966, 2975, 2994, 3022, 3025, 3033,

28329, 28404, 28405, 28515, 28517,  
 28549, 28815, 28823, 28826, 28905  
`\s_fp_(type)` ..... 1010  
`\s_fp_division` ..... 22551  
`\s_fp_exact` ..... 22551, 22556,  
 22557, 22558, 22559, 22560, 25127  
`\s_fp_expr_mark` .....  
 ... 1016, 1017, 1020, 1042, 1045,  
 22546, 24201, 24214, 24295, 24337  
`\s_fp_expr_stop` ..... 986,  
 22546, 22744, 24103, 24202, 24206,  
 24215, 25209, 25220, 25230, 25238  
`\s_fp_invalid` ..... 22551  
`\s_fp_mark` 22548, 22693, 22694, 22698  
`\s_fp_overflow` ..... 22551, 22577  
`\s_fp_stop` ..... 984, 22548,  
 22550, 22594, 22670, 22681, 22688,  
 22694, 22698, 22712, 22731, 23525,  
 23529, 24033, 24038, 24634, 24656,  
 24789, 24795, 24839, 24840, 24865,  
 24866, 25032, 25033, 25034, 25201,  
 25202, 26822, 26837, 28156, 28160  
`\s_fp_tuple` ..... 983, 22654,  
 22660, 22661, 22738, 22740, 24414,  
 24626, 24641, 24666, 24668, 24685,  
 24686, 24688, 24786, 24910, 24911,  
 26051, 26052, 26058, 26059, 28132  
`\s_fp_underflow` ..... 22551, 22575  
`\s_int_mark` .....  
 ... 17181, 17395, 17398, 17464, 17471  
`\s_int_stop` .....  
 ... 824, 835, 836, 17181, 17183,  
 17374, 17390, 17392, 17396, 17409,  
 17464, 17471, 17875, 17881, 17898  
`\s_iow_mark` . 10425, 10763, 10770,  
 10782, 10856, 10857, 10858, 10859  
`\s_iow_stop` .....  
 ... 10425, 10427, 10649, 10690,  
 10748, 10786, 10799, 10856, 10859  
`\s_kernel_stop` 2235, 2243, 2252, 2261  
`\s_keys_mark` .....  
 ... 20856, 20909, 20912, 20920, 20927,  
 21627, 21630, 21635, 21641, 21875,  
 21878, 21887, 21889, 21894, 21897  
`\s_keys_nil` .....  
 ... 20856, 20904, 20905, 20907,  
 20909, 20912, 20917, 20926, 20927,  
 20935, 21622, 21623, 21625, 21627,  
 21630, 21633, 21641, 21642, 21874,  
 21877, 21883, 21885, 21893, 21896  
`\s_keys_stop` .....  
 ... 20856, 20945, 20955, 21129,  
 21234, 21241, 21610, 21620, 21807,  
 21827, 21950, 21957, 21962, 21967

`\s_keyval_mark` ..... 924–926,  
 929, 20593, 20609, 20619, 20630,  
 20631, 20632, 20633, 20639, 20640,  
 20642, 20647, 20648, 20651, 20652,  
 20653, 20658, 20659, 20663, 20664,  
 20667, 20668, 20671, 20672, 20675,  
 20678, 20679, 20684, 20687, 20688,  
 20692, 20693, 20696, 20699, 20703,  
 20704, 20705, 20706, 20713, 20714,  
 20723, 20724, 20726, 20730, 20742,  
 20743, 20751, 20752, 20763, 20764,  
 20765, 20766, 20768, 20789, 20790,  
 20795, 20799, 20801, 20803, 20815  
`\s_keyval_nil` ..... 925,  
 20593, 20638, 20646, 20651, 20653,  
 20654, 20655, 20657, 20663, 20666,  
 20671, 20675, 20677, 20684, 20686,  
 20692, 20696, 20698, 20703, 20705,  
 20713, 20724, 20742, 20751, 20788,  
 20792, 20808, 20811, 20815, 20816  
`\s_keyval_stop` ... 20593, 20631,  
 20633, 20644, 20652, 20664, 20672,  
 20675, 20681, 20693, 20696, 20698,  
 20699, 20703, 20713, 20742, 20743,  
 20751, 20752, 20763, 20766, 20768  
`\s_keyval_tail` ..... 925, 20593,  
 20609, 20619, 20627, 20628, 20637,  
 20721, 20723, 20729, 20730, 20765  
`\s_msg_mark` .....  
 ... 9262, 9611, 9676, 9677, 9682, 9685  
`\s_msg_stop` ..... 9262,  
 9264, 9613, 9617, 9619, 9678, 10128  
`\s_pdf_stop` . 36570, 36620, 36621,  
 36629, 36641, 36654, 36658, 36660  
`\s_peek_mark` .....  
 ... 19485, 19648, 19649, 19656  
`\s_peek_stop` .....  
 ... 19485, 19487, 19637, 19650, 19659  
`\s_prg_mark` ..... 1658, 1660, 1668  
`\s_prg_stop` 1685, 1690, 1709, 1717,  
 1725, 1779, 1783, 1785, 1787, 1789  
`\s_prop` .....  
 ... 892, 896, 903, 904, 1395, 19681,  
 19681, 19682, 19685, 19783, 19786,  
 19921, 19944, 19993, 19994, 19995,  
 19996, 20000, 20001, 20002, 20003,  
 20017, 20031, 20032, 20033, 20034,  
 20038, 20039, 20040, 20041, 20062,  
 20064, 20073, 20076, 36931, 36936  
`\s_prop_mark` .....  
 ... 895, 896, 19686, 19783, 19785, 19786  
`\s_prop_stop` ..... 895,  
 896, 19686, 19783, 19786, 36920, 36927  
`\s_quark` .....

16178, 16413, 16415, 16416, 16427,  
 16430, 16435, 16438, 16440, 16459  
 $\backslash g_{\_}scan_{\_}marks_{\_}tl$  ..... 795, 16444, 16450, 16454  
 $\backslash s_{\_}seq$  ..... 796, 800,  
 806, 810, 813, 1395, 1398, 16469,  
 16480, 16510, 16515, 16520, 16525,  
 16536, 16568, 16596, 16604, 16608,  
 16831, 16879, 17099, 17152, 36941,  
 36947, 36978, 37039, 37051, 37053  
 $\backslash s_{\_}seq_{\_}mark$  .....  
 .. 16470, 17087, 17088, 17102, 17105  
 $\backslash s_{\_}seq_{\_}stop$  .....  
 16470, 16784, 16787, 16795, 16797,  
 16878, 16879, 17089, 17102, 17105,  
 17107, 36940, 36941, 36943, 36947,  
 36951, 36953, 36958, 37042, 37053  
 $\backslash s_{\_}skip_{\_}stop$  ... 20430, 20491, 20493  
 $\backslash s_{\_}sort_{\_}mark$  ..... 424, 427–429,  
 3341, 3536, 3540, 3546, 3550, 3556,  
 3559, 3624, 3625, 3627, 3664, 3666,  
 3669, 3673, 3676, 3679, 3681, 3684  
 $\backslash s_{\_}sort_{\_}stop$  ... 426, 428, 429, 3341,  
 3612, 3621, 3625, 3627, 3664, 3665,  
 3666, 3671, 3673, 3677, 3679, 3687  
 $\backslash s_{\_}str$  ..... 738,  
 746, 764, 767, 13997, 14146, 14150,  
 14332, 14379, 14447, 14450, 14894,  
 14906, 14911, 14921, 14926, 14931,  
 14934, 14949, 14962, 14965, 15100,  
 15101, 15118, 15124, 15140, 15146,  
 15147, 15252, 15267, 15276, 15277  
 $\backslash s_{\_}str_{\_}mark$  .....  
 714, 720, 727, 13255, 13443, 13478,  
 13485, 13568, 13585, 13833, 13835  
 $\backslash s_{\_}str_{\_}stop$  ... 720, 724, 762, 766,  
 767, 13255, 13257, 13258, 13350,  
 13443, 13478, 13485, 13568, 13577,  
 13583, 13585, 13591, 13608, 13627,  
 13689, 13746, 13758, 13796, 13812,  
 13819, 13827, 13829, 13833, 13835,  
 14146, 14152, 14194, 14199, 14207,  
 14402, 14405, 14424, 14430, 14809,  
 14811, 14819, 14907, 14943, 15057,  
 15059, 15063, 15075, 15215, 15217,  
 15221, 15233, 15242, 15249, 15270  
 $\backslash s_{\_}text_{\_}recursion_{\_}stop$  ... 29873,  
 29876, 30222, 30236, 30245, 30340,  
 30349, 30489, 30497, 30576, 30584  
 $\backslash s_{\_}text_{\_}recursion_{\_}tail$  .....  
 ..... 29873, 29880, 29881, 30222  
 $\backslash s_{\_}text_{\_}stop$  .....  
 .. 29864, 29952, 29954, 30510, 30511  
 $\backslash s_{\_}tl$  433–436, 443–445, 3749, 3750,  
 3985, 4021, 4027, 4052, 4070, 4075,  
 4092, 4096, 4128, 4131, 4313, 4317  
 $\backslash s_{\_}tl_{\_}act_{\_}stop$  ..... 704, 12969,  
 12975, 12976, 12979, 12982, 12986,  
 12995, 12998, 13001, 13004, 13007,  
 13009, 13011, 13015, 13018, 13024  
 $\backslash s_{\_}tl_{\_}mark$  693, 12602, 12612, 12731,  
 12732, 12735, 12738, 12739, 13244  
 $\backslash s_{\_}tl_{\_}nil$  ..... 698, 12766,  
 12770, 12789, 12792, 12795, 13244  
 $\backslash s_{\_}tl_{\_}stop$  .....  
 .... 683, 693, 697, 12344, 12346,  
 12560, 12566, 12602, 12612, 12617,  
 12618, 12627, 12631, 12633, 12635,  
 12637, 12646, 12647, 12657, 12658,  
 12667, 12672, 12674, 12676, 12733,  
 12735, 12740, 12742, 12772, 12795,  
 12817, 12819, 12837, 12852, 12866,  
 12890, 12915, 13209, 13219, 13244  
 $\backslash s_{\_}token_{\_}mark$  .....  
 .... 884, 19105, 19464, 19465, 19474  
 $\backslash s_{\_}token_{\_}stop$  ... 878, 880, 19105,  
 19210, 19213, 19243, 19278, 19386,  
 19390, 19396, 19419, 19466, 19474  
 $\backslash scan_{\_}text_{\_}tokens$  ..... 897  
 $\backslash scan_{\_}tokens$  ..... 561  
 $\backslash script_{\_}baseline_{\_}shift_{\_}factor$  ..... 1156  
 $\backslash script_{\_}font$  ..... 429  
 $\backslash scriptscript_{\_}baseline_{\_}shift_{\_}factor$  . 1158  
 $\backslash scriptscript_{\_}font$  ..... 430  
 $\backslash scriptscript_{\_}style$  ..... 431  
 $\backslash scriptsize$  ..... 32519  
 $\backslash scriptspace$  ..... 432  
 $\backslash scriptstyle$  ..... 433  
 $\backslash scrollmode$  ..... 434  
 $\backslash scshape$  ..... 32508  
 $\sec$  ..... 255  
 $\sec d$  ..... 255  
 $\backslash selectfont$  ..... 32480  
seq commands:  
 $\backslash c_{\_}empty_{\_}seq$  154, 797, 16480, 16484,  
 16488, 16491, 16685, 16763, 16771  
 $\backslash seq_{\_}clear:N$  ..... 143, 154, 6230,  
 7114, 9674, 9737, 11507, 11600,  
 16487, 16487, 16489, 16494, 16629  
 $\backslash seq_{\_}clear_{\_}new:N$  .....  
 ..... 143, 16493, 16493, 16495  
 $\backslash seq_{\_}concat:NNN$  .....  
 144, 154, 11513, 16582, 16582, 16586  
 $\backslash seq_{\_}const_{\_}from_{\_}clist:Nn$  .....  
 ..... 144, 16533, 16533, 16538  
 $\backslash seq_{\_}count:N$  .....  
 ..... 146, 151, 153, 237, 7113,

11625, 16700, 16890, 16904, [17051](#),  
[17051](#), 17074, 17079, 37002, 37077  
\seq\_elt:w ..... 796  
\seq\_elt\_end: ..... 796  
\seq\_gclear:N .... [143](#), [421](#), 3486,  
[16487](#), 16490, 16492, 16497, 16717  
\seq\_gclear\_new:N .....  
..... [143](#), [16493](#), 16496, 16498  
\seq\_gconcat:NNN .....  
.... [144](#), 11526, [16582](#), 16584, 16587  
\seq\_get:NN [152](#), 6572, 6577, [17132](#),  
17132, 17133, 17138, 17139, 35484  
\seq\_get:NNTF ..... [152](#), [17138](#)  
\seq\_get\_left:NN ..... [145](#),  
[16779](#), 16779, 16789, 16849, 16850,  
16853, 17132, 17133, 17138, 17139  
\seq\_get\_left:NNTF ..... [147](#), [16849](#)  
\seq\_get\_right:NN ..... [145](#), [16804](#),  
16804, 16821, 16851, 16852, 16855  
\seq\_get\_right:NNTF ..... [147](#), [16849](#)  
\seq\_gpop:NN .....  
..... [152](#), 11436, [17132](#), 17136,  
17137, 17142, 17143, 29084, 35477  
\seq\_gpop:NNTF .....  
.... [153](#), 10226, 10450, [17138](#), 29055, 29067  
\seq\_gpop\_item:NnN .....  
.... [311](#), [37060](#), 37062, 37065, 37068, 37071  
\seq\_gpop\_item:NnNTF [311](#), [37060](#), 37063  
\seq\_gpop\_left:NN .....  
.... [146](#), [16790](#), 16792, 16803, 16860,  
16871, 17136, 17137, 17142, 17143  
\seq\_gpop\_left:NNTF ..... [147](#), [16857](#)  
\seq\_gpop\_right:NN .....  
.... [146](#), [16822](#), 16824, 16848, 16866, 16875  
\seq\_gpop\_right:NNTF ..... [147](#), [16857](#)  
\seq\_gpush:Nn .....  
.... [30](#), [153](#), 10251, 10474, 11421,  
[17112](#), 17122, 17123, 17124, 17125,  
17126, 17127, 17128, 17129, 17130,  
17131, 29059, 29069, 29078, 35413  
\seq\_gput\_left:Nn .....  
.... [145](#), [16592](#), 16600, 16611, 16612,  
17122, 17123, 17124, 17125, 17126,  
17127, 17128, 17129, 17130, 17131  
\seq\_gput\_right:Nn .....  
.... [145](#), 10875, 10882,  
11410, [16613](#), 16615, 16617, 16618  
\seq\_gremove\_all:Nn .....  
.... [148](#), [16639](#), 16641, 16664  
\seq\_gremove\_duplicates:N .....  
.... [148](#), [16623](#), 16625, 16638  
\seq\_greverse:N .....  
.... [148](#), [16665](#), 16667, 16682  
\seq\_gset\_eq:NN .....  
.. [143](#), 3460, 16491, [16499](#), 16503,  
16504, 16505, 16506, 16626, 16697  
\seq\_gset\_filter:NNn [310](#), [36961](#), 36963  
\seq\_gset\_from\_clist:NN .....  
.... [144](#), [16507](#), 16517, 16530, 16531  
\seq\_gset\_from\_clist:Nn .....  
.... [144](#), [16507](#), 16522, 16532  
\seq\_gset\_from\_function:NnN .....  
.... [310](#), [36981](#), 36983  
\seq\_gset\_from\_inline\_x:Nnn .....  
.... [310](#), 3478, 16712, [36971](#), 36973, 36984  
\seq\_gset\_item:Nnn .....  
.... [311](#), [36986](#), 36988, 36991, 36994, 36997  
\seq\_gset\_item:NnnTF ..... [311](#), [36986](#)  
\seq\_gset\_map:NNn .. [151](#), [17041](#), 17043  
\seq\_gset\_map\_x:NNn [151](#), [17031](#), 17033  
\seq\_gset\_split:Nnn .....  
.... [144](#), [16539](#), 16541, 16579  
\seq\_gset\_split\_keep\_spaces:Nnn .....  
.... [144](#), [16539](#), 16545, 16581  
\seq\_gshuffle:N .....  
.... [148](#), [16693](#), 16697, 16736, 16739  
\seq\_gsort:Nn .....  
.... [148](#), [3456](#), 3459, 3461, [16683](#)  
\seq\_if\_empty:N ..... 16683, 16691  
\seq\_if\_empty:NTF .....  
.... [148](#), 7110, [16683](#), 16903, 18141, 29118  
\seq\_if\_empty\_p:N ..... [148](#), [16683](#)  
\seq\_if\_exist:N ..... 16588, 16590  
\seq\_if\_exist:NTF .....  
.... [145](#), 16494, 16497, [16588](#), 17077  
\seq\_if\_exist\_p:N ..... [145](#), [16588](#)  
\seq\_if\_in:Nn ..... [854](#), 16740, 16759  
\seq\_if\_in:NnTF ..... [149](#),  
.... [153](#), [154](#), 10250, 10473, 16632, [16740](#)  
\seq\_indexed\_map\_function:NN .....  
.... [37653](#), 37656  
\seq\_indexed\_map\_inline:Nn .....  
.... [37653](#), 37654  
\seq\_item:Nn .. [56](#), [146](#), 809, 9755,  
9756, 9761, [16877](#), 16877, 16900, 16904  
\seq\_log:N ... [155](#), [17144](#), 17146, 17147  
\seq\_map\_break: .....  
.... [150](#), [151](#), [310](#), [16907](#),  
16907, 16908, 16910, 16920, 16961,  
16971, 16994, 17001, 17010, 21670  
\seq\_map\_break:n .....  
.... [150](#), 809, 3457, 3460,  
9694, 9708, 11046, 11146, [16907](#), 16909  
\seq\_map\_function:NN .. [6](#), [149](#),  
.... [308](#), [811](#), 6250, 6318, 9759, 11516,  
[16911](#), 16911, 16934, 17159, 18147

```

\seq_map_indexed_function:NN . . .
    . . . 149, 16998, 16998, 37655, 37656
\seq_map_indexed_inline:Nn . . .
    . . . 150, 16998, 17003, 37653, 37654
\seq_map_inline:Nn . . . 149,
    154, 1396, 3457, 3460, 9689, 11145,
    16630, 16957, 16957, 16963, 21663
\seq_map_tokens:Nn . . . .
    149, 11045, 11629, 16964, 16964, 16973
\seq_map_variable:NNn . . . .
    . . . 149, 16986, 16986, 16996, 16997
\seq_mapthread_function:NNN . . .
    . . . 310, 36939, 36939, 36960
\seq_new:N . . . 6, 143, 3328, 4541,
    4985, 6336, 6337, 7025, 9644, 9645,
    10178, 10409, 10867, 10892, 10898,
    10899, 16481, 16481, 16486, 16494,
    16497, 16622, 16695, 17169, 17170,
    17171, 17172, 18292, 18844, 18847,
    20846, 28909, 28910, 28911, 35397
\seq_pop:NN . . . .
    . . . 152, 6401, 6439, 6441, 7183,
    17132, 17134, 17135, 17140, 17141
\seq_pop:NNTF . . . 153, 17138
\seq_pop_item:NnN . . . .
    311, 37060, 37060, 37064, 37066, 37070
\seq_pop_item:NnNTF 311, 37060, 37061
\seq_pop_left:NN . . . .
    . . . 145, 16790, 16790, 16802, 16857,
    16869, 17134, 17135, 17140, 17141
\seq_pop_left:NNTF . . . 147, 16857
\seq_pop_right:NN 146, 6179, 6257,
    16822, 16822, 16847, 16863, 16873
\seq_pop_right:NNTF . . . 147, 16857
\seq_push:Nn . . . .
    153, 6411, 6432, 6434, 7343, 17112,
    17112, 17113, 17114, 17115, 17116,
    17117, 17118, 17119, 17120, 17121
\seq_put_left:Nn . . . 145,
    9684, 16592, 16592, 16609, 16610,
    17112, 17113, 17114, 17115, 17116,
    17117, 17118, 17119, 17120, 17121
\seq_put_right:Nn . . 145, 153, 154,
    6182, 6255, 9745, 11602, 16613,
    16613, 16619, 16620, 16633, 37497
\seq_rand_item:N . . . .
    . . . 146, 16901, 16901, 16906
\seq_remove_all:Nn . . 144, 148, 153,
    154, 16639, 16639, 16663, 18324, 37499
\seq_remove_duplicates:N . . 148,
    153, 154, 11514, 16623, 16623, 16637
\seq_reverse:N . . . .
    . . . 148, 802, 16665, 16665, 16681
\seq_set_eq:NN . . . 143,
    154, 3457, 16488, 16499, 16499,
    16500, 16501, 16502, 16624, 16696
\seq_set_filter:NNn . . . .
    . . . 310, 812, 6313, 36961, 36961
\seq_set_from_clist:NN . . . .
    144, 16507, 16507, 16527, 16528, 18323
\seq_set_from_clist:Nn . . . .
    . . . 144, 173, 798, 11510,
    11524, 16507, 16512, 16529, 21578
\seq_set_from_function:NnN . . .
    . . . 310, 7736, 36981, 36981
\seq_set_from_inline_x:Nnn . . .
    . . . 310, 1397, 36971, 36971, 36982
\seq_set_item:Nnn . . . 311, 1399,
    36986, 36986, 36990, 36992, 36996
\seq_set_item:NnnTF . . . 311, 36986
\seq_set_map:NNn . . . 151, 17041, 17041
\seq_set_map_x:NNn . . . .
    . . . 151, 812, 6326, 17031, 17031
\seq_set_split:Nnn 144, 6312, 6325,
    16539, 16539, 16578, 18845, 18848
\seq_set_split_keep_spaces:Nnn . .
    . . . 144, 16539, 16543, 16580
\seq_show:N . . . .
    . . . 155, 607, 710, 17144, 17144, 17145
\seq_shuffle:N . . . 148, 16693,
    16696, 16731, 16734, 16736, 16738
\seq_sort:Nn . . . .
    . . . 44, 148, 3456, 3456, 3458, 16683
\seq_use:Nn . . . .
    . . . 152, 6329, 17075, 17109, 17111
\seq_use:Nnn . . . .
    . . . 151, 17075, 17075, 17097, 17110
\g_tmpa_seq . . . 155, 17169
\l_tmpa_seq . . . 155, 17169
\g_tmpb_seq . . . 155, 17169
\l_tmpb_seq . . . 155, 17169
seq internal commands:
\__seq_count:w . . . .
    813, 17051, 17056, 17069, 17072, 17073
\__seq_count_end:w . . . 812,
    17051, 17058, 17059, 17060, 17061,
    17062, 17063, 17064, 17065, 17073
\__seq_get_left:wnw . . . .
    . . . 16779, 16783, 16787
\__seq_get_right_end:NnN . . . .
    . . . 16804, 16812, 16820
\__seq_get_right_loop:nw . . . .
    . . . 806, 16804, 16809, 16815, 16818
\__seq_if_in: . . . 16740, 16749, 16757
\__seq_int_eval:w . . . .
    . . . 36985, 36985, 37041, 37051

```

```

\l__seq_internal_a_int ..... 16707, 16713, 16722, 16724, 16725
\l__seq_internal_a_t1 ..... 799, 1398, 1399, 16477, 16551, 16555,
16561, 16566, 16568, 16654, 16659, 16744, 16748, 37000, 37056, 37074
\l__seq_internal_b_int ..... 16723, 16726, 16727
\l__seq_internal_b_t1 ..... 1399, 16477, 16650, 16654, 16747, 16748, 37075, 37083, 37090
\g__seq_internal_seq ..... 16693
\__seq_item:n . 796, 801, 804, 805, 807–810, 813, 1395, 1396, 16472, 16472, 16596, 16604, 16614, 16616, 16621, 16671, 16672, 16674, 16679, 16709, 16745, 16784, 16787, 16797, 16812, 16815, 16828, 16829, 16840, 16884, 16893, 16918, 16923, 16924, 16925, 16926, 16938, 16943, 16949, 16953, 16969, 16975, 16976, 16977, 16978, 17021, 17023, 17037, 17047, 17058, 17059, 17060, 17061, 17062, 17063, 17064, 17065, 17070, 17071, 17086, 17101, 17104, 17107, 36977, 36978, 37000, 37046, 37049, 37087
\__seq_item:nN .. 16877, 16882, 16887
\__seq_item:nwn ..... 16877, 16881, 16893, 16898
\__seq_item:wNn .. 16877, 16878, 16879
\__seq_map_function:Nw ..... 809, 16911, 16914, 16922, 16932
\__seq_map_indexed>NN ..... 17000, 17008, 17013
\__seq_map_indexed:nNN .. 16998
\__seq_map_indexed:Nw ..... 811, 16998, 17015, 17023, 17027
\__seq_map_tokens:Nw ..... 16964, 16967, 16974, 16984
\__seq_mapthread_function:Nnnwnn ..... 36939, 36949, 36953, 36958
\__seq_mapthread_function:wNN .. 36939, 36940, 36941
\__seq_mapthread_function:wNw .. 36939, 36943, 36947
\__seq_pop:NNNN ..... 16761, 16761, 16791, 16793, 16823, 16825
\__seq_pop_item:nn 1399, 37078, 37080
\__seq_pop_item:NnNN ..... 37067, 37069, 37072
\__seq_pop_item:NnNNN .. 37060
\__seq_pop_item_aux:w ..... 1399, 37078, 37087
\__seq_pop_item_def: ..... 796, 16661, 16711, 16935, 16951, 16961, 16994, 17039, 17049, 36969, 36979
\__seq_pop_left:NNN ..... 16790, 16791, 16793, 16794, 16859, 16862
\__seq_pop_left:wnwNNN ..... 16790, 16795, 16796
\__seq_pop_right:NNN . 801, 16822, 16823, 16825, 16826, 16865, 16868
\__seq_pop_right_loop:nn ..... 16822, 16833, 16842, 16845
\__seq_pop_TF:NNNN ..... 807, 16761, 16769, 16850, 16852, 16859, 16862, 16865, 16868
\__seq_push_item_def: ..... 16708, 16935, 16937, 16942, 16945
\__seq_push_item_def:n ..... 796, 16645, 16935, 16935, 16940, 16959, 16988, 17037, 17047, 36967, 36977
\__seq_put_left_aux:w ..... 800, 16592, 16597, 16605, 16608
\__seq_remove_all_aux:NNn ..... 16639, 16640, 16642, 16643
\__seq_remove_duplicates:NN ..... 16623, 16624, 16626, 16627
\l__seq_remove_seq ..... 16622, 16629, 16632, 16633, 16635
\__seq_reverse:NN ..... 16665, 16666, 16668, 16669
\__seq_reverse_item:nw ..... 802, 803
\__seq_reverse_item:nwn ..... 16665, 16672, 16676
\__seq_set_filter:NNNn ..... 36961, 36962, 36964, 36965
\__seq_set_from_inline_x:NNnn ... 36971, 36972, 36974, 36975
\__seq_set_item:NnnNN .. 36986, 36987, 36989, 36993, 36995, 36998
\__seq_set_item:nnNNNN ..... 1399, 36986, 37001, 37004, 37076
\__seq_set_item:nNnnNNNN ..... 1398, 36986, 37007, 37012, 37035
\__seq_set_item:wn ..... 36986, 37040, 37046, 37050
\__seq_set_item_end:w ..... 1398, 36986, 37048, 37053
\__seq_set_item_false:nnNNNN ... 1398, 36986, 37015, 37017
\__seq_set_map:NNNn ..... 17041, 17042, 17044, 17045
\__seq_set_map_x:NNNn ..... 17031, 17032, 17034, 17035
\__seq_set_split:NNnn ..... 16539

```

\__seq_set_split:NNNnn . . . . .	255
.. 16540, 16542, 16544, 16546, 16547	
\__seq_set_split:Nw . . . . .	37450
.. 16539, 16557, 16564, 16570	
\__seq_set_split:w 16539, 16572, 16576	37470
\__seq_set_split_auxi:w . . . . .	1161
\__seq_set_split_auxii:w . . . . .	445
\__seq_set_split_end: 799, 16539,	446, 19302
16559, 16563, 16570, 16574, 16576	
\__seq_show:NN . . . . .	skip commands:
.. 17144, 17144, 17146, 17148	\c_max_skip . . . . . 218, 20516
\__seq_show_validate:mn . . . . .	\skip_add:Nn . 216, 20467, 20471
.. 17144, 17153, 17163, 17167	\skip_const:Nn . . . . . 216,
\__seq_shuffle:NN . . . . .	921, 20437, 20437, 20442, 20516, 20517
.. 16693, 16696, 16697, 16698	\skip_eval:n . . . . . 217, 20440,
\__seq_shuffle_item:n . . . . .	20481, 20496, 20496, 20511, 20515
.. 16693, 16709, 16720	\skip_gadd:Nn . 216, 20467, 20469, 20472
\__seq_tmp:w . . . . .	.skip_gset:N . . . . . 227, 21393
.. 16479, 16479,	\skip_gset:Nn . . . . .
16479, 16671, 16674, 16828, 16840	.. 216, 917, 20457, 20459, 20462
\__seq_use:NNNnNnn . . . . .	\skip_gset_eq:NN . . . . .
.. 17075, 17082, 17083, 17098	.. 216, 20463, 20465, 20466
\__seq_use:nwwn . 17075, 17088, 17107	\skip_gsub:Nn . 216, 20467, 20475, 20478
\__seq_use:nwwwnwn . . . . .	\skip_gzero:N . . . . .
.. 17075, 17087, 17099, 17100	.. 216, 20443, 20444, 20446, 20450
\__seq_use_setup:w 17075, 17086, 17099	\skip_gzero_new:N . . . . .
\__seq_wrap_item:n . . . . .	.. 216, 20447, 20449, 20452
.. 799, 1396, 16510, 16515,	\skip_horizontal:N . . . . .
16520, 16525, 16536, 16552, 16577,	.. 218, 20500, 20500, 20502, 20506
16621, 16621, 16657, 17166, 36967	\skip_horizontal:n . 218, 20500, 20501
\setbox . . . . .	\skip_if_eq:nn . . . . . 20479
\setfontid . . . . .	\skip_if_eq:nnTF . . . . . 217, 20479
\setlanguage . . . . .	\skip_if_eq_p:nn . . . . . 217, 20479
\setrandomseed . . . . .	\skip_if_exist:N . . . . . 20453, 20455
\c_seven . . . . .	\skip_if_exist:NTF . . . . .
\sfcode . . . . .	.. 216, 20448, 20450, 20453
\sffamily . . . . .	\skip_if_exist_p:N . . . . . 216, 20453
\shapemode . . . . .	\skip_if_finite:n . . . . . 20487
\shellescape . . . . .	\skip_if_finite:nTF . . . . . 217, 20485
\Shipout . . . . .	\skip_if_finite_p:n . . . . . 217, 20485
\shipout . . . . .	\skip_log:N .. 218, 20512, 20512, 20513
\ShortText . . . . .	\skip_log:n . . . . . 218, 20512, 20514
\show . . . . .	\skip_new:N . . . . . 215, 216,
\showbox . . . . .	20431, 20431, 20436, 20439, 20448,
\showboxbreadth . . . . .	20450, 20518, 20519, 20520, 20521
\showboxdepth . . . . .	.skip_set:N . . . . . 227, 21393
\showgroups . . . . .	\skip_set:Nn . 216, 20457, 20457, 20461
\showifs . . . . .	\skip_set_eq>NN . . . . .
\showlists . . . . .	.. 216, 20463, 20463, 20464
\showmode . . . . .	\skip_show:N . 217, 20508, 20508, 20509
\showstream . . . . .	\skip_show:n . . . . . 217, 920, 20510, 20510
\showthe . . . . .	\skip_sub:Nn . 216, 20467, 20473, 20477
\showtokens . . . . .	\skip_use:N . . . . .
sign . . . . .	.. 217, 20490, 20497, 20498, 20498, 20499
sin . . . . .	\skip_vertical:N . . . . .
	.. 219, 20500, 20503, 20505, 20507
	\skip_vertical:n . . . . . 219, 20500, 20504

```

\skip_zero:N ..... 216,
    219, 907, 20443, 20443, 20445, 20448
\skip_zero_new:N ..... 216, 20447, 20447, 20451
\g_tmpa_skip ..... 218, 20518
\l_tmpa_skip ..... 218, 20518
\g_tmpb_skip ..... 218, 20518
\l_tmpb_skip ..... 218, 20518
\c_zero_skip ..... 218, 907, 20100, 20102, 20516
skip internal commands:
    \__skip_if_finite:wwNw .....
        ..... 20485, 20489, 20493
    \__skip_tmp:w ..... 20485, 20495
\skipdef ..... 447
\slshape ..... 32509
\small ..... 32520
sort commands:
    \sort_ordered: ..... 37566
    \sort_return_same: .....
        ..... 43, 44, 424, 3539, 3539, 37567
    \sort_return_swapped: .....
        ..... 43, 44, 424, 3539, 3549, 37569
    \sort_reversed: ..... 37568
sort internal commands:
    \__sort:nnNnm ..... 425, 426
    \l__sort_A_int .. 423, 3338, 3345,
        3352, 3355, 3364, 3503, 3508, 3511,
        3531, 3563, 3570, 3585, 3587, 3588
    \l__sort_B_int ..... 423, 3338,
        3508, 3512, 3520, 3522, 3523, 3575,
        3576, 3585, 3586, 3595, 3596, 3598
    \l__sort_begin_int .....
        ..... 418, 423, 3336, 3500, 3588, 3598
    \l__sort_block_int . 417, 418, 422,
        3335, 3347, 3352, 3356, 3359, 3364,
        3365, 3430, 3491, 3494, 3501, 3504
    \l__sort_C_int ..... 423, 3338,
        3509, 3513, 3520, 3521, 3532, 3564,
        3571, 3575, 3577, 3578, 3595, 3597
    \__sort_compare:nn 420, 424, 3429, 3530
    \__sort_compute_range: .....
        ..... 417-419, 3369,
        3369, 3377, 3385, 3393, 3406, 3417
    \__sort_copy_block: .....
        ..... 422, 3510, 3518, 3518, 3526
    \__sort_disable_toksdef: .....
        ..... 3416, 3695, 3695
    \__sort_disabled_toksdef:n .....
        ..... 3695, 3696, 3697
    \l__sort_end_int ..... 418,
        422, 423, 3336, 3492, 3500, 3501,
        3502, 3503, 3504, 3505, 3506, 3523
    \__sort_error: 3689, 3689, 3701, 3719
    \__sort_i:nnnnNn ..... 427
    \g__sort_internal_seq .....
        ..... 420, 421, 3328, 3478, 3485, 3486
    \g__sort_internal_tl .....
        ..... 3328, 3441, 3444, 3445
    \l__sort_length_int .....
        ..... 417, 418, 3330, 3427, 3491
    \__sort_level: .....
        ..... 420, 430, 3431, 3489, 3489, 3495, 3693
    \__sort_loop:wNn ..... 426, 427
    \__sort_main>NNNn .....
        ..... 421, 3414, 3414, 3440, 3477
    \l__sort_max_int .....
        ..... 417, 418, 3330, 3349, 3421
    \c__sort_max_length_int ..... 3369
    \__sort_merge_blocks: .....
        ..... 3493, 3498, 3498, 3515, 3692
    \__sort_merge_blocks_aux: .....
        ..... 422, 3514, 3528, 3528, 3581, 3591, 3691
    \__sort_merge_blocks_end: .....
        ..... 425, 3589, 3593, 3593, 3601
    \l__sort_min_int .....
        ..... 417, 418, 420, 3330, 3346,
        3354, 3371, 3387, 3395, 3408, 3418,
        3428, 3442, 3481, 3492, 3717, 3718
    \__sort_quick_cleanup:w .....
        ..... 3603, 3624, 3627
    \__sort_quick_end:nnTFNn .....
        ..... 428, 429, 3623,
        3663, 3663, 3669, 3676, 3681, 3684
    \__sort_quick_only_i:NnnnnNn ...
        ..... 3628, 3631, 3635, 3638
    \__sort_quick_only_i_end:nnnw ...
        ..... 3639, 3663, 3666
    \__sort_quick_only_ii:NnnnnNn ...
        ..... 3628, 3630, 3642, 3644
    \__sort_quick_only_ii_end:nnnw ..
        ..... 3646, 3663, 3673
    \__sort_quick_prepare:Nnnn .....
        ..... 3603, 3609, 3616, 3619
    \__sort_quick_prepare_end:NNNnw ..
        ..... 3603, 3611, 3621
    \__sort_quick_single_end:nnnw ...
        ..... 3632, 3663, 3664
    \__sort_quick_split:NnNn ...
        ..... 427, 428, 3623,
        3628, 3628, 3668, 3675, 3681, 3683
    \__sort_quick_split_end:nnnw ...
        ..... 3653, 3660, 3663, 3679
    \__sort_quick_split_i:NnnnnNn ...
        ..... 426, 3628, 3645, 3649, 3652, 3659
    \__sort_quick_split_ii:NnnnnNn ...
        ..... 3628, 3637, 3651, 3656, 3658

```

\\_\_sort\_redefine\_compute\_range: . . . . .  
     3369, 3376, 3381, 3401  
 \\_\_sort\_return\_mark:w . . . . .  
     424, 3534, 3535,  
     3539, 3540, 3545, 3550, 3555, 3559  
 \\_\_sort\_return\_none\_error: . . . . .  
     424, 3537, 3539, 3560, 3565, 3573, 3583  
 \\_\_sort\_return\_same:w . . . . .  
     424, 3547, 3565, 3573, 3573  
 \\_\_sort\_return\_swapped:w . . . . .  
     3557, 3583, 3583  
 \\_\_sort\_return\_two\_error: . . . . .  
     424, 3539, 3544, 3554, 3567  
 \\_\_sort\_seq>NNNNn . . . . .  
     420, 3456, 3457, 3460, 3464, 3470, 3474  
 \\_\_sort\_shrink\_range: . . . . .  
     418,  
     419, 3343, 3343, 3373, 3389, 3397, 3410  
 \\_\_sort\_shrink\_range\_loop: . . . . .  
     3343, 3348, 3362, 3366  
 \\_\_sort\_t1>NNn . . . . .  
     420, 3433, 3433, 3435, 3437  
 \\_\_sort\_t1\_toks:w . . . . .  
     420, 3433, 3442, 3448, 3452  
 \\_\_sort\_too\_long\_error>NNw . . . . .  
     3422, 3712, 3712  
 \l\_\_sort\_top\_int . . . . .  
     417, 420, 423, 3330, 3418,  
     3421, 3424, 3425, 3428, 3450, 3481,  
     3502, 3505, 3506, 3509, 3578, 3718  
 \l\_\_sort\_true\_max\_int . . . . .  
     417, 418, 3330, 3346,  
     3359, 3372, 3388, 3396, 3409, 3717  
 sp . . . . . 258  
 spac commands:  
     \spac\_directions\_normal\_body\_dir . . . . . 1369  
     \spac\_directions\_normal\_page\_dir . . . . . 1370  
 \spacefactor . . . . . 448  
 \spaceskip . . . . . 449  
 \span . . . . . 450  
 \special . . . . . 451  
 \splitbotmark . . . . . 452  
 \splitbotmarks . . . . . 565  
 \spliddiscards . . . . . 566  
 \splitfirstmark . . . . . 453  
 \splitfirstmarks . . . . . 567  
 \splitmaxdepth . . . . . 454  
 \splittopskip . . . . . 455  
 sqrt . . . . . 256  
 \SS . . . . . 30153, 31878, 32605  
 \ss . . . . . 30153, 31878, 32601  
 \stockheight . . . . . 36687, 36695, 36699, 36703  
 \stockwidth . . . . . 36688, 36696, 36699, 36704  
 str commands:  
     \c\_ampersand\_str . . . . . 133, 13958  
     \c\_atsign\_str . . . . . 133, 13958  
     \c\_backslash\_str . . . . . 133, 4788,  
     5390, 13958, 14660, 14662, 14685,  
     14714, 14716, 14748, 14757, 14761  
     \c\_circumflex\_str . . . . . 133, 13958  
     \c\_colon\_str . . . . . 133, 13958, 19213,  
     19390, 19396, 20955, 35581, 35586  
     \c\_dollar\_str . . . . . 133, 13958  
     \c\_hash\_str . . . . . 133, 13958,  
     14628, 14731, 15306, 15307, 15310,  
     15313, 29750, 29779, 29783, 29852  
     \c\_left\_brace\_str . . . . .  
     133, 476, 4851, 5263,  
     5267, 5287, 5300, 5324, 5798, 5809,  
     5813, 5892, 5916, 7092, 13958, 30289  
     \c\_percent\_str 133, 13958, 14630, 14784  
     \c\_right\_brace\_str . . . . .  
     133, 4887, 5273, 5293, 5306, 5816,  
     5820, 5913, 7089, 13958, 21967, 30297  
 str\_byte . . . . . 14027  
 \str\_case:Nn . . . . . 124, 13422, 13447  
 \str\_case:nn . . . . . 124, 5530,  
     10101, 11299, 13422, 13422, 13444,  
     13445, 13447, 36099, 36137, 37095  
 \str\_case:nnn . . . . . 37570, 37572  
 \str\_case:NnTF . . . . .  
     124, 13422, 13448, 13449, 13450  
 \str\_case:nnTF . . . . . 124, 910,  
     6140, 8893, 8928, 9237, 9939, 13422,  
     13427, 13432, 13437, 13448, 13449,  
     13450, 21152, 21198, 37571, 37573  
 \str\_case\_e:nn 125, 13422, 13457, 37575  
 \str\_case\_e:nnTF . . . . . 125, 2712,  
     5271, 13422, 13462, 13467, 13472,  
     14683, 37577, 37579, 37581, 37583  
 \str\_case\_x:nn . . . . . 37574  
 \str\_case\_x:nnn . . . . . 37576  
 \str\_case\_x:nnTF . . . . . 37578, 37580, 37582  
 \str\_casfold:n 131–133, 268, 13836,  
     13836, 13839, 23616, 35768, 37642,  
     37643, 37644, 37645, 37646, 37647,  
     37648, 37649, 37691, 37692, 37699,  
     37700, 37705, 37706, 37711, 37712  
 \c\_str\_cctab . . . . . 263, 1196, 29189  
 \str\_clear:N . . . . .  
     122, 13263, 20937, 21085, 21608, 21609  
 \str\_clear\_new:N . . . . . 122, 13263  
 \str\_compare:nNn . . . . . 13377, 13384  
 \str\_compare:nNnTF . . . . . 125, 13377  
 \str\_compare\_p:nNn . . . . . 125, 13377  
 \str\_concat:NNN . . . . .  
     122, 13263, 13286, 13288

```

\str_const:Nn ..... 122, 8801, 8820, 8838, 8844,
9199, 11684, 11691, 11695, 11699,
13290, 13958, 13959, 13960, 13961,
13962, 13963, 13964, 13965, 13966,
13967, 13968, 13969, 13970, 14724,
14725, 14747, 20823, 20824, 20825,
20826, 20827, 20828, 20829, 37093
\str_convert_pdfname:n ..... 136, 15282, 15282, 35865
\str_count:N ..... 127, 4189, 9383, 9384, 9565, 9566,
10549, 10627, 13768, 13768, 13769
\str_count:n ..... 127, 4183, 13768, 13768, 13770
\str_count_ignore_spaces:n ..... 127, 725, 3795, 13768, 13783
\str_count_spaces:N ..... 127, 13748, 13748, 13750
\str_count_spaces:n ..... 127, 725, 13748, 13749, 13751, 13774
\str_declare_eight_bit_encoding:nmn
..... 37650, 37651
str_end ..... 15185
str_error ..... 14027
\str_fold_case:n . 37634, 37643, 37645
\str_foldcase:n ..... 133, 198, 37646, 37647, 37649
\str_gclear:N ..... 122, 13263
\str_gclear_new:N ..... 13263
\str_gconcat:NNN ..... 122, 13263, 13287, 13289
\str_gput_left:Nn ..... 123, 13290
\str_gput_right:Nn ..... 123, 13290
\str_gremove_all:Nn ..... 130, 13360, 13362, 13365
\str_gremove_once:Nn ..... 130, 13354, 13356, 13359
\str_greplace_all:Nnn ..... 130, 13314, 13320, 13325, 13363
\str_greplace_once:Nnn ..... 130, 13314, 13316, 13323, 13357
.str_gset:N ..... 227, 21401
\str_gset:Nn ..... 123, 11442, 11443, 11444, 13290
\str_gset_convert:Nnnn ..... 136, 14166, 14168, 14180
\str_gset_convert:NnnnTF . 136, 14166
\str_gset_eq:NN ..... 122, 11429,
11430, 11431, 13263, 13283, 13285
.str_gset_x:N ..... 227, 21401
\str_head:N ..... 128, 726, 13806, 13806, 13807
\str_head:n ..... 128, 700,
726, 12853, 12898, 13806, 13806, 13808
\str_head_ignore_spaces:n ..... 128, 13806, 13816
\str_if_empty:N ..... 13370, 13372
\str_if_empty:n ..... 13374
\str_if_empty:NTF ..... 123, 11162,
13366, 20893, 20916, 21833, 29832
\str_if_empty:nTF ..... 123, 13366
\str_if_empty_p:N ..... 123, 13366
\str_if_empty_p:n ..... 123, 13366
\str_if_eq:NN ..... 13401, 13406
\str_if_eq:nn ..... 199, 893, 902, 13389, 13394, 13396
\str_if_eq:NNTF ..... 123, 716, 13401
\str_if_eq:nNTF ..... 105, 106, 124, 125, 203,
204, 801, 879, 2733, 5173, 8290,
8434, 8856, 8877, 8908, 9075, 9705,
9748, 10028, 10031, 11485, 11548,
11563, 13389, 13453, 13481, 15067,
15070, 15225, 15228, 16647, 19216,
19273, 19842, 19971, 20481, 20988,
21667, 23486, 23559, 24786, 29761,
29779, 29781, 29830, 29852, 30423,
30479, 30830, 30912, 32439, 35054,
35372, 35536, 35570, 35615, 35673,
35967, 35977, 37587, 37589, 37591
\str_if_eq_p:NN ..... 123, 13401
\str_if_eq_p:nn . 124, 8816, 8843,
8844, 8916, 8917, 9207, 9209, 11703,
13389, 29881, 30171, 30189, 30484,
33731, 34615, 35366, 36573, 37585
\str_if_eq_x:nnTF . 37586, 37588, 37590
\str_if_eq_x_p:nn ..... 37584
\str_if_exist:N ..... 13366, 13368
\str_if_exist:NTF ..... 122, 8875, 8944, 13366
\str_if_exist_p:N ..... 122, 13366
\str_if_in:Nn ..... 13408, 13414
\str_if_in:nn ..... 13416
\str_if_in:NnTF ..... 124, 13408
\str_if_in:nNTF . 124, 3264, 13408, 29178
\str_item:Nn . 128, 13610, 13610, 13611
\str_item:nn ..... 128, 721, 725, 13610, 13610, 13612
\str_item_ignore_spaces:nn ..... 128, 721, 13610, 13620
\str_log:N . 132, 13975, 13983, 13988
\str_log:n ..... 132, 13975, 13982
\str_lower_case:n . 37634, 37635, 37637
\str_lowercase:n . 131, 268, 13836,
13837, 13840, 37634, 37635, 37636,
37637, 37693, 37694, 37707, 37708

```

\str\_map\_break: 126, 6039, 13487,  
13493, 13502, 13519, 13527, 13539,  
13545, 13551, 13552, 13554, 13561  
\str\_map\_break:n . . . . .  
..... 126, 127, 3268, 13487, 13553  
\str\_map\_function:NN . . . . .  
..... 125, 720, 13487, 13495, 13506  
\str\_map\_function:NN .. 125, 126,  
718, 6032, 13487, 13487, 13496, 15285  
\str\_map\_inline:Nn . . . . .  
..... 126, 13487, 13522, 13524  
\str\_map\_inline:nn . . . . .  
. 126, 3262, 6788, 13487, 13507, 13523  
\str\_map\_tokens:Nn . . . . .  
..... 126, 13555, 13563, 13564  
\str\_map\_tokens:nn . . . . .  
..... 126, 13555, 13555, 13563  
\str\_map\_variable:NNn . . . . .  
..... 126, 13487, 13541, 13550  
\str\_map\_variable:nNn . . . . .  
..... 126, 13487, 13531, 13542  
\str\_new:N . . . . .  
... 122, 9260, 9261, 10864, 10865,  
10866, 10895, 10896, 10897, 11721,  
13263, 13971, 13972, 13973, 13974,  
20833, 20835, 20838, 20840, 20843  
str\_overflow . . . . . 15185  
\str\_put\_left:Nn . . . . . 123, 13290  
\str\_put\_right:Nn . . . . . 123, 13290  
\str\_range:NNn 129, 13671, 13671, 13672  
\str\_range:nnn . . . . . 95, 129,  
725, 4186, 6142, 13671, 13671, 13673  
\str\_range\_ignore\_spaces:nnn . . .  
..... 129, 13671, 13681  
\str\_remove\_all:Nn . . . . .  
..... 130, 13360, 13360, 13364  
\str\_remove\_once:Nn . . . . .  
..... 130, 13354, 13354, 13358  
\str\_replace\_all:NNn . . . . .  
... 130, 13314, 13318, 13324, 13361  
\str\_replace\_once:NNn . . . . .  
... 130, 13314, 13314, 13322, 13355  
.str\_set:N . . . . . 227, 21401  
\str\_set:Nn . . . . . 123, 130,  
227, 9346, 9347, 9553, 9554, 11498,  
11499, 11500, 13290, 13546, 20872,  
20874, 21466, 21468, 21617, 21742  
\str\_set\_convert:Nnnn . . . . . 136,  
137, 737, 748, 14166, 14166, 14171  
\str\_set\_convert:NnnnTF . . . . .  
..... 136, 737, 14166  
\str\_set\_eq:NN 122, 13263, 13282, 13284  
.str\_set\_x:N . . . . . 227, 21401  
\str\_show:N .. 132, 13975, 13976, 13981

\str\_show:n . . . . . 132, 13975, 13975  
\str\_tail:N . . . 128, 13821, 13821, 13822  
\str\_tail:n . . . . .  
    128, 438, 13821, 13821, 13823, 29926  
\str\_tail\_ignore\_spaces:n . . . . .  
    128, 13821, 13830  
\str\_titlecase:n . . . . . 37697, 37698  
\str\_upper\_case:n . . . . . 37634, 37639, 37641  
\str\_uppercase:n . . . . . 131, 268, 13836,  
    13838, 13841, 37638, 37639, 37640,  
    37641, 37695, 37696, 37709, 37710  
\str\_use:N . . . . . 127, 13263  
\c\_tilde\_str . . . . . 133, 13958  
\g\_tmprn\_str . . . . . 133, 13971  
\l\_tmprn\_str . . . . . 130, 133, 13971  
\g\_tmprb\_str . . . . . 133, 13971  
\l\_tmprb\_str . . . . . 133, 13971  
\c\_underscore\_str . . . . . 133, 13958  
\c\_zero\_str . . . . . 133, 13958

str internal commands:

\g\_\_str\_alias\_prop . . . . . 740, 13999, 14237  
\c\_\_str\_byte\_-1\_tl . . . . . 14080  
\c\_\_str\_byte\_0\_tl . . . . . 14080  
\c\_\_str\_byte\_1\_tl . . . . . 14080  
\c\_\_str\_byte\_255\_tl . . . . . 14080  
\c\_\_str\_byte\_{number}\_tl . . . . . 735  
\\_\_str\_case:nnTF . . . . . 13422,  
    13425, 13430, 13435, 13440, 13442  
\\_\_str\_case:nw . . . . .  
    13422, 13443, 13451, 13455  
\\_\_str\_case\_e:nnTF . . . . . 13422,  
    13460, 13465, 13470, 13475, 13477  
\\_\_str\_case\_e:nw . . . . .  
    13422, 13478, 13479, 13483  
\\_\_str\_case\_end:nw . . . . .  
    13422, 13454, 13482, 13485  
\\_\_str\_change\_case:nn . . . . .  
    13836, 13836, 13837, 13838, 13842  
\\_\_str\_change\_case\_aux:nn . . . . .  
    13836, 13844, 13847  
\\_\_str\_change\_case\_char:nN . . . . .  
    13836, 13861, 13870  
\\_\_str\_change\_case\_char:nnn . . . . .  
    13836, 13881, 13899,  
    13902, 13908, 13917, 13930, 13939  
\\_\_str\_change\_case\_char:nnnn . . . . .  
    13836, 13942, 13944  
\\_\_str\_change\_case\_char\_aux:nN . . . . .  
    13836, 13895, 13898  
\\_\_str\_change\_case\_char\_aux:nn . . . . .  
    13836, 13934, 13940  
\\_\_str\_change\_case\_codepoint:nN . . . . .  
    13836, 13874, 13880, 13883

\\_\_str\_change\_case\_codepoint:nNN  
     ..... [13836](#), [13888](#), [13900](#)  
 \\_\_str\_change\_case\_codepoint:nNNN  
     ..... [13836](#), [13891](#), [13906](#)  
 \\_\_str\_change\_case\_codepoint:nNNNN  
     ..... [13836](#), [13915](#)  
 \\_\_str\_change\_case\_codepoint:nNNNNN  
     ..... [13892](#)  
 \\_\_str\_change\_case\_end:nw .. [13836](#)  
 \\_\_str\_change\_case\_end:wn ..  
     ..... [13855](#), [13873](#)  
 \\_\_str\_change\_case\_loop:nw ..  
     .. [13836](#), [13849](#), [13857](#), [13868](#), [13937](#)  
 \\_\_str\_change\_case\_output:nw ..  
     .. [13836](#), [13852](#), [13854](#), [13867](#), [13932](#)  
 \\_\_str\_change\_case\_result:n ..  
     .. [13836](#), [13850](#), [13852](#), [13853](#), [13855](#)  
 \\_\_str\_change\_case\_space:n ..  
     ..... [13836](#), [13860](#), [13865](#)  
 \\_\_str\_collect\_delimit\_by\_q\_-  
     stop:w .. [13699](#), [13722](#), [13722](#)  
 \\_\_str\_collect\_end:nnnnnnnnw ..  
     ..... [724](#), [13722](#), [13741](#), [13746](#)  
 \\_\_str\_collect\_end:wn ..  
     ..... [13722](#), [13729](#), [13739](#)  
 \\_\_str\_collect\_loop:wn ..  
     ..... [13722](#), [13723](#), [13724](#), [13735](#)  
 \\_\_str\_collect\_loop:wnNNNNNNN ..  
     ..... [13722](#), [13727](#), [13733](#)  
 \\_\_str\_convert:nnn ..  
     ..... [739](#), [740](#), [14209](#), [14210](#), [14224](#), [14224](#)  
 \\_\_str\_convert:nnnn ..  
     ..... [740](#), [14224](#), [14228](#), [14233](#)  
 \\_\_str\_convert:NNnNN ..  
     ..... [14206](#), [14211](#), [14214](#)  
 \\_\_str\_convert:nNNnnn .. [14166](#),  
     ..... [14167](#), [14169](#), [14174](#), [14183](#), [14188](#)  
 \\_\_str\_convert:wwnnn ..  
     ..... [739](#), [14193](#), [14198](#), [14206](#), [14206](#)  
 \\_\_str\_convert\_decode\_ : ..  
     ..... [14197](#), [14329](#), [14329](#)  
 \\_\_str\_convert\_decode\_clist: ..  
     ..... [14369](#), [14369](#)  
 \\_\_str\_convert\_decode\_eight\_-  
     bit:n .. [14390](#), [14434](#), [14434](#)  
 \\_\_str\_convert\_decode\_utf16: .. [15056](#)  
 \\_\_str\_convert\_decode\_utf16be: .. [15056](#)  
 \\_\_str\_convert\_decode\_utf16le: .. [15056](#)  
 \\_\_str\_convert\_decode\_utf32: .. [15214](#)  
 \\_\_str\_convert\_decode\_utf32be: .. [15214](#)  
 \\_\_str\_convert\_decode\_utf32le: .. [15214](#)  
 \\_\_str\_convert\_decode\_utf8: .. [14875](#)  
 \\_\_str\_convert\_encode\_ : ..  
     ..... [14202](#), [14333](#), [14339](#), [14345](#)  
 \\_\_str\_convert\_encode\_clist: ...  
     ..... [14380](#), [14380](#)  
 \\_\_str\_convert\_encode\_eight\_-  
     bit:n .. [14392](#), [14461](#), [14462](#)  
 \\_\_str\_convert\_encode\_utf16: .. [14971](#)  
 \\_\_str\_convert\_encode\_utf16be: .. [14971](#)  
 \\_\_str\_convert\_encode\_utf16le: .. [14971](#)  
 \\_\_str\_convert\_encode\_utf32: .. [15154](#)  
 \\_\_str\_convert\_encode\_utf32be: .. [15154](#)  
 \\_\_str\_convert\_encode\_utf32le: .. [15154](#)  
 \\_\_str\_convert\_encode\_utf8: .. [14800](#)  
 \\_\_str\_convert\_escape\_ : ..  
     ..... [14327](#), [14327](#), [14328](#)  
 \\_\_str\_convert\_escape\_bytes: ...  
     ..... [14327](#), [14328](#)  
 \\_\_str\_convert\_escape\_hex: ...  
     ..... [14720](#), [14720](#)  
 \\_\_str\_convert\_escape\_name: ...  
     ..... [754](#), [14724](#), [14726](#)  
 \\_\_str\_convert\_escape\_string: ...  
     ..... [14747](#), [14749](#)  
 \\_\_str\_convert\_escape\_url: ...  
     ..... [14779](#), [14779](#)  
 \\_\_str\_convert\_gmap:N ..  
     ..... [14124](#), [14124](#), [14330](#),  
         ..... [14442](#), [14721](#), [14727](#), [14750](#), [14780](#)  
 \\_\_str\_convert\_gmap\_internal:N ..  
     ..... [14140](#),  
         ..... [14140](#), [14340](#), [14348](#), [14382](#), [14471](#),  
             ..... [14801](#), [14984](#), [15156](#), [15160](#), [15162](#)  
 \\_\_str\_convert\_gmap\_internal\_-  
     loop:Nw .. [14140](#)  
 \\_\_str\_convert\_gmap\_internal\_-  
     loop:Nww .. [14144](#), [14150](#), [14154](#)  
 \\_\_str\_convert\_gmap\_loop:NN ..  
     ..... [14124](#), [14128](#), [14134](#), [14138](#)  
 \\_\_str\_convert\_lowercase\_-  
     alphanum:n .. [14229](#), [14261](#), [14261](#)  
 \\_\_str\_convert\_lowercase\_-  
     alphanum\_loop:N ..  
         ..... [14261](#), [14263](#), [14267](#), [14285](#)  
 \\_\_str\_convert\_pdfname:n ..  
     ..... [15282](#), [15285](#), [15291](#), [15318](#)  
 \\_\_str\_convert\_pdfname\_bytes:n ..  
     ..... [15282](#), [15294](#), [15297](#)  
 \\_\_str\_convert\_pdfname\_bytes\_-  
     aux:n .. [15282](#), [15299](#), [15302](#)  
 \\_\_str\_convert\_pdfname\_bytes\_-  
     aux:nnm ..  
         ..... [15282](#)  
 \\_\_str\_convert\_pdfname\_bytes\_-  
     aux:nnnm .. [15303](#), [15304](#)  
 \\_\_str\_convert\_unescape\_ : ..  
     ..... [14311](#), [14317](#), [14325](#), [14326](#)

\\_\_str\_convert\_unescape\_bytes: . . . . .  
     . . . . .  
 \\_\_str\_convert\_unescape\_hex: . . . . .  
     . . . . .  
 \\_\_str\_convert\_unescape\_name: . . . . .  
     . . . . .  
 \\_\_str\_convert\_unescape\_string: . . . . .  
     . . . . .  
 \\_\_str\_convert\_unescape\_url: . . . . .  
 \\_\_str\_count:n . . . . .  
     . . . . .  
 \\_\_str\_count\_aux:n . . . . .  
     . . . . .  
 \\_\_str\_count\_loop:NNNNNNNNN . . . . .  
     . . . . .  
 \\_\_str\_count\_spaces\_loop:w . . . . .  
     . . . . .  
 \\_\_str\_declare\_eight\_bit\_- aux:NNnn . . . . .  
     . . . . .  
 \\_\_str\_declare\_eight\_bit\_- encoding:nnnn . . . . .  
     . . . . .  
 \\_\_str\_declare\_eight\_bit\_loop:Nn . . . . .  
     . . . . .  
 \\_\_str\_declare\_eight\_bit\_- loop:Nnn . . . . .  
     . . . . .  
 \\_\_str\_decode\_clist\_char:n . . . . .  
     . . . . .  
 \\_\_str\_decode\_eight\_bit\_aux:n . . . . .  
     . . . . .  
 \\_\_str\_decode\_eight\_bit\_aux:Nn . . . . .  
     . . . . .  
 \\_\_str\_decode\_native\_char:N . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_viii\_aux:wNnnwN . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_viii\_continuation:wwN . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_viii\_end: . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_viii\_overflow:w . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_viii\_start:N . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xvi:Nw . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xvi\_bom>NN . . . . .  
     . . . . .

\\_\_str\_decode\_utf\_xvi\_error:nNN . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xvi\_extra:Nw . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xvi\_pair>NN . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xvi\_pair\_- end:Nw . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xvi\_quad:NNwNN . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xxxii:Nw . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xxxii\_bom>NNNN . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xxxii\_end:w . . . . .  
     . . . . .  
 \\_\_str\_decode\_utf\_xxxii\_loop:NNNN . . . . .  
     . . . . .  
 \\_\_str\_encode\_clist\_char:n . . . . .  
     . . . . .  
 \\_\_str\_encode\_eight\_bit\_aux:Nm . . . . .  
     . . . . .  
 \\_\_str\_encode\_eight\_bit\_aux:nnN . . . . .  
     . . . . .  
 \\_\_str\_encode\_native\_char:n . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_vii\_loop:wwnw . . . . .  
 \\_\_str\_encode\_utf\_viii\_char:n . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_viii\_aux:N . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_viii\_be:nn . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_xvi\_char:n . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_xvi\_be:n . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_xxxii\_be:-aux:nn . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_xxxii\_le:n . . . . .  
     . . . . .  
 \\_\_str\_encode\_utf\_xxxii\_le\_-aux:nn . . . . .  
     . . . . .  
 \l\_\_str\_end\_flag . . . . .  
 \g\_\_str\_error\_bool . . . . .  
 \\_\_str\_escape\_hex\_char:N . . . . .  
     . . . . .  
 \\_\_str\_escape\_name\_char:n . . . . .  
     . . . . .

```

\c__str_escape_name_not_str . . .
    ..... 753, 14724
\c__str_escape_name_str .. 753, 14724
\__str_escape_string_char:N . .
    ..... 14747, 14750, 14751
\c__str_escape_string_str . . . 14747
\__str_escape_url_char:n . .
    ..... 14779, 14780, 14781
\l__str_extra_flag . . . 14822, 15005
\__str_filter_bytes:n . . . 14287,
    14293, 14310, 14321, 14602, 14664
\__str_filter_bytes_aux:N . .
    ..... 14287, 14295, 14299, 14307
\__str_head:w 726, 13806, 13810, 13814
\__str_hexadecimal_use:N . . . 14061
\__str_hexadecimal_use:NTF . . .
    750, 14061, 14556, 14566, 14605, 14607
\__str_if_contains_char:Nn . . . 14029
\__str_if_contains_char:nn . . . 14038
\__str_if_contains_char:NnTF . .
    ..... 14029, 14736, 14742, 14755
\__str_if_contains_char:nnTF . .
    ..... 734, 14029, 14789, 14795
\__str_if_contains_char_aux:nn . .
    ..... 14029, 14031, 14036
\__str_if_contains_char_auxi:Nn . .
    .. 14029, 14037, 14040, 14044, 14049
\__str_if_contains_char_true: . .
    ..... 14029, 14047, 14051
\__str_if_eq:nn 716, 13376, 13376,
    13380, 13386, 13391, 13398, 13403
\__str_if_escape_name:n . . . 14733
\__str_if_escape_name:nTF . . .
    ..... 14724, 14730
\__str_if_escape_string:N . . . 14767
\__str_if_escape_string:NTF . . .
    ..... 14747, 14753
\__str_if_escape_url:n . . . 14786
\__str_if_escape_url:nTF 14779, 14783
\__str_if_flag_error:nnn . .
    ..... 737, 738, 14156, 14156,
    14175, 14184, 14322, 14349, 14443,
    14472, 14550, 14596, 14597, 14655,
    14656, 14888, 14985, 15088, 15245
\__str_if_flag_no_error:nnn . .
    .. 737, 14156, 14162, 14175, 14184
\__str_if_flag_times:nTF . 14164,
    14164, 14831, 14832, 14833, 14834,
    15020, 15021, 15022, 15192, 15193
\__str_if_recursion_tail_-
    break:NN 13261, 13261, 13527, 13545
\__str_if_recursion_tail_stop_-
    do:Nn . . . 13261, 13262, 13872
\l__str_internal_t1 . . .
    ..... 740, 13992, 14081,
    14082, 14084, 14237, 14238, 14239,
    14241, 14245, 14249, 14256, 14388
\__str_item:nn . . .
    .... 721, 13610, 13616, 13621, 13622
\__str_item:w 721, 13610, 13624, 13629
\__str_map_function:nn . . .
    718, 13487, 13490, 13499, 13504, 13558
\__str_map_function:w . . .
    718, 13487, 13489, 13497, 13498, 13558
\__str_map_inline>NN . . .
    ..... 13487, 13514, 13525, 13529
\__str_map_variable:Nnn . . .
    ..... 13487, 13535, 13543, 13548
\c__str_max_byte_int .. 13996, 14354
\l__str_missing_flag .. 14822, 15005
\l__str_modulo_int . . . 14461
\__str_octal_use:N . . . 14053
\__str_octal_use:NTF . . .
    734, 735, 14053, 14667, 14669, 14671
\__str_output_byte:n . . . 765,
    14092, 14092, 14121, 14122, 14282,
    14487, 14814, 14820, 15172, 15181
\__str_output_byte:w . . .
    ..... 750, 14092, 14093,
    14094, 14543, 14569, 14604, 14666
\__str_output_byte_pair:nnN . .
    ..... 14108, 14110, 14115, 14118
\__str_output_byte_pair_be:n . .
    .. 14108, 14108, 14973, 14977, 15171
\__str_output_byte_pair_le:n . .
    ..... 14108, 14113, 14979, 15182
\__str_output_end: . . .
    ..... 750, 14092, 14093, 14106,
    14548, 14568, 14618, 14700, 14704
\__str_output_hexadecimal:n . .
    .. 14092, 14100, 14723, 14731,
    14784, 15306, 15307, 15310, 15313
\l__str_overflow_flag . . . 14822
\l__str_overlong_flag . . . 14822
\__str_range:nnn . . .
    ..... 13671, 13677, 13682, 13683
\__str_range:nnw . 13671, 13693, 13697
\__str_range:w .. 13671, 13685, 13691
\__str_range_normalize:nn . .
    ..... 13694, 13695, 13703, 13703
\__str_replace:NNNnn . . . 13314,
    13315, 13317, 13319, 13321, 13326
\__str_replace_aux:NNNnnn . . .
    ..... 13314, 13335, 13341
\__str_replace_next:w . . . 13314,
    13319, 13321, 13343, 13346, 13353

```

```

\c__str_replacement_char_int . . .
    ..... 13995, 14456,
    14900, 14924, 14938, 14958, 14965,
    14995, 15147, 15256, 15261, 15277
\g__str_result_tl . . .
    .... 732, 736–738, 742, 744, 750,
    762, 765, 767, 13994, 14126, 14130,
    14142, 14146, 14192, 14204, 14320,
    14321, 14371, 14372, 14375, 14383,
    14541, 14545, 14590, 14592, 14643,
    14646, 14649, 14652, 14882, 14884,
    14974, 15057, 15059, 15063, 15082,
    15157, 15215, 15217, 15220, 15239
\__str_skip_end>NNNNNNNN . . .
    ..... 722, 13650, 13667, 13670
\__str_skip_end:w 13650, 13655, 13665
\__str_skip_exp_end:w . . .
    ..... 722, 724, 13637,
    13646, 13650, 13650, 13662, 13701
\__str_skip_loop:wNNNNNNNN . . .
    ..... 13650, 13653, 13660
\__str_tail_auxi:w 13821, 13825, 13829
\__str_tail_auxii:w . . .
    ..... 727, 13821, 13832, 13835
\__str_tmp:n . . .
    13264,
    13270, 13273, 13291, 13301, 13304
\__str_tmp:w . . .
    746, 750,
    760, 762, 767, 13992, 13992, 14436,
    14442, 14464, 14471, 14582, 14628,
    14630, 14635, 14660, 14983, 14990,
    14995, 14997, 15000, 15001, 15081,
    15096, 15101, 15112, 15115, 15121,
    15122, 15238, 15253, 15258, 15264
\__str_to_other_end:w . . .
    ..... 720, 13565, 13580, 13585
\__str_to_other_fast_end:w . . .
    ..... 13588, 13603, 13608
\__str_to_other_fast_loop:w . . .
    ..... 13590, 13599, 13606
\__str_to_other_loop:w . . .
    ..... 720, 13565, 13567, 13576, 13582
\__str_unescape_hex_auxi:N . . .
    .. 14536, 14544, 14553, 14560, 14569
\__str_unescape_hex_auxii:N . . .
    ..... 14536, 14557, 14563, 14573
\__str_unescape_name_loop:wNN . . .
    ..... 14582, 14629
\__str_unescape_string_loop:wNNN
    .. 14632, 14651, 14662, 14701, 14704
\__str_unescape_string_newlines:wN
    ..... 14632, 14645, 14705, 14709
\__str_unescape_string_repeat>NNNNNN
    .. 14632, 14676, 14678, 14680, 14703
\__str_unescape_url_loop:wNN . . .
    ..... 14582, 14631
\__str_use_i_delimit_by_s_-stop:nw . . .
    726, 13257, 13258,
    13636, 13645, 13764, 13815, 13818
\__str_use_none_delimit_by_s_-stop:w . . .
    ..... 13257,
    13257, 13348, 13634, 13643, 13802,
    14152, 14424, 14430, 14815, 14907
\strcmp ..... 22
\string ..... 456
\sum ..... 1359
\suppressfontnotfounderror ..... 713
\suppressifcsnameerror ..... 900
\suppresslongerror ..... 901
\suppressmathparerror ..... 902
\suppressoutererror ..... 903
\suppressprimitiveerror ..... 904
\synctex ..... 693
sys commands:
\c_sys_backend_str ... 76, 8872, 8944
\c_sys_day_int ..... 72, 9070
\c_sys_engine_exec_str . . .
    ..... 73, 586, 8818, 11593
\c_sys_engine_format_str . . .
    ..... 73, 586, 8818, 11594
\c_sys_engine_str . . .
    ..... 73, 586, 661, 8801, 37095
\c_sys_engine_version_str 311, 37093
\sys_ensure_backend: .. 75, 8942, 8942
\sys_everyjob: ..... 9060, 9060, 9190
\sys_finalise: .. 76, 8874, 9188, 9188
\sys_get_shell:nnN 74, 8956, 8956, 8961
\sys_get_shell:nnNTF . . .
    ..... 74, 307, 8956, 8958
\sys_gset_rand_seed:n . . .
    ..... 74, 257, 9116, 9120, 9124, 9127
\c_sys_hour_int ..... 72, 9070
\sys_if_engine_luatex:TF .... 73,
    98, 2719, 8801, 8826, 8851, 8910,
    8920, 8921, 8996, 9018, 9050, 9142,
    9169, 10240, 11018, 11251, 11413,
    11682, 11731, 18905, 19305, 28916,
    28954, 28993, 29006, 29032, 29100,
    29124, 29161, 37537, 37539, 37541
\sys_if_engine_luatex_p: . . .
    ..... 73, 8801, 10404, 14017,
    14289, 14313, 14335, 14505, 15288,
    19023, 29211, 29237, 29299, 30018,
    30063, 31059, 32545, 32630, 37535
\sys_if_engine_pdftex:TF 73, 8801,
    8822, 8846, 9221, 37551, 37553, 37555
\sys_if_engine_pdftex_p: . . .
    ..... 73, 8801, 8860, 37549

```

\sys\_if\_engine\_ptex:TF . . . . . 73,  
   8801, 8824, 8849, 30041, 30108, 31024  
\sys\_if\_engine\_ptex\_p: 73, 3813, 8801  
\sys\_if\_engine\_uptex:TF . . . . .  
   . . . . . 73, 8801, 8825, 8850  
\sys\_if\_engine\_uptex\_p: 73, 3814, 8801  
\sys\_if\_engine\_xetex:TF . . . . .  
   . . . . . 6, 73, 2718, 8801, 8823, 8848,  
   8891, 9216, 18906, 37625, 37627, 37629  
\sys\_if\_engine\_xetex\_p: . . . . .  
   . . . . . 73, 8801, 9081, 14018,  
   14290, 14314, 14336, 14506, 15289,  
   19024, 29211, 29237, 29300, 30019,  
   30064, 31060, 32546, 32631, 37623  
\sys\_if\_output\_dvi:TF . . . . . 73, 9197  
\sys\_if\_output\_dvi\_p: . . . . . 73, 9197  
\sys\_if\_output\_pdf:TF . . . . .  
   . . . . . 73, 8906, 9197, 9219  
\sys\_if\_output\_pdf\_p: . . . . . 73, 9197  
\sys\_if\_platform\_unix:TF . . . . .  
   . . . . . 74, 8872, 11700  
\sys\_if\_platform\_unix\_p: . . . . .  
   . . . . . 74, 8872, 11700  
\sys\_if\_platform\_windows:TF . . . . .  
   . . . . . 74, 8872, 11700  
\sys\_if\_platform\_windows\_p: . . . . .  
   . . . . . 74, 8872, 11700  
\sys\_if\_rand\_exist:TF . . . . . 311, 587,  
   8870, 9104, 9118, 22460, 28444, 28468  
\sys\_if\_rand\_exist\_p: . . . . . 311, 8870  
\sys\_if\_shell: . . . . . 75  
\sys\_if\_shell:TF 75, 8963, 9177, 37149  
\sys\_if\_shell\_p: . . . . . 75, 9177  
\sys\_if\_shell\_restricted:TF 75, 9177  
\sys\_if\_shell\_restricted\_p: 75, 9177  
\sys\_if\_shell\_unrestricted:TF . . . . .  
   . . . . . 75, 9177  
\sys\_if\_shell\_unrestricted\_p: . . . . .  
   . . . . . 75, 9177  
\sys\_if\_timer\_exist:TF . . . . . 9131  
\sys\_if\_timer\_exist\_p: . . . . . 9131  
\c\_sys\_jobname\_str . . . . .  
   . . . . . 72, 93, 595, 9068, 37435  
\sys\_load\_backend:n . . . . .  
   . . . . . 75, 76, 8872, 8872, 8945  
\sys\_load\_debug: . . . . . 76, 8948, 8948  
\sys\_load\_deprecation: . . . . . 37658, 37659  
\c\_sys\_minute\_int . . . . . 72, 9070  
\c\_sys\_month\_int . . . . . 72, 9070  
\c\_sys\_output\_str . . . . . 73, 9197  
\c\_sys\_platform\_str . . . . .  
   . . . . . 74, 8872, 11682, 11703  
\sys\_rand\_seed: . . . . .  
   . . . . . 74, 148, 257, 9102, 9105, 9107, 9111  
\c\_sys\_shell\_escape\_int . . . . .  
   . . . . . 75, 9165, 9180, 9182, 9184  
\sys\_shell\_now:n . . . . .  
   . . . . . 75, 8998, 9020, 9024, 9027  
\sys\_shell\_shipout:n . . . . .  
   . . . . . 75, 9029, 9052, 9056, 9059  
\sys\_timer: . . . . .  
   . . . . . 73, 9131, 9144, 9150, 9154, 9158  
\c\_sys\_year\_int . . . . . 72, 9070  
sys internal commands:  
\g\_\_sys\_backend\_tl . . . . .  
   . . . . . 8882, 8883, 8884, 9211  
\\_\_sys\_const:nn . . . . .  
   . . . . . 8785, 8785, 8815, 8870,  
   9163, 9179, 9181, 9183, 9206, 9208  
\g\_\_sys\_debug\_bool . . . . . 8947, 8950, 8952  
\\_\_sys\_elapsedtime: . . . . . 9131, 9145, 9164  
\\_\_sys\_everyjob:n . . . . . 9060, 9065, 9068,  
   9070, 9102, 9116, 9165, 9177, 9186  
\g\_\_sys\_everyjob\_tl . . . . . 9060  
\\_\_sys\_finalise:n . . . . .  
   . . . . . 9188, 9194, 9197, 9212, 9229  
\g\_\_sys\_finalise\_tl . . . . . 9188  
\\_\_sys\_get:nnN . . . . . 8956, 8964, 8967  
\\_\_sys\_get\_do:Nw . . . . . 8956, 8981, 8990  
\l\_\_sys\_internal\_tl . . . . . 8954  
\\_\_sys\_load\_backend\_check:N . . . . .  
   . . . . . 8872, 8883, 8889  
\c\_\_sys\_marker\_tl . . . . . 8955, 8979, 8991  
\\_\_sys\_shell\_now:n . . . . . 8998, 9021  
\\_\_sys\_shell\_shipout:n . . . . . 9029, 9053  
\c\_\_sys\_shell\_stream\_int . . . . .  
   . . . . . 8996, 9025, 9057  
\\_\_sys\_tmp:w . . . . .  
   . . . . . 9073, 9094, 9096, 9097, 9098, 9099  
syst commands:  
\c\_syst\_catcodes\_n . . . . . 29167, 29171  
\c\_syst\_last\_allocated\_toks . . . . . 3402  
T  
\T . . . . . 86  
\t . . . . . 30140, 32670  
\tabskip . . . . . 457  
>tagcode . . . . . 694  
tan . . . . . 255  
tand . . . . . 255  
\state . . . . . 1162  
\tbaselineshift . . . . . 1163  
\c\_ten . . . . . 37458  
TEX and L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> commands:  
\@ . . . . . 13959  
\@##@hyph . . . . . 342  
\@end . . . . . 1199, 1200  
\@hyph . . . . . 1203, 1206

\@@input .....	1201	\dimen .....	879
\@@italiccorr .....	1207	\dimendef .....	879
\@@shipout .....	1209, 1210	\dimexpr .....	1290
\@@tracingfonts .....	343, 1245	\directlua .....	98
\@@underline .....	1208	\dp .....	275, 1022, 1023
\@addtofilelist .....	11409	\dup@shipout .....	1219
\@changed@cmd .....	30537, 32452	\e@alloc@ccodetable@count .....	29165
\@classoptionslist ..	9231, 9233, 9235	\e@alloc@top .....	419, 3388
\@current@cmd .....	30536, 32451	\edef .....	3, 6, 676
\@currnamestack .....	642, 10886, 10888, 10889	\end .....	342, 605
\@expl@finalise@setup@@ .....	29202, 29203, 30172, 30174, 31885, 31887, 36574, 36576	\endcsname .....	21
\@expl@luadata@bytecode .....	36	\endinput .....	81
\@filelist .....	97, 643, 656, 659, 11408, 11508, 11511, 11520, 11525	\endlinechar .....	87, 118, 119, 682, 683, 880, 1194, 1195, 1197
\@firstofone .....	23	\endtemplate .....	71, 583
\@firstoftwo .....	24, 362	\errhelp .....	601
\@gobbletwo .....	25	\errmessage .....	601, 602
\@gobble .....	25	\errorcontextlines .....	352, 602, 710, 1293
\@kernel@after@begindocument .....	30176, 31889, 36578	\escapechar ...	108, 371, 383, 459, 634
\@kernel@before@begindocument .....	36681, 36683	\everyeof .....	683
\@protected@testopt .....	1234, 30524	\everyjob .....	592
\@secondoftwo .....	24, 362	\everypar .....	28, 193, 386, 404
\@tempa .....	104, 106, 1217, 1231, 1234	\expandafter .....	38, 40
\@tfor .....	342, 1217	\expanded .....	6, 25, 33, 35, 388, 391, 398, 400, 405, 411, 682, 699
\@uclclist .....	1265, 31914	\fi .....	196
\@unexpandable@protect .....	1021	\firstmark .....	406, 880
\@unusedoptionlist .....	9250	\fmtname .....	73
\active@prefix .....	30386	\font .....	196, 878
\afterassignment .....	453, 454, 553	\fontdimen .....	60, 238, 969–972
\AtBeginDocument .....	342	\frozen@everydisplay .....	1204
\botmark .....	880	\frozen@everymath .....	1205
\box .....	281	\futurelet .....	434, 437, 438, 449, 453, 583, 885, 888
\catcodetable .....	1194, 1197, 1200	\global .....	322
\char .....	197	\GPTorg@shipout .....	1220
\chardef .....	190, 573, 576, 821, 1222	\halign .....	71, 386, 583, 871
\conditionally@traceoff .....	635, 9648, 10607	\hskip .....	218
\conditionally@traceon .....	9666	\ht .....	275, 1022, 1023
\copy .....	274	\hyphen .....	880
\count .....	197, 419	\hyphenchar .....	969
\cr .....	583	\ifcase .....	168
\CROP@shipout .....	1218	\ifdim .....	221
\csname .....	21, 350, 644, 645	\ifeof .....	93
\csstring .....	372	\iffalse .....	65, 674
\currentgrouplevel ..	383, 1196, 1393	\ifhbox .....	284
\currentgroupype .....	383, 1393	\ifmmode .....	674
\def .....	197	\ifnum .....	168
\detokenize .....	108	\ifodd .....	169, 890
\development@branch@name ..	11596, 11597	\iftrue .....	65, 674
		\ifvbox .....	284
		\ifvoid .....	284
		\ifx .....	27, 319
		\indent .....	386

\infty ..... 250  
 \input ..... 342  
 \input@path .....  
     . 94, 648, 11047, 11049, 11147, 11149  
 \italiccorr ..... 880  
 \jobname ..... 72, 592  
 \lastnamedcs ..... 374  
 \lccode ..... 437, 441, 839  
 \leavevmode ..... 28  
 \let ..... 322  
 \letcharcode ..... 868  
 \LL@shipout ..... 1221  
 \loctoks ..... 419  
 \long ..... 5, 197, 399, 710  
 \lower ..... 1391  
 \lowercase ..... 452, 541, 542  
 \luaescapestring ..... 99  
 \makeatletter ..... 9  
 \mathchar ..... 197  
 \mathchardef ..... 190, 821, 1222  
 \mathop ..... 1357  
 \mathord ..... 295  
 \meaning ..... 20,  
     187, 196, 197, 436, 878, 879, 888, 890  
 \mem@oldshipout ..... 1222  
 \message ..... 33  
 \newcatcodetable ..... 1194  
 \newif ..... 65, 101  
 \newlinechar ..... 118,  
     119, 352, 374, 602, 632, 682, 683, 710  
 \newread ..... 623  
 \newtoks ..... 43, 430, 459  
 \newwrite ..... 629  
 \noexpand ..... 38, 196, 398–401  
 \nullfont ..... 880, 881  
 \number ..... 168, 818, 1077  
 \numexpr ..... 351, 402  
 \opem@shipout ..... 1223  
 \or ..... 168  
 \outer ..... 197, 319, 434,  
     450, 451, 623, 629, 870, 890, 1408, 1410  
 \parindent ..... 28  
 \pdfescapehex ..... 748  
 \pdfescapename ..... 135, 748  
 \pdfescapestring ..... 135, 748  
 \pdffilesize ..... 647, 648  
 \pdfmapfile ..... 344  
 \pdfmapline ..... 344  
 \pdfstrcmp ..... 316, 317, 319, 333  
 \pdfuniformdeviate ..... 257  
 \pgfpages@originalshipout ..... 1224  
 \pi ..... 250  
 \pr@shipout ..... 1225  
 \primitive ..... 342, 398–400, 593  
 \protect ..... 636, 1020, 1021, 1278  
 \protected ..... 197, 399, 710  
 \protected@edef ..... 1227  
 \ProvidesClass ..... 9  
 \ProvidesFile ..... 9  
 \ProvidesPackage ..... 9  
 \quitvmode ..... 386  
 \read ..... 87, 627  
 \readline ..... 88, 627  
 \relax . 27, 196, 319, 367, 372, 383,  
     451, 576, 644, 841, 977, 979, 1004, 1036  
 \RequirePackage ..... 10, 319, 642  
 \reserveinserts ..... 319  
 \romannumeral ..... 40, 977, 1221, 1229  
 \savecatcodetable ..... 1196  
 \scantokens ..... 119, 137, 647, 681  
 \shipout ..... 342  
 \show ..... 20, 110, 383  
 \showbox ..... 1293  
 \showgroups ..... 14, 384  
 \showthe ..... 383, 838, 916, 920, 922  
 \showtokens ..... 110, 607, 710  
 \sin ..... 250  
 \skip ..... 442, 443  
 \space ..... 880  
 \splitbotmark ..... 880  
 \splitfirstmark ..... 880  
 \SS ..... 1281  
 \strcmp ..... 316, 333  
 \string ..... 187, 437, 438, 440  
 \tenrm ..... 196  
 \the ..... 159, 196, 213, 217,  
     220, 390, 398–400, 402, 403, 818, 1290  
 \toks ..... 43, 148, 168, 402–404, 417–  
     425, 430, 436, 438, 440, 441, 443,  
     445, 459, 460, 510, 517, 518, 522,  
     523, 533, 541, 549, 561, 562, 571, 803  
 \toks@ ..... 404  
 \toksdef ..... 430  
 \topmark ..... 197, 880  
 \tracingfonts ..... 343  
 \tracingnesting ..... 646, 681  
 \tracingonline ..... 1293  
 \typeout ..... 636  
 \Ucharcat ..... 870  
 \unexpanded ..... 39, 109, 110, 115, 116,  
     146, 151, 152, 175, 178–181, 202,  
     309, 312, 398–401, 676, 699, 700, 824  
 \unhbox ..... 281  
 \unhcopy ..... 278  
 \uniformdeviate ..... 257  
 \unless ..... 27  
 \unvbox ..... 281  
 \unvcopy ..... 280

```

\uppercase ..... 541
\usepackage ..... 642
\UTFviii@four@octets ..... 30385
\UTFviii@three@octets ..... 30384
\UTFviii@two@octets ..... 30383
\valign ..... 583
\value ..... 157
\verb ..... 119
\verso@orig@shipout ..... 1227
\vskip ..... 219
\vtop ..... 1313
\wd ..... 275, 1022, 1023
\write ..... 90, 632

tex commands:
\tex_above:D ..... 190
\tex_abovedisplayshortskip:D .. 191
\tex_abovedisplayskip:D ..... 192
\tex_abovewithdelims:D ..... 193
\tex Accent:D ..... 194
\tex adjdemerits:D ..... 195
\tex adjustspacing:D ..... 650, 912
\tex advance:D .. 196, 3494, 3501,
   3504, 3922, 3924, 3957, 3959, 5469,
   17336, 17338, 17340, 17342, 17348,
   17350, 17352, 17354, 20128, 20131,
   20137, 20140, 20468, 20470, 20474,
   20476, 20561, 20563, 20567, 20569
\tex_afterassignment:D .....
   ..... 197, 3863, 3906,
   4354, 4359, 7723, 7774, 7929, 19494
\tex_aftergroup:D ... 1199, 198, 1464
\tex_alignmark:D ..... 783, 1250
\tex_aligntab:D ..... 784, 1251
\tex_atop:D ..... 199
\tex_atopwithdelims:D ..... 200
\tex_attribute:D ..... 785, 1252
\tex_attributedef:D ..... 786, 1253
\tex_automaticdiscretionary:D .. 788
\tex_automatichyphenmode:D .... 789
\tex_automatichyphenpenalty:D .. 791
\tex_automospacing:D ..... 1111
\tex_autoxspacing:D ..... 1112
\tex_badness:D ..... 201
\tex_baselineskip:D ..... 202
\tex_batchmode:D ..... 203, 9530
\tex_begincname:D ..... 792
\tex_begingroup:D 204, 1212, 1314, 1459
\tex_beginL:D ..... 512
\tex_beginR:D ..... 513
\tex_belowdisplayshortskip:D .. 205
\tex_belowdisplayskip:D ..... 206
\tex_binoppenalty:D ..... 207
\tex_bodydir:D ..... 793, 1289, 1369
\tex_bodydirection:D ..... 794
\tex_botmark:D ..... 208
\tex_botmarks:D ..... 514
\tex_box:D ... 209, 32925, 32927, 32970
\tex_boxdir:D ..... 795, 1290
\tex_boxdirection:D ..... 796
\tex_boxmaxdepth:D ..... 210
\tex_breakafterdirmode:D ..... 797
\tex_brokenpenalty:D ..... 211
\tex_catcode:D ..... 212,
   2615, 7199, 8774, 12140, 18751, 18753
\tex_catcodetable:D ..... 798, 1254, 29010, 29017
\tex_char:D ..... 213
\tex_chardef:D ..... 360,
   214, 1452, 1481, 1483, 1791, 1792,
   8478, 8500, 8505, 10237, 10461,
   17311, 19375, 30204, 30206, 30208
\tex_cleaders:D ..... 215
\tex_clearmarks:D ..... 799, 1255
\tex_closein:D ..... 216, 10248
\tex_closeout:D ..... 217, 10471
\tex_clubpenalties:D ..... 515
\tex_clubpenalty:D ..... 218
\tex_copy:D ..... 219, 32919,
   32921, 32945, 32954, 32963, 32971
\tex_copyfont:D ..... 651, 913
\tex_count:D ..... 220, 3371, 3387,
   3395, 3396, 10185, 10187, 10416, 10418
\tex_countdef:D ..... 221
\tex_cr:D ..... 222
\tex_crampeddisplaystyle:D 800, 1256
\tex_crampedscriptscriptstyle:D .
   ..... 802, 1257
\tex_crampedscriptstyle:D . 803, 1259
\tex_crampedtextstyle:D .. 804, 1260
\tex_crcr:D ..... 223
\tex_creationdate:D ..... 773
\tex_curname:D ..... 224, 1446
\tex_csstream:D ..... 805
\tex_currentcjktoken:D ... 1113, 1174
\tex_currentgrouplevel:D .. 1199,
   516, 28999, 29079, 29088, 29095, 35802
\tex_currentgrouptype:D ..... 517
\tex_currentifbranch:D ..... 518
\tex_currentiflevel:D ..... 519
\tex_currentiftype:D ..... 520
\tex_currentspacingmode:D .... 1114
\tex_currentxspacingmode:D ... 1115
\tex_day:D ..... 225, 1321, 1325
\tex_deadcycles:D ..... 226
\tex_def:D ..... 227, 697, 698, 699, 1465, 1467,
   1469, 1470, 1491, 1493, 1494, 1495,
   1497, 1498, 1499, 1501, 1502, 1503

```

\tex\_defaulthyphenchar:D . . . . . 228  
 \tex\_defaultskewchar:D . . . . . 229  
 \tex\_delcode:D . . . . . 230  
 \tex\_delimiter:D . . . . . 231  
 \tex\_delimiterfactor:D . . . . . 232  
 \tex\_delimiter shortfall:D . . . . . 233  
 \tex\_detokenize:D . . . . . 521, 1455, 1457  
 \tex\_dimen:D . . . . . 234  
 \tex\_dimendef:D . . . . . 235  
 \tex\_dimexpr:D . . . . . 522, 20083, 32895  
 \tex\_directlua:D . . . . . 808,  
     1243, 1244, 8819, 9171, 11685, 11710  
 \tex\_disablecjktoken:D . . . . . 1175  
 \tex\_discretionary:D . . . . . 236  
 \tex\_discretionaryligaturemode:D 807  
 \tex\_dsinhibitglue:D . . . . . 1116  
 \tex\_displayindent:D . . . . . 237  
 \tex\_displaylimits:D . . . . . 238, 35448  
 \tex\_displaystyle:D . . . . . 239  
 \tex\_displaywidowpenalties:D . . . . . 523  
 \tex\_displaywidowpenalty:D . . . . . 240  
 \tex\_displaywidth:D . . . . . 241  
 \tex\_divide:D . . . . . 242, 3365, 5470  
 \tex\_doublehyphendemerits:D . . . . . 243  
 \tex\_dp:D . . . . . 244, 32935  
 \tex\_draftmode:D . . . . . 652, 914  
 \tex\_dtou:D . . . . . 1117  
 \tex\_dump:D . . . . . 245  
 \tex\_dviextension:D . . . . . 809  
 \tex\_dvifeedback:D . . . . . 810  
 \tex\_dvivariable:D . . . . . 811  
 \tex\_eachlinedepth:D . . . . . 653  
 \tex\_eachlineheight:D . . . . . 654  
 \tex\_edef:D . . . . . 246,  
     1213, 1214, 1230, 1315, 1316, 1321,  
     1322, 1327, 1328, 1333, 1334, 1492,  
     1496, 1500, 1504, 10699, 10757, 37401  
 \tex\_efcode:D . . . . . 685  
 \tex\_elapsedtime:D . . . . .  
     . . . . . 655, 774, 9148, 9151, 9164, 9857  
 \tex\_else:D . . . . .  
     . . . . . 247, 1216, 1242, 1318, 1324,  
     1330, 1336, 1432, 1484, 1487, 1529  
 \tex\_emergencystretch:D . . . . . 248  
 \tex\_enablecjktoken:D . . . . . 1176, 8807  
 \tex\_end:D . . . . . 249, 1200, 1352, 1902  
 \tex\_endcsname:D . . . . . 250, 1447  
 \tex\_endgroup:D . . . . .  
     . . . . . 251, 1198, 1238, 1339, 1460  
 \tex\_endinput:D 252, 9539, 11394, 11606  
 \tex\_endL:D . . . . . 524  
 \tex\_endlinechar:D . . . . .  
     . . . . . 158, 159, 173, 253, 10320,  
     10322, 10323, 12279, 12280, 12281,  
     12315, 28961, 28965, 28978, 29012,  
     29013, 29028, 29193, 29208, 29244  
 \tex\_endR:D . . . . . 525  
 \tex\_epTeXinputencoding:D . . . . . 1118  
 \tex\_epTeXversion:D 1119, 37113, 37136  
 \tex\_eqno:D . . . . . 254  
 \tex\_errhelp:D . . . . . 255, 9399  
 \tex\_errmessage:D . . . . . 256, 1894, 9419  
 \tex\_errorcontextlines:D . . . . .  
     . . . . . 257, 2221, 2228,  
     2229, 9414, 9433, 9628, 13212, 33034  
 \tex\_errorstopmode:D . . . . . 258  
 \tex\_escapechar:D . . . . . 645, 259, 2206,  
     3828, 3890, 3891, 4232, 4333, 4334,  
     4337, 4377, 4474, 10342, 10561,  
     10608, 10614, 14540, 14589, 14642  
 \tex\_eTeXglueshrinkorder:D . . . . . 812  
 \tex\_eTeXgluestretchorder:D . . . . . 813  
 \tex\_eTeXrevision:D . . . . . 526  
 \tex\_eTeXversion:D . . . . . 527  
 \tex\_etoksapp:D . . . . . 814  
 \tex\_etokspre:D . . . . . 815  
 \tex\_euc:D . . . . . 1120  
 \tex\_everycr:D . . . . . 260  
 \tex\_everydisplay:D . . . . . 261, 1204  
 \tex\_everyeof:D . . . . . 528,  
     4352, 4353, 8979, 11011, 12288, 12340  
 \tex\_everyhbox:D . . . . . 262  
 \tex\_everyjob:D . . . . . 263, 1346, 1353  
 \tex\_everymath:D . . . . . 264, 1205  
 \tex\_everypar:D . . . . . 265  
 \tex\_everyvbox:D . . . . . 266  
 \tex\_exceptionpenalty:D . . . . . 816  
 \tex\_exhyphenpenalty:D . . . . . 267  
 \tex\_expandafter:D . . . . .  
     268, 702, 1217, 1231, 1233, 1234, 1448  
 \tex\_expanded:D 411, 899, 818, 1362,  
     1528, 1529, 2298, 2301, 2368, 2371,  
     2404, 2410, 2501, 2504, 2525, 2528,  
     2593, 2596, 2624, 3101, 3141, 3149,  
     10070, 12800, 12804, 18913, 19856,  
     19866, 20601, 20605, 20759, 29368  
 \tex\_explicitdiscretionary:D . . . . . 819  
 \tex\_explicithyphenpenalty:D . . . . . 817  
 \tex\_fam:D . . . . . 269  
 \tex\_fi:D . . . . . 270, 703, 1202,  
     1211, 1235, 1237, 1246, 1247, 1248,  
     1306, 1308, 1309, 1313, 1320, 1326,  
     1332, 1338, 1344, 1347, 1350, 1363,  
     1371, 1376, 1433, 1489, 1490, 1531  
 \tex\_filedump:D . . . . .  
     . . . . . 712, 775, 1419, 11244, 11256, 11263  
 \tex\_filemoddate:D . . . . .  
     . . . . . 711, 776, 1415, 11365

\tex\_filesize:D .....  
     ..... 649, 650, 709, 777, 1402,  
     11030, 11106, 11138, 11263, 11290  
 \tex\_finalhyphendemerits:D ..... 271  
 \tex\_firstlineheight:D ..... 656  
 \tex\_firstmark:D ..... 272  
 \tex\_firstmarks:D ..... 529  
 \tex\_firstvalidlanguage:D ..... 820  
 \tex\_floatingpenalty:D ..... 273  
 \tex\_font:D ..... 274, 22264  
 \tex\_fontchardp:D ..... 530  
 \tex\_fontcharht:D ..... 531  
 \tex\_fontcharic:D ..... 532  
 \tex\_fontcharwd:D ..... 533  
 \tex\_fondimen:D ..... 275, 22256  
 \tex\_fontexpand:D ..... 657, 915  
 \tex\_fontid:D ..... 821, 1261  
 \tex\_fontname:D ..... 276  
 \tex\_fontsize:D ..... 658  
 \tex\_forcecjktoken:D ..... 1177  
 \tex\_formatname:D ..... 822, 1262  
 \tex\_futurelet:D .....  
     .... 277, 3836, 3894, 4339, 4355,  
     4360, 4364, 4425, 4437, 19489, 19491  
 \tex\_gdef:D 278, 1505, 1508, 1512, 1516  
 \tex\_gleaders:D ..... 828, 1263  
 \tex\_global:D .....  
     908, 179, 183, 279, 704, 1233, 1319,  
     1325, 1331, 1337, 1971, 1978, 8478,  
     8505, 10237, 10461, 12103, 12115,  
     17289, 17295, 17299, 17318, 17329,  
     17340, 17342, 17352, 17354, 17362,  
     19109, 19111, 19121, 19491, 20097,  
     20102, 20118, 20125, 20131, 20140,  
     20440, 20460, 20465, 20470, 20476,  
     20531, 20537, 20553, 20558, 20563,  
     20569, 22264, 32921, 32927, 33000,  
     33053, 33065, 33078, 33098, 33145,  
     33157, 33169, 33182, 33203, 33218  
 \tex\_globaldefs:D ..... 280  
 \tex\_glueexpr:D ..... 534, 20458,  
     20460, 20468, 20470, 20474, 20476,  
     20490, 20497, 20502, 20505, 28378  
 \tex\_glueshrink:D ..... 535  
 \tex\_glueshrinkorder:D ..... 536  
 \tex\_gluestretch:D ... 537, 4053, 4059  
 \tex\_gluestretchorder:D ..... 538  
 \tex\_gluetomu:D ..... 539  
 \tex\_halign:D ..... 281  
 \tex\_hangafter:D ..... 282  
 \tex\_hangindent:D ..... 283  
 \tex\_hbadness:D ..... 284  
 \tex\_hbox:D ..... 285, 33045, 33048,  
     33053, 33060, 33065, 33072, 33078,  
     33092, 33098, 33106, 33111, 34558  
 \tex\_hfi:D ..... 1121  
 \tex\_hfil:D ..... 286  
 \tex\_hfill:D ..... 287  
 \tex\_hfilneg:D ..... 288  
 \tex\_hfuzz:D ..... 289  
 \tex\_hjcode:D ..... 823  
 \tex\_hffset:D ..... 290, 1365  
 \tex\_holdinginserts:D ..... 291  
 \tex\_hpack:D ..... 824  
 \tex\_hrule:D ..... 292  
 \tex\_hsize:D .....  
     .... 293, 33708, 33733, 33734, 33784  
 \tex\_hskip:D ..... 294, 20500  
 \tex\_hss:D .....  
     295, 33115, 33117, 33119, 33562, 33571  
 \tex\_ht:D ..... 296, 32934  
 \tex\_hyphen:D ..... 189, 1206  
 \tex\_hyphenation:D ..... 297  
 \tex\_hyphenationbounds:D ..... 825  
 \tex\_hyphenationmin:D ..... 826  
 \tex\_hyphenchar:D ..... 298, 22257  
 \tex\_hyphenpenalty:D ..... 299  
 \tex\_hyphenpenaltymode:D ..... 827  
 \tex\_if:D ..... 300, 1435, 1436  
 \tex\_ifabsdim:D ..... 647, 916  
 \tex\_ifabsnum:D .....  
     .... 968, 648, 917, 22323, 22327  
 \tex\_ifcase:D ..... 301, 17180  
 \tex\_ifcat:D ..... 302, 1437  
 \tex\_ifcondition:D ..... 829  
 \tex\_ifcsname:D ..... 540, 1445  
 \tex\_ifdbox:D ..... 1122  
 \tex\_ifddir:D ..... 1123  
 \tex\_ifdefined:D ..... 541, 701,  
     1199, 1203, 1209, 1240, 1243, 1249,  
     1308, 1309, 1340, 1345, 1348, 1351,  
     1364, 1372, 1444, 1482, 1485, 1529  
 \tex\_ifdim:D ..... 303, 20082  
 \tex\_ifeof:D ..... 304, 10284  
 \tex\_iffalse:D ..... 305, 1430  
 \tex\_iffontchar:D ..... 542  
 \tex\_ifhbox:D ..... 306, 32982  
 \tex\_ifhmode:D ..... 307, 1441  
 \tex\_ifincsname:D ..... 686  
 \tex\_ifinner:D ..... 308, 1443  
 \tex\_ifjfont:D ..... 1124  
 \tex\_ifmbox:D ..... 1125  
 \tex\_ifmdir:D ..... 1126  
 \tex\_ifmmode:D ..... 309, 1440  
 \tex\_ifnum:D ..... 310, 1307, 1462  
 \tex\_ifodd:D .. 311, 1439, 8471, 17179

\tex\_ifprimitive:D ..... 649, 779  
 \tex\_iftbox:D ..... 1127  
 \tex\_iftdir:D ..... 1129  
 \tex\_iftfont:D ..... 1128  
 \tex\_iftrue:D ..... 312, 1429  
 \tex\_ifvbox:D ..... 313, 32983  
 \tex\_ifvmode:D ..... 314, 1442  
 \tex\_ifvoid:D ..... 315, 32984  
 \tex\_ifx:D ..... 316, 1215,  
     1232, 1317, 1323, 1329, 1335, 1438  
 \tex\_ifybox:D ..... 1130  
 \tex\_ifydir:D ..... 1131  
 \tex\_ignoredimensions:D ..... 659  
 \tex\_ignoreligaturesinfont:D .. 918  
 \tex\_ignorespaces:D ..... 317  
 \tex\_immediate:D .....  
     318, 1911, 1913, 10463, 10471, 10528  
 \tex\_immediateassigned:D ..... 830  
 \tex\_immediateassignment:D .... 831  
 \tex\_indent:D ..... 319, 2282  
 \tex\_inhibitglue:D ..... 1132  
 \tex\_inhibitxspcode:D ..... 1133  
 \tex\_initcatcodetable:D .....  
     832, 1264, 28926  
 \tex\_input:D .....  
     320, 1201, 1354, 8984, 11017, 11412  
 \tex\_inputlineno:D ... 321, 1909, 9337  
 \tex\_insert:D ..... 322  
 \tex\_insertht:D ..... 660, 919  
 \tex\_insertpenalties:D ..... 323  
 \tex\_interactionmode:D ..... 543,  
     2219, 2224, 2225, 33018, 33021, 33023  
 \tex\_interlinepenalties:D ..... 544  
 \tex\_interlinepenalty:D ..... 324  
 \tex\_italiccorrection:D .....  
     188, 1207, 1366  
 \tex\_jcharwidowpenalty:D ..... 1134  
 \tex\_jfam:D ..... 1135  
 \tex\_jfont:D ..... 1136  
 \tex\_jis:D ..... 1137  
 \tex\_jobname:D .....  
     325, 9069, 9187, 10877, 10878  
 \tex\_kanjiskip:D ..... 1138, 8805  
 \tex\_kansuji:D ..... 1139  
 \tex\_kansujichar:D ..... 1140  
 \tex\_kcatcode:D ..... 1141  
 \tex\_kchar:D ..... 1178  
 \tex\_kchardef:D ..... 1179  
 \tex\_kern:D ..... 326, 32899  
 \tex\_kuten:D ..... 1142, 1180  
 \tex\_language:D ..... 327, 1355  
 \tex\_lastbox:D ... 328, 32998, 33000  
 \tex\_lastkern:D ..... 329  
 \tex\_lastlinedepth:D ..... 661  
 \tex\_lastlinefit:D ..... 545  
 \tex\_lastnamedcs:D ..... 833  
 \tex\_lastnodechar:D ..... 1143  
 \tex\_lastnodefont:D ..... 1144  
 \tex\_lastnodesubtype:D ..... 1145  
 \tex\_lastnodestype:D ..... 546  
 \tex\_lastpenalty:D ..... 330  
 \tex\_lastskip:D ..... 331  
 \tex\_lastxpos:D ..... 662, 926  
 \tex\_lastypos:D ..... 663, 927  
 \tex\_latelua:D ..... 834, 1265, 11711  
 \tex\_lateluafunction:D ..... 835  
 \tex\_lccode:D .....  
     332, 3809, 3819, 3888, 3890,  
     3893, 3923, 4301, 7366, 7418, 13571,  
     13572, 13594, 13595, 18827, 18829  
 \tex\_leaders:D ..... 333  
 \tex\_left:D ..... 334, 1373  
 \tex\_leftghost:D ..... 836, 1291  
 \tex\_lefthyphenmin:D ..... 335  
 \tex\_leftmarginkern:D ..... 687  
 \tex\_leftskip:D ..... 336  
 \tex\_leqno:D ..... 337  
 \tex Let:D .... 1408, 180, 183, 338,  
     704, 1200, 1201, 1204, 1205, 1206,  
     1207, 1208, 1210, 1233, 1239, 1241,  
     1245, 1250, 1251, 1252, 1253, 1254,  
     1255, 1256, 1257, 1259, 1260, 1261,  
     1262, 1263, 1264, 1265, 1266, 1267,  
     1268, 1269, 1270, 1271, 1272, 1273,  
     1274, 1275, 1276, 1277, 1278, 1279,  
     1280, 1282, 1283, 1285, 1286, 1287,  
     1289, 1290, 1291, 1292, 1293, 1295,  
     1296, 1297, 1298, 1299, 1300, 1301,  
     1302, 1303, 1304, 1305, 1311, 1312,  
     1319, 1325, 1331, 1337, 1341, 1342,  
     1343, 1346, 1349, 1352, 1353, 1354,  
     1355, 1356, 1357, 1358, 1359, 1360,  
     1361, 1362, 1365, 1366, 1367, 1368,  
     1369, 1370, 1373, 1374, 1375, 1429,  
     1430, 1431, 1432, 1433, 1434, 1435,  
     1436, 1437, 1438, 1439, 1440, 1441,  
     1442, 1443, 1444, 1445, 1446, 1447,  
     1448, 1449, 1450, 1451, 1453, 1454,  
     1455, 1456, 1457, 1458, 1459, 1460,  
     1462, 1463, 1464, 1480, 1491, 1492,  
     1505, 1506, 1967, 3810, 3820, 4258,  
     4330, 12101, 12103, 12113, 12115,  
     19109, 19111, 19121, 37365, 37368  
 \tex\_letcharcode:D ..... 837  
 \tex\_letterspacefont:D ..... 688  
 \tex\_limits:D ..... 339, 35446  
 \tex\_linedir:D ..... 838  
 \tex\_linedirection:D ..... 839

\tex\_lineendmode:D ..... 1151  
 \tex\_linepenalty:D ..... 340  
 \tex\_lineskip:D ..... 341  
 \tex\_lineskiplimit:D ..... 342  
 \tex\_localbrokenpenalty:D . 840, 1292  
 \tex\_localinterlinepenalty:D ...  
       ..... 841, 1293  
 \tex\_localleftbox:D ..... 846, 1295  
 \tex\_localrightbox:D .... 847, 1296  
 \tex\_long:D ..... 343, 697, 698,  
       699, 1465, 1467, 1470, 1493, 1494,  
       1495, 1496, 1497, 1499, 1501, 1502,  
       1503, 1504, 1508, 1510, 1516, 1518  
 \tex\_looseness:D ..... 344  
 \tex\_lower:D ..... 345, 32981  
 \tex\_lowercase:D .....  
       .... 870, 1405, 346, 3810, 3820,  
       3889, 4302, 7367, 7419, 9406, 13573,  
       13596, 18858, 18960, 37306, 37609  
 \tex\_lpcode:D ..... 689  
 \tex\_luabytecode:D ..... 842  
 \tex\_luabytecodecall:D ..... 843  
 \tex\_luacopyinputnodes:D ..... 844  
 \tex\_luadef:D ..... 845  
 \tex\_luaescapestring:D .....  
       ..... 848, 1266, 11709  
 \tex\_luafunction:D ..... 849, 1267  
 \tex\_luafunctioncall:D ..... 850  
 \tex\_luatexbanner:D ..... 851  
 \tex\_luatexrevision:D .... 852, 37120  
 \tex\_luatexversion:D .....  
       .... 853, 1309, 1340, 1482, 8803,  
       10405, 13878, 17306, 17984, 37118  
 \tex\_mag:D ..... 347, 36691  
 \tex\_mapfile:D ..... 664, 1311  
 \tex\_mapline:D ..... 665, 1312  
 \tex\_mark:D ..... 348  
 \tex\_marks:D ..... 547  
 \tex\_mathaccent:D ..... 349  
 \tex\_mathbin:D ..... 350  
 \tex\_mathchar:D ..... 351  
 \tex\_mathchardef:D 360, 352, 1488,  
       17314, 17315, 30205, 30207, 30209  
 \tex\_mathchoice:D ..... 353  
 \tex\_mathclose:D ..... 354  
 \tex\_mathcode:D ... 355, 18821, 18823  
 \tex\_mathdefaultsmode:D ..... 854  
 \tex\_mathdelimitersmode:D .... 855  
 \tex\_mathdir:D ..... 856, 1297  
 \tex\_mathdirection:D ..... 857  
 \tex\_mathdisplayskipmode:D .... 858  
 \tex\_matheqdirmode:D ..... 859  
 \tex\_matheqnogapstep:D ..... 860  
 \tex\_mathinner:D ..... 356  
 \tex\_mathnolimitsmode:D ..... 861  
 \tex\_mathop:D ..... 357, 1356  
 \tex\_mathopen:D ..... 358  
 \tex\_mathoption:D ..... 862  
 \tex\_mathord:D ..... 359  
 \tex\_mathpenaltiesmode:D ..... 863  
 \tex\_mathpunct:D ..... 360  
 \tex\_mathrel:D ..... 361  
 \tex\_mathrulesfam:D ..... 864  
 \tex\_mathscriptboxmode:D ..... 866  
 \tex\_mathscriptcharchmode:D ..... 867  
 \tex\_mathscriptsmode:D ..... 865  
 \tex\_mathstyle:D ..... 868, 1268  
 \tex\_mathsurround:D ..... 362  
 \tex\_mathsurroundmode:D ..... 869  
 \tex\_mathsurroundskip:D ..... 870  
 \tex\_maxdeadcycles:D ..... 363  
 \tex\_maxdepth:D ..... 364  
 \tex\_mdfive:sum:D 710, 778, 1406, 11200  
 \tex\_meaning:D .. 365, 1214, 1231,  
       1315, 1321, 1327, 1333, 1453, 1454  
 \tex\_medmuskip:D ..... 366  
 \tex\_message:D ..... 367  
 \tex\_middle:D ..... 548, 1374  
 \tex\_mkern:D ..... 368  
 \tex\_month:D ... 369, 1327, 1331, 1357  
 \tex\_moveleft:D ..... 370, 32975  
 \tex\_moveright:D ..... 371, 32977  
 \tex\_mskip:D ..... 372  
 \tex\_muexpr:D .. 549, 20551, 20553,  
       20561, 20563, 20567, 20569, 20573  
 \tex\_multiply:D ..... 373  
 \tex\_muskip:D ..... 374  
 \tex\_muskipdef:D ..... 375  
 \tex\_mutoglue:D ..... 349, 550  
 \tex\_newlinechar:D .....  
       .... 376, 1893, 9412, 9626,  
       10527, 12281, 12307, 12311, 13210  
 \tex\_noalign:D ..... 377  
 \tex\_noautospacing:D ..... 1146  
 \tex\_noautoxspacing:D ..... 1147  
 \tex\_noboundary:D ..... 378  
 \tex\_noexpand:D ..... 379, 1449  
 \tex\_nohrule:D ..... 871  
 \tex\_noindent:D ..... 380  
 \tex\_nokerns:D ..... 872, 1269  
 \tex\_noligatures:D ..... 666  
 \tex\_noligs:D ..... 873, 1270  
 \tex\_nolimits:D ..... 381, 35447  
 \tex\_nonscript:D ..... 382  
 \tex\_nonstopmode:D ..... 383  
 \tex\_normaldeviate:D ..... 667, 928  
 \tex\_nospaces:D ..... 874  
 \tex\_novrule:D ..... 875

\tex_nulldelimiterspace:D . . . . .	384	\tex_partokennname:D . . . . .	1191
\tex_nullfont:D . . . . .	385, 19404	\tex_patterns:D . . . . .	411
\tex_number:D . . . . .	386, 17176, 33611	\tex_pausing:D . . . . .	412
\tex_numexpr:D . . . . .	551, 4472, 17177, 18974, 22528, 36985	\tex_pdfannot:D . . . . .	578
\tex_odelcode:D . . . . .	1183	\tex_pdfcatalog:D . . . . .	579
\tex_odelimiter:D . . . . .	1184	\tex_pdfcolorstack:D . . . . .	581
\tex_omathaccent:D . . . . .	1185	\tex_pdfcolorstackinit:D . . . . .	582
\tex_omathchar:D . . . . .	1186	\tex_pdfcompresslevel:D . . . . .	580
\tex_omathchardef:D . . . . .	1187, 1485, 1486, 17307, 17309, 17310	\tex_pdfcreationdate:D . . . . .	583
\tex_omathcode:D . . . . .	1188	\tex_pdfdecimaldigits:D . . . . .	584
\tex_omit:D . . . . .	387	\tex_pdfdest:D . . . . .	585
\tex_openin:D . . . . .	388, 10239	\tex_pdfdestmargin:D . . . . .	586
\tex_openout:D . . . . .	389, 10463	\tex_pfdendlink:D . . . . .	587
\tex_or:D . . . . .	390, 1431	\tex_pfdendthread:D . . . . .	588
\tex_oradical:D . . . . .	1189	\tex_pfdextension:D . . . . .	885
\tex_outer:D . . . . .	391, 1358, 37401	\tex_pdffeedback:D . . . . .	886
\tex_output:D . . . . .	392	\tex_pdffontattr:D . . . . .	589
\tex_outputbox:D . . . . .	876, 1271	\tex_pdffontname:D . . . . .	590
\tex_outputpenalty:D . . . . .	393	\tex_pdffontobjnum:D . . . . .	591
\tex_over:D . . . . .	394, 1359	\tex_pdfgamma:D . . . . .	592
\tex_overfullrule:D . . . . .	395	\tex_pgfentounicode:D . . . . .	595
\tex_overline:D . . . . .	396	\tex_pdfglyptounicode:D . . . . .	596
\tex_overwithdelims:D . . . . .	397	\tex_pdhorigin:D . . . . .	597
\tex_pagebottomoffset:D . . . . .	877, 1298	\tex_pdfimageapplygamma:D . . . . .	593
\tex_pagedepth:D . . . . .	398	\tex_pdfimagegamma:D . . . . .	594
\tex_pagedir:D . . . . .	878, 1299, 1370	\tex_pdfimagehicolor:D . . . . .	598
\tex_pagedirection:D . . . . .	879	\tex_pdfimageresolution:D . . . . .	599
\tex_pagediscards:D . . . . .	552	\tex_pdfincludechars:D . . . . .	600
\tex_pagefillstretch:D . . . . .	399	\tex_pdfinclusioncopyfonts:D . . . . .	601
\tex_pagefillstretch:D . . . . .	400	\tex_pdfinclusionerrorlevel:D . . . . .	603
\tex_pagefilstretch:D . . . . .	401	\tex_pdfinfo:D . . . . .	604
\tex_pagefiststretch:D . . . . .	1148	\tex_pdflastannot:D . . . . .	605
\tex_pagegoal:D . . . . .	402	\tex_pdflastlink:D . . . . .	606
\tex_pageheight:D . . . . .	668, 930, 1300	\tex_pdflastobj:D . . . . .	607
\tex_pageleftoffset:D . . . . .	880, 1272	\tex_pdflastxform:D . . . . .	608, 921
\tex_pagerightoffset:D . . . . .	881, 1301	\tex_pdflastximage:D . . . . .	609, 923
\tex_pageshrink:D . . . . .	403	\tex_pdflastximagecolordepth:D . . . . .	611
\tex_pagestretch:D . . . . .	404	\tex_pdflastximagepages:D . . . . .	612, 925
\tex_pagetopoffset:D . . . . .	882, 1273	\tex_pdflinkmargin:D . . . . .	613
\tex_pagetotal:D . . . . .	405	\tex_pdfliteral:D . . . . .	614
\tex_pagewidth:D . . . . .	669, 931, 1302	\tex_pdfmajorversion:D . . . . .	615
\tex_par:D . . . . .	406	\tex_pdfminorversion:D . . . . .	616
\tex_pardir:D . . . . .	883, 1303	\tex_pdfnames:D . . . . .	617
\tex_pardirection:D . . . . .	884	\tex_pdfobj:D . . . . .	618
\tex_parfillskip:D . . . . .	407	\tex_pdfobjcompresslevel:D . . . . .	619
\tex_parindent:D . . . . .	408	\tex_pdfoutline:D . . . . .	620
\tex_parshape:D . . . . .	409	\tex_pdfoutput:D . . . . .	586, 621, 929, 1349, 8847, 8853, 8861, 9202
\tex_parshapedimen:D . . . . .	553	\tex_pdfpageattr:D . . . . .	622
\tex_parshapeindent:D . . . . .	554	\tex_pdfpagebox:D . . . . .	624
\tex_parshapelengeth:D . . . . .	555	\tex_pdfpageref:D . . . . .	625
\tex_parskip:D . . . . .	410	\tex_pdfpageresources:D . . . . .	626
\tex_partokencontext:D . . . . .	1190	\tex_pdfpagesattr:D . . . . .	623, 627
		\tex_pdfsrefobj:D . . . . .	628

\tex\_pdfrefxform:D ..... 629, 935  
\tex\_pdfsimage:D ..... 630, 936  
\tex\_pdfrestore:D ..... 631  
\tex\_pfdretval:D ..... 632  
\tex\_pdfsave:D ..... 633  
\tex\_pdfsetmatrix:D ..... 634  
\tex\_pdfstartlink:D ..... 635  
\tex\_pdfstartthread:D ..... 636  
\tex\_pdfsuppressptexinfo:D .... 637  
\tex\_pdftexbanner:D ..... 682, 1341  
\tex\_pdftexrevision:D 683, 1342, 37101  
\tex\_pdftexversion:D .....  
... 345, 684, 1308, 1343, 8804, 37099  
\tex\_pdfthread:D ..... 638  
\tex\_pdfthreadmargin:D ..... 639  
\tex\_pdftrailer:D ..... 640  
\tex\_pduiniqueresname:D ..... 641  
\tex\_pdfvariable:D ..... 887  
\tex\_pdfvorigin:D ..... 642  
\tex\_pdxform:D ..... 643, 938  
\tex\_pdxformname:D ..... 644  
\tex\_pdximage:D ..... 645, 939  
\tex\_pdximagebbox:D ..... 646  
\tex\_penalty:D ..... 413  
\tex\_pkmode:D ..... 670  
\tex\_pkresolution:D ..... 671  
\tex\_postbreakpenalty:D ..... 1149  
\tex\_postdisplaypenalty:D .... 414  
\tex\_postexhyphenchar:D ... 888, 1274  
\tex\_posthyphenchar:D .... 889, 1275  
\tex\_prebinopenpenalty:D ..... 890  
\tex\_preibreakpenalty:D ..... 1150  
\tex\_predisplaydirection:D .... 556  
\tex\_predisplaygapfactor:D .... 891  
\tex\_predisplaypenalty:D .... 415  
\tex\_predisplaysize:D ..... 416  
\tex\_preehyphenchar:D .... 892, 1276  
\tex\_prehyphenchar:D .... 893, 1277  
\tex\_prerelpenalty:D ..... 894  
\tex\_pretolerance:D ..... 417  
\tex\_prevdepth:D ..... 418  
\tex\_prevgraf:D ..... 419  
\tex\_primitive:D .....  
... 400, 672, 780, 2675, 9078, 9088  
\tex\_protected:D .....  
... 557, 1493, 1495, 1497,  
1498, 1499, 1500, 1501, 1502, 1503,  
1504, 1512, 1514, 1516, 1518, 37401  
\tex\_protrudechars:D ..... 673, 932  
\tex\_ptexminorversion:D .....  
... 1152, 37110, 37129  
\tex\_ptexrevision:D 1153, 37111, 37130  
\tex\_ptexversion:D .....  
... 1154, 37105, 37108, 37124, 37127  
\tex\_pxdimen:D ..... 674, 933  
\tex\_quitvmode:D ..... 690  
\tex\_radical:D ..... 420  
\tex\_raise:D ..... 421, 32979  
\tex\_randomseed:D .... 675, 934, 9105  
\tex\_read:D ..... 422, 9531, 10304  
\tex\_readline:D ..... 558, 10321  
\tex\_readpapersizespecial:D .. 1155  
\tex\_relax:D .....  
... 349, 979, 423, 1458, 17178, 20084  
\tex\_relpriority:D ..... 424  
\tex\_resettimer:D ..... 676, 781  
\tex\_right:D ..... 425, 1375  
\tex\_rightghost:D ..... 895, 1304  
\tex\_righthyphenmin:D ..... 426  
\tex\_rightmarginkern:D ..... 691  
\tex\_rightsskip:D ..... 427  
\tex\_roman numeral:D 371, 397, 428,  
1451, 1463, 1796, 18870, 22530, 28999  
\tex\_rpcode:D ..... 692  
\tex\_savecatcodetable:D .....  
... 896, 1278, 28960, 29016  
\tex\_savepos:D ..... 677, 937  
\tex\_savinghyphcodes:D ..... 559  
\tex\_savingvdiscards:D ..... 560  
\tex\_scantextokens:D ..... 897, 1279  
\tex\_scantokens:D .. 561, 12293, 12354  
\tex\_scriptbaselineshiftfactor:D  
..... 1157  
\tex\_scriptfont:D ..... 429  
\tex\_scriptscripbaselineshiftfactor:D  
..... 1159  
\tex\_scriptscripfont:D ..... 430  
\tex\_scriptscripststyle:D ..... 431  
\tex\_scriptspace:D ..... 432  
\tex\_scriptstyle:D ..... 433  
\tex\_scrollmode:D ..... 434  
\tex\_setbox:D ..... 435,  
32919, 32921, 32925, 32927, 32945,  
32954, 32963, 32998, 33000, 33048,  
33053, 33060, 33065, 33072, 33078,  
33092, 33098, 33140, 33145, 33152,  
33157, 33164, 33169, 33176, 33182,  
33197, 33203, 33214, 33218, 34558  
\tex\_setfontid:D ..... 898  
\tex\_setlanguage:D ..... 436  
\tex\_setrandomseed:D .. 678, 940, 9121  
\tex\_sfcode:D .... 437, 18839, 18841  
\tex\_shapemode:D ..... 899  
\tex\_shellescape:D .... 679, 782, 9174  
\tex\_shipout:D ..... 438, 1210, 1234  
\tex\_show:D ..... 439  
\tex\_showbox:D ..... 440, 33035  
\tex\_showboxbreadth:D ... 441, 33031

\tex\_showboxdepth:D ..... 442, 33032  
\tex\_showgroups:D ..... 562, 2223  
\tex\_showifs:D ..... 563  
\tex\_showlists:D ..... 443  
\tex\_showmode:D ..... 1160  
\tex\_showstream:D ..... 1192  
\tex\_showthe:D ..... 444  
\tex\_showtokens:D .....  
..... 710, 564, 1368, 9630, 13214  
\tex\_sjis:D ..... 1161  
\tex\_skewchar:D ..... 445  
\tex\_skip:D .....  
..... 446, 3926, 3955, 3974, 4037, 4053, 4059  
\tex\_skipdef:D ..... 447  
\tex\_space:D ..... 187  
\tex\_spacefactor:D ..... 448  
\tex\_spaceskip:D ..... 449  
\tex\_span:D ..... 450  
\tex\_special:D ..... 451  
\tex\_splitbotmark:D ..... 452  
\tex\_splitbotmarks:D ..... 565  
\tex\_splitdiscards:D ..... 566  
\tex\_splitfirstmark:D ..... 453  
\tex\_splitfirstmarks:D ..... 567  
\tex\_splitmaxdepth:D ..... 454  
\tex\_splittopskip:D ..... 455  
\tex\_strcmp:D .....  
..... 708, 1379, 11324, 13376, 22901  
\tex\_string:D ..... 456, 1213,  
1217, 1316, 1322, 1328, 1334, 1456  
\tex\_suppressfontnotfounderror:D  
..... 714, 1287  
\tex\_suppressifcsnameerror:D ...  
..... 900, 1280  
\tex\_suppresslongerror:D .. 901, 1282  
\tex\_suppressmathparerror:D 902, 1283  
\tex\_suppressoutererror:D . 903, 1285  
\tex\_suppressprimitiveerror:D .. 905  
\tex\_synctex:D ..... 693  
\tex\_tabskip:D ..... 457  
\tex\_tagcode:D ..... 694  
\tex\_tate:D ..... 1162  
\tex\_tbaselineshift:D ..... 1163  
\tex\_textbaselineshiftfactor:D 1165  
\tex\_textdir:D ..... 906, 1305  
\tex\_textdirection:D ..... 907  
\tex\_textfont:D ..... 458  
\tex\_textstyle:D ..... 459  
\tex\_TeXXeTstate:D ..... 568  
\tex\_tfont:D ..... 1166  
\tex\_the:D ..... 349,  
390, 1016, 1022, 1023, 159, 460,  
1909, 2192, 2341, 2345, 2674, 2716,  
2807, 2826, 2841, 2846, 3451, 3483,  
3531, 3532, 3563, 3564, 3570, 3571,  
4052, 4166, 4352, 4475, 4494, 4500,  
4503, 9105, 16714, 17189, 17190,  
17365, 17366, 18753, 18823, 18829,  
18835, 18841, 20314, 20315, 20316,  
20386, 20387, 23521, 24009, 33018  
\tex\_thickmuskip:D ..... 461  
\tex\_thinmuskip:D ..... 462  
\tex\_time:D ..... 463, 1315, 1319  
\tex\_toks:D ..... 464, 2817,  
2846, 3424, 3451, 3483, 3520, 3531,  
3532, 3563, 3564, 3570, 3571, 3575,  
3585, 3595, 3871, 3889, 4052, 4475,  
4478, 4480, 4485, 4493, 4494, 4499,  
4500, 4503, 16714, 16725, 16726, 16727  
\tex\_toksapp:D ..... 908  
\tex\_toksdef:D ..... 465, 3702  
\tex\_tokspre:D ..... 909  
\tex\_tolerance:D ..... 466  
\tex\_topmark:D ..... 467  
\tex\_topmarks:D ..... 569  
\tex\_topskip:D ..... 468  
\tex\_toucs:D ..... 1167  
\tex\_tpack:D ..... 910  
\tex\_tracingassigns:D ..... 570  
\tex\_tracingcommands:D ..... 469  
\tex\_tracingfonts:D .....  
..... 680, 941, 1239, 1241, 1245  
\tex\_tracinggroups:D ..... 571  
\tex\_tracingifs:D ..... 572  
\tex\_tracinglostchars:D ..... 470  
\tex\_tracingmacros:D ..... 471  
\tex\_tracingnesting:D .....  
..... 573, 8978, 11010, 12278  
\tex\_tracingonline:D .....  
..... 472, 2220, 2226, 2227, 33033  
\tex\_tracingoutput:D ..... 473  
\tex\_tracingpages:D ..... 474  
\tex\_tracingparagraphs:D ..... 475  
\tex\_tracingrestores:D ..... 476  
\tex\_tracingscantokens:D ..... 574  
\tex\_tracingstacklevels:D .... 1193  
\tex\_tracingstats:D ..... 477  
\tex\_uccode:D ..... 478, 18833, 18835  
\tex\_Uchar:D ..... 943, 1286  
\tex\_Ucharcat:D ..... 944, 1391, 18911  
\tex\_uchyph:D ..... 479  
\tex\_UCS:D ..... 1168  
\tex\_Udelcode:D ..... 945  
\tex\_Udelcodenum:D ..... 946  
\tex\_Udelimiter:D ..... 947  
\tex\_Udelimiterover:D ..... 948  
\tex\_Udelimiterunder:D ..... 949  
\tex\_Uhextensible:D ..... 950

\tex_Umathaccent:D . . . . .	951	\tex_Umathopeninnerspacing:D . . . . .	1014
\tex_Umathaxis:D . . . . .	952	\tex_Umathopenopenspacing:D . . . . .	1015
\tex_Umathbinbinspacing:D . . . . .	953	\tex_Umathopenopspacing:D . . . . .	1016
\tex_Umathbinclosespacing:D . . . . .	954	\tex_Umathopenordspacing:D . . . . .	1017
\tex_Umathbininnerspacing:D . . . . .	955	\tex_Umathopenpunctspacing:D . . . . .	1018
\tex_Umathbinopenspacing:D . . . . .	956	\tex_Umathopenrelspacing:D . . . . .	1019
\tex_Umathbinopsspacing:D . . . . .	957	\tex_Umathoperatorsize:D . . . . .	1020
\tex_Umathbinordspacing:D . . . . .	958	\tex_Umathopinnerspacing:D . . . . .	1021
\tex_Umathbinpunctspacing:D . . . . .	959	\tex_Umathopopenspacing:D . . . . .	1022
\tex_Umathbinrelspacing:D . . . . .	960	\tex_Umathopopspacing:D . . . . .	1023
\tex_Umathchar:D . . . . .	961	\tex_Umathopordspacing:D . . . . .	1024
\tex_Umathcharclass:D . . . . .	962	\tex_Umathoppunctspacing:D . . . . .	1025
\tex_Umathchardef:D . . . . .	963	\tex_Umathoprelspacing:D . . . . .	1026
\tex_Umathcharfam:D . . . . .	964	\tex_Umathordbinspacing:D . . . . .	1027
\tex_Umathcharnum:D . . . . .	965	\tex_Umathordclosespacing:D . . . . .	1028
\tex_Umathcharnumdef:D . . . . .	966	\tex_Umathordinnerspacing:D . . . . .	1029
\tex_Umathcharslot:D . . . . .	967	\tex_Umathordopenspacing:D . . . . .	1030
\tex_Umathclosebinspacing:D . . . . .	968	\tex_Umathordopspacing:D . . . . .	1031
\tex_Umathcloseclosespacing:D . . . . .	970	\tex_Umathordordspacing:D . . . . .	1032
\tex_Umathcloseinnerspacing:D . . . . .	972	\tex_Umathordpunctspacing:D . . . . .	1033
\tex_Umathcloseopenspacing:D . . . . .	973	\tex_Umathordrelspacing:D . . . . .	1034
\tex_Umathcloseopspacing:D . . . . .	974	\tex_Umathoverbarkern:D . . . . .	1035
\tex_Umathcloseordspacing:D . . . . .	975	\tex_Umathoverbarrule:D . . . . .	1036
\tex_Umathsepunctspacing:D . . . . .	977	\tex_Umathoverbarvgap:D . . . . .	1037
\tex_Umathcloserelspacing:D . . . . .	978	\tex_Umathoverdelimiterbgap:D . . . . .	1039
\tex_Umathcode:D . . . . .	979	\tex_Umathoverdelimitervgap:D . . . . .	1041
\tex_Umathcodenum:D . . . . .	980	\tex_Umathpunctbinspacing:D . . . . .	1042
\tex_Umathconnectoroverlapmin:D . . . . .	982	\tex_Umathpunctclosespacing:D . . . . .	1044
\tex_Umathfractiondelsize:D . . . . .	983	\tex_Umathpunctinnerspacing:D . . . . .	1046
\tex_Umathfractiondenomdown:D . . . . .	985	\tex_Umathpunctopenspacing:D . . . . .	1047
\tex_Umathfractiondenomvgap:D . . . . .	987	\tex_Umathpuncttopspacing:D . . . . .	1048
\tex_Umathfractionnumup:D . . . . .	988	\tex_Umathpunctordspacing:D . . . . .	1049
\tex_Umathfractionnumvgap:D . . . . .	989	\tex_Umathpunctpunctspacing:D . . . . .	1051
\tex_Umathfractionrule:D . . . . .	990	\tex_Umathpunctrelspacing:D . . . . .	1052
\tex_Umathinnerbinspacing:D . . . . .	991	\tex_Umathquad:D . . . . .	1053
\tex_Umathinnerclosespacing:D . . . . .	993	\tex_Umathradicaldegreeafter:D . . . . .	1055
\tex_Umathinnerinnerspacing:D . . . . .	995	\tex_Umathradicaldegreebefore:D . . . . .	1057
\tex_Umathinneropenspacing:D . . . . .	996	\tex_Umathradicaldegreeraise:D . . . . .	1059
\tex_Umathinneropspacing:D . . . . .	997	\tex_Umathradicalkern:D . . . . .	1060
\tex_Umathinnerordspacing:D . . . . .	998	\tex_Umathradicalrule:D . . . . .	1061
\tex_Umathinnerpunctspacing:D . . . . .	1000	\tex_Umathradicalvgap:D . . . . .	1062
\tex_Umathinnerrelspacing:D . . . . .	1001	\tex_Umathrelbinspacing:D . . . . .	1063
\tex_Umathlimitabovebgap:D . . . . .	1002	\tex_Umathrelclosespacing:D . . . . .	1064
\tex_Umathlimitabovékern:D . . . . .	1003	\tex_Umathrelinnerspacing:D . . . . .	1065
\tex_Umathlimitabovevgap:D . . . . .	1004	\tex_Umathrelopspacing:D . . . . .	1066
\tex_Umathlimitbelowbgap:D . . . . .	1005	\tex_Umathrellospacing:D . . . . .	1067
\tex_Umathlimitbelowkern:D . . . . .	1006	\tex_Umathrelordspacing:D . . . . .	1068
\tex_Umathlimitbelowvgap:D . . . . .	1007	\tex_Umathrelpunctspacing:D . . . . .	1069
\tex_Umathnolimitssubfactor:D . . . . .	1008	\tex_Umathrelrelspacing:D . . . . .	1070
\tex_Umathnolimitsupfactor:D . . . . .	1009	\tex_Umathskewedfractionhgap:D . . . . .	1072
\tex_Umathopbinspacing:D . . . . .	1010	\tex_Umathskewedfractionvgap:D . . . . .	1074
\tex_Umathopclosespacing:D . . . . .	1011	\tex_Umathspaceafterscript:D . . . . .	1075
\tex_Umathopenbinspacing:D . . . . .	1012	\tex_Umathstackdenomdown:D . . . . .	1076
\tex_Umathopenclosespacing:D . . . . .	1013	\tex_Umathstacknumup:D . . . . .	1077

\tex\_Umathstackvgap:D ..... 1078  
 \tex\_Umathsubshiftdown:D ..... 1079  
 \tex\_Umathsubshiftdrop:D ..... 1080  
 \tex\_Umathsubsupshiftdown:D .. 1081  
 \tex\_Umathsubsupvgap:D ..... 1082  
 \tex\_Umathsubtopmax:D ..... 1083  
 \tex\_Umathsupbottommin:D ..... 1084  
 \tex\_Umathsupshiftdrop:D ..... 1085  
 \tex\_Umathsupshiftup:D ..... 1086  
 \tex\_Umathsupsubbottommax:D .. 1087  
 \tex\_Umathunderbarkern:D ..... 1088  
 \tex\_Umathunderbarrule:D ..... 1089  
 \tex\_Umathunderbarvgap:D ..... 1090  
 \tex\_Umathunderdelimiterbgap:D 1092  
 \tex\_Umathunderdelimitervgap:D 1094  
 \tex\_undefined:D ... 452, 880, 881,  
     1239, 1311, 1312, 1319, 1325, 1331,  
     1337, 1341, 1342, 1343, 1984, 1992,  
     3374, 3810, 3820, 3898, 3999, 4300,  
     18009, 21052, 21067, 21147, 21168  
 \tex\_underline:D ..... 480, 1208  
 \tex\_unexpanded:D .....  
     ..... 396, 575, 1361, 1450, 2590  
 \tex\_unhbox:D ..... 481, 33121  
 \tex\_unhcopy:D ..... 482, 33120  
 \tex\_uniformdeviate:D .... 803,  
     1179, 1180, 681, 942, 8871, 16693,  
     16724, 28473, 28474, 28655, 28658  
 \tex\_unkern:D ..... 483  
 \tex\_unless:D ..... 576, 1434  
 \tex\_Unosubscript:D ..... 1095  
 \tex\_Unosuperscript:D ..... 1096  
 \tex\_unpenalty:D ..... 484  
 \tex\_unskip:D ..... 485  
 \tex\_unvbox:D ..... 486, 33210  
 \tex\_unvcopy:D ..... 487, 33209  
 \tex\_Uoverdelimiter:D ..... 1097  
 \tex\_uppercase:D ..... 488, 37611  
 \tex\_uptexrevision:D ... 1181, 37134  
 \tex\_uptexversion:D ... 1182, 37133  
 \tex\_Uradical:D ..... 1098  
 \tex\_Uroot:D ..... 1099  
 \tex\_Uskewed:D ..... 1100  
 \tex\_Uskewedwithdelims:D ..... 1101  
 \tex\_Ustack:D ..... 1102  
 \tex\_Ustartdisplaymath:D ..... 1103  
 \tex\_Ustartmath:D ..... 1104  
 \tex\_Ustopdisplaymath:D ..... 1105  
 \tex\_Ustopmath:D ..... 1106  
 \tex\_Usubscript:D ..... 1107  
 \tex\_Usuperscript:D ..... 1108  
 \tex\_Uunderdelimter:D ..... 1109  
 \tex\_Uvextensible:D ..... 1110  
 \tex\_vadjust:D ..... 489  
 \tex\_valign:D ..... 490  
 \tex\_vbadness:D ..... 491  
 \tex\_vbox:D ..... 492, 33125,  
     33130, 33135, 33140, 33145, 33164,  
     33169, 33176, 33182, 33197, 33203  
 \tex\_vccenter:D ..... 493, 1360  
 \tex\_vfi:D ..... 1173  
 \tex\_vfil:D ..... 494  
 \tex\_vfill:D ..... 495  
 \tex\_vfilneg:D ..... 496  
 \tex\_vfuzz:D ..... 497  
 \tex\_voffset:D ..... 498, 1367  
 \tex\_vpack:D ..... 911  
 \tex\_vrule:D ..... 499, 34623  
 \tex\_vsized:D ..... 500  
 \tex\_vskip:D ..... 501, 20503  
 \tex\_vsplit:D ..... 502, 33214, 33219  
 \tex\_vss:D ..... 503  
 \tex\_vtop:D ... 504, 33127, 33152, 33157  
 \tex\_wd:D ..... 505, 32936  
 \tex\_widowpenalties:D ..... 577  
 \tex\_widowpenalty:D ..... 506  
 \tex\_write:D .....  
     ..... 507, 1911, 1913, 10508, 10511, 10528  
 \tex\_xdef:D 508, 1506, 1510, 1514, 1518  
 \tex\_XeTeXcharclass:D ..... 715  
 \tex\_XeTeXcharglyph:D ..... 716  
 \tex\_XeTeXcountfeatures:D ..... 717  
 \tex\_XeTeXcountglyphs:D ..... 718  
 \tex\_XeTeXcountselectors:D .... 719  
 \tex\_XeTeXcountvariations:D ... 720  
 \tex\_XeTeXdashbreakstate:D .... 722  
 \tex\_XeTeXdefaultencoding:D ... 721  
 \tex\_XeTeXfeaturecode:D ..... 723  
 \tex\_XeTeXfeaturename:D ..... 724  
 \tex\_XeTeXfindfeaturebyname:D ... 726  
 \tex\_XeTeXfindselectorbyname:D .. 728  
 \tex\_XeTeXfindvariationbyname:D 730  
 \tex\_XeTeXfirstfontchar:D ..... 731  
 \tex\_XeTeXfonttype:D ..... 732  
 \tex\_XeTeXgenerateactualtext:D .. 734  
 \tex\_XeTeXglyph:D ..... 735  
 \tex\_XeTeXglyphbounds:D ..... 736  
 \tex\_XeTeXglyphindex:D ..... 737  
 \tex\_XeTeXglyphname:D ..... 738  
 \tex\_XeTeXinputencoding:D ..... 739  
 \tex\_XeTeXinputnormalization:D .. 741  
 \tex\_XeTeXinterchartokenstate:D 743  
 \tex\_XeTeXinterchartoks:D ..... 744  
 \tex\_XeTeXisdefaultselector:D ... 746  
 \tex\_XeTeXisexclusivefeature:D .. 748  
 \tex\_XeTeXlastfontchar:D ..... 749  
 \tex\_XeTeXlinebreaklocale:D ... 751  
 \tex\_XeTeXlinebreakpenalty:D ... 752

```

\tex_XeTeXlinebreakskip:D ..... 750
\tex_XeTeXOTcountfeatures:D ... 753
\tex_XeTeXOTcountlanguages:D .. 754
\tex_XeTeXOTcountsscripts:D ... 755
\tex_XeTeXOTfeaturetag:D ..... 756
\tex_XeTeXOTlanguagetag:D .... 757
\tex_XeTeXOTscripttag:D ..... 758
\tex_XeTeXpdffile:D ..... 759
\tex_XeTeXpdfpagecount:D ..... 760
\tex_XeTeXpicfile:D ..... 761
\tex_XeTeXrevision:D 762, 9084, 37142
\tex_XeTeXselectorname:D ..... 763
\tex_XeTeXtracingfonts:D ..... 764
\tex_XeTeXupwardsmode:D ..... 765
\tex_XeTeXuseglyphmetrics:D ... 766
\tex_XeTeXvariation:D ..... 767
\tex_XeTeXvariationdefault:D .. 768
\tex_XeTeXvariationmax:D ..... 769
\tex_XeTeXvariationmin:D ..... 770
\tex_XeTeXvariationname:D .... 771
\tex_XeTeXversion:D ..... 772, 8811, 13877, 17308, 17985, 37141
\tex_xkanjiskip:D ..... 1169
\tex_xleaders:D ..... 509
\tex_xspaceskip:D ..... 510
\tex_xspcde:D ..... 1170
\tex_ybaselineshift:D ..... 1171
\tex_year:D ..... 511, 1333, 1337
\tex_yoko:D ..... 1172
\tex ..... 32485
text commands:
  \l_text_accents_t1 ..... 267, 270, 30138, 30440, 32619
  \l_text_case_exclude_arg_t1 ... 268, 270, 30156, 30817
  \text_case_switch:nnnn ..... 269, 30398, 30866, 31092, 31092
  \text_declare_case_equivalent:Nn ..... 269, 31087, 31087
  \text_declare_expand_equivalent:Nn ..... 267, 30598, 30598, 30603
  \text_declare_purify_equivalent:Nn ..... 270, 32465, 32465, 32470,
                                         32478, 32479, 32480, 32481, 32498,
                                         32523, 32524, 32526, 32527, 32529,
                                         32532, 32533, 32539, 32541, 32542,
                                         32543, 32551, 32563, 32605, 32620
  \text_expand ..... 268
  \text_expand:n ..... 267, 270,
                     271, 30210, 30210, 30631, 31925, 32282
  \l_text_expand_exclude_t1 ..... 267, 270, 30167, 30404
  \l_text_letterlike_t1 ..... 267, 270, 30138, 30460
  \text_lowercase:n ..... 131, 185, 268, 30609, 30609, 37661,
                     37663, 37685, 37686, 37701, 37702
  \text_lowercase:nn ..... 268, 30609, 30617, 37664, 37666
  \text_map_break: ..... 271,
                     31924, 31930, 31949, 31965, 32006,
                     32156, 32258, 32259, 32261, 32269
  \text_map_break:n .. 271, 31924, 32260
  \text_map_function:nN ..... 271, 31924, 31924, 32267
  \text_map_inline:nn 271, 32262, 32262
  \l_text_math_arg_t1 ..... 267, 270, 30163, 30403, 30816, 32398
  \l_text_math_delims_t1 ..... 267, 270, 30165, 30322, 30746, 32327
  \text_purify:n ..... 270, 32276, 32276
  \text_titlecase:n ..... 131, 198, 268, 269, 30609,
                     30613, 37673, 37675, 37689, 37690
  \text_titlecase:nn ..... 268, 30609, 30621, 37676, 37678
  \l_text_titlecase_check_letter_-_bool ..... 269, 270, 30607, 30967
  \text_titlecase_first:n ..... 268, 269, 30609, 30615
  \text_titlecase_first:nn ..... 268, 30609, 30623
  \text_uppercase:n ..... 131,
                     185, 198, 268, 30609, 30611, 37667,
                     37669, 37687, 37688, 37703, 37704
  \text_uppercase:nn ..... 268, 30609, 30619, 37670, 37672
text internal commands:
  \__text_case_switch_marker: .... 31092, 31094, 31097
  \__text_change_case:nnn ..... 30610, 30612, 30614, 30616, 30618,
                             30620, 30622, 30624, 30625, 30625
  \__text_change_case_auxi:nnn ... 30625, 30630, 30635
  \__text_change_case_auxii:nnn ... 30625, 30652, 30653,
                                30655, 30697, 30711, 30723, 30847
  \__text_change_case_BCP:nnn ... 30625, 30636, 30637
  \__text_change_case_BCP:nnnw ... 30625, 30648, 30649
  \__text_change_case_BCP:nnw ... 30625, 30639, 30644
  \__text_change_case_boundary_-_upper_el:nnN .. 31475, 31479, 31483
  \__text_change_case_boundary_-_upper_el:nnn .. 31475, 31489, 31493

```

\\_\_text\_change\_case\_boundary\_-  
     upper\_el:nnN . [31475](#), [31505](#), [31516](#)  
 \\_\_text\_change\_case\_boundary\_-  
     upper\_el:Nnnw ..... [31475](#), [31475](#)  
 \\_\_text\_change\_case\_boundary\_-  
     upper\_el:nnnw . [31475](#), [31498](#), [31501](#)  
 \\_\_text\_change\_case\_break:w ....  
     .. [30625](#), [30685](#), [30726](#), [30889](#), [31086](#)  
 \\_\_text\_change\_case\_catcode:nn ..  
     ..... [31045](#), [31062](#), [31066](#),  
[31105](#), [31275](#), [31279](#), [31510](#), [31531](#),  
[31545](#), [31547](#), [31561](#), [31563](#), [31628](#),  
[31630](#), [31632](#), [31645](#), [31682](#), [31708](#),  
[31789](#), [31804](#), [31828](#), [31836](#), [31848](#)  
 \\_\_text\_change\_case\_codepoint:nn ..  
     ... [30625](#), [31009](#), [31013](#), [31156](#),  
[31166](#), [31236](#), [31244](#), [31262](#), [31287](#)  
 \\_\_text\_change\_case\_codepoint:nnn ..  
     ..... [30625](#),  
[31015](#), [31018](#), [31023](#), [31027](#), [31028](#)  
 \\_\_text\_change\_case\_codepoint:nnnn ..  
     ... [30625](#), [30928](#), [30935](#), [30997](#),  
[31001](#), [31110](#), [31133](#), [31537](#), [31552](#),  
[31568](#), [31571](#), [31583](#), [31594](#), [31640](#),  
[31703](#), [31741](#), [31754](#), [31793](#), [31852](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     aux:nn ..... [31020](#), [31035](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     aux:nnn . [30625](#), [31026](#), [31032](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     aux:nnnn . [30625](#), [31037](#), [31039](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     lower:nnn ..... [30625](#), [30919](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     title:nn ..... [30625](#), [30988](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     title:nnm .... [30625](#), [30965](#), [30987](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     title:nnnn .....  
         .. [30625](#), [30979](#), [30989](#), [30991](#), [30992](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     titleonly:nn ..... [30625](#), [30990](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     titleonly:nnn .... [30625](#), [30986](#)  
 \\_\_text\_change\_case\_codepoint\_-  
     upper:nnn ..... [30625](#), [30925](#)  
 \\_\_text\_change\_case\_cs\_check:nnN ..  
     ... [30625](#), [30757](#), [30802](#)  
 \\_\_text\_change\_case\_end:w ....  
     .. [30625](#), [30668](#), [30688](#), [30740](#), [30782](#)  
 \\_\_text\_change\_case\_exclude:nnN ..  
     ..... [30625](#), [30805](#), [30812](#)  
 \\_\_text\_change\_case\_exclude:nnNN ..  
     ..... [30625](#), [30823](#), [30826](#), [30835](#)  
 \\_\_text\_change\_case\_exclude:nnN ..  
     ..... [30625](#), [30814](#), [30821](#)  
 \\_\_text\_change\_case\_exclude:nnNnn ..  
     ..... [30625](#), [30838](#), [30839](#)  
 \\_\_text\_change\_case\_exclude:nnNw ..  
     ..... [30625](#), [30833](#), [30837](#)  
 \\_\_text\_change\_case\_group\_-  
     lower:nn ..... [30625](#), [30690](#), [30703](#)  
 \\_\_text\_change\_case\_group\_-  
     title:nn ..... [30625](#), [30704](#)  
 \\_\_text\_change\_case\_group\_-  
     titleonly:nn ..... [30625](#), [30716](#)  
 \\_\_text\_change\_case\_group\_-  
     upper:nnn ..... [30625](#), [30702](#)  
 \\_\_text\_change\_case\_if\_greek:n ..  
     ... [31114](#), [31337](#), [31339](#), [31342](#)  
 \\_\_text\_change\_case\_if\_greek:nTF ..  
     ..... [31114](#), [31116](#)  
 \\_\_text\_change\_case\_if\_greek\_-  
     accent:n [31114](#), [31366](#), [31368](#), [31371](#)  
 \\_\_text\_change\_case\_if\_greek\_-  
     accent:nTF ..... [31114](#), [31230](#)  
 \\_\_text\_change\_case\_if\_greek\_-  
     accent\_p:n ..... [31114](#)  
 \\_\_text\_change\_case\_if\_greek\_-  
     diacritic:n .....  
         ... [31114](#), [31399](#), [31402](#), [31405](#)  
 \\_\_text\_change\_case\_if\_greek\_-  
     diacritic:nTF ..... [31114](#), [31233](#)  
 \\_\_text\_change\_case\_if\_greek\_-  
     diacritic\_p:n ..... [31114](#)  
 \\_\_text\_change\_case\_if\_greek\_p:n ..  
     ... [31114](#)  
 \\_\_text\_change\_case\_if\_takes\_-  
     dialytika:n .....  
         ... [31114](#), [31429](#), [31431](#), [31434](#)  
 \\_\_text\_change\_case\_if\_takes\_-  
     dialytika:nTF . [31114](#), [31258](#), [31304](#)  
 \\_\_text\_change\_case\_if\_takes\_-  
     ypogegrammeni:n .....  
         ... [31114](#), [31454](#), [31456](#), [31459](#)  
 \\_\_text\_change\_case\_if\_takes\_-  
     ypogegrammeni:nTF .. [31114](#), [31170](#)  
 \\_\_text\_change\_case\_letterlike:nnnn ..  
     ... [30625](#), [30892](#), [30896](#), [30898](#), [30899](#)  
 \\_\_text\_change\_case\_letterlike\_-  
     lower:nnN .... [30625](#), [30891](#), [30894](#)  
 \\_\_text\_change\_case\_letterlike\_-  
     title:nnN ..... [30625](#), [30895](#)  
 \\_\_text\_change\_case\_letterlike\_-  
     titleonly:nnN ..... [30625](#), [30897](#)  
 \\_\_text\_change\_case\_letterlike\_-  
     upper:nnN ..... [30625](#), [30893](#)

```

\__text_change_case_loop:nnw . . .
  30625, 30659, 30674, 30700, 30714,
  30735, 30785, 30851, 30863, 30874,
  30879, 30884, 30916, 30945, 30963,
  31078, 31131, 31157, 31167, 31237,
  31246, 31290, 31296, 31309, 31316,
  31322, 31335, 31480, 31486, 31499,
  31512, 31526, 31533, 31634, 31654,
  31685, 31791, 31806, 31830, 31838
\__text_change_case_lower_-
  az:nnnn . . . . . 31854, 31854
\__text_change_case_lower_-
  la-x-medieval:nnnn . . . . . 31574
\__text_change_case_lower_lt:nnN
  . . . . . 31596, 31653, 31657
\__text_change_case_lower_lt:nnn
  . . . . . 31596, 31660, 31662
\__text_change_case_lower_-
  lt:nnnn . . . . . 31596
\__text_change_case_lower_lt:nnw
  . . . . . 31596, 31647, 31650
\__text_change_case_lower_lt_-
  auxi:nnnn . . . . . 31598, 31609
\__text_change_case_lower_lt_-
  auxii:nnnn . . . . . 31613, 31637
\__text_change_case_lower_-
  sigma:nnN . . . . . 30625, 30941, 30949
\__text_change_case_lower_-
  sigma:nnnn . . . . . 30625, 30922, 30931
\__text_change_case_lower_-
  sigma:nnnw . . . . . 30625, 30934, 30938
\__text_change_case_lower_-
  tr:NnnN . . . . . 31779, 31799, 31810
\__text_change_case_lower_-
  tr:Nnnn . . . . . 31779, 31813, 31815
\__text_change_case_lower_-
  tr:nnnn . . . . . 31779, 31779, 31855
\__text_change_case_lower_-
  tr:nnNw . . . . . 31779, 31782, 31796
\__text_change_case_math_-
  group:nnNn . . . . . 30625, 30774, 30788
\__text_change_case_math_-
  loop:nnNw . . . . . 30625,
  30763, 30768, 30786, 30791, 30800
\__text_change_case_math_N_-
  type:nnNN . . . . . 30625, 30771, 30779
\__text_change_case_math_-
  search:nnNNN . . . . .
  . . . . . 30625, 30750, 30754, 30766
\__text_change_case_math_-
  space:nnNw . . . . . 30625, 30775, 30795
\__text_change_case_N_type:nnN . . .
  . . . . . 30625, 30677, 30737
\__text_change_case_N_type:nnnN . . .
  . . . . . 30625, 30745, 30748
\__text_change_case_N_type_-
  aux:nnN . . . . . 30625, 30741, 30743
\__text_change_case_next_end:nn
  . . . . . 30625, 31085
\__text_change_case_next_-
  lower:nn . . . . .
  . . . . . 30625, 31077, 31080, 31082, 31084
\__text_change_case_next_-
  title:nn . . . . . 30625, 31081
\__text_change_case_next_-
  titleonly:nn . . . . . 30625, 31083
\__text_change_case_next_-
  upper:nn . . . . . 30625, 31079
\__text_change_case_replace:nnN
  . . . . . 30625, 30829, 30853
\__text_change_case_replace:nnn
  . . . . . 30625, 30857, 30862
\__text_change_case_result:n . . .
  . . . . . 30625, 30661, 30666, 30667, 30668
\__text_change_case_setup:NN . . .
  . . . . . 31859, 31866, 31868
\__text_change_case_setup:Nn . . .
  . . . . . 31892, 31912, 31914
\__text_change_case_space:nnw . . .
  . . . . . 30625, 30681, 30730
\__text_change_case_store:n 30625,
  30663, 30665, 30687, 30692, 30706,
  30718, 30733, 30762, 30783, 30790,
  30799, 30842, 30844, 30873, 30878,
  30883, 30888, 30903, 30908, 30943,
  30951, 31006, 31008, 31102, 31127,
  31155, 31165, 31235, 31241, 31243,
  31261, 31268, 31286, 31507, 31528,
  31542, 31558, 31578, 31589, 31625,
  31642, 31679, 31705, 31750, 31773,
  31786, 31801, 31825, 31833, 31845
\__text_change_case_store:nw . . .
  . . . . . 30625, 30664, 30666
\__text_change_case_switch:nnN . . .
  . . . . . 30625, 30860, 30864
\__text_change_case_switch_-
  lower:nnNnnn . . . . . 30625, 30871
\__text_change_case_switch_-
  title:nnNnnn . . . . . 30625, 30881
\__text_change_case_switch_-
  titleonly:nnNnnn . . . . . 30625, 30886
\__text_change_case_switch_-
  upper:nnNnnn . . . . . 30625, 30876
\__text_change_case_title_-
  el:nnnn . . . . . 31536, 31536
\__text_change_case_title_-
  hy-x-yiwn:nnnn . . . . . 31538

```

\\_\_text\_change\_case\_title\_-  
   hy:nnnn ..... [31538](#), [31554](#)  
 \\_\_text\_change\_case\_title\_nl:nnN  
   ..... [31737](#), [31759](#), [31763](#)  
 \\_\_text\_change\_case\_title\_-  
   nl:nnnn ..... [31737](#), [31737](#)  
 \\_\_text\_change\_case\_title\_nl:nnw  
   ..... [31737](#), [31752](#), [31756](#)  
 \\_\_text\_change\_case\_title\_nl\_-  
   aux:nnnn ..... [31737](#), [31740](#), [31744](#)  
 \\_\_text\_change\_case\_upper\_-  
   az:nnnn ..... [31854](#), [31856](#)  
 \\_\_text\_change\_case\_upper\_-  
   de-alt:nnnn ..... [31098](#)  
 \\_\_text\_change\_case\_upper\_-  
   de-x-eszett:nnnn ..... [31098](#)  
 \\_\_text\_change\_case\_upper\_-  
   el-x-iota:n ..... [31114](#)  
 \\_\_text\_change\_case\_upper\_-  
   el-x-iota:nnnn ..... [31114](#)  
 \\_\_text\_change\_case\_upper\_el:n ..  
   ..... [31114](#), [31138](#)  
 \\_\_text\_change\_case\_upper\_el:nnn  
   ..... [31114](#), [31118](#), [31145](#)  
 \\_\_text\_change\_case\_upper\_-  
   el:nnnN ..... [31114](#), [31153](#), [31161](#)  
 \\_\_text\_change\_case\_upper\_-  
   el:nnnn ..... [31114](#), [31114](#), [31137](#)  
 \\_\_text\_change\_case\_upper\_-  
   el:nnnw ..... [31114](#), [31148](#), [31150](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   aux:nnnN ..... [31114](#),  
   [31175](#), [31186](#), [31192](#), [31217](#), [31220](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   aux:nnnn ..... [31114](#), [31223](#), [31225](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   dialytika:n .....  
   ..... [31114](#), [31259](#), [31266](#), [31306](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   dialytika:nnn .. [31114](#), [31228](#), [31256](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   gobble:nnN ... [31114](#), [31315](#), [31319](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   gobble:nnn ... [31114](#), [31325](#), [31329](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   gobble:nnw .....  
   ... [31114](#), [31264](#), [31307](#), [31311](#), [31334](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   hiatus:nnN ... [31114](#), [31289](#), [31293](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   hiatus:nnn ... [31114](#), [31299](#), [31302](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   hiatus:nnnw ... [31114](#), [31231](#), [31283](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   ypogegrammeni:nnnmN .....  
   ..... [31114](#), [31183](#), [31189](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   ypogegrammeni:nnnnnm .....  
   ..... [31114](#), [31196](#), [31202](#)  
 \\_\_text\_change\_case\_upper\_el\_-  
   ypogegrammeni:nnnnmw .....  
   .. [31114](#), [31172](#), [31178](#), [31206](#), [31214](#)  
 \\_\_text\_change\_case\_upper\_-  
   hy-x-yiwn:nnnn ..... [31538](#)  
 \\_\_text\_change\_case\_upper\_-  
   hy:nnnn ..... [31538](#), [31538](#)  
 \\_\_text\_change\_case\_upper\_-  
   la-x-medieval:nnnn ..... [31574](#)  
 \\_\_text\_change\_case\_upper\_lt:nnN  
   ..... [31687](#), [31716](#), [31720](#)  
 \\_\_text\_change\_case\_upper\_lt:nnm  
   ..... [31687](#), [31723](#), [31725](#)  
 \\_\_text\_change\_case\_upper\_-  
   lt:nnnn ..... [31687](#)  
 \\_\_text\_change\_case\_upper\_lt:nnw  
   ..... [31687](#), [31710](#), [31713](#)  
 \\_\_text\_change\_case\_upper\_lt\_-  
   aux:nnnn ..... [31689](#), [31700](#)  
 \\_\_text\_change\_case\_upper\_-  
   tr:nnnn ..... [31841](#), [31841](#), [31857](#)  
 \\_\_text\_change\_cases\_lower\_-  
   lt:nnnn ..... [31596](#)  
 \\_\_text\_change\_cases\_lower\_lt\_-  
   auxi:nnnn ..... [31596](#)  
 \\_\_text\_change\_cases\_lower\_lt\_-  
   auxii:nnnn ..... [31596](#)  
 \\_\_text\_change\_cases\_upper\_-  
   lt:nnnn ..... [31687](#)  
 \\_\_text\_change\_cases\_upper\_lt\_-  
   aux:nnnn ..... [31687](#)  
 \\_\_text\_char\_catcode:N ... [29970](#),  
   [29970](#), [30944](#), [30960](#), [30961](#), [31063](#),  
   [31069](#), [31579](#), [31590](#), [31751](#), [31774](#)  
 \c\_\_text\_chardef\_group\_begin\_-  
   token ..... [30204](#), [30286](#)  
 \c\_\_text\_chardef\_group\_end\_token  
   ..... [30204](#), [30294](#)  
 \c\_\_text\_chardef\_space\_token ...  
   ..... [30204](#), [30273](#)  
 \\_\_text\_codepoint\_compare:nNn ...  
   ..... [30067](#), [30076](#)  
 \\_\_text\_codepoint\_compare:nNnTF .  
   .... [30062](#), [30933](#), [31041](#), [31068](#),  
   [31100](#), [31125](#), [31204](#), [31227](#), [31240](#),  
   [31540](#), [31556](#), [31576](#), [31587](#), [31781](#),  
   [31784](#), [31843](#), [31980](#), [31986](#), [32031](#),  
   [32037](#), [32084](#), [32090](#), [32189](#), [32195](#)

`\__text_codepoint_compare_p:nNn .`  
 .... [30062](#), [31271](#), [31272](#), [31496](#),  
[31497](#), [31673](#), [31674](#), [31675](#), [31733](#),  
[31823](#), [32063](#), [32064](#), [32231](#), [32232](#)  
`\__text_codepoint_from_chars:N ..`  
 .. [30062](#), [30100](#), [30104](#), [30118](#), [30121](#)  
`\__text_codepoint_from_chars:NN ..`  
 .... [30062](#), [30087](#), [30105](#), [30115](#)  
`\__text_codepoint_from_chars:NNN ..`  
 .... [30062](#), [30092](#), [30127](#)  
`\__text_codepoint_from_chars:NNNN ..`  
 .... [30062](#), [30096](#), [30129](#)  
`\__text_codepoint_from_chars:Nw ..`  
[30062](#), [30072](#), [30078](#), [30082](#), [30110](#),  
[31016](#), [31120](#), [31340](#), [31369](#), [31403](#),  
[31432](#), [31457](#), [31600](#), [31615](#), [31691](#)  
`\__text_codepoint_process:nN ..`  
 .... [30017](#), [30021](#),  
[30024](#), [30043](#), [30807](#), [31147](#), [31194](#),  
[31222](#), [31298](#), [31324](#), [31488](#), [31659](#),  
[31722](#), [31812](#), [31974](#), [32164](#), [32253](#)  
`\__text_codepoint_process:nNN ..`  
 .... [30017](#), [30029](#), [30039](#), [30048](#)  
`\__text_codepoint_process:nNNN ..`  
 .... [30017](#), [30032](#), [30056](#)  
`\__text_codepoint_process:nNNNN ..`  
 .... [30017](#), [30033](#), [30058](#)  
`\__text_data_auxi:w ..` [29828](#), [29855](#)  
`\__text_data_auxii:w ..` [29840](#), [29842](#)  
`\g__text_data_ior ..`  
 .... [29823](#), [29825](#), [29850](#), [29858](#)  
`\__text_end_env:n` [32526](#), [32527](#), [32528](#)  
`\__text_expand:n ..`  
 .... [30210](#), [30215](#), [30218](#), [30254](#)  
`\__text_expand_accent:N ..`  
 .... [30210](#), [30422](#), [30437](#)  
`\__text_expand_accent>NN ..`  
 .... [30210](#), [30439](#), [30443](#), [30455](#)  
`\__text_expand_cs:N ..`  
 .... [30210](#), [30466](#), [30477](#)  
`\__text_expand_cs_expand:N ..`  
 .... [30210](#), [30555](#), [30558](#)  
`\__text_expand_encoding:N ..`  
 .... [30210](#), [30526](#), [30533](#)  
`\__text_expand_encoding_escape:N ..`  
 .... [30210](#)  
`\__text_expand_encoding_escape>NN ..`  
 .... [30538](#), [30541](#)  
`\__text_expand_end:w ..`  
 .. [30210](#), [30230](#), [30267](#), [30354](#), [30504](#)  
`\__text_expand_exclude:N ..`  
 .... [30210](#), [30374](#), [30396](#)  
`\__text_expand_exclude>NN ..`  
 .... [30210](#), [30416](#), [30419](#), [30428](#)  
`\__text_expand_exclude:nN ..`  
 .... [30210](#), [30401](#), [30414](#)  
`\__text_expand_exclude:Nnn ..`  
 .... [30210](#), [30431](#), [30432](#)  
`\__text_expand_exclude:Nw ..`  
 .... [30210](#), [30426](#), [30430](#)  
`\__text_expand_exclude_switch:Nnnnn ..`  
 .... [30210](#), [30399](#), [30409](#)  
`\__text_expand_explicit:N ..`  
 .... [30210](#), [30329](#), [30371](#)  
`\__text_expand_group:n ..`  
 .... [30210](#), [30242](#), [30247](#)  
`\__text_expand_implicit:N ..` [30210](#)  
`\__text_expand_letterlike:N ..`  
 .... [30210](#), [30446](#), [30457](#)  
`\__text_expand_letterlike>NN ..`  
 .... [30210](#), [30459](#), [30463](#), [30475](#)  
`\__text_expand_loop:w ..`  
 .... [30210](#), [30221](#), [30236](#), [30257](#),  
[30262](#), [30290](#), [30298](#), [30308](#), [30314](#),  
[30357](#), [30389](#), [30392](#), [30412](#), [30435](#),  
[30452](#), [30472](#), [30495](#), [30520](#), [30531](#),  
[30538](#), [30557](#), [30564](#), [30568](#), [30596](#)  
`\__text_expand_math_group:Nn ..`  
 .... [30210](#), [30346](#), [30360](#)  
`\__text_expand_math_loop:Nw` [30210](#),  
[30335](#), [30340](#), [30358](#), [30363](#), [30369](#)  
`\__text_expand_math_N_type:NN ..`  
 .... [30210](#), [30343](#), [30351](#)  
`\__text_expand_math_search:NNN ..`  
 .... [30210](#), [30321](#), [30326](#), [30338](#)  
`\__text_expand_math_space:Nw ..`  
 .... [30210](#), [30347](#), [30365](#)  
`\__text_expand_N_type:N ..`  
 .... [30210](#), [30239](#), [30264](#)  
`\__text_expand_N_type_auxi:N ..`  
 .... [30210](#), [30281](#), [30283](#)  
`\__text_expand_N_type_auxii:N ..`  
 .... [30210](#), [30300](#), [30303](#)  
`\__text_expand_N_type_auxiii:N ..`  
 .... [30210](#), [30316](#), [30319](#)  
`\__text_expand_protect:N ..`  
 .... [30210](#), [30492](#), [30499](#)  
`\__text_expand_protect:nN ..`  
 .... [30210](#), [30506](#), [30509](#)  
`\__text_expand_protect:Nw ..`  
 .... [30210](#), [30510](#), [30511](#)  
`\__text_expand_protect:w ..`  
 .... [30210](#), [30480](#), [30489](#)  
`\__text_expand_replace:N ..`  
 .... [30210](#), [30486](#), [30539](#), [30542](#)  
`\__text_expand_replace:n ..`  
 .... [30210](#), [30552](#), [30557](#)

```

\__text_expand_result:n . . . . .
    .. 30210, 30223, 30228, 30229, 30230
\__text_expand_space:w . . . . .
    .. 30210, 30243, 30259, 30280
\__text_expand_store:n . . . . .
    .. 30210, 30225, 30227, 30249,
        30261, 30289, 30297, 30334, 30355,
        30362, 30368, 30391, 30411, 30434,
        30451, 30471, 30494, 30503, 30516,
        30517, 30519, 30530, 30567, 30595
\__text_expand_store:nw . . . . .
    .. 30210, 30226, 30228
\__text_expand_testopt:N . . . . .
    .. 30210, 30485, 30522
\__text_expand_testopt:NNn . . . . .
    .. 30210, 30525, 30528
\__text_expand_unexpanded:N . . .
    .. 1236, 30210, 30582, 30586
\__text_expand_unexpanded:n . . .
    .. 30210, 30579, 30593
\__text_expand_unexpanded:w . . .
    .. 30210, 30563, 30571, 30581
\__text_expand_unexpanded_test:w . . .
    .. 30210, 30573, 30576
\c__text_grapheme_Control_clist . .
    .. 32071
\__text_if_expandable:N . . . . .
    .. 30002
\__text_if_expandable:NTF . . . .
    .. 30002, 30560, 32455
\__text_if_q_recursion_tail_-
    stop_do:Nn . . . . .
        .. 29871, 29871, 30328, 30421,
            30445, 30465, 30739, 30756, 30781,
            30828, 31962, 32003, 32153, 32247,
            32321, 32333, 32374, 32402, 32445
\__text_if_q_recursion_tail_-
    stop_do:nn . . . . .
        .. 29871, 29872, 32029, 32082, 32187
\__text_if_recursion_tail_stop:N . . .
    .. 32275, 32275
\__text_if_s_recursion_tail_-
    stop_do:Nn . . . . .
        .. 29877, 29877, 30266, 30353, 30501
\__text_loop:Nn 32548, 32556, 32560,
    32568, 32579, 32622, 32627, 32654
\__text_loop:NNn . 32673, 32679, 32681
\__text_map_class:Nnnn . . . . .
    .. 31924, 31988,
        32017, 32099, 32101, 32103, 32105,
        32107, 32109, 32111, 32113, 32115
\__text_map_class:nNnn . . . . .
    .. 31924, 32019, 32022
\__text_map_class_end:nw . 31924,
    .. 32033, 32040, 32045, 32086, 32093
\__text_map_class_loop:Nnnnw . . .
    .. 31924, 32024, 32027, 32038
\__text_map_codepoint:Nnn . . . . .
    .. 31924, 31975, 31978
\__text_map_Control:Nnn 31924, 32046
\__text_map_CR:NnN 31924, 31994, 32001
\__text_map_CR:Nnw 31924, 31983, 31991
\__text_map_Extend:Nnn . . . . .
    .. 31924, 32052, 32054
\__text_map_function:nN . . . . .
    .. 31924, 31925, 31926
\__text_map_group:Nnn . . . . .
    .. 31924, 31938, 31943
\__text_map_hangul:NnnN . . . . .
    .. 31924, 32144, 32151
\__text_map_hangul:Nnnn . . . . .
    .. 31924, 32165, 32168
\__text_map_hangul:nNnnnw . . . . .
    .. 31924, 32177, 32180
\__text_map_hangul:Nnnw . . . . .
    .. 31924, 32124, 32130,
        32137, 32141, 32208, 32213, 32219
\__text_map_hangul_aux:Nnnnw . 31924
\__text_map_hangul_aux:Nnnw . . . . .
    .. 32170, 32173, 32204
\__text_map_hangul_end:nw . . . . .
    .. 31924, 32191, 32198, 32205
\__text_map_hangul_L:Nnn 31924, 32206
\__text_map_hangul_loop:Nnnnnw . . .
    .. 31924, 32182, 32185, 32196
\__text_map_hangul_LV:Nnn . . . . .
    .. 31924, 32211, 32216
\__text_map_hangul_LVT:Nnn . . . . .
    .. 31924, 32217, 32222
\__text_map_hangul_next:Nnnn . . .
    .. 31924, 32188, 32192, 32203
\__text_map_hangul_T:Nnn 31924, 32222
\__text_map_hangul_V:Nnn 31924, 32216
\__text_map_L:Nnn . . . . .
    .. 31924, 32121
\__text_map_lookahead:NnNN . . . . .
    .. 31924, 32241, 32245
\__text_map_lookahead:NnNw . . . . .
    .. 31924, 32058, 32226, 32238
\__text_map_loop:Nnw . . . . .
    .. 31924, 31928, 31932, 31947, 31951,
        31958, 31971, 31987, 31997, 32013,
        32015, 32050, 32053, 32067, 32083,
        32087, 32094, 32119, 32147, 32161,
        32176, 32234, 32236, 32242, 32251
\__text_map_LV:Nnn 31924, 32127, 32133
\__text_map_LVT:Nnn . . . . .
    .. 31924, 32134, 32140
\__text_map_N_type:NnN . . . . .
    .. 31924, 31935, 31960

```

\\_\_text\_map\_not\_Control:Nnn . . . . .  
 ..... 31924, 32098  
 \\_\_text\_map\_not\_Extend:Nnn . . . . .  
 ..... 31924, 32100  
 \\_\_text\_map\_not\_L:Nnn . 31924, 32106  
 \\_\_text\_map\_not\_LV:Nnn . 31924, 32108  
 \\_\_text\_map\_not\_LVT:Nnn . 31924, 32112  
 \\_\_text\_map\_not\_Pprepend:Nnn . . . . .  
 ..... 31924, 32104  
 \\_\_text\_map\_not\_Regional\_-  
 Indicator:Nnn . . . . . 32116  
 \\_\_text\_map\_not\_SpacingMark:Nnn .  
 ..... 31924, 32102  
 \\_\_text\_map\_not\_T:Nnn . 31924, 32114  
 \\_\_text\_map\_not\_V:Nnn . 31924, 32110  
 \\_\_text\_map\_output:Nn . 31924,  
 31945, 31956, 31964, 31969, 31982,  
 32012, 32048, 32049, 32057, 32118,  
 32123, 32129, 32136, 32225, 32256  
 \\_\_text\_map\_Pprepend:Nnn . 31924, 32055  
 \\_\_text\_map\_Pprepend:nNnn . . . . .  
 ..... 31924, 32070, 32075  
 \\_\_text\_map\_Pprepend\_aux:Nnn . . . . .  
 ..... 31924, 32058, 32060  
 \\_\_text\_map\_Pprepend\_loop:Nnnw . . . . .  
 ..... 31924, 32077, 32080, 32091  
 \\_\_text\_map\_Regional\_Indicator:Nnn  
 ..... 31924, 32223  
 \\_\_text\_map\_Regional\_Indicator\_-  
 aux:Nnn . . . . . 31924, 32226, 32228  
 \\_\_text\_map\_space:Nnw . . . . .  
 ..... 31924, 31939, 31954  
 \\_\_text\_map\_SpacingMark:Nnn . . . . .  
 ..... 31924, 32054  
 \\_\_text\_map\_T:Nnn . . . . . 31924, 32140  
 \\_\_text\_map\_V:Nnn . . . . . 31924, 32133  
 \l\_\_text\_math\_mode\_t1 . . . . . 30203  
 \c\_\_text\_mathchardef\_group\_-  
 begin\_token . . . . . 30204, 30287  
 \c\_\_text\_mathchardef\_group\_end\_-  
 token . . . . . 30204, 30295  
 \c\_\_text\_mathchardef\_space\_token  
 ..... 30204, 30277  
 \\_\_text\_purify:n . 32276, 32281, 32285  
 \\_\_text\_purify\_accent>NN . . . . .  
 ..... 32606, 32606, 32620  
 \\_\_text\_purify\_encoding:N . . . . .  
 ..... 32276, 32441, 32448  
 \\_\_text\_purify\_encoding\_escape>NN  
 ..... 32276, 32453, 32460  
 \\_\_text\_purify\_end:w . . . . .  
 .. 32276, 32296, 32321, 32359, 32445  
 \\_\_text\_purify\_expand:N . . . . .  
 ..... 32276, 32428, 32437  
 \\_\_text\_purify\_group:n . . . . .  
 ..... 32276, 32308, 32313  
 \\_\_text\_purify\_loop:w 32276, 32288,  
 32302, 32313, 32317, 32354, 32432,  
 32436, 32446, 32456, 32457, 32463  
 \\_\_text\_purify\_math\_cmd:N . . . . .  
 ..... 32276, 32334, 32395  
 \\_\_text\_purify\_math\_cmd:n . . . . .  
 ..... 32407, 32411  
 \\_\_text\_purify\_math\_cmd>NN . . . . .  
 ..... 32276, 32397, 32400, 32409  
 \\_\_text\_purify\_math\_cmd:Nn . 32276  
 \\_\_text\_purify\_math\_end:w . . . . .  
 ..... 32276, 32351, 32377, 32412  
 \\_\_text\_purify\_math\_group>NNN . . . . .  
 ..... 32276, 32367, 32383  
 \\_\_text\_purify\_math\_loop>NNw . . . . .  
 ..... 32276,  
 32344, 32361, 32380, 32386, 32393  
 \\_\_text\_purify\_math\_N\_type>NNN . . . . .  
 ..... 32276, 32364, 32372  
 \\_\_text\_purify\_math\_result:n . . . . .  
 ..... 32345,  
 32349, 32350, 32351, 32356, 32412  
 \\_\_text\_purify\_math\_search>NNN . . . . .  
 ..... 32276, 32326, 32331, 32340  
 \\_\_text\_purify\_math\_space>NNw . . . . .  
 ..... 32276, 32368, 32388  
 \\_\_text\_purify\_math\_start>NNw . . . . .  
 ..... 32276, 32338, 32342  
 \\_\_text\_purify\_math\_stop:Nw . . . . .  
 ..... 32356, 32375  
 \\_\_text\_purify\_math\_store:n . . . . .  
 .. 32276, 32347, 32379, 32385, 32392  
 \\_\_text\_purify\_math\_store:nw . . . . .  
 ..... 32276, 32348, 32349  
 \\_\_text\_purify\_N\_type:N . . . . .  
 ..... 32276, 32305, 32319  
 \\_\_text\_purify\_N\_type\_aux:N . . . . .  
 ..... 32276, 32322, 32324  
 \\_\_text\_purify\_protect:N . . . . .  
 ..... 32276, 32440, 32443  
 \\_\_text\_purify\_replace:N . . . . .  
 ..... 32276, 32403, 32413  
 \\_\_text\_purify\_replace:n . . . . .  
 ..... 32276, 32423, 32436  
 \\_\_text\_purify\_result:n . . . . .  
 ..... 32290, 32294, 32295, 32296  
 \\_\_text\_purify\_space:w . . . . .  
 ..... 32276, 32309, 32314  
 \\_\_text\_purify\_store:n . . . . .  
 ..... 32276, 32292,  
 32316, 32353, 32358, 32430, 32462

```

\__text_purify_store:nw ..... 32276, 32293, 32294
\__text_quark_if_nil:n ..... 29866
\__text_quark_if_nil:nTF 29866, 30513
\__text_quark_if_nil_p:n ..... 29866
\__text_tmp:n ..... 32565,
..... 32570, 32626, 32633, 32640, 32678
\__text_tmp:nmm ..... 32570, 32571, 32645, 32646
\__text_tmp:w ..... 30179, 30197
\__text_tmp_aux:n ..... 32642, 32645
\I__text_tmpa_str ..... 29826, 29835, 29836, 29844, 29846
\l__text_tmppb_str ..... 29827, 29830, 29832, 29834, 29838
\__text_token_to_explicit:N .... 29885, 29887, 32431
\__text_token_to_explicit:n .... 29885, 29933, 29937
\__text_token_to_explicit_auxi:w ..... 29885, 29939, 29954
\__text_token_to_explicit_- auxii:w ..... 29885, 29959, 29967
\__text_token_to_explicit_- auxiii:w ..... 29885, 29961, 29969
\__text_token_to_explicit_char:N ..... 29885, 29897, 29923
\__text_token_to_explicit_cs:N .. ..... 29885, 29895, 29902
\__text_token_to_explicit_cs_- aux:N ..... 29885, 29906, 29912
\__text_use_i_delimit_by_q_- recursion_stop:nw ..... 29869, 29869, 30332, 30425, 30449, 30469, 30760, 30832, 32337, 32406
\__text_use_i_delimit_by_s_- recursion_stop:nw ..... 29875, 29875, 29882
\textbaselineshiftfactor ..... 1164
\textbf ..... 32490
\textdir ..... 906
\textdirection ..... 907
\textfont ..... 458
\textit ..... 32492
\textmd ..... 32491
\textnormal ..... 32486
\textrm ..... 32487
\textsc ..... 32495
\textsf ..... 32488
\textsl ..... 32493
\textstyle ..... 459
\texttt ..... 25023, 32489
\textulc ..... 32496
\textup ..... 32494
\TeXeTstate ..... 568
\tfnt ..... 1166
\TH ..... 30154, 31879, 32589
\th ..... 30154, 31879, 32602
\the ..... 46, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 460
\thickmuskip ..... 461
\thinmuskip ..... 462
\c_thirteen ..... 37464
\c_thirty_two ..... 37472
thousand commands:
\c_one_thousand ..... 37480
\c_ten_thousand ..... 37482
\c_three ..... 37444
\time ..... 463, 1316, 9094, 9096
\tiny ..... 32521, 34618
tl commands:
\c_catcode_active_space_tl 313, 37303
\c_catcode_active_tl ..... 187, 876, 19125, 19185
\c_catcode_other_space_tl ..... 183, 637, 10572, 10616, 10696, 10785, 10861, 19103
\c_empty_tl ..... 119, 832, 845, 2698, 9549, 12085, 12101, 12103, 12138, 12442, 17708, 17714, 18061, 18077, 37213, 37250, 37269
\c_job_name_tl ..... 37434
\c_novalue_tl . 106, 119, 12139, 12551
\c_space_tl ..... 119, 3763, 9341, 11594, 12143, 13015, 13865, 18571, 22038, 29688, 29783, 30191, 30259, 30280, 30366, 30730, 30796, 31954, 32314, 32390, 32539, 34403, 34447, 34514, 34937, 34938, 34939, 34946, 34947, 35110, 35111, 35117, 35118, 35119, 35912, 36025, 36068, 36069, 36091, 36858, 36860
\tl_analysis_log:N ... 45, 4102, 4104
\tl_analysis_log:n ... 45, 4116, 4118
\tl_analysis_map_inline:Nn ..... 45, 4078, 4078, 5999
\tl_analysis_map_inline:nn ..... 45, 194, 524, 558, 559, 4078, 4079, 4080, 6778, 7595
\tl_analysis_show:N ..... 45, 4102, 4102, 37593
\tl_analysis_show:n ..... 45, 4116, 4116, 37595
\tl_build_begin:N ..... 312, 313, 505, 1401, 5090, 5601, 6176, 6277, 6810, 7084, 37169, 37169, 37184
\tl_build_clear:N ..... 312, 6845, 7017, 8011, 37184, 37184

```

\tl\_build\_end:N . . . 312, 313, 505,  
   1401, 1402, 5120, 5128, 5611, 6233,  
   6296, 7118, 8037, 8101, 37257, 37257  
 \tl\_build\_gbegin:N . . . . .  
   . . . . . 312, 313, 37169, 37171, 37185  
 \tl\_build\_gclear:N . . . 312, 37184, 37185  
 \tl\_build\_gend:N . . . 313, 37257, 37262  
 \tl\_build\_get:NN . . . . .  
   . . . . . 313, 6836, 37243, 37243  
 \tl\_build\_gput\_left:Nn . . . . .  
   . . . . . 313, 37226, 37229, 37231  
 \tl\_build\_gput\_right:Nn . . . . .  
   . . . . . 313, 37186, 37196, 37201  
 \tl\_build\_put\_left:Nn . . . . .  
   . . . . . 313, 37226, 37226, 37228  
 \tl\_build\_put\_right:Nn . . . . .  
   . . . . . 313, 536, 1403,  
   5097, 5115, 5123, 5127, 5191, 5194,  
   5234, 5248, 5252, 5377, 5391, 5432,  
   5454, 5467, 5499, 5512, 5516, 5598,  
   5604, 5610, 5614, 5657, 5947, 5951,  
   5958, 5964, 5985, 6001, 6019, 6247,  
   6292, 6305, 6879, 7107, 7172, 7241,  
   7298, 7301, 7315, 7383, 8015, 8117,  
   8120, 8128, 8131, 37186, 37186, 37191  
 \tl\_case:Nn . . . . .  
   . . . . . 107, 12581, 12581, 12609, 12610  
 \tl\_case:Nnn . . . . . 37596, 37598  
 \tl\_case:NnTF . . . . . 107, 12581,  
   12586, 12591, 12596, 37597, 37599  
 \tl\_case:nnTF . . . . . 717, 826, 1394  
 \tl\_clear:N . . . . .  
   . . . . . 104, 4424, 4766, 6811, 6846,  
   10665, 10666, 10669, 10678, 10788,  
   10791, 10851, 11294, 12100, 12100,  
   12104, 12107, 12269, 14249, 18119,  
   18120, 21232, 21585, 21697, 30181,  
   35530, 35958, 37074, 37260, 37321  
 \tl\_clear\_new:N . . . . . 104, 11534,  
   11535, 11536, 11537, 11538, 12106,  
   12106, 12110, 18123, 18124, 30600,  
   31089, 32467, 35539, 35572, 35613  
 \tl\_concat:NNN . . . . .  
   104, 895, 12118, 12118, 12134, 13286  
 \tl\_const:Nn . . 104, 572, 3757, 4460,  
   4544, 7452, 8955, 9265, 9266, 9305,  
   9310, 9312, 9314, 9316, 9318, 9323,  
   9324, 9331, 10562, 10568, 10989,  
   12088, 12088, 12093, 12098, 12099,  
   12138, 12141, 12143, 12261, 14086,  
   14091, 16480, 16535, 18117, 18962,  
   19103, 19127, 19685, 19750, 22556,  
   22557, 22558, 22559, 22560, 22568,  
   22657, 24750, 26067, 26514, 26515,  
   26516, 26517, 26518, 26519, 26520,  
   26521, 26522, 29509, 29545, 29729,  
   29741, 29769, 31862, 31864, 31882,  
   31883, 31900, 31907, 32625, 32676,  
   35681, 35682, 35683, 35684, 35685,  
   35731, 35732, 35733, 35743, 35744,  
   35745, 35746, 35747, 35748, 35749,  
   35750, 35869, 35884, 35898, 35932,  
   36188, 36193, 36198, 36359, 37306  
 \tl\_count:N . . . . .  
   . . . . . 32, 106, 109, 12701, 12706, 12713  
 \tl\_count:n . . . . .  
   . . . . . 32, 106, 109, 379, 725, 819, 984,  
   1638, 1642, 2033, 2084, 5863, 7525,  
   7529, 7548, 7552, 7622, 7626, 12701,  
   12701, 12712, 13068, 13083, 13095  
 \tl\_count\_tokens:n . . . . .  
   . . . . . 109, 12714, 12714, 12727  
 \tl\_gclear:N . . . . . 104, 421, 3445,  
   7086, 9063, 9192, 12100, 12102,  
   12105, 12109, 18121, 18122, 37265  
 \tl\_gclear\_new:N . . . . .  
   104, 12106, 12108, 12111, 18125, 18126  
 \tl\_gconcat:NNN . . . . .  
   . . . . . 104, 12118, 12126, 12135, 13287  
 \tl\_gput\_left:Nn . . . . . 104, 12164,  
   12184, 12189, 12194, 12202, 12208,  
   12209, 12210, 12211, 14974, 15157  
 \tl\_gput\_right:Nn . . 105, 1582, 1583,  
   7116, 7167, 7168, 7244, 9066, 9195,  
   12212, 12229, 12231, 12236, 12244,  
   12250, 12251, 12252, 12253, 16450,  
   16616, 28742, 29203, 30174, 30176,  
   31887, 31889, 36576, 36578, 36683  
 \tl\_gremove\_all:Nn . . . . .  
   . . . . . 118, 12434, 12436, 12439  
 \tl\_gremove\_once:Nn . . . . .  
   . . . . . 117, 12428, 12430, 12433  
 \tl\_greplace\_all:Nnn . . . . .  
   . . . . . 117, 12360, 12366, 12371, 12437  
 \tl\_greplace\_once:Nnn . . . . .  
   . . . . . 117, 12360, 12362, 12369, 12431  
 \tl\_greverse:N 109, 13051, 13053, 13056  
 .tl\_gset:N . . . . . 227, 21417  
 \tl\_gset:Nn . . . . . 104, 145, 313,  
   676, 685, 1404, 7150, 7159, 8882,  
   8896, 8902, 8911, 8912, 8922, 8923,  
   8937, 9240, 9242, 9244, 9246, 9248,  
   12144, 12150, 12152, 12154, 12160,  
   12161, 12162, 12163, 12273, 16585,  
   16793, 16862, 19798, 19827, 19907  
 \tl\_gset\_eq:NN 104, 3435, 7156, 7447,  
   8494, 12112, 12114, 12117, 13283,  
   14169, 14185, 16503, 16504, 16505,

16506, 18131, 18132, 18133, 18134,  
 19714, 19715, 19716, 19717, 24755  
`\tl_gset_from_file:Nnn` ..... 37600  
`\tl_gset_from_file_x:Nnn` ..... 37602  
`\tl_gset_rescan:Nnn` .....  
     ... 118, 12262, 12272, 12303, 12304  
`.tl_gset_x:N` ..... 228, 21417  
`\tl_gsort:Nn` .....  
     ... 117, 3433, 3435, 3436, 12800  
`\tl_gtrim_spaces:N` .....  
     ... 110, 12744, 12756, 12759  
`\tl_head:N` ... 113, 12800, 12838, 24786  
`\tl_head:n` ... 113, 114, 399, 699, 700,  
     707, 2656, 12800, 12802, 12814,  
     12835, 12838, 13092, 29725, 29734  
`\tl_head:w` ... 114, 700, 701, 12800,  
     12836, 29750, 29779, 29852, 36089  
`\tl_if_blank:n` ..... 12474, 12482  
`\tl_if_blank:nTF` .....  
     . 105, 113, 114, 3607, 6790, 8881,  
     9458, 10343, 10871, 11038, 11043,  
     11062, 11097, 11194, 11255, 11262,  
     11335, 11344, 12473, 12842, 13082,  
     13266, 13293, 13950, 13953, 15308,  
     15311, 18567, 18681, 19009, 19012,  
     19093, 19096, 21604, 21810, 21960,  
     29334, 29337, 29340, 29372, 29375,  
     29479, 29500, 29513, 29530, 29561,  
     29745, 29773, 29854, 30841, 31046,  
     31050, 31611, 31639, 31702, 32257,  
     35003, 35030, 35039, 35043, 35146,  
     35375, 35590, 35598, 36226, 36229,  
     36232, 36256, 36282, 36308, 37413  
`\tl_if_blank_p:n` .....  
     ... 105, 11212, 12473, 35025  
`\tl_if_empty:N` ..... 12440,  
     12448, 13370, 13372, 18372, 18374  
`\tl_if_empty:n` .....  
     ... 12450, 12458, 12465, 13374  
`\tl_if_empty:NTF` .....  
     ... 105, 7178, 9357, 9367, 10682,  
     10772, 10807, 10888, 11131, 11286,  
     11320, 12440, 21632, 21637, 21786,  
     34975, 35321, 35602, 35964, 35985  
`\tl_if_empty:nTF` ... 105, 500, 688,  
     690, 691, 845, 854, 860, 1678, 1772,  
     2078, 2635, 2754, 4352, 4464, 4465,  
     5943, 7770, 7945, 8427, 8568, 9623,  
     9715, 9730, 9818, 9822, 9886, 9991,  
     10024, 10027, 10077, 10078, 10087,  
     10094, 10100, 10107, 10676, 10923,  
     11463, 11469, 11471, 11473, 11680,  
     12374, 12450, 12460, 12532, 12572,  
     12932, 13193, 13226, 13328, 14217,  
     15272, 16196, 16203, 16220, 16370,  
     16549, 18071, 18090, 18099, 18102,  
     18385, 18418, 18531, 18617, 19392,  
     19666, 20956, 21048, 22009, 23260,  
     24113, 28421, 28489, 29767, 36563  
`\tl_if_empty_p:N` .....  
     ... 105, 11597, 12440, 35365  
`\tl_if_empty_p:n` ... 105, 12450, 12460  
`\tl_if_eq:NN` ..... 12484, 12492  
`\tl_if_eq:Nn` ..... 12496, 12508  
`\tl_if_eq:nn` ..... 12509  
`\tl_if_eq:NNTF` ..... 105–  
     107, 123, 138, 716, 801, 9696, 9750,  
     12484, 12605, 13230, 16654, 19958,  
     29616, 34681, 34684, 35052, 35324  
`\tl_if_eq:NnTF` ..... 105, 12496  
`\tl_if_eq:nnTF` .....  
     106, 124, 148, 175, 801, 12509, 18406  
`\tl_if_eq_p:NN` ..... 105, 12484  
`\tl_if_exist:N` .....  
     ... 12136, 12137, 13366, 13368  
`\tl_if_exist:NTF` ..... 104,  
     4108, 11616, 12107, 12109, 12136,  
     12694, 29818, 31885, 35645, 35929  
`\tl_if_exist_p:N` .....  
     ... 104, 11596, 12136, 30172, 36574  
`\tl_if_head_eq_catcode:nN` .....  
     ... 701, 702, 12862, 12878  
`\tl_if_head_eq_catcode:nTF` .....  
     ... 107, 12848  
`\tl_if_head_eq_catcode_p:nN` .....  
     ... 107, 12848  
`\tl_if_head_eq_charcode:nN` .....  
     ... 700, 702, 12848, 12860  
`\tl_if_head_eq_charcode:nTF` ...  
     ... 107, 12848, 29502  
`\tl_if_head_eq_charcode_p:nN` ...  
     ... 107, 12848  
`\tl_if_head_eq_meaning:nN` 701, 12880  
`\tl_if_head_eq_meaning:nTF` ...  
     ... 107, 6030, 12848  
`\tl_if_head_eq_meaning_p:nN` ...  
     ... 107, 5862, 12848,  
     29863, 30383, 30384, 30385, 30386  
`\tl_if_head_is_group:n` ..... 12939  
`\tl_if_head_is_group:ntf` ... 107,  
     2632, 2751, 12868, 12906, 12939,  
     12991, 18097, 30241, 30345, 30578,  
     30679, 30773, 31937, 32307, 32366  
`\tl_if_head_is_group_p:n` ... 107, 12939  
`\tl_if_head_is_N_type:n` ... 701, 12919  
`\tl_if_head_is_N_type:nTF` ... 108,  
     2629, 2693, 2739, 2745, 2790, 2805,  
     2850, 12569, 12851, 12865, 12882,

\tl\_if\_head\_is\_N\_type\_p:n . . . . .  
 \tl\_if\_head\_is\_space:n . . . . .  
 \tl\_if\_head\_is\_space:nTF . . . . .  
 \tl\_if\_head\_is\_space\_p:n . . . . .  
 \tl\_if\_in:Nn . . . . .  
 \tl\_if\_in:nn . . . . .  
 \tl\_if\_in:NnTF . . . . .  
 \tl\_if\_in:nnTF . . . . .  
 \tl\_if\_novalue:n . . . . .  
 \tl\_if\_novalue:nTF . . . . .  
 \tl\_if\_novalue\_p:n . . . . .  
 \tl\_if\_single:n . . . . .  
 \tl\_if\_single:NTF . . . . .  
 \tl\_if\_single:nTF . . . . .  
 \tl\_if\_single\_p:N . . . . .  
 \tl\_if\_single\_p:n . . . . .  
 \tl\_if\_single\_token:n . . . . .  
 \tl\_if\_single\_token:nTF . . . . .  
 \tl\_if\_single\_token\_p:n . . . . .  
 \tl\_item:Nn . . . . .  
 \tl\_item:nn . . . . .  
 \tl\_log:N . . . . .  
 \tl\_log:n . . . . .  
 \tl\_lower\_case:n . . . . .  
 \tl\_lower\_case:nn . . . . .  
 \tl\_map\_break: . . . . .  
 \tl\_map\_break:n . . . . .  
 \tl\_map\_function:NN . . . . .  
 \tl\_map\_function:nN . . . . .  
 \tl\_map\_inline:Nn . . . . .  
 \tl\_map\_inline:nn . . . . .  
 \tl\_map\_tokens:Nn . . . . .  
 \tl\_map\_tokens:nn . . . . .  
 \tl\_map\_variable:NNn . . . . .  
 \tl\_map\_variable:nNn . . . . .  
 \tl\_mixed\_case:n . . . . .  
 \tl\_mixed\_case:nn . . . . .  
 \tl\_new:N . . . . .  
 \tl\_put\_left:Nn . . . . .  
 \tl\_put\_right:Nn . . . . .  
 \tl\_rand\_item:N . . . . .

12919, 13153, 30238, 30342, 30491,  
 30582, 30676, 30770, 30940, 31152,  
 31181, 31288, 31314, 31478, 31504,  
 31652, 31715, 31758, 31798, 31934,  
 31993, 32143, 32240, 32304, 32363  
 108, 12919  
 108, 114, 12954, 13135, 13144, 13859  
 108, 12954  
 106, 12396, 12522, 12523, 12524, 16444  
 106, 690, 691, 716, 4663,  
 8969, 9609, 9611, 12310, 12380,  
 12382, 12522, 12523, 12524, 12527,  
 13411, 13419, 19664, 34494, 37155  
 106, 12540  
 106, 12538  
 106, 12538  
 12556  
 106, 12552, 12553, 12554, 12555  
 106, 576,  
 691, 6024, 6060, 6075, 6108, 12553,  
 12554, 12555, 12556, 30969, 31739  
 106, 12552  
 106, 12552,  
 12556, 31004, 31667, 31730, 31820  
 12567  
 106, 5164, 12567, 31895  
 106, 12567  
 115, 13057, 13078, 13079  
 115, 546, 707,  
 7495, 7535, 13057, 13057, 13078, 13083  
 111, 4105, 13185, 13187, 13188, 13986  
 111, 383, 1056, 2184, 2200,  
 8532, 10258, 10481, 13187, 13220,  
 13220, 13982, 18020, 24780, 36855  
 37661, 37662  
 37661, 37665  
 60, 112, 446, 461, 4092,  
 4093, 12619, 12633, 12648, 12659,  
 12674, 12685, 12685, 12686, 12688  
 112, 113, 3440,  
 11050, 11150, 12685, 12687, 35460  
 111, 112,  
 310, 5987, 12614, 12621, 12623, 12709  
 111, 2077, 5211,  
 12614, 12622, 12704, 16552  
 111,  
 112, 3440, 11149, 12638, 12651,  
 12653, 14082, 14084, 32619, 35456  
 111, 112,  
 141, 461, 3402, 6384, 6838, 8813,  
 10563, 12638, 12638, 12652, 19622,  
 19624, 19626, 24166, 27254, 30182,  
 32471, 32482, 32499, 32530, 37713  
 112, 11049, 12654, 12661, 12663  
 112, 7157, 12654, 12654, 12662, 12679  
 112, 12678, 12682, 12684  
 112, 695, 12678, 12678, 12683  
 37661, 37674  
 37661, 37677  
 103, 104,  
 187, 678, 3329, 3754, 3773, 4535,  
 4536, 4542, 4543, 6753, 6758, 6759,  
 6765, 6766, 6767, 7024, 7026, 7146,  
 7147, 7952, 7953, 7954, 7955, 8954,  
 9067, 9196, 9211, 9259, 9641, 9642,  
 10176, 10179, 10201, 10399, 10410,  
 10444, 10539, 10541, 10554, 10556,  
 10557, 10559, 10863, 10893, 10894,  
 12082, 12082, 12087, 12107, 12109,  
 12494, 12495, 13247, 13248, 13249,  
 13250, 13993, 13994, 16477, 16478,  
 18062, 18114, 18115, 18902, 19480,  
 19684, 20831, 20834, 20839, 20841,  
 20847, 20848, 20850, 20851, 20853,  
 28738, 28913, 28914, 30138, 30141,  
 30156, 30163, 30165, 30167, 30203,  
 33580, 33605, 33606, 34612, 34853,  
 34856, 34952, 34953, 34954, 34955,  
 35318, 35395, 35522, 35640, 37309  
 104, 12164, 12164, 12169, 12174,  
 12182, 12204, 12205, 12206, 12207  
 105, 1402, 4304, 4383, 4393, 4432,  
 4452, 4773, 10813, 10816, 10821,  
 11167, 12212, 12212, 12214, 12219,  
 12227, 12246, 12247, 12248, 12249,  
 16614, 18304, 18933, 18935, 18939,  
 18940, 18942, 18944, 18946, 18947,  
 18949, 18951, 18953, 18955, 20864,  
 30194, 36121, 36167, 37348, 37353  
 115, 13080, 13085, 13086

```

\tl_rand_item:n ..... 115, 13080, 13080, 13085
\tl_range:Nnn 116, 13087, 13087, 13088
\tl_range:nnn ..... 116, 129, 312, 13087, 13087, 13089
\tl_range_braced:Nnn ..... 312, 37275, 37275, 37276
\tl_range_braced:nnn ..... 116, 312, 37275, 37275, 37277
\tl_range_unbraced:Nnn ..... 312, 37275, 37278, 37280
\tl_range_unbraced:nnn ..... 116, 312, 37275, 37279, 37281
\tl_remove_all:Nn ..... 117, 118, 12434, 12434, 12438
\tl_remove_once:Nn ..... 117, 12428, 12428, 12432
\tl_replace_all:Nnn ... 117, 799, 852, 12360, 12364, 12370, 12435, 16561
\tl_replace_once:Nnn ..... 117, 12360, 12360, 12368, 12429
\tl_rescan:nn ..... 118, 119, 263, 682, 12262, 12262
\tl_reverse:N 109, 13051, 13051, 13055
\tl_reverse:n ..... 109, 110, 13032, 13032, 13044, 13052, 13054
\tl_reverse_items:n ..... 109, 110, 12728, 12728
.tl_set:N ..... 227, 21417
\tl_set:Nn ..... 104, 118, 119, 145, 228, 313, 405, 451, 609, 676, 678, 685, 934, 1404, 4227, 5131, 5894, 5971, 6236, 6299, 6829, 6850, 6860, 6876, 6881, 6924, 6956, 6994, 7345, 8006, 8007, 8067, 8959, 8994, 9396, 9625, 9653, 9736, 10302, 10315, 10348, 10630, 10667, 10993, 11028, 11124, 11152, 11267, 11269, 11271, 11273, 11313, 11547, 11550, 11551, 11552, 11553, 11560, 11561, 11562, 11564, 11568, 12144, 12144, 12146, 12148, 12156, 12157, 12158, 12159, 12271, 12499, 12512, 12513, 12681, 13209, 13229, 14238, 14388, 16551, 16555, 16583, 16650, 16659, 16744, 16747, 16764, 16772, 16791, 16800, 16859, 16990, 17605, 18216, 18222, 18231, 18238, 18492, 18931, 19555, 19792, 19808, 19809, 19817, 19818, 19820, 19826, 19829, 19896, 19897, 19906, 19918, 19941, 19980, 20356, 20842, 21024, 21233, 21448, 21457, 21487, 21499, 21507, 21529, 21541, 21549, 21560, 21569, 21580, 21703, 25097, 29037, 29047, 29549, 29599, 29618, 29629, 29634, 30139, 30142, 30157, 30164, 30166, 30168, 30199, 30601, 31090, 32468, 33851, 34495, 34496, 34617, 34854, 34882, 34958, 34985, 34992, 35009, 35017, 35020, 35058, 35073, 35329, 35335, 35336, 35340, 35384, 35392, 35396, 35532, 35540, 35557, 35560, 35601, 35617, 35641, 35650, 35670, 35704, 35706, 35708, 35711, 35728, 35922, 36110, 37000, 37075, 37320
\tl_set_eq:NN ..... 104, 173, 573, 3433, 7018, 7442, 7925, 8493, 9699, 9707, 11170, 12112, 12112, 12116, 13282, 14167, 14176, 16499, 16500, 16501, 16502, 18127, 18128, 18129, 18130, 19710, 19711, 19712, 19713, 20932, 21588, 21612, 21692, 21713, 24754, 29613, 34872, 34987, 35072, 35603, 35623, 35646, 35965
\tl_set_from_file:Nnn ..... 37604
\tl_set_from_file_x:Nnn ..... 37606
\tl_set_rescan:Nnn ..... 118, 119, 263, 646, 682, 12262, 12264, 12270, 12301, 12302
.tl_set_x:N ..... 228, 21417
\tl_show:N ..... 110, 111, 173, 447, 4103, 13185, 13185, 13186, 13979
\tl_show:n ..... 83, 110, 111, 383, 710, 1056, 1393, 2180, 2197, 8530, 10256, 10479, 13185, 13205, 13205, 13975, 18019, 18755, 18825, 18831, 18837, 18843, 24778, 36853
\tl_show_analysis:N ..... 37592
\tl_show_analysis:n ..... 37594
\tl_sort:Nn 117, 3433, 3433, 3434, 12800
\tl_sort:nN ..... 117, 426, 427, 3603, 3603, 12800
\tl_tail:N ..... 114, 744, 5837, 12800, 12847, 14383
\tl_tail:n ..... 114, 12800, 12839, 12846, 12847
\tl_to_lowercase:n ..... 37608
\tl_to_str:N ..... 92, 108, 121, 539, 636, 714, 10625, 10636, 11511, 11525, 12690, 12690, 12691, 13234, 13235, 13336, 13403, 13411, 13978, 13985, 14545, 18728
\tl_to_str:n ..... 51, 53, 74, 92, 108, 118, 119, 121, 131, 132, 199, 200, 223, 355, 366, 539, 688, 692, 714, 720, 726, 879, 901, 1455, 1455, 1478, 1569, 1574,

```

1663, 1748, 2235, 2916, 2930, 2933,  
 2940, 2944, 3228, 3260, 3278, 5211,  
 6170, 7302, 7461, 7529, 7552, 7626,  
 8964, 9492, 9493, 10139, 10140,  
 10141, 10142, 10547, 10563, 10928,  
 11057, 11089, 11178, 12283, 12377,  
 12452, [12689](#), 12689, 13206, 13221,  
 13298, 13337, 13411, 13419, 13568,  
 13590, 13614, 13621, 13675, 13682,  
 13756, 13775, 13786, 13811, 13819,  
 13827, 13833, 13845, 13856, 13947,  
 14081, 14194, 14199, 14264, 15284,  
 16429, 16440, 17880, 17897, 17941,  
 18026, 19090, 19209, 19213, 19243,  
 19244, 19278, 19293, 19295, 19297,  
 19386, 19659, 19664, 19779, 19838,  
 19920, 19943, 19966, 20076, 20196,  
 20396, 20495, 21873, 21947, 21949,  
 21955, 21956, 22434, 22694, 22698,  
 22715, 22909, 22910, 23523, 23524,  
 23529, 23533, 28177, 28231, 28305,  
 28812, 29111, 29968, 30636, 30640,  
 30645, 34638, 34711, 34964, 35574,  
 35767, 36373, 36594, 36595, 36613,  
 36858, 36860, 36864, 36866, 36871,  
 36873, 37150, 37383, 37405, 37408  
`\tl_to_uppercase:n` ..... 37610  
`\tl_trim_spaces:N` .....  
     ... [110](#), [12744](#), 12754, 12758  
`\tl_trim_spaces:n` ... [110](#), 698, 958,  
     10976, [12744](#), 12744, 12750, 12755,  
     12757, 16540, 16542, 29847, 37413  
`\tl_trim_spaces_apply:nN` .....  
     ... [110](#), 930, 10973, [12744](#), 12751,  
     12753, 18073, 18534, 18621, 18694  
`\tl_upper_case:n` ..... 37661, 37668  
`\tl_upper_case:nn` ..... 37661, 37671  
`\tl_use:N` . [109](#), [213](#), [217](#), [220](#), 9062,  
     9191, [12692](#), 12692, 12700, 21146,  
     29614, 29621, 29626, 29630, 29635,  
     29638, 29656, 29657, 29799, 29819  
`\g_tmpa_tl` ..... 120, [13247](#)  
`\l_tmpa_tl` ..... 6, 58, [118](#), [119](#),  
     1213, 1215, 1232, 1315, 1317, 1321,  
     1323, 1327, 1329, 1333, 1335, [13249](#)  
`\g_tmpb_tl` ..... 120, [13247](#)  
`\l_tmpb_tl` ..... [119](#), 1214,  
     1215, 1230, 1232, 1316, 1317, 1322,  
     1323, 1328, 1329, 1334, 1335, [13249](#)  
 tl internal commands:  
     `\__tl_act:NNNn` .....  
         ... [704](#), [705](#), [12718](#), [12970](#), 13021, 13037  
     `\__tl_act_count_group:n` [12720](#), [12727](#)  
     `\__tl_act_count_group:nn` ..... [12714](#)  
     `\__tl_act_count_normal:N` [12719](#), [12725](#)  
     `\__tl_act_count_normal:nN` ..... [12714](#)  
     `\__tl_act_count_space:` . [12721](#), [12726](#)  
     `\__tl_act_count_space:n` ..... [12714](#)  
     `\__tl_act_end:wn` .....  
         ... [696](#), [12970](#), 13005, 13009  
     `\__tl_act_group:nwNNN` .....  
         ... [12970](#), [12992](#), 13007  
     `\__tl_act_if_head_is_space:nTF` ..  
         ... [704](#), [12970](#), [12972](#), [12988](#), [12997](#)  
     `\__tl_act_if_head_is_space:w` ...  
         ... [12970](#), [12974](#), [12978](#)  
     `\__tl_act_if_head_is_space_-`  
         `true:w` ..... [12970](#), [12975](#), [12981](#)  
     `\__tl_act_loop:w` ..... [704](#), [12970](#),  
         12986, 13001, 13011, 13018, 13024  
     `\__tl_act_normal:NwNNN` .....  
         ... [12970](#), [12993](#), [12998](#)  
     `\__tl_act_output:n` . [705](#), [12970](#), 13028  
     `\__tl_act_result:n` ... [705](#), 13005,  
         13026, 13028, 13029, 13030, 13031  
     `\__tl_act_reverse` ..... [705](#)  
     `\__tl_act_reverse_output:n` ...  
         ... [12970](#), 13030, 13046, 13048, 13050  
     `\__tl_act_space:wwNNN` .....  
         ... [704](#), [12970](#), [12989](#), 13015  
     `\__tl_analysis:n` .....  
         ... [437](#), [447](#), [3796](#), 3796, 4082, 4110, 4122  
     `\__tl_analysis_a:n` .. 3800, [3825](#), 3825  
     `\__tl_analysis_a_bgroup:w` .....  
         ... [3856](#), [3878](#), 3880  
     `\__tl_analysis_a_cs:ww` .....  
         ... [3935](#), 3949, 3952  
     `\__tl_analysis_a_egroup:w` .....  
         ... [3858](#), [3878](#), 3883  
     `\__tl_analysis_a_group:nw` .....  
         ... [3878](#), 3881, 3884, 3886  
     `\__tl_analysis_a_group_aux:w` ...  
         ... [3878](#), 3894, 3896  
     `\__tl_analysis_a_group_auxii:w` ...  
         ... [3878](#), 3901, 3904  
     `\__tl_analysis_a_group_test:w` ...  
         ... [3878](#), 3906, 3911  
     `\__tl_analysis_a_loop:w` ... [3832](#),  
         [3835](#), 3835, 3876, 3918, 3932, 3950  
     `\__tl_analysis_a_safe:N` .....  
         ... [3857](#), 3899, [3935](#), 3935  
     `\__tl_analysis_a_space:w` .....  
         ... [3855](#), [3861](#), 3861  
     `\__tl_analysis_a_space_test:w` ...  
         ... [440](#), [3861](#), 3863, 3868  
     `\__tl_analysis_a_store:` .....  
         ... [440](#), 3872, 3914, [3920](#), 3920

```

\__tl_analysis_a_type:w ..... 3836, 3837, 3837
\__tl_analysis_b:n .. 3801, 3963, 3963
\__tl_analysis_b_char:Mn ..... 452, 3990, 3997, 3997, 4305
\__tl_analysis_b_char_aux:nww ..... 444, 3991, 3997, 4019
\__tl_analysis_b_cs:Nww ..... 3993, 4025, 4025
\__tl_analysis_b_cs_test:ww .... 4025, 4028, 4030
\__tl_analysis_b_loop:w ..... 445, 3963, 3967, 3971, 4071, 4076
\__tl_analysis_b_normal:wwN .... 3976, 3981, 3983, 4046
\__tl_analysis_b_normals:ww .... 444, 445, 3973, 3976, 3976, 4022, 4032
\__tl_analysis_b_special:w .... 3979, 4043, 4045
\__tl_analysis_b_special_char:wN ..... 4043, 4060, 4068
\__tl_analysis_b_special_space:w ..... 4043, 4062, 4073
\__tl_analysis_char_arg:Nw ..... 4201, 4201, 4370, 4429
\__tl_analysis_char_arg_aux:Nw .. 4201, 4204, 4207
\l__tl_analysis_char_token .... 434, 440, 3751, 3865, 3870, 3908, 3913
\__tl_analysis_cs_space_count>NN ..... 3780, 3780, 3949, 4028
\__tl_analysis_cs_space_count:w .. 3780, 3784, 3788, 3792
\__tl_analysis_cs_space_count_- end:w ..... 3780, 3786, 3794
\__tl_analysis_disable:n ..... 3805, 3807, 3816, 3827, 3893, 3946
\__tl_analysis_extract_charcode: ..... 3774, 3774, 3888, 4332
\__tl_analysis_extract_charcode_- aux:w ..... 3774, 3776, 3779
\l__tl_analysis_index_int ..... 441, 442, 3770, 3830, 3833, 3871, 3889, 3926, 3929, 3955, 3957, 4049
\__tl_analysis_map:Nn 4078, 4084, 4087
\__tl_analysis_map:NwNw ..... 4078, 4090, 4096, 4100
\l__tl_analysis_nesting_int .... 438, 3771, 3831, 3922, 3931
\l__tl_analysis_next_token .... 434, 454, 3751, 4355, 4360, 4442
\l__tl_analysis_normal_int .... 3769, 3829, 3874, 3916, 3927, 3930, 3947, 3956, 3961
\g__tl_analysis_result_tl ..... 446, 3773, 3965, 4091, 4127
\__tl_analysis_show: ..... 4112, 4123, 4125, 4125
\__tl_analysis_show:Nn ..... 4116, 4117, 4119, 4120
\__tl_analysis_show:NNN ..... 4102, 4103, 4105, 4106
\__tl_analysis_show_active:n ... 4140, 4169, 4171
\__tl_analysis_show_cs:n ..... 4136, 4169, 4169
\c__tl_analysis_show_etc_str ... 448, 4189, 4191, 4460
\__tl_analysis_show_long:nn .... 4169, 4170, 4172, 4173
\__tl_analysis_show_long_- aux:nnnn ... 4169, 4175, 4180, 4195
\__tl_analysis_show_loop:wNw ... 4125, 4127, 4131, 4147
\__tl_analysis_show_normal:n ... 4143, 4149, 4149
\__tl_analysis_show_value:N ... 4154, 4154, 4178
\l__tl_analysis_token ..... 434, 435, 439, 440, 449, 3751, 3777, 3836, 3840, 3843, 3846, 3894, 3898, 3913, 4203, 4330, 4339, 4344, 4365, 4425, 4437, 4442
\l__tl_analysis_type_int 439, 441, 442, 3772, 3839, 3854, 3922, 3924, 3928
\__tl_build_begin>NN ..... 37169, 37170, 37172, 37173, 37214
\__tl_build_begin:NNN ..... 1402, 37169, 37174, 37175
\__tl_build_end_loop>NN ..... 37257, 37260, 37265, 37267, 37273
\__tl_build_get:NNN ..... 37243, 37244, 37245, 37259, 37264
\__tl_build_get:w ..... 37243, 37246, 37247, 37253
\__tl_build_get_end:w ..... 37243, 37251, 37255
\__tl_build_last>NNn ..... 1401, 1402, 37181, 37186, 37206, 37210, 37224, 37225, 37247
\__tl_build_put:nn ..... 1402, 37186, 37221, 37223, 37238
\__tl_build_put:nw ..... 1402, 37186, 37223, 37224
\__tl_build_put_left>NNn ..... 37226, 37227, 37230, 37232
\__tl_case:NnTF ..... 12584, 12589, 12594, 12599, 12601

```

\\_\\_tl\\_case:nnTF . . . . . [12581](#)  
 \\_\\_tl\\_case:Nw . . . . .  
     . . . . . [12581](#), [12602](#), [12603](#), [12607](#)  
 \\_\\_tl\\_case\_end:nw [12581](#), [12606](#), [12612](#)  
 \\_\\_tl\\_count:n . . . . .  
     . . . . . [696](#), [12701](#), [12704](#), [12709](#), [12711](#)  
 \\_\\_tl\\_head\\_aux:n . . . . . [12805](#), [12807](#)  
 \\_\\_tl\\_head\\_auxi:nw . . . . .  
     . . . . . [12800](#), [12817](#), [12819](#), [12832](#)  
 \\_\\_tl\\_head\\_auxii:n [12800](#), [12821](#), [12825](#)  
 \\_\\_tl\\_head\\_exp\\_not:w . . . . .  
     . . . . . [702](#), [12848](#), [12852](#), [12866](#), [12915](#)  
 \\_\\_tl\\_if\\_blank\_p:NNw . . . . . [12473](#)  
 \\_\\_tl\\_if\\_empty\_if:n . . . . . [688](#), [689](#),  
     [789](#), [12460](#), [12460](#), [12467](#), [12476](#),  
     [12543](#), [12570](#), [12574](#), [12827](#), [12966](#)  
 \\_\\_tl\\_if\\_head\_eq\_empty\_arg:w . . . . .  
     [700](#), [702](#), [12848](#), [12852](#), [12866](#), [12917](#)  
 \\_\\_tl\\_if\\_head\_eq\_meaning\_-  
     normal:nN . . . . . [12883](#), [12887](#)  
 \\_\\_tl\\_if\\_head\_eq\_meaning\_-  
     special:nN . . . . . [12884](#), [12896](#)  
 \\_\\_tl\\_if\\_head\_is\_group\_fi\_-  
     false:w . . . . . [12939](#), [12945](#), [12953](#)  
 \\_\\_tl\\_if\\_head\_is\_N\_type\_auxi:w . . . . .  
     . . . . . [702](#), [12919](#), [12922](#), [12930](#)  
 \\_\\_tl\\_if\\_head\_is\_N\_type\_auxii:n . . . . .  
     . . . . . [12934](#), [12937](#)  
 \\_\\_tl\\_if\\_head\_is\_N\_type\_auxii:nn . . . . .  
     . . . . . [12919](#)  
 \\_\\_tl\\_if\\_head\_is\_N\_type\_auxiii:n . . . . .  
     . . . . . [12919](#)  
 \\_\\_tl\\_if\\_head\_is\_space:w . . . . .  
     . . . . . [12954](#), [12957](#), [12964](#)  
 \\_\\_tl\\_if\\_novalue:w . . . . .  
     . . . . . [691](#), [12538](#), [12543](#), [12549](#)  
 \\_\\_tl\\_if\\_recursion\_tail\_break:nN . . . . .  
     . . . . . [706](#), [12259](#), [12259](#), [13073](#)  
 \\_\\_tl\\_if\\_recursion\_tail\_stop:nTF . . . . .  
     . . . . . [12259](#)  
 \\_\\_tl\\_if\\_recursion\_tail\_stop:p:n . . . . .  
     . . . . . [12259](#)  
 \\_\\_tl\\_if\\_single:nnw . . . . .  
     . . . . . [691](#), [12556](#), [12558](#), [12566](#)  
 \\_\\_tl\\_internal\_a\_tl . . . . .  
     . . . . . [710](#), [4424](#), [4432](#), [4443](#), [4454](#), [12264](#),  
     [12266](#), [12269](#), [12494](#), [12512](#), [12516](#),  
     [13209](#), [13215](#), [13229](#), [13230](#), [13235](#)  
 \\_\\_tl\\_internal\_b\_tl . . . . .  
     . . . . . [12494](#), [12499](#), [12502](#), [12513](#), [12516](#)  
 \\_\\_tl\\_item:nn . . . . .  
     . . . . . [13057](#), [13059](#), [13071](#), [13076](#)  
 \\_\\_tl\\_item\\_aux:nn [13057](#), [13060](#), [13065](#)  
  
 \\_\\_tl\\_map\\_function:Nnnnnnnnn . . . . .  
     . . . . . [694](#), [12614](#), [12616](#), [12624](#), [12629](#), [12643](#)  
 \\_\\_tl\\_map\\_function\_end:w . . . . .  
     . . . . . [693](#), [12614](#), [12627](#), [12631](#), [12635](#)  
 \\_\\_tl\\_map\\_tokens:nnnnnnnn . . . . .  
     . . . . . [12654](#), [12656](#), [12664](#), [12670](#)  
 \\_\\_tl\\_map\\_tokens\_end:w . . . . .  
     . . . . . [12654](#), [12667](#), [12672](#), [12676](#)  
 \\_\\_tl\\_map\\_variable:Nnn . . . . .  
     . . . . . [12678](#), [12679](#), [12680](#)  
 \\_\\_tl\\_peek\\_analysis\\_active\\_str:n . . . . .  
     . . . . . [4208](#), [4379](#), [4381](#)  
 \\_\\_tl\\_peek\\_analysis\\_char:N . . . . .  
     . . . . . [4208](#), [4286](#), [4296](#)  
 \\_\\_tl\\_peek\\_analysis\\_char:w . . . . .  
     . . . . . [452](#), [4208](#), [4309](#), [4317](#)  
 \\_\\_tl\\_peek\\_analysis\\_collect:n . . . . .  
     . . . . . [4208](#), [4429](#), [4430](#)  
 \\_\\_tl\\_peek\\_analysis\\_collect:w . . . . .  
     . . . . . [4208](#), [4426](#), [4428](#), [4447](#)  
 \\_\\_tl\\_peek\\_analysis\\_collect\_-  
     end:NNN . . . . . [4208](#), [4444](#), [4449](#)  
 \\_\\_tl\\_peek\\_analysis\\_collect\_-  
     loop: . . . . . [4208](#), [4433](#), [4435](#)  
 \\_\\_tl\\_peek\\_analysis\\_collect\_-  
     test: . . . . . [4208](#), [4438](#), [4440](#)  
 \\_\\_tl\\_peek\\_analysis\\_cs:N . . . . .  
     . . . . . [4208](#), [4288](#), [4292](#)  
 \\_\\_tl\\_peek\\_analysis\\_escape: . . . . .  
     . . . . . [4208](#), [4377](#), [4422](#)  
 \\_\\_tl\\_peek\\_analysis\\_exp:N . . . . .  
     . . . . . [4208](#), [4246](#), [4254](#)  
 \\_\\_tl\\_peek\\_analysis\\_exp\\_active:N . . . . .  
     . . . . . [4208](#), [4268](#), [4278](#)  
 \\_\\_tl\\_peek\\_analysis\\_explicit:n . . . . .  
     . . . . . [4208](#), [4376](#), [4391](#)  
 \\_\\_tl\\_peek\\_analysis\\_loop>NNn . . . . .  
     . . . . . [4208](#), [4218](#), [4222](#), [4224](#)  
 \\_\\_tl\\_peek\\_analysis\\_next: . . . . .  
     . . . . . [4208](#), [4347](#), [4350](#)  
 \\_\\_tl\\_peek\\_analysis\\_nextii: . . . . .  
     . . . . . [4208](#), [4354](#), [4357](#)  
 \\_\\_tl\\_peek\\_analysis\\_nonexp:N . . . . .  
     . . . . . [4208](#), [4249](#), [4280](#)  
 \\_\\_tl\\_peek\\_analysis\\_normal:N . . . . .  
     . . . . . [4345](#)  
 \\_\\_tl\\_peek\\_analysis\\_retest: . . . . .  
     . . . . . [453](#), [4208](#), [4340](#), [4342](#)  
 \\_\\_tl\\_peek\\_analysis\\_special: . . . . .  
     . . . . . [4208](#), [4251](#), [4328](#)  
 \\_\\_tl\\_peek\\_analysis\\_str: . . . . .  
     . . . . . [4208](#), [4359](#), [4362](#)  
 \\_\\_tl\\_peek\\_analysis\\_str:n . . . . .  
     . . . . . [4208](#), [4370](#), [4371](#)

```

\__tl_peek_analysis_str:w ..... 4208, 4366, 4369
\__tl_peek_analysis_test: ..... 4208, 4233, 4235
\c__tl_peek_catcodes_tl ..... 3755
\l__tl_peek_charcode_int ..... 4200, 4331, 4333, 4336, 4375
\l__tl_peek_code_tl 451, 3754, 4227,
4256, 4259, 4276, 4293, 4304, 4313,
4383, 4389, 4393, 4420, 4452, 4458
\__tl_quark_if_nil:n ..... 12260
\__tl_quark_if_nil:nTF ..... 12385
\__tl_range:Nnnn ..... 13087, 13089, 13090, 37277, 37282
\__tl_range:nnNn . 13087, 13100, 13110
\__tl_range:nnnNn 13087, 13094, 13098
\__tl_range:w 707, 13087, 13089, 13128
\__tl_range_braced:w ..... 707, 1405, 37275, 37277, 37283
\__tl_range_collect:nn 1405, 13087,
13130, 13139, 13146, 13151, 13165
\__tl_range_collect_braced:w ...
707, 1405, 37275, 37284, 37287, 37290
\__tl_range_collect_group:nN . 13087
\__tl_range_collect_group:nn ...
13155, 13164
\__tl_range_collect_N:nN ..... 13087, 13154, 13163
\__tl_range_collect_space:nw ...
13087, 13147, 13162
\__tl_range_collect_unbraced:w ...
37275, 37286, 37295, 37298
\__tl_range_items:nnNn ..... 707
\__tl_range_normalize:nn ..... 13102, 13106, 13166, 13166
\__tl_range_skip:w ..... 707, 13087, 13117, 13119, 13122
\__tl_range_skip_spaces:n ...
13087, 13131, 13133, 13136
\__tl_range_unbraced:w ..... 37275, 37282, 37285
\__tl_replace:NnNNNnn ..... 684, 685, 12361, 12363,
12365, 12367, 12372, 12372, 12383
\__tl_replace_auxi:NnnNNNnn ...
686, 12372, 12386, 12387, 12394, 12397
\__tl_replace_auxii:nNNNnn ...
685, 686, 12372, 12390, 12398, 12400
\__tl_replace_next:w ..... 684, 686, 687, 12365,
12367, 12372, 12405, 12425, 12427
\__tl_replace_next_aux:w ..... 12372, 12414, 12425
\__tl_replace_wrap:w ..... 684, 686, 687, 12361,
12363, 12372, 12403, 12407, 12426
\__tl_rescan>NNw ..... 681, 12262, 12289, 12296, 12346, 12351
\__tl_rescan_aux: 12262, 12265, 12268
\c__tl_rescan_marker_tl .....
683, 12261, 12288, 12296, 12326, 12358
\__tl_reverse_group_preserve:n .. .
13039, 13047
\__tl_reverse_group_preserve:nn ..
13032
\__tl_reverse_items:nwNwn .. .
12728, 12730, 12731, 12735, 12738
\__tl_reverse_items:wn .. .
12728, 12732, 12739, 12742
\__tl_reverse_normal:N . 13038, 13045
\__tl_reverse_normal:nn ..... 13032
\__tl_reverse_space: .. 13040, 13049
\__tl_reverse_space:n ..... 13032
\__tl_set_rescan:nNN .. .
681, 683, 12283, 12305, 12305
\__tl_set_rescan>NNnn .. .
682, 12262, 12271, 12273, 12274
\__tl_set_rescan_multi:nNN .. .
681, 683, 12262, 12286, 12313, 12335
\__tl_set_rescan_single:nnNN .. .
683, 12305, 12316, 12320, 12332
\__tl_set_rescan_single:NNww .. 683
\__tl_set_rescan_single_aux:nnNNN .. .
12305, 12325, 12338
\__tl_set_rescan_single_aux:w .. .
683, 12305, 12343, 12357
\__tl_show:n ..... 13205, 13206, 13207
\__tl_show>NN .. .
13185, 13185, 13187, 13189
\__tl_show:w ..... 13205, 13209, 13219
\__tl_tl_head:w . 12800, 12837, 12890
\__tl_tmp:w ..... 690,
697, 12531, 12532, 12538, 12551,
12760, 12799, 12970, 12985, 13251
\__tl_trim_mark: ..... 697, 698,
12747, 12752, 12760, 12767, 12768,
12775, 12779, 12781, 12784, 12797
\__tl_trim_spaces:nn .. .
930, 12746, 12752, 12760, 12762
\__tl_trim_spaces_auxi:w .. .
697, 12760, 12764, 12775, 12778, 12784
\__tl_trim_spaces_auxii:w .. .
698, 12760, 12768, 12783
\__tl_trim_spaces_auxiii:w .. .
698, 12760, 12769, 12786, 12789, 12793
\__tl_trim_spaces_auxiv:w .. .
698, 12760, 12771, 12795

```

\\_\_tl\_use\_none\_delimit\_by\_q\_act\_-  
     stop:w ..... 12970  
 \\_\_tl\_use\_none\_delimit\_by\_s\_act\_-  
     stop:w ..... 13004, 13009  
 \\_\_tl\_use\_none\_delimit\_by\_s\_-  
     stop:w ..... 693, 12614,  
         12626, 12633, 12637, 12666, 12674  
 token commands:  
 \c\_alignment\_token .....  
     ..... 187, 875, 1357, 4007,  
         19041, 19107, 19146, 29975, 35424  
 \c\_catcode\_letter\_token .... 187,  
     876, 4003, 19053, 19107, 19175, 29987  
 \c\_catcode\_other\_token .... 187,  
     876, 4001, 19056, 19107, 19180, 29990  
 \c\_group\_begin\_token ..... 187  
 \c\_group\_end\_token ..... 187  
 \c\_math\_subscript\_token .....  
     ..... 187, 876, 4011, 19047,  
         19107, 19165, 29946, 29981, 35439  
 \c\_math\_superscript\_token .....  
     ..... 187, 876, 4009,  
         19044, 19107, 19160, 29978, 35440  
 \c\_math\_toggle\_token .....  
     ..... 187, 875, 4005,  
         19038, 19107, 19141, 29943, 29972  
 \c\_parameter\_token .....  
     .. 187, 532, 875, 19107, 19150, 19153  
 \c\_space\_token .... 38, 108, 119,  
     187, 193, 314, 701, 876, 2774, 3840,  
     3870, 4013, 4203, 4240, 4395, 4841,  
     4882, 7195, 10737, 12870, 12908,  
     19050, 19107, 19170, 19506, 19531,  
     19644, 29947, 29984, 30270, 37338  
 \token\_case\_catcode:Nn .....  
     ..... 191, 19437, 19437  
 \token\_case\_catcode:NnTF .....  
     191, 19437, 19439, 19441, 19443, 35437  
 \token\_case\_charcode:Nn .....  
     ..... 191, 19437, 19445  
 \token\_case\_charcode:NnTF .....  
     .... 191, 19437, 19447, 19449, 19451  
 \token\_case\_meaning:Nn .....  
     ..... 191, 19437, 19453  
 \token\_case\_meaning:NnTF ... 191,  
     6061, 6076, 6109, 6119, 6130, 8538,  
     19437, 19455, 19457, 19459, 35444  
 \token\_get\_arg\_spec:N ..... 37612  
 \token\_get\_prefix\_spec:N ..... 37614  
 \token\_get\_replacement\_spec:N .. 37616  
 \token\_if\_active:N ..... 19183  
 \token\_if\_active:NTF .... 189, 19183  
 \token\_if\_active\_p:N .....  
     ..... 189, 19183, 30377, 30549,  
         30957, 30975, 31005, 31523, 32420  
 \token\_if\_alignment:N ..... 19144  
 \token\_if\_alignment:NTF .. 188, 19144  
 \token\_if\_alignment\_p:N .. 188, 19144  
 \token\_if\_chardef:NTF 190, 4158, 19256  
 \token\_if\_chardef\_p:N .....  
     ..... 190, 19256, 29915  
 \token\_if\_cs:N ..... 19220  
 \token\_if\_cs:NTF 189, 19220, 30373,  
     30804, 31163, 31191, 31295, 31321,  
     31485, 31967, 32158, 32248, 32427  
 \token\_if\_cs\_p:N .....  
     ..... 189, 19220, 30548, 31668,  
         31731, 31766, 31821, 32009, 32419  
 \token\_if\_dim\_register:NTF .....  
     ..... 190, 4160, 19256  
 \token\_if\_dim\_register\_p:N 190, 19256  
 \token\_if\_eq\_catcode>NN ..... 19193  
 \token\_if\_eq\_catcode:NNTF .....  
     ..... 189, 191, 192, 314, 2774,  
         19193, 19438, 19440, 19442, 19444  
 \token\_if\_eq\_catcode\_p:NN 189, 19193  
 \token\_if\_eq\_charcode>NN ..... 19198  
 \token\_if\_eq\_charcode:NNTF .....  
     .. 189, 191, 192, 314, 4882, 4887,  
         5522, 5722, 5735, 5737, 5775, 5913,  
         7181, 7195, 9941, 10737, 19198,  
         19446, 19448, 19450, 19452, 26777  
 \token\_if\_eq\_charcode\_p:NN 189, 19198  
 \token\_if\_eq\_meaning>NN ..... 19188  
 \token\_if\_eq\_meaning:NNTF .....  
     ..... 189, 191-  
         193, 314, 2777, 2788, 5218, 5225,  
         5528, 5561, 5710, 5733, 5765, 5900,  
         5908, 5911, 7250, 7256, 7283, 7290,  
         7308, 7331, 7348, 10789, 19188,  
         19454, 19456, 19458, 19460, 23030,  
         24041, 24100, 24797, 25045, 25047,  
         25052, 25116, 25302, 27374, 30305,  
         30311, 30330, 30356, 30524, 30562,  
         30758, 30784, 32335, 32376, 35458  
 \token\_if\_eq\_meaning\_p:NN .....  
     .. 189, 19188, 30010, 30270, 30272,  
         30276, 30286, 30287, 30294, 30295  
 \token\_if\_expandable:N ..... 19225  
 \token\_if\_expandable:NTF .....  
     ..... 189, 4156, 19225, 30004  
 \token\_if\_expandable\_p:N .. 189, 19225  
 \token\_if\_font\_selection:NTF ...  
     ..... 190, 19256  
 \token\_if\_font\_selection\_p:N ...  
     ..... 190, 19256

\token\_if\_group\_begin:N . . . . . 19129  
 \token\_if\_group\_begin:NTF 188, 19129  
 \token\_if\_group\_begin\_p:N 188, 19129  
 \token\_if\_group\_end:N . . . . . 19134  
 \token\_if\_group\_end:NTF . 188, 19134  
 \token\_if\_group\_end\_p:N . 188, 19134  
 \token\_if\_int\_register:NTF . . . . .  
     . . . . . 190, 4161, 19256  
 \token\_if\_int\_register\_p:N 190, 19256  
 \token\_if\_letter:N . . . . . 878, 19173  
 \token\_if\_letter:NTF . . . . . 189, 19173  
 \token\_if\_letter\_p:N . . . . .  
     . . . . . 189, 19173, 30954, 30972, 31520  
 \token\_if\_long\_macro:NTF . 189, 19256  
 \token\_if\_long\_macro\_p:N . 189, 19256  
 \token\_if\_macro:N . . . . . 19205  
 \token\_if\_macro:NTF . . . . .  
     . . . . . 189, 2240, 2249, 2258, 19203, 19381  
 \token\_if\_macro\_p:N . . . . . 189, 19203  
 \token\_if\_math\_subscript:N . . 19163  
 \token\_if\_math\_subscript:NTF . .  
     . . . . . 188, 19163  
 \token\_if\_math\_subscript\_p:N . .  
     . . . . . 188, 19163  
 \token\_if\_math\_superscript:N . 19157  
 \token\_if\_math\_superscript:NTF . .  
     . . . . . 188, 19157  
 \token\_if\_math\_superscript\_p:N . .  
     . . . . . 188, 19157  
 \token\_if\_math\_toggle:N . . . . . 19139  
 \token\_if\_math\_toggle:NTF 188, 19139  
 \token\_if\_math\_toggle\_p:N 188, 19139  
 \token\_if\_mathchardef:NTF . . . . .  
     . . . . . 190, 4159, 19256  
 \token\_if\_mathchardef\_p:N . . . . .  
     . . . . . 190, 19256, 29916  
 \token\_if\_muskip\_register:NTF . . . . .  
     . . . . . 190, 19256  
 \token\_if\_muskip\_register\_p:N . . . . .  
     . . . . . 190, 19256  
 \token\_if\_other:N . . . . . 19178  
 \token\_if\_other:NTF . . . . . 189, 19178  
 \token\_if\_other\_p:N . . . . . 189, 19178  
 \token\_if\_parameter:N . . . . . 19151  
 \token\_if\_parameter:NTF . . . . . 188, 19149  
 \token\_if\_parameter\_p:N . . . . . 188, 19149  
 \token\_if\_primitive:N . . . . . 19369, 19378  
 \token\_if\_primitive:NTF . . . . . 191, 19305  
 \token\_if\_primitive\_p:N . . . . . 191, 19305  
 \token\_if\_protected\_long\_-  
     macro:NTF . . . . . 190, 2671, 19256  
 \token\_if\_protected\_long\_macro\_-  
     p:N . . . . . 190, 19256, 30009, 30187, 30382  
 \token\_if\_protected\_macro:NTF . . . . .  
     . . . . . 189, 2670, 19256  
 \token\_if\_protected\_macro\_p:N . . . . .  
     . . . . . 189, 19256, 30008, 30186, 30381  
 \token\_if\_skip\_register:NTF . . . . .  
     . . . . . 191, 4162, 19256  
 \token\_if\_skip\_register\_p:N . . . . .  
     . . . . . 191, 19256  
 \token\_if\_space:N . . . . . 19168  
 \token\_if\_space:NTF . . . . . 188, 19168  
 \token\_if\_space\_p:N . . . . . 188, 19168  
 \token\_if\_toks\_register:NTF . . . . .  
     . . . . . 191, 404, 2873, 4163, 19256  
 \token\_if\_toks\_register\_p:N . . . . .  
     . . . . . 191, 19256  
 \token\_new:Nn . . . . . 37618  
 \token\_to\_meaning:N . . . . .  
     . . . . . 20, 187, 196, 877, 880, 1453,  
     1453, 1469, 1480, 1919, 2243, 2252,  
     2261, 2879, 2930, 3777, 4152, 4177,  
     5167, 8545, 13201, 13240, 19107,  
     19209, 19277, 19385, 19647, 29952  
 \token\_to\_str:N . . . . . 7,  
     21, 92, 121, 187, 196, 371, 451, 453,  
     454, 645, 702, 703, 824, 879, 1017,  
     1019, 1455, 1456, 1469, 1469, 1641,  
     1650, 1682, 1705, 1756, 1761, 1776,  
     1797, 1798, 1818, 1919, 2045, 2081,  
     2088, 2176, 2196, 2209, 2856, 2950,  
     3035, 3050, 3065, 3072, 3098, 3107,  
     3158, 3224, 3245, 3700, 3716, 3763,  
     3764, 3765, 3766, 3785, 3866, 3909,  
     3939, 3988, 4000, 4002, 4004, 4014,  
     4054, 4065, 4112, 4151, 4176, 4265,  
     4279, 4284, 4367, 4387, 4396, 4461,  
     4784, 4791, 4901, 4905, 5640, 7176,  
     7473, 7528, 7551, 7625, 8245, 8449,  
     8451, 8540, 8541, 8545, 8603, 8955,  
     9075, 10267, 10269, 10490, 10492,  
     10609, 10610, 10611, 10612, 10613,  
     10620, 10925, 10942, 10989, 11116,  
     11298, 12141, 12261, 12923, 12943,  
     13197, 13201, 13234, 13240, 13528,  
     14055, 14063, 14066, 16275, 16299,  
     16303, 16318, 16447, 16703, 17158,  
     17403, 18722, 18728, 19107, 19291,  
     19292, 19297, 19299, 19300, 19301,  
     19302, 19303, 20068, 22132, 22180,  
     22301, 22343, 22451, 22693, 22708,  
     22915, 22916, 23407, 23408, 23437,  
     23604, 23655, 23687, 23707, 23722,  
     23734, 23735, 23748, 23749, 23774,  
     23783, 23785, 23810, 23813, 23838,  
     23840, 23854, 23870, 23888, 23958,

23968, 23969, 23984, 23985, 24312,	\topmark . . . . .	467
24354, 24546, 24792, 28786, 29106,	\topmarks . . . . .	569
29151, 29157, 29926, 29927, 30545,	\topskip . . . . .	468
30553, 30600, 30601, 30855, 30858,	\toucs . . . . .	1167
30901, 30904, 30913, 31089, 31090,	\tpack . . . . .	910
31862, 31864, 31882, 31883, 31898,	\tracingassigns . . . . .	570
31901, 31905, 31908, 32416, 32424,	\tracingcommands . . . . .	469
32467, 32468, 32609, 32612, 32616,	\tracingfonts . . . . .	941
32625, 32677, 33040, 33630, 33850,	\tracinggroups . . . . .	571
34781, 37022, 37382, 37405, 37408	\tracingifs . . . . .	572
token internal commands:	\tracinglostchars . . . . .	470
\c__token_A_int . . . . .	\tracingmacros . . . . .	471
\__token_case>NNnTF . . . . .	\tracingnesting . . . . .	573
. . . . .	\tracingonline . . . . .	472
. . . . .	\tracingoutput . . . . .	473
. . . . .	\tracingpages . . . . .	474
. . . . .	\tracingparagraphs . . . . .	475
. . . . .	\tracingrestores . . . . .	476
. . . . .	\tracingscantokens . . . . .	574
. . . . .	\tracingstacklevels . . . . .	1193
. . . . .	\tracingstats . . . . .	477
true . . . . .	true . . . . .	258
trunc . . . . .	trunc . . . . .	254
try commands:	try_commands:	
try_require . . . . .	try_require . . . . .	11974
\ttfamily . . . . .	\ttfamily . . . . .	32504
\c_twelve . . . . .	\c_twelve . . . . .	37462
\c_two . . . . .	\c_two . . . . .	37442
\c_two_hundred_fifty_five . . . . .	\c_two_hundred_fifty_five . . . . .	37476
\c_two_hundred_fifty_six . . . . .	\c_two_hundred_fifty_six . . . . .	37478
	U	
\u . . . . .	\u . . . . .	30140, 32660, 32741, 32742, 32757,
		32758, 32767, 32768, 32781, 32782,
		32783, 32809, 32810, 32835, 32836
\uccode . . . . .	\uccode . . . . .	478
\Uchar . . . . .	\Uchar . . . . .	943
\Ucharcat . . . . .	\Ucharcat . . . . .	944
\uchyph . . . . .	\uchyph . . . . .	479
\ucs . . . . .	\ucs . . . . .	1168
\Udelcode . . . . .	\Udelcode . . . . .	945
\Udelcodenum . . . . .	\Udelcodenum . . . . .	946
\Udelimiter . . . . .	\Udelimiter . . . . .	947
\Udelimiterover . . . . .	\Udelimiterover . . . . .	948
\Udelimiterunder . . . . .	\Udelimiterunder . . . . .	949
\Uhextensible . . . . .	\Uhextensible . . . . .	950
\Umathaccent . . . . .	\Umathaccent . . . . .	951
\Umathaxis . . . . .	\Umathaxis . . . . .	952
\Umathbinbinspacing . . . . .	\Umathbinbinspacing . . . . .	953
\Umathbinclosespacing . . . . .	\Umathbinclosespacing . . . . .	954
\Umathbininnerspacing . . . . .	\Umathbininnerspacing . . . . .	955
\Umathbinopenspacing . . . . .	\Umathbinopenspacing . . . . .	956
\Umathbinopspacing . . . . .	\Umathbinopspacing . . . . .	957
\Umathbinordspacing . . . . .	\Umathbinordspacing . . . . .	958

\Umathbinpunctspacing . . . . .	959	\Umathopopenspacing . . . . .	1022
\Umathbinrelspacing . . . . .	960	\Umathopopspace . . . . .	1023
\Umathchar . . . . .	961	\Umathopordspacing . . . . .	1024
\Umathcharclass . . . . .	962	\Umathoppunctspacing . . . . .	1025
\Umathchardef . . . . .	963	\Umathoprelspacing . . . . .	1026
\Umathcharfam . . . . .	964	\Umathordbinspace . . . . .	1027
\Umathcharnum . . . . .	965	\Umathordclosespace . . . . .	1028
\Umathcharnumdef . . . . .	966	\Umathordinnerspace . . . . .	1029
\Umathcharslot . . . . .	967	\Umathordopenspace . . . . .	1030
\Umathclosebinspace . . . . .	968	\Umathorddospacing . . . . .	1031
\Umathcloseclosespace . . . . .	969	\Umathordordspacing . . . . .	1032
\Umathcloseinnerspace . . . . .	971	\Umathordpunctspacing . . . . .	1033
\Umathcloseopenspace . . . . .	973	\Umathordrelspacing . . . . .	1034
\Umathcloseopspacing . . . . .	974	\Umathoverbarkern . . . . .	1035
\Umathcloseordspacing . . . . .	975	\Umathoverbarrule . . . . .	1036
\Umathclosepunctspacing . . . . .	976	\Umathoverbarvgap . . . . .	1037
\Umathcloserelspace . . . . .	978	\Umathoverdelimiterbgap . . . . .	1038
\Umathcode . . . . .	979	\Umathoverdelimitervgap . . . . .	1040
\Umathcodenum . . . . .	980	\Umathpunctbinspace . . . . .	1042
\Umathconnectoroverlapmin . . . . .	981	\Umathpunctclosespace . . . . .	1043
\Umathfractiondelsize . . . . .	983	\Umathpunctinnerspace . . . . .	1045
\Umathfractiondenomdown . . . . .	984	\Umathpunctopenspace . . . . .	1047
\Umathfractiondenomvgap . . . . .	986	\Umathpuncttopspace . . . . .	1048
\Umathfractionnumup . . . . .	988	\Umathpunctordspacing . . . . .	1049
\Umathfractionnumvgap . . . . .	989	\Umathpunctpunctspacing . . . . .	1050
\Umathfractionrule . . . . .	990	\Umathpunctrelspacing . . . . .	1052
\Umathinnerbinspace . . . . .	991	\Umathquad . . . . .	1053
\Umathinnerclosespace . . . . .	992	\Umathradicaldegreeafter . . . . .	1054
\Umathinnerinnerspace . . . . .	994	\Umathradicaldegreebefore . . . . .	1056
\Umathinneropenspace . . . . .	996	\Umathradicaldegreeraise . . . . .	1058
\Umathinneropspacing . . . . .	997	\Umathradicalkern . . . . .	1060
\Umathinnerordspacing . . . . .	998	\Umathradicalrule . . . . .	1061
\Umathinnerpunctspacing . . . . .	999	\Umathradicalvgap . . . . .	1062
\Umathinnerrelspacing . . . . .	1001	\Umathrelbinspace . . . . .	1063
\Umathlimitabovebgap . . . . .	1002	\Umathrelclosespace . . . . .	1064
\Umathlimitabovekern . . . . .	1003	\Umathrelinnerspace . . . . .	1065
\Umathlimitabovevgap . . . . .	1004	\Umathrelopinspace . . . . .	1066
\Umathlimitbelowbgap . . . . .	1005	\Umathrelopspace . . . . .	1067
\Umathlimitbelowkern . . . . .	1006	\Umathrelordspacing . . . . .	1068
\Umathlimitbelowvgap . . . . .	1007	\Umathrelpunctspacing . . . . .	1069
\Umathnolimitssubfactor . . . . .	1008	\Umathrelrelspacing . . . . .	1070
\Umathnolimitsupfactor . . . . .	1009	\Umathskewedfractiongap . . . . .	1071
\Umathopbinspace . . . . .	1010	\Umathskewedfractionvgap . . . . .	1073
\Umathopclosespace . . . . .	1011	\Umathspaceafterscript . . . . .	1075
\Umathopenbinspace . . . . .	1012	\Umathstackdenomdown . . . . .	1076
\Umathopenclosespace . . . . .	1013	\Umathstacknumup . . . . .	1077
\Umathopeninnerspace . . . . .	1014	\Umathstackvgap . . . . .	1078
\Umathopenopenspace . . . . .	1015	\Umathsubshiftdown . . . . .	1079
\Umathopenopspacing . . . . .	1016	\Umathsubshiftdrop . . . . .	1080
\Umathopenordspacing . . . . .	1017	\Umathsubsupshiftdown . . . . .	1081
\Umathopenpunctspacing . . . . .	1018	\Umathsubsupvgap . . . . .	1082
\Umathopenrelspacing . . . . .	1019	\Umathsubtopmax . . . . .	1083
\Umathoperatorsize . . . . .	1020	\Umathsupbottommin . . . . .	1084
\Umathopinspace . . . . .	1021	\Umathsupshiftdrop . . . . .	1085

\Umathsupshiftup . . . . .	1086
\Umathsupsubbottommax . . . . .	1087
\Umathunderbarkern . . . . .	1088
\Umathunderbarrule . . . . .	1089
\Umathunderbarvgap . . . . .	1090
\Umathunderdelimterbgap . . . . .	1091
\Umathunderdelimtervgap . . . . .	1093
<b>undefined commands:</b>	
<b>.undefine:</b> . . . . .	<a href="#">228</a> , <a href="#">21433</a>
\underline . . . . .	480
\unexpanded . . . . .	<a href="#">575</a> , <a href="#">2758</a> , <a href="#">2782</a>
\unhbox . . . . .	481
\unhcopy . . . . .	482
\uniformdeviate . . . . .	942
\unkern . . . . .	483
\unless . . . . .	576
\Unosubscript . . . . .	1095
\Unosuperscript . . . . .	1096
\unpenalty . . . . .	484
\unskip . . . . .	485
\unvbox . . . . .	486
\unvcopy . . . . .	487
\Uoverdelimter . . . . .	1097
\uppercase . . . . .	488
\upshape . . . . .	32510
\uptexrevision . . . . .	1181
\uptexversion . . . . .	1182
\Uradical . . . . .	1098
\Uroot . . . . .	1099
<b>usage commands:</b>	
<b>.usage:n</b> . . . . .	<a href="#">231</a> , <a href="#">21435</a>
<b>use commands:</b>	
<b>\use:N</b> . . . . .	
<b>20</b> , <b>21</b> , <b>170</b> , <b>367</b> , <b>1521</b> , <b>1521</b> , <b>1677</b> , <b>1771</b> , <b>1884</b> , <b>1886</b> , <b>1888</b> , <b>1890</b> , <b>5105</b> , <b>5939</b> , <b>7187</b> , <b>7358</b> , <b>8579</b> , <b>8601</b> , <b>9353</b> , <b>9363</b> , <b>9366</b> , <b>9559</b> , <b>9591</b> , <b>9597</b> , <b>9604</b> , <b>9658</b> , <b>10694</b> , <b>11195</b> , <b>11285</b> , <b>17401</b> , <b>17837</b> , <b>17847</b> , <b>17952</b> , <b>17956</b> , <b>17958</b> , <b>17960</b> , <b>17961</b> , <b>17965</b> , <b>20199</b> , <b>20946</b> , <b>20952</b> , <b>21244</b> , <b>29802</b> , <b>30680</b> , <b>30808</b> , <b>30867</b> , <b>30868</b> , <b>30905</b> , <b>30915</b> , <b>30978</b> , <b>30981</b> , <b>30983</b> , <b>31011</b> , <b>31107</b> , <b>31129</b> , <b>31549</b> , <b>31565</b> , <b>31580</b> , <b>31591</b> , <b>31717</b> , <b>31734</b> , <b>31735</b> , <b>31760</b> , <b>31775</b> , <b>31777</b> , <b>31850</b> , <b>32030</b> , <b>32034</b> , <b>32041</b> , <b>32199</b> , <b>32526</b> , <b>34852</b> , <b>34887</b> , <b>34890</b> , <b>34891</b> , <b>34896</b> , <b>34898</b> , <b>34902</b> , <b>34903</b> , <b>35102</b> , <b>35183</b> , <b>35385</b> , <b>35519</b> , <b>35780</b> , <b>35805</b> , <b>35861</b> , <b>36150</b> , <b>36183</b> , <b>36362</b> , <b>36620</b>	
<b>\use:n</b> . . . . .	<a href="#">23</a> , <a href="#">25</a> , <a href="#">103</a> , <b>195</b> , <b>361</b> , <b>411</b> , <b>417</b> , <b>427</b> , <b>524</b> , <b>560</b> , <b>562</b> , <b>619</b> , <b>682</b> , <b>810</b> , <b>956</b> , <b>1017</b> , <b>1293</b> , <b>1522</b> , <b>1522</b> , <b>1528</b> , <b>1528</b> , <b>1530</b> , <b>1533</b> ,
<b>\use:nn</b> . . . . .	<a href="#">23</a> , <a href="#">1533</a> , <a href="#">1534</a> , <a href="#">2313</a> , <a href="#">4134</a> , <b>8032</b> , <b>8989</b> , <b>11023</b> , <b>12295</b> , <b>12356</b> , <b>13557</b> , <b>14111</b> , <b>20195</b> , <b>23518</b> , <b>23527</b> , <b>23531</b> , <b>26957</b> , <b>28831</b> , <b>29601</b> , <b>29905</b> , <b>35909</b> , <b>36863</b> , <b>36865</b> , <b>36870</b> , <b>36872</b>
<b>\use:nnn</b> . . . . .	<a href="#">23</a> , <a href="#">1533</a> , <a href="#">1535</a> , <a href="#">2042</a>
<b>\use:nnnn</b> . . . . .	<a href="#">23</a> , <a href="#">1533</a> , <a href="#">1536</a>
<b>\use_i:nn</b> . . . . .	<a href="#">24</a> , <a href="#">360</a> , <a href="#">365</a> – <a href="#">367</a> , <a href="#">793</a> , <a href="#">896</a> , <b>904</b> , <b>1143</b> , <b>1146</b> , <b>1159</b> , <b>1163</b> , <b>1164</b> , <b>1397</b> – <b>1399</b> , <b>1473</b> , <b>1537</b> , <b>1537</b> , <b>1587</b> , <b>1671</b> , <b>1693</b> , <b>1735</b> , <b>1834</b> , <b>1862</b> , <b>2021</b> , <b>2769</b> , <b>2799</b> , <b>2812</b> , <b>2857</b> , <b>3131</b> , <b>3202</b> , <b>3524</b> , <b>3579</b> , <b>3589</b> , <b>3599</b> , <b>3941</b> , <b>4514</b> , <b>5026</b> , <b>5037</b> , <b>5046</b> , <b>5049</b> , <b>5058</b> , <b>5945</b> , <b>8791</b> , <b>12359</b> , <b>14611</b> , <b>14616</b> , <b>14697</b> , <b>14701</b> , <b>16390</b> , <b>16583</b> , <b>16585</b> , <b>16967</b> , <b>17018</b> , <b>17055</b> , <b>17153</b> , <b>19785</b> , <b>19972</b> , <b>20030</b> , <b>20063</b> , <b>22295</b> , <b>22297</b> , <b>22674</b> , <b>23298</b> , <b>23488</b> , <b>24865</b> , <b>25201</b> , <b>25496</b> , <b>25984</b> , <b>26267</b> , <b>26786</b> , <b>26952</b> , <b>27264</b> , <b>27274</b> , <b>27278</b> , <b>27786</b> , <b>27991</b> , <b>28552</b> , <b>28577</b> , <b>34868</b> , <b>35462</b> , <b>36987</b> , <b>36989</b>
<b>\use_i:nnn</b> . . . . .	<a href="#">24</a> , <a href="#">750</a> , <b>1539</b> , <b>1539</b> , <b>2243</b> , <b>3236</b> , <b>4837</b> , <b>4862</b> , <b>7260</b> , <b>14097</b> , <b>14619</b> , <b>15125</b> , <b>16832</b> , <b>18013</b> , <b>23457</b> , <b>25453</b> , <b>26927</b> , <b>28765</b>
<b>\use_i:nnnn</b> . . . . .	<a href="#">24</a> , <a href="#">349</a> , <a href="#">577</a> , <a href="#">578</a> , <a href="#">1539</a> , <a href="#">1543</a> , <b>8573</b> , <b>8575</b> , <b>8590</b> , <b>8595</b> , <b>8611</b> , <b>8613</b>

25035, 25471, 25478, 25671, 28563  
 \use\_i\_delimit\_by\_q\_nil:nw . . . . .  
     . . . . . 25, 1551, 1551  
 \use\_i\_delimit\_by\_q\_recursion\_-  
     stop:nw . . . . .  
     . . . . . 25, 1551, 1553, 16189, 16205, 35312  
 \use\_i\_delimit\_by\_q\_recursion\_-  
     stop:w . . . . . 140  
 \use\_i\_delimit\_by\_q\_stop:nw . . . . .  
     . . . . . 25, 1551, 1552  
 \use\_i\_ii:nnn . . . . .  
     . . . . . 24, 366, 367, 1539, 1542,  
     1662, 2339, 16419, 16808, 16913, 19989  
 \use\_ii:nn . . . . . 24,  
     70, 360, 365, 500, 507, 793, 896,  
     1143, 1146, 1159, 1163, 1164, 1176,  
     1397–1399, 697, 702, 1475, 1537,  
     1538, 1589, 1695, 1730, 1735, 1836,  
     1864, 2019, 2267, 2771, 2814, 2859,  
     3943, 4516, 4876, 5028, 5034, 5039,  
     5051, 5060, 5220, 5569, 5691, 5868,  
     5951, 6269, 7530, 7553, 7627, 8797,  
     12308, 12830, 12899, 12976, 12982,  
     16392, 19786, 22875, 22898, 23300,  
     24664, 24865, 24866, 25498, 26788,  
     27270, 27276, 27280, 27788, 27993,  
     28423, 28554, 34869, 36993, 36995  
 \use\_ii:nnn . . . . . 24, 368,  
     1539, 1540, 2252, 4874, 5174, 12828  
 \use\_ii:nnnn . . . . .  
     . . . . . 24, 577, 578, 1539, 1544, 8590  
 \use\_ii\_i:nn . . . . . 24, 739,  
     1547, 1547, 14116, 14201, 18534, 18621  
 \use\_iii:nnn . . . . .  
     . . . . . 24, 1539, 1541, 2261, 2272, 22680  
 \use\_iii:nnnn . . . . . 24, 577,  
     578, 1539, 1545, 8590, 8612, 8614, 8615  
 \use\_iv:nnnn . . . . . 24,  
     577, 578, 1539, 1546, 8590, 8610, 24652  
 \use\_none:n . 25, 444, 446, 451, 602,  
     636, 689, 758, 793, 849, 853, 899,  
     1014, 1015, 1018, 1358, 698, 704,  
     1555, 1555, 1661, 1713, 1714, 1733,  
     1734, 1990, 2046, 2212, 2659, 2790,  
     3169, 3170, 3939, 3988, 4098, 4133,  
     4265, 4272, 4284, 4305, 4644, 4914,  
     5072, 5736, 6031, 6160, 6894, 8790,  
     8795, 9084, 9422, 9624, 9818, 9822,  
     10667, 10722, 10778, 11080, 11485,  
     11488, 12342, 12419, 12476, 12570,  
     12810, 12827, 12844, 12903, 12938,  
     12942, 12967, 13144, 13153, 14052,  
     14075, 14091, 14103, 14136, 14269,  
     14301, 14555, 14565, 14603, 14665,  
     14829, 14913, 15018, 15190, 16191,  
     16206, 16330, 16346, 16416, 16475,  
     16817, 17707, 17713, 18003, 18102,  
     18146, 18243, 18344, 18371, 18410,  
     19425, 19868, 19885, 21043, 21745,  
     22395, 22410, 22669, 22818, 22822,  
     22826, 22830, 24115, 24364, 24371,  
     24388, 24407, 24430, 24498, 24539,  
     24664, 24679, 24700, 24701, 24927,  
     24928, 25472, 25475, 26455, 28147,  
     28432, 29601, 29835, 29883, 32478,  
     32533, 35450, 36376, 36955, 36956  
 \use\_none:nn . . . . . 25, 687,  
     691, 807, 1398, 1555, 1556, 1643,  
     1651, 3268, 6562, 7273, 8568, 10723,  
     10767, 12404, 12560, 12743, 12933,  
     14160, 16655, 16839, 18071, 18385,  
     18531, 18617, 22734, 22817, 22821,  
     22825, 22829, 28142, 32479, 37002  
 \use\_none:nnn . . . . .  
     . . . . . 25, 701, 1555, 1557, 3036, 3051,  
     4164, 7803, 8018, 10724, 12890,  
     16319, 21184, 21193, 22816, 22820,  
     22824, 22828, 23457, 32525, 35665  
 \use\_none:nnnn . . . . . 25,  
     1555, 1558, 10725, 20326, 32481, 35972  
 \use\_none:nnnnn . . . . . 25, 363,  
     638, 998, 1555, 1559, 10726, 10736,  
     22948, 22982, 23008, 23016, 25061  
 \use\_none:nnnnnn . . . . .  
     . . . . . 25, 1555, 1560, 1778, 10727, 37270  
 \use\_none:nnnnnnn . . . . .  
     . . . . . 25, 998, 1555, 1561, 22950,  
     22984, 23010, 23018, 23341, 25512  
 \use\_none:nnnnnnnn . . . . .  
     . . . . . 25, 367, 1555, 1562, 1684, 3117  
 \use\_none:nnnnnnnnn . . . 25, 1555, 1563  
 \use\_none\_delimit\_by\_q\_nil:w . . . . .  
     . . . . . 25, 1548, 1548  
 \use\_none\_delimit\_by\_q\_recursion\_-  
     stop:w . . . . . 25,  
     140, 365, 1548, 1550, 16183, 16198  
 \use\_none\_delimit\_by\_q\_stop:w . . . . .  
     . . . . . 25, 758, 795, 1548, 1549  
 \use\_none\_delimit\_by\_s\_stop:w . . . . .  
     . . . . . 142, 16465, 16465  
 \useboxresource . . . . . 935  
 \usefont . . . . . 32481  
 \useimageresource . . . . . 936  
 \Uskewed . . . . . 1100  
 \Uskewedwithdelims . . . . . 1101  
 \Ustack . . . . . 1102  
 \Ustartdisplaymath . . . . . 1103  
 \Ustartmath . . . . . 1104

\Ustopdisplaymath . . . . .	1105	\vbox_to_zero:n . . . . .	279, 33128, 33133
\Ustopmath . . . . .	1106	\vbox_top:n . . . . .	279, 33124, 33126
\Usubscript . . . . .	1107	\vbox_unpack:N . . . . .	
\Usuperscript . . . . .	1108	280, 33209, 33209, 33211, 33713, 33790	
\Underdelimiter . . . . .	1109	\vbox_unpack_clear:N . . . . .	37620
\Vextensible . . . . .	1110	\vbox_unpack_drop:N . . . . .	
		281, 33209, 33210, 33212, 37621	
		\vcenter . . . . .	493
		vcoffin commands:	
		\vcoffin_gset:Nnn . . . . .	
		286, 33690, 33696, 33701	
		\vcoffin_gset:Nnw . . . . .	
		286, 33765, 33772, 33778	
		\vcoffin_gset_end: . . . . .	
		286, 33765, 33775, 33806	
		\vcoffin_set:Nnn . . . . .	
		286, 33690, 33690, 33695	
		\vcoffin_set:Nnw . . . . .	
		286, 33765, 33765, 33771	
		\vcoffin_set_end: . . . . .	
		286, 33765, 33768, 33805	
		\vfi . . . . .	1173
		\vfil . . . . .	494
		\vfill . . . . .	495
		\vfilneg . . . . .	496
		\vfuzz . . . . .	497
		\voffset . . . . .	498
		\vpack . . . . .	911
		\vrule . . . . .	499
		\vsize . . . . .	500
		\vskip . . . . .	501
		\vsplit . . . . .	502
		\vss . . . . .	503
		\vtop . . . . .	504
		W	
		\wd . . . . .	505
		\widowpenalties . . . . .	577
		\widowpenalty . . . . .	506
		\write . . . . .	89, 507
		X	
		\xdef . . . . .	508
		xetex commands:	
		\xetex_if_engine:TF . . . . .	
		37624, 37626, 37628	
		\xetex_if_engine_p: . . . . .	
		37622	
		\XeTeXcharclass . . . . .	
		715	
		\XeTeXcharglyph . . . . .	
		716	
		\XeTeXcountfeatures . . . . .	
		717	
		\XeTeXcountglyphs . . . . .	
		718	
		\XeTeXcountselectors . . . . .	
		719	
		\XeTeXcountvariations . . . . .	
		720	
		\XeTeXdashbreakstate . . . . .	
		722	

\XeTeXdefaultencoding .....	721	\XeTeXOTlanguagetag .....	757
\XeTeXfeaturecode .....	723	\XeTeXOTscripttag .....	758
\XeTeXfeaturename .....	724	\XeTeXpdffile .....	759
\XeTeXfindfeaturebyname .....	725	\XeTeXpdfpagecount .....	760
\XeTeXfindselectorbyname .....	727	\XeTeXpicfile .....	761
\XeTeXfindvariationbyname .....	729	\XeTeXrevision .....	762
\XeTeXfirstfontchar .....	731	\XeTeXselectorname .....	763
\XeTeXfonttype .....	732	\XeTeXtracingfonts .....	764
\XeTeXgenerateactualtext .....	733	\XeTeXupwardsmode .....	765
\XeTeXglyph .....	735	\XeTeXuseglyphmetrics .....	766
\XeTeXglyphbounds .....	736	\XeTeXvariation .....	767
\XeTeXglyphindex .....	737	\XeTeXvariationdefault .....	768
\XeTeXglyphname .....	738	\XeTeXvariationmax .....	769
\XeTeXinputencoding .....	739	\XeTeXvariationmin .....	770
\XeTeXinputnormalization .....	740	\XeTeXvariationname .....	771
\XeTeXinterchartokenstate .....	742	\XeTeXversion .....	772
\XeTeXinterchartoks .....	744	\xkanjiskip .....	1169
\XeTeXisdefaultselector .....	745	\xleaders .....	509
\XeTeXisexclusivefeature .....	747	\xspaceskip .....	510
\XeTeXlastfontchar .....	749	\xspcode .....	1170
\XeTeXlinebreaklocale .....	751	Y	
\XeTeXlinebreakpenalty .....	752	\ybaselineshift .....	1171
\XeTeXlinebreakskip .....	750	\year .....	511, 1334, 9099
\XeTeXOTcountfeatures .....	753	\yoko .....	1172
\XeTeXOTcountlanguages .....	754	Z	
\XeTeXOTcountsheets .....	755	Z	
\XeTeXOTfeaturetag .....	756	\c_zero .....	37438