

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

The L<sup>A</sup>T<sub>E</sub>X3 Project\*

Released 2019-01-28

## 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.

At present, the `expl3` modules are designed to be loaded on top of L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$ . In time, a L<sup>A</sup>T<sub>E</sub>X3 format will be produced based on this code. This allows the code to be used in L<sup>A</sup>T<sub>E</sub>X 2 $\varepsilon$  packages *now* while a stand-alone L<sup>A</sup>T<sub>E</sub>X3 is developed.

**While `expl3` is still experimental, the bundle is now regarded as broadly stable. The syntax conventions and functions provided are now ready for wider use. There may still be changes to some functions, but these will be minor when compared to the scope of `expl3`.**

**New modules will be added to the distributed version of `expl3` as they reach maturity.**

---

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

# Contents

<b>I</b>	<b>Introduction to <code>expl3</code> and this document</b>	<b>1</b>
1	Naming functions and variables	1
1.1	Terminological inexactitude . . . . .	3
2	Documentation conventions	3
3	Formal language conventions which apply generally	5
4	<code>TeX</code> concepts not supported by <code>L<sup>A</sup>T<sub>E</sub>X3</code>	5
<b>II</b>	<b>The <code>l3bootstrap</code> package: Bootstrap code</b>	<b>6</b>
1	Using the <code>L<sup>A</sup>T<sub>E</sub>X3</code> modules	6
<b>III</b>	<b>The <code>l3names</code> package: Namespace for primitives</b>	<b>7</b>
1	Setting up the <code>L<sup>A</sup>T<sub>E</sub>X3</code> programming language	7
<b>IV</b>	<b>The <code>l3basics</code> package: Basic definitions</b>	<b>8</b>
1	No operation functions	8
2	Grouping material	8
3	Control sequences and functions	9
3.1	Defining functions . . . . .	9
3.2	Defining new functions using parameter text . . . . .	10
3.3	Defining new functions using the signature . . . . .	11
3.4	Copying control sequences . . . . .	14
3.5	Deleting control sequences . . . . .	14
3.6	Showing control sequences . . . . .	14
3.7	Converting to and from control sequences . . . . .	15
4	Analysing control sequence names	16
5	Using or removing tokens and arguments	16
5.1	Selecting tokens from delimited arguments . . . . .	18
6	Predicates and conditionals	19
6.1	Tests on control sequences . . . . .	20
6.2	Primitive conditionals . . . . .	20
<b>V</b>	<b>The <code>l3expn</code> package: Argument expansion</b>	<b>22</b>

1	Defining new variants	22
2	Methods for defining variants	23
3	Introducing the variants	24
4	Manipulating the first argument	26
5	Manipulating two arguments	28
6	Manipulating three arguments	29
7	Unbraced expansion	30
8	Preventing expansion	31
9	Controlled expansion	32
10	Internal functions	34
<b>VI</b>	<b>The <code>l3tl</code> package: Token lists</b>	<b>35</b>
1	Creating and initialising token list variables	35
2	Adding data to token list variables	36
3	Modifying token list variables	37
4	Reassigning token list category codes	37
5	Token list conditionals	38
6	Mapping to token lists	40
7	Using token lists	42
8	Working with the content of token lists	43
9	The first token from a token list	44
10	Using a single item	47
11	Viewing token lists	49
12	Constant token lists	49
13	Scratch token lists	50
<b>VII</b>	<b>The <code>l3str</code> package: Strings</b>	<b>51</b>
1	Building strings	51

<b>2</b>	<b>Adding data to string variables</b>	<b>52</b>
<b>3</b>	<b>Modifying string variables</b>	<b>53</b>
<b>4</b>	<b>String conditionals</b>	<b>53</b>
<b>5</b>	<b>Mapping to strings</b>	<b>55</b>
<b>6</b>	<b>Working with the content of strings</b>	<b>57</b>
<b>7</b>	<b>String manipulation</b>	<b>60</b>
<b>8</b>	<b>Viewing strings</b>	<b>61</b>
<b>9</b>	<b>Constant token lists</b>	<b>62</b>
<b>10</b>	<b>Scratch strings</b>	<b>62</b>
<b>VIII</b>	<b>The <code>l3quark</code> package: Quarks</b>	<b>63</b>
<b>1</b>	<b>Quarks</b>	<b>63</b>
<b>2</b>	<b>Defining quarks</b>	<b>63</b>
<b>3</b>	<b>Quark tests</b>	<b>64</b>
<b>4</b>	<b>Recursion</b>	<b>64</b>
<b>5</b>	<b>An example of recursion with quarks</b>	<b>65</b>
<b>6</b>	<b>Scan marks</b>	<b>66</b>
<b>IX</b>	<b>The <code>l3seq</code> package: Sequences and stacks</b>	<b>68</b>
<b>1</b>	<b>Creating and initialising sequences</b>	<b>68</b>
<b>2</b>	<b>Appending data to sequences</b>	<b>69</b>
<b>3</b>	<b>Recovering items from sequences</b>	<b>69</b>
<b>4</b>	<b>Recovering values from sequences with branching</b>	<b>71</b>
<b>5</b>	<b>Modifying sequences</b>	<b>72</b>
<b>6</b>	<b>Sequence conditionals</b>	<b>72</b>
<b>7</b>	<b>Mapping to sequences</b>	<b>73</b>
<b>8</b>	<b>Using the content of sequences directly</b>	<b>74</b>
<b>9</b>	<b>Sequences as stacks</b>	<b>75</b>

10	Sequences as sets	76
11	Constant and scratch sequences	77
12	Viewing sequences	78
<b>X</b>	<b>The <code>I3int</code> package: Integers</b>	<b>79</b>
1	Integer expressions	79
2	Creating and initialising integers	80
3	Setting and incrementing integers	81
4	Using integers	82
5	Integer expression conditionals	82
6	Integer expression loops	84
7	Integer step functions	86
8	Formatting integers	87
9	Converting from other formats to integers	88
10	Random integers	89
11	Viewing integers	89
12	Constant integers	90
13	Scratch integers	90
	13.1 Direct number expansion . . . . .	91
14	Primitive conditionals	91
<b>XI</b>	<b>The <code>I3flag</code> package: Expandable flags</b>	<b>93</b>
1	Setting up flags	93
2	Expandable flag commands	94
<b>XII</b>	<b>The <code>I3prg</code> package: Control structures</b>	<b>95</b>
1	Defining a set of conditional functions	95
2	The boolean data type	97
3	Boolean expressions	99

4	Logical loops	101
5	Producing multiple copies	102
6	Detecting <b>T<sub>E</sub>X</b> 's mode	102
7	Primitive conditionals	103
8	Nestable recursions and mappings	103
8.1	Simple mappings . . . . .	103
9	Internal programming functions	104
	<b>XIII The <b>I<sub>3</sub>sys</b> package: System/runtime functions</b>	105
1	The name of the job	105
2	Date and time	105
3	Engine	105
4	Output format	106
	<b>XIV The <b>I<sub>3</sub>clist</b> package: Comma separated lists</b>	107
1	Creating and initialising comma lists	107
2	Adding data to comma lists	109
3	Modifying comma lists	109
4	Comma list conditionals	110
5	Mapping to comma lists	111
6	Using the content of comma lists directly	113
7	Comma lists as stacks	113
8	Using a single item	115
9	Viewing comma lists	115
10	Constant and scratch comma lists	116
	<b>XV The <b>I<sub>3</sub>token</b> package: Token manipulation</b>	117
1	Creating character tokens	117
2	Manipulating and interrogating character tokens	119

3	<b>Generic tokens</b>	122
4	<b>Converting tokens</b>	122
5	<b>Token conditionals</b>	123
6	<b>Peeking ahead at the next token</b>	126
7	<b>Decomposing a macro definition</b>	128
8	<b>Description of all possible tokens</b>	129
	<b>XVI The <code>l3prop</code> package: Property lists</b>	132
1	<b>Creating and initialising property lists</b>	132
2	<b>Adding entries to property lists</b>	133
3	<b>Recovering values from property lists</b>	133
4	<b>Modifying property lists</b>	134
5	<b>Property list conditionals</b>	134
6	<b>Recovering values from property lists with branching</b>	135
7	<b>Mapping to property lists</b>	135
8	<b>Viewing property lists</b>	136
9	<b>Scratch property lists</b>	137
10	<b>Constants</b>	137
	<b>XVII The <code>l3msg</code> package: Messages</b>	138
1	<b>Creating new messages</b>	138
2	<b>Contextual information for messages</b>	139
3	<b>Issuing messages</b>	140
4	<b>Redirecting messages</b>	142
	<b>XVIII The <code>l3file</code> package: File and I/O operations</b>	144
1	<b>Input–output stream management</b>	144
1.1	Reading from files . . . . .	146

<b>2</b>	<b>Writing to files</b>	<b>148</b>
2.1	Wrapping lines in output . . . . .	150
2.2	Constant input–output streams, and variables . . . . .	151
2.3	Primitive conditionals . . . . .	151
<b>3</b>	<b>File operation functions</b>	<b>151</b>
<b>XIX The <code>l3skip</code> package: Dimensions and skips</b>		<b>154</b>
1	Creating and initialising <code>dim</code> variables	154
2	Setting <code>dim</code> variables	155
3	Utilities for dimension calculations	155
4	Dimension expression conditionals	156
5	Dimension expression loops	158
6	Dimension step functions	159
7	Using <code>dim</code> expressions and variables	160
8	Viewing <code>dim</code> variables	161
9	Constant dimensions	162
10	Scratch dimensions	162
11	Creating and initialising <code>skip</code> variables	162
12	Setting <code>skip</code> variables	163
13	Skip expression conditionals	164
14	Using <code>skip</code> expressions and variables	164
15	Viewing <code>skip</code> variables	164
16	Constant skips	165
17	Scratch skips	165
18	Inserting skips into the output	165
19	Creating and initialising <code>muskip</code> variables	166
20	Setting <code>muskip</code> variables	166
21	Using <code>muskip</code> expressions and variables	167
22	Viewing <code>muskip</code> variables	167

<b>23</b>	<b>Constant muskips</b>	<b>168</b>
<b>24</b>	<b>Scratch muskips</b>	<b>168</b>
<b>25</b>	<b>Primitive conditional</b>	<b>168</b>
<b>XX</b>	<b>The <code>l3keys</code> package: Key–value interfaces</b>	<b>169</b>
<b>1</b>	<b>Creating keys</b>	<b>170</b>
<b>2</b>	<b>Sub-dividing keys</b>	<b>174</b>
<b>3</b>	<b>Choice and multiple choice keys</b>	<b>174</b>
<b>4</b>	<b>Setting keys</b>	<b>177</b>
<b>5</b>	<b>Handling of unknown keys</b>	<b>177</b>
<b>6</b>	<b>Selective key setting</b>	<b>178</b>
<b>7</b>	<b>Utility functions for keys</b>	<b>179</b>
<b>8</b>	<b>Low-level interface for parsing key–val lists</b>	<b>180</b>
<b>XXI</b>	<b>The <code>l3intarray</code> package: fast global integer arrays</b>	<b>182</b>
<b>1</b>	<b><code>l3intarray</code> documentation</b>	<b>182</b>
1.1	Implementation notes . . . . .	183
<b>XXII</b>	<b>The <code>l3fp</code> package: Floating points</b>	<b>184</b>
<b>1</b>	<b>Creating and initialising floating point variables</b>	<b>185</b>
<b>2</b>	<b>Setting floating point variables</b>	<b>186</b>
<b>3</b>	<b>Using floating points</b>	<b>186</b>
<b>4</b>	<b>Floating point conditionals</b>	<b>188</b>
<b>5</b>	<b>Floating point expression loops</b>	<b>189</b>
<b>6</b>	<b>Some useful constants, and scratch variables</b>	<b>191</b>
<b>7</b>	<b>Floating point exceptions</b>	<b>192</b>
<b>8</b>	<b>Viewing floating points</b>	<b>193</b>

<b>9</b>	<b>Floating point expressions</b>	<b>193</b>
9.1	Input of floating point numbers . . . . .	193
9.2	Precedence of operators . . . . .	195
9.3	Operations . . . . .	195
<b>10</b>	<b>Disclaimer and roadmap</b>	<b>202</b>
 <b>XXIII The <code>I3fparray</code> package: fast global floating point arrays</b>		<b>205</b>
1	<b><code>I3fparray</code> documentation</b>	<b>205</b>
 <b>XXIV The <code>I3sort</code> package: Sorting functions</b>		<b>206</b>
1	<b>Controlling sorting</b>	<b>206</b>
 <b>XXV The <code>I3tl-analysis</code> package: Analysing token lists</b>		<b>207</b>
1	<b><code>I3tl-analysis</code> documentation</b>	<b>207</b>
 <b>XXVI The <code>I3regex</code> package: Regular expressions in <code>TEX</code></b>		<b>208</b>
1	<b>Syntax of regular expressions</b>	<b>208</b>
2	<b>Syntax of the replacement text</b>	<b>213</b>
3	<b>Pre-compiling regular expressions</b>	<b>215</b>
4	<b>Matching</b>	<b>215</b>
5	<b>Submatch extraction</b>	<b>216</b>
6	<b>Replacement</b>	<b>217</b>
7	<b>Constants and variables</b>	<b>217</b>
8	<b>Bugs, misfeatures, future work, and other possibilities</b>	<b>218</b>
 <b>XXVII The <code>I3box</code> package: Boxes</b>		<b>221</b>
1	<b>Creating and initialising boxes</b>	<b>221</b>
2	<b>Using boxes</b>	<b>221</b>
3	<b>Measuring and setting box dimensions</b>	<b>222</b>
4	<b>Box conditionals</b>	<b>223</b>
5	<b>The last box inserted</b>	<b>223</b>

6	Constant boxes	223
7	Scratch boxes	223
8	Viewing box contents	224
9	Boxes and color	224
10	Horizontal mode boxes	224
11	Vertical mode boxes	225
12	Using boxes efficiently	227
13	Affine transformations	228
14	Primitive box conditionals	231
XXVIII	The <b>l3coffins</b> package: Coffin code layer	232
1	Creating and initialising coffins	232
2	Setting coffin content and poles	232
3	Joining and using coffins	234
4	Measuring coffins	235
5	Coffin diagnostics	235
6	Constants and variables	236
XXIX	The <b>l3color-base</b> package: Color support	237
1	Color in boxes	237
XXX	The <b>l3luatex</b> package: LuaTeX-specific functions	238
1	Breaking out to Lua	238
2	Lua interfaces	239
XXXI	The <b>l3unicode</b> package: Unicode support functions	240
XXXII	The <b>l3candidates</b> package: Experimental additions to <b>l3kernel</b>	241
1	Important notice	241

<b>2</b>	<b>Additions to <code>I3basics</code></b>	<b>242</b>
<b>3</b>	<b>Additions to <code>I3box</code></b>	<b>243</b>
3.1	Viewing part of a box . . . . .	243
<b>4</b>	<b>Additions to <code>I3coffins</code></b>	<b>243</b>
<b>5</b>	<b>Additions to <code>I3expan</code></b>	<b>244</b>
<b>6</b>	<b>Additions to <code>I3fp</code></b>	<b>244</b>
<b>7</b>	<b>Additions to <code>I3fparay</code></b>	<b>244</b>
<b>8</b>	<b>Additions to <code>I3file</code></b>	<b>245</b>
<b>9</b>	<b>Additions to <code>I3flag</code></b>	<b>246</b>
<b>10</b>	<b>Additions to <code>I3int</code></b>	<b>246</b>
<b>11</b>	<b>Additions to <code>I3intarray</code></b>	<b>247</b>
11.1	Working with contents of integer arrays . . . . .	247
<b>12</b>	<b>Additions to <code>I3msg</code></b>	<b>247</b>
<b>13</b>	<b>Additions to <code>I3prg</code></b>	<b>248</b>
<b>14</b>	<b>Additions to <code>I3prop</code></b>	<b>249</b>
<b>15</b>	<b>Additions to <code>I3seq</code></b>	<b>250</b>
<b>16</b>	<b>Additions to <code>I3skip</code></b>	<b>251</b>
<b>17</b>	<b>Additions to <code>I3sys</code></b>	<b>252</b>
<b>18</b>	<b>Additions to <code>I3tl</code></b>	<b>254</b>
<b>19</b>	<b>Additions to <code>I3token</code></b>	<b>259</b>
 <b>XXXIII The <code>I3drivers</code> package: Drivers</b>		<b>261</b>
<b>1</b>	<b>Box clipping</b>	<b>261</b>
<b>2</b>	<b>Box rotation and scaling</b>	<b>262</b>
<b>3</b>	<b>Color support</b>	<b>262</b>

<b>4</b>	<b>Drawing</b>	<b>262</b>
4.1	Path construction . . . . .	263
4.2	Stroking and filling . . . . .	264
4.3	Stroke options . . . . .	265
4.4	Color . . . . .	265
4.5	Inserting $\text{\TeX}$ material . . . . .	266
4.6	Coordinate system transformations . . . . .	266
<b>5</b>	<b>PDF Features</b>	<b>266</b>
5.1	PDF Objects . . . . .	266
5.2	PDF structure . . . . .	267
<b>XXXIV Implementation</b>		<b>267</b>
<b>1</b>	<b>I3bootstrap implementation</b>	<b>267</b>
1.1	Format-specific code . . . . .	267
1.2	The <code>\pdfstrcmp</code> primitive in $\text{X}\mathbb{\text{\TeX}}$ . . . . .	268
1.3	Loading support Lua code . . . . .	269
1.4	Engine requirements . . . . .	269
1.5	Extending allocators . . . . .	271
1.6	Character data . . . . .	272
1.7	The $\text{L}\mathbb{\text{\TeX}}\text{3}$ code environment . . . . .	273
<b>2</b>	<b>I3names implementation</b>	<b>275</b>
2.1	Deprecated functions . . . . .	298
<b>3</b>	<b>Internal kernel functions</b>	<b>310</b>
<b>4</b>	<b>I3basics implementation</b>	<b>315</b>
4.1	Renaming some $\text{\TeX}$ primitives (again) . . . . .	315
4.2	Defining some constants . . . . .	317
4.3	Defining functions . . . . .	318
4.4	Selecting tokens . . . . .	318
4.5	Gobbling tokens from input . . . . .	320
4.6	Debugging and patching later definitions . . . . .	320
4.7	Conditional processing and definitions . . . . .	328
4.8	Dissecting a control sequence . . . . .	334
4.9	Exist or free . . . . .	337
4.10	Preliminaries for new functions . . . . .	338
4.11	Defining new functions . . . . .	340
4.12	Copying definitions . . . . .	341
4.13	Undefining functions . . . . .	342
4.14	Generating parameter text from argument count . . . . .	342
4.15	Defining functions from a given number of arguments . . . . .	343
4.16	Using the signature to define functions . . . . .	344
4.17	Checking control sequence equality . . . . .	346
4.18	Diagnostic functions . . . . .	347
4.19	Doing nothing functions . . . . .	348
4.20	Breaking out of mapping functions . . . . .	348

<b>5</b>	<b>l3expan implementation</b>	<b>349</b>
5.1	General expansion . . . . .	349
5.2	Hand-tuned definitions . . . . .	352
5.3	Definitions with the automated technique . . . . .	355
5.4	Last-unbraced versions . . . . .	356
5.5	Preventing expansion . . . . .	359
5.6	Controlled expansion . . . . .	360
5.7	Emulating e-type expansion . . . . .	360
5.8	Defining function variants . . . . .	366
<b>6</b>	<b>l3tl implementation</b>	<b>375</b>
6.1	Functions . . . . .	375
6.2	Constant token lists . . . . .	377
6.3	Adding to token list variables . . . . .	378
6.4	Reassigning token list category codes . . . . .	380
6.5	Modifying token list variables . . . . .	383
6.6	Token list conditionals . . . . .	386
6.7	Mapping to token lists . . . . .	390
6.8	Using token lists . . . . .	392
6.9	Working with the contents of token lists . . . . .	393
6.10	Token by token changes . . . . .	395
6.11	The first token from a token list . . . . .	397
6.12	Using a single item . . . . .	401
6.13	Viewing token lists . . . . .	404
6.14	Scratch token lists . . . . .	405
<b>7</b>	<b>l3str implementation</b>	<b>406</b>
7.1	Creating and setting string variables . . . . .	406
7.2	Modifying string variables . . . . .	407
7.3	String comparisons . . . . .	408
7.4	Mapping to strings . . . . .	411
7.5	Accessing specific characters in a string . . . . .	413
7.6	Counting characters . . . . .	418
7.7	The first character in a string . . . . .	419
7.8	String manipulation . . . . .	420
7.9	Viewing strings . . . . .	422
7.10	Deprecated functions . . . . .	422
<b>8</b>	<b>l3quark implementation</b>	<b>422</b>
8.1	Quarks . . . . .	422
8.2	Scan marks . . . . .	425

<b>9</b>	<b>I3seq implementation</b>	<b>426</b>
9.1	Allocation and initialisation . . . . .	427
9.2	Appending data to either end . . . . .	430
9.3	Modifying sequences . . . . .	431
9.4	Sequence conditionals . . . . .	433
9.5	Recovering data from sequences . . . . .	434
9.6	Mapping to sequences . . . . .	438
9.7	Using sequences . . . . .	440
9.8	Sequence stacks . . . . .	441
9.9	Viewing sequences . . . . .	442
9.10	Scratch sequences . . . . .	442
<b>10</b>	<b>I3int implementation</b>	<b>442</b>
10.1	Integer expressions . . . . .	443
10.2	Creating and initialising integers . . . . .	445
10.3	Setting and incrementing integers . . . . .	447
10.4	Using integers . . . . .	449
10.5	Integer expression conditionals . . . . .	449
10.6	Integer expression loops . . . . .	453
10.7	Integer step functions . . . . .	454
10.8	Formatting integers . . . . .	456
10.9	Converting from other formats to integers . . . . .	461
10.10	Viewing integer . . . . .	464
10.11	Random integers . . . . .	465
10.12	Constant integers . . . . .	465
10.13	Scratch integers . . . . .	465
10.14	Deprecated . . . . .	466
<b>11</b>	<b>I3flag implementation</b>	<b>467</b>
11.1	Non-expandable flag commands . . . . .	467
11.2	Expandable flag commands . . . . .	468
<b>12</b>	<b>I3prg implementation</b>	<b>469</b>
12.1	Primitive conditionals . . . . .	469
12.2	Defining a set of conditional functions . . . . .	469
12.3	The boolean data type . . . . .	470
12.4	Boolean expressions . . . . .	472
12.5	Logical loops . . . . .	476
12.6	Producing multiple copies . . . . .	477
12.7	Detecting T <sub>E</sub> X's mode . . . . .	479
12.8	Internal programming functions . . . . .	480
12.9	Deprecated functions . . . . .	480
<b>13</b>	<b>I3sys implementation</b>	<b>481</b>
13.1	The name of the job . . . . .	481
13.2	Time and date . . . . .	481
13.3	Detecting the engine . . . . .	482
13.4	Detecting the output . . . . .	482
13.5	Randomness . . . . .	483

<b>14</b>	<b>I3clist implementation</b>	<b>483</b>
14.1	Removing spaces around items . . . . .	484
14.2	Allocation and initialisation . . . . .	485
14.3	Adding data to comma lists . . . . .	487
14.4	Comma lists as stacks . . . . .	488
14.5	Modifying comma lists . . . . .	490
14.6	Comma list conditionals . . . . .	493
14.7	Mapping to comma lists . . . . .	494
14.8	Using comma lists . . . . .	497
14.9	Using a single item . . . . .	498
14.10	Viewing comma lists . . . . .	500
14.11	Scratch comma lists . . . . .	500
<b>15</b>	<b>I3token implementation</b>	<b>501</b>
15.1	Manipulating and interrogating character tokens . . . . .	501
15.2	Creating character tokens . . . . .	503
15.3	Generic tokens . . . . .	507
15.4	Token conditionals . . . . .	508
15.5	Peeking ahead at the next token . . . . .	515
15.6	Decomposing a macro definition . . . . .	520
<b>16</b>	<b>I3prop implementation</b>	<b>521</b>
16.1	Allocation and initialisation . . . . .	522
16.2	Accessing data in property lists . . . . .	524
16.3	Property list conditionals . . . . .	528
16.4	Recovering values from property lists with branching . . . . .	529
16.5	Mapping to property lists . . . . .	530
16.6	Viewing property lists . . . . .	531
<b>17</b>	<b>I3msg implementation</b>	<b>531</b>
17.1	Creating messages . . . . .	532
17.2	Messages: support functions and text . . . . .	533
17.3	Showing messages: low level mechanism . . . . .	534
17.4	Displaying messages . . . . .	536
17.5	Kernel-specific functions . . . . .	545
17.6	Expandable errors . . . . .	553
<b>18</b>	<b>I3file implementation</b>	<b>556</b>
18.1	Input operations . . . . .	556
18.1.1	Variables and constants . . . . .	556
18.1.2	Stream management . . . . .	557
18.1.3	Reading input . . . . .	560
18.2	Output operations . . . . .	562
18.2.1	Variables and constants . . . . .	562
18.3	Stream management . . . . .	563
18.3.1	Deferred writing . . . . .	565
18.3.2	Immediate writing . . . . .	565
18.3.3	Special characters for writing . . . . .	566
18.3.4	Hard-wrapping lines to a character count . . . . .	566
18.4	File operations . . . . .	576

18.5	Messages . . . . .	583
<b>19</b>	<b>I3skip implementation</b>	<b>584</b>
19.1	Length primitives renamed . . . . .	584
19.2	Creating and initialising <code>dim</code> variables . . . . .	584
19.3	Setting <code>dim</code> variables . . . . .	586
19.4	Utilities for dimension calculations . . . . .	587
19.5	Dimension expression conditionals . . . . .	588
19.6	Dimension expression loops . . . . .	590
19.7	Dimension step functions . . . . .	591
19.8	Using <code>dim</code> expressions and variables . . . . .	593
19.9	Viewing <code>dim</code> variables . . . . .	594
19.10	Constant dimensions . . . . .	595
19.11	Scratch dimensions . . . . .	595
19.12	Creating and initialising <code>skip</code> variables . . . . .	595
19.13	Setting <code>skip</code> variables . . . . .	596
19.14	Skip expression conditionals . . . . .	597
19.15	Using <code>skip</code> expressions and variables . . . . .	598
19.16	Inserting skips into the output . . . . .	598
19.17	Viewing <code>skip</code> variables . . . . .	599
19.18	Constant skips . . . . .	599
19.19	Scratch skips . . . . .	599
19.20	Creating and initialising <code>muskip</code> variables . . . . .	600
19.21	Setting <code>muskip</code> variables . . . . .	601
19.22	Using <code>muskip</code> expressions and variables . . . . .	602
19.23	Viewing <code>muskip</code> variables . . . . .	602
19.24	Constant muskips . . . . .	603
19.25	Scratch muskips . . . . .	603
<b>20</b>	<b>I3keys Implementation</b>	<b>603</b>
20.1	Low-level interface . . . . .	603
20.2	Constants and variables . . . . .	607
20.3	The key defining mechanism . . . . .	609
20.4	Turning properties into actions . . . . .	611
20.5	Creating key properties . . . . .	616
20.6	Setting keys . . . . .	620
20.7	Utilities . . . . .	626
20.8	Messages . . . . .	627
<b>21</b>	<b>I3intarray implementation</b>	<b>628</b>
21.1	Allocating arrays . . . . .	628
21.2	Array items . . . . .	629
21.3	Working with contents of integer arrays . . . . .	632
21.4	Random arrays . . . . .	633
<b>22</b>	<b>I3fp implementation</b>	<b>635</b>

<b>23</b>	<b>l3fp-aux implementation</b>	<b>635</b>
23.1	Access to primitives . . . . .	635
23.2	Internal representation . . . . .	635
23.3	Using arguments and semicolons . . . . .	636
23.4	Constants, and structure of floating points . . . . .	637
23.5	Overflow, underflow, and exact zero . . . . .	639
23.6	Expanding after a floating point number . . . . .	639
23.7	Other floating point types . . . . .	641
23.8	Packing digits . . . . .	644
23.9	Decimate (dividing by a power of 10) . . . . .	646
23.10	Functions for use within primitive conditional branches . . . . .	648
23.11	Integer floating points . . . . .	650
23.12	Small integer floating points . . . . .	650
23.13	x-like expansion expandably . . . . .	651
23.14	Fast string comparison . . . . .	651
23.15	Name of a function from its l3fp-parse name . . . . .	652
23.16	Messages . . . . .	652
<b>24</b>	<b>l3fp-traps Implementation</b>	<b>652</b>
24.1	Flags . . . . .	653
24.2	Traps . . . . .	653
24.3	Errors . . . . .	656
24.4	Messages . . . . .	657
<b>25</b>	<b>l3fp-round implementation</b>	<b>657</b>
25.1	Rounding tools . . . . .	658
25.2	The round function . . . . .	662
<b>26</b>	<b>l3fp-parse implementation</b>	<b>665</b>
26.1	Work plan . . . . .	665
26.1.1	Storing results . . . . .	667
26.1.2	Precedence and infix operators . . . . .	668
26.1.3	Prefix operators, parentheses, and functions . . . . .	671
26.1.4	Numbers and reading tokens one by one . . . . .	671
26.2	Main auxiliary functions . . . . .	673
26.3	Helpers . . . . .	674
26.4	Parsing one number . . . . .	675
26.4.1	Numbers: trimming leading zeros . . . . .	681
26.4.2	Number: small significand . . . . .	683
26.4.3	Number: large significand . . . . .	685
26.4.4	Number: beyond 16 digits, rounding . . . . .	687
26.4.5	Number: finding the exponent . . . . .	689
26.5	Constants, functions and prefix operators . . . . .	692
26.5.1	Prefix operators . . . . .	692
26.5.2	Constants . . . . .	695
26.5.3	Functions . . . . .	697
26.6	Main functions . . . . .	697
26.7	Infix operators . . . . .	699
26.7.1	Closing parentheses and commas . . . . .	700
26.7.2	Usual infix operators . . . . .	702

26.7.3	Juxtaposition . . . . .	703
26.7.4	Multi-character cases . . . . .	703
26.7.5	Ternary operator . . . . .	704
26.7.6	Comparisons . . . . .	704
26.8	Tools for functions . . . . .	706
26.9	Messages . . . . .	708
<b>27</b>	<b>I3fp-assign implementation</b>	<b>709</b>
27.1	Assigning values . . . . .	709
27.2	Updating values . . . . .	710
27.3	Showing values . . . . .	711
27.4	Some useful constants and scratch variables . . . . .	711
<b>28</b>	<b>I3fp-logic Implementation</b>	<b>712</b>
28.1	Syntax of internal functions . . . . .	712
28.2	Existence test . . . . .	712
28.3	Comparison . . . . .	712
28.4	Floating point expression loops . . . . .	715
28.5	Extrema . . . . .	719
28.6	Boolean operations . . . . .	720
28.7	Ternary operator . . . . .	721
<b>29</b>	<b>I3fp-basics Implementation</b>	<b>722</b>
29.1	Addition and subtraction . . . . .	723
29.1.1	Sign, exponent, and special numbers . . . . .	723
29.1.2	Absolute addition . . . . .	725
29.1.3	Absolute subtraction . . . . .	727
29.2	Multiplication . . . . .	732
29.2.1	Signs, and special numbers . . . . .	732
29.2.2	Absolute multiplication . . . . .	733
29.3	Division . . . . .	735
29.3.1	Signs, and special numbers . . . . .	735
29.3.2	Work plan . . . . .	736
29.3.3	Implementing the significand division . . . . .	739
29.4	Square root . . . . .	744
29.5	About the sign and exponent . . . . .	751
29.6	Operations on tuples . . . . .	752
<b>30</b>	<b>I3fp-extended implementation</b>	<b>753</b>
30.1	Description of fixed point numbers . . . . .	753
30.2	Helpers for numbers with extended precision . . . . .	754
30.3	Multiplying a fixed point number by a short one . . . . .	755
30.4	Dividing a fixed point number by a small integer . . . . .	755
30.5	Adding and subtracting fixed points . . . . .	757
30.6	Multiplying fixed points . . . . .	757
30.7	Combining product and sum of fixed points . . . . .	759
30.8	Extended-precision floating point numbers . . . . .	761
30.9	Dividing extended-precision numbers . . . . .	763
30.10	Inverse square root of extended precision numbers . . . . .	766
30.11	Converting from fixed point to floating point . . . . .	768

<b>31</b>	<b>l3fp-expo implementation</b>	<b>770</b>
31.1	Logarithm . . . . .	771
31.1.1	Work plan . . . . .	771
31.1.2	Some constants . . . . .	771
31.1.3	Sign, exponent, and special numbers . . . . .	772
31.1.4	Absolute ln . . . . .	772
31.2	Exponential . . . . .	779
31.2.1	Sign, exponent, and special numbers . . . . .	779
31.3	Power . . . . .	783
<b>32</b>	<b>l3fp-trig Implementation</b>	<b>790</b>
32.1	Direct trigonometric functions . . . . .	791
32.1.1	Filtering special cases . . . . .	791
32.1.2	Distinguishing small and large arguments . . . . .	794
32.1.3	Small arguments . . . . .	795
32.1.4	Argument reduction in degrees . . . . .	795
32.1.5	Argument reduction in radians . . . . .	796
32.1.6	Computing the power series . . . . .	804
32.2	Inverse trigonometric functions . . . . .	806
32.2.1	Arctangent and arccotangent . . . . .	807
32.2.2	Arcsine and arccosine . . . . .	812
32.2.3	Arccosecant and arcsecant . . . . .	814
<b>33</b>	<b>l3fp-convert implementation</b>	<b>815</b>
33.1	Dealing with tuples . . . . .	815
33.2	Trimming trailing zeros . . . . .	816
33.3	Scientific notation . . . . .	816
33.4	Decimal representation . . . . .	818
33.5	Token list representation . . . . .	820
33.6	Formatting . . . . .	821
33.7	Convert to dimension or integer . . . . .	821
33.8	Convert from a dimension . . . . .	822
33.9	Use and eval . . . . .	823
33.10	Convert an array of floating points to a comma list . . . . .	823
<b>34</b>	<b>l3fp-random Implementation</b>	<b>824</b>
34.1	Engine support . . . . .	824
34.2	Random floating point . . . . .	828
34.3	Random integer . . . . .	828
<b>35</b>	<b>l3fparray implementation</b>	<b>833</b>
35.1	Allocating arrays . . . . .	834
35.2	Array items . . . . .	835

<b>36</b>	<b><code>l3sort</code> implementation</b>	<b>838</b>
36.1	Variables . . . . .	838
36.2	Finding available <code>\toks</code> registers . . . . .	839
36.3	Protected user commands . . . . .	841
36.4	Merge sort . . . . .	843
36.5	Expandable sorting . . . . .	846
36.6	Messages . . . . .	851
<b>37</b>	<b><code>l3tl-analysis</code> implementation</b>	<b>852</b>
37.1	Internal functions . . . . .	852
37.2	Internal format . . . . .	853
37.3	Variables and helper functions . . . . .	853
37.4	Plan of attack . . . . .	855
37.5	Disabling active characters . . . . .	856
37.6	First pass . . . . .	857
37.7	Second pass . . . . .	862
37.8	Mapping through the analysis . . . . .	865
37.9	Showing the results . . . . .	865
37.10	Messages . . . . .	868
37.11	Deprecated functions . . . . .	868
<b>38</b>	<b><code>l3regex</code> implementation</b>	<b>868</b>
38.1	Plan of attack . . . . .	868
38.2	Helpers . . . . .	870
38.2.1	Constants and variables . . . . .	871
38.2.2	Testing characters . . . . .	872
38.2.3	Character property tests . . . . .	876
38.2.4	Simple character escape . . . . .	878
38.3	Compiling . . . . .	883
38.3.1	Variables used when compiling . . . . .	884
38.3.2	Generic helpers used when compiling . . . . .	885
38.3.3	Mode . . . . .	886
38.3.4	Framework . . . . .	889
38.3.5	Quantifiers . . . . .	891
38.3.6	Raw characters . . . . .	894
38.3.7	Character properties . . . . .	896
38.3.8	Anchoring and simple assertions . . . . .	897
38.3.9	Character classes . . . . .	898
38.3.10	Groups and alternations . . . . .	901
38.3.11	Catcodes and csnames . . . . .	904
38.3.12	Raw token lists with <code>\u</code> . . . . .	907
38.3.13	Other . . . . .	909
38.3.14	Showing regexes . . . . .	910
38.4	Building . . . . .	914
38.4.1	Variables used while building . . . . .	914
38.4.2	Framework . . . . .	914
38.4.3	Helpers for building an NFA . . . . .	916
38.4.4	Building classes . . . . .	918
38.4.5	Building groups . . . . .	919
38.4.6	Others . . . . .	924

38.5	Matching . . . . .	925
38.5.1	Variables used when matching . . . . .	926
38.5.2	Matching: framework . . . . .	928
38.5.3	Using states of the NFA . . . . .	932
38.5.4	Actions when matching . . . . .	933
38.6	Replacement . . . . .	935
38.6.1	Variables and helpers used in replacement . . . . .	935
38.6.2	Query and brace balance . . . . .	936
38.6.3	Framework . . . . .	938
38.6.4	Submatches . . . . .	940
38.6.5	Csnames in replacement . . . . .	941
38.6.6	Characters in replacement . . . . .	942
38.6.7	An error . . . . .	946
38.7	User functions . . . . .	946
38.7.1	Variables and helpers for user functions . . . . .	948
38.7.2	Matching . . . . .	949
38.7.3	Extracting submatches . . . . .	950
38.7.4	Replacement . . . . .	953
38.7.5	Storing and showing compiled patterns . . . . .	955
38.8	Messages . . . . .	955
38.9	Code for tracing . . . . .	961
<b>39</b>	<b>I3box implementation</b>	<b>962</b>
39.1	Support code . . . . .	962
39.2	Creating and initialising boxes . . . . .	962
39.3	Measuring and setting box dimensions . . . . .	963
39.4	Using boxes . . . . .	964
39.5	Box conditionals . . . . .	964
39.6	The last box inserted . . . . .	965
39.7	Constant boxes . . . . .	965
39.8	Scratch boxes . . . . .	965
39.9	Viewing box contents . . . . .	966
39.10	Horizontal mode boxes . . . . .	967
39.11	Vertical mode boxes . . . . .	969
39.12	Affine transformations . . . . .	972
39.13	Deprecated functions . . . . .	981
<b>40</b>	<b>I3coffins Implementation</b>	<b>982</b>
40.1	Coffins: data structures and general variables . . . . .	982
40.2	Basic coffin functions . . . . .	983
40.3	Measuring coffins . . . . .	988
40.4	Coffins: handle and pole management . . . . .	989
40.5	Coffins: calculation of pole intersections . . . . .	992
40.6	Aligning and typesetting of coffins . . . . .	995
40.7	Coffin diagnostics . . . . .	1000
40.8	Messages . . . . .	1006
<b>41</b>	<b>I3color-base Implementation</b>	<b>1006</b>

<b>42</b>	<b>I3luatex implementation</b>	<b>1008</b>
42.1	Breaking out to Lua . . . . .	1008
42.2	Messages . . . . .	1009
42.3	Deprecated functions . . . . .	1009
42.4	Lua functions for internal use . . . . .	1009
42.5	Generic Lua and font support . . . . .	1012
<b>43</b>	<b>I3unicode implementation</b>	<b>1013</b>
<b>44</b>	<b>I3candidates Implementation</b>	<b>1016</b>
44.1	Additions to I3basics . . . . .	1016
44.2	Additions to I3box . . . . .	1016
44.2.1	Viewing part of a box . . . . .	1016
44.3	Additions to I3coffins . . . . .	1019
44.3.1	Rotating coffins . . . . .	1019
44.3.2	Resizing coffins . . . . .	1024
44.4	Additions to I3fp-convert . . . . .	1027
44.5	Additions to I3file . . . . .	1027
44.6	Additions to I3flag . . . . .	1029
44.7	Additions to I3msg . . . . .	1029
44.8	Additions to I3msg . . . . .	1030
44.9	Additions to I3prg . . . . .	1031
44.10	Additions to I3prop . . . . .	1032
44.11	Additions to I3seq . . . . .	1033
44.12	Additions to I3skip . . . . .	1037
44.13	Additions to I3sys . . . . .	1038
44.14	Additions to I3tl . . . . .	1042
44.14.1	Deprecated functions . . . . .	1044
44.14.2	Unicode case changing . . . . .	1044
44.14.3	Building a token list . . . . .	1067
44.14.4	Other additions to I3tl . . . . .	1070
44.15	Additions to I3token . . . . .	1071
<b>45</b>	<b>I3drivers Implementation</b>	<b>1074</b>
45.1	Color support . . . . .	1075
45.1.1	dvips-style . . . . .	1075
45.1.2	pdfmode . . . . .	1076
45.2	dvips driver . . . . .	1078
45.2.1	Basics . . . . .	1078
45.2.2	Box operations . . . . .	1079
45.2.3	Images . . . . .	1081
45.2.4	Drawing . . . . .	1081
45.2.5	PDF Features . . . . .	1087
45.3	pdfmode driver . . . . .	1088
45.3.1	Basics . . . . .	1089
45.3.2	Box operations . . . . .	1089
45.3.3	Images . . . . .	1091
45.3.4	PDF Objects . . . . .	1093
45.3.5	PDF Structure . . . . .	1094
45.4	dvipdfmx driver . . . . .	1095

45.4.1	Basics . . . . .	1095
45.4.2	Box operations . . . . .	1095
45.4.3	Images . . . . .	1096
45.4.4	PDF Objects . . . . .	1098
45.4.5	PDF Structure . . . . .	1099
45.5	<code>xdvipdfmx</code> driver . . . . .	1100
45.5.1	Images . . . . .	1100
45.6	Drawing commands: <code>pdfmode</code> and <code>(x)dvipdfmx</code> . . . . .	1101
45.6.1	Drawing . . . . .	1101
45.7	<code>dvisvgm</code> driver . . . . .	1107
45.7.1	Basics . . . . .	1107
45.7.2	Driver-specific auxiliaries . . . . .	1108
45.7.3	Box operations . . . . .	1108
45.7.4	Images . . . . .	1110
45.7.5	PDF Features . . . . .	1111
45.7.6	Drawing . . . . .	1111
<b>46</b>	<b><code>l3deprecation</code> implementation</b>	<b>1117</b>
<b>Index</b>		<b>1121</b>

# Part I

## 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 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:

- D The D specifier means *do not use*. All of the T<sub>E</sub>X primitives are initially \let to a D name, and some are then given a second name. Only the kernel team should use anything with a D specifier!
- 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.
- 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}.
  - o 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.
  - x The x specifier stands for *exhaustive expansion*: every token in the argument is fully expanded until only unexpandable ones remain. The T<sub>E</sub>X \edef primitive carries out this type of expansion. Functions which feature an x-type argument are in general *not* expandable, unless specifically noted.

- f** The **f** specifier stands for *full expansion*, and in contrast to **x** stops at the first non-expandable item (reading the argument from left to right) without trying to expand it. 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.

- p** The letter **p** indicates *TeX parameters*. Normally this will be used for delimited functions as `expl3` provides better methods for creating simple sequential arguments.
- w** 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 arbitrary string).

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`.

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>1</sup> name and then a descriptive part. Variables end with a short identifier to show the variable type:

**bool** Either true or false.

**box** Box register.

**clist** Comma separated list.

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

**dim** “Rigid” lengths.

**fp** floating-point values;

---

<sup>1</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.

**int** Integer-valued count register.

**prop** Property list.

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

**skip** “Rubber” lengths.

**stream** An input or output stream (for reading from or writing to, respectively).

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

### 1.1 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 in truth one and the same. 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.

## 2 Documentation conventions

This document is typeset with the experimental `l3doc` 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:

---

<code>\ExplSyntaxOn</code> <code>\ExplSyntaxOff</code>	<code>\ExplSyntaxOn ... \ExplSyntaxOff</code>
---	---

---

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:

---

`\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 argument (in plain `TeX` terms, inside an `\edef`), 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 `LATEX 2&` or plain `TeX`. In these cases, the text will include an extra “**TeXhackers note**” section:

---

`\token_to_str:N` ★ `\token_to_str:N <token>`

The normal description text.

**TEXhackers note:** Detail for the experienced TeX or L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</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.

### 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 is evaluated to give a logically `TRUE` or `FALSE` result. Depending on this result, either the `<true code>` or the `<false code>` 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.

### 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>ε</sub> if `\outer` tokens are used in the arguments.

## Part II

# The **l3bootstrap** package

## Bootstrap code

### 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. These modules will also form the basis of the L<sup>A</sup>T<sub>E</sub>X3 format, but work in this area is incomplete and not included in this documentation at present.

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
\ExplSyntaxOff
```

---

Updated: 2011-08-13

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

The `\ExplSyntaxOn` function switches to a category code régime in which spaces 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 régime.

---

```
\ProvidesExplPackage
\ProvidesExplClass
\ProvidesExplFile
```

---

Updated: 2017-03-19

```
\RequirePackage{expl3}
\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>`. If the `<version>` is given then it will be prefixed with `v` in the package identifier line.

---

```
\GetIdInfo
```

---

Updated: 2012-06-04

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

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>ε</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}
```

# Part III

## The **I3names** package

### Namespace for primitives

#### 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;
- switches to the category code régime for programming;
- provides support settings for building the code as a T<sub>E</sub>X format.

This module is entirely dedicated to primitives, 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 IV

## 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.

## 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.

## 2 Grouping material

---

`\group_begin:`  
`\group_end:`

`\group_begin:`  
`\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  $\langle token \rangle$  to the list of  $\langle tokens \rangle$  to be inserted when the current group level ends. The list of  $\langle tokens \rangle$  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  $\langle token \rangle$  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.

## 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 by 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.

### 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 expansion.

Finally, the functions in Subsections 3.2 and 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 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 2.

### 3.2 Defining new functions using parameter text

---

```
\cs_new:Npn \cs_new:cpx
```

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

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:cpx
```

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

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:cpx
```

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

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 argument. The definition is global and an error results if the `<function>` is already defined.

---

```
\cs_new_protected_nopar:Npn \cs_new_protected_nopar:cpx
```

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

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 argument. The definition is global and an error results if the `<function>` is already defined.

---

```
\cs_set:Npn \cs_set:cpx
```

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

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:cpx
```

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

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:cpx
```

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

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 argument.

---

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

---

```
\cs_gset:Npn \cs_gset:cpx
\cs_gset:Npx \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: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:cpx
\cs_gset_protected:Npx \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 argument.

---

```
\cs_gset_protected_nopar:Npn \cs_gset_protected_nopar:Npn <function> <parameters> {<code>}
\cs_gset_protected_nopar:cpx
\cs_gset_protected_nopar:Npx
\cs_gset_protected_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. The *<function>* will not expand within an x-type argument.

### 3.3 Defining new functions using the signature

---

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

---

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.

---

```
\cs_new_nopar:Nn
```

```
\cs_new_nopar:(cn|Nx|cx)
```

---

```
\cs_new_nopar:Nn <function> {{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.

---

```
\cs_new_protected:Nn
```

```
\cs_new_protected:(cn|Nx|cx)
```

---

```
\cs_new_protected:Nn <function> {{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 argument. The definition is global and an error results if the *<function>* is already defined.

---

```
\cs_new_protected_nopar:Nn
```

```
\cs_new_protected_nopar:(cn|Nx|cx)
```

---

```
\cs_new_protected_nopar:Nn <function> {{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 argument. The definition is global and an error results if the *<function>* is already defined.

---

```
\cs_set:Nn
```

```
\cs_set:(cn|Nx|cx)
```

---

```
\cs_set:Nn <function> {{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 T<sub>E</sub>X group level.

---

```
\cs_set_nopar:Nn
```

```
\cs_set_nopar:(cn|Nx|cx)
```

---

```
\cs_set_nopar:Nn <function> {{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. 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 T<sub>E</sub>X group level.

---

```
\cs_set_protected:Nn
```

```
\cs_set_protected:(cn|Nx|cx)
```

---

```
\cs_set_protected:Nn <function> {{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 *<function>* will not expand within an x-type argument. The assignment of a meaning to the *<function>* 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; {&lt;code&gt;}</code>
<code>\cs_set_protected_nopar:(cn Nx cx)</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 argument. The assignment of a meaning to the <i>&lt;function&gt;</i> is restricted to the current TeX group level.
<code>\cs_gset:Nn</code>	<code>\cs_gset:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_gset:(cn Nx cx)</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; {&lt;code&gt;}</code>
<code>\cs_gset_nopar:(cn Nx cx)</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; {&lt;code&gt;}</code>
<code>\cs_gset_protected:(cn Nx cx)</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 argument. The assignment of a meaning to the <i>&lt;function&gt;</i> is global.
<code>\cs_gset_protected_nopar:Nn</code>	<code>\cs_gset_protected_nopar:Nn &lt;function&gt; {&lt;code&gt;}</code>
<code>\cs_gset_protected_nopar:(cn Nx cx)</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 argument. The assignment of a meaning to the <i>&lt;function&gt;</i> is global.
<code>\cs_generate_from_arg_count&gt;NNnn</code>	<code>\cs_generate_from_arg_count&gt;NNnn &lt;function&gt; &lt;creator&gt; {&lt;number&gt;} {&lt;code&gt;}</code>
<code>\cs_generate_from_arg_count:(cNnn Ncnn)</code>	Uses the <i>&lt;creator&gt;</i> function (which should have signature Npn, for example \cs_new:Npn) to define a <i>&lt;function&gt;</i> which takes <i>&lt;number&gt;</i> arguments and has <i>&lt;code&gt;</i> as replacement text. The <i>&lt;number&gt;</i> of arguments is an integer expression, evaluated as detailed for \int_eval:n.

Updated: 2012-01-14

## 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:(Nc|cN|cc)
```

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:(Nc|cN|cc)
```

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:(Nc|cN|cc)
```

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.

## 3.5 Deleting control sequences

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

---

```
\cs_undefine:N
\cs_undefine:c
```

Updated: 2011-09-15

$\langle control\ sequence \rangle$

Sets  $\langle control\ sequence \rangle$  to be globally undefined.

## 3.6 Showing control sequences

---

```
\cs_meaning:N *
\cs_meaning:c *
```

Updated: 2011-12-22

$\langle control\ sequence \rangle$

This function expands to the *meaning* of the  $\langle control\ sequence \rangle$  control sequence. For a macro, this includes the  $\langle replacement\ text \rangle$ .

**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.

---

\cs\_show:N  
\cs\_show:c  
Updated: 2017-02-14

\cs\_show:N *(control sequence)*

Displays the definition of the *(control sequence)* on the terminal.

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

---

\cs\_log:N  
\cs\_log:c  
New: 2014-08-22  
Updated: 2017-02-14

\cs\_log:N *(control sequence)*

Writes the definition of the *(control sequence)* in the log file. See also \cs\_show:N which displays the result in the terminal.

### 3.7 Converting to and from control sequences

---

\use:c *\*\use:c {<control sequence name>}*

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).

**TeXhackers note:** Protected macros that appear in a c-type argument are expanded despite being protected; \exp\_not:n also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

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 *\*\cs\_if\_exist\_use:cTF*
\cs\_if\_exist\_use:cTF *\**

New: 2012-11-10

Tests whether the *(control sequence)* is currently defined according to the conditional \cs\_if\_exist:NTF (whether as a function or another control sequence type), and if it is 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:` ★

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 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 Analysing control sequence names

---

`\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).

## 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 {{group <sub>1</sub> }}
\use:nn	*	\use:nn {{group <sub>1</sub> }} {{group <sub>2</sub> }}
\use:nnn	*	\use:nnn {{group <sub>1</sub> }} {{group <sub>2</sub> }} {{group <sub>3</sub> }}
\use:nnnn	*	\use:nnnn {{group <sub>1</sub> }} {{group <sub>2</sub> }} {{group <sub>3</sub> }} {{group <sub>4</sub> }}

---

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.

---

\use_i:nn	*	\use_i:nn {{arg <sub>1</sub> }} {{arg <sub>2</sub> }}
\use_ii:nn	*	\use_ii:nn {{arg <sub>2</sub> }}

---

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.

---

\use_i:nnn	*	\use_i:nnn {{arg <sub>1</sub> }} {{arg <sub>2</sub> }} {{arg <sub>3</sub> }}
\use_ii:nnn	*	\use_ii:nnn {{arg <sub>2</sub> }} {{arg <sub>3</sub> }}
\use_iii:nnn	*	\use_iii:nnn {{arg <sub>3</sub> }}

---

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 {{arg <sub>1</sub> }} {{arg <sub>2</sub> }} {{arg <sub>3</sub> }} {{arg <sub>4</sub> }}
\use_ii:nnnn	*	\use_ii:nnnn {{arg <sub>2</sub> }} {{arg <sub>3</sub> }} {{arg <sub>4</sub> }}
\use_iii:nnnn	*	\use_iii:nnnn {{arg <sub>3</sub> }} {{arg <sub>4</sub> }}
\use_iv:nnnn	*	\use_iv:nnnn {{arg <sub>4</sub> }}

---

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\_ii:nnnn, \use\_iii: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_none:n` ★ `\use_none:n {<group1>}`

`\use_none:nn` ★ `\use_none:nn {<group1>} {<group2>}`

`\use_none:nnn` ★ `\use_none:nnn {<group1>} {<group2>} {<group3>}`

`\use_none:nnnn` ★ `\use_none:nnnn {<group1>} {<group2>} {<group3>} {<group4>}`

`\use_none:nnnnn` ★ `\use_none:nnnnn {<group1>} {<group2>} {<group3>} {<group4>} {<group5>}`

`\use_none:nnnnnn` ★ `\use_none:nnnnnn {<group1>} {<group2>} {<group3>} {<group4>} {<group5>} {<group6>}`

`\use_none:nnnnnnn` ★ `\use_none:nnnnnnn {<group1>} {<group2>} {<group3>} {<group4>} {<group5>} {<group6>} {<group7>}`

`\use_none:nnnnnnnn` ★ `\use_none:nnnnnnnn {<group1>} {<group2>} {<group3>} {<group4>} {<group5>} {<group6>} {<group7>} {<group8>}`

`\use_none:nnnnnnnnn` ★ `\use_none:nnnnnnnnn {<group1>} {<group2>} {<group3>} {<group4>} {<group5>} {<group6>} {<group7>} {<group8>} {<group9>}`

---

`\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.

## 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>* form the input stream delimited by the marker given in the function name, leaving nothing in the input stream.

---

<code>\use_i_delimit_by_q_nil:nw</code> <code>\use_i_delimit_by_q_stop:nw</code> <code>\use_i_delimit_by_q_recursion_stop:nw</code>	<span style="color: red;">*</span> <code>\use_i_delimit_by_q_nil:nw {&lt;inserted tokens&gt;} &lt;balanced text&gt;</code> <span style="color: red;">*</span> <code>\q_nil</code> <span style="color: red;">*</span> <code>\use_i_delimit_by_q_stop:nw {&lt;inserted tokens&gt;} &lt;balanced text&gt; \q_stop</code> <span style="color: red;">*</span> <code>\use_i_delimit_by_q_recursion_stop:nw {&lt;inserted tokens&gt;} &lt;balanced text&gt; \q_recursion_stop</code>
---	--

---

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

## 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} {<true code>} {<false code>}
```

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
} {<true code>} {<false code>}
```

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>ε</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`  
`\c_false_bool`

---

Constants that represent `true` and `false`, respectively. Used to implement predicates.

## 6.1 Tests on control sequences

---

`\cs_if_eq_p:NN` ★  
`\cs_if_eq:NNTF` ★

---

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:c` ★  
`\cs_if_exist:NTF` ★  
`\cs_if_exist:cTF` ★

---

`\cs_if_exist_p:N` *(control sequence)*  
`\cs_if_exist:NTF` *(control sequence)* {*(true code)*} {*(false code)*}

Tests whether the *(control sequence)* is currently defined (whether as a function or another control sequence type). Any definition of *(control sequence)* other than `\relax` evaluates as `true`.

---

`\cs_if_free_p:N` ★  
`\cs_if_free_p:c` ★  
`\cs_if_free:NTF` ★  
`\cs_if_free:cTF` ★

---

`\cs_if_free_p:N` *(control sequence)*  
`\cs_if_free:NTF` *(control sequence)* {*(true code)*} {*(false code)*}

Tests whether the *(control sequence)* is currently free to be defined. This test is `false` if the *(control sequence)* currently exists (as defined by `\cs_if_exist:N`).

## 6.2 Primitive conditionals

The ε-T<sub>E</sub>X 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_false:` ★  
`\else:` ★  
`\fi:` ★  
`\reverse_if:N` ★

---

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

**TeXhackers note:** These are equivalent to their corresponding TeX primitive conditionals; `\reverse_if:N` is ε-T<sub>E</sub>X's `\unless`.

---

**\if\_meaning:w** ★ `\if_meaning:w <arg1> <arg2> <true code> \else: <false code> \fi:`  
\if\_meaning:w executes *true code* when *arg<sub>1</sub>* and *arg<sub>2</sub>* are the same, otherwise it executes *false code*. *arg<sub>1</sub>* and *arg<sub>2</sub>* 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*. Similar for the other functions.  
**\if\_mode\_math:** ★  
**\if\_mode\_inner:** ★

## Part V

# The **I3expan** 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 LATEX3 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.

### 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:N { \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.

## 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 *parent control sequence* {*variant argument specifiers*}

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 *parent 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.

### 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.

## 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>`.

**TeXhackers note:** Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

---

**\exp\_args:No** ★ `\exp_args:No <function> {\<tokens>} ...`

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

---

**\exp\_args:NV** ★ `\exp_args:NV <function> <variable>`

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

---

**\exp\_args:Nv** ★ `\exp_args:Nv <function> {\<tokens>}`

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. This control sequence should be the name of a *<variable>*. The content of the *<variable>* are recovered and placed inside braces into the input stream *after* reinsertion of the *<function>*. Thus the *<function>* may take more than one argument: all others are left unchanged.

**TeXhackers note:** Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

---

**\exp\_args:Ne** ★ `\exp_args:Ne <function> {\<tokens>}`

New: 2018-05-15

This function absorbs two arguments (the *<function>* name and the *<tokens>*) and exhaustively expands the *<tokens>*. The result is inserted in braces into the input stream *after* reinsertion of the *<function>*. Thus the *<function>* 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** ★ `\exp_args:Nf <function> {\<tokens>}`

This function absorbs two arguments (the *<function>* name and the *<tokens>*). The *<tokens>* 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 *<function>*. Thus the *<function>* may take more than one argument: all others are left unchanged.

---

```
\exp_args:Nx
```

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

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

## 5 Manipulating two arguments

---

```
\exp_args:NNc *
\exp_args:NNo *
\exp_args:NNV *
\exp_args:NNv *
\exp_args:NNe *
\exp_args:NNf *
\exp_args:Ncc *
\exp_args:Nco *
\exp_args:Ncv *
\exp_args:Ncf *
\exp_args:NVV *
```

Updated: 2018-05-15

---

```
\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:Nff *
```

Updated: 2018-05-15

---

```
\exp_args:NNx
\exp_args:Ncx
\exp_args:Nnx
\exp_args:Nox
\exp_args:Nxo
\exp_args:Nxx
```

```
\exp_args:NNx <token1> <token2> {{tokens}}
```

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 need slower processing.

## 6 Manipulating three arguments

<hr/> <pre>\exp_args:NNNo ★\exp_args:NNNV ★\exp_args:Nccc ★\exp_args:NcNc ★\exp_args:NcNo ★\exp_args:Ncco ★</pre> <hr/>	<pre>\exp_args:NNNo &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; &lt;token<sub>3</sub>&gt; {&lt;tokens&gt;}</pre> <p>These optimized 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, <i>etc.</i></p>
<hr/> <pre>\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:Nnff ★\exp_args:Nooo ★\exp_args:Noof ★\exp_args:Nffo ★</pre> <hr/>	<pre>\exp_args:NNoo &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {&lt;token<sub>3</sub>&gt;} {&lt;tokens&gt;}</pre> <p>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, <i>etc.</i> These functions need slower processing.</p>
<hr/> <pre>\exp_args:NNNx ★\exp_args:NNnx ★\exp_args:NNox ★\exp_args:Nccx ★\exp_args:Ncnx ★\exp_args:Nnnx ★\exp_args:Nnox ★\exp_args:Noox ★</pre> <hr/>	<pre>\exp_args:NNnx &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt; {&lt;tokens<sub>1</sub>&gt;} {&lt;tokens<sub>2</sub>&gt;}</pre> <p>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, <i>etc.</i></p>

New: 2015-08-12

---

## 7 Unbraced expansion

\exp_last_unbraced:N	★ \exp_last_unbraced:Nno ⟨token⟩ {⟨tokens <sub>1</sub> ⟩} {⟨tokens <sub>2</sub> ⟩}
\exp_last_unbraced:(NV Nv Nf)	★
\exp_last_unbraced:Ne	★
\exp_last_unbraced:NNo	★
\exp_last_unbraced:(NNV NNf Nco NcV)	★
\exp_last_unbraced:Nno	★
\exp_last_unbraced:(Noo Nfo)	★
\exp_last_unbraced:NNNo	★
\exp_last_unbraced:(NNNV NNNf)	★
\exp_last_unbraced:NnNo	★
\exp_last_unbraced:NNNNo	★
\exp_last_unbraced:NNNNf	★

Updated: 2018-05-15

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.

\exp_last_unbraced:Nx	\exp_last_unbraced:Nx ⟨function⟩ {⟨tokens⟩}
-----------------------	---

This function fully expands the ⟨tokens⟩ and leaves the result in the input stream after reinsertion of the ⟨function⟩. This function is not expandable.

\exp_last_two_unbraced:Noo ★	\exp_last_two_unbraced:Noo ⟨token⟩ {⟨tokens <sub>1</sub> ⟩} {⟨tokens <sub>2</sub> ⟩}
------------------------------	--

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 ⟨token <sub>1</sub> ⟩ ⟨token <sub>2</sub> ⟩
-----------------	---

Carries out a single expansion of ⟨token<sub>2</sub>⟩ (which may consume arguments) prior to the expansion of ⟨token<sub>1</sub>⟩. If ⟨token<sub>2</sub>⟩ has no expansion (for example, if it is a character) then it is left unchanged. It is important to notice that ⟨token<sub>1</sub>⟩ may be *any* single token, including group-opening and -closing tokens ({ or } assuming normal T<sub>E</sub>X 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 T<sub>E</sub>X primitive \expandafter renamed.

## 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  $\star$  `\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  $\star$  `\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`.

**TeXhackers note:** Protected macros that appear in a `c`-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

---

\exp\_not:n  $\star$  `\exp_not:n {<tokens>}`

Prevents expansion of the  $\langle tokens \rangle$  in an `e` or `x`-type argument. In all other cases the  $\langle tokens \rangle$  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 ε-Tex `\unexpanded` primitive. In an `x`-expanding definition (`\cs_new:Npx`), `\exp_not:n {#1}` is equivalent to `##1` rather than `#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  $\star$  `\exp_not:o {<tokens>}`

Expands the  $\langle tokens \rangle$  once, then prevents any further expansion in `x`-type arguments using `\exp_not:n`.

---

\exp\_not:v  $\star$  `\exp_not:v {variable}`

Recovering the content of the  $\langle variable \rangle$ , then prevents expansion of this material in `x`-type arguments using `\exp_not:n`.

---

`\exp_not:v` ★ `\exp_not:v {<tokens>}`

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

**TeXhackers note:** Protected macros that appear in a v-type argument are expanded despite being protected; `\exp_not:n` also has no effect. An internal error occurs if non-characters or active characters remain after full expansion, as the conversion to a control sequence is not possible.

---

`\exp_not:e` ★ `\exp_not:e {<tokens>}`

Expands  $\langle tokens \rangle$  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  $\langle tokens \rangle$  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 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  $\langle tokens \rangle$  are expandable `\exp_stop_f:` terminates the expansion of tokens even if  $\langle more tokens \rangle$  are also expandable. The function itself is an implicit space token. Inside an x-type expansion, it retains its form, but when typeset it produces the underlying space ( $\sqcup$ ).

## 9 Controlled expansion

The `expl3` language makes all efforts to hide the complexity of T<sub>E</sub>X 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 T<sub>E</sub>X 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  $\langle expandable-tokens \rangle$  as that will break badly if unexpandable tokens are encountered in that place!

---

\exp:w ★  
\exp\_end: ★  
New: 2015-08-23

---

\exp:w ⟨expandable tokens⟩ \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>2</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.

---

\exp:w ★  
\exp\_end\_continue\_f:w ★  
New: 2015-08-23

---

\exp:w ⟨expandable-tokens⟩ \exp\_end\_continue\_f:w ⟨further-tokens⟩

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.

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>3</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.

---

<sup>2</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!

<sup>3</sup>In this particular case you may get a character into the output as well as an error message.

---

```
\exp:w          *
\exp_end_continue_f:nw *
```

---

New: 2015-08-23

`\exp:w <expandable-tokens> \exp_end_continue_f:nw <further-tokens>`

The difference to `\exp_end_continue_f:w` is that we first 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.

## 10 Internal functions

---

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

`\:::N`  
`\:::p`  
`\:::c`  
`\:::o`  
`\:::e`  
`\:::f`  
`\:::x`  
`\:::v`  
`\:::V`  
`\:::`

---

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 should not be used outside this module.

---

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

`\:::e_unbraced`  
`\:::f_unbraced`  
`\:::x_unbraced`  
`\:::v_unbraced`  
`\:::V_unbraced`

---

Internal forms for the expansion types which leave the terminal argument 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.

# Part VI

## The **l3tl** package

### Token lists

TeX works with tokens, and L<sup>A</sup>T<sub>E</sub>X3 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.

## 1 Creating and initialising token list variables

---

```
\tl_new:N \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.

---

```
\tl_const:Nn \tl_const:(Nx|cn|cx)
```

Creates a new constant `\tl var` or raises an error if the name is already taken. The value of the `\tl var` is set globally to the `\token list`.

---

```
\tl_clear:N \tl_clear:c  
\tl_gclear:N \tl_gclear:c
```

Clears all entries from the `\tl var`.

---

```
\tl_clear_new:N  
\tl_clear_new:c  
\tl_gclear_new:N  
\tl_gclear_new:c
```

---

```
\tl_clear_new:N <tl var>
```

Ensures that the  $\langle tl\ var \rangle$  exists globally by applying `\tl_new:N` if necessary, then applies `\tl_(g)clear:N` to leave the  $\langle tl\ var \rangle$  empty.

---

```
\tl_set_eq:NN  
\tl_set_eq:(cN|Nc|cc)  
\tl_gset_eq:NN  
\tl_gset_eq:(cN|Nc|cc)
```

---

```
\tl_set_eq:NN <tl var1> <tl var2>
```

Sets the content of  $\langle tl\ var_1 \rangle$  equal to that of  $\langle tl\ var_2 \rangle$ .

---

```
\tl_concat:NNN  
\tl_concat:ccc  
\tl_gconcat:NNN  
\tl_gconcat:ccc
```

---

```
\tl_concat:NNN <tl var1> <tl var2> <tl var3>
```

Concatenates the content of  $\langle tl\ var_2 \rangle$  and  $\langle tl\ var_3 \rangle$  together and saves the result in  $\langle tl\ var_1 \rangle$ . The  $\langle tl\ var_2 \rangle$  is placed at the left side of the new token list.

New: 2012-05-18

---

---

```
\tl_if_exist_p:N *  
\tl_if_exist_p:c *  
\tl_if_exist:NTF *  
\tl_if_exist:cTF *
```

---

```
\tl_if_exist_p:N <tl var>
```

```
\tl_if_exist:NTF <tl var> {(true code)} {(false code)}
```

Tests whether the  $\langle tl\ var \rangle$  is currently defined. This does not check that the  $\langle tl\ var \rangle$  really is a token list variable.

New: 2012-03-03

---

## 2 Adding data to token list variables

---

```
\tl_set:Nn  
\tl_set:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)  
\tl_gset:Nn  
\tl_gset:(NV|Nv|No|Nf|Nx|cn|cV|cv|co|cf|cx)
```

---

```
\tl_set:Nn <tl var> {(tokens)}
```

Sets  $\langle tl\ var \rangle$  to contain  $\langle tokens \rangle$ , removing any previous content from the variable.

---

```
\tl_put_left:Nn  
\tl_put_left:(NV|No|Nx|cn|cV|co|cx)  
\tl_gput_left:Nn  
\tl_gput_left:(NV|No|Nx|cn|cV|co|cx)
```

---

```
\tl_put_left:Nn <tl var> {(tokens)}
```

Appends  $\langle tokens \rangle$  to the left side of the current content of  $\langle tl\ var \rangle$ .

---

```
\tl_put_right:Nn  
\tl_put_right:(NV|No|Nx|cn|cV|co|cx)  
\tl_gput_right:Nn  
\tl_gput_right:(NV|No|Nx|cn|cV|co|cx)
```

---

```
\tl_put_right:Nn <tl var> {(tokens)}
```

Appends  $\langle tokens \rangle$  to the right side of the current content of  $\langle tl\ var \rangle$ .

### 3 Modifying token list variables

---

```
\tl_replace_once:Nnn
\tl_replace_once:cnn
\tl_greplace_once:Nnn
\tl_greplace_once:cnn
```

---

Updated: 2011-08-11

```
\tl_replace_once:Nnn <tl var> {<old tokens>} {<new tokens>}
```

Replaces the first (leftmost) occurrence of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old 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).

---

```
\tl_replace_all:Nnn
\tl_replace_all:cnn
\tl_greplace_all:Nnn
\tl_greplace_all:cnn
```

---

Updated: 2011-08-11

```
\tl_replace_all:Nnn <tl var> {<old tokens>} {<new tokens>}
```

Replaces all occurrences of *<old tokens>* in the *<tl var>* with *<new tokens>*. *<Old 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 *<old tokens>* may remain after the replacement (see `\tl_remove_all:Nn` for an example).

---

```
\tl_remove_once:Nn
\tl_remove_once:cn
\tl_gremove_once:Nn
\tl_gremove_once:cn
```

---

Updated: 2011-08-11

```
\tl_remove_once:Nn <tl var> {<tokens>}
```

Removes the first (leftmost) occurrence 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).

---

```
\tl_remove_all:Nn
\tl_remove_all:cn
\tl_gremove_all:Nn
\tl_gremove_all:cn
```

---

Updated: 2011-08-11

```
\tl_remove_all:Nn <tl var> {<tokens>}
```

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.

### 4 Reassigning token list 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:(Nno|Nnx|cnn|cno|cnx)

\tl\_gset\_rescan:Nnn

\tl\_gset\_rescan:(Nno|Nnx|cnn|cno|cnx)

---

Updated: 2015-08-11

\tl\_set\_rescan:Nnn *tl var* {\i<setup>} {\i<tokens>}

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 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 {\i<setup>} {\i<tokens>}

---

Updated: 2015-08-11

Rescans *tokens* applying the category code régime specified in the *<setup>*, and leaves the resulting tokens in the input stream. (Category codes applied to tokens not explicitly covered by the *<setup>* are those in force at the point of use of \tl\_rescan:nn.) The *<setup>* is run within a group and may contain any valid input, although only changes in category codes are relevant. See also \tl\_set\_rescan:Nnn, which is more robust than using \tl\_set:Nn in the *tokens* argument of \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.

## 5 Token list conditionals

---

\tl\_if\_blank\_p:n \*

\tl\_if\_blank\_p:n {\i<token list>}

\tl\_if\_blank\_p:(V|o) \*

\tl\_if\_blank:nTF {\i<token list>} {\i<true code>} {\i<false code>}

\tl\_if\_blank:nTF \*

Tests if the *<token list>* consists only of blank spaces (*i.e.* contains no item). The test is **true** if *<token list>* is zero or more explicit space characters (explicit tokens with character 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 ★           Tests 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.

---

```
\tl_if_eq:nnTF
```

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.

---

```
\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:(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).

---

```
\tl_if_novalue_p:n ★
\tl_if_novalue:nTF ★
```

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:c ★
\tl_if_single:NNTF ★
\tl_if_single:cNTF ★
```

Updated: 2011-08-13

---

```
\tl_if_single_p:N <tl var>
\tl_if_single:NNTF <tl var> {<true code>} {<false code>}
```

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, 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:nNTF ★
```

Updated: 2011-08-13

---

```
\tl_if_single_p:n {<token list>}
\tl_if_single:nNTF {<token list>} {<true code>} {<false code>}
```

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_case:Nn ★
\tl_case:cn ★
\tl_case:NnTF ★
\tl_case:cnTF ★
```

New: 2013-07-24

---

```
\tl_case:NnTF <test token list variable>
{
  <token list variable case1> {<code case1>}
  <token list variable case2> {<code case2>}
  ...
  <token list variable casen> {<code casen>}
}
{<true code>}
{<false code>}
```

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.

## 6 Mapping to token lists

---

```
\tl_map_function:NN ★
\tl_map_function:cN ★
```

Updated: 2012-06-29

---

```
\tl_map_function:NN <tl var> <function>
```

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`.

---

```
\tl_map_function:nN ★
```

Updated: 2012-06-29

---

```
\tl_map_function:nN {<token list>} <function>
```

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:cn
```

Updated: 2012-06-29

---

```
\tl_map_inline:Nn <tl var> {<inline function>}
```

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`.

---

**\tl\_map\_inline:nn**

Updated: 2012-06-29

---

```
\tl_map_inline:nn {\token list} {\inline function}
```

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\_variable:Nn****\tl\_map\_variable:cNn**

Updated: 2012-06-29

---

```
\tl_map_variable:Nn \tl var {variable} {\code}
```

Stores each *item* of the *tl var* 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. See also \tl\_map\_inline:Nn.

---

**\tl\_map\_variable:nNn**

Updated: 2012-06-29

---

```
\tl_map_variable:nNn {\token list} {variable} {\code}
```

Stores each *item* of the *token 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. See also \tl\_map\_inline:nn.

---

**\tl\_map\_break:** 

Updated: 2012-06-29

---

```
\tl_map_break:
```

Used to terminate a \tl\_map\_... function before all entries in the *token list variable* have been processed. 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: }
    % 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 *tokens* are inserted into the input stream. This depends on the design of the mapping function.

---

\tl\_map\_break:n 

Updated: 2012-06-29

\tl\_map\_break:n {<code>}

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.

## 7 Using token lists

---

\tl\_to\_str:n 
\tl\_to\_str:v 

\tl\_to\_str:n {<token list>}

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.

**T<sub>E</sub>Xhackers note:** This is the  $\varepsilon$ -T<sub>E</sub>X 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 {\\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:c 

\tl\_to\_str:N {tl var}

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  $\langle tl\ var\rangle$  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  $\langle tl\ var\rangle$  directly without an accessor function.

## 8 Working with the content of token lists

---

```
\tl_count:n ★ \tl_count:n {<tokens>}
\tl_count:(V|o) ★
```

New: 2012-05-13

---

Counts the number of  $\langle items\rangle$  in  $\langle tokens\rangle$  and leaves this information in the input stream. Unbraced tokens count as one element as do each token group  $\{\dots\}$ . This process ignores any unprotected spaces within  $\langle tokens\rangle$ . See also  $\backslash tl\_count:N$ . This function requires three expansions, giving an  $\langle integer\ denotation\rangle$ .

---

```
\tl_count:N ★ \tl_count:N <tl var>
\tl_count:c ★
```

New: 2012-05-13

---

Counts the number of token groups in the  $\langle tl\ var\rangle$  and leaves this information in the input stream. Unbraced tokens count as one element as do each token group  $\{\dots\}$ . This process ignores any unprotected spaces within the  $\langle tl\ var\rangle$ . See also  $\backslash tl\_count:n$ . This function requires three expansions, giving an  $\langle integer\ denotation\rangle$ .

---

```
\tl_reverse:n ★ \tl_reverse:n {<token list>}
```

\tl\_reverse:(V|o) ★

---

Updated: 2012-01-08

---

Reverses the order of the  $\langle items\rangle$  in the  $\langle token\ list\rangle$ , 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 preserves unprotected space within the  $\langle token\ list\rangle$ . Tokens are not reversed within braced token groups, which keep their outer set of braces. In situations where performance is important, consider  $\backslash tl\_reverse\_items:n$ . See also  $\backslash tl\_reverse:N$ .

**TeXhackers note:** The result is returned within  $\backslash unexpanded$ , which means that the token list does not expand further when appearing in an x-type argument expansion.

---

```
\tl_reverse:N ★ \tl_reverse:N <tl var>
```

\tl\_reverse:c ★

\tl\_greverse:N ★

\tl\_greverse:c ★

Updated: 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 preserves unprotected spaces within the  $\langle token\ list\ variable\rangle$ . Braced token groups are copied without reversing the order of tokens, but keep the outer set of braces. See also  $\backslash tl\_reverse:n$ , and, for improved performance,  $\backslash tl\_reverse\_items:n$ .

---

```
\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  $\backslash tl\_reverse:n$ .

**TeXhackers note:** The result is returned within  $\backslash unexpanded$ , which means that the token list does not expand further when appearing in an x-type argument expansion.

---

```
\tl_trim_spaces:n ★
\tl_trim_spaces:o ★
```

New: 2011-07-09  
Updated: 2012-06-25

---

```
\tl_trim_spaces:n {<token list>}
```

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the *<token list>* and leaves the result in the input stream.

**TeXhackers note:** The result is returned within `\unexpanded`, which means that the token list does not expand further when appearing in an x-type argument expansion.

---

```
\tl_trim_spaces_apply:nN ★
\tl_trim_spaces_apply:on ★
```

New: 2018-04-12

---

```
\tl_trim_spaces_apply:nN {<token list>} <function>
```

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the *<token list>* and passes the result to the *<function>* as an n-type argument.

---

```
\tl_trim_spaces:N
\tl_trim_spaces:c
\tl_gtrim_spaces:N
\tl_gtrim_spaces:c
```

New: 2011-07-09

---

```
\tl_trim_spaces:N <tl var>
```

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the content of the *<tl var>*. Note that this therefore *resets* the content of the variable.

---

```
\tl_sort:Nn
\tl_sort:cn
\tl_gsort:Nn
\tl_gsort:cn
```

New: 2017-02-06

---

```
\tl_sort:Nn <tl var> {<comparison code>}
```

Sorts the items in the *<tl var>* according to the *<comparison code>*, and assigns the result to *<tl var>*. The details of sorting comparison are described in Section 1.

---

```
\tl_sort:nN ★
```

New: 2017-02-06

---

```
\tl_sort:nN {<token list>} <conditional>
```

Sorts the items in the *<token list>*, using the *<conditional>* to compare items, and leaves the result in the input stream. The *<conditional>* 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 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 argument expansion.

## 9 The first token from a token list

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      *
\tl_head:n      *
\tl_head:(V|v|f) *
```

Updated: 2012-09-09

---

```
\tl_head:n {<token list>}
```

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.

**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 argument expansion.

---

```
\tl_head:w *
```

---

```
\tl_head:w <token list> { } \q_stop
```

Updated: 2012-09-01

---

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      *
\tl_tail:(V|v|f) *
```

Updated: 2012-09-01

---

```
\tl_tail:n {<token list>}
```

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first *item* in the *<token list>*, and leaves the remaining tokens 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 argument expansion.

---

```
\tl_if_head_eq_catcode_p:nN ★ \tl_if_head_eq_catcode_p:nN {⟨token list⟩} ⟨test token⟩
\tl_if_head_eq_catcode:nNTF ★ \tl_if_head_eq_catcode: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 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⟩
                                {⟨true code⟩} {⟨false code⟩}
\tl_if_head_eq_charcode:nNTF ★
\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 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 12 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.

## 10 Using a single item

---

\tl\_item:nn ★  
\tl\_item:Nn ★  
\tl\_item:c ★  
New: 2014-07-17

---

\tl\_item:nn {⟨token list⟩} {⟨integer expression⟩}

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. 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 argument expansion.

---

\tl\_rand\_item:N ★  
\tl\_rand\_item:c ★  
\tl\_rand\_item:n ★  
New: 2016-12-06

---

\tl\_rand\_item:N ⟨tl var⟩  
\tl\_rand\_item:n {⟨token list⟩}

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>H</sub>TeX.

**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 argument expansion.

---

\tl\_range:Nnn ★  
\tl\_range:nnn ★

---

New: 2017-02-17  
Updated: 2017-07-15

---

\tl\_range:Nnn <tl var> {<start index>} {<end index>}  
\tl\_range:nnn {<token list>} {<start index>} {<end index>}

Leaves in the input stream the items from the *<start index>* to the *<end index>* inclusive. Spaces and braces are preserved between the items returned (but never at either end of the list).

Here *<start index>* and *<end index>* 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  $s > l$ .

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 the third to the last, the call is `\tl_range:nnn { <tl> } { 3 } { -2 }`. Similarly, for discarding the last item, we can do `\tl_range:nnn { <tl> } { 1 } { -2 }`.

The behavior of \tl\_range:Nnn is exactly the same, acting on the contents of the tl variable.

For improved 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 *<item>* does not expand further when appearing in an x-type argument expansion.

## 11 Viewing token lists

---

\tl\_show:N      \tl\_show:N *tl var*

\tl\_show:c      Displays the content of the *tl var* on the terminal.

Updated: 2015-08-01

---

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

---

\tl\_show:n      \tl\_show:n {*token list*}

Updated: 2015-08-07

---

**TeXhackers note:** This is similar to the ε-TEx 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.

Updated: 2015-08-01

---

---

\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

---

## 12 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:VnTF \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.

## 13 Scratch token lists

\l\_tmpa\_t1  
\l\_tmpb\_t1

Scratch token lists 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\_t1  
\g\_tmpb\_t1

Scratch token lists 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.

## Part VII

# 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.

## 1 Building strings

---

`\str_new:N`  
`\str_new:c`  
New: 2015-09-18

---

`\str_new:N <str var>`

Creates a new `<str var>` or raises an error if the name is already taken. The declaration is global. The `<str var>` is initially empty.

---

`\str_const:Nn`  
`\str_const:(NV|Nx|cn|cV|cx)`  
New: 2015-09-18  
Updated: 2018-07-28

---

`\str_const:Nn <str var> {<token list>}`

Creates a new constant `<str var>` or raises an error if the name is already taken. The value of the `<str var>` is set globally to the `<token list>`, converted to a string.

---

```
\str_clear:N  
\str_clear:c  
\str_gclear:N  
\str_gclear:c
```

New: 2015-09-18

---

```
\str_clear:N <str var>
```

Clears the content of the *<str var>*.

---

```
\str_clear_new:N  
\str_clear_new:c
```

New: 2015-09-18

---

```
\str_clear_new:N <str var>
```

Ensures that the *<str var>* exists globally by applying `\str_new:N` if necessary, then applies `\str_(g)clear:N` to leave the *<str var>* empty.

---

```
\str_set_eq:NN  
\str_set_eq:(cN|Nc|cc)  
\str_gset_eq:NN  
\str_gset_eq:(cN|Nc|cc)
```

New: 2015-09-18

---

```
\str_set_eq:NN <str var1> <str var2>
```

Sets the content of *<str var<sub>1</sub>>* equal to that of *<str var<sub>2</sub>>*.

---

```
\str_concat:NNN  
\str_concat:ccc  
\str_gconcat:NNN  
\str_gconcat:ccc
```

New: 2017-10-08

---

```
\str_concat:NNN <str var1> <str var2> <str var3>
```

Concatenates the content of *<str var<sub>2</sub>>* and *<str var<sub>3</sub>>* together and saves the result in *<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.

## 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)
```

New: 2015-09-18

Updated: 2018-07-28

---

```
\str_set:Nn <str var> {<token list>}
```

Converts the *<token list>* to a *<string>*, and stores the result in *<str var>*.

---

```
\str_put_left:Nn  
\str_put_left:(NV|Nx|cn|cV|cx)  
\str_gput_left:Nn  
\str_gput_left:(NV|Nx|cn|cV|cx)
```

New: 2015-09-18

Updated: 2018-07-28

---

```
\str_put_left:Nn <str var> {<token list>}
```

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.

---

```
\str_put_right:Nn          \str_put_right:Nn <str var> {<token list>}
\str_put_right:(NV|Nx|cn|cV|cx)
\str_gput_right:Nn
\str_gput_right:(NV|Nx|cn|cV|cx)
```

---

New: 2015-09-18

Updated: 2018-07-28

---

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.

### 3 Modifying string variables

---

```
\str_replace_once:Nnn
\str_replace_once:cnn
\str_greplace_once:Nnn
\str_greplace_once:cnn
```

---

New: 2017-10-08

---

```
\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>*.

---

```
\str_replace_all:Nnn
\str_replace_all:cnn
\str_greplace_all:Nnn
\str_greplace_all:cnn
```

---

New: 2017-10-08

---

```
\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).

---

```
\str_remove_once:Nn
\str_remove_once:cn
\str_gremove_once:Nn
\str_gremove_once:cn
```

---

New: 2017-10-08

---

```
\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>*.

---

```
\str_remove_all:Nn
\str_remove_all:cn
\str_gremove_all:Nn
\str_gremove_all:cn
```

---

New: 2017-10-08

---

```
\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,

```
\str_set:Nn \l_tmpa_str {abbcd} \str_remove_all:Nn \l_tmpa_str
{bc}
```

results in *\l\_tmpa\_str* containing abcd.

### 4 String conditionals

---

```
\str_if_exist_p:N *
\str_if_exist_p:c *
\str_if_exist:NTF *
\str_if_exist:cTF *
```

---

New: 2015-09-18

---

```
\str_if_exist_p:N <str var>
\str_if_exist:NTF <str var> {<true code>} {<false code>}
```

Tests whether the *<str var>* is currently defined. This does not check that the *<str var>* really is a string.

---

```
\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 ★
```

---

New: 2015-09-18

---

```
\str_if_eq_p:NN ★ \str_if_eq_p:NN <str var1> <str var2>
\str_if_eq_p:(Nc|cN|cc) ★ \str_if_eq:NNTF <str var1> <str var2> {{true code}} {{false code}}
\str_if_eq:NNTF ★ Compares the content of two <str variables> and is logically true if the two contain the
\str_if_eq:(Nc|cN|cc)TF ★ same characters in the same order.
```

---

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) ★ \str_if_eq:nnTF {{tl1}} {{tl2}} {{true code}} {{false code}}
\str_if_eq:nnTF ★
\str_if_eq:(Vn|on|no|nV|VV|vn|nv)TF ★
\str_if_eq_p:ee ★
\str_if_eq:eeTF ★
```

---

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.

---

```
\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
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 <string2> is found inside
<string1>.
```

---

---

\str_case:nn	★	\str_case:nnTF {⟨test string⟩}
\str_case:(on nV nv)	★	{
\str_case:nnTF	★	{⟨string case <sub>1</sub> ⟩} {⟨code case <sub>1</sub> ⟩}
\str_case:(on nV nv)TF	★	{⟨string case <sub>2</sub> ⟩} {⟨code case <sub>2</sub> ⟩}
		...
		{⟨string case <sub>n</sub> ⟩} {⟨code case <sub>n</sub> ⟩}
		}
		{⟨true code⟩}
		{⟨false code⟩}

---

New: 2013-07-24  
Updated: 2015-02-28

---

```
\str_case:nnTF {⟨test string⟩}
{
  {⟨string case1⟩} {⟨code case1⟩}
  {⟨string case2⟩} {⟨code case2⟩}
  ...
  {⟨string casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}
```

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.

---

\str_case_e:nn	★	\str_case_e:nnTF {⟨test string⟩}
\str_case_e:nnTF	★	{

---

New: 2018-06-19

---

```
\str_case_e:nnTF {⟨test string⟩}
{
  {⟨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.

## 5 Mapping to strings

---

\str_map_function:NN	★	\str_map_function:NN {⟨str var⟩} {⟨function⟩}
\str_map_function:cN	★	

---

New: 2017-11-14

---

Applies ⟨function⟩ to every ⟨character⟩ in the ⟨str var⟩ including spaces. See also \str\_map\_function:nN.

---

\str_map_function:nN	★	\str_map_function:nN {⟨token list⟩} {⟨function⟩}
----------------------	---	--

---

New: 2017-11-14

---

Converts the ⟨token list⟩ to a ⟨string⟩ then applies ⟨function⟩ to every ⟨character⟩ in the ⟨string⟩ including spaces. See also \str\_map\_function:NN.

---

\str\_map\_inline:Nn  
\str\_map\_inline:cn

---

New: 2017-11-14

\str\_map\_inline:Nn <str var> {<inline function>}

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. See also \str\_map\_function:NN.

---

\str\_map\_inline:nn  
New: 2017-11-14

---

\str\_map\_inline:nn {<token list>} {<inline function>}

Converts the *<token list>* to a *<string>* then applies the *<inline function>* to every *<character>* in the *<string>* including spaces. The *<inline function>* should consist of code which receives the *<character>* as #1. See also \str\_map\_function:NN.

---

\str\_map\_variable>NNn  
\str\_map\_variable:cNn

---

New: 2017-11-14

\str\_map\_variable>NNn <str var> <variable> {<code>}

Stores each *<character>* of 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. See also \str\_map\_inline:Nn.

---

\str\_map\_variable:nNn  
New: 2017-11-14

---

\str\_map\_variable:nNn {<token list>} <variable> {<code>}

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. See also \str\_map\_inline:Nn.

---

\str\_map\_break: ☆  
New: 2017-10-08

---

\str\_map\_break:

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 T<sub>E</sub>X 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 ☆

New: 2017-10-08

\str\_map\_break:n {<code>}

Used to terminate a \str\_map\_... function before all characters in the *<string>* have been processed, inserting the *<code>* 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.

**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.

## 6 Working with the content of strings

---

\str\_use:N ☆

\str\_use:c ☆

New: 2015-09-18

\str\_use:N <str var>

Recovers the content of a *<str 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 *<str>* directly without an accessor function.

---

\str\_count:N

\* \str\_count:n {<token list>}

\str\_count:c

\*

\str\_count:n

\*

\str\_count\_ignore\_spaces:n \*

New: 2015-09-18

Leaves in the input stream the number of characters in the string representation of *<token list>*, 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 \*

\* \str\_count\_spaces:n {<token list>}

\str\_count\_spaces:c \*

Leaves in the input stream the number of space characters in the string representation of *<token list>*, 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.

---

<u>\str_range:Nnn</u> <u>\str_range:cnn</u> <u>\str_range:nnn</u> <u>\str_range_ignore_spaces:nnn</u>	<span style="color: red;">*</span> \str_range:nnn { <i>token list</i> } { <i>start index</i> } { <i>end index</i> }
--	---

---

New: 2015-09-18

Converts the *token list* to a *string*, and leaves in the input stream the characters from the *start index* to the *end index* inclusive. Spaces are preserved and counted as items (contrast this with `\tl_range:nnn` where spaces are not counted as items and are possibly discarded from the output).

Here *start index* and *end index* 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 *start index* must always be smaller than or equal to the *end index*: 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 `\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 **bcd**, four instances of **bc e** and eight instances of **bcd**.

## 7 String manipulation

---

```
\str_lower_case:n ★ \str_lower_case:n {⟨tokens⟩}
\str_lower_case:f ★ \str_upper_case:n {⟨tokens⟩}
```

---

Converts the input ⟨tokens⟩ to their string representation, as described for `\tl_to_str:n`, and then to the lower or upper case representation using a one-to-one mapping as described by the Unicode Consortium file `UnicodeData.txt`.

New: 2015-03-01

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_upper_case:f { \tl_head:n {#1} }
        \str_lower_case: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_fold_case:n` for this situation (case folding is distinct from lower casing).
- Case changing text for typesetting: see the `\tl_lower_case:n(n)`, `\tl_upper_case:n(n)` and `\tl_mixed_case:n(n)` functions which correctly deal with context-dependence and other factors appropriate to text case changing.

**TeXhackers note:** As with all `expl3` functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with pdfTeX *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both XeTeX and LuaTeX.

---

```
\str_fold_case:n ★
\str_fold_case:v ★
```

New: 2014-06-19  
Updated: 2016-03-07

---

```
\str_fold_case:n {⟨tokens⟩}
```

Converts the input ⟨tokens⟩ to their string representation, as described for `\tl_to_str:n`, 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_fold_case: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_fold_case: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).

**TeXhackers note:** As with all `expl3` functions, the input supported by `\str_fold_case:n` is *engine-native* characters which are or interoperate with UTF-8. As such, when used with `pdfTeX` *only* the Latin alphabet characters A–Z are case-folded (*i.e.* the ASCII range which coincides with UTF-8). Full UTF-8 support is available with both `XeTeX` and `LuaTeX`, subject only to the fact that `XeTeX` in particular has issues with characters of code above hexadecimal 0xFFFF when interacting with `\tl_to_str:n`.

## 8 Viewing strings

---

```
\str_show:N
\str_show:c
\str_show:n
```

New: 2015-09-18

---

```
\str_show:N ⟨str var⟩
```

Displays the content of the ⟨str var⟩ on the terminal.

## 9 Constant token lists

---

<code>\c_ampersand_str</code>	Constant strings, containing a single character token, with category code 12.
<code>\c_atsign_str</code>	
<code>\c_backslash_str</code>	
<code>\c_left_brace_str</code>	
<code>\c_right_brace_str</code>	
<code>\c_circumflex_str</code>	
<code>\c_colon_str</code>	
<code>\c_dollar_str</code>	
<code>\c_hash_str</code>	
<code>\c_percent_str</code>	
<code>\c_tilde_str</code>	
<code>\c_underscore_str</code>	

---

New: 2015-09-19

---

## 10 Scratch strings

---

<code>\l_tmpa_str</code>	Scratch strings 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.
<code>\g_tmpa_str</code>	Scratch strings 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.

---

# Part VIII

## The **I3quark** package

### Quarks

Two special types of constants in LATEX3 are “quarks” and “scan marks”. By convention all constants of type quark start out with `\q_`, and scan marks start with `\s_`.

## 1 Quarks

Quarks are control sequences 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. An example of the quark testing functions and their use in recursion can be seen in the implementation of `\clist_map_function:NN`.

## 2 Defining quarks

---

`\quark_new:N` `\quark_new:N` *<quark>*

Creates a new *<quark>* which expands only to *<quark>*. The *<quark>* 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}
```

<u>\q_mark</u>	Used as a marker for delimited arguments when \q_stop is already in use.
<u>\q_nil</u>	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).
<u>\q_no_value</u>	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.

### 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:NTF *
```

Tests if the *<token>* is equal to \q\_nil.

---

```
\quark_if_nil_p:n   *
\quark_if_nil_p:(o|V) *
\quark_if_nil:nTF   *
\quark_if_nil:(o|V)TF *
```

Tests if the *<token list>* contains only \q\_nil (distinct from *<token list>* being empty or containing \q\_nil plus one or more other tokens).

---

```
\quark_if_no_value_p:N *
\quark_if_no_value_p:c *
\quark_if_no_value:NTF *
\quark_if_no_value:cTF *
```

Tests if the *<token>* is equal to \q\_no\_value.

---

```
\quark_if_no_value_p:n *
\quark_if_no_value_p:nTF *
```

Tests if the *<token list>* contains only \q\_no\_value (distinct from *<token list>* being empty or containing \q\_no\_value plus one or more other tokens).

### 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 5.

---

\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_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. 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_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. 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 {{<token list>}} \<type>_map_break:
```

---

```
\quark_if_recursion_tail_break:nN
```

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:`.

## 5 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`.

## 6 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 T<sub>E</sub>X 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 `\bgroup`).

---

`\scan_new:N`

New: 2018-04-01

---

`\scan_new:N <scan mark>`

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`

New: 2018-04-01

---

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`.

---

```
\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 T<sub>E</sub>X error if `\s_stop` is absent.

# Part IX

## The **l3seq** package

### Sequences and stacks

LATEX3 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 LATEX3. This is achieved using a number of dedicated stack functions.

## 1 Creating and initialising sequences

---

`\seq_new:N`  
`\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:c`  
`\seq_gclear:N`  
`\seq_gclear:c`

Clears all items from the *(sequence)*.

---

`\seq_clear_new:N`  
`\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:(cN|Nc|cc)`  
`\seq_gset_eq:NN`  
`\seq_gset_eq:(cN|Nc|cc)`

Sets the content of *(sequence<sub>1</sub>)* equal to that of *(sequence<sub>2</sub>)*.

---

<code>\seq_set_from_clist:NN</code>	<code>\seq_set_from_clist:NN &lt;sequence&gt; &lt;comma-list&gt;</code>
<code>\seq_set_from_clist:(cN Nc cc)</code>	
<code>\seq_set_from_clist:Nn</code>	
<code>\seq_set_from_clist:cn</code>	
<code>\seq_gset_from_clist:NN</code>	
<code>\seq_gset_from_clist:(cN Nc cc)</code>	
<code>\seq_gset_from_clist:Nn</code>	
<code>\seq_gset_from_clist:cn</code>	

New: 2014-07-17

Converts the data in the *(comma list)* into a *(sequence)*: the original *(comma list)* is unchanged.

---

```
\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
```

New: 2011-08-15  
Updated: 2012-07-02

---

```
\seq_set_split:Nnn <sequence> {(delimiter)} {<token list>}
```

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. 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>*.

---

```
\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc
```

---

```
\seq_concat:NNN <sequence>_1 <sequence>_2 <sequence>_3
```

Concatenates the content of *<sequence>\_2* and *<sequence>\_3* together and saves the result in *<sequence>\_1*. The items in *<sequence>\_2* are placed at the left side of the new sequence.

---

```
\seq_if_exist_p:N *
\seq_if_exist_p:c *
\seq_if_exist:NTF *
\seq_if_exist:cTF *
```

New: 2012-03-03

---

```
\seq_if_exist_p:N <sequence>
\seq_if_exist:NTF <sequence> {(true code)} {<false code>}
```

Tests whether the *<sequence>* is currently defined. This does not check that the *<sequence>* really is a sequence variable.

## 2 Appending data to sequences

---

```
\seq_put_left:Nn
\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)
```

---

```
\seq_put_left:Nn <sequence> {<item>}
```

Appends the *<item>* to the left of the *<sequence>*.

---

```
\seq_put_right:Nn
\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)
```

---

```
\seq_put_right:Nn <sequence> {<item>}
```

Appends the *<item>* to the right of the *<sequence>*.

## 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:cN
```

Updated: 2012-05-14

---

```
\seq_get_left:NN <sequence> <token list variable>
```

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`.

---

```
\seq_get_right:NN  
\seq_get_right:cN
```

---

Updated: 2012-05-19

```
\seq_get_right:NN <sequence> <token list variable>
```

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`.

---

```
\seq_pop_left:NN  
\seq_pop_left:cN
```

---

Updated: 2012-05-14

```
\seq_pop_left:NN <sequence> <token list variable>
```

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`.

---

```
\seq_gpop_left:NN  
\seq_gpop_left:cN
```

---

Updated: 2012-05-14

```
\seq_gpop_left:NN <sequence> <token list variable>
```

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>*. 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:cN
```

---

Updated: 2012-05-19

```
\seq_pop_right:NN <sequence> <token list variable>
```

Pops the right-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`.

---

```
\seq_gpop_right:NN  
\seq_gpop_right:cN
```

---

Updated: 2012-05-19

```
\seq_gpop_right:NN <sequence> <token list variable>
```

Pops the right-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>*. 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:cn *
```

---

New: 2014-07-17

```
\seq_item:Nn <sequence> {\<integer expression>}
```

Indexing items in the *<sequence>* from 1 at the top (left), this function evaluates the *<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 argument expansion.

---

```
\seq_rand_item:N *
```

---

```
\seq_rand_item:c *
```

---

New: 2016-12-06

```
\seq_rand_item:N <seq var>
```

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 XeTeX.

**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 argument expansion.

## 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.

---

\seq\_get\_left:NNTF  
\seq\_get\_left:cNTF

---

New: 2012-05-14  
Updated: 2012-05-19

\seq\_get\_left:NNTF *sequence* *token list variable* {*true code*} {*false code*}

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 left-most item from the *sequence* in the *token list variable* without removing it from the *sequence*, then leaves the *true code* in the input stream. The *token list variable* is assigned locally.

---

\seq\_get\_right:NNTF  
\seq\_get\_right:cNTF

---

New: 2012-05-19

\seq\_get\_right:NNTF *sequence* *token list variable* {*true code*} {*false code*}

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 right-most item from the *sequence* in the *token list variable* without removing it from the *sequence*, then leaves the *true code* in the input stream. The *token list variable* is assigned locally.

---

\seq\_pop\_left:NNTF  
\seq\_pop\_left:cNTF

---

New: 2012-05-14  
Updated: 2012-05-19

\seq\_pop\_left:NNTF *sequence* *token list variable* {*true code*} {*false code*}

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 left-most item from the *sequence* in the *token list variable*, *i.e.* removes the item from the *sequence*, then leaves the *true code* in the input stream. Both the *sequence* and the *token list variable* are assigned locally.

---

\seq\_gpop\_left:NNTF  
\seq\_gpop\_left:cNTF

---

New: 2012-05-14  
Updated: 2012-05-19

\seq\_gpop\_left:NNTF *sequence* *token list variable* {*true code*} {*false code*}

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 left-most item from the *sequence* in the *token list variable*, *i.e.* removes the item from the *sequence*, then leaves the *true code* in the input stream. The *sequence* is modified globally, while the *token list variable* is assigned locally.

---

\seq\_pop\_right:NNTF  
\seq\_pop\_right:cNTF

---

New: 2012-05-19

\seq\_pop\_right:NNTF *sequence* *token list variable* {*true code*} {*false code*}

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 right-most item from the *sequence* in the *token list variable*, *i.e.* removes the item from the *sequence*, then leaves the *true code* in the input stream. Both the *sequence* and the *token list variable* are assigned locally.

---

```
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
```

New: 2012-05-19

---

`\seq_gpop_right:NNTF <sequence> <token list variable> {<true code>} {<false code>}`

If the  $\langle \text{sequence} \rangle$  is empty, leaves the  $\langle \text{false code} \rangle$  in the input stream. The value of the  $\langle \text{token list variable} \rangle$  is not defined in this case and should not be relied upon. If the  $\langle \text{sequence} \rangle$  is non-empty, pops the right-most item from the  $\langle \text{sequence} \rangle$  in the  $\langle \text{token list variable} \rangle$ , i.e. removes the item from the  $\langle \text{sequence} \rangle$ , then leaves the  $\langle \text{true code} \rangle$  in the input stream. The  $\langle \text{sequence} \rangle$  is modified globally, while the  $\langle \text{token list variable} \rangle$  is assigned locally.

## 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:c
\seq_gremove_duplicates:N
\seq_gremove_duplicates:c
```

`\seq_remove_duplicates:N <sequence>`

Removes duplicate items from the  $\langle \text{sequence} \rangle$ , leaving the left most copy of each item in the  $\langle \text{sequence} \rangle$ . The  $\langle \text{item} \rangle$  comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

**TeXhackers note:** This function iterates through every item in the  $\langle \text{sequence} \rangle$  and does a comparison with the  $\langle \text{items} \rangle$  already checked. It is therefore relatively slow with large sequences.

---

```
\seq_remove_all:Nn
\seq_remove_all:cn
\seq_gremove_all:Nn
\seq_gremove_all:cn
```

`\seq_remove_all:Nn <sequence> {<item>}`

Removes every occurrence of  $\langle \text{item} \rangle$  from the  $\langle \text{sequence} \rangle$ . The  $\langle \text{item} \rangle$  comparison takes place on a token basis, as for `\tl_if_eq:nnTF`.

---

```
\seq_reverse:N
\seq_reverse:c
\seq_greverse:N
\seq_greverse:c
```

New: 2014-07-18

---

`\seq_reverse:N <sequence>`

Reverses the order of the items stored in the  $\langle \text{sequence} \rangle$ .

---

```
\seq_sort:Nn
\seq_sort:cn
\seq_gsort:Nn
\seq_gsort:cn
```

New: 2017-02-06

---

`\seq_sort:Nn <sequence> {<comparison code>}`

Sorts the items in the  $\langle \text{sequence} \rangle$  according to the  $\langle \text{comparison code} \rangle$ , and assigns the result to  $\langle \text{sequence} \rangle$ . The details of sorting comparison are described in Section 1.

## 6 Sequence conditionals

---

```
\seq_if_empty_p:N *
\seq_if_empty_p:c *
\seq_if_empty:NTF *
\seq_if_empty:cTF *
```

`\seq_if_empty_p:N <sequence>`

`\seq_if_empty:c <sequence> {<true code>} {<false code>}`

Tests if the  $\langle \text{sequence} \rangle$  is empty (containing no items).

---

```
\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*.

## 7 Mapping to sequences

---

```
\seq_map_function>NN ★
\seq_map_function:cN ★
```

---

Updated: 2012-06-29

```
\seq_map_function:NN <sequence> <function>
```

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. 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:cN
```

---

Updated: 2012-06-29

```
\seq_map_inline:Nn <sequence> {\inline function}
```

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_variable>NNn
\seq_map_variable:(Ncn|cNn|ccn)
```

---

Updated: 2012-06-29

```
\seq_map_variable>NNn <sequence> <variable> {\code}
```

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. The *items* are returned from left to right.

---

```
\seq_map_break: ★
```

---

Updated: 2012-06-29

```
\seq_map_break:
```

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 ★

Updated: 2012-06-29

\seq\_map\_break:n {<code>}

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\_count:N ★  
\seq\_count:c ★

New: 2012-07-13

\seq\_count:N {<sequence>}

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.

## 8 Using the content of sequences directly

---

\seq\_use:Nnnn ★  
\seq\_use:cnnn ★

New: 2013-05-26

\seq\_use:Nnnn {<seq var>} {<separator between two>} {<separator between more than two>} {<separator between final two>}

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 argument expansion.

---

\seq\_use:Nn ★  
\seq\_use:cN ★  
New: 2013-05-26

---

\seq\_use:Nn *seq var* {*separator*}

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.

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 argument expansion.

## 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:cN  
Updated: 2012-05-14

---

\seq\_get:NN *sequence* <token list variable>

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:cN  
Updated: 2012-05-14

---

\seq\_pop:NN *sequence* <token list variable>

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:cN  
Updated: 2012-05-14

---

\seq\_gpop:NN *sequence* <token list variable>

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:cNTF  
New: 2012-05-14  
Updated: 2012-05-19

---

\seq\_get:NNTF *sequence* <token list variable> {*true code*} {*false code*}

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:cNTF

New: 2012-05-14  
Updated: 2012-05-19

---

\seq\_pop:NNTF *sequence* {*token list variable*} {*true code*} {*false code*}

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.

---

\seq\_gpop:NNTF  
\seq\_gpop:cNTF

New: 2012-05-14  
Updated: 2012-05-19

---

\seq\_gpop:NNTF *sequence* {*token list variable*} {*true code*} {*false code*}

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:(NV|Nv|No|Nx|cn|cV|cv|co|cx)  
\seq\_gpush:Nn  
\seq\_gpush:(NV|Nv|No|Nx|cn|cV|cv|co|cx)

\seq\_push:Nn *sequence* {*item*}

Adds the {*item*} to the top of the *sequence*.

## 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 *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 *seq var* are straightforward. For instance, \seq\_count:N *seq var* expands to the number of items, while \seq\_if\_in:NnTF *seq var* {*item*} tests if the *item* is in the set.

Adding an *item* to a set *seq var* can be done by appending it to the *seq var* if it is not already in the *seq var*:

```
\seq_if_in:NnF seq var {item}  
{ \seq_put_right:Nn seq var {item} }
```

Removing an *item* from a set *seq var* can be done using \seq\_remove\_all:Nn,

```
\seq_remove_all:Nn seq var {item}
```

The intersection of two sets *seq var<sub>1</sub>* and *seq var<sub>2</sub>* can be stored into *seq var<sub>3</sub>* by collecting items of *seq var<sub>1</sub>* which are in *seq var<sub>2</sub>*.

```

\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 \backslash \text{l\_}\langle \text{pkg} \rangle\text{\_internal\_seq} <seq var1>
\seq_map_inline:Nn <seq var2>
{ \seq_remove_all:Nn \backslash \text{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> \backslash \text{l\_}\langle \text{pkg} \rangle\text{\_internal\_seq}

```

## 11 Constant and scratch sequences

---

**\c\_empty\_seq** Constant that is always empty.

New: 2012-07-02

---

---

`\l_tmpa_seq`  
`\l_tmpb_seq`

New: 2012-04-26

Scratch sequences 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_seq`  
`\g_tmpb_seq`

New: 2012-04-26

Scratch sequences 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.

## 12 Viewing sequences

---

`\seq_show:N`  
`\seq_show:c`

Updated: 2015-08-01

`\seq_show:N <sequence>`

Displays the entries in the *<sequence>* in the terminal.

---

`\seq_log:N`  
`\seq_log:c`

New: 2014-08-12

Updated: 2015-08-01

`\seq_log:N <sequence>`

Writes the entries in the *<sequence>* in the log file.

# Part X

## 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`”).

## 1 Integer expressions

---

`\int_eval:n`  $\star$  `\int_eval:n {<integer expression>}`

Evaluates the  $\langle\text{integer expression}\rangle$ , expanding any integer and token list variables within the  $\langle\text{expression}\rangle$  to their content (without requiring `\int_use:N`/`\tl_use:N`) and applying the standard mathematical rules. For example both

```
\int_eval: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_eval:n { \l_my_tl + \l_my_int * 3 - ( 3 + 4 * 5 ) }
```

both evaluate to  $-6$ . The  $\{<\text{integer expression}>\}$  may contain the operators `+`, `-`, `*` and `/`, along with parenthesis `(` and `)`. Any functions within the expressions should expand to an  $\langle\text{integer denotation}\rangle$ : a sequence of a sign and digits matching the regex `\-?[0-9]+`. After expansion `\int_eval:n` yields an  $\langle\text{integer denotation}\rangle$  which is left in the input stream.

**TEXhackers note:** Exactly two expansions are needed to evaluate `\int_eval:n`. The result is *not* an  $\langle\text{internal integer}\rangle$ , and therefore requires suitable termination if used in a TeX-style integer assignment.

---

`\int_eval:w`  $\star$  `\int_eval:w <integer expression>`

New: 2018-03-30

Evaluates the  $\langle\text{integer expression}\rangle$  as described for `\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 `\scan_stop`: it is removed, otherwise not. Spaces do *not* terminate the expression.

---

`\int_abs:n`  $\star$  `\int_abs:n {<integer expression>}`

Updated: 2012-09-26

Evaluates the  $\langle\text{integer expression}\rangle$  as described for `\int_eval:n` and leaves the absolute value of the result in the input stream as an  $\langle\text{integer denotation}\rangle$  after two expansions.

---

**\int\_div\_round:nn** ★ `\int_div_round:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Updated: 2012-09-26

Evaluates the two *⟨integer expressions⟩* 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 *⟨integer expression⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

---

**\int\_div\_truncate:nn** ★ `\int_div_truncate:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

Updated: 2012-02-09

Evaluates the two *⟨integer expressions⟩* 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 *⟨integer denotation⟩* after two expansions.

---

**\int\_max:nn** ★ `\int_max:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`  
**\int\_min:nn** ★ `\int_min:nn {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}`

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 {⟨intexpr₁⟩} {⟨intexpr₂⟩}` times *⟨intexpr₂⟩* from *⟨intexpr₁⟩*. Thus, the result has the same sign as *⟨intexpr₁⟩* and its absolute value is strictly less than that of *⟨intexpr₂⟩*. The result is left in the input stream as an *⟨integer denotation⟩* after two expansions.

## 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⟩ {⟨integer expression⟩}`  
**\int\_const:cn**

Updated: 2011-10-22

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**  
**\int\_gzero:N** `\int_gzero:N ⟨integer⟩`  
**\int\_gzero:c**

Sets *⟨integer⟩* to 0.

---

**\int\_zero\_new:N** `\int_zero_new:N ⟨integer⟩`  
**\int\_zero\_new:c**  
**\int\_gzero\_new:N** `\int_gzero_new:N ⟨integer⟩`  
**\int\_gzero\_new:c**

New: 2011-12-13

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.

---

```
\int_set_eq:NN
\int_set_eq:(cN|Nc|cc)
\int_gset_eq:NN
\int_gset_eq:(cN|Nc|cc)
```

```
\int_set_eq:NN <integer1> <integer2>
```

Sets the content of  $\langle \text{integer}_1 \rangle$  equal to that of  $\langle \text{integer}_2 \rangle$ .

---

```
\int_if_exist_p:N *
\int_if_exist_p:c *
\int_if_exist:NTF *
\int_if_exist:cTF *
```

```
\int_if_exist_p:N <int>
```

```
\int_if_exist:NTF <int> {<true code>} {<false code>}
```

Tests whether the  $\langle \text{int} \rangle$  is currently defined. This does not check that the  $\langle \text{int} \rangle$  really is an integer variable.

New: 2012-03-03

---

### 3 Setting and incrementing integers

---

```
\int_add:Nn
\int_add:cn
\int_gadd:Nn
\int_gadd:cn
```

```
\int_add:Nn <integer> {<integer expression>}
```

Adds the result of the  $\langle \text{integer expression} \rangle$  to the current content of the  $\langle \text{integer} \rangle$ .

Updated: 2011-10-22

---

---

```
\int_decr:N
\int_decr:c
\int_gdecr:N
\int_gdecr:c
```

```
\int_decr:N <integer>
```

Decreases the value stored in  $\langle \text{integer} \rangle$  by 1.

---

```
\int_incr:N
\int_incr:c
\int_gincr:N
\int_gincr:c
```

```
\int_incr:N <integer>
```

Increases the value stored in  $\langle \text{integer} \rangle$  by 1.

---

```
\int_set:Nn
\int_set:cn
\int_gset:Nn
\int_gset:cn
```

```
\int_set:Nn <integer> {<integer expression>}
```

Sets  $\langle \text{integer} \rangle$  to the value of  $\langle \text{integer expression} \rangle$ , which must evaluate to an integer (as described for `\int_eval:n`).

Updated: 2011-10-22

---

---

```
\int_sub:Nn
\int_sub:cn
\int_gsub:Nn
\int_gsub:cn
```

```
\int_sub:Nn <integer> {<integer expression>}
```

Subtracts the result of the  $\langle \text{integer expression} \rangle$  from the current content of the  $\langle \text{integer} \rangle$ .

Updated: 2011-10-22

---

## 4 Using integers

---

```
\int_use:N      *
\int_use:c     *
```

Updated: 2011-10-22

---

`\int_use:N <integer>`

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`).

**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.

## 5 Integer expression conditionals

---

```
\int_compare_p:nNn * \int_compare_p:nNn {{intexpr_1}} {relation} {{intexpr_2}}
\int_compare:nNnTF *
```

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	<

---

```
\int_compare_p:n ★ \int_compare_p:n
\int_compare:nTF ★
  {
    ⟨intexpr_1⟩ ⟨relation_1⟩
    ...
    ⟨intexpr_N⟩ ⟨relation_N⟩
    ⟨intexpr_{N+1}⟩
  }
\int_compare:nTF
{
  ⟨intexpr_1⟩ ⟨relation_1⟩
  ...
  ⟨intexpr_N⟩ ⟨relation_N⟩
  ⟨intexpr_{N+1}⟩
}
{⟨true code⟩} {⟨false code⟩}
```

---

Updated: 2013-01-13

---

This function evaluates the  $\langle\text{integer expressions}\rangle$  as described for `\int_eval:n` and compares consecutive result using the corresponding  $\langle\text{relation}\rangle$ , namely it compares  $\langle\text{intexpr}_1\rangle$  and  $\langle\text{intexpr}_2\rangle$  using the  $\langle\text{relation}_1\rangle$ , then  $\langle\text{intexpr}_2\rangle$  and  $\langle\text{intexpr}_3\rangle$  using the  $\langle\text{relation}_2\rangle$ , until finally comparing  $\langle\text{intexpr}_N\rangle$  and  $\langle\text{intexpr}_{N+1}\rangle$  using the  $\langle\text{relation}_N\rangle$ . The test yields `true` if all comparisons are `true`. Each  $\langle\text{integer expression}\rangle$  is evaluated only once, and the evaluation is lazy, in the sense that if one comparison is `false`, then no other  $\langle\text{integer expression}\rangle$  is evaluated and no other comparison is performed. The  $\langle\text{relations}\rangle$  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	$\neq$

---

```
\int_case:nn  * \int_case:nnTF {<test integer expression>}
\int_case:nnTF *
{
  {<intexpr case1>} {<code case1>}
  {<intexpr case2>} {<code case2>}
  ...
  {<intexpr casen>} {<code casen>}
}
{<true code>}
{<false code>}
```

---

New: 2013-07-24

---

This function evaluates the *<test integer expression>* and compares this in turn to each of the *<integer 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 *\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_p:n  *
\int_if_odd:nTF *
```

---

This function first evaluates the *<integer expression>* as described for *\int\_eval:n*. It then evaluates if this is odd or even, as appropriate.

## 6 Integer expression loops

---

```
\int_do_until:nNnn  * \int_do_until:nNnn {<intexpr1>} <relation> {<intexpr2>} {<code>}
```

---

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<integer expressions>* as described for *\int\_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**.

---

```
\int_do_while:nNnn  * \int_do_while:nNnn {<intexpr1>} <relation> {<intexpr2>} {<code>}
```

---

Places the *<code>* in the input stream for TeX to process, and then evaluates the relationship between the two *<integer expressions>* as described for *\int\_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>\int_until_do:nNnn</code>	<code>\int_until_do:nNnn {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}</code> Evaluates the relationship between the two <i>&lt;integer expressions&gt;</i> as described for <code>\int_compare:nNnTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <code>false</code> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nNnm</code>	<code>\int_while_do:nNnm {\langle intexpr_1 \rangle} {\langle relation \rangle} {\langle intexpr_2 \rangle} {\langle code \rangle}</code> Evaluates the relationship between the two <i>&lt;integer expressions&gt;</i> as described for <code>\int_compare:nNnTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <code>true</code> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\int_do_until:nn</code>	<code>\int_do_until:nn {\langle integer relation \rangle} {\langle code \rangle}</code> Places the <i>&lt;code&gt;</i> in the input stream for TeX to process, and then evaluates the <i>&lt;integer relation&gt;</i> as described for <code>\int_compare:nTF</code> . If the test is <code>false</code> 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 <code>true</code> .
<code>\int_do_while:nn</code>	<code>\int_do_while:nn {\langle integer relation \rangle} {\langle code \rangle}</code> Places the <i>&lt;code&gt;</i> in the input stream for TeX to process, and then evaluates the <i>&lt;integer relation&gt;</i> as described for <code>\int_compare:nTF</code> . If the test is <code>true</code> 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 <code>false</code> .
<code>\int_until_do:nn</code>	<code>\int_until_do:nn {\langle integer relation \rangle} {\langle code \rangle}</code> Evaluates the <i>&lt;integer relation&gt;</i> as described for <code>\int_compare:nTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <code>false</code> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\int_while_do:nn</code>	<code>\int_while_do:nn {\langle integer relation \rangle} {\langle code \rangle}</code> Evaluates the <i>&lt;integer relation&gt;</i> as described for <code>\int_compare:nTF</code> , and then places the <i>&lt;code&gt;</i> in the input stream if the <i>&lt;relation&gt;</i> is <code>true</code> . After the <i>&lt;code&gt;</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>false</code> .

## 7 Integer step functions

---

```
\int_step_function:nN ★  
\int_step_function:nnN ★  
\int_step_function:nnnN ★
```

---

New: 2012-06-04  
Updated: 2018-04-22

---

```
\int_step_function:nN {<final value>} <function>  
\int_step_function:nnN {<initial value>} {<final value>} <function>  
\int_step_function:nnnN {<initial value>} {<step>} {<final value>} <function>
```

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer expressions. 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 }  
\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 `\int_step_function:nN` and `\int_step_function:nnN` both use a fixed *<step>* of 1, and in the case of `\int_step_function:nN` the *<initial value>* is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

---

```
\int_step_inline:nn  
\int_step_inline:nnn  
\int_step_inline:nnnn
```

---

New: 2012-06-04  
Updated: 2018-04-22

---

```
\int_step_inline:nn {<final value>} {<code>}  
\int_step_inline:nnn {<initial value>} {<final value>} {<code>}  
\int_step_inline:nnnn {<initial value>} {<step>} {<final value>} {<code>}
```

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer 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 `#1` replaced by the current *<value>*. Thus the *<code>* should define a function of one argument (`#1`).

The functions `\int_step_inline:nn` and `\int_step_inline:nnn` both use a fixed *<step>* of 1, and in the case of `\int_step_inline:nn` the *<initial value>* is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

---

```
\int_step_variable:nNn  
\int_step_variable:nnNn  
\int_step_variable:nnnNn
```

---

New: 2012-06-04  
Updated: 2018-04-22

---

```
\int_step_variable:nNn {<final value>} {tl var} {<code>}  
\int_step_variable:nnNn {<initial value>} {<final value>} {tl var} {<code>}  
\int_step_variable:nnnNn {<initial value>} {<step>} {<final value>} {tl var}  
{<code>}
```

This function first evaluates the *<initial value>*, *<step>* and *<final value>*, all of which should be integer 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>*.

The functions `\int_step_variable:nNn` and `\int_step_variable:nnNn` both use a fixed *<step>* of 1, and in the case of `\int_step_variable:nNn` the *<initial value>* is also fixed as 1. These functions are provided as simple short-cuts for code clarity.

## 8 Formatting integers

Integers can be placed into the output stream with formatting. These conversions apply to any integer expressions.

---

\int\_to\_arabic:n \*

Updated: 2011-10-22

\int\_to\_arabic:n {<integer expression>}

Places the value of the *<integer expression>* in the input stream as digits, with category code 12 (other).

---

\int\_to\_alpha:n \*

\int\_to\_Alph:n \*

Updated: 2011-09-17

\int\_to\_alpha:n {<integer expression>}

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 \*

Updated: 2011-09-17

\int\_to\_symbols:nnn

{<integer expression>} {<total symbols>}  
{<value to symbol mapping>}

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` ★

New: 2014-02-11

`\int_to_bin:n {<integer expression>}`

Calculates the value of the  $\langle\text{integer expression}\rangle$  and places the binary representation of the result in the input stream.

---

`\int_to_hex:n` ★

`\int_to_Hex:n` ★

New: 2014-02-11

`\int_to_hex:n {<integer expression>}`

Calculates the value of the  $\langle\text{integer expression}\rangle$  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` ★

New: 2014-02-11

`\int_to_oct:n {<integer expression>}`

Calculates the value of the  $\langle\text{integer expression}\rangle$  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` ★

Updated: 2014-02-11

`\int_to_base:nn {<integer expression>} {<base>}`

Calculates the value of the  $\langle\text{integer expression}\rangle$  and converts it into the appropriate representation in the  $\langle\text{base}\rangle$ ; 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  $\langle\text{base}\rangle$  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` ★

Updated: 2011-10-22

`\int_to_roman:n {<integer expression>}`

Places the value of the  $\langle\text{integer expression}\rangle$  in the input stream as Roman numerals, either lower case (`\int_to_roman:n`) or upper case (`\int_to_Roman:n`). The Roman numerals are letters with category code 11 (letter).

## 9 Converting from other formats to integers

---

`\int_from_alpha:n` ★

Updated: 2014-08-25

`\int_from_alpha:n {<letters>}`

Converts the  $\langle\text{letters}\rangle$  into the integer (base 10) representation and leaves this in the input stream. The  $\langle\text{letters}\rangle$  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_Alph:n`.

---

`\int_from_bin:n` ★

New: 2014-02-11

Updated: 2014-08-25

`\int_from_bin:n {<binary number>}`

Converts the  $\langle\text{binary number}\rangle$  into the integer (base 10) representation and leaves this in the input stream. The  $\langle\text{binary number}\rangle$  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** ★

New: 2014-02-11

Updated: 2014-08-25

---

\int\_from\_hex:n {*hexadecimal number*}

Converts the *hexadecimal number* into the integer (base 10) representation and leaves this in 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** ★

New: 2014-02-11

Updated: 2014-08-25

---

\int\_from\_oct:n {*octal number*}

Converts the *octal number* into the integer (base 10) representation and leaves this in 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** ★Updated: 2014-08-25

---

\int\_from\_roman:n {*roman numeral*}

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** ★Updated: 2014-08-25

---

\int\_from\_base:nn {*number*} {*base*}

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.

## 10 Random integers

---

**\int\_rand:nn** ★

New: 2016-12-06

Updated: 2018-04-27

---

\int\_rand:nn {*intexpr*<sub>1</sub>} {*intexpr*<sub>2</sub>}

Evaluates the two *integer expressions* and produces a pseudo-random number between the two (with bounds included). This is not available in older versions of X<sub>Ǝ</sub>T<sub>E</sub>X.

---

**\int\_rand:n** ★New: 2018-05-05

---

\int\_rand:n {*intexpr*}

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>Ǝ</sub>T<sub>E</sub>X.

## 11 Viewing integers

---

**\int\_show:N**\int\_show:N {*integer*}

---

**\int\_show:c**Displays the value of the *integer* on the terminal.

<code>\int_show:n</code>	<code>\int_show:n {&lt;integer expression&gt;}</code>
New: 2011-11-22	Displays the result of evaluating the <i>&lt;integer expression&gt;</i> on the terminal.
Updated: 2015-08-07	
<code>\int_log:N</code>	<code>\int_log:N {integer}</code>
<code>\int_log:c</code>	Writes the value of the <i>integer</i> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	
<code>\int_log:n</code>	<code>\int_log:n {&lt;integer expression&gt;}</code>
New: 2014-08-22	Writes the result of evaluating the <i>&lt;integer expression&gt;</i> in the log file.
Updated: 2015-08-07	

## 12 Constant integers

<code>\c_zero_int</code>	Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.
<code>\c_one_int</code>	
New: 2018-05-07	
<code>\c_max_int</code>	The maximum value that can be stored as an integer.
<code>\c_max_register_int</code>	Maximum number of registers.
<code>\c_max_char_int</code>	Maximum character code completely supported by the engine.

## 13 Scratch integers

<code>\l_tmpa_int</code>	Scratch integer 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.
<code>\l_tmpb_int</code>	
<code>\g_tmpa_int</code>	Scratch integer 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.
<code>\g_tmpb_int</code>	

## 13.1 Direct number expansion

---

`\int_value:w *`    `\int_value:w <integer>`  
New: 2018-03-27    `\int_value:w <integer denotation> <optional space>`

---

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`.

## 14 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: <case<sub>1</sub>>*  
            *\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`.

## Part XI

# 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.

## 1 Setting up flags

---

### `\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 name}`

The *flag*’s height is set to zero. The assignment is local.

---

### `\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 name}`

Displays the *flag*’s height in the terminal.

---

### `\flag_log:n {flag name}`

Writes the *flag*’s height to the log file.

## 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.

# Part XII

## 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.

## 1 Defining a set of conditional functions

---

```
\prg_new_conditional:Npnn
\prg_set_conditional:Npnn
\prg_new_conditional:Nnn
\prg_set_conditional:Nnn
```

---

Updated: 2012-02-06

---

```
\prg_new_conditional:Npnn \<name>:<arg spec> <parameters> {\<conditions>} {\<code>}
\prg_new_conditional:Nnn \<name>:<arg spec> {\<conditions>} {\<code>}
```

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.

---

```
\prg_new_protected_conditional:Npnn
\prg_set_protected_conditional:Npnn
\prg_new_protected_conditional:Nnn
\prg_set_protected_conditional:Nnn
```

---

Updated: 2012-02-06

---

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

These functions create a family of protected conditionals using the same `{<code>}` to perform the test created. The `<code>` 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 `<conditions>`, 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  $\text{\prg\_set\_conditional:Npnn}$ : 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.*  $\text{\cs_new:Nn}$ , *etc.*). Within the  $\langle code \rangle$ , the functions  $\text{\prg_return_true:}$  and  $\text{\prg_return_false:}$  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_t1 #1
    \prg_return_true:
  \else:
    \if_meaning:w \l_tmpa_t1 #2
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  \fi:
}
```

This defines the function  $\text{\foo_if_bar_p:NN}$ ,  $\text{\foo_if_bar:NNTF}$  and  $\text{\foo_if_bar:NNT}$  but not  $\text{\foo_if_bar:NNF}$  (because F is missing from the  $\langle conditions \rangle$  list). The return statements take care of resolving the remaining  $\text{\else:}$  and  $\text{\fi:}$  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.

---

$\text{\prg_new_eq_conditional:NNn}$     $\text{\prg_new_eq_conditional:NNn} \langle name_1 \rangle : \langle arg\ spec_1 \rangle \langle name_2 \rangle : \langle arg\ spec_2 \rangle$   
 $\text{\prg_set_eq_conditional:NNn}$     $\{ \langle conditions \rangle \}$

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

---

```
\prg_return_true: *
\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:Npn`, *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.

---

<code>\prg_generate_conditional_variant:Nnn</code>	<code>\prg_generate_conditional_variant:Nnn \⟨name⟩:{arg spec}</code>
	<code>{⟨variant argument specifiers⟩} {⟨condition specifiers⟩}</code>

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:Npn`, and they should be defined.

## 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 \⟨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_set_false:N  
\bool_set_false:c  
\bool_gset_false:N  
\bool_gset_false:c
```

---

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

---

```
\bool_set_true:N  
\bool_set_true:c  
\bool_gset_true:N  
\bool_gset_true:c
```

---

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

---

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

---

`\bool_set_eq:NN <boolean1> <boolean2>`  
Sets `<boolean1>` to the current value of `<boolean2>`.

---

```
\bool_set:Nn  
\bool_set:cn  
\bool_gset:Nn  
\bool_gset:cn
```

---

Updated: 2017-07-15

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

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

---

```
\bool_if_p:N ★  
\bool_if_p:c ★  
\bool_if:NTF ★  
\bool_if:cTF ★
```

---

Updated: 2017-07-15

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

Tests the current truth of `<boolean>`, and continues expansion based on this result.

---

```
\bool_show:N  
\bool_show:c
```

---

New: 2012-02-09  
Updated: 2015-08-01

`\bool_show:N <boolean>`

Displays the logical truth of the `<boolean>` on the terminal.

---

```
\bool_show:n  
New: 2012-02-09  
Updated: 2017-07-15
```

---

`\bool_show:n {<boolean expression>}`

Displays the logical truth of the `<boolean expression>` on the terminal.

---

```
\bool_log:N  
\bool_log:c  
New: 2014-08-22  
Updated: 2015-08-03
```

---

`\bool_log:N <boolean>`

Writes the logical truth of the `<boolean>` in the log file.

---

```
\bool_log:n  
New: 2014-08-22  
Updated: 2017-07-15
```

---

`\bool_log:n {<boolean expression>}`

Writes the logical truth of the `<boolean expression>` in the log file.

---

`\bool_if_exist_p:N` ★      `\bool_if_exist_p:N <boolean>`  
`\bool_if_exist_p:c` ★      `\bool_if_exist:NTF <boolean> {{true code}} {{false code}}`  
`\bool_if_exist:NTF` ★      Tests whether the `<boolean>` is currently defined. This does not check that the `<boolean>` really is a boolean variable.  
`\bool_if_exist:cTF` ★

New: 2012-03-03

---

`\l_tmpa_bool`  
`\l_tmpb_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`  
`\g_tmpb_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.

### 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 T<sub>E</sub>X. 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:mnTF`, `\bool_lazy_any:nTF`, or `\bool_lazy_or:mnTF`, 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:nTF ★`

Updated: 2017-07-15

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

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.

---

`\bool_lazy_all_p:n ★`  
`\bool_lazy_all:nTF ★`

New: 2015-11-15

Updated: 2017-07-15

`\bool_lazy_all_p:n { {<boolexpr_1>} {<boolexpr_2>} ... {<boolexpr_N>} }`  
`\bool_lazy_all:nTF { {<boolexpr_1>} {<boolexpr_2>} ... {<boolexpr_N>} } {<true code>} {<false code>}`

Implements the “And” operation on the `<boolean expressions>`, hence is `true` if all of them are `true` and `false` if any of them is `false`. Contrarily to the infix operator `&&`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_all:nTF` are evaluated. See also `\bool_lazy_and:nnTF` when there are only two `<boolean expressions>`.

---

`\bool_lazy_and_p:nn ★`  
`\bool_lazy_and:nnTF ★`

New: 2015-11-15  
 Updated: 2017-07-15

`\bool_lazy_and_p:nn {<boolexpr_1>} {<boolexpr_2>}`  
`\bool_lazy_and:nnTF {<boolexpr_1>} {<boolexpr_2>} {<true code>} {<false code>}`

Implements the “And” operation between two boolean expressions, hence is `true` if both are `true`. Contrarily to the infix operator `&&`, the `<boolexpr_2>` is only evaluated if it is needed to determine the result of `\bool_lazy_and:nnTF`. See also `\bool_lazy_all:nTF` when there are more than two `<boolean expressions>`.

---

`\bool_lazy_any_p:n ★`  
`\bool_lazy_any:nTF ★`

New: 2015-11-15  
 Updated: 2017-07-15

`\bool_lazy_any_p:n { {<boolexpr_1>} {<boolexpr_2>} ... {<boolexpr_N>} }`  
`\bool_lazy_any:nTF { {<boolexpr_1>} {<boolexpr_2>} ... {<boolexpr_N>} } {<true code>} {<false code>}`

Implements the “Or” operation on the `<boolean expressions>`, hence is `true` if any of them is `true` and `false` if all of them are `false`. Contrarily to the infix operator `||`, only the `<boolean expressions>` which are needed to determine the result of `\bool_lazy_any:nTF` are evaluated. See also `\bool_lazy_or:nnTF` when there are only two `<boolean expressions>`.

---

\bool\_lazy\_or\_p:nn ★ \bool\_lazy\_or:nnTF ★

New: 2015-11-15

Updated: 2017-07-15

---

\bool\_lazy\_or\_p:nn {*boolexpr<sub>1</sub>*} {*boolexpr<sub>2</sub>*}  
\bool\_lazy\_or:nnTF {*boolexpr<sub>1</sub>*} {*boolexpr<sub>2</sub>*} {*true code*} {*false code*}

Implements the “Or” operation between two boolean expressions, hence is **true** if either one is **true**. Contrarily to the infix operator `||`, the *boolexpr<sub>2</sub>* is only evaluated if it is needed to determine the result of `\bool_lazy_or:nnTF`. See also `\bool_lazy_any:nTF` when there are more than two *boolean expressions*.

---

\bool\_not\_p:n ★ \bool\_not\_p:n {*boolean expression*}

Updated: 2017-07-15

---

Function version of `!(<boolean expression>)` within a boolean expression.

---

\bool\_xor\_p:nn ★ \bool\_xor:nnTF ★

New: 2018-05-09

---

\bool\_xor\_p:nn {*boolexpr<sub>1</sub>*} {*boolexpr<sub>2</sub>*}  
\bool\_xor:nnTF {*boolexpr<sub>1</sub>*} {*boolexpr<sub>2</sub>*} {*true code*} {*false code*}

Implements an “exclusive or” operation between two boolean expressions. There is no infix operation for this logical operation.

## 4 Logical loops

Loops using either boolean expressions or stored boolean values.

---

\bool\_do\_until:Nn ★ \bool\_do\_until:cn ★

Updated: 2017-07-15

---

\bool\_do\_until:Nn {*boolean*} {*code*}

Places the *code* in the input stream for TeX to process, and then checks the logical value of the *boolean*. If it is **false** then the *code* is inserted into the input stream again and the process loops until the *boolean* is **true**.

---

\bool\_do\_while:Nn ★ \bool\_do\_while:cn ★

Updated: 2017-07-15

---

\bool\_do\_while:Nn {*boolean*} {*code*}

Places the *code* in the input stream for TeX to process, and then checks the logical value of the *boolean*. If it is **true** then the *code* is inserted into the input stream again and the process loops until the *boolean* is **false**.

---

\bool\_until\_do:Nn ★ \bool\_until\_do:cn ★

Updated: 2017-07-15

---

\bool\_until\_do:Nn {*boolean*} {*code*}

This function firsts checks the logical value of the *boolean*. If it is **false** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean* is re-evaluated. The process then loops until the *boolean* is **true**.

---

\bool\_while\_do:Nn ★ \bool\_while\_do:cn ★

Updated: 2017-07-15

---

\bool\_while\_do:Nn {*boolean*} {*code*}

This function firsts checks the logical value of the *boolean*. If it is **true** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean* is re-evaluated. The process then loops until the *boolean* is **false**.

---

\bool\_do\_until:nn ★

Updated: 2017-07-15

---

\bool\_do\_until:nn {*boolean expression*} {*code*}

Places the *code* in the input stream for TeX to process, and then checks the logical value of the *boolean expression* as described for `\bool_if:nTF`. If it is **false** then the *code* is inserted into the input stream again and the process loops until the *boolean expression* evaluates to **true**.

---

\bool\_do\_while:nn ★

Updated: 2017-07-15

\bool\_do\_while:nn {*boolean expression*} {*code*}

Places the *code* in the input stream for TeX to process, and then checks the logical value of the *boolean expression* as described for \bool\_if:nTF. If it is **true** then the *code* is inserted into the input stream again and the process loops until the *boolean expression* evaluates to **false**.

---

\bool\_until\_do:nn ★

Updated: 2017-07-15

\bool\_until\_do:nn {*boolean expression*} {*code*}

This function firsts checks the logical value of the *boolean expression* (as described for \bool\_if:nTF). If it is **false** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean expression* is re-evaluated. The process then loops until the *boolean expression* is **true**.

---

\bool\_while\_do:nn ★

Updated: 2017-07-15

\bool\_while\_do:nn {*boolean expression*} {*code*}

This function firsts checks the logical value of the *boolean expression* (as described for \bool\_if:nTF). If it is **true** the *code* is placed in the input stream and expanded. After the completion of the *code* the truth of the *boolean expression* is re-evaluated. The process then loops until the *boolean expression* is **false**.

## 5 Producing multiple copies

---

\prg\_replicate:nn ★

Updated: 2011-07-04

\prg\_replicate:nn {*integer expression*} {*tokens*}

Evaluates the *integer expression* (which should be zero or positive) and creates the resulting number of copies of the *tokens*. The function is both expandable and safe for nesting. It yields its result after two expansion steps.

## 6 Detecting TeX's mode

---

\mode\_if\_horizontal\_p: ★

\mode\_if\_horizontal:TF ★

\mode\_if\_horizontal\_p:

\mode\_if\_horizontal:TF {*true code*} {*false code*}

Detects if TeX is currently in horizontal mode.

---

\mode\_if\_inner\_p: ★

\mode\_if\_inner:TF ★

\mode\_if\_inner\_p:

\mode\_if\_inner:TF {*true code*} {*false code*}

Detects if TeX is currently in inner mode.

---

\mode\_if\_math\_p: ★

\mode\_if\_math:TF ★

Updated: 2011-09-05

\mode\_if\_math\_p:

\mode\_if\_math:TF {*true code*} {*false code*}

Detects if TeX is currently in maths mode.

---

\mode\_if\_vertical\_p: ★

\mode\_if\_vertical:TF ★

\mode\_if\_vertical\_p:

\mode\_if\_vertical:TF {*true code*} {*false code*}

Detects if TeX is currently in vertical mode.

## 7 Primitive conditionals

---

`\if_predicate:w *` `\if_predicate:w <predicate> <true code> \else: <false code> \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 <boolean> <true code> \else: <false code> \fi:`

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

## 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 \<type>_map_break: {<code>}`

New: 2018-03-26

Used to mark the end of a recursion or mapping: the functions `\<type>_map_break:` and `\<type>_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_i:n`.

---

`\prg_map_break:Nn *` `\prg_map_break:Nn \<type>_map_break: {<user code>}`  
...  
`\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.

### 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: *` `\prg_break_point: is used to mark the end of a fast short-term recursion: 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 (see for instance \prop_get:Nn), and inserts the <code> in the input stream.`

## 9 Internal programming functions

---

```
\group_align_safe_begin: *
\group_align_safe_end: *
```

---

Updated: 2011-08-11

```
\group_align_safe_begin:
...
\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.

# Part XIII

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

### 1 The name of the job

---

`\c_sys_jobname_str`

New: 2015-09-19

---

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

**TeXhackers note:** This copies the contents of the primitive `\jobname`. It is a constant that is set by TeX and should not be overwritten by the package.

### 2 Date and time

---

`\c_sys_minute_int`

`\c_sys_hour_int`

`\c_sys_day_int`

`\c_sys_month_int`

`\c_sys_year_int`

New: 2015-09-22

---

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

**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.

### 3 Engine

---

`\sys_if_engine_luatex_p: *`      `\sys_if_engine_pdftex:TF {<true code>} {<false code>}`

Conditionals which allow engine-specific code to be used. The names follow naturally from those of the engine binaries: note that the (u)ptex tests are for  $\varepsilon$ -pTeX and  $\varepsilon$ -upTeX as expl3 requires the  $\varepsilon$ -TeX extensions. Each conditional is true for *exactly one* supported engine. In particular, `\sys_if_engine_ptex_p:` is true for  $\varepsilon$ -pTeX but false for  $\varepsilon$ -upTeX.

New: 2015-09-07

---

`\c_sys_engine_str`

New: 2015-09-19

---

The current engine given as a lower case string: one of `luatex`, `pdftex`, `ptex`, `uptex` or `xetex`.

## 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
```

---

---

```
\c_sys_output_str
```

The current output mode given as a lower case string; one of dvi or pdf.  
New: 2015-09-19

---

# Part XIV

## The `\clist` package

### Comma separated lists

Comma lists 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 `\seq`) 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 comma list data.

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 items so the following gives an empty comma list:

```
\clist_clear_new:N \l_my_clist
\clist_put_right: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.

Almost all operations on comma lists are noticeably slower than those on sequences so converting the data to sequences using `\seq_set_from_clist:Nn` (see `\seq`) may be advisable if speed is important. The exception is that `\clist_if_in:NnTF` and `\clist_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). In addition, comma lists cannot store quarks `\q_mark` or `\q_stop`. The sequence data type should thus certainly be preferred to comma lists to store such items.

## 1 Creating and initialising comma lists

---

```
\clist_new:N <comma list>
```

---

```
\clist_new:c
```

Creates a new `<comma list>` or raises an error if the name is already taken. The declaration is global. The `<comma list>` initially contains no items.

---

```
\clist_const:Nn
\clist_const:(Nx|cn|cx)

```

New: 2014-07-05

---

```
\clist_const:Nn <clist var> {{comma list}}
```

Creates a new constant *<clist var>* or raises an error if the name is already taken. The value of the *<clist var>* is set globally to the *<comma list>*.

---

```
\clist_clear:N
\clist_clear:c
\clist_gclear:N
\clist_gclear:c

```

---

```
\clist_clear:N <comma list>
```

Clears all items from the *<comma list>*.

---

```
\clist_clear_new:N
\clist_clear_new:c
\clist_gclear_new:N
\clist_gclear_new:c

```

---

```
\clist_clear_new:N <comma list>
```

Ensures that the *<comma list>* exists globally by applying `\clist_new:N` if necessary, then applies `\clist_(g)clear:N` to leave the list empty.

---

```
\clist_set_eq:NN
\clist_set_eq:(cN|Nc|cc)
\clist_gset_eq:NN
\clist_gset_eq:(cN|Nc|cc)

```

---

```
\clist_set_eq:NN <comma list1> <comma list2>
```

Sets the content of *<comma list<sub>1</sub>>* equal to that of *<comma list<sub>2</sub>>*.

---

```
\clist_set_from_seq:NN
\clist_set_from_seq:(cN|Nc|cc)
\clist_gset_from_seq:NN
\clist_gset_from_seq:(cN|Nc|cc)

```

---

```
\clist_set_from_seq:NN <comma list> <sequence>
```

New: 2014-07-17

---

Converts the data in the *<sequence>* into a *<comma list>*: the original *<sequence>* is unchanged. Items which contain either spaces or commas are surrounded by braces.

---

```
\clist_concat:NNN
\clist_concat:ccc
\clist_gconcat:NNN
\clist_gconcat:ccc

```

---

```
\clist_concat:NNN <comma list1> <comma list2> <comma list3>
```

Concatenates the content of *<comma list<sub>2</sub>>* and *<comma list<sub>3</sub>>* together and saves the result in *<comma list<sub>1</sub>>*. The items in *<comma list<sub>2</sub>>* are placed at the left side of the new comma list.

---

```
\clist_if_exist_p:N *
\clist_if_exist_p:c *
\clist_if_exist:NTF *
\clist_if_exist:cTF *

```

---

```
\clist_if_exist_p:N <comma list>
```

```
\clist_if_exist:NTF <comma list> {{true code}} {{false code}}
```

Tests whether the *<comma list>* is currently defined. This does not check that the *<comma list>* really is a comma list.

New: 2012-03-03

---

## 2 Adding data to comma lists

---

```
\clist_set:Nn          \clist_set:Nn <comma list> {\langle item_1\rangle,\dots,\langle item_n\rangle}
\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  $\langle \text{comma list} \rangle$  to contain the  $\langle \text{items} \rangle$ , 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  $\langle \text{tokens} \rangle$  as a single  $\langle \text{item} \rangle$  even if the  $\langle \text{tokens} \rangle$  contain commas or spaces, add a set of braces: `\clist_set:Nn <comma list> {\{\langle tokens \rangle\}}`.

---

```
\clist_put_left:Nn      \clist_put_left:Nn <comma list> {\langle item_1\rangle,\dots,\langle item_n\rangle}
\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  $\langle \text{items} \rangle$  to the left of the  $\langle \text{comma list} \rangle$ . 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  $\langle \text{tokens} \rangle$  as a single  $\langle \text{item} \rangle$  even if the  $\langle \text{tokens} \rangle$  contain commas or spaces, add a set of braces: `\clist_put_left:Nn <comma list> {\{\langle tokens \rangle\}}`.

---

```
\clist_put_right:Nn     \clist_put_right:Nn <comma list> {\langle item_1\rangle,\dots,\langle item_n\rangle}
\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  $\langle \text{items} \rangle$  to the right of the  $\langle \text{comma list} \rangle$ . 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  $\langle \text{tokens} \rangle$  as a single  $\langle \text{item} \rangle$  even if the  $\langle \text{tokens} \rangle$  contain commas or spaces, add a set of braces: `\clist_put_right:Nn <comma list> {\{\langle tokens \rangle\}}`.

## 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:nn(TF)`.

**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
\clist_gremove_all:Nn
\clist_gremove_all:cn
```

---

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:nn(TF)`.

**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
\clist_greverse:c
```

---

New: 2014-07-18

Reverses the order of items stored in the *<comma list>*.

---

```
\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 argument expansion.

---

```
\clist_sort:Nn                \clist_sort:Nn <clist var> {\<comparison code>}
\clist_sort:cn
\clist_gsort:Nn
\clist_gsort:cn
```

---

New: 2017-02-06

Sorts the items in the *<clist var>* according to the *<comparison code>*, and assigns the result to *<clist var>*. The details of sorting comparison are described in Section 1.

---

```
\clist_if_empty_p:N *        \clist_if_empty_p:N <comma list>
\clist_if_empty_p:c *
\clist_if_empty:NTF *
\clist_if_empty:cTF *
```

---

Tests if the *<comma list>* is empty (containing no items).

---

```
\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).

## 5 Mapping to comma lists

The functions described in this section apply a specified function to each item of a comma list.

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 ★
\clist_map_function:cN ★
\clist_map_function:nN ★
```

---

Updated: 2012-06-29

```
\clist_map_function:NN {⟨comma list⟩} {⟨function⟩}
```

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.

---

```
\clist_map_inline:Nn {⟨comma list⟩} {⟨inline function⟩}
```

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.

---

```
\clist_map_variable:Nn
\clist_map_variable:cNn
\clist_map_variable:nNn
```

Updated: 2012-06-29

---

```
\clist_map_variable:Nn \commalist \variable {code}
```

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. The *items* are returned from left to right.

---

```
\clist_map_break: *
```

Updated: 2012-06-29

---

```
\clist_map_break:
```

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.

**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.

---

```
\clist_map_break:n *
```

Updated: 2012-06-29

---

```
\clist_map_break:n {code}
```

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 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.

---

```
\clist_count:N *
\clist_count:c *
\clist_count:n *
```

New: 2012-07-13

---

```
\clist_count:N \commalist
```

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 duplicates, *i.e.* every item in a *comma list* is counted.

## 6 Using the content of comma lists directly

---

\clist\_use:Nnnn ★  
\clist\_use:cnnn ★  
New: 2013-05-26

---

```
\clist_use:Nnnn <clist var> {<separator between two>}  
{<separator between more than two>} {<separator between final two>}
```

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:Nnn \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 argument expansion.

---

\clist\_use:Nn ★  
\clist\_use:cn ★  
New: 2013-05-26

---

```
\clist_use:Nn <clist var> {<separator>}
```

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 argument expansion.

## 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:cN  
Updated: 2012-05-14

\clist\_get:NN *comma list* *token list variable*

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. If the *comma list* is empty the *token list variable* is set to the marker value \q\_no\_value.

---

\clist\_get:NNTF  
\clist\_get:cNTF  
New: 2012-05-14

\clist\_get:NNTF *comma list* *token list variable* {*true code*} {*false code*}

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, stores the top item from the *comma list* in the *token list variable* without removing it from the *comma list*. The *token list variable* is assigned locally.

---

\clist\_pop:NN  
\clist\_pop:cN  
Updated: 2011-09-06

\clist\_pop:NN *comma list* *token list variable*

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:cN

\clist\_gpop:NN *comma list* *token list variable*

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:cNTF  
New: 2012-05-14

\clist\_pop:NNTF *comma list* *token list variable* {*true code*} {*false code*}

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:cNTF  
New: 2012-05-14

\clist\_gpop:NNTF *comma list* *token list variable* {*true code*} {*false code*}

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:(NV|No|Nx|cn|cV|co|cx)  
\clist\_gpush:Nn  
\clist\_gpush:(NV|No|Nx|cn|cV|co|cx)

\clist\_push:Nn *comma list* {*items*}

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.

## 8 Using a single item

---

\clist\_item:Nn ★  
\clist\_item:cn ★  
\clist\_item:nn ★  
New: 2014-07-17

---

\clist\_item:Nn *comma list* {*integer expression*}

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 argument expansion.

---

\clist\_rand\_item:N ★  
\clist\_rand\_item:c ★  
\clist\_rand\_item:n ★  
New: 2016-12-06

---

\clist\_rand\_item:N *clist var*  
\clist\_rand\_item:n {*comma list*}

Selects a pseudo-random item of the *comma list*. If the *comma list* has no item, the result is empty.

**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 argument expansion.

## 9 Viewing comma lists

---

\clist\_show:N  
\clist\_show:c  
Updated: 2015-08-03

---

\clist\_show:N *comma list*

Displays the entries in the *comma list* in the terminal.

---

\clist\_show:n  
Updated: 2013-08-03

---

\clist\_show:n {*tokens*}

Displays the entries in the comma list in the terminal.

---

\clist\_log:N  
\clist\_log:c  
New: 2014-08-22  
Updated: 2015-08-03

---

\clist\_log:N *comma list*

Writes the entries in the *comma list* in the log file. See also \clist\_show:N which displays the result in the terminal.

---

\clist\_log:n  
New: 2014-08-22

---

\clist\_log:n {*tokens*}

Writes the entries in the comma list in the log file. See also \clist\_show:n which displays the result in the terminal.

## 10 Constant and scratch comma lists

---

### `\c_empty_clist`

New: 2012-07-02

---

Constant that is always empty.

---

### `\l_tmpa_clist`

---

### `\l_tmpb_clist`

New: 2011-09-06

---

Scratch comma lists 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_clist`

---

### `\g_tmpb_clist`

New: 2011-09-06

---

Scratch comma lists 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.

# Part XV

## 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 T\texEX, 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 T\texEX, 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, T\texEX 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 8.

## 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 {<integer expression>} <function>  
\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** \*

New: 2015-09-09  
Updated: 2019-01-16

---

```
\char_generate:nn {<charcode>} {<catcode>}
```

Generates a character token of the given  $\langle \text{charcode} \rangle$  and  $\langle \text{catcode} \rangle$  (both of which may be integer expressions). The  $\langle \text{catcode} \rangle$  may be one of

- 1 (begin group)
- 2 (end group)
- 3 (math toggle)
- 4 (alignment)
- 6 (parameter)
- 7 (math superscript)
- 8 (math subscript)
- 11 (letter)
- 12 (other)
- 13 (active)

and other values raise an error. The  $\langle \text{charcode} \rangle$  may be any one valid for the engine in use. Active characters cannot be generated in older versions of X<sub>E</sub>T<sub>E</sub>X.

**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).

## 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**Updated: 2015-11-11

---

\char\_set\_catcode:nn {*integer expression*} {*integer expression*}

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**Updated: 2015-08-06

---

\char\_set\_lccode:nn {*integer expression*} {*integer expression*}

Sets up the behaviour of the *character* when found inside `\tl_lower_case: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**Updated: 2015-08-06

---

\char\_set\_uccode:nn {*integer expression*} {*integer expression*}

Sets up the behaviour of the *character* when found inside `\tl_upper_case: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>character</i> with character code given by the <i>integer expression</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>character</i> with character code given by the <i>integer expression</i> on the terminal.
<code>\char_set_mathcode:nn</code> Updated: 2015-08-06	<code>\char_set_mathcode:nn {&lt;intexpr1&gt;} {&lt;intexpr2&gt;}</code>
	This function sets up the math code of <i>character</i> . The <i>character</i> is specified as an <i>integer expression</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>character</i> with character code given by the <i>integer expression</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>character</i> with character code given by the <i>integer expression</i> on the terminal.
<code>\char_set_sfcode:nn</code> Updated: 2015-08-06	<code>\char_set_sfcode:nn {&lt;intexpr1&gt;} {&lt;intexpr2&gt;}</code>
	This function sets up the space factor for the <i>character</i> . The <i>character</i> is specified as an <i>integer expression</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>character</i> with character code given by the <i>integer expression</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>character</i> with character code given by the <i>integer expression</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>active</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>letter</i> (catcode 11) or <i>other</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.

### 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.

### 4 Converting tokens

---

```
\token_to_meaning:N *
\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: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.

## 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 *</code>	<code>\token_if_letter_p:N &lt;token&gt;</code>
<code>\token_if_letter:NTF *</code>	<code>\token_if_letter:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if <code>&lt;token&gt;</code> has the category code of a letter token.	
<code>\token_if_other_p:N *</code>	<code>\token_if_other_p:N &lt;token&gt;</code>
<code>\token_if_other:NTF *</code>	<code>\token_if_other:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if <code>&lt;token&gt;</code> has the category code of an “other” token.	
<code>\token_if_active_p:N *</code>	<code>\token_if_active_p:N &lt;token&gt;</code>
<code>\token_if_active:NTF *</code>	<code>\token_if_active:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if <code>&lt;token&gt;</code> has the category code of an active character.	
<code>\token_if_eq_catcode_p:NN *</code>	<code>\token_if_eq_catcode_p:NN &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt;</code>
<code>\token_if_eq_catcode:NNTF *</code>	<code>\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 <code>&lt;tokens&gt;</code> have the same category code.	
<code>\token_if_eq_charcode_p:NN *</code>	<code>\token_if_eq_charcode_p:NN &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt;</code>
<code>\token_if_eq_charcode:NNTF *</code>	<code>\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 <code>&lt;tokens&gt;</code> have the same character code.	
<code>\token_if_eq_meaning_p:NN *</code>	<code>\token_if_eq_meaning_p:NN &lt;token<sub>1</sub>&gt; &lt;token<sub>2</sub>&gt;</code>
<code>\token_if_eq_meaning:NNTF *</code>	<code>\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 <code>&lt;tokens&gt;</code> have the same meaning when expanded.	
<code>\token_if_macro_p:N *</code>	<code>\token_if_macro_p:N &lt;token&gt;</code>
<code>\token_if_macro:NTF *</code>	<code>\token_if_macro:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Updated: 2011-05-23	Tests if the <code>&lt;token&gt;</code> is a TeX macro.
<code>\token_if_cs_p:N *</code>	<code>\token_if_cs_p:N &lt;token&gt;</code>
<code>\token_if_cs:NTF *</code>	<code>\token_if_cs:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if the <code>&lt;token&gt;</code> is a control sequence.	
<code>\token_if_expandable_p:N *</code>	<code>\token_if_expandable_p:N &lt;token&gt;</code>
<code>\token_if_expandable:NTF *</code>	<code>\token_if_expandable:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Tests if the <code>&lt;token&gt;</code> is expandable. This test returns <code>false</code> for an undefined token.	
<code>\token_if_long_macro_p:N *</code>	<code>\token_if_long_macro_p:N &lt;token&gt;</code>
<code>\token_if_long_macro:NTF *</code>	<code>\token_if_long_macro:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Updated: 2012-01-20	Tests if the <code>&lt;token&gt;</code> is a long macro.
<code>\token_if_protected_macro_p:N *</code>	<code>\token_if_protected_macro_p:N &lt;token&gt;</code>
<code>\token_if_protected_macro:NTF *</code>	<code>\token_if_protected_macro:NTF &lt;token&gt; {\{true code\}} {\{false code\}}</code>
Updated: 2012-01-20	Tests if the <code>&lt;token&gt;</code> is a protected macro: for a macro which is both protected and long this returns <code>false</code> .

---

```
\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  $\langle token \rangle$  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  $\langle token \rangle$  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  $\langle token \rangle$  is defined to be a mathchardef.

---

```
\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  $\langle token \rangle$  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  $\langle token \rangle$  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  $\langle token \rangle$  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  $\langle token \rangle$  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: 2011-05-23

Tests if the *<token>* is an engine primitive.

## 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.

---

```
\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 T<sub>E</sub>X 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\_ignore\_spaces:NTF

Updated: 2012-12-20

\peek\_catcode\_ignore\_spaces:NTF *<test token>* {{*true code*} } {{*false code*} }

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test \token\_if\_eq\_catcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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

Updated: 2012-12-20

\peek\_catcode\_remove:NTF *<test token>* {{*true code*} } {{*false code*} }

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\_catcode\_remove\_ignore\_spaces:NTF

Updated: 2012-12-20

\peek\_catcode\_remove\_ignore\_spaces:NTF *<test token>* {{*true code*} } {{*false code*} }

Tests if the next non-space *<token>* in the input stream has the same category code as the *<test token>* (as defined by the test \token\_if\_eq\_catcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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

Updated: 2012-12-20

\peek\_charcode:NTF *<test token>* {{*true code*} } {{*false code*} }

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\_ignore\_spaces:NTF

Updated: 2012-12-20

\peek\_charcode\_ignore\_spaces:NTF *<test token>* {{*true code*} } {{*false code*} }

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token\_if\_eq\_charcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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

Updated: 2012-12-20

\peek\_charcode\_remove:NTF *<test token>* {{*true code*} } {{*false code*} }

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\_charcode\_remove\_ignore\_spaces:NTF \peek\_charcode\_remove\_ignore\_spaces:NTF {test token}  
  {{true code}} {{false code}}  
Updated: 2012-12-20

Tests if the next non-space *<token>* in the input stream has the same character code as the *<test token>* (as defined by the test \token\_if\_eq\_charcode:NNTF). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed 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\_ignore\_spaces:NTF    \peek\_meaning\_ignore\_spaces:NTF {test token} {\{true code\}} {\{false code\}}

Tests if the next non-space  $\langle token \rangle$  in the input stream has the same meaning as the  $\langle test\ token \rangle$  (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the  $\langle token \rangle$  is left in the input stream after the  $\langle true\ code \rangle$  or  $\langle false\ code \rangle$  (as appropriate to the result of the test).

<code>\peek_meaning_remove:NTF</code>	<code>\peek_meaning_remove:NTF &lt;test token&gt; {&lt;true code&gt;} {&lt;false code&gt;}</code>
Updated: 2011-07-02	Tests if the next <i>&lt;token&gt;</i> in the input stream has the same meaning as

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\_meaning\_remove\_ignore\_spaces:NTF \peek\_meaning\_remove\_ignore\_spaces:NTF {test token}  
  {<true code>} {<false code>}

Tests if the next non-space  $\langle token \rangle$  in the input stream has the same meaning as the  $\langle test\ token \rangle$  (as defined by the test `\token_if_eq_meaning:NNTF`). Explicit and implicit space tokens (with character code 32 and category code 10) are ignored and removed by the test and the  $\langle token \rangle$  is removed from the input stream if the test is true. The function then places either the  $\langle true\ code \rangle$  or  $\langle false\ code \rangle$  in the input stream (as appropriate to the result of the test).

## 7 Decomposing a macro definition

These functions decompose TeX macros into their constituent parts: if the  $\langle token \rangle$  passed is not a macro then no decomposition can occur. In the latter case, all three functions leave `\scan_stop:` in the input stream.

---

**\token\_get\_arg\_spec:N** \*

```
\token_get_arg_spec:N <token>
```

If the *<token>* is a macro, this function leaves the primitive TeX argument specification in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token \next defined by

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

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 arg spec. contains the string ->, then the spec function produces incorrect results.

---

**\token\_get\_replacement\_spec:N** \*

```
\token_get_replacement_spec:N <token>
```

If the *<token>* is a macro, this function leaves the replacement text in input stream as a string of tokens of category code 12 (with spaces having category code 10). Thus for example for a token \next defined by

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

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 arg spec. contains the string ->, then the spec function produces incorrect results.

---

**\token\_get\_prefix\_spec:N** \*

```
\token_get_prefix_spec:N <token>
```

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

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

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

## 8 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, \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).<sup>4</sup>

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`.

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 LATEX3 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`).

---

<sup>4</sup>In LuaTeX, there is also the case of “bytes”, which behave as character tokens of category code 12 (other) and character code between 1114112 and 1114366. They are used to output individual bytes to files, rather than UTF-8.

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 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

*<properties> macro:<parameters>-><replacement>*

where *<properties>* is among **\protected****\long****\outer**, *<parameters>* describes parameters that the macro expects, such as #1#2#3, and *<replacement>* describes how the parameters are manipulated, such as #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.

# Part XVI

## The **I3prop** package

### Property lists

LATEX3 implements a “property list” data type, which contain an unordered list of entries each of which consists of a  $\langle key \rangle$  and an associated  $\langle value \rangle$ . The  $\langle key \rangle$  and  $\langle value \rangle$  may both be any  $\langle balanced\ text \rangle$ . 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  $\langle key \rangle$ : if an entry is added to a property list which already contains the  $\langle key \rangle$  then the new entry overwrites the existing one. The  $\langle keys \rangle$  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 `keys` module.

## 1 Creating and initialising property lists

---

`\prop_new:N`      `\prop_new:c`

Creates a new  $\langle property\ list \rangle$  or raises an error if the name is already taken. The declaration is global. The  $\langle property\ list \rangle$  initially contains no entries.

---

`\prop_clear:N`      `\prop_clear:c`  
`\prop_gclear:N`      `\prop_gclear:c`

Clears all entries from the  $\langle property\ list \rangle$ .

---

`\prop_clear_new:N`      `\prop_clear_new:c`  
`\prop_gclear_new:N`      `\prop_gclear_new:c`

Ensures that the  $\langle property\ list \rangle$  exists globally by applying `\prop_new:N` if necessary, then applies `\prop_(g)clear:N` to leave the list empty.

---

`\prop_set_eq:NN`      `\prop_set_eq:(cN|Nc|cc)`  
`\prop_gset_eq:NN`      `\prop_gset_eq:(cN|Nc|cc)`

`\prop_set_eq:NN`  $\langle property\ list_1 \rangle$   $\langle property\ list_2 \rangle$

Sets the content of  $\langle property\ list_1 \rangle$  equal to that of  $\langle property\ list_2 \rangle$ .

## 2 Adding entries to property lists

---

```
\prop_put:Nnn
\prop_put:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)
\prop_gput:Nnn
\prop_gput:(NnV|Nno|Nnx|NVn|NVV|Non|Noo|cnn|cnV|cno|cnx|cVn|cVV|con|coo)
```

---

Updated: 2012-07-09

Adds an entry to the *property list* which may be accessed using the *key* and which has *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. If the *key* is already present in the *property list*, the existing entry is overwritten by the new *value*.

---

```
\prop_put_if_new:Nnn \prop_put_if_new:cnn
\prop_gput_if_new:Nnn \prop_gput_if_new:cnn
```

---

If the *key* is present in the *property list* then no action is taken. If the *key* is not present in the *property list* then a new entry is added. 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.

## 3 Recovering values from property lists

---

```
\prop_get:NnN \prop_get:NnN \prop_get:(NVN|NoN|cnN|cVN|coN)
```

---

Updated: 2011-08-28

Recover 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 T<sub>E</sub>X group. See also `\prop_get:NnNTF`.

---

```
\prop_pop:NnN \prop_pop:(NoN|cnN|coN)
```

---

Updated: 2011-08-18

```
\prop_pop:NnN \prop_pop:(NoN|cnN|coN)
```

Recover 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)
```

---

Updated: 2011-08-18

```
\prop_gpop:NnN \prop_gpop:(NoN|cnN|coN)
```

Recover 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_get:NnNTF`.

---

```
\prop_item:Nn ★  
\prop_item:cn ★  
New: 2014-07-17
```

---

```
\prop_item:Nn <property list> {<key>}
```

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 argument expansion.

## 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.

## 5 Property list conditionals

---

```
\prop_if_exist_p:N ★  
\prop_if_exist_p:c ★  
\prop_if_exist:NTF ★  
\prop_if_exist:cTF ★  
New: 2012-03-03
```

---

```
\prop_if_exist_p:N <property list>  
\prop_if_exist:NTF <property list> {<true code>} {<false code>}
```

Tests whether the *<property list>* is currently defined. This does not check that the *<property list>* really is a property list variable.

---

```
\prop_if_empty_p:N ★  
\prop_if_empty_p:c ★  
\prop_if_empty:NTF ★  
\prop_if_empty:cTF ★
```

---

```
\prop_if_empty_p:N <property list>  
\prop_if_empty:NTF <property list> {<true code>} {<false code>}
```

Tests if the *<property list>* is empty (containing no entries).

---

```
\prop_if_in_p:Nn ★  
\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

```
          \prop_if_in:NnTF <property list> {<key>} {<true code>} {<false code>}
```

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`.

## 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:(NVN|NoN|cnN|cVN|coN)TF

---

Updated: 2012-05-19

\prop\_get:NnNTF *property list* {<key>} {<token list variable>} {<true code>} {<false code>}

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:cnNTF

---

New: 2011-08-18  
Updated: 2012-05-19

\prop\_pop:NnNTF *property list* {<key>} {<token list variable>} {<true code>} {<false code>}

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>*, 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:cnNTF

---

New: 2011-08-18  
Updated: 2012-05-19

\prop\_gpop:NnNTF *property list* {<key>} {<token list variable>} {<true code>} {<false code>}

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>*, 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.

## 7 Mapping to property lists

---

\prop\_map\_function:NN ☆  
\prop\_map\_function:cN ☆

---

Updated: 2013-01-08

\prop\_map\_function:NN *property list* *function*

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.

---

\prop\_map\_inline:Nn  
\prop\_map\_inline:cN

---

Updated: 2013-01-08

\prop\_map\_inline:Nn *property list* {<inline function>}

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\_break:** ☆Updated: 2012-06-29

---

**\prop\_map\_break:**

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** ☆Updated: 2012-06-29

---

**\prop\_map\_break:n {<code>}**

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.

## 8 Viewing property lists

---

**\prop\_show:N****\prop\_show:c**Updated: 2015-08-01

---

**\prop\_show:N {<property list>}**

Displays the entries in the *<property list>* in the terminal.

---

`\prop_log:N`  
`\prop_log:c`

New: 2014-08-12  
Updated: 2015-08-01

---

`\prop_log:N` *property list*)  
Writes the entries in the *property list* in the log file.

---

`\l_tmpa_prop`  
`\l_tmpb_prop`

New: 2012-06-23

---

Scratch property lists 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_prop`  
`\g_tmpb_prop`

New: 2012-06-23

---

Scratch property lists 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.

## 10 Constants

---

`\c_empty_prop`

---

A permanently-empty property list used for internal comparisons.

# Part XVII

## 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.

## 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:nnnn { mymodule } { submodule / message } ...
```

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

---

```
\msg_new:nnnn
\msg_new:nnn
```

---

Updated: 2011-08-16

```
\msg_new:nnnn {\<module>} {\<message>} {\<text>} {\<more text>}
```

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:nnn
\msg_gset:nnnn
\msg_gset:nnn
```

---

```
\msg_set:nnnn {\<module>} {\<message>} {\<text>} {\<more text>}
```

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.

## 2 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\_module\_name:n** \*

New: 2018-10-10

\msg\_module\_name:n {*module*}

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** \*

New: 2018-10-10

\msg\_module\_type:n {*module*}

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.

---

**\msg\_see\_documentation\_text:n** \*

Updated: 2018-09-30

\msg\_see\_documentation\_text:n {*module*}

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* may be altered by use of **\g\_msg\_module\_documentation\_prop**

---

**\g\_msg\_module\_name\_prop**

New: 2018-10-10

Provides a mapping between the module name used for messages, and that for documentation. For example, L<sup>A</sup>T<sub>E</sub>X3 core messages are stored in the reserved L<sup>A</sup>T<sub>E</sub>X tree, but are printed as L<sup>A</sup>T<sub>E</sub>X3.

---

**\g\_msg\_module\_type\_prop**

New: 2018-10-10

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.

### 3 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.

---

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

---

Updated: 2012-08-11

```
\msg_fatal:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a fatal error the TeX run halts.

---

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

---

Updated: 2012-08-11

```
\msg_critical:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. After issuing a critical error, TeX stops reading the current input file. This may halt the TeX run (if the current file is the main file) or may abort reading a sub-file.

**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.

---

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

---

Updated: 2012-08-11

```
\msg_error:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* error *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The error interrupts processing and issues the text at the terminal. After user input, the run continues.

---

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

---

Updated: 2012-08-11

```
\msg_warning:nnxxxx {\module} {\message} {\arg one} {\arg two} {\arg three}
{\arg four}
```

Issues *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text is added to the log file and the terminal, but the TeX run is not interrupted.

---

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

---

Updated: 2012-08-11

```
\msg_info:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Issues *module* information *message*, passing *arg one* to *arg four* to the text-creating functions. The information text is added to the log file.

---

```
\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_log:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Issues *module* information *message*, passing *arg one* to *arg four* to the text-creating functions. The information text is added to the log file: the output is briefer than \msg\_info:nnnnnn.

---

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

---

Updated: 2012-08-11

```
\msg_none:nnnnnnn {\module} {\message} {\arg one} {\arg two} {\arg three} {\arg four}
```

Does nothing: used as a message class to prevent any output at all (see the discussion of message redirection).

## 4 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
```

Updated: 2012-04-27

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

Changes the behaviour of messages of *<class one>* so that they are processed using the code for those of *<class two>*.

---

```
\msg_redirect_module:nnn
```

Updated: 2012-04-27

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

Redirects message of *<class one>* for *<module>* to act as though they were from *<class two>*. Messages of *<class one>* from sources other than *<module>* 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 *<module>* could be turned off with:

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

---

```
\msg_redirect_name:nnn
```

Updated: 2012-04-27

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

Redirects a specific *<message>* from a specific *<module>* to act as a member of *<class>* 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 }
```

# Part XVIII

## 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  $\text{\TeX}$  attempts to locate them using both the operating system path and entries in the  $\text{\TeX}$  file database (most  $\text{\TeX}$  systems use such a database). Thus the “current path” for  $\text{\TeX}$  is somewhat broader than that for other programs.

For functions which expect a  $\langle\text{file name}\rangle$  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. File names are quoted using " tokens if they contain spaces: as a result, " tokens are *not* permitted in file names.

## 1 Input–output stream management

As  $\text{\TeX}$  engines have a limited number of input and output streams, direct use of the streams by the programmer is not supported in  $\text{\LaTeX}3$ . 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:c`  
`\iow_new:N`  
`\iow_new:c`

---

New: 2011-09-26  
Updated: 2011-12-27

`\ior_new:N`  $\langle\text{stream}\rangle$   
`\iow_new:N`  $\langle\text{stream}\rangle$

Globally reserves the name of the  $\langle\text{stream}\rangle$ , either for reading or for writing as appropriate. The  $\langle\text{stream}\rangle$  is not opened until the appropriate `\..._open:Nn` function is used. Attempting to use a  $\langle\text{stream}\rangle$  which has not been opened is an error, and the  $\langle\text{stream}\rangle$  will behave as the corresponding `\c_term_...`.

---

`\ior_open:Nn`  
`\ior_open:cn`

---

Updated: 2012-02-10

`\ior_open:Nn`  $\langle\text{stream}\rangle$   $\{\langle\text{file name}\rangle\}$

Opens  $\langle\text{file name}\rangle$  for reading using  $\langle\text{stream}\rangle$  as the control sequence for file access. If the  $\langle\text{stream}\rangle$  was already open it is closed before the new operation begins. The  $\langle\text{stream}\rangle$  is available for access immediately and will remain allocated to  $\langle\text{file name}\rangle$  until a `\ior_close:N` instruction is given or the  $\text{\TeX}$  run ends. If the file is not found, an error is raised.

---

**\ior\_open:NnTF**

**\ior\_open:cNTF**

New: 2013-01-12

---

`\ior_open:NnTF <stream> {\<file name>} {\<true code>} {\<false code>}`

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.

---

**\iow\_open:Nn**

**\iow\_open:cn**

Updated: 2012-02-09

---

`\iow_open:Nn <stream> {\<file name>}`

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).

---

**\ior\_close:N**

**\ior\_close:c**

**\iow\_close:N**

**\iow\_close:c**

Updated: 2012-07-31

---

`\ior_close:N <stream>`

`\iow_close:N <stream>`

Closes the *<stream>*. Streams should always be closed when they are finished with as this ensures that they remain available to other programmers.

---

**\ior\_show\_list:**

**\ior\_log\_list:**

**\iow\_show\_list:**

**\iow\_log\_list:**

New: 2017-06-27

---

`\ior_show_list:`

`\ior_log_list:`

`\iow_show_list:`

`\iow_log_list:`

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.

## 1.1 Reading from files

---

`\ior_get:NN` `\ior_get:NN <stream> <token list variable>`

New: 2012-06-24

Function that reads one or more lines (until an equal number of left and right braces are found) from the input `<stream>` and stores the result locally in the `<token list>` variable. If the `<stream>` is not open, input is requested from the terminal. 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_t1  
\tl_set:Nn \l_tmpb_t1 { \par }  
\tl_if_eq:NNF \l_tmpa_t1 \l_tmpb_t1  
...
```

Also notice that if multiple lines are read to match braces then the resulting token list can contain `\par` tokens.

**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>`

New: 2016-12-04

Function that reads one line from the input `<stream>` and stores the result locally in the `<token list>` variable. If the `<stream>` is not open, input is requested from the terminal. 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). Multiple whitespace characters are retained by this process. It always only reads one line and any blank lines in the input result in the `<token list variable>` being empty. Unlike `\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.

**TeXhackers note:** This protected macro is a wrapper around the  $\varepsilon$ -TeX primitive `\readline`. Regardless of settings, 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 `\ior_str_get:NN`.

---

**\ior\_map\_inline:Nn**New: 2012-02-11

---

\ior\_map\_inline:Nn *stream* {*inline function*}

Applies the *inline function* to each set of *lines* obtained by calling \ior\_get:NN until reaching the end of the file. T<sub>E</sub>X ignores any trailing new-line marker from the file it reads. The *inline function* should consist of code which receives the *line* as #1.

---

**\ior\_str\_map\_inline:Nn**New: 2012-02-11

---

\ior\_str\_map\_inline:Nn *stream* {*inline function*}

Applies the *inline function* to every *line* in the *stream*. 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 *inline function* should consist of code which receives the *line* as #1. Note that T<sub>E</sub>X removes trailing space and tab characters (character codes 32 and 9) from every line upon input. T<sub>E</sub>X also ignores any trailing new-line marker from the file it reads.

---

**\ior\_map\_break:**New: 2012-06-29

---

\ior\_map\_break:

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 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.

---

`\ior_map_break:n`

New: 2012-06-29

---

`\ior_map_break:n {<code>}`

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 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.

---

`\ior_if_eof_p:N *`  
`\ior_if_eof:NTF *`

Updated: 2012-02-10

---

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

Tests if the end of a `<stream>` has been reached during a reading operation. The test also returns a `true` value if the `<stream>` is not open.

## 2 Writing to files

---

`\iow_now:Nn`

`\iow_now:(Nx|cn|cx)`

Updated: 2012-06-05

---

`\iow_now:Nn <stream> {<tokens>}`

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:x`

`\iow_log:n {<tokens>}`

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:x`

`\iow_term:n {<tokens>}`

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:(Nx|cn|cx)`

---

`\iow_shipout:Nn <stream> {\langle tokens \rangle}`

This function writes  $\langle tokens \rangle$  to the specified  $\langle stream \rangle$  when the current page is finalised (*i.e.* at shipout). The x-type variants expand the  $\langle tokens \rangle$  at the point where the function is used but *not* when the resulting tokens are written to the  $\langle stream \rangle$  (*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:(Nx|cn|cx)`

---

Updated: 2012-09-08

---

`\iow_shipout_x:Nn <stream> {\langle tokens \rangle}`

This function writes  $\langle tokens \rangle$  to the specified  $\langle stream \rangle$  when the current page is finalised (*i.e.* at shipout). The  $\langle tokens \rangle$  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 \langle char \rangle`

Inserts  $\langle char \rangle$  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  $\langle tokens \rangle$  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.

## 2.1 Wrapping lines in output

---

\iow\_wrap:nnnN \iow\_wrap:nxnN    \iow\_wrap:nnnN {\text} {\run-on text} {\set up} {function}

---

New: 2012-06-28  
Updated: 2017-12-04

---

This function wraps the  $\langle text \rangle$  to a fixed number of characters per line. At the start of each line which is wrapped, the  $\langle run-on text \rangle$  is inserted. The line character count targeted is the value of `\l_iow_line_count_int` minus the number of characters in the  $\langle run-on text \rangle$  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  $\langle text \rangle$  and  $\langle run-on text \rangle$  are exhaustively expanded by the function, with the following substitutions:

- `\backslash\backslash` 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  $\langle text \rangle$  (not the  $\langle run-on text \rangle$ ).

Additional functions may be added to the wrapping by using the  $\langle set up \rangle$ , which is executed before the wrapping takes place: this may include overriding the substitutions listed.

Any expandable material in the  $\langle text \rangle$  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  $\langle function \rangle$ , which is typically a wrapper around a write operation. The output of `\iow_wrap:nnnN` (*i.e.* the argument passed to the  $\langle function \rangle$ ) 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:nnnN` carries out an x-type expansion on the  $\langle text \rangle$  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  $\langle text \rangle$ .

---

\iow\_indent:n    \iow\_indent:n {\text}

---

New: 2011-09-21

---

In the first argument of `\iow_wrap:nnnN` (for instance in messages), indents  $\langle text \rangle$  by four spaces. This function does not cause a line break, and only affects lines which start within the scope of the  $\langle text \rangle$ . In case the indented  $\langle text \rangle$  should appear on separate lines from the surrounding text, use `\backslash\backslash` 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:nnnN` function. This value depends on the TeX system in use: the standard value is 78, which is typically correct for unmodified TeXlive and MiKTeX systems.

## 2.2 Constant input–output streams, and variables

---

`\c_term_iow`

Constant input stream for reading from the terminal. Reading from this stream using `\ior_get:NN` or similar results in a prompt from `TeX` of the form

`<tl>=`

---

`\g_tmpa_iow`  
`\g_tmpb_iow`

New: 2017-12-11

---

Scratch input stream for global use. 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.

---

`\c_log_iow`  
`\c_term_iow`

---

`\g_tmpa_iow`  
`\g_tmpb_iow`

New: 2017-12-11

---

Constant output streams for writing to the log and to the terminal (plus the log), respectively.

Scratch output stream for global use. 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.

## 2.3 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 `TeX` primitive `\ifeof`.

## 3 File operation functions

---

`\g_file_curr_dir_str`  
`\g_file_curr_name_str`  
`\g_file_curr_ext_str`

New: 2017-06-21

---

Contain the directory, name and extension of the current file. The directory is empty if the file was loaded without an explicit path (*i.e.* if it is in the `TeX` search path), and does *not* end in `/` other than the case that it is exactly equal to the root directory. The `<name>` 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 `TeX` 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**New: 2017-06-18

---

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.

**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**Updated: 2012-02-10

---

```
\file_if_exist:nTF {\<file name>} {\<true code>} {\<false code>}
```

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:nnNTF**New: 2019-01-16

---

```
\file_get:nnN {\<filename>} {\<setup>} {\<tl>}
\file_get:nnNTF {\<filename>} {\<setup>} {\<tl>} {\<true code>} {\<false code>}
```

Defines `<tl>` to the contents of `<filename>`. Category codes may need to be set appropriately via the `<setup>` argument. While `\file_get:nnN` produces an error if the file is not found, `\file_get:nnNTF` runs the `<true code>` (after the assignment to `<tl>`) if the file is found and `<false code>` otherwise.

---

**\file\_get\_full\_name:nN**

---

**\file\_get\_full\_name:VN**Updated: 2017-06-26

---

```
\file_get_full_name:nN {\<file name>} {\<str var>}
```

Searches for `<file name>` in the path as detailed for `\file_if_exist:nTF`, and if found sets the `<str 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. If the file is not found then the `<str var>` is empty.

---

```
\file_parse_full_name:nNNN {\<full name>} {\<dir>} {\<name>} {\<ext>}
```

Parses the `<full name>` and splits it into three parts, each of which is returned by setting the appropriate local string variable:

- 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 /.

This function does not expand the `<full name>` before turning it to a string. It assumes that the `<full name>` either contains no quote ("") characters or is surrounded by a pair of quotes.

---

**\file\_input:n**      \file\_input:n {*file name*}

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<sup>A</sup>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\_show\_list:**      \file\_show\_list:  
**\file\_log\_list:**      \file\_log\_list:

These functions list all files loaded by L<sup>A</sup>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.

# Part XIX

## The **I3skip** 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.

## 1 Creating and initialising `dim` variables

---

`\dim_new:N` `\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:cn`

New: 2012-03-05

`\dim_const:<dimension> {<dimension expression>}`

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>`.

---

`\dim_zero:N`  
`\dim_zero:c`  
`\dim_gzero:N`  
`\dim_gzero:c`

`\dim_zero:<dimension>`

Sets `<dimension>` to 0 pt.

---

`\dim_zero_new:N`

`\dim_zero_new:c`  
`\dim_gzero_new:N`  
`\dim_gzero_new:c`

New: 2012-01-07

`\dim_zero_new:<dimension>`

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_if_exist_p:N *`  
`\dim_if_exist_p:c *`

`\dim_if_exist:NTF` `\dim_if_exist:cTF`

Tests whether the `<dimension>` is currently defined. This does not check that the `<dimension>` really is a dimension variable.

New: 2012-03-03

## 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`      Sets `<dimension>` to the value of `<dimension expression>`, which must evaluate to a length with units.  
`\dim_gset:Nn`  
`\dim_gset:cn`

Updated: 2011-10-22

---

`\dim_set_eq:NN`  
`\dim_set_eq:(cN|Nc|cc)`  
`\dim_gset_eq:NN`  
`\dim_gset_eq:(cN|Nc|cc)`

---

`\dim_set_eq:NN <dimension1> <dimension2>`  
Sets the content of `<dimension1>` equal to that of `<dimension2>`.

---

`\dim_sub:Nn`      `\dim_sub:Nn <dimension> {<dimension expression>}`  
`\dim_sub:cn`      Subtracts the result of the `<dimension expression>` from the current content of the `<dimension>`.  
`\dim_gsub:Nn`  
`\dim_gsub:cn`

Updated: 2011-10-22

---

## 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 {<dimexpr1>} {<dimexpr2>}`  
`\dim_min:nn`      `\dim_min:nn {<dimexpr1>} {<dimexpr2>}`

New: 2012-09-09  
Updated: 2012-09-26

---

Evaluates the two `<dimension expressions>` and leaves either the maximum or minimum value in the input stream as appropriate, as a `<dimension denotation>`.

---

\dim\_ratio:nn ★

Updated: 2011-10-22

---

\dim\_ratio:nn {*dimexpr<sub>1</sub>*} {*dimexpr<sub>2</sub>*}

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.

## 4 Dimension expression conditionals

---

\dim\_compare\_p:nNn ★  
\dim\_compare:nNnTF ★

---

\dim\_compare\_p:nNn {*dimexpr<sub>1</sub>*} {*relation*} {*dimexpr<sub>2</sub>*}
\dim\_compare:nNnTF
{*dimexpr<sub>1</sub>*} {*relation*} {*dimexpr<sub>2</sub>*}
{*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	<

---

```
\dim_compare_p:n ★ \dim_compare:nTF ★
\dim_compare_p:n
{
  ⟨dimexpr1⟩ ⟨relation1⟩
  ...
  ⟨dimexprN⟩ ⟨relationN⟩
  ⟨dimexprN+1⟩
}
\dim_compare:nTF
{
  ⟨dimexpr1⟩ ⟨relation1⟩
  ...
  ⟨dimexprN⟩ ⟨relationN⟩
  ⟨dimexprN+1⟩
}
{⟨true code⟩} {⟨false code⟩}
```

---

Updated: 2013-01-13

---

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	$\neq$

---

\dim\_case:nn ★ \dim\_case:nnTF ★  
New: 2013-07-24

---

```
\dim_case:nnTF {\test dimension expression}
{
  {⟨dimexpr case1⟩} {⟨code case1⟩}
  {⟨dimexpr case2⟩} {⟨code case2⟩}
  ...
  {⟨dimexpr casen⟩} {⟨code casen⟩}
}
{⟨true code⟩}
{⟨false code⟩}
```

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.

## 5 Dimension expression loops

---

\dim\_do\_until:nNnn ★ \dim\_do\_until:nNnn {\⟨dimexpr<sub>1</sub>⟩} ⟨relation⟩ {\⟨dimexpr<sub>2</sub>⟩} {⟨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 {\⟨dimexpr<sub>1</sub>⟩} ⟨relation⟩ {\⟨dimexpr<sub>2</sub>⟩} {⟨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 {\⟨dimexpr<sub>1</sub>⟩} ⟨relation⟩ {\⟨dimexpr<sub>2</sub>⟩} {⟨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 <code>true</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>false</code> .
<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 <code>false</code> then the <i>code</i> is inserted into the input stream again and a loop occurs until the <i>relation</i> is <code>true</code> .
<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 <code>true</code> then the <i>code</i> is inserted into the input stream again and a loop occurs until the <i>relation</i> is <code>false</code> .
<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 <code>false</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
<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 <code>true</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>false</code> .

## 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  
New: 2018-02-18

---

\dim\_step\_variable:nnnNn  
  {<initial value>} {<step>} {<final value>} {<tl var>} {<code>}

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>*.

## 7 Using `dim` expressions and variables

---

\dim\_eval:n ★  
Updated: 2011-10-22

---

\dim\_eval:n {<dimension expression>}

Evaluates the *<dimension expression>*, expanding any dimensions and token list variables within the *<expression>* to their content (without requiring `\dim_use:N/\t1_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\_use:N ★  
\dim\_use:c ★

---

\dim\_use:N {<dimension>}

Recovering 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 ★  
New: 2014-07-15

---

\dim\_to\_decimal:n {<dimexpr>}

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** \*

New: 2014-07-15

\dim\_to\_decimal\_in\_bp:n {*dimexpr*}

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** \*

New: 2015-05-18

\dim\_to\_decimal\_in\_sp:n {*dimexpr*}

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** \*

New: 2014-07-15

\dim\_to\_decimal\_in\_unit:nn {*dimexpr*<sub>1</sub>} {*dimexpr*<sub>2</sub>}

Evaluates the *dimension expressions*, and leaves the value of *dimexpr*<sub>1</sub>, expressed in a unit given by *dimexpr*<sub>2</sub>, 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** \*

New: 2012-05-08

\dim\_to\_fp:n {*dimexpr*}

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.

## 8 Viewing dim variables

---

**\dim\_show:N****\dim\_show:c**\dim\_show:N *dimension*

Displays the value of the *dimension* on the terminal.

<code>\dim_show:n</code>	<code>\dim_show:n {dimension expression}</code>
New: 2011-11-22	Displays the result of evaluating the <i>dimension expression</i> on the terminal.
Updated: 2015-08-07	
<code>\dim_log:N</code>	<code>\dim_log:N {dimension}</code>
<code>\dim_log:c</code>	Writes the value of the <i>dimension</i> in the log file.
New: 2014-08-22	
Updated: 2015-08-03	
<code>\dim_log:n</code>	<code>\dim_log:n {dimension expression}</code>
New: 2014-08-22	Writes the result of evaluating the <i>dimension expression</i> in the log file.
Updated: 2015-08-07	

## 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.

## 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.

## 11 Creating and initialising skip variables

`\skip_new:N` `\skip_new:c` `\skip_new:N {skip}` 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.

`\skip_const:Nn` `\skip_const:cn` `\skip_const:Nn {skip} {skip expression}` Creates a new constant *skip* or raises an error if the name is already taken. The value of the *skip* is set globally to the *skip expression*.

---

```
\skip_zero:N \skip_zero:c
```

```
\skip_gzero:N \skip_gzero:c
```

---

`\skip_zero:N <skip>`

Sets `<skip>` to 0 pt.

---

```
\skip_zero_new:N \skip_zero_new:c
```

```
\skip_gzero_new:N \skip_gzero_new:c
```

---

New: 2012-01-07

---

`\skip_zero_new:N <skip>`

Ensures that the `<skip>` exists globally by applying `\skip_new:N` if necessary, then applies `\skip_(g)zero:N` to leave the `<skip>` set to zero.

---

```
\skip_if_exist_p:N * \skip_if_exist_p:c *
```

```
\skip_if_exist:NTF * \skip_if_exist:cTF *
```

---

New: 2012-03-03

---

`\skip_if_exist_p:N <skip>`

`\skip_if_exist:NTF <skip> {<true code>} {<false code>}`

Tests whether the `<skip>` is currently defined. This does not check that the `<skip>` really is a skip variable.

---

```
\skip_add:Nn \skip_add:cn
```

```
\skip_gadd:Nn \skip_gadd:cn
```

---

Updated: 2011-10-22

---

`\skip_add:Nn <skip> {<skip expression>}`

Adds the result of the `<skip expression>` to the current content of the `<skip>`.

---

```
\skip_set:Nn \skip_set:cn
```

```
\skip_gset:Nn \skip_gset:cn
```

---

Updated: 2011-10-22

---

`\skip_set:Nn <skip> {<skip expression>}`

Sets `<skip>` to the value of `<skip expression>`, which must evaluate to a length with units and may include a rubber component (for example 1 cm plus 0.5 cm).

---

```
\skip_set_eq:NN
```

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

```
\skip_gset_eq:NN
```

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

---

`\skip_set_eq:NN <skip1> <skip2>`

Sets the content of `<skip1>` equal to that of `<skip2>`.

---

```
\skip_sub:Nn \skip_sub:cn
```

```
\skip_gsub:Nn \skip_gsub:cn
```

---

Updated: 2011-10-22

`\skip_sub:Nn <skip> {<skip expression>}`

Subtracts the result of the `<skip expression>` from the current content of the `<skip>`.

## 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.

## 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` ★

Recovering 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.

## 15 Viewing skip variables

---

```
\skip_show:N          \skip_show:N {⟨skip⟩}
```

---

`\skip_show:c`

Displays the value of the *⟨skip⟩* on the terminal.

Updated: 2015-08-03

---

```
\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:c`

New: 2014-08-22  
Updated: 2015-08-03

`\skip_log:N <skip>`  
Writes the value of the `<skip>` in the log file.

---

`\skip_log:n`  
`\skip_log:n {<skip expression>}`

New: 2014-08-22  
Updated: 2015-08-07

Writes the result of evaluating the `<skip expression>` in the log file.

## 16 Constant skips

---

`\c_max_skip`

Updated: 2012-11-02

The maximum value that can be stored as a skip (equal to `\c_max_dim` in length), with no stretch nor shrink component.

---

`\c_zero_skip`

Updated: 2012-11-01

A zero length as a skip, with no stretch nor shrink component.

## 17 Scratch skips

---

`\l_tmpa_skip`  
`\l_tmpb_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`  
`\g_tmpb_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.

## 18 Inserting skips into the output

---

`\skip_horizontal:N`  
`\skip_horizontal:c`  
`\skip_horizontal:n`

Updated: 2011-10-22

`\skip_horizontal:N <skip>`  
`\skip_horizontal:n {<skipexpr>}`

Inserts a horizontal `<skip>` into the current list. The argument can also be a `<dim>`.

**TeXhackers note:** `\skip_horizontal:N` is the TeX primitive `\hskip` renamed.

---

`\skip_vertical:N`  
`\skip_vertical:c`  
`\skip_vertical:n`

Updated: 2011-10-22

`\skip_vertical:N <skip>`  
`\skip_vertical:n {<skipexpr>}`

Inserts a vertical `<skip>` into the current list. The argument can also be a `<dim>`.

**TeXhackers note:** `\skip_vertical:N` is the TeX primitive `\vskip` renamed.

## 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:cn
```

---

New: 2012-03-05

---

```
\muskip_const:Nn <muskip> {{muskip expression}}
```

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)`.

---

```
\muskip_zero:N     \muskip_zero:c
\muskip_zero:N     \muskip_gzero:N
\muskip_zero:c     \muskip_gzero:c
```

---

```
\skip_zero:N <muskip>
```

Sets `<muskip>` to 0 mu.

---

```
\muskip_zero_new:N \muskip_zero_new:c
\muskip_gzero_new:N \muskip_gzero_new:c
```

---

New: 2012-01-07

---

```
\muskip_zero_new:N <muskip>
```

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.

---

```
\muskip_if_exist_p:N *
\muskip_if_exist_p:c *
\muskip_if_exist:NTF *
\muskip_if_exist:cTF *
```

---

New: 2012-03-03

---

```
\muskip_if_exist_p:N <muskip>
```

```
\muskip_if_exist:NTF <muskip> {{true code}} {{false code}}
```

Tests whether the `<muskip>` is currently defined. This does not check that the `<muskip>` really is a muskip variable.

---

```
\muskip_add:Nn    \muskip_add:cn
\muskip_gadd:Nn   \muskip_gadd:cn
```

---

Updated: 2011-10-22

---

```
\muskip_add:Nn <muskip> {{muskip expression}}
```

Adds the result of the `<muskip expression>` to the current content of the `<muskip>`.

---

```
\muskip_set:Nn    \muskip_set:cn
\muskip_gset:Nn   \muskip_gset:cn
```

---

Updated: 2011-10-22

---

```
\muskip_set:Nn <muskip> {{muskip expression}}
```

Sets `<muskip>` to the value of `<muskip expression>`, which must evaluate to a math length with units and may include a rubber component (for example 1 mu plus 0.5 mu).

---

```
\muskip_set_eq:NN  \muskip_set_eq:(cN|Nc|cc)
\muskip_gset_eq:NN \muskip_gset_eq:(cN|Nc|cc)
```

---

Sets the content of `<muskip1>` equal to that of `<muskip2>`.

---

```
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn
```

---

Updated: 2011-10-22

```
\muskip_sub:Nn <muskip> {<muskip expression>}
```

Subtracts the result of the *<muskip expression>* from the current content of the *<skip>*.

---

```
\muskip_eval:n *
```

---

Updated: 2011-10-22

```
\muskip_eval:n {<muskip expression>}
```

Evaluates the *<muskip expression>*, expanding any skips and token list variables within the *<expression>* to their content (without requiring `\muskip_use:N/\tl_use:N`) and applying the standard mathematical rules. The result of the calculation is left in the input stream as a *<muglue denotation>* after two expansions. This is expressed in `mu`, and requires suitable termination if used in a TeX-style assignment as it is *not* an *(internal muglue)*.

---

```
\muskip_use:N *
\muskip_use:c *
```

---

```
\muskip_use:N <muskip>
```

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>* 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.

## 22 Viewing muskip variables

---

```
\muskip_show:N
\muskip_show:c
```

---

Updated: 2015-08-03

```
\muskip_show:N <muskip>
```

Displays the value of the *<muskip>* on the terminal.

---

```
\muskip_show:n
```

---

New: 2011-11-22

Updated: 2015-08-07

```
\muskip_show:n {<muskip expression>}
```

Displays the result of evaluating the *<muskip expression>* on the terminal.

---

```
\muskip_log:N
\muskip_log:c
```

---

New: 2014-08-22

Updated: 2015-08-03

```
\muskip_log:N <muskip>
```

Writes the value of the *<muskip>* in the log file.

---

```
\muskip_log:n
```

---

New: 2014-08-22

Updated: 2015-08-07

```
\muskip_log:n {<muskip expression>}
```

Writes the result of evaluating the *<muskip expression>* in the log file.

## 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.

## 24 Scratch muskips

\l\_tmpa\_muskip \l\_tmpb\_muskip Scratch muskip 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\_muskip \g\_tmpb\_muskip Scratch muskip 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 Primitive conditional

\if\_dim:w  
  \if\_dim:w *dimen*<sub>1</sub> < i*relation* < i*dimen*<sub>2</sub>  
    *true code*  
  \else:  
    *false*  
  \fi:

Compare two dimensions. The *relation* is one of <, = or > with category code 12.

**TeXhackers note:** This is the TeX primitive \ifdim.

## Part XX

# 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  
}
```

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 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`.

## 1 Creating keys

---

`\keys_define:nn`

Updated: 2017-11-14

`\keys_define:nn {<module>} {<keyval list>}`

Parses the `<keyval list>` and defines the keys listed there for `<module>`. The `<module>` name should be a text value, but there are no restrictions on the nature of the text. 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 `.undefined:` property, all key properties define the key within the current TeX scope.

---

`.bool_set:N`      `<key> .bool_set:N = <boolean>`  
`.bool_set:c`  
`.bool_gset:N`  
`.bool_gset:c`

Updated: 2013-07-08

---

`.bool_set_inverse:N`      `<key> .bool_set_inverse:N = <boolean>`

`.bool_set_inverse:c`  
`.bool_gset_inverse:N`  
`.bool_gset_inverse:c`

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 3.

---

`.choices:nn`      `<key> .choices:nn = {\<choices>} {\<code>}`

`.choices:(Vn|on|xn)`

New: 2011-08-21

Updated: 2013-07-10

---

Sets `<key>` to act as a choice key, and defines a series `<choices>` which are implemented using the `<code>`. Inside `<code>`, `\l_keys_choice_t1` 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 3.

---

`.clist_set:N`      `<key> .clist_set:N = <comma list variable>`

`.clist_set:c`  
`.clist_gset:N`  
`.clist_gset:c`

New: 2011-09-11

---

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.

---

`.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>`.

---

.default:n  
.default:(v|o|x)  
Updated: 2013-07-09

---

`<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  
.dim\_set:c  
.dim\_gset:N  
.dim\_gset:c

---

`<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.

---

.fp\_set:N  
.fp\_set:c  
.fp\_gset:N  
.fp\_gset:c

---

`<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.

---

.groups:n  
New: 2013-07-14

---

`<key> .groups:n = {<groups>}`

Defines `<key>` as belonging to the `<groups>` declared. Groups provide a “secondary axis” for selectively setting keys, and are described in Section 6.

---

.inherit:n  
New: 2016-11-22

---

`<key> .inherit:n = {<parents>}`

Specifies that the `<key>` path should inherit the keys listed as `<parents>`. For example, after setting

```
\keys_define:n { foo } { test .code:n = \tl_show:n {#1} }
\keys_define:n { } { bar .inherit:n = foo }
```

setting

```
\keys_set:n { bar } { test = a }
```

will be equivalent to

```
\keys_set:n { foo } { test = a }
```

---

.initial:n  
.initial:(V|o|x)

---

Updated: 2013-07-09

`<key> .initial:n = {<value>}`

Initialises the `<key>` with the `<value>`, equivalent to

`\keys_set:nn {<module>} { <key> = <value> }`

---

.int\_set:N  
.int\_set:c  
.int\_gset:N  
.int\_gset:c

---

`<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.

---

.meta:n

---

Updated: 2013-07-10

`<key> .meta:n = {<keyval list>}`

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

---

New: 2013-07-10

`<key> .meta:nn = {<path>} {<keyval list>}`

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:

---

New: 2011-08-21

`<key> .multichoice:`

Sets `<key>` to act as a multiple choice key. Each valid choice for `<key>` must then be created, as discussed in section 3.

---

.multichoice:nn  
.multichoice:(Vn|on|xn)

---

New: 2011-08-21

Updated: 2013-07-10

`<key> .multichoice:nn {<choices>} {<code>}`

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\_t1 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 3.

---

.skip\_set:N  
.skip\_set:c  
.skip\_gset:N  
.skip\_gset:c

---

`<key> .skip_set:N = <skip>`

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.

---

.tl\_set:N  
.tl\_set:c  
.tl\_gset:N  
.tl\_gset:c

---

`<key> .tl_set:N = <token list variable>`

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  
.tl\_set\_x:c  
.tl\_gset\_x:N  
.tl\_gset\_x:c

---

`<key> .tl_set_x:N = <token list variable>`

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.

<code>.undefine:</code>	<code>&lt;key&gt; .undefine:</code>
<code>New: 2015-07-14</code>	Removes the definition of the <code>&lt;key&gt;</code> within the current scope.
<code>.value_forbidden:n</code>	<code>&lt;key&gt; .value_forbidden:n = true false</code>
<code>New: 2015-07-14</code>	Specifies that <code>&lt;key&gt;</code> cannot receive a <code>&lt;value&gt;</code> when used. If a <code>&lt;value&gt;</code> is given then an error will be issued. Setting the property <code>false</code> cancels the restriction.
<code>.value_required:n</code>	<code>&lt;key&gt; .value_required:n = true false</code>
<code>New: 2015-07-14</code>	Specifies that <code>&lt;key&gt;</code> must receive a <code>&lt;value&gt;</code> when used. If a <code>&lt;value&gt;</code> is not given then an error will be issued. Setting the property <code>false</code> cancels the restriction.

## 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 { module / 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 `module/subgroup/key`.

As illustrated in the next section, this subdivision is particularly relevant to making multiple choices.

## 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 \l\_keys\_choice\_tl

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 5. 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`.

## 4 Setting keys

---

```
\keys_set:nn
\keys_set:(nV|nv|no)
```

---

Updated: 2017-11-14

---

```
\l_keys_key_tl
\l_keys_path_tl
\l_keys_value_tl
```

---

Updated: 2015-07-14

```
\keys_set:nn {<module>} {<keyval list>}
```

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.

For each key processed, information of the full *path* of the key, the *name* of the key and the *value* of the key is available within three token list variables. These may be used within the code of the key.

The *value* is everything after the `=`, which may be empty if no value was given. This is stored in `\l_keys_value_tl`, 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_tl`, and will have been processed by `\tl_to_str:n`.

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_tl`, and will have been processed by `\tl_to_str:n`.

## 5 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_tl'~to~'#1'.
}
```

---

```
\keys_set_known:nnN           \keys_set_known:nnN {<module>} {<keyval list>} <t1>
\keys_set_known:(nVN|nvN|noN)
\keys_set_known:nn
\keys_set_known:(nV|nv|no)
```

---

New: 2011-08-23

Updated: 2017-05-27

---

In some cases, the desired behavior is to simply ignore unknown keys, collecting up information on these for later processing. The `\keys_set_known:nnN` function parses the `<keyval list>`, and sets those keys which are defined for `<module>`. Any keys which are unknown are not processed further by the parser. The key-value pairs for each *unknown* key name are stored in the `<t1>` in a comma-separated form (*i.e.* an edited version of the `<keyval list>`). The `\keys_set_known:nn` version skips this stage.

Use of `\keys_set_known:nnN` can be nested, with the correct residual `<keyval list>` returned at each stage.

## 6 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.

---

```
\keys_set_filter:nnnN          \keys_set_filter:nnnN {\<module>} {\<groups>} {\<keyval list>} {\<tl>}
\keys_set_filter:(nnVN|nnvN|nnoN)
\keys_set_filter:nnn
\keys_set_filter:(nnV|nnv|nno)
```

---

New: 2013-07-14

Updated: 2017-05-27

---

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:nnnN` can be nested, with the correct residual *<keyval list>* returned at each stage.

---

```
\keys_set_groups:nnn          \keys_set_groups:nnn {\<module>} {\<groups>} {\<keyval list>}
\keys_set_groups:(nnV|nnv|nno)
```

---

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.

## 7 Utility functions for keys

---

```
\keys_if_exist_p:nn ★      \keys_if_exist_p:nn {\<module>} {\<key>}
\keys_if_exist:nnTF ★      \keys_if_exist:nnTF {\<module>} {\<key>} {\<true code>} {\<false code>}
```

---

Updated: 2017-11-14

Tests if the *<key>* exists for *<module>*, *i.e.* if any code has been defined for *<key>*.

---

```
\keys_if_choice_exist_p:nnn ★ \keys_if_choice_exist_p:nnn {\<module>} {\<key>} {\<choice>}
\keys_if_choice_exist:nnnTF ★ \keys_if_choice_exist:nnnTF {\<module>} {\<key>} {\<choice>} {\<true code>} {\<false code>}
```

---

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 {\<module>} {\<key>}
```

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 {\<module>} {\<key>}
```

New: 2014-08-22

Updated: 2015-08-09

---

Writes in the log file the information associated to the *<key>* for a *<module>*. See also `\keys_show:nn` which displays the result in the terminal.

## 8 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  $\langle\text{key-value list}\rangle$  into  $\langle\text{keys}\rangle$  and associated  $\langle\text{values}\rangle$ . 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

Updated: 2011-09-08

---

\keyval\_parse:NNn <function<sub>1</sub>> <function<sub>2</sub>> {<key-value list>}

Parses the <key-value list> into a series of <keys> and associated <values>, or keys alone (if no <value> was given). <function<sub>1</sub>> should take one argument, while <function<sub>2</sub>> should absorb two arguments. After \keyval\_parse:NNn has parsed the <key-value list>, <function<sub>1</sub>> is used to process keys given with no value and <function<sub>2</sub>> is used to process keys given with a value. The order of the <keys> in the <key-value list> 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 <key> and <value>, then one *outer* set of braces is removed from the <key> and <value> as part of the processing.

## Part XXI

# The **l3intarray** package: fast global integer arrays

## 1 **l3intarray** documentation

For applications requiring heavy use of integers, this module provides arrays which can be accessed in constant time (contrast **l3seq**, 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`

New: 2018-03-29

---

`\intarray_new:Nn <intarray var> {<size>}`

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.

---

`\intarray_count:N *`

New: 2018-03-29

---

`\intarray_count:N <intarray var>`

Expands to the number of entries in the *<integer array variable>*. Contrarily to `\seq_count:N` this is performed in constant time.

---

`\intarray_gset:Nnn`

New: 2018-03-29

---

`\intarray_gset:Nnn <intarray var> {<position>} {<value>}`

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.

---

`\intarray_gzero:N`

New: 2018-05-04

---

`\intarray_gzero:N <intarray var>`

Sets all entries of the *<integer array variable>* to zero. Assignments are always global.

---

`\intarray_item:Nn *`

New: 2018-03-29

---

`\intarray_item:Nn <intarray var> {<position>}`

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 *`

New: 2018-05-05

---

`\intarray_rand_item:N <intarray var>`

Selects a pseudo-random item of the *<integer array>*. If the *<integer array>* is empty, produce an error.

## 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 TeXLive settings).

## Part XXII

# The **I3fp** 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  $!x$ , conjunction  $x \&& y$ , disjunction  $x || y$ , ternary operator  $x ? y : z$ .
- Exponentials:  $\exp x$ ,  $\ln x$ ,  $x^y$ .
- Trigonometry:  $\sin x$ ,  $\cos x$ ,  $\tan x$ ,  $\cot x$ ,  $\sec x$ ,  $\csc x$  expecting their arguments in radians, and  $\text{sind } x$ ,  $\text{cosd } x$ ,  $\text{tand } x$ ,  $\text{cotd } x$ ,  $\text{seed } x$ ,  $\text{csed } x$  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  $\text{asind } x$ ,  $\text{acosd } x$ ,  $\text{atand } x$ ,  $\text{cotd } x$ ,  $\text{seed } x$ ,  $\text{csed } x$  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,  $\text{NaN}$  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:  $\text{rand}()$ ,  $\text{randint}(m, n)$ .
- Constants:  $\text{pi}$ ,  $\text{deg}$  (one degree in radians).
- Dimensions, automatically expressed in points, e.g.,  $\text{pc}$  is 12.
- Automatic conversion (no need for  $\backslash\langle \text{type} \rangle_{\text{use}}:\text{N}$ ) 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 9.1 for a description of what a floating point is, section 9.2 for details about how an expression is parsed, and section 9.3 to know what the various operations do. Some operations may raise exceptions (error messages), described in section 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`.

## 1 Creating and initialising floating point variables

---

`\fp_new:N`  
`\fp_new:c`  
 Updated: 2012-05-08

---

`\fp_new:N <fp var>`

Creates a new  $\langle fp \, var \rangle$  or raises an error if the name is already taken. The declaration is global. The  $\langle fp \, var \rangle$  is initially `+0`.

---

`\fp_const:Nn`  
`\fp_const:cn`  
 Updated: 2012-05-08

---

`\fp_const:Nn <fp var> {<floating point expression>}`

Creates a new constant  $\langle fp \, var \rangle$  or raises an error if the name is already taken. The  $\langle fp \, var \rangle$  is set globally equal to the result of evaluating the  $\langle floating \, point \, expression \rangle$ .

---

`\fp_zero:N`  
`\fp_zero:c`  
`\fp_gzero:N`  
`\fp_gzero:c`  
 Updated: 2012-05-08

---

`\fp_zero:N <fp var>`

Sets the  $\langle fp \, var \rangle$  to `+0`.

---

```
\fp_zero_new:N  
\fp_zero_new:c  
\fp_gzero_new:N  
\fp_gzero_new:c
```

Updated: 2012-05-08

```
\fp_zero_new:N <fp var>
```

Ensures that the  $\langle fp \ var \rangle$  exists globally by applying `\fp_new:N` if necessary, then applies `\fp_(g)zero:N` to leave the  $\langle fp \ var \rangle$  set to +0.

Updated: 2012-05-08

## 2 Setting floating point variables

---

```
\fp_set:Nn  
\fp_set:cn  
\fp_gset:Nn  
\fp_gset:cn
```

Updated: 2012-05-08

```
\fp_set:Nn <fp var> {<floating point expression>}
```

Sets  $\langle fp \ var \rangle$  equal to the result of computing the  $\langle floating \ point \ expression \rangle$ .

---

```
\fp_set_eq:NN  
\fp_set_eq:(cN|Nc|cc)  
\fp_gset_eq:NN  
\fp_gset_eq:(cN|Nc|cc)
```

Updated: 2012-05-08

```
\fp_set_eq:NN <fp var1> <fp var2>
```

Sets the floating point variable  $\langle fp \ var_1 \rangle$  equal to the current value of  $\langle fp \ var_2 \rangle$ .

---

```
\fp_add:Nn  
\fp_add:cn  
\fp_gadd:Nn  
\fp_gadd:cn
```

Updated: 2012-05-08

```
\fp_add:Nn <fp var> {<floating point expression>}
```

Adds the result of computing the  $\langle floating \ point \ expression \rangle$  to the  $\langle fp \ var \rangle$ . This also applies if  $\langle fp \ var \rangle$  and  $\langle floating \ point \ expression \rangle$  evaluate to tuples of the same size.

---

```
\fp_sub:Nn  
\fp_sub:cn  
\fp_gsub:Nn  
\fp_gsub:cn
```

Updated: 2012-05-08

```
\fp_sub:Nn <fp var> {<floating point expression>}
```

Subtracts the result of computing the  $\langle floating \ point \ expression \rangle$  from the  $\langle fp \ var \rangle$ . This also applies if  $\langle fp \ var \rangle$  and  $\langle floating \ point \ expression \rangle$  evaluate to tuples of the same size.

## 3 Using floating points

---

```
\fp_eval:n  *
```

New: 2012-05-08  
Updated: 2012-07-08

```
\fp_eval:n {<floating point expression>}
```

Evaluates the  $\langle floating \ point \ expression \rangle$  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  $\text{NaN}$  trigger an “invalid operation” exception. For a tuple, each item is converted using `\fp_eval: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`.

<hr/> <p><b>\fp_to_decimal:N</b> ★  <b>\fp_to_decimal:c</b> ★  <b>\fp_to_decimal:n</b> ★</p> <p>New: 2012-05-08  Updated: 2012-07-22</p> <hr/> <p><b>\fp_to_dim:N</b> ★  <b>\fp_to_dim:c</b> ★  <b>\fp_to_dim:n</b> ★</p> <p>Updated: 2016-03-22</p> <hr/> <p><b>\fp_to_int:N</b> ★  <b>\fp_to_int:c</b> ★  <b>\fp_to_int:n</b> ★</p> <p>Updated: 2012-07-08</p> <hr/> <p><b>\fp_to_scientific:N</b> ★  <b>\fp_to_scientific:c</b> ★  <b>\fp_to_scientific:n</b> ★</p> <p>New: 2012-05-08  Updated: 2016-03-22</p> <hr/> <p><b>\fp_to_t1:N</b> ★  <b>\fp_to_t1:c</b> ★  <b>\fp_to_t1:n</b> ★</p> <p>Updated: 2016-03-22</p>	<pre>\fp_to_decimal:N &lt;fp var&gt; \fp_to_decimal:n {&lt;floating point expression&gt;}</pre> <p>Evaluates the <i>&lt;floating point expression&gt;</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 <math>\pm\infty</math> and NaN trigger an “invalid operation” exception. For a tuple, each item is converted using <code>\fp_to_decimal:n</code> and they are combined as <math>(\langle fp_1 \rangle, \lfloor \langle fp_2 \rangle, \lfloor \dots \langle fp_n \rangle)</math> if <math>n &gt; 1</math> and <math>(\langle fp_1 \rangle,)</math> or <math>()</math> for fewer items.</p> <pre>\fp_to_dim:N &lt;fp var&gt; \fp_to_dim:n {&lt;floating point expression&gt;}</pre> <p>Evaluates the <i>&lt;floating point expression&gt;</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 <math>[-2^{14} + 2^{-17}, 2^{14} - 2^{-17}]</math> of valid TeX dimensions, leading to overflow errors if used as a dimension. Tuples, as well as the values <math>\pm\infty</math> and NaN, trigger an “invalid operation” exception.</p> <pre>\fp_to_int:N &lt;fp var&gt; \fp_to_int:n {&lt;floating point expression&gt;}</pre> <p>Evaluates the <i>&lt;floating point expression&gt;</i>, and rounds the result to the closest integer, rounding exact ties to an even integer. The result may be outside the range <math>[-2^{31} + 1, 2^{31} - 1]</math> of valid TeX integers, leading to overflow errors if used in an integer expression. Tuples, as well as the values <math>\pm\infty</math> and NaN, trigger an “invalid operation” exception.</p> <pre>\fp_to_scientific:N &lt;fp var&gt; \fp_to_scientific:n {&lt;floating point expression&gt;}</pre> <p>Evaluates the <i>&lt;floating point expression&gt;</i> and expresses the result in scientific notation:  <math>\langle optional - \rangle \langle digit \rangle . \langle 15 digits \rangle e \langle optional sign \rangle \langle exponent \rangle</math></p> <p>The leading <i>&lt;digit&gt;</i> is non-zero except in the case of <math>\pm 0</math>. The values <math>\pm\infty</math> and NaN trigger an “invalid operation” exception. Normal category codes apply: thus the <code>e</code> is category code 11 (a letter). For a tuple, each item is converted using <code>\fp_to_scientific:n</code> and they are combined as <math>(\langle fp_1 \rangle, \lfloor \langle fp_2 \rangle, \lfloor \dots \langle fp_n \rangle)</math> if <math>n &gt; 1</math> and <math>(\langle fp_1 \rangle,)</math> or <math>()</math> for fewer items.</p> <pre>\fp_to_t1:N &lt;fp var&gt; \fp_to_t1:n {&lt;floating point expression&gt;}</pre> <p>Evaluates the <i>&lt;floating point expression&gt;</i> and expresses the result in (almost) the shortest possible form. Numbers in the ranges <math>(0, 10^{-3})</math> and <math>[10^{16}, \infty)</math> are expressed in scientific notation with trailing zeros trimmed and no decimal separator when there is a single significant digit (this differs from <code>\fp_to_scientific:n</code>). Numbers in the range <math>[10^{-3}, 10^{16})</math> are expressed in a decimal notation without exponent, with trailing zeros trimmed, and no decimal separator for integer values (see <code>\fp_to_decimal:n</code>). Negative numbers start with <code>-</code>. The special values <math>\pm 0</math>, <math>\pm\infty</math> and NaN are rendered as <code>0</code>, <code>-0</code>, <code>inf</code>, <code>-inf</code>, and <code>nan</code> respectively. Normal category codes apply and thus <code>inf</code> or <code>nan</code>, if produced, are made up of letters. For a tuple, each item is converted using <code>\fp_to_t1:n</code> and they are combined as <math>(\langle fp_1 \rangle, \lfloor \langle fp_2 \rangle, \lfloor \dots \langle fp_n \rangle)</math> if <math>n &gt; 1</math> and <math>(\langle fp_1 \rangle,)</math> or <math>()</math> for fewer items.</p>
---	---

---

\fp\_use:N ★  
\fp\_use:c ★  
Updated: 2012-07-08

---

\fp\_use:N *fp var*

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.

## 4 Floating point conditionals

---

\fp\_if\_exist\_p:N ★  
\fp\_if\_exist\_p:c ★  
\fp\_if\_exist:NTF ★  
\fp\_if\_exist:cTF ★  
Updated: 2012-05-08

---

\fp\_if\_exist\_p:N *fp var*  
\fp\_if\_exist:NTF *fp var* {*true code*} {*false code*}

Tests whether the *fp var* is currently defined. This does not check that the *fp var* really is a floating point variable.

---

\fp\_compare\_p:nNn ★  
\fp\_compare:nNnTF ★  
Updated: 2012-05-08

---

\fp\_compare\_p:nNn {*fpexpr*<sub>1</sub>} *relation* {*fpexpr*<sub>2</sub>}  
\fp\_compare:nNnTF {*fpexpr*<sub>1</sub>} *relation* {*fpexpr*<sub>2</sub>} {*true code*} {*false code*}

Compares the *fpexpr*<sub>1</sub> and the *fpexpr*<sub>2</sub>, and returns true if the *relation* 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.

---

\fp\_compare\_p:n ★  
\fp\_compare:nTF ★  
Updated: 2013-12-14

---

```
\fp_compare_p:n
{
  ⟨fpexpr₁⟩ ⟨relation₁⟩
  ...
  ⟨fpexprₙ⟩ ⟨relationₙ⟩
  ⟨fpexpr_{n+1}⟩
}
\fp_compare:nTF
{
  ⟨fpexpr₁⟩ ⟨relation₁⟩
  ...
  ⟨fpexprₙ⟩ ⟨relationₙ⟩
  ⟨fpexpr_{n+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 *⟨intexpr₁⟩* and *⟨intexpr₂⟩* using the *⟨relation₁⟩*, then *⟨intexpr₂⟩* and *⟨intexpr₃⟩* using the *⟨relation₂⟩*, until finally comparing *⟨intexprₙ⟩* and *⟨intexpr\_{n+1}⟩* using the *⟨relationₙ⟩*. 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).

## 5 Floating point expression loops

---

\fp\_do\_until:nNnn ★  
New: 2012-08-16

---

```
\fp_do_until:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨code⟩}
```

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 ★  
New: 2012-08-16

---

```
\fp_do_while:nNnn {⟨fpexpr₁⟩} ⟨relation⟩ {⟨fpexpr₂⟩} {⟨code⟩}
```

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 {\langle fpexpr_1\rangle} {\langle relation\rangle} {\langle fpexpr_2\rangle} {\langle code\rangle}</code>
		Evaluates the relationship between the two <i>floating point expressions</i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is <code>false</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
<code>\fp_while_do:nNnn</code>	☆	<code>\fp_while_do:nNnn {\langle fpexpr_1\rangle} {\langle relation\rangle} {\langle fpexpr_2\rangle} {\langle code\rangle}</code>
		Evaluates the relationship between the two <i>floating point expressions</i> as described for <code>\fp_compare:nNnTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is <code>true</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>false</code> .
<code>\fp_do_until:nn</code>	☆	<code>\fp_do_until:nn {\langle fpexpr_1\rangle} {\langle relation\rangle} {\langle fpexpr_2\rangle} {\langle code\rangle}</code>
	New: 2012-08-16	Places the <i>code</i> in the input stream for TeX to process, and then evaluates the relationship between the two <i>floating point expressions</i> as described for <code>\fp_compare:nTF</code> . If the test is <code>false</code> then the <i>code</i> is inserted into the input stream again and a loop occurs until the <i>relation</i> is <code>true</code> .
	Updated: 2013-12-14	
<code>\fp_do_while:nn</code>	☆	<code>\fp_do_while:nn {\langle fpexpr_1\rangle} {\langle relation\rangle} {\langle fpexpr_2\rangle} {\langle code\rangle}</code>
	New: 2012-08-16	Places the <i>code</i> in the input stream for TeX to process, and then evaluates the relationship between the two <i>floating point expressions</i> as described for <code>\fp_compare:nTF</code> . If the test is <code>true</code> then the <i>code</i> is inserted into the input stream again and a loop occurs until the <i>relation</i> is <code>false</code> .
	Updated: 2013-12-14	
<code>\fp_until_do:nn</code>	☆	<code>\fp_until_do:nn {\langle fpexpr_1\rangle} {\langle relation\rangle} {\langle fpexpr_2\rangle} {\langle code\rangle}</code>
	New: 2012-08-16	Evaluates the relationship between the two <i>floating point expressions</i> as described for <code>\fp_compare:nTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is <code>false</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>true</code> .
	Updated: 2013-12-14	
<code>\fp_while_do:nn</code>	☆	<code>\fp_while_do:nn {\langle fpexpr_1\rangle} {\langle relation\rangle} {\langle fpexpr_2\rangle} {\langle code\rangle}</code>
	New: 2012-08-16	Evaluates the relationship between the two <i>floating point expressions</i> as described for <code>\fp_compare:nTF</code> , and then places the <i>code</i> in the input stream if the <i>relation</i> is <code>true</code> . After the <i>code</i> has been processed by TeX the test is repeated, and a loop occurs until the test is <code>false</code> .
	Updated: 2013-12-14	

\fp\_step\_function:nnnN ☆  
\fp\_step\_function:nnnc ☆

New: 2016-11-21  
Updated: 2016-12-06

\fp\_step\_function:nnnN {<initial value>} {<step>} {<final value>} <function>

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:nnn  
New: 2016-11-21  
Updated: 2016-12-06

\fp\_step\_inline:nnn {<initial value>} {<step>} {<final value>} {<code>}

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:nnnNn  
New: 2017-04-12

\fp\_step\_variable:nnnNn  
{<initial value>} {<step>} {<final value>} {tl var} {<code>}

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>*.

## 6 Some useful constants, and scratch variables

\c\_zero\_fp  
\c\_minus\_zero\_fp  
New: 2012-05-08

Zero, with either sign.

\c\_one\_fp  
New: 2012-05-08

One as an fp: useful for comparisons in some places.

---

`\c_inf_fp`  
`\c_minus_inf_fp`

New: 2012-05-08

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

---

`\c_e_fp`

Updated: 2012-05-08

The value of the base of the natural logarithm,  $e = \exp(1)$ .

---

`\c_pi_fp`

Updated: 2013-11-17

The value of  $\pi$ . This can be input directly in a floating point expression as `pi`.

---

`\c_one_degree_fp`

New: 2012-05-08

Updated: 2013-11-17

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`  
`\l_tmpb_fp`

Scratch floating points 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_fp`  
`\g_tmpb_fp`

Scratch floating points 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.

## 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`

New: 2012-07-19  
Updated: 2017-02-13

`\fp_trap:nn {<exception>} {<trap type>}`

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

- `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 TeX 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`  
`flag_fp_underflow`  
`flag_fp_invalid_operation`  
`flag_fp_division_by_zero`

Flags denoting the occurrence of various floating-point exceptions.

## 8 Viewing floating points

---

`\fp_show:N`  
`\fp_show:c`  
`\fp_show:n`

New: 2012-05-08  
Updated: 2015-08-07

`\fp_show:N {fp var}`  
`\fp_show:n {floating point expression}`

Evaluates the `<floating point expression>` and displays the result in the terminal.

---

`\fp_log:N`  
`\fp_log:c`  
`\fp_log:n`

New: 2014-08-22  
Updated: 2015-08-07

`\fp_log:N {fp var}`  
`\fp_log:n {floating point expression}`

Evaluates the `<floating point expression>` and writes the result in the log file.

## 9 Floating point expressions

### 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**, 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**.

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.

## 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 `+`, `-`, `!`.
- Binary `*`, `/`, and implicit multiplication by juxtaposition (`2pi`, `3(4+5)`, *etc*).
- 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, those precedences imply that

$$\begin{aligned}\sin 2\pi &= \sin(2)\pi! = 0, \\ 2^2 \max(3, 5) &= 2^2 \max(3, 5) = 20.\end{aligned}$$

Functions are called on the value of their argument, contrarily to `\TeX` macros.

## 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 ⟨⟨operator<sub>2</sub>⟩⟩ if ⟨⟨operator<sub>1</sub>⟩⟩ is true (not ±0), and ⟨⟨operator<sub>3</sub>⟩⟩ if ⟨⟨operator<sub>1</sub>⟩⟩ is false (±0). All three ⟨⟨operator⟩⟩ 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 ⟨⟨operator<sub>1</sub>⟩⟩ is true (not ±0), use that value, otherwise the value of ⟨⟨operator<sub>2</sub>⟩⟩. Both ⟨⟨operator⟩⟩ are evaluated in all cases; they may be tuples. In ⟨⟨operator<sub>1</sub>⟩⟩ || ⟨⟨operator<sub>2</sub>⟩⟩ || ... || ⟨⟨operator<sub>n</sub>⟩⟩, the first true (nonzero) ⟨⟨operator⟩⟩ 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 ⟨⟨operator<sub>1</sub>⟩⟩ is false (equal to ±0), use that value, otherwise the value of ⟨⟨operator<sub>2</sub>⟩⟩. Both ⟨⟨operator⟩⟩ are evaluated in all cases; they may be tuples. In ⟨⟨operator<sub>1</sub>⟩⟩ && ⟨⟨operator<sub>2</sub>⟩⟩ && ... && ⟨⟨operator<sub>n</sub>⟩⟩, the first false (±0) ⟨⟨operator⟩⟩ is used and if none is zero the last one is used.

---

```
<
=
>
?
_____
Updated: 2013-12-14
```

---

```
\fp_eval:n
{
  ⟨operator1⟩ ⟨relation1⟩
  ...
  ⟨operatorN⟩ ⟨relationN⟩
  ⟨operatorN+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 ⟨⟨operator<sub>i</sub>⟩⟩ ⟨⟨relation<sub>i</sub>⟩⟩ ⟨⟨operator<sub>i+1</sub>⟩⟩ are true, and +0 otherwise. All ⟨⟨operator⟩⟩ are evaluated (once) in all cases. See \fp\_compare:nTF for details.

**+** \fp\_eval:n { ⟨operator<sub>1</sub>⟩ + ⟨operator<sub>2</sub>⟩ }
**-** \fp\_eval:n { ⟨operator<sub>1</sub>⟩ - ⟨operator<sub>2</sub>⟩ }

Computes the sum or the difference of its two ⟨⟨operator⟩⟩. 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.

- $\backslash \text{fp\_eval:n} \{ \langle \text{operand}_1 \rangle * \langle \text{operand}_2 \rangle \}$   
 $\backslash \text{fp\_eval:n} \{ \langle \text{operand}_1 \rangle / \langle \text{operand}_2 \rangle \}$ 

Computes the product or the ratio of its two  $\langle \text{operands} \rangle$ . 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  $\langle \text{operand}_1 \rangle$  is a tuple and  $\langle \text{operand}_2 \rangle$  is a floating point number, each item of  $\langle \text{operand}_1 \rangle$  is multiplied or divided by  $\langle \text{operand}_2 \rangle$ . Multiplication also supports the case where  $\langle \text{operand}_1 \rangle$  is a floating point number and  $\langle \text{operand}_2 \rangle$  a tuple. Other combinations yield an “invalid operation” exception and a NaN result.
- $\backslash \text{fp\_eval:n} \{ + \langle \text{operand} \rangle \}$   
 $\backslash \text{fp\_eval:n} \{ - \langle \text{operand} \rangle \}$   
 $\backslash \text{fp\_eval:n} \{ ! \langle \text{operand} \rangle \}$ 

The unary + does nothing, the unary - changes the sign of the  $\langle \text{operand} \rangle$  (for a tuple, of all its components), and !  $\langle \text{operand} \rangle$  evaluates to 1 if  $\langle \text{operand} \rangle$  is false (is  $\pm 0$ ) and 0 otherwise (this is the not boolean function). Those operations never raise exceptions.
- $\backslash \text{fp\_eval:n} \{ \langle \text{operand}_1 \rangle ^\ast \langle \text{operand}_2 \rangle \}$   
 $\backslash \text{fp\_eval:n} \{ \langle \text{operand}_1 \rangle ^\wedge \langle \text{operand}_2 \rangle \}$ 

Raises  $\langle \text{operand}_1 \rangle$  to the power  $\langle \text{operand}_2 \rangle$ . This operation is right associative, hence  $2 ^\ast 2 ^\ast 3 = 2^{2^3} = 256$ . If  $\langle \text{operand}_1 \rangle$  is negative or  $-0$  then: the result’s sign is + if the  $\langle \text{operand}_2 \rangle$  is infinite and  $(-1)^p$  if the  $\langle \text{operand}_2 \rangle$  is  $p/5^q$  with  $p, q$  integers; the result is  $+0$  if  $\text{abs}(\langle \text{operand}_1 \rangle)^\wedge \langle \text{operand}_2 \rangle$  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  $\backslash \text{fp\_eval:n} \{ \text{abs}( \langle \text{fpexpr} \rangle ) \}$ 

Computes the absolute value of the  $\langle \text{fpexpr} \rangle$ . If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases. See also  $\backslash \text{fp\_abs:n}$ .
- exp  $\backslash \text{fp\_eval:n} \{ \text{exp}( \langle \text{fpexpr} \rangle ) \}$ 

Computes the exponential of the  $\langle \text{fpexpr} \rangle$ . “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
- ln  $\backslash \text{fp\_eval:n} \{ \text{ln}( \langle \text{fpexpr} \rangle ) \}$ 

Computes the natural logarithm of the  $\langle \text{fpexpr} \rangle$ . Negative numbers have no (real) logarithm, hence the “invalid operation” is raised in that case, including for  $\text{ln}(-0)$ . “Division by zero” occurs when evaluating  $\text{ln}(+0) = -\infty$ . “Underflow” and “overflow” occur when appropriate. If the operand is a tuple, “invalid operation” occurs.
- max  $\backslash \text{fp\_eval:n} \{ \text{max}( \langle \text{fpexpr}_1 \rangle , \langle \text{fpexpr}_2 \rangle , \dots ) \}$   
min  $\backslash \text{fp\_eval:n} \{ \text{min}( \langle \text{fpexpr}_1 \rangle , \langle \text{fpexpr}_2 \rangle , \dots ) \}$ 

Evaluates each  $\langle \text{fpexpr} \rangle$  and computes the largest (smallest) of those. If any of the  $\langle \text{fpexpr} \rangle$  is a NaN or tuple, the result is NaN. If any operand is a tuple, “invalid operation” occurs; these operations do not raise exceptions in other cases.

---

**round**  
**trunc**  
**ceil**  
**floor**

New: 2013-12-14  
Updated: 2015-08-08

---

```
\fp_eval:n { round ( <fpexpr> ) }
\fp_eval:n { round ( <fpexpr1> , <fpexpr2> ) }
\fp_eval:n { round ( <fpexpr1> , <fpexpr2> , <fpexpr3> ) }
```

Only **round** accepts a third argument. Evaluates  $\langle fpexpr_1 \rangle = x$  and  $\langle fpexpr_2 \rangle = n$  and  $\langle fpexpr_3 \rangle = t$  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 **NaN**; if  $n$  is neither  $\pm \infty$  nor an integer, then an “invalid operation” exception is raised. When  $\langle fpexpr_2 \rangle$  is omitted,  $n = 0$ , i.e.,  $\langle fpexpr_1 \rangle$  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 **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  $\langle fpexpr_2 \rangle < -9984$ ). If any operand is a tuple, “invalid operation” occurs.

---

**sign**

```
\fp_eval:n { sign( <fpexpr> ) }
```

Evaluates the  $\langle fpexpr \rangle$  and determines its sign:  $+1$  for positive numbers and for  $+\infty$ ,  $-1$  for negative numbers and for  $-\infty$ ,  $\pm 0$  for  $\pm 0$ , and **NaN** for **NaN**. If the operand is a tuple, “invalid operation” occurs. This operation does not raise exceptions in other cases.

---

**sin**  
**cos**  
**tan**  
**cot**  
**csc**  
**sec**

Updated: 2013-11-17

---

```
\fp_eval:n { sin( <fpexpr> ) }
\fp_eval:n { cos( <fpexpr> ) }
\fp_eval:n { tan( <fpexpr> ) }
\fp_eval:n { cot( <fpexpr> ) }
\fp_eval:n { csc( <fpexpr> ) }
\fp_eval:n { sec( <fpexpr> ) }
```

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  $\sin(8 \times 180)$  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 **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.

---

```
atan          \fp_eval:n { atan( <fpexpr> ) }
acot          \fp_eval:n { atan( <fpexpr1> , <fpexpr2> ) }
New: 2013-11-02 \fp_eval:n { acot( <fpexpr> ) }
                           \fp_eval:n { acot( <fpexpr1> , <fpexpr2> ) }
```

---

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.

---

```
atand          \fp_eval:n { atand( <fpexpr> ) }
acotd          \fp_eval:n { atand( <fpexpr1> , <fpexpr2> ) }
New: 2013-11-02 \fp_eval:n { acotd( <fpexpr> ) }
                           \fp_eval:n { acotd( <fpexpr1> , <fpexpr2> ) }
```

---

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.

---

```
sqrt           \fp_eval:n { sqrt( <fpexpr> ) }
```

---

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 `<fpexpr>` or between `<fpexpr1>` and `<fpexpr2>` 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>	$1\text{sp} = 2^{-16}\text{pt} = 1.52587890625e - 5\text{pt}.$

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.

<code>true</code>	Other names for 1 and +0.
<code>false</code>	

---

### `\fp_abs:n` ★ `\fp_abs:n {<floating point expression>}`

New: 2012-05-14  
Updated: 2012-07-08

Evaluates the  $\langle\text{floating point expression}\rangle$  as described for `\fp_eval:n` and leaves the absolute value of the result in the input stream. If the argument is a tuple, “invalid operation” occurs, but no other case raises exceptions. Within floating point expressions, `abs()` can be used.

---

### `\fp_max:nn` ★ `\fp_min:nn` ★ `\fp_max:nn {<fp expression 1>} {<fp expression 2>}`

New: 2012-09-26

Evaluates the  $\langle\text{floating point expressions}\rangle$  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.

## 10 Disclaimer and roadmap

The package may break down if the escape character is among 0123456789\_+, or if it receives a TeX 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 {<fpexpr>} {<format>}`, but what should  $\langle\text{format}\rangle$  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 `\fp_if_nan:nTF`, and an `isnan` function?
- 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.
- `0e99999999999` 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 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 T<sub>E</sub>X fp packages, the international standards,...
- Also take into account the “inexact” exception?
- Support multi-character prefix operators (*e.g.*, @/ or whatever)?

## Part XXIII

# The **I3farray** package: fast global floating point arrays

## 1 I3farray documentation

For applications requiring heavy use of floating points, this module provides arrays which can be accessed in constant time (contrast **I3seq**, where access time is linear). The interface is very close to that of **I3intarray**. The size of the array is fixed and must be given at point of initialisation

Currently *all* functions in this module are candidates. Their documentation can be found in **I3candidates**.

# Part XXIV

## The **I3sort** package

### Sorting functions

#### 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:`  
`\sort_return_swapped:`

---

New: 2017-02-06

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

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.

## Part XXV

# The **I3tl-analysis** package: Analysing token lists

## 1 I3tl-analysis documentation

This module mostly provides internal functions for use in the `I3regex` module. However, it provides as a side-effect a user debugging function, very similar to the `\ShowTokens` macro from the `ted` package.

---

```
\tl_analysis_show:N  
\tl_analysis_show:n
```

---

New: 2018-04-09

```
\tl_analysis_show:n {\langle token list\rangle}
```

Displays to the terminal 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
```

---

New: 2018-04-09

```
\tl_analysis_map_inline:nn {\langle token list\rangle} {\langle inline function\rangle}
```

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:

- `\langle tokens\rangle`, which both `\o-`expand and `\x-`expand to the `\langle token\rangle`. The detailed form of `\langle token\rangle` may change in later releases.
- `\langle char code\rangle`, a decimal representation of the character code of the token, `-1` if it is a control sequence (with `\langle catcode\rangle 0`).
- `\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).

## Part XXVI

# The `\3regex` package: Regular expressions in `TEX`

The `\3regex` 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_const:Nn`. For example,

```
\regex_const:Nn \c_foo_regex { \c{begin} \cB. (\c[^BE].*) \cE. }
```

stores in `\c_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.

## 1 Syntax of regular expressions

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|[cem]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+|\cC.)\,*` 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.

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).

Character types.

- . 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).

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.

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.

Alternation and capturing groups.

A|B|C Either one of A, B, or C.

(...) Capturing group.

(?:...) Non-capturing group.

(?|...) 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.

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, character property, 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.

`\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{<tl var name>}` matches the exact contents of the token list  $\langle tl\ var \rangle$ . Within a `\c{...}` control sequence matching, the `\u` escape sequence only expands its argument once, in effect performing `\tl_to_str:v`. Quantifiers are not supported directly: use a group.

The option `(?i)` makes the match case insensitive (identifying `A-Z` with `a-z`; no Unicode support yet). 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)[Y-\\"]` is equivalent to `[YZ\[\\yz]`, and `(?i)[^aeiou]` matches any character which is not a vowel. Neither character properties, nor `\c{...}` nor `\u{...}` are affected by the `i` option.

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.

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.

## 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>}`;
- `\_` inserts a space (spaces are ignored when not escaped);

- `\a`, `\e`, `\f`, `\n`, `\r`, `\t`, `\xhh`, `\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 TeX, 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]?1|o) . } { (\O--\1) } \l_my_t1
```

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

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.

The characters inserted by the replacement have category code 12 (other) by default, with the exception of space characters. Spaces inserted through `\_` have category code 10, while spaces inserted through `\x20` or `\x{20}` have category code 12. The escape sequence `\c` allows to insert characters with arbitrary category codes, as well as control sequences.

`\c{X}{...}` 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 `\c{L}(Hello\c{S}\ 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{\(tl var name\)}` allows to insert the contents of the token list with name `\(tl var name\)` 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_\0_t1} } \l_my_t1
```

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

### 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`

New: 2017-05-26

---

`\regex_new:N <regex var>`

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_gset:Nn`  
`\regex_const:Nn`

New: 2017-05-26

---

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

Stores a compiled version of the `<regular expression>` in the `<regex var>`. 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) }`

The assignment is local for `\regex_set:Nn` and global for `\regex_gset:Nn`. Use `\regex_const:Nn` for compiled expressions which never change.

---

`\regex_show:n`  
`\regex_show:N`

New: 2017-05-26

---

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

Shows how `l3regex` interprets the `<regex>`. For instance, `\regex_show:n {{\A X|Y}}` shows

`+--branch`

`anchor at start (\A)`

`char code 88`

`+--branch`

`char code 89`

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

### 4 Matching

All regular expression functions are available in both `:n` and `:N` variants. The former require a “standard” regular expression, while the latter require a compiled expression as generated by `\regex_(g)set:Nn`.

---

`\regex_match:nnTF`

`\regex_match:NnTF`

New: 2017-05-26

---

`\regex_match:nnTF {{<regex>}} {{<token list>}} {{<true code>}} {{<false code>}}`

Tests whether the `<regular expression>` matches any part of the `<token list>`. For instance,

`\regex_match:nnTF { b [cde]* } { abecdcx } { 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
```

---

New: 2017-05-26

```
\regex_count:nnN {<regex>} {<token list>} <int var>
```

Sets *<int var>* within the current TeX 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) } { abbabababb } \l_foo_int
```

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

## 5 Submatch extraction

---

```
\regex_extract_once:nnN
\regex_extract_once:nnNTF
\regex_extract_once:NnN
\regex_extract_once:NnNTF
```

---

New: 2017-05-26

```
\regex_extract_once:nnN {<regex>} {<token list>} <seq var>
\regex_extract_once:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}
```

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:nnNTF
\regex_extract_all:NnN
\regex_extract_all:NnNTF
```

---

New: 2017-05-26

```
\regex_extract_all:nnN {<regex>} {<token list>} <seq var>
\regex_extract_all:nnNTF {<regex>} {<token list>} <seq var> {<true code>} {<false code>}
```

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:nnNTF  
\regex_split:NnN  
\regex_split:NnNTF
```

New: 2017-05-26

```
\regex_split:nnN {\<regular expression>} {\<token list>} {\<seq var>}  
\regex_split:nnNTF {\<regular expression>} {\<token list>} {\<seq var>} {\<true code>}  
{\<false code>}
```

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

```
\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.

## 6 Replacement

```
\regex_replace_once:nnN  
\regex_replace_once:nnNTF  
\regex_replace_once:NnN  
\regex_replace_once:NnNTF
```

New: 2017-05-26

```
\regex_replace_once:nnN {\<regular expression>} {\<replacement>} {\<tl var>}  
\regex_replace_once:nnNTF {\<regular expression>} {\<replacement>} {\<tl var>} {\<true code>} {\<false code>}
```

Searches for the *<regular expression>* in the *<token list>* and replaces the first match with the *<replacement>*. The result is assigned locally to *<tl var>*. In the *<replacement>*, *\0* represents the full match, *\1* represent the contents of the first capturing group, *\2* of the second, *etc.*

```
\regex_replace_all:nnN  
\regex_replace_all:nnNTF  
\regex_replace_all:NnN  
\regex_replace_all:NnNTF
```

New: 2017-05-26

```
\regex_replace_all:nnN {\<regular expression>} {\<replacement>} {\<tl var>}  
\regex_replace_all:nnNTF {\<regular expression>} {\<replacement>} {\<tl var>} {\<true code>} {\<false code>}
```

Replaces all occurrences of the *\regular expression* in the *<token list>* 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>*.

## 7 Constants and variables

```
\l_tmpa_regex  
\l_tmpb_regex
```

New: 2017-12-11

Scratch regex 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.

```
\g_tmpa_regex  
\g_tmpb_regex
```

New: 2017-12-11

Scratch regex 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.

## 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 `.s` 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 `current_state` and `current_submatches`.
  - If possible, when a state is reused by the same thread, kill other subthreads.
  - Use an array rather than `\l_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.
- Provide a syntax such as `\ur{l_my_regex}` to use an already-compiled regex in a more complicated regex. This makes regexes more easily composable.
- Allowing `\u{l_my_t1}` in more places, for instance as the number of repetitions in a quantifier.

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: 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.
- `\ddd`, matching the character with octal code `ddd`: we already have `\x{...}` and the syntax is confusingly close to what we could have used for backreferences (`\1`, `\2`, ...), making it harder to produce useful error message.
- `\cx`, similar to `\TeX`'s own `\^{\^}x`.
- Comments: `\TeX` already has its own system for comments.
- `\Q...E` escaping: this would require to read the argument verbatim, which is not in the scope of this module.
- `\C` single byte in UTF-8 mode: `Xe\TeX` and `Lua\TeX` serve us characters directly, and splitting those into bytes is tricky, encoding dependent, and most likely not useful anyways.

# Part XXVII

## The **I3box** package

### Boxes

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_`.

## 1 Creating and initialising boxes

---

`\box_new:N` `\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:c`  
`\box_gclear:N`  
`\box_gclear:c`

Clears the content of the `<box>` by setting the box equal to `\c_empty_box`.

---

`\box_clear_new:N`  
`\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:(cN|Nc|cc)`  
`\box_gset_eq:NN`  
`\box_gset_eq:(cN|Nc|cc)`

`\box_set_eq:NN` `<box1>` `<box2>`

Sets the content of `<box1>` equal to that of `<box2>`.

---

`\box_if_exist_p:N *`  
`\box_if_exist_p:c *`  
`\box_if_exist:NTF *`  
`\box_if_exist:cTF *`

`\box_if_exist_p:N` `<box>`  
`\box_if_exist:NTF` `<box>` {`<true code>`} {`<false code>`}

Tests whether the `<box>` is currently defined. This does not check that the `<box>` really is a box.

New: 2012-03-03

## 2 Using boxes

---

`\box_use:N` `\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_left:nn`

---

`\box_move_right:nn {<dimexpr>} {<box function>}`

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_down:nn`

---

`\box_move_up:nn {<dimexpr>} {<box function>}`

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 }`.

### 3 Measuring and setting box dimensions

---

`\box_dp:N`  
`\box_dp:c`

---

`\box_dp:N <box>`

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:c`

---

`\box_ht:N <box>`

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:c`

---

`\box_wd:N <box>`

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_set_dp:Nn`  
`\box_set_dp:cn`  
`\box_gset_dp:Nn`  
`\box_gset_dp:cn`

---

Updated: 2019-01-22

---

`\box_set_dp:Nn <box> {<dimension expression>}`

Set the depth (below the baseline) of the *<box>* to the value of the *{<dimension expression>}*.

---

`\box_set_ht:Nn`  
`\box_set_ht:cn`  
`\box_gset_ht:Nn`  
`\box_gset_ht:cn`

---

Updated: 2019-01-22

---

`\box_set_ht:Nn <box> {<dimension expression>}`

Set the height (above the baseline) of the *<box>* to the value of the *{<dimension expression>}*.

---

```
\box_set_wd:Nn
\box_set_wd:cn
\box_gset_wd:Nn
\box_gset_wd:cn
```

---

Updated: 2019-01-22

`\box_set_wd:Nn <box> {<dimension expression>}`

Set the width of the `<box>` to the value of the `{<dimension expression>}`.

---

```
\box_if_empty_p:N *
\box_if_empty_p:c *
\box_if_empty:NTF *
\box_if_empty:cTF *
```

---

`\box_if_empty_p:N <box>`

`\box_if_empty:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a empty (equal to `\c_empty_box`).

---

```
\box_if_horizontal_p:N *
\box_if_horizontal_p:c *
\box_if_horizontal:NTF *
\box_if_horizontal:cTF *
```

---

`\box_if_horizontal_p:N <box>`

`\box_if_horizontal:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a horizontal box.

---

```
\box_if_vertical_p:N *
\box_if_vertical_p:c *
\box_if_vertical:NTF *
\box_if_vertical:cTF *
```

---

`\box_if_vertical_p:N <box>`

`\box_if_vertical:NTF <box> {<true code>} {<false code>}`

Tests if `<box>` is a vertical box.

## 5 The last box inserted

---

```
\box_set_to_last:N
\box_set_to_last:c
\box_gset_to_last:N
\box_gset_to_last:c
```

---

`\box_set_to_last:N <box>`

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.

## 6 Constant boxes

---

```
\c_empty_box
```

---

Updated: 2012-11-04

This is a permanently empty box, which is neither set as horizontal nor vertical.

**TeXhackers note:** At the TeX level this is a void box.

## 7 Scratch boxes

---

```
\l_tmpa_box
\l_tmpb_box
```

---

Updated: 2012-11-04

Scratch boxes 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_box`  
`\g_tmpb_box`

Scratch boxes 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 Viewing box contents

---

`\box_show:N`  
`\box_show:c`

Updated: 2012-05-11

`\box_show:N <box>`

Shows full details of the content of the `<box>` in the terminal.

---

`\box_show:Nnn`  
`\box_show:cnn`

New: 2012-05-11

`\box_show:Nnn <box> {\<intexpr1>} {\<intexpr2>}`

Display the contents of `<box>` in the terminal, showing the first `<intexpr1>` items of the box, and descending into `<intexpr2>` group levels.

---

`\box_log:N`  
`\box_log:c`

New: 2012-05-11

`\box_log:N <box>`

Writes full details of the content of the `<box>` to the log.

---

`\box_log:Nnn`  
`\box_log:cnn`

New: 2012-05-11

`\box_log:Nnn <box> {\<intexpr1>} {\<intexpr2>}`

Writes the contents of `<box>` to the log, showing the first `<intexpr1>` items of the box, and descending into `<intexpr2>` group levels.

## 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.

## 10 Horizontal mode boxes

---

`\hbox:n`

Updated: 2017-04-05

`\hbox:n {\<contents>}`

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`

Updated: 2017-04-05

`\hbox_to_wd:nn {\<dimexpr>} {\<contents>}`

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`

Updated: 2017-04-05

`\hbox_to_zero:n {\<contents>}`

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:cN
\hbox_gset:Nn
\hbox_gset:cN
```

---

Updated: 2017-04-05

```
\hbox_set:Nn <box> {{contents}}
```

Typesets the *contents* at natural width and then stores the result inside the *box*.

---

```
\hbox_set_to_wd:Nnn
\hbox_set_to_wd:cnn
\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
```

---

Updated: 2017-04-05

```
\hbox_set_to_wd:Nnn <box> {{dimexpr}} {{contents}}
```

Typesets the *contents* to the width given by the *dimexpr* and then stores the result inside the *box*.

---

```
\hbox_overlap_right:n
```

---

Updated: 2017-04-05

```
\hbox_overlap_right:n {{contents}}
```

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
```

---

Updated: 2017-04-05

```
\hbox_overlap_left:n {{contents}}
```

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:
```

---

Updated: 2017-04-05

```
\hbox_set:Nw <box> <contents> \hbox_set_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.

---

```
\hbox_set_to_wd:Nnw
\hbox_set_to_wd:cnw
\hbox_gset_to_wd:Nnw
\hbox_gset_to_wd:cnw
```

---

New: 2017-06-08

```
\hbox_set_to_wd:Nnw <box> {{dimexpr}} <contents> \hbox_set_end:
```

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

---

```
\hbox_unpack:N
\hbox_unpack:c
```

---

```
\hbox_unpack:N <box>
```

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`.

## 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 {<contents>}`

Updated: 2017-04-05

Typesets the `<contents>` into a vertical box of natural height and includes this box in the current list for typesetting.

---

`\vbox_top:n` `\vbox_top:n {<contents>}`

Updated: 2017-04-05

Typesets the `<contents>` 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 {<dimexpr>} {<contents>}`

Updated: 2017-04-05

Typesets the `<contents>` into a vertical box of height `<dimexpr>` and then includes this box in the current list for typesetting.

---

`\vbox_to_zero:n` `\vbox_to_zero:n {<contents>}`

Updated: 2017-04-05

Typesets the `<contents>` into a vertical box of zero height and then includes this box in the current list for typesetting.

---

`\vbox_set:Nn` `\vbox_set:cn`  
`\vbox_gset:Nn` `\vbox_gset:cn`

Updated: 2017-04-05

`\vbox_set:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`.

---

`\vbox_set_top:Nn` `\vbox_set_top:cn`  
`\vbox_gset_top:Nn` `\vbox_gset_top:cn`

Updated: 2017-04-05

`\vbox_set_top:Nn <box> {<contents>}`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. 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:cnn`  
`\vbox_gset_to_ht:Nnn` `\vbox_gset_to_ht:cnn`

Updated: 2017-04-05

`\vbox_set_to_ht:Nnn <box> {<dimexpr>} {<contents>}`

Typesets the `<contents>` to the height given by the `<dimexpr>` and then stores the result inside the `<box>`.

---

`\vbox_set:Nw` `\vbox_set:cw`  
`\vbox_set_end:` `\vbox_gset:Nw`  
`\vbox_gset:cw` `\vbox_gset_end:`

Updated: 2017-04-05

`\vbox_set:Nw <box> <contents> \vbox_set_end:`

Typesets the `<contents>` at natural height and then stores the result inside the `<box>`. In contrast to `\vbox_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.

---

```
\vbox_set_to_ht:Nnw
\vbox_set_to_ht:cnw
\vbox_gset_to_ht:Nnw
\vbox_gset_to_ht:cnw
```

New: 2017-06-08

---

`\vbox_set_to_ht:Nnw <box> {\<dimexpr>} <contents> \vbox_set_end:`

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

---

```
\vbox_set_split_to_ht>NNn
\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

---

`\vbox_set_split_to_ht>NNn <box>1 <box>2 {\<dimexpr>}`

Sets `<box>1` to contain material to the height given by the `<dimexpr>` by removing content from the top of `<box>2` (which must be a vertical box).

---

```
\vbox_unpack:N <box>
```

---

```
\vbox_unpack:c
```

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`.

## 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:c
```

---

```
\box_use_drop:N <box>
```

Inserts the current content of the  $\langle box \rangle$  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:(cN|Nc|cc)
```

New: 2019-01-17

---

```
\box_set_eq_drop:NN <box1> <box2>
```

Sets the content of  $\langle box_1 \rangle$  equal to that of  $\langle box_2 \rangle$ , then drops  $\langle box_2 \rangle$ .

---

```
\box_gset_eq_drop:NN  
\box_gset_eq_drop:(cN|Nc|cc)
```

New: 2019-01-17

---

```
\box_gset_eq_drop:NN <box1> <box2>
```

Sets the content of  $\langle box_1 \rangle$  globally equal to that of  $\langle box_2 \rangle$ , then drops  $\langle box_2 \rangle$ .

---

```
\hbox_unpack_drop:N  
\hbox_unpack_drop:c
```

New: 2019-01-17

---

```
\hbox_unpack_drop:N <box>
```

Unpacks the content of the horizontal  $\langle box \rangle$ , retaining any stretching or shrinking applied when the  $\langle box \rangle$  was set. The original  $\langle box \rangle$  is then dropped.

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

---

```
\vbox_unpack_drop:N  
\vbox_unpack_drop:c
```

New: 2019-01-17

---

```
\vbox_unpack_drop:N <box>
```

Unpacks the content of the vertical  $\langle box \rangle$ , retaining any stretching or shrinking applied when the  $\langle box \rangle$  was set. The original  $\langle box \rangle$  is then dropped.

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

## 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:Nnn**  
**\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\}}
```

**\box\\_autosize\\_to\\_wd\\_and\\_ht\\_plus\\_dp:Nnn**  
**\box\\_autosize\\_to\\_wd\\_and\\_ht\\_plus\\_dp:cnn**  
**\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> {\{y-size\}}
```

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*.

---

Updated: 2019-01-22

---

```
\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  $\langle y\text{-size} \rangle$  (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 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  $\langle y\text{-size} \rangle$  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_wd:Nn      \box_resize_to_wd:Nn <box> {<x-size>}
```

---

Resizes the  $\langle box \rangle$  to  $\langle x\text{-size} \rangle$  (horizontally), scaling the vertical size by the same amount;  $\langle x\text{-size} \rangle$  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  $\langle x\text{-size} \rangle$  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  $\langle x\text{-size} \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_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  $\langle x\text{-size} \rangle$  (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 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  $\langle y\text{-size} \rangle$  results in the  $\langle box \rangle$  having a depth dependent on the height of the original and *vice versa*.

---

```
\box_resize_to_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  $\langle x\text{-size} \rangle$  (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. 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_rotate:Nn
\box_rotate:cn
\box_grotate:Nn
\box_grotate:cn
```

Updated: 2019-01-22

---

```
\box_rotate:Nn <box> {<angle>}
```

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.

---

```
\box_scale:Nnn
\box_scale:cnn
\box_gscale:Nnn
\box_gscale:cnn
```

Updated: 2019-01-22

---

```
\box_scale:Nnn <box> {<x-scale>} {<y-scale>}
```

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*.

## 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`.

# Part XXVIII

## 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).

## 1 Creating and initialising coffins

---

`\coffin_new:N` `\coffin_new:c`  
New: 2011-08-17

`\coffin_clear:N` `\coffin_clear:c`  
`\coffin_gclear:N` `\coffin_gclear:c`  
New: 2011-08-17  
Updated: 2019-01-21

---

Creates a new `\coffin` or raises an error if the name is already taken. The declaration is global. The `\coffin` is initially empty.

`\coffin_clear:N` `\coffin_clear:c`  
`\coffin_gclear:N` `\coffin_gclear:c`

Clears the content of the `\coffin`.

---

`\coffin_set_eq:NN` `\coffin_set_eq:(Nc|cN)`  
`\coffin_set_eq:cc` `\coffin_gset_eq:NN`  
`\coffin_gset_eq:Nc`  
`\coffin_gset_eq:(cN|cc)`

New: 2011-08-17  
Updated: 2019-01-21

---

Sets both the content and poles of `\coffin1` equal to those of `\coffin2`.

---

`\coffin_if_exist_p:N` \* `\coffin_if_exist_p:N` `\coffin_if_exist_p:c` \* `\coffin_if_exist_p:cTF` \* `\coffin_if_exist_p:cTF` \*  
`\coffin_if_exist_p:c` \* `\coffin_if_exist_p:cTF` \* `\coffin_if_exist_p:cTF` \*  
`\coffin_if_exist:NTF` \* `\coffin_if_exist:c` \* `\coffin_if_exist:cTF` \*

Tests whether the `\coffin` is currently defined.

New: 2012-06-20

---

## 2 Setting coffin content and poles

All coffin functions create and manipulate coffins locally within the current `TEX` group level.

---

```
\hcoffin_set:Nn
\hcoffin_set:cn
\hcoffin_gset:Nn
\hcoffin_gset:cn
```

New: 2011-08-17  
Updated: 2011-09-03

---

```
\hcoffin_set:Nn <coffin> {\<material>}
```

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.

---

```
\hcoffin_set:Nw
\hcoffin_set:cw
\hcoffin_set_end:
\hcoffin_gset:Nw
\hcoffin_gset:cw
\hcoffin_gset_end:
```

New: 2011-09-10  
Updated: 2019-01-21

---

```
\hcoffin_set:Nw <coffin> {\<material>} \hcoffin_set_end:
```

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. These functions are useful for setting the entire contents of an coffin.

---

```
\vcoffin_set:Nnn
\vcoffin_set:cnn
\vcoffin_gset:Nnn
\vcoffin_gset:cnn
```

New: 2011-08-17  
Updated: 2019-01-21

---

```
\vcoffin_set:Nnn <coffin> {\<width>} {\<material>}
```

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.

---

```
\vcoffin_set:Nnw
\vcoffin_set:cnw
\vc coffin_set_end:
\vc coffin_gset:Nnw
\vc coffin_gset:cnn
\vc coffin_gset_end:
```

New: 2011-09-10  
Updated: 2019-01-21

---

```
\vcoffin_set:Nnw <coffin> {\<width>} {\<material>} \vcoffin_set_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.

---

```
\coffin_set_horizontal_pole:Nnn
\coffin_set_horizontal_pole:cnn
\coffin_gset_horizontal_pole:Nnn
\coffin_gset_horizontal_pole:cnn
```

New: 2012-07-20  
Updated: 2019-01-21

---

```
\coffin_set_horizontal_pole:Nnn <coffin>
{\<pole>} {\<offset>}
```

Sets the *<pole>* to run horizontally through the *<coffin>*. The *<pole>* is placed at the *<offset>* from the bottom edge of the bounding box 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.

### 3 Joining and using coffins

---

```
\coffin_attach:NnnNnnnn \coffin_attach:NnnNnnnn
\coffin_attach:(cnnNnnnn|Nnncnnnn|cnncnnnn) {\{coffin1-pole1\}} {\{coffin1-pole2\}}
\coffin_gattach:NnnNnnnn \coffin_gattach:(cnnNnnnn|Nnncnnnn|cnncnnnn) {\{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.

---

```
\coffin_join:NnnNnnnn \coffin_join:NnnNnnnn
\coffin_join:(cnnNnnnn|Nnncnnnn|cnncnnnn) {\{coffin1-pole1\}} {\{coffin1-pole2\}}
\coffin_gjoin:NnnNnnnn \coffin_gjoin:(cnnNnnnn|Nnncnnnn|cnncnnnn) {\{coffin2-pole1\}} {\{coffin2-pole2\}}
{\{x-offset\}} {\{y-offset\}}
```

---

Updated: 2019-01-22

This function joins  $\langle coffin_2 \rangle$  to  $\langle coffin_1 \rangle$  such that the bounding box of  $\langle coffin_1 \rangle$  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  $\langle handle_1 \rangle$ , 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.

---

```
\coffin_typeset:Nnnnn {\{coffin\}} {\{pole1\right\rangle\}} {\{pole2\right\rangle\}}
{\{x-offset\}} {\{y-offset\}}
```

---

Updated: 2012-07-20

Typesetting is carried out by first calculating  $\langle handle \rangle$ , the point of intersection of  $\langle pole_1 \rangle$  and  $\langle pole_2 \rangle$ . The coffin is then typeset in horizontal mode such that the relationship between the current reference point in the document and the  $\langle handle \rangle$  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.

## 4 Measuring coffins

---

`\coffin_dp:N`    `\coffin_dp:c`    `\coffin_dp:N <coffin>`  
Calculates the depth (below the baseline) of the `<coffin>` in a form suitable for use in a *(dimension expression)*.

---

`\coffin_ht:N`    `\coffin_ht:c`    `\coffin_ht:N <coffin>`  
Calculates the height (above the baseline) of the `<coffin>` in a form suitable for use in a *(dimension expression)*.

---

`\coffin_wd:N`    `\coffin_wd:c`    `\coffin_wd:N <coffin>`  
Calculates the width of the `<coffin>` in a form suitable for use in a *(dimension expression)*.

## 5 Coffin diagnostics

---

`\coffin_display_handles:Nn`    `\coffin_display_handles:cn`  
Updated: 2011-09-02    `\coffin_display_handles:Nn <coffin> {\color{<color>}}`  
This function first calculates the intersections between all of the `<pole>` of the `<coffin>` to give a set of `<handles>`. It then prints the `<coffin>` at the current location in the source, with the position of the `<handles>` marked on the coffin. The `<handles>` are labelled as part of this process: the locations of the `<handles>` and the labels are both printed in the `<color>` specified.

---

`\coffin_mark_handle:Nnnn`    `\coffin_mark_handle:cnnn`  
Updated: 2011-09-02    `\coffin_mark_handle:Nnnn <coffin> {\pole_1} {\pole_2} {\color{<color>}}`  
This function first calculates the `<handle>` for the `<coffin>` as defined by the intersection of `<pole_1>` and `<pole_2>`. 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:c`  
Updated: 2015-08-01    `\coffin_show_structure:N <coffin>`  
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:c`  
New: 2014-08-22    Updated: 2015-08-01    `\coffin_log_structure:N <coffin>`  
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.

## 6 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`

`\g_tmpb_coffin`

New: 2019-01-24

---

Scratch coffins 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.

# Part XXIX

## The **l3color-base** package

### Color support

This module provides support for color in L<sup>A</sup>T<sub>E</sub>X3. At present, the material here is mainly intended to support a small number of low-level requirements in other l3kernel modules.

## 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_end:  
New: 2011-09-03
```

---

Creates a color group: one used to “trap” color settings.

---

```
\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.

## Part XXX

# The **l3luatex** 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.

## 1 Breaking out to Lua

---

`\lua_now:n` \*

`\lua_now:e` \*

---

New: 2018-06-18

`\lua_now:n {<token list>}`

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`

---

New: 2018-06-18

`\lua_shipout:n {<token list>}`

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:e` \*

---

New: 2015-06-29

`\lua_escape:n {<token list>}`

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.

## 2 Lua interfaces

As well as interfaces for  $\text{\TeX}$ , there are a small number of Lua functions provided here.

**13kernel** All public interfaces provided by the module are stored within the `13kernel` table.

**13kernel.charcat** `13kernel.charcat(<charcode>, <catcode>)`  
Constructs a character of `<charcode>` and `<catcode>` and returns the result to  $\text{\TeX}$ .

**13kernel.elapsedtime** `13kernel.elapsedtime()`  
Returns the time in `<scaled seconds>` since the start of the  $\text{\TeX}$  run or since `13kernel.resettimer` was issued.

**13kernel.filemdfivesum** `13kernel.filemdfivesum(<file>)`  
Returns the of 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  $\text{\TeX}$  behaviour. If the `<file>` is not found, nothing is returned with *no error raised*.

**13kernel.filemoddate** `13kernel.filemoddate(<file>)`  
Returns the of 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*.

**13kernel.filesize** `13kernel.filesize(<file>)`  
Returns the size of the `<file>` in bytes. If the `<file>` is not found, nothing is returned with *no error raised*.

**13kernel.resettimer** `13kernel.resettimer()`  
Resets the timer used by `13kernel.elapsedtime`.

**13kernel.strcmp** `13kernel.strcmp(<str one>, <str two>)`  
Compares the two strings and returns 0 to  $\text{\TeX}$  if the two are identical.

## Part XXXI

# The **I3unicode** package: Unicode support functions

This module provides Unicode-specific functions along with loading data from a range of Unicode Consortium files. At present, it provides no public functions.

# Part XXXII

## The **I3candidates** package

## Experimental additions to **I3kernel**

### 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!

## 2 Additions to l3basics

---

```
\debug_on:n  
\debug_off:n
```

---

New: 2017-07-16  
Updated: 2017-08-02

---

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

Turn on and off within a group various debugging code, some of which is also available 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_resume:
```

---

New: 2017-11-28

---

```
\debug_suspend: ... \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 `l3coffins`.

---

```
\mode_leave_vertical:
```

---

New: 2017-07-04

---

```
\mode_leave_vertical:
```

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 L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  `\leavevmode` approach, no box is used by the method implemented here.

## 3 Additions to **I3box**

### 3.1 Viewing part of a box

---

```
\box_clip:N  
\box_clip:c  
\box_gclip:N  
\box_gclip:c
```

Updated: 2019-01-23

`\box_clip:N <box>`

Clips the `<box>` in the output so that only material inside the bounding box is displayed in the output. The updated `<box>` is an hbox, irrespective of the nature of the `<box>` before the clipping is applied.

These functions require the **LATEX3** native drivers: they do not work with the **LATEX 2 $\epsilon$**  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:cnnnn  
\box_gset_trim:Nnnnn  
\box_gset_trim:cnnnn
```

New: 2019-01-23

`\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.

---

```
\box_set_viewport:Nnnnn  
\box_set_viewport:cnnnn  
\box_gset_viewport:Nnnnn  
\box_gset_viewport:cnnnn
```

New: 2019-01-23

`\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.

## 4 Additions to **I3coffins**

---

```
\coffin_resize:Nnn  
\coffin_resize:cnn  
\coffin_gresize:Nnn  
\coffin_gresize:cnn
```

Updated: 2019-01-23

`\coffin_resize:Nnn <coffin> {<width>} {<total-height>}`

Resized the `<coffin>` to `<width>` and `<total-height>`, both of which should be given as dimension expressions.

---

```
\coffin_rotate:Nn  
\coffin_rotate:cn  
\coffin_grotate:Nn  
\coffin_grotate:cn
```

`\coffin_rotate:Nn <coffin> {<angle>}`

Rotates the `<coffin>` by the given `<angle>` (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:cnn
\coffin_gscale:Nnn
\coffin_gscale:cnn
```

---

Updated: 2019-01-23

`\coffin_scale:Nnn <coffin> {<x-scale>} {<y-scale>}`

Scales the `<coffin>` by a factors `<x-scale>` and `<y-scale>` in the horizontal and vertical directions, respectively. The two scale factors should be given as real numbers.

---

```
\exp_args_generate:n
```

---

New: 2018-04-04

`\exp_args_generate:n {<variant argument specifiers>}`

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, 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.

## 5 Additions to `I3expan`

---

```
logb *
```

---

New: 2018-11-03

`\fp_eval:n { logb( <fpexpr> ) }`

Determines the exponent of the `<fpexpr>`, namely the floor of the base-10 logarithm of its absolute value. “Division by zero” occurs when evaluating  $\log_10(\pm 0) = -\infty$ . Other special values are  $\log_10(\pm \infty) = +\infty$  and  $\log_10(\text{NaN}) = \text{NaN}$ . If the operand is a tuple or is `NaN`, then “invalid operation” occurs and the result is `NaN`.

---

```
\fp_sign:n *
```

---

New: 2018-11-03

`\fp_sign:n {<fpexpr>}`

Evaluates the `<fpexpr>` and leaves its sign in the input stream using `\fp_eval:n {sign(<result>)}`: +1 for positive numbers and for  $+\infty$ , -1 for negative numbers and for  $-\infty$ , ±0 for  $\pm 0$ . If the operand is a tuple or is `NaN`, then “invalid operation” occurs and the result is 0.

## 7 Additions to `I3fpararray`

---

```
\fpararray_new:Nn
```

---

New: 2018-05-05

`\fpararray_new:Nn <fparray var> {<size>}`

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.

---

```
\fpararray_count:N *
```

---

New: 2018-05-05

`\fpararray_count:N <fparray var>`

Expands to the number of entries in the `<floating point array variable>`. This is performed in constant time.

---

```
\fpararray_gset:Nnn
```

---

New: 2018-05-05

`\fpararray_gset:Nnn <fparray var> {<position>} {<value>}`

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 `\fpararray_count:N`, an error occurs. Assignments are always global.

---

`\fpararray_gzero:N`

New: 2018-05-05

`\fpararray_gzero:N <fparray var>`

Sets all entries of the *<floating point array variable>* to +0. Assignments are always global.

---

`\fpararray_item:Nn` \*

`\fpararray_item_to_tl:Nn` \*

New: 2018-05-05

`\fpararray_item:Nn <fparray var> {<position>}`

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 `\fpararray_count:N`, an error occurs.

## 8 Additions to l3file

---

`\file_get_mdfive_hash:nN`

New: 2017-07-11

`\file_get_mdfive_hash:nN {<file name>} <str var>`

Searches for *<file name>* using the current T<sub>E</sub>X search path and the additional paths controlled by `\file_path_include:n`. If found, sets the *<str var>* to the MD5 sum generated from the content of the file. The file is read as bytes, which means that in contrast to most T<sub>E</sub>X 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. Where the file is not found, the *<str var>* will be empty.

---

`\file_get_size:nN`

New: 2017-07-09

`\file_get_size:nN {<file name>} <str var>`

Searches for *<file name>* using the current T<sub>E</sub>X search path and the additional paths controlled by `\file_path_include:n`. If found, sets the *<str var>* to the size of the file in bytes. Where the file is not found, the *<str var>* will be empty. This is not available in older versions of X<sub>E</sub>T<sub>E</sub>X.

---

`\file_get_timestamp:nN`

New: 2017-07-09

`\file_get_timestamp:nN {<file name>} <str var>`

Searches for *<file name>* using the current T<sub>E</sub>X search path and the additional paths controlled by `\file_path_include:n`. If found, sets the *<str var>* to the modification timestamp of the file in the form D:<year><month><day><hour><minute><second><offset>, where the latter may be Z (UTC) or <plus-minus><hours>'<minutes>'. Where the file is not found, the *<str var>* will be empty. This is not available in older versions of X<sub>E</sub>T<sub>E</sub>X.

---

`\file_if_exist_input:n`

`\file_if_exist_input:nF`

New: 2014-07-02

`\file_if_exist_input:n {<file name>}`

`\file_if_exist_input:nF {<file name>} {<false code>}`

Searches for *<file name>* using the current T<sub>E</sub>X search path and the additional paths controlled by `\file_path_include:n`. If found then reads in the file as additional L<sup>A</sup>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:**

---

New: 2017-07-07

\file\_input\_stop:

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.

**TeXhackers note:** This function must be used on a line on its own: TeX 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!

---

**\ior\_map\_variable:NNn**

---

New: 2019-01-13

\ior\_map\_variable:NNn *stream* {*tl var*} {{*code*}}

For each set of *lines* obtained by calling `\ior_get:NN` until reaching the end of the file, stores the *lines* in the *tl var* then applies the *code*. The *code* will usually make use of the *variable*, but this is not enforced. The assignments to the *variable* are local. TeX ignores any trailing new-line marker from the file it reads. This function is typically faster than `\ior_map_inline:Nn`.

---

**\ior\_str\_map\_variable:NNn**

---

New: 2019-01-13

\ior\_str\_map\_variable:NNn *stream* {*tl var*} {{*code*}}

For each *line* in the *stream*, stores the *line* in the *tl var* 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. 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`.

---

**\iow\_allow\_break:**

---

New: 2018-12-29

\iow\_allow\_break:

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.

## 9 Additions to I3flag

---

**\flag\_raise\_if\_clear:n \***

---

New: 2018-04-02

\flag\_raise\_if\_clear:n {{*flag name*}}

Ensures the *flag* is raised by making its height at least 1, locally.

## 10 Additions to I3int

---

**\int\_sign:n \***

---

New: 2018-11-03

\int\_sign:n {{*intexpr*}}

Evaluates the *integer expression* then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

## 11 Additions to **I3intarray**

---

```
\intarray_gset_rand:Nnn
\intarray_gset_rand:Nn
```

---

New: 2018-05-05

```
\intarray_gset_rand:Nnn <intarray var> {<minimum>} {<maximum>}
\intarray_gset_rand:Nn <intarray var> {<maximum>}
```

Evaluates the integer expressions  $\langle \text{minimum} \rangle$  and  $\langle \text{maximum} \rangle$  then sets each entry (independently) of the  $\langle \text{integer array variable} \rangle$  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  $\langle \text{minimum} \rangle$  is 1. Assignments are always global. This is not available in older versions of Xe<sup>T</sup>EX.

### 11.1 Working with contents of integer arrays

---

```
\intarray_const_from_clist:Nn ☆ \intarray_const_from_clist:Nn <intarray var> <intexpr clist>
```

---

New: 2018-05-04

Creates a new constant  $\langle \text{integer array variable} \rangle$  or raises an error if the name is already taken. The  $\langle \text{integer array variable} \rangle$  is set (globally) to contain as its items the results of evaluating each  $\langle \text{integer expression} \rangle$  in the  $\langle \text{comma list} \rangle$ .

---

```
\intarray_to_clist:N ☆ \intarray_to_clist:N <intarray var>
```

---

New: 2018-05-04

Converts the  $\langle \text{intarray} \rangle$  to integer denotations separated by commas. All tokens have category code other. If the  $\langle \text{intarray} \rangle$  has no entry the result is empty; otherwise the result has one fewer comma than the number of items.

---

```
\intarray_show:N
\intarray_log:N
```

---

New: 2018-05-04

```
\intarray_show:N <intarray var>
\intarray_log:N <intarray var>
```

Displays the items in the  $\langle \text{integer array variable} \rangle$  in the terminal or writes them in the log file.

## 12 Additions to **I3msg**

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:nnnnnn` 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, the message text and arguments are not expanded, 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. Another particularity of expandable messages is that they cannot be redirected or turned off by the user.

---

```
\msg_expandable_error:nnnnn ★ \msg_expandable_error:nnnnnn {⟨module⟩} {⟨message⟩} {⟨arg one⟩} {⟨arg two⟩} {⟨arg three⟩} {⟨arg four⟩}
\msg_expandable_error:nnffff ★
\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

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.

---

```
\msg_show_eval:Nn \msg_show_eval:Nn ⟨function⟩ {⟨expression⟩}
```

---

Shows or logs the ⟨expression⟩ (turned into a string), an equal sign, and the result of applying the ⟨function⟩ to the {⟨expression⟩} (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:nnnnnn \msg_show:nnxxxx \msg_show:nnnnn \msg_show:nnxxx \msg_show:nnnn \msg_show:nnxx \msg_show:nnn \msg_show:nnx \msg_show:nn
```

---

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 `\tl_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

---

```
\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.

## 13 Additions to `I3prg`

---

```
\bool_const:Nn \bool_const:Nn ⟨boolean⟩ {⟨boolexpr⟩}
```

---

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⟩.

---

```
\bool_set_inverse:N  
\bool_set_inverse:c  
\bool_gset_inverse:N  
\bool_gset_inverse:c
```

New: 2018-05-10

---

```
\bool_set_inverse:N <boolean>
```

Toggles the *<boolean>* from `true` to `false` and conversely: sets it to the inverse of its current value.

---

```
\prop_count:N ★  
\prop_count:c ★
```

```
\prop_count:N <property list>
```

Leaves the number of key–value pairs in the *<property list>* in the input stream as an *(integer denotation)*.

---

```
\prop_map_tokens:Nn ★  
\prop_map_tokens:cn ★
```

```
\prop_map_tokens:Nn <property list> {<code>}
```

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_rand_key_value:N ★  
\prop_rand_key_value:c ★
```

New: 2016-12-06

---

```
\prop_rand_key_value:N <prop var>
```

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 X<sub>E</sub>T<sub>X</sub>.

**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 argument expansion.

---

```
\prop_set_from_keyval:Nn  
\prop_set_from_keyval:cn  
\prop_gset_from_keyval:Nn  
\prop_gset_from_keyval:cn
```

New: 2017-11-28

---

```
\prop_set_from_keyval:Nn <prop var>  
{  
<key1> = <value1> ,  
<key2> = <value2> , ...  
}
```

Sets *<prop var>* to contain key–value pairs given in the second argument.

---

```
\prop_const_from_keyval:Nn  
\prop_const_from_keyval:cn
```

New: 2017-11-28

---

```
\prop_const_from_keyval:Nn <prop var>  
{  
<key1> = <value1> ,  
<key2> = <value2> , ...  
}
```

Creates a new constant *<prop var>* or raises an error if the name is already taken. The *<prop var>* is set globally to contain key–value pairs given in the second argument.

## 15 Additions to `I3seq`

---

`\seq_mapthread_function:NNN` ☆  
`\seq_mapthread_function:(NcN|cNN|ccN)` ☆

`\seq_mapthread_function:NNN <seq1> <seq2> <function>`

Applies *<function>* to every pair of items *<seq<sub>1</sub>-item>*–*<seq<sub>2</sub>-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_gset_filter:NNn`

`\seq_set_filter:NNn <sequence1> <sequence2> {<inline boolexpr>}`

Evaluates the *<inline boolexpr>* for every *<item>* stored within the *<sequence<sub>2</sub>>*. The *<inline boolexpr>* receives the *<item>* as #1. The sequence of all *<items>* for which the *<inline boolexpr>* evaluated to `true` is assigned to *<sequence<sub>1</sub>>*.

**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:NNn`  
`\seq_gset_map:NNn`

New: 2011-12-22

`\seq_set_map:NNn <sequence1> <sequence2> {<inline function>}`

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.

**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_const_from_clist:Nn`  
`\seq_const_from_clist:cn`

New: 2017-11-28

`\seq_const_from_clist:Nn <seq var> {<comma-list>}`

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_from_function:NnN`  
`\seq_gset_from_function:NnN`

New: 2018-04-06

`\seq_set_from_function:NnN <seq var> {<loop code>} <function>`

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:nnN {<initial value>} {<step>} {<final value>}`, but not the analogous “inline” mappings.

---

```
\seq_shuffle:N <seq var>
```

---

```
\seq_gshuffle:N
```

New: 2018-04-29

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.

**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.

---

```
\seq_indexed_map_function:NN \seq_indexed_map_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_indexed_map_inline:Nn \seq_indexed_map_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.

## 16 Additions to l3skip

---

```
\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.

---

```
\skip_split_finite_else_action:nnNN \skip_split_finite_else_action:nnNN {<skipexpr>} {<action>}
                                         <dimen1> <dimen2>
```

---

Checks if the *<skipexpr>* contains finite glue. If it does then it assigns *<dimen1>* the stretch component and *<dimen2>* the shrink component. If it contains infinite glue set *<dimen1>* and *<dimen2>* to 0pt and place #2 into the input stream: this is usually an error or warning message of some sort.

## 17 Additions to l3sys

---

```
\sys_shell_get:nnN
\sys_shell_get:nnNTF
New: 2019-01-16
```

---

Defines  $\langle tl \rangle$  to the text returned by the  $\langle shell command \rangle$ . The  $\langle shell command \rangle$  is converted to a string using `\tl_to_str:n`. Category codes may need to be set appropriately via the  $\langle setup \rangle$  argument, which is run just before running the  $\langle shell command \rangle$  (in a group). If shell escape is disabled, the  $\langle tl var \rangle$  will be empty. Note that quote characters ("") cannot be used inside the  $\langle shell command \rangle$ . The `\sys_shell_get:nnNTF` conditional returns `true` if the shell is available and no quote is detected, and `false` otherwise.

---

```
\c_sys_engine_version_str
New: 2018-05-02
```

---

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

$\langle major \rangle.\langle minor \rangle.\langle revision \rangle$

For XeTeX, the form is

$\langle major \rangle.\langle minor \rangle$

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.

$\langle major \rangle.\langle minor \rangle.\langle revision \rangle-u\langle major \rangle.\langle minor \rangle-\langle epTeX \rangle$

where the u part is only present for upTeX.

---

```
\sys_if_rand_exist_p: *
\sys_if_rand_exist:TF *
New: 2017-05-27
```

---

`\sys_if_rand_exist_p:`  
`\sys_if_rand_exist:TF` { $\langle true code \rangle$ } { $\langle false code \rangle$ }

Tests if the engine has a pseudo-random number generator. Currently this is the case in pdfTeX, LuaTeX, pTeX and upTeX.

---

```
\sys_rand_seed: *
New: 2017-05-27
```

---

`\sys_rand_seed:`

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
New: 2017-05-27
```

---

`\sys_gset_rand_seed:n` { $\langle intexpr \rangle$ }

Globally sets the seed for the engine's pseudo-random number generator to the  $\langle integer expression \rangle$ . 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  $[0, 2^{28} - 1]$ .

---

```
\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
```

---

New: 2018-07-27

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

---

```
\c_sys_shell_escape_int
```

---

New: 2017-05-27

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 *
```

---

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 *
```

---

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 *
```

---

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:{tokens}
```

---

Execute *{tokens}* through shell escape immediately.

New: 2017-05-27

---

```
\sys_shell_shipout:n \sys_shell_shipout:{tokens}
```

---

Execute *{tokens}* through shell escape at shipout.

New: 2017-05-27

## 18 Additions to `\3tl`

---

`\tl_if_single_token_p:n` ★  
`\tl_if_single_token:nTF` ★

---

`\tl_if_single_token_p:n {<token list>}`  
`\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_reverse_tokens:n` ★

---

`\tl_reverse_tokens:n {<tokens>}`

This function, which works directly on TeX tokens, reverses the order of the `<tokens>`: the first becomes the last and the last becomes first. Spaces are preserved. The reversal also operates within brace groups, but the braces themselves are not exchanged, as this would lead to an unbalanced token list. For instance, `\tl_reverse_tokens:n {a~{b()}}` leaves `{()}~a` in the input stream. This function requires two steps of expansion.

**TeXhackers note:** The result is returned within the `\unexpanded` primitive (`\exp_not:n`), which means that the token list does not expand further when appearing in an `x`-type argument expansion.

---

`\tl_count_tokens:n` ★

---

`\tl_count_tokens:n {<tokens>}`

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. This function requires three expansions, giving an `<integer denotation>`.

---

`\tl_lower_case:n` ★  
`\tl_upper_case:n` ★  
`\tl_mixed_case:n` ★  
`\tl_lower_case:nn` ★  
`\tl_upper_case:nn` ★  
`\tl_mixed_case:nn` ★

---

`\tl_upper_case:n {<tokens>}`  
`\tl_upper_case:nn {<language>} {<tokens>}`

These functions are intended to be applied to input which may be regarded broadly as “text”. They traverse the `<tokens>` and change the case of characters as discussed below. The character code of the characters replaced may be arbitrary: the replacement characters have standard document-level category codes (11 for letters, 12 for letter-like characters which can also be case-changed). Begin-group and end-group characters in the `<tokens>` are normalized and become `{` and `}`, respectively.

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_lower_case:n`, `\str_upper_case:n` and `\str_fold_case:n`.

The functions perform expansion on the input in most cases. In particular, input in the form of token lists or expandable functions is expanded *unless* it falls within one of the special handling classes described below. This expansion approach means that in general the result of case changing matches the “natural” outcome expected from a “functional” approach to case modification. For example

```
\tl_set:Nn \l_tmpa_tl { hello }
\tl_upper_case:n { \l_tmpa_tl \c_space_tl world }
```

produces

```
HELLO WORLD
```

The expansion approach taken means that in package mode any L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  “robust” commands which may appear in the input should be converted to engine-protected versions using for example the `\robustify` command from the `etoolbox` package.

---

`\l_t1_case_change_math_t1` Case changing does not take place within math mode material so for example

```
\tl_upper_case:n { Some-text-$y = mx + c$~with~{Braces} }
```

becomes

```
SOME TEXT $y = mx + c$ WITH {BRACES}
```

Material inside math mode is left entirely unchanged: in particular, no expansion is undertaken.

Detection of math mode is controlled by the list of tokens in `\l_t1_case_change_math_t1`, which should be in open–close pairs. In package mode the standard settings is

```
$ $ \(\ )
```

Note that while expansion occurs when searching the text it does not apply to math mode material (which should be unaffected by case changing). As such, whilst the opening token for math mode may be “hidden” inside a command/macro, the closing one cannot be as this is being searched for in math mode. Typically, in the types of “text” the case changing functions are intended to apply to this should not be an issue.

---

`\l_t1_case_change_exclude_t1`

Case changing can be prevented by using any command on the list `\l_t1_case_change_exclude_t1`. Each entry should be a function to be followed by one argument: the latter will be preserved as-is with no expansion. Thus for example following

```
\tl_put_right:Nn \l_t1_case_change_exclude_t1 { \NoChangeCase }
```

the input

```
\tl_upper_case:n
  { Some-text-$y = mx + c$~with~\NoChangeCase {Protection} }
```

will result in

```
SOME TEXT $y = mx + c$ WITH \NoChangeCase {Protection}
```

Notice that the case changing mapping preserves the inclusion of the escape functions: it is left to other code to provide suitable definitions (typically equivalent to `\use:n`). In particular, the result of case changing is returned protected by `\exp_not:n`.

When used with L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  the commands `\cite`, `\ensuremath`, `\label` and `\ref` are automatically included in the list for exclusion from case changing.

---

### `\l_t1_case_change_accents_t1`

This list specifies accent commands which should be left unexpanded in the output. This allows for example

```
\t1_upper_case:n { \" { a } }
```

to yield

```
\" { A }
```

irrespective of the expandability of `\"`.

The standard contents of this variable is `\"`, `\'`, `\.,`, `\^,`, `\``, `\~`, `\c,`, `\H,`, `\k,`, `\r,`, `\t,`, `\u` and `\v`.

“Mixed” case conversion may be regarded informally as converting the first character of the *(tokens)* to upper case and the rest to lower case. However, the process is more complex than this as there are some situations where a single lower case character maps to a special form, for example `ij` in Dutch which becomes `IJ`. As such, `\t1_mixed_case:n(n)` implement a more sophisticated mapping which accounts for this and for modifying accents on the first letter. Spaces at the start of the *(tokens)* are ignored when finding the first “letter” for conversion.

```
\t1_mixed_case:n { hello-WORLD } % => "Hello world"  
\t1_mixed_case:n { ~hello-WORLD } % => " Hello world"  
\t1_mixed_case:n { {hello}~WORLD } % => "{Hello} world"
```

When finding the first “letter” for this process, any content in math mode or covered by `\l_t1_case_change_exclude_t1` is ignored.

(Note that the Unicode Consortium describe this as “title case”, but that in English title case applies on a word-by-word basis. The “mixed” case implemented here is a lower level concept needed for both “title” and “sentence” casing of text.)

---

### `\l_t1_mixed_case_ignore_t1`

The list of characters to ignore when searching for the first “letter” in mixed-casing is determined by `\l_t1_mixed_change_ignore_t1`. This has the standard setting

```
( [ { ‘ -
```

where comparisons are made on a character basis.

As is generally true for `expl3`, these functions are designed to work with Unicode input only. As such, UTF-8 input is assumed for *all* engines. When used with X<sub>E</sub>T<sub>X</sub> or L<sub>A</sub>T<sub>E</sub>X a full range of Unicode transformations are enabled. Specifically, the standard mappings here follow those defined by the [Unicode Consortium](#) in `UnicodeData.txt` and `SpecialCasing.txt`. In the case of 8-bit engines, mappings are provided for characters which can be represented in output typeset using the T1 font encoding. Thus for example `ä` can be case-changed using pdfT<sub>E</sub>X. For p<sub>T</sub>E<sub>X</sub> only the ASCII range is covered as the engine treats input outside of this range as east Asian.

Context-sensitive mappings are enabled: language-dependent cases are discussed below. Context detection expands input but treats any unexpandable control sequences as “failures” to match a context.

Language-sensitive conversions are enabled using the *(language)* argument, and follow Unicode Consortium guidelines. Currently, the languages recognised for special handling are as follows.

- 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 lower casing I-dot and introduced when upper casing i-dotless.
- German (**de-alt**). An alternative mapping for German in which the lower case *Eszett* maps to a *großes Eszett*.
- Lithuanian (**lt**). The lower case letters i and j should retain a dot above when the accents grave, acute or tilde are present. This is implemented for lower casing of the relevant upper case letters both when input as single Unicode codepoints and when using combining accents. The combining dot is removed when upper casing in these cases. Note that *only* the accents used in Lithuanian are covered: the behaviour of other accents are not modified.
- Dutch (**n1**). Capitalisation of **ij** at the beginning of mixed cased input produces **IJ** rather than **Ij**. The output retains two separate letters, thus this transformation *is* available using pdfTEX.

Creating additional context-sensitive mappings requires knowledge of the underlying mapping implementation used here. The team are happy to add these to the kernel where they are well-documented (*e.g.* in Unicode Consortium or relevant government publications).

---

\tl_range_braced:Nnn	★ \tl_range_braced:Nnn <i>tl var</i> { <i>start index</i> } { <i>end index</i> }
\tl_range_braced:cnn	★ \tl_range_braced:nnn { <i>token list</i> } { <i>start index</i> } { <i>end index</i> }
\tl_range_braced:nnn	★ \tl_range_unbraced:Nnn <i>tl var</i> { <i>start index</i> } { <i>end index</i> }
\tl_range_unbraced:Nnn	★ \tl_range_unbraced:nnn { <i>token list</i> } { <i>start index</i> } { <i>end index</i> }
\tl_range_unbraced:cnn	★ \tl_range_unbraced:nnn { <i>token list</i> } { <i>start index</i> } { <i>end index</i> }
\tl_range_unbraced:nnn	★

New: 2017-07-15

---

Leaves in the input stream the items from the *start index* to the *end index* inclusive, using the same indexing as \tl\_range:nnn. Spaces are ignored. Regardless of whether items appear with or without braces in the *token list*, 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 bcde{}, cde{}f, e{}f, 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 *item* does not expand further when appearing in an x-type argument expansion.

---

### \tl\_build\_begin:N *tl var*

Clears the *tl var* 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 *tl var* respectively.

---

### \tl\_build\_clear:N *tl var*

Clears the *tl var* and sets it up to support other \tl\_build\_... functions. The clear and gclear functions must be used for local and global *tl var* respectively.

---

```
\tl_build_put_left:Nn
\tl_build_put_left:Nx
\tl_build_gput_left:Nn
\tl_build_gput_left:Nx
\tl_build_put_right:Nn
\tl_build_put_right:Nx
\tl_build_gput_right:Nn
\tl_build_gput_right:Nx
```

---

New: 2018-04-01

```
\tl_build_put_left:Nn <tl var> {<tokens>}
\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.

---

```
\tl_build_get:NN
```

---

New: 2018-04-01

```
\tl_build_get:N <tl var1> <tl var2>
```

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_gend:N
```

---

New: 2018-04-01

```
\tl_build_end:N <tl var>
```

Gets the contents of *<tl var>* and stores that into the *<tl var>* using `\tl_set: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.

## 19 Additions to `I3token`

---

```
\c_catcode_active_space_tl
```

---

New: 2017-08-07

Token list containing one character with category code 13, (“active”), and character code 32 (space).

---

```
\char_lower_case:N *
```

---

```
\char_upper_case:N *
```

---

```
\char_mixed_case:N *
```

---

```
\char_fold_case:N *
```

---

New: 2018-04-06

```
\char_lower_case:N <char>
```

Converts the *<char>* to the equivalent case-changed character as detailed by the function name (see `\str_fold_case:n` and `\tl_mixed_case:n` for details of these terms). The case mapping is carried out with no context-dependence (*cf.* `\tl_upper_case:n`, *etc.*)

---

```
\char_codepoint_to_bytes:n *
```

---

```
\char_codepoint_to_bytes:n {<codepoint>}
```

---

New: 2018-06-01

Converts the (Unicode) *<codepoint>* 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.

---

**\peek\_N\_type:TF**

Updated: 2012-12-20

```
\peek_N_type:TF {\<true code>} {\<false code>}
```

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_catcode_collect_inline:Nn
\peek_charcode_collect_inline:Nn
\peek_meaning_collect_inline:Nn
```

New: 2018-09-23

```
\peek_catcode_collect_inline:Nn {\<test token>} {\<inline code>}
\peek_charcode_collect_inline:Nn {\<test token>} {\<inline code>}
\peek_meaning_collect_inline:Nn {\<test token>} {\<inline code>}
```

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} }
```

---

**\peek\_remove\_spaces:n**

New: 2018-10-01

```
\peek_remove_spaces:n {\<code>}
```

Removes explicit and implicit space tokens (category code 10 and character code 32) from the input stream, then inserts *<code>*.

## Part XXXIII

# The **I3drivers** package

## Drivers

$\text{\TeX}$  relies on drivers in order to carry out a number of tasks, such as using color, including graphics and setting up hyper-links. The nature of the code required depends on the exact driver in use. Currently,  $\text{\LaTeX}3$  is aware of the following drivers:

- **pdfmode**: The “driver” for direct PDF output by *both*  $\text{pdf\TeX}$  and  $\text{Lua\TeX}$  (no separate driver is used in this case: the engine deals with PDF creation itself).
- **dvips**: The  $\text{dvips}$  program, which works in conjugation with  $\text{pdf\TeX}$  or  $\text{Lua\TeX}$  in DVI mode.
- **dvipdfmx**: The  $\text{dvipdfmx}$  program, which works in conjugation with  $\text{pdf\TeX}$  or  $\text{Lua\TeX}$  in DVI mode.
- **dvisvgm**: The  $\text{dvisvgm}$  program, which works in conjugation with  $\text{pdf\TeX}$  or  $\text{Lua\TeX}$  when run in DVI mode as well as with (u) $\text{p\TeX}$  and  $\text{X\TeX}$ .
- **xdvipdfmx**: The driver used by  $\text{X\TeX}$ .

This module provides code closely tied to the exact driver in use: broadly, the functions here are implemented entirely independently for each case. As such, they often rely on higher-level code to provide necessary but shared operations. For example, in box rotation and scaling the functions here do no correct the final size of the box: this will always be required and thus is handled in the `box` module.

Several of the operations here are low-level, and so may be used only in restricted contexts. Some also require understanding of PostScript/PDF concepts to be used correctly as they take “raw” arguments, similar in format to those used by the underlying driver.

The functions in this module should be regarded as experimental with the following exceptions:

- ...

### 1 Box clipping

---

`\driver_box_use_clip:N`

New: 2017-12-13

---

`\driver_box_use_clip:N <box>`

Inserts the content of the  $\langle box \rangle$  at the current insertion point such that any material outside of the bounding box is not displayed by the driver. The material in the  $\langle box \rangle$  is still placed in the output stream: the clipping takes place at a driver level.

## 2 Box rotation and scaling

---

\driver\_box\_use\_rotate:Nn

New: 2017-12-13  
Updated: 2018-04-26

\driver\_box\_use\_rotate:Nn <box> {<angle>}

Inserts the content of the `<box>` at the current insertion point rotated by the `<angle>` (an *fp expression*) expressed in degrees). The material is rotated such the the T<sub>E</sub>X reference point of the box is the center of rotation and remains the reference point after rotation. It is the responsibility of the code using this function to adjust the apparent size of the inserted material.

---

\driver\_box\_use\_scale:Nnn

New: 2017-12-13  
Updated: 2018-04-26

\driver\_box\_use\_scale:Nnn <box> {<x-scale>} {<y-scale>}

Inserts the content of the `<box>` at the current insertion point scale by the `<x-scale>` and `<y-scale>` (both *fp expressions*). The reference point of the material will be unchanged. It is the responsibility of the code using this function to adjust the apparent size of the inserted material.

## 3 Color support

---

\driver\_color\_cmyk:nnnn

New: 2018-02-20

\driver\_color\_cmyk:nnnn {<cyan>} {<magenta>} {<yellow>} {<black>}

Sets the color to the CMYK values specified, all of which are fp denotations in the range 0 and 1. For drawing colors, see \driver\_draw\_stroke\_cmyk:nnnn, *etc.*

---

\driver\_color\_gray:n

New: 2018-02-20

\driver\_color\_gray:n {<gray>}

Sets the color to the grayscale value specified, which is fp denotations in the range 0 and 1. For drawing colors, see \driver\_draw\_stroke\_gray:n, *etc.*

---

\driver\_color\_rgb:nnn

New: 2018-02-20

\driver\_color\_rgb:nnn {<red>} {<green>} {<blue>}

Sets the color to the RGB values specified, all of which are fp denotations in the range 0 and 1. For drawing colors, see \driver\_draw\_stroke\_rgb:nnn, *etc.*

---

\driver\_color\_pickup:N

New: 2018-02-20

\driver\_color\_pickup:N <tl>

In L<sub>A</sub>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package mode, collects data on the current color from \current@color and stores it in the low-level format used by expl3 in the `<tl>`.

## 4 Drawing

The drawing functions provided here are *highly* experimental. They are inspired heavily by the system layer of pgf (most have the same interface as the same functions in the latter's \pgf<sub>sys</sub>@... namespace). They are intended to form the basis for higher level drawing interfaces, which themselves are likely to be further abstracted for user access. Again, this model is heavily inspired by pgf and Tikz.

These low level drawing interfaces abstract from the driver raw requirements but still require an appreciation of the concepts of PostScript/PDF/SVG graphic creation.

---

```
\driver_draw_begin:  
\driver_draw_end:
```

\driver\_draw\_begin:  
(*content*)  
\driver\_draw\_end:  
Defines a drawing environment. This is a scope for the purposes of the graphics state. Depending on the driver, other set up may or may not take place here. The natural size of the *(content)* should be zero from the TeX perspective: allowance for the size of the content must be made at a higher level (or indeed this can be skipped if the content is to overlap other material).

---

```
\driver_draw_scope_begin:  
\driver_draw_scope_end:
```

\driver\_draw\_scope\_begin:  
(*content*)  
\driver\_draw\_scope\_end:  
Defines a scope for drawing settings and so on. Changes to the graphic state and concepts such as color or linewidth are localised to a scope. This function pair must never be used if an partial path is under construction: such paths must be entirely contained at one unbroken scope level. Note that scopes do not form TeX groups and may not be aligned with them.

## 4.1 Path construction

---

```
\driver_draw_moveto:nn
```

\driver\_draw\_move:nn {*x*} {*y*}  
Moves the current drawing reference point to (*x*, *y*); any active transformation matrix applies.

---

```
\driver_draw_lineto:nn
```

\driver\_draw\_lineto:nn {*x*} {*y*}  
Adds a path from the current drawing reference point to (*x*, *y*); any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

---

```
\driver_draw_curveto:nnnnnn \driver_draw_curveto:mnnnnn {x1} {y1}  
{x2} {y2} {x3} {y3}
```

Adds a Bezier curve path from the current drawing reference point to (*x*<sub>3</sub>, *y*<sub>3</sub>), using (*x*<sub>1</sub>, *y*<sub>1</sub>) and (*x*<sub>2</sub>, *y*<sub>2</sub>) as control points; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

---

```
\driver_draw_rectangle:nnnn \driver_draw_rectangle:nnnn {x} {y} {width} {height}
```

Adds rectangular path from (*x*<sub>1</sub>, *y*<sub>1</sub>) of *height* and *width*; any active transformation matrix applies. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

---

```
\driver_draw_closepath: \driver_draw_closepath:
```

Closes an existing path, adding a line from the current point to the start of path. Note that nothing is drawn until a fill or stroke operation is applied, and that the path may be discarded or used as a clip without appearing itself.

## 4.2 Stroking and filling

---

```
\driver_draw_stroke:  
\driver_draw_closestroke:
```

*(path construction)*  
\driver\_draw\_stroke:  
Draws a line along the current path, which is also closed in the case of \driver\_draw\_closestroke:. The nature of the line drawn is influenced by settings for

- Line thickness
- Stroke color (or the current color if no specific stroke color is set)
- Line capping (how non-closed line ends should look)
- Join style (how a bend in the path should be rendered)
- Dash pattern

The path may also be used for clipping.

---

```
\driver_draw_fill:  
\driver_draw_fillstroke:
```

*(path construction)*  
\driver\_draw\_fill:  
Fills the area surrounded by the current path: this will be closed prior to filling if it is not already. The `fillstroke` version also strokes the path as described for \driver\_draw\_stroke:. The fill is influenced by the setting for fill color (or the current color if no specific stroke color is set). The path may also be used for clipping. For paths which are self-intersecting or comprising multiple parts, the determination of which areas are inside the path is made using the non-zero winding number rule unless the even-odd rule is active.

---

```
\driver_draw_nonzero_rule:  
\driver_draw_evenodd_rule:
```

`\driver_draw_nonzero_rule:`  
Active either the non-zero winding number or the even-odd rule, respectively, for determining what is inside a fill or clip area. For technical reasons, these command are not influenced by scoping and apply on an ongoing basis.

---

```
\driver_draw_clip:  
\driver_draw_clip:
```

*(path construction)*  
\driver\_draw\_clip:  
Indicates that the current path should be used for clipping, such that any subsequent material outside of the path (but within the current scope) will not be shown. This command should be given once a path is complete but before it is stroked or filled (if appropriate). This command is *not* affected by scoping: it applies to exactly one path as shown.

---

```
\driver_draw_discardpath:
```

*(path construction)*  
\driver\_draw\_discardpath:  
Discards the current path without stroking or filling. This is primarily useful for paths constructed purely for clipping, as this alone does not end the paths existence.

### 4.3 Stroke options

---

```
\driver_draw_linewidth:n \driver_draw_linewidth:n {\dimexpr}
```

Sets the width to be used for stroking to  $\langle\dimexpr\rangle$ .

---

```
\driver_draw_dash_pattern:nn \driver_draw_dash:nn {\dash pattern} {\phase}
```

Sets the pattern of dashing to be used when stroking a line. The  $\langle\text{dash pattern}\rangle$  should be a comma-separated list of dimension expressions. This is then interpreted as a series of pairs of line-on and line-off lengths. For example 3pt, 4pt means that 3 pt on, 4 pt off, 3 pt on, and so on. A more complex pattern will also repeat: 3pt, 4pt, 1pt, 2pt results in 3 pt on, 4 pt off, 1 pt on, 2 pt off, 3 pt on, and so on. An odd number of entries means that the last is repeated, for example 3pt is equal to 3pt, 3pt. An empty pattern yields a solid line.

The  $\langle\text{phase}\rangle$  specifies an offset at the start of the cycle. For example, with a pattern 3pt a phase of 1pt means that the output is 2 pt on, 3 pt off, 3 pt on, 3 pt on, etc.

---

```
\driver_draw_cap_butt: \driver_draw_cap_butt:  
\driver_draw_cap_rectangle:  
\driver_draw_cap_round:
```

Sets the style of terminal stroke position to one of butt, rectangle or round.

---

```
\driver_draw_join_bevel: \driver_draw_cap_butt:
```

Sets the style of stroke joins to one of bevel, miter or round.

---

```
\driver_draw_miterlimit:n \driver_draw_miterlimit:n {\factor}
```

Sets the miter limit of lines joined as a miter, as described in the PDF and PostScript manuals. The  $\langle\text{factor}\rangle$  here is an  $\langle\text{fp expression}\rangle$ .

### 4.4 Color

---

```
\driver_draw_color_fill_cmyk:nnnn \driver_draw_color_fill_cmyk:nnnn {\cyan} {\magenta} {\yellow}  
\driver_draw_color_stroke_cmyk:nnnn {\black}
```

Sets the color for drawing to the CMYK values specified, all of which are fp denotations in the range 0 and 1.

---

```
\driver_draw_color_fill_gray:n \driver_draw_color_fill_gray:n {\gray}  
\driver_draw_color_stroke_gray:n
```

Sets the color for drawing to the grayscale value specified, which is fp denotations in the range 0 and 1.

---

```
\driver_draw_color_fill_rgb:nnn \driver_draw_color_fill_rgb:nnn {\red} {\green} {\blue}  
\driver_draw_color_stroke_rgb:nnn
```

Sets the color for drawing to the RGB values specified, all of which are fp denotations in the range 0 and 1.

## 4.5 Inserting T<sub>E</sub>X material

---

`\driver_draw_box_use:Nnnnn`

```
\driver_draw_box:Nnnnnnnn <box>
{<a>} {<b>} {<c>} {<d>} {<x>} {<y>}
```

Inserts the  $\langle box \rangle$  as an hbox with the box reference point placed at  $(x, y)$ . The transformation matrix  $[abcd]$  is applied to the box, allowing it to be in synchronisation with any scaling, rotation or skewing applying more generally. Note that T<sub>E</sub>X material should not be inserted directly into a drawing as it would not be in the correct location. Also note that as for other drawing elements the box here has no size from a T<sub>E</sub>X perspective.

## 4.6 Coordinate system transformations

---

`\driver_draw_cm:nnnn`

```
\driver_draw_cm:nnnn {<a>} {<b>} {<c>} {<d>}
```

Applies the transformation matrix  $[abcd]$  to the current graphic state. This affects any subsequent items in the same scope but not those already given.

# 5 PDF Features

A range of PDF features are exposed by pdfT<sub>E</sub>X and LuaT<sub>E</sub>X in direct PDF output mode, and the vast majority of these are also controllable using the (x)dvipdfmx driver (as DVI instructions are converted directly to PDF). Some of these functions are also available for cases where PDFs are generated by dvips: this depends on being able to pass information through correctly.

## 5.1 PDF Objects

Objects are used to provide a range of data structures in a PDF. At the driver level, different PDF object types are declared separately. Objects are only *written* to the PDF when referenced.

---

`\driver_pdf_object_new:n`

```
\driver_pdf_object_new:n {<name>} {<type>}
```

Declares  $\langle name \rangle$  as a PDF object. The `type` should be one of `array` or `dict`, `fstream` or `stream`.

---

`\driver_pdf_object_ref:n *`

```
\driver_pdf_object_ref:n {<object>}
```

Inserts the appropriate information to reference the  $\langle object \rangle$  in for example page resource allocation.

---

```
\driver_pdf_object_write:nn \driver_pdf_object_write:nn {⟨name⟩} {⟨data⟩}
```

Writes the ⟨data⟩ 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: ⟨content⟩ and ⟨file name⟩

**stream** Two brace groups: ⟨content⟩ and ⟨additional attributes⟩

## 5.2 PDF structure

---

```
\driver_pdf_compresslevel:n \driver_pdf_compresslevel:n {⟨level⟩}
```

Sets the degree of compression used for PDF files: the ⟨level⟩ should be in the range 0 to 9 (higher is more compression). Typically, either compression is disabled (0) or maximised (9). When used with (x)dvipdfmx, this setting may only be applied globally: it should be set only once.

---

```
\driver_pdf_objects_enable: \driver_pdf_objects_disable:  
\driver_pdf_objects_disable:
```

Enables or disables the creation of PDF objects. These objects are used to reduce the size of PDFs, and typically are enabled as standard. When used with (x)dvipdfmx, object creation can be disabled but not re-enabled, and this setting may only be applied globally: it should be set only once.

# Part XXXIV

# Implementation

## 1 l3bootstrap implementation

1 ⟨\*initex | package⟩  
2 ⟨@@=kernel⟩

### 1.1 Format-specific code

The very first thing to do is to bootstrap the iniTeX system so that everything else will actually work. TeX does not start with some pretty basic character codes set up.

```
3 ⟨*initex⟩  
4 \catcode ‘\{ = 1 %  
5 \catcode ‘\} = 2 %  
6 \catcode ‘\# = 6 %  
7 \catcode ‘\^ = 7 %  
8 ⟨/initex⟩
```

Tab characters should not show up in the code, but to be on the safe side.

```

9  <*initex>
10 \catcode '\^I = 10 %
11 </initex>
```

For  $\text{Lua}\text{\TeX}$ , the extra primitives need to be enabled. This is not needed in package mode: common formats have the primitives enabled.

```

12 <*initex>
13 \begingroup\expandafter\expandafter\expandafter\endgroup
14 \expandafter\ifx\csname directlua\endcsname\relax
15 \else
16   \directlua{tex.enableprimitives("", tex.extraprimitives())}%
17 \fi
18 </initex>
```

Depending on the versions available, the  $\text{\LaTeX}$  format may not have the raw  $\text{\Umath}$  primitive names available. We fix that globally: it should cause no issues. Older  $\text{Lua}\text{\TeX}$  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  $\text{\U}$  this should be reasonably safe.

```

19 <*package>
20 \begingroup
21   \expandafter\ifx\csname directlua\endcsname\relax
22   \else
23     \directlua{%
24       local i
25       local t = {}%
26       for _,i in pairs(tex.extraprimitives("luatex")) do
27         if string.match(i,"^U") then
28           if not string.match(i,"^Uchar$") then %$
29             table.insert(t,i)
30           end
31         end
32       end
33       tex.enableprimitives("", t)
34     }%
35   \fi
36 \endgroup
37 </package>
```

## 1.2 The $\text{\pdfstrcmp}$ primitive in $\text{X}\text{\TeX}$

Only  $\text{pdf}\text{\TeX}$  has a primitive called  $\text{\pdfstrcmp}$ . The  $\text{X}\text{\TeX}$  version is just  $\text{\strcmp}$ , so there is some shuffling to do. As this is still a real primitive, using the  $\text{pdf}\text{\TeX}$  name is “safe”.

```

38 \begingroup\expandafter\expandafter\expandafter\endgroup
39   \expandafter\ifx\csname pdfstrcmp\endcsname\relax
40   \let\pdfstrcmp\strcmp
41 \fi
```

### 1.3 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.

```
42 \begingroup\expandafter\expandafter\expandafter\endgroup
43 \expandafter\ifx\csname directlua\endcsname\relax
44 \else
45   \ifnum\luatexversion<70 %
46   \else
```

In package mode a category code table is needed: either use a pre-loaded allocator or provide one using the  $\text{\LaTeX} 2\varepsilon$ -based generic code. In format mode the table used here can be hard-coded into the Lua.

```
47 (*package)
48   \begingroup\expandafter\expandafter\expandafter\endgroup
49   \expandafter\ifx\csname newcatcodetable\endcsname\relax
50     \input{l3luatex}%
51   \fi
52   \newcatcodetable\ucharcat@table
53   \directlua{
54     l3kernel = l3kernel or {}
55     local charcat_table = \number\ucharcat@table\space
56     l3kernel.charcat_table = charcat_table
57   }%
58 
```

```
59 \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.

```
60   \ifnum 0%
61     \directlua{
62       if status.ini_version then
63         tex.write("1")
64       end
65     }>0 %
66     \everyjob\expandafter{%
67       \the\expandafter\everyjob
68       \csname\detokenize{lua_now:n}\endcsname{require("expl3")}%
69     }%
70   \fi
71   \fi
72 \fi
```

### 1.4 Engine requirements

The code currently requires  $\varepsilon$ - $\text{\TeX}$  and functionality equivalent to `\pdfstrcmp`, and also driver and Unicode character support. This is available in a reasonably-wide range of engines.

```
73 \begingroup
74   \def\next{\endgroup}%
75   \def\ShortText{Required primitives not found}%
76   \def\LongText%
```

```

77   {%
78     LaTeX3 requires the e-TeX primitives and additional functionality as
79     described in the README file.
80     \LineBreak
81       These are available in the engines\LineBreak
82       - pdfTeX v1.40\LineBreak
83       - XeTeX v0.99992\LineBreak
84       - LuaTeX v0.76\LineBreak
85       - e-(u)pTeX mid-2012\LineBreak
86       or later.\LineBreak
87         \LineBreak
88   }%
89 \ifnum0%
90   \expandafter\ifx\csname pdfstrcmp\endcsname\relax
91   \else
92     \expandafter\ifx\csname pdftexversion\endcsname\relax
93       \expandafter\ifx\csname Ucharcat\endcsname\relax
94         \expandafter\ifx\csname kanjiskip\endcsname\relax
95         \else
96           1%
97           \fi
98         \else
99           1%
100           \fi
101         \else
102           \ifnum\pdftexversion<140 \else 1\fi
103         \fi
104       \fi
105     \expandafter\ifx\csname directlua\endcsname\relax
106     \else
107       \ifnum\luatexversion<76 \else 1\fi
108     \fi
109     =0 %
110     \newlinechar`^\^J %
111   {*initex}
112     \def\LineBreak{^\^J}%
113     \edef\next
114     {%
115       \errhelp
116       {%
117         \LongText
118         For pdfTeX and XeTeX the '-etex' command-line switch is also
119         needed.\LineBreak
120         \LineBreak
121         Format building will abort!\LineBreak
122       }%
123       \errmessage{\ShortText}%
124       \endgroup
125       \noexpand\end
126     }%
127   
```

```

128   {*package}
129     \def\LineBreak{\noexpand\MessageBreak}%
130     \expandafter\ifx\csname PackageError\endcsname\relax

```

```

131      \def\LineBreak{^^J}%
132      \def\PackageError#1#2#3%
133      {%
134          \errhelp{#3}%
135          \errmessage{#1 Error: #2}%
136      }%
137      \fi
138      \edef\next
139      {%
140          \noexpand\PackageError{expl3}{\ShortText}%
141          {\LongText Loading of expl3 will abort!}%
142          \endgroup
143          \noexpand\endinput
144      }%
145  
```

## 1.5 Extending allocators

In format mode, allocating registers is handled by `\l3alloc`. However, in package mode it's much safer to rely on more general code. For example, 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  $\varepsilon$ -`\TeX` extensions and `\pdfstrcmp` or equivalent are available. Thus there is no danger of an “uncontrolled” error if the engine requirements are not met.

For `LATEX 2 $\varepsilon$`  we need to make sure that the extended pool is being used: `expl3` uses a lot of registers. For formats from 2015 onward there is nothing to do as this is automatic. For older formats, the `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 `\reserveinserts` is `\outer` and so is accessed here by `csname`. In earlier versions, loading `etex` was done directly and so `\reserveinserts` appeared in the code: this then required a `\relax` after `\RequirePackage` to prevent an error with “unsafe” definitions as seen for example with `capoptions`. The optional loading here is done using a group and `\ifx` test as we are not quite in the position to have a single name for `\pdfstrcmp` just yet.

```

148  /*package)
149  \begingroup
150    \def\@tempa{LaTeX2e}%
151    \def\next{}%
152    \ifx\fmtname\@tempa
153        \expandafter\ifx\csname extrafloats\endcsname\relax
154            \def\next
155            {%
156                \RequirePackage{etex}%
157                \csname reserveinserts\endcsname{32}%
158            }%
159        \fi
160    \fi
161    \expandafter\endgroup
162    \next
163 
```

## 1.6 Character data

TEX needs various pieces of data to be set about characters, in particular which ones to treat as letters and which `\lccode` values apply as these affect hyphenation. It makes most sense to set this and related information up in one place. Whilst for LuaTEX hyphenation patterns can be read anywhere, other engines have to build them into the format and so we *must* do this set up before reading the patterns. For the Unicode engines, there are shared loaders available to obtain the relevant information directly from the Unicode Consortium data files. These need standard (Ini)TEX category codes and primitive availability and must therefore loaded *very* early. This has a knock-on effect on the 8-bit set up: it makes sense to do the definitions for those here as well so it is all in one place.

For XFTEX and LuaTEX, which are natively Unicode engines, simply load the Unicode data.

```

164  {*initex}
165  \ifdefined\Umathcode
166    \input load-unicode-data %
167    \input load-unicode-math-classes %
168 \else

```

For the 8-bit engines a font encoding scheme must be chosen. At present, this is the EC (T1) scheme, with the assumption that languages for which this is not appropriate will be used with one of the Unicode engines.

```
169  \begingroup
```

Lower case chars: map to themselves when lower casing and down by "20 when upper casing. (The characters `a`–`z` are set up correctly by iniTEX.)

```

170  \def\temp{%
171    \ifnum\count0>\count2 %
172    \else
173      \global\lccode\count0 = \count0 %
174      \global\uccode\count0 = \numexpr\count0 - "20\relax
175      \advance\count0 by 1 %
176      \expandafter\temp
177    \fi
178  }
179  \count0 = "A0 %
180  \count2 = "BC %
181  \temp
182  \count0 = "E0 %
183  \count2 = "FF %
184  \temp

```

Upper case chars: map up by "20 when lower casing, to themselves when upper casing and require an `\sfcodes` of 999. (The characters `A`–`Z` are set up correctly by iniTEX.)

```

185  \def\temp{%
186    \ifnum\count0>\count2 %
187    \else
188      \global\lccode\count0 = \numexpr\count0 + "20\relax
189      \global\uccode\count0 = \count0 %
190      \global\sfcodes\count0 = 999 %
191      \advance\count0 by 1 %
192      \expandafter\temp
193    \fi

```

```

194      }
195      \count0 = "80 %
196      \count2 = "9C %
197      \temp
198      \count0 = "C0 %
199      \count2 = "DF %
200      \temp

```

A few special cases where things are not as one might expect using the above pattern: dotless-I, dotless-J, dotted-I and d-bar.

```

201      \global\lccode`^\^Y = `^\^Y %
202      \global\uccode`^\^Y = `I %
203      \global\lccode`^\^Z = `^\^Z %
204      \global\uccode`^\^Y = `J %
205      \global\lccode"9D = `i %
206      \global\uccode"9D = "9D %
207      \global\lccode"9E = "9E %
208      \global\uccode"9E = "D0 %

```

Allow hyphenation at a zero-width glyph (used to break up ligatures or to place accents between characters).

```

209      \global\lccode23 = 23 %
210      \endgroup
211      \fi
212  </initex>

```

## 1.7 The L<sup>A</sup>T<sub>E</sub>X3 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 **\ExplSyntaxOff** becomes a “do nothing” command until **\ExplSyntaxOn** is used. For format mode, there is no need to save category codes so that step is skipped.

```

213  \protected\def\ExplSyntaxOff{}%
214  {*package}
215  \protected\edef\ExplSyntaxOff
216  {}%
217  \protected\def\ExplSyntaxOff{}%
218  \catcode 9 = \the\catcode 9\relax
219  \catcode 32 = \the\catcode 32\relax
220  \catcode 34 = \the\catcode 34\relax
221  \catcode 38 = \the\catcode 38\relax
222  \catcode 58 = \the\catcode 58\relax
223  \catcode 94 = \the\catcode 94\relax
224  \catcode 95 = \the\catcode 95\relax
225  \catcode 124 = \the\catcode 124\relax
226  \catcode 126 = \the\catcode 126\relax
227  \endlinechar = \the\endlinechar\relax
228  \chardef\csname\detokenize{l__kernel_expl_bool}\endcsname = 0\relax
229  {}%
230  </package>

```

(End definition for `\ExplSyntaxOff`. This function is documented on page 6.)

The code environment is now set up.

```
231 \catcode 9 = 9\relax
232 \catcode 32 = 9\relax
233 \catcode 34 = 12\relax
234 \catcode 38 = 4\relax
235 \catcode 58 = 11\relax
236 \catcode 94 = 7\relax
237 \catcode 95 = 11\relax
238 \catcode 124 = 12\relax
239 \catcode 126 = 10\relax
240 \endlinechar = 32\relax
```

`\l_kernel_expl_bool` The status for experimental code syntax: this is on at present.

```
241 \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!

```
242 \protected \def \ExplSyntaxOn
243   {
244     \bool_if:NF \l_kernel_expl_bool
245     {
246       \cs_set_protected:Npx \ExplSyntaxOff
247       {
248         \char_set_catcode:nn { 9 } { \char_value_catcode:n { 9 } }
249         \char_set_catcode:nn { 32 } { \char_value_catcode:n { 32 } }
250         \char_set_catcode:nn { 34 } { \char_value_catcode:n { 34 } }
251         \char_set_catcode:nn { 38 } { \char_value_catcode:n { 38 } }
252         \char_set_catcode:nn { 58 } { \char_value_catcode:n { 58 } }
253         \char_set_catcode:nn { 94 } { \char_value_catcode:n { 94 } }
254         \char_set_catcode:nn { 95 } { \char_value_catcode:n { 95 } }
255         \char_set_catcode:nn { 124 } { \char_value_catcode:n { 124 } }
256         \char_set_catcode:nn { 126 } { \char_value_catcode:n { 126 } }
257         \tex_endlinechar:D =
258           \tex_the:D \tex_endlinechar:D \scan_stop:
259         \bool_set_false:N \l_kernel_expl_bool
260         \cs_set_protected:Npn \ExplSyntaxOff { }
261       }
262     }
263     \char_set_catcode_ignore:n      { 9 } % tab
264     \char_set_catcode_ignore:n      { 32 } % space
265     \char_set_catcode_other:n      { 34 } % double quote
266     \char_set_catcode_alignment:n   { 38 } % ampersand
267     \char_set_catcode_letter:n     { 58 } % colon
268     \char_set_catcode_math_superscript:n { 94 } % circumflex
269     \char_set_catcode_letter:n     { 95 } % underscore
270     \char_set_catcode_other:n      { 124 } % pipe
271     \char_set_catcode_space:n      { 126 } % tilde
272     \tex_endlinechar:D = 32 \scan_stop:
273     \bool_set_true:N \l_kernel_expl_bool
```

274 }

(End definition for \ExplSyntaxOn. This function is documented on page 6.)

275 ⟨/initex | package⟩

## 2 I3names implementation

276 ⟨\*initex | package⟩

The prefix here is `kernel`. A few places need `@@` to be left as is; this is obtained as `@@@@`.

277 ⟨@@=kernel⟩

The code here simply renames all of the primitives to new, internal, names. In format mode, it also deletes all of the existing names (although some do come back later).

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.

278 \let \tex\_global:D \global  
279 \let \tex\_let:D \let

Everything is inside a (rather long) group, which keeps `\_kernel_primitive:NN` trapped.

280 \begingroup

`\_kernel_primitive:NN` A temporary function to actually do the renaming. This also allows the original names to be removed in format mode.

281 \long \def \\_kernel\_primitive:NN #1#2  
282 {  
283 \tex\_global:D \tex\_let:D #2 #1  
284 ⟨\*initex⟩  
285 \tex\_global:D \tex\_let:D #1 \tex\_undefined:D  
286 ⟨/initex⟩  
287 }

(End definition for `\_kernel_primitive:NN`.)

To allow extracting “just the names”, a bit of DocStrip fiddling.

288 ⟨/initex | package⟩  
289 ⟨\*initex | 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.

290 \\_kernel\_primitive:NN \tex\_space:D  
291 \\_kernel\_primitive:NN \tex\_italiccorrection:D  
292 \\_kernel\_primitive:NN \tex\_hyphen:D

Now all the other primitives.

293 \\_kernel\_primitive:NN \above \tex\_above:D  
294 \\_kernel\_primitive:NN \abovedisplayshortskip \tex\_abovedisplayshortskip:D  
295 \\_kernel\_primitive:NN \abovedisplayskip \tex\_abovedisplayskip:D  
296 \\_kernel\_primitive:NN \abovewithdelims \tex\_abovewithdelims:D  
297 \\_kernel\_primitive:NN \accent \tex\_accent:D  
298 \\_kernel\_primitive:NN \adjdemerits \tex\_adjdemerits:D

```

299  \__kernel_primitive:NN \advance          \tex_advance:D
300  \__kernel_primitive:NN \afterassignment \tex_afterassignment:D
301  \__kernel_primitive:NN \aftergroup     \tex_aftergroup:D
302  \__kernel_primitive:NN \atop           \tex_atop:D
303  \__kernel_primitive:NN \atopwithdelims \tex_atopwithdelims:D
304  \__kernel_primitive:NN \badness        \tex_badness:D
305  \__kernel_primitive:NN \baselineskip   \tex_baselineskip:D
306  \__kernel_primitive:NN \batchmode      \tex_batchmode:D
307  \__kernel_primitive:NN \begingroup     \tex_begingroup:D
308  \__kernel_primitive:NN \belowdisplayshortskip \tex_belowdisplayshortskip:D
309  \__kernel_primitive:NN \belowdisplayskip  \tex_belowdisplayskip:D
310  \__kernel_primitive:NN \binoppenalty   \tex_binoppenalty:D
311  \__kernel_primitive:NN \botmark        \tex_botmark:D
312  \__kernel_primitive:NN \box            \tex_box:D
313  \__kernel_primitive:NN \boxmaxdepth   \tex_boxmaxdepth:D
314  \__kernel_primitive:NN \brokenpenalty  \tex_brokenpenalty:D
315  \__kernel_primitive:NN \catcode       \tex_catcode:D
316  \__kernel_primitive:NN \char          \tex_char:D
317  \__kernel_primitive:NN \chardef       \tex_chardef:D
318  \__kernel_primitive:NN \cleaders     \tex_cleaders:D
319  \__kernel_primitive:NN \closein      \tex_closein:D
320  \__kernel_primitive:NN \closeout     \tex_closeout:D
321  \__kernel_primitive:NN \clubpenalty   \tex_clubpenalty:D
322  \__kernel_primitive:NN \copy          \tex_copy:D
323  \__kernel_primitive:NN \count         \tex_count:D
324  \__kernel_primitive:NN \countdef     \tex_countdef:D
325  \__kernel_primitive:NN \cr            \tex_cr:D
326  \__kernel_primitive:NN \crcr         \tex_crcr:D
327  \__kernel_primitive:NN \csname       \tex_csname:D
328  \__kernel_primitive:NN \day          \tex_day:D
329  \__kernel_primitive:NN \deadcycles   \tex_deadcycles:D
330  \__kernel_primitive:NN \def          \tex_def:D
331  \__kernel_primitive:NN \defaulthyphenchar \tex_defaulthyphenchar:D
332  \__kernel_primitive:NN \defaultskewchar \tex_defaultskewchar:D
333  \__kernel_primitive:NN \delcode      \tex_delcode:D
334  \__kernel_primitive:NN \delimiter    \tex_delimiter:D
335  \__kernel_primitive:NN \delimiterfactor \tex_delimiterfactor:D
336  \__kernel_primitive:NN \delimitershortfall \tex_delimitershortfall:D
337  \__kernel_primitive:NN \dimen        \tex_dimen:D
338  \__kernel_primitive:NN \dimendef    \tex_dimendef:D
339  \__kernel_primitive:NN \discretionary \tex_discretionary:D
340  \__kernel_primitive:NN \displayindent \tex_displayindent:D
341  \__kernel_primitive:NN \displaylimits \tex_displaylimits:D
342  \__kernel_primitive:NN \displaystyle   \tex_displaystyle:D
343  \__kernel_primitive:NN \displaywidowpenalty \tex_displaywidowpenalty:D
344  \__kernel_primitive:NN \displaywidth  \tex_displaywidth:D
345  \__kernel_primitive:NN \divide       \tex_divide:D
346  \__kernel_primitive:NN \doublehyphendemerits \tex_doublehyphendemerits:D
347  \__kernel_primitive:NN \dp           \tex_dp:D
348  \__kernel_primitive:NN \dump         \tex_dump:D
349  \__kernel_primitive:NN \edef          \tex_edef:D
350  \__kernel_primitive:NN \else         \tex_else:D
351  \__kernel_primitive:NN \emergencystretch \tex_emergencystretch:D
352  \__kernel_primitive:NN \end          \tex_end:D

```

```

353  \__kernel_primitive:NN \endcsname          \tex_endcsname:D
354  \__kernel_primitive:NN \endgroup           \tex_endgroup:D
355  \__kernel_primitive:NN \endinput            \tex_endinput:D
356  \__kernel_primitive:NN \endlinechar         \tex_endlinechar:D
357  \__kernel_primitive:NN \eqno               \tex_eqno:D
358  \__kernel_primitive:NN \errhelp             \tex_errhelp:D
359  \__kernel_primitive:NN \errmessage          \tex_errmessage:D
360  \__kernel_primitive:NN \errorcontextlines   \tex_errorcontextlines:D
361  \__kernel_primitive:NN \errorstopmode        \tex_errorstopmode:D
362  \__kernel_primitive:NN \escapechar          \tex_escapechar:D
363  \__kernel_primitive:NN \everycr              \tex_everycr:D
364  \__kernel_primitive:NN \everydisplay         \tex_everydisplay:D
365  \__kernel_primitive:NN \everyhbox             \tex_everyhbox:D
366  \__kernel_primitive:NN \everyjob              \tex_everyjob:D
367  \__kernel_primitive:NN \everymath             \tex_everymath:D
368  \__kernel_primitive:NN \everypar              \tex_everypar:D
369  \__kernel_primitive:NN \everyvbox             \tex_everyvbox:D
370  \__kernel_primitive:NN \exhyphenpenalty      \tex_exhyphenpenalty:D
371  \__kernel_primitive:NN \expandafter          \tex_expandafter:D
372  \__kernel_primitive:NN \fam                  \tex_fam:D
373  \__kernel_primitive:NN \fi                   \tex_fi:D
374  \__kernel_primitive:NN \finalhyphendemerits  \tex_finalhyphendemerits:D
375  \__kernel_primitive:NN \firstmark            \tex_firstmark:D
376  \__kernel_primitive:NN \floatingpenalty       \tex_floatingpenalty:D
377  \__kernel_primitive:NN \font                 \tex_font:D
378  \__kernel_primitive:NN \fontdimen            \tex_fontdimen:D
379  \__kernel_primitive:NN \fontname              \tex_fontname:D
380  \__kernel_primitive:NN \futurelet             \tex_futurelet:D
381  \__kernel_primitive:NN \gdef                 \tex_gdef:D
382  \__kernel_primitive:NN \global                \tex_global:D
383  \__kernel_primitive:NN \globaldefs            \tex_globaldefs:D
384  \__kernel_primitive:NN \halign                \tex_halign:D
385  \__kernel_primitive:NN \hangafter             \tex_hangafter:D
386  \__kernel_primitive:NN \hangindent            \tex_hangindent:D
387  \__kernel_primitive:NN \hbadness              \tex_hbadness:D
388  \__kernel_primitive:NN \hbox                 \tex_hbox:D
389  \__kernel_primitive:NN \hfil                 \tex_hfil:D
390  \__kernel_primitive:NN \hfill                 \tex_hfill:D
391  \__kernel_primitive:NN \hfilneg               \tex_hfilneg:D
392  \__kernel_primitive:NN \hfuzz                 \tex_hfuzz:D
393  \__kernel_primitive:NN \hoffset               \tex_hoffset:D
394  \__kernel_primitive:NN \holdinginserts        \tex_holdinginserts:D
395  \__kernel_primitive:NN \hrule                \tex_hrule:D
396  \__kernel_primitive:NN \hsize                \tex_hsize:D
397  \__kernel_primitive:NN \hskip                 \tex_hskip:D
398  \__kernel_primitive:NN \hss                  \tex_hss:D
399  \__kernel_primitive:NN \ht                   \tex_ht:D
400  \__kernel_primitive:NN \hyphenation          \tex_hyphenation:D
401  \__kernel_primitive:NN \hyphenchar            \tex_hyphenchar:D
402  \__kernel_primitive:NN \hyphenpenalty         \tex_hyphenpenalty:D
403  \__kernel_primitive:NN \if                   \tex_if:D
404  \__kernel_primitive:NN \ifcase               \tex_ifcase:D
405  \__kernel_primitive:NN \ifcat                \tex_ifcat:D
406  \__kernel_primitive:NN \ifdim                \tex_ifdim:D

```

```

407  \__kernel_primitive:NN \ifeof          \tex_ifeof:D
408  \__kernel_primitive:NN \iffalse         \tex_iffalse:D
409  \__kernel_primitive:NN \ifhbox          \tex_ifhbox:D
410  \__kernel_primitive:NN \ifhmode         \tex_ifhmode:D
411  \__kernel_primitive:NN \ifinner        \tex_ifinner:D
412  \__kernel_primitive:NN \ifmmode         \tex_ifmmode:D
413  \__kernel_primitive:NN \ifnum           \tex_ifnum:D
414  \__kernel_primitive:NN \ifodd           \tex_ifodd:D
415  \__kernel_primitive:NN \iftrue          \tex_iftrue:D
416  \__kernel_primitive:NN \ifvbox          \tex_ifvbox:D
417  \__kernel_primitive:NN \ifvmode         \tex_ifvmode:D
418  \__kernel_primitive:NN \ifvoid          \tex_ifvoid:D
419  \__kernel_primitive:NN \ifx             \tex_ifx:D
420  \__kernel_primitive:NN \ignorespaces    \tex_ignorespaces:D
421  \__kernel_primitive:NN \immediate       \tex_immediate:D
422  \__kernel_primitive:NN \indent          \tex_indent:D
423  \__kernel_primitive:NN \input           \tex_input:D
424  \__kernel_primitive:NN \inputlineno     \tex_inputlineno:D
425  \__kernel_primitive:NN \insert          \tex_insert:D
426  \__kernel_primitive:NN \insertpenalties \tex_insertpenalties:D
427  \__kernel_primitive:NN \interlinepenalty \tex_interlinepenalty:D
428  \__kernel_primitive:NN \jobname         \tex_jobname:D
429  \__kernel_primitive:NN \kern             \tex_kern:D
430  \__kernel_primitive:NN \language        \tex_language:D
431  \__kernel_primitive:NN \lastbox         \tex_lastbox:D
432  \__kernel_primitive:NN \lastkern        \tex_lastkern:D
433  \__kernel_primitive:NN \lastpenalty     \tex_lastpenalty:D
434  \__kernel_primitive:NN \lastskip        \tex_lastskip:D
435  \__kernel_primitive:NN \lccode          \tex_lccode:D
436  \__kernel_primitive:NN \leaders         \tex_leaders:D
437  \__kernel_primitive:NN \left            \tex_left:D
438  \__kernel_primitive:NN \lefthyphenmin   \tex_lefthyphenmin:D
439  \__kernel_primitive:NN \leftskip        \tex_leftskip:D
440  \__kernel_primitive:NN \leqno           \tex_leqno:D
441  \__kernel_primitive:NN \let              \tex_let:D
442  \__kernel_primitive:NN \limits          \tex_limits:D
443  \__kernel_primitive:NN \linepenalty     \tex_linepenalty:D
444  \__kernel_primitive:NN \lineskip        \tex_lineskip:D
445  \__kernel_primitive:NN \lineskiplimit   \tex_lineskiplimit:D
446  \__kernel_primitive:NN \long             \tex_long:D
447  \__kernel_primitive:NN \looseness       \tex_looseness:D
448  \__kernel_primitive:NN \lower            \tex_lower:D
449  \__kernel_primitive:NN \lowercase       \tex_lowercase:D
450  \__kernel_primitive:NN \mag              \tex_mag:D
451  \__kernel_primitive:NN \mark             \tex_mark:D
452  \__kernel_primitive:NN \mathaccent       \tex_mathaccent:D
453  \__kernel_primitive:NN \mathbin          \tex_mathbin:D
454  \__kernel_primitive:NN \mathchar         \tex_mathchar:D
455  \__kernel_primitive:NN \mathchardef     \tex_mathchardef:D
456  \__kernel_primitive:NN \mathchoice       \tex_mathchoice:D
457  \__kernel_primitive:NN \mathclose        \tex_mathclose:D
458  \__kernel_primitive:NN \mathcode         \tex_mathcode:D
459  \__kernel_primitive:NN \mathinner        \tex_mathinner:D
460  \__kernel_primitive:NN \mathop            \tex_mathop:D

```

```

461  \__kernel_primitive:NN \mathopen          \tex_mathopen:D
462  \__kernel_primitive:NN \mathord           \tex_mathord:D
463  \__kernel_primitive:NN \mathpunct        \tex_mathpunct:D
464  \__kernel_primitive:NN \mathrel          \tex_mathrel:D
465  \__kernel_primitive:NN \mathsurround     \tex_mathsurround:D
466  \__kernel_primitive:NN \maxdeadcycles   \tex_maxdeadcycles:D
467  \__kernel_primitive:NN \maxdepth        \tex_maxdepth:D
468  \__kernel_primitive:NN \meaning         \tex_meaning:D
469  \__kernel_primitive:NN \medmuskip       \tex_medmuskip:D
470  \__kernel_primitive:NN \message         \tex_message:D
471  \__kernel_primitive:NN \mkern            \tex_mkern:D
472  \__kernel_primitive:NN \month            \tex_month:D
473  \__kernel_primitive:NN \moveleft         \tex_moveleft:D
474  \__kernel_primitive:NN \moveright        \tex_moveright:D
475  \__kernel_primitive:NN \mskip             \tex_msip:D
476  \__kernel_primitive:NN \multiply         \tex_multiply:D
477  \__kernel_primitive:NN \muskip            \tex_muskip:D
478  \__kernel_primitive:NN \muskipdef        \tex_muskipdef:D
479  \__kernel_primitive:NN \newlinechar      \tex_newlinechar:D
480  \__kernel_primitive:NN \noalign            \tex_noalign:D
481  \__kernel_primitive:NN \noboundary        \tex_noboundary:D
482  \__kernel_primitive:NN \noexpand          \tex_noexpand:D
483  \__kernel_primitive:NN \noindent          \tex_noindent:D
484  \__kernel_primitive:NN \nolimits           \tex_nolimits:D
485  \__kernel_primitive:NN \nonscript         \tex_nonscript:D
486  \__kernel_primitive:NN \nonstopmode       \tex_nonstopmode:D
487  \__kernel_primitive:NN \nulldelimiterspace \tex_nulldelimiterspace:D
488  \__kernel_primitive:NN \nullfont          \tex_nullfont:D
489  \__kernel_primitive:NN \number            \tex_number:D
490  \__kernel_primitive:NN \omit              \tex_omit:D
491  \__kernel_primitive:NN \openin             \tex_openin:D
492  \__kernel_primitive:NN \openout            \tex_openout:D
493  \__kernel_primitive:NN \or                \tex_or:D
494  \__kernel_primitive:NN \outer              \tex_outer:D
495  \__kernel_primitive:NN \output             \tex_output:D
496  \__kernel_primitive:NN \outputpenalty     \tex_outputpenalty:D
497  \__kernel_primitive:NN \over               \tex_over:D
498  \__kernel_primitive:NN \overfullrule      \tex_overfullrule:D
499  \__kernel_primitive:NN \overline            \tex_overline:D
500  \__kernel_primitive:NN \overwithdelims    \tex_overwithdelims:D
501  \__kernel_primitive:NN \pagedepth          \tex_pagedepth:D
502  \__kernel_primitive:NN \pagefilllstretch   \tex_pagefilllstretch:D
503  \__kernel_primitive:NN \pagefillstretch    \tex_pagefillstretch:D
504  \__kernel_primitive:NN \pagefilstretch     \tex_pagefilstretch:D
505  \__kernel_primitive:NN \pagegoal           \tex_pagegoal:D
506  \__kernel_primitive:NN \pageshrink         \tex_pageshrink:D
507  \__kernel_primitive:NN \pagestretch        \tex_pagestretch:D
508  \__kernel_primitive:NN \pagetotal          \tex_pagetotal:D
509  \__kernel_primitive:NN \par                \tex_par:D
510  \__kernel_primitive:NN \parfillskip        \tex_parfillskip:D
511  \__kernel_primitive:NN \parindent          \tex_parindent:D
512  \__kernel_primitive:NN \parshape           \tex_parshape:D
513  \__kernel_primitive:NN \parskip             \tex_parskip:D
514  \__kernel_primitive:NN \patterns           \tex_patterns:D

```

```

515  \__kernel_primitive:NN \pausing          \tex_pausing:D
516  \__kernel_primitive:NN \penalty          \tex_penalty:D
517  \__kernel_primitive:NN \postdisplaypenalty \tex_postdisplaypenalty:D
518  \__kernel_primitive:NN \predisplaypenalty \tex_predisplaypenalty:D
519  \__kernel_primitive:NN \predisplaysize    \tex_predisplaysize:D
520  \__kernel_primitive:NN \pretolerance     \tex_pretolerance:D
521  \__kernel_primitive:NN \prevdepth        \tex_prevdepth:D
522  \__kernel_primitive:NN \prevgraf         \tex_prevgraf:D
523  \__kernel_primitive:NN \radical          \tex_radical:D
524  \__kernel_primitive:NN \raise            \tex_raise:D
525  \__kernel_primitive:NN \read             \tex_read:D
526  \__kernel_primitive:NN \relax            \tex_relax:D
527  \__kernel_primitive:NN \relpenalty       \tex_relpenalty:D
528  \__kernel_primitive:NN \right            \tex_right:D
529  \__kernel_primitive:NN \righthyphenmin   \tex_righthyphenmin:D
530  \__kernel_primitive:NN \rightskip        \tex_rightskip:D
531  \__kernel_primitive:NN \romannumeral    \tex_roman numeral:D
532  \__kernel_primitive:NN \scriptfont      \tex_scriptfont:D
533  \__kernel_primitive:NN \scriptscriptfont \tex_scriptscriptfont:D
534  \__kernel_primitive:NN \scriptstyle      \tex_scriptstyle:D
535  \__kernel_primitive:NN \scriptspace     \tex_scriptspace:D
536  \__kernel_primitive:NN \scrollmode      \tex_scrollmode:D
537  \__kernel_primitive:NN \setbox           \tex_setbox:D
538  \__kernel_primitive:NN \setlanguage     \tex_setlanguage:D
539  \__kernel_primitive:NN \sfcode          \tex_sfcode:D
540  \__kernel_primitive:NN \shipout         \tex_shipout:D
541  \__kernel_primitive:NN \show             \tex_show:D
542  \__kernel_primitive:NN \showbox         \tex_showbox:D
543  \__kernel_primitive:NN \showboxbreadth \tex_showboxbreadth:D
544  \__kernel_primitive:NN \showboxdepth    \tex_showboxdepth:D
545  \__kernel_primitive:NN \showlists       \tex_showlists:D
546  \__kernel_primitive:NN \showthe         \tex_showthe:D
547  \__kernel_primitive:NN \skewchar        \tex_skewchar:D
548  \__kernel_primitive:NN \skip             \tex_skip:D
549  \__kernel_primitive:NN \skipdef         \tex_skipdef:D
550  \__kernel_primitive:NN \spacefactor     \tex_spacefactor:D
551  \__kernel_primitive:NN \spaceskip       \tex_spaceskip:D
552  \__kernel_primitive:NN \span             \tex_span:D
553  \__kernel_primitive:NN \special          \tex_special:D
554  \__kernel_primitive:NN \splitbotmark   \tex_splitbotmark:D
555  \__kernel_primitive:NN \splitfirstmark \tex_splitfirstmark:D
556  \__kernel_primitive:NN \splitmaxdepth   \tex_splitmaxdepth:D
557  \__kernel_primitive:NN \splittopskip    \tex_splittopskip:D
558  \__kernel_primitive:NN \string          \tex_string:D
559  \__kernel_primitive:NN \tabskip         \tex_tabskip:D
560  \__kernel_primitive:NN \textfont        \tex_textfont:D
561  \__kernel_primitive:NN \textstyle       \tex_textstyle:D
562  \__kernel_primitive:NN \the              \tex_the:D
563  \__kernel_primitive:NN \thickmuskip    \tex_thickmuskip:D
564  \__kernel_primitive:NN \thinmuskip     \tex_thinmuskip:D
565  \__kernel_primitive:NN \time            \tex_time:D
566  \__kernel_primitive:NN \toks             \tex_toks:D
567  \__kernel_primitive:NN \toksdef         \tex_toksdef:D

```

```

569  \__kernel_primitive:NN \tolerance          \tex_tolerance:D
570  \__kernel_primitive:NN \topmark            \tex_topmark:D
571  \__kernel_primitive:NN \topskip             \tex_topskip:D
572  \__kernel_primitive:NN \tracingcommands    \tex_tracingcommands:D
573  \__kernel_primitive:NN \tracinglostchars   \tex_tracinglostchars:D
574  \__kernel_primitive:NN \tracingmacros      \tex_tracingmacros:D
575  \__kernel_primitive:NN \tracingonline      \tex_tracingonline:D
576  \__kernel_primitive:NN \tracingoutput       \tex_tracingoutput:D
577  \__kernel_primitive:NN \tracingpages        \tex_tracingpages:D
578  \__kernel_primitive:NN \tracingparagraphs   \tex_tracingparagraphs:D
579  \__kernel_primitive:NN \tracingrestores     \tex_tracingrestores:D
580  \__kernel_primitive:NN \tracingstats        \tex_tracingstats:D
581  \__kernel_primitive:NN \uccode              \tex_uccode:D
582  \__kernel_primitive:NN \uchyph               \tex_uchyph:D
583  \__kernel_primitive:NN \underline           \tex_underline:D
584  \__kernel_primitive:NN \unhbox              \tex_unhbox:D
585  \__kernel_primitive:NN \unhcopy             \tex_unhcopy:D
586  \__kernel_primitive:NN \unkern              \tex_unkern:D
587  \__kernel_primitive:NN \unpenalty           \tex_unpenalty:D
588  \__kernel_primitive:NN \unskip               \tex_unskip:D
589  \__kernel_primitive:NN \unvbox              \tex_unvbox:D
590  \__kernel_primitive:NN \unvcopy             \tex_unvcopy:D
591  \__kernel_primitive:NN \uppercase           \tex_uppercase:D
592  \__kernel_primitive:NN \vadjust              \tex_vadjust:D
593  \__kernel_primitive:NN \valign              \tex_valign:D
594  \__kernel_primitive:NN \vbadness             \tex_vbadness:D
595  \__kernel_primitive:NN \vbox                \tex_vbox:D
596  \__kernel_primitive:NN \vcenter              \tex_vcenter:D
597  \__kernel_primitive:NN \vfil                \tex_vfil:D
598  \__kernel_primitive:NN \vfill               \tex_vfill:D
599  \__kernel_primitive:NN \vfilneg              \tex_vfilneg:D
600  \__kernel_primitive:NN \vfuzz               \tex_vfuzz:D
601  \__kernel_primitive:NN \voffset             \tex_voffset:D
602  \__kernel_primitive:NN \vrule               \tex_vrule:D
603  \__kernel_primitive:NN \vsize               \tex_vsize:D
604  \__kernel_primitive:NN \vskip               \tex_vskip:D
605  \__kernel_primitive:NN \vsplit              \tex_vsplit:D
606  \__kernel_primitive:NN \vss                 \tex_vss:D
607  \__kernel_primitive:NN \vtop                \tex_vtop:D
608  \__kernel_primitive:NN \wd                  \tex_wd:D
609  \__kernel_primitive:NN \widowpenalty       \tex_widowpenalty:D
610  \__kernel_primitive:NN \write               \tex_write:D
611  \__kernel_primitive:NN \xdef                \tex_xdef:D
612  \__kernel_primitive:NN \xleaders            \tex_xleaders:D
613  \__kernel_primitive:NN \xspaceskip         \tex_xspaceskip:D
614  \__kernel_primitive:NN \year                \tex_year:D

```

Primitives introduced by  $\varepsilon$ -TEX.

```

615  \__kernel_primitive:NN \beginL            \tex_beginL:D
616  \__kernel_primitive:NN \beginR            \tex_beginR:D
617  \__kernel_primitive:NN \botmarks          \tex_botmarks:D
618  \__kernel_primitive:NN \clubpenalties    \tex_clubpenalties:D
619  \__kernel_primitive:NN \currentgrouplevel \tex_currentgrouplevel:D
620  \__kernel_primitive:NN \currentgrouptype  \tex_currentgrouptype:D
621  \__kernel_primitive:NN \currentifbranch   \tex_currentifbranch:D

```

```

622  \__kernel_primitive:NN \currentiflevel          \tex_currentiflevel:D
623  \__kernel_primitive:NN \currentiftype          \tex_currentiftype:D
624  \__kernel_primitive:NN \detokenize             \tex_detokenize:D
625  \__kernel_primitive:NN \dimexpr                \tex_dimexpr:D
626  \__kernel_primitive:NN \displaywidowpenalties \tex_displaywidowpenalties:D
627  \__kernel_primitive:NN \endL                   \tex_endL:D
628  \__kernel_primitive:NN \endR                   \tex_endR:D
629  \__kernel_primitive:NN \eTeXrevision           \tex_eTeXrevision:D
630  \__kernel_primitive:NN \eTeXversion            \tex_eTeXversion:D
631  \__kernel_primitive:NN \everyeof               \tex_everyeof:D
632  \__kernel_primitive:NN \firstmarks            \tex_firstmarks:D
633  \__kernel_primitive:NN \fontchardp            \tex_fontchardp:D
634  \__kernel_primitive:NN \fontcharht            \tex_fontcharht:D
635  \__kernel_primitive:NN \fontcharic            \tex_fontcharic:D
636  \__kernel_primitive:NN \fontcharwd            \tex_fontcharwd:D
637  \__kernel_primitive:NN \glueexpr              \tex_glueexpr:D
638  \__kernel_primitive:NN \glueshrink            \tex_glueshrink:D
639  \__kernel_primitive:NN \glueshrinkorder       \tex_glueshrinkorder:D
640  \__kernel_primitive:NN \gluestretch            \tex_gluestretch:D
641  \__kernel_primitive:NN \gluestretchorder       \tex_gluestretchorder:D
642  \__kernel_primitive:NN \gluetomu              \tex_gluetomu:D
643  \__kernel_primitive:NN \ifcsname              \tex_ifcsname:D
644  \__kernel_primitive:NN \ifdefined              \tex_ifdefined:D
645  \__kernel_primitive:NN \iffontchar             \tex_iffontchar:D
646  \__kernel_primitive:NN \interactionmode        \tex_interactionmode:D
647  \__kernel_primitive:NN \interlinepenalties     \tex_interlinepenalties:D
648  \__kernel_primitive:NN \lastlinefit            \tex_lastlinefit:D
649  \__kernel_primitive:NN \lastnodetype           \tex_lastnodetype:D
650  \__kernel_primitive:NN \marks                  \tex_marks:D
651  \__kernel_primitive:NN \middle                 \tex_middle:D
652  \__kernel_primitive:NN \muexpr                \tex_muexpr:D
653  \__kernel_primitive:NN \mutoglue              \tex_mutoglue:D
654  \__kernel_primitive:NN \numexpr                \tex_numexpr:D
655  \__kernel_primitive:NN \pagediscards          \tex_pagediscards:D
656  \__kernel_primitive:NN \parshapedimen         \tex_parshapedimen:D
657  \__kernel_primitive:NN \parshapeindent         \tex_parshapeindent:D
658  \__kernel_primitive:NN \parshapelen            \tex_parshapelen:D
659  \__kernel_primitive:NN \predisplaydirection   \tex_predisplaydirection:D
660  \__kernel_primitive:NN \protected              \tex_protected:D
661  \__kernel_primitive:NN \readline               \tex_readline:D
662  \__kernel_primitive:NN \savinghyphcodes       \tex_savinghyphcodes:D
663  \__kernel_primitive:NN \savingvdiscards        \tex_savingvdiscards:D
664  \__kernel_primitive:NN \scantokens            \tex_scantokens:D
665  \__kernel_primitive:NN \showgroups            \tex_showgroups:D
666  \__kernel_primitive:NN \showifs               \tex_showifs:D
667  \__kernel_primitive:NN \showtokens            \tex_showtokens:D
668  \__kernel_primitive:NN \splitbotmarks         \tex_splitbotmarks:D
669  \__kernel_primitive:NN \spliddiscards         \tex_spliddiscards:D
670  \__kernel_primitive:NN \splitfirstmarks       \tex_splitfirstmarks:D
671  \__kernel_primitive:NN \TeXeTstate            \tex_TeXXeTstate:D
672  \__kernel_primitive:NN \topmarks              \tex_topmarks:D
673  \__kernel_primitive:NN \tracingassigns        \tex_tracingassigns:D
674  \__kernel_primitive:NN \tracinggroups         \tex_tracinggroups:D
675  \__kernel_primitive:NN \tracingifs            \tex_tracingifs:D

```

```

676  \__kernel_primitive:NN \tracingnesting          \tex_tracingnesting:D
677  \__kernel_primitive:NN \tracingscantokens      \tex_tracingscantokens:D
678  \__kernel_primitive:NN \unexpanded             \tex_unexpanded:D
679  \__kernel_primitive:NN \unless                 \tex_unless:D
680  \__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.

```

681  \__kernel_primitive:NN \pdfannot                \tex_pdfannot:D
682  \__kernel_primitive:NN \pdfcatalog              \tex_pdfcatalog:D
683  \__kernel_primitive:NN \pdfcompresslevel       \tex_pdfcompresslevel:D
684  \__kernel_primitive:NN \pdfcolorstack          \tex_pdfcolorstack:D
685  \__kernel_primitive:NN \pdfcolorstackinit      \tex_pdfcolorstackinit:D
686  \__kernel_primitive:NN \pdfcreationdate       \tex_pdfcreationdate:D
687  \__kernel_primitive:NN \pdfdecimaldigits      \tex_pdfdecimaldigits:D
688  \__kernel_primitive:NN \pdfdest                 \tex_pdfdest:D
689  \__kernel_primitive:NN \pdfdestmargin         \tex_pdfdestmargin:D
690  \__kernel_primitive:NN \pdfendlink             \tex_pdfendlink:D
691  \__kernel_primitive:NN \pdfendthread          \tex_pdfendthread:D
692  \__kernel_primitive:NN \pdffontattr            \tex_pdffontattr:D
693  \__kernel_primitive:NN \pdffontname           \tex_pdffontname:D
694  \__kernel_primitive:NN \pdffontobjnum        \tex_pdffontobjnum:D
695  \__kernel_primitive:NN \pdfgamma               \tex_pdfgamma:D
696  \__kernel_primitive:NN \pdfimageapplygamma    \tex_pdfimageapplygamma:D
697  \__kernel_primitive:NN \pdfimagegamma         \tex_pdfimagegamma:D
698  \__kernel_primitive:NN \pdfgentounicode      \tex_pdfgentounicode:D
699  \__kernel_primitive:NN \pdfglyptounicode     \tex_pdfglyptounicode:D
700  \__kernel_primitive:NN \pdfhorigin            \tex_pdfhorigin:D
701  \__kernel_primitive:NN \pdfimagehicolor      \tex_pdfimagehicolor:D
702  \__kernel_primitive:NN \pdfimageresolution   \tex_pdfimageresolution:D
703  \__kernel_primitive:NN \pdfincludechars      \tex_pdfincludechars:D
704  \__kernel_primitive:NN \pdfinclusioncopyfonts \tex_pdfinclusioncopyfonts:D
705  \__kernel_primitive:NN \pdfinclusionerrorlevel \tex_pdfinclusionerrorlevel:D
706  \__kernel_primitive:NN \pdfinfo                \tex_pdfinfo:D
707  \__kernel_primitive:NN \pdflastannot          \tex_pdflastannot:D
708  \__kernel_primitive:NN \pdflastlink           \tex_pdflastlink:D
709  \__kernel_primitive:NN \pdflastobj            \tex_pdflastobj:D
710  \__kernel_primitive:NN \pdflastxform         \tex_pdflastxform:D
711  \__kernel_primitive:NN \pdflastximage        \tex_pdflastximage:D
712  \__kernel_primitive:NN \pdflastximagecolordepth \tex_pdflastximagecolordepth:D
713  \__kernel_primitive:NN \pdflastximagepages   \tex_pdflastximagepages:D
714  \__kernel_primitive:NN \pdflinkmargin        \tex_pdflinkmargin:D
715  \__kernel_primitive:NN \pdfliteral             \tex_pdfliteral:D
716  \__kernel_primitive:NN \pdfminorversion      \tex_pdfminorversion:D
717  \__kernel_primitive:NN \pdfnames              \tex_pdfnames:D
718  \__kernel_primitive:NN \pdfobj                \tex_pdfobj:D
719  \__kernel_primitive:NN \pdfobjcompresslevel  \tex_pdfobjcompresslevel:D
720  \__kernel_primitive:NN \pdfoutline            \tex_pdfoutline:D
721  \__kernel_primitive:NN \pdfoutput             \tex_pdfoutput:D
722  \__kernel_primitive:NN \pdfpageattr          \tex_pdfpageattr:D
723  \__kernel_primitive:NN \pdfpageattr

```

```

725  \__kernel_primitive:NN \pdfpagebox           \tex_pdfpagebox:D
726  \__kernel_primitive:NN \pdfpageref            \tex_pdfpageref:D
727  \__kernel_primitive:NN \pdfpageresources     \tex_pdfpageresources:D
728  \__kernel_primitive:NN \pdfpagesattr          \tex_pdfpagesattr:D
729  \__kernel_primitive:NN \pdfrefobj             \tex_pdfrefobj:D
730  \__kernel_primitive:NN \pdfrefxform           \tex_pdfrefxform:D
731  \__kernel_primitive:NN \pdfrefximage          \tex_pdfrefximage:D
732  \__kernel_primitive:NN \pdfrestore             \tex_pdfrestore:D
733  \__kernel_primitive:NN \pdfretval              \tex_pdfretval:D
734  \__kernel_primitive:NN \pdfsave                \tex_pdfsave:D
735  \__kernel_primitive:NN \pdfsetmatrix           \tex_pdfsetmatrix:D
736  \__kernel_primitive:NN \pdfstartlink          \tex_pdfstartlink:D
737  \__kernel_primitive:NN \pdfstartthread         \tex_pdfstartthread:D
738  \__kernel_primitive:NN \pdfsuppressptexinfo    \tex_pdfsuppressptexinfo:D
739  \__kernel_primitive:NN \pdfthread              \tex_pdfthread:D
740  \__kernel_primitive:NN \pdfthreadmargin        \tex_pdfthreadmargin:D
741  \__kernel_primitive:NN \pdftrailer              \tex_pdftrailer:D
742  \__kernel_primitive:NN \pdfuniqueresname       \tex_pdfuniqueresname:D
743  \__kernel_primitive:NN \pdfvorigin             \tex_pdfvorigin:D
744  \__kernel_primitive:NN \pdfxform               \tex_pdfxform:D
745  \__kernel_primitive:NN \pdfxformattr           \tex_pdfxformattr:D
746  \__kernel_primitive:NN \pdfxformname           \tex_pdfxformname:D
747  \__kernel_primitive:NN \pdfxformresources      \tex_pdfxformresources:D
748  \__kernel_primitive:NN \pdfximage              \tex_pdfximage:D
749  \__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.

```

750  \__kernel_primitive:NN \ifpdfabsdim           \tex_ifabsdim:D
751  \__kernel_primitive:NN \ifpdfabsnum          \tex_ifabsnum:D
752  \__kernel_primitive:NN \ifpdfprimitive        \tex_ifprimitive:D
753  \__kernel_primitive:NN \pdfadjustspacing      \tex_adjustspacing:D
754  \__kernel_primitive:NN \pdfcopyfont           \tex_copyfont:D
755  \__kernel_primitive:NN \pdfdraftmode          \tex_draftmode:D
756  \__kernel_primitive:NN \pdfeachlinedepth       \tex_eachlinedepth:D
757  \__kernel_primitive:NN \pdfeachlineheight     \tex_eachlineheight:D
758  \__kernel_primitive:NN \pdfelapsedtime        \tex_elapsedtime:D
759  \__kernel_primitive:NN \pdffiledump            \tex_filedump:D
760  \__kernel_primitive:NN \pdffilemoddate        \tex_filemoddate:D
761  \__kernel_primitive:NN \pdffilesize            \tex_filesize:D
762  \__kernel_primitive:NN \pdffirstlineheight     \tex_firstlineheight:D
763  \__kernel_primitive:NN \pdffontexpand          \tex_fontexpand:D
764  \__kernel_primitive:NN \pdffontsize             \tex_fontsize:D
765  \__kernel_primitive:NN \pdfignoreddimen       \tex_ignoreddimen:D
766  \__kernel_primitive:NN \pdfinsertht            \tex_insertht:D
767  \__kernel_primitive:NN \pdflastlinedepth        \tex_lastlinedepth:D
768  \__kernel_primitive:NN \pdflastxpos            \tex_lastxpos:D
769  \__kernel_primitive:NN \pdflasttypos           \tex_lasttypos:D
770  \__kernel_primitive:NN \pdfmapfile              \tex_mapfile:D
771  \__kernel_primitive:NN \pdfmapline              \tex_mapline:D
772  \__kernel_primitive:NN \pdfmdfivesum          \tex_mdfivesum:D
773  \__kernel_primitive:NN \pdfnoligatures         \tex_noligatures:D
774  \__kernel_primitive:NN \pdfnormaldeviate       \tex_normaldeviate:D

```

```

775  \__kernel_primitive:NN \pdfpageheight          \tex_pageheight:D
776  \__kernel_primitive:NN \pdfpagewidth           \tex_pagewidth:D
777  \__kernel_primitive:NN \pdfpkmode              \tex_pkmode:D
778  \__kernel_primitive:NN \pdfpkresolution        \tex_pkresolution:D
779  \__kernel_primitive:NN \pdfprimitive           \tex_primitive:D
780  \__kernel_primitive:NN \pdfprotrudechars       \tex_protrudechars:D
781  \__kernel_primitive:NN \pdfpxdimen             \tex_pxdimen:D
782  \__kernel_primitive:NN \pdfrandomseed          \tex_randomseed:D
783  \__kernel_primitive:NN \pdfresettimer          \tex_resettimer:D
784  \__kernel_primitive:NN \pdfsavepos              \tex_savepos:D
785  \__kernel_primitive:NN \pdfstrcmp               \tex_strcmp:D
786  \__kernel_primitive:NN \pdfsetrandomseed        \tex_setrandomseed:D
787  \__kernel_primitive:NN \pdfshellescape          \tex_shellescape:D
788  \__kernel_primitive:NN \pdftracingfonts         \tex_tracingfonts:D
789  \__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.

```

790  \__kernel_primitive:NN \pdftexbanner           \tex_pdftexbanner:D
791  \__kernel_primitive:NN \pdftexrevision          \tex_pdftexrevision:D
792  \__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.

```

793  \__kernel_primitive:NN \efcode                 \tex_efcode:D
794  \__kernel_primitive:NN \ifinccsname            \tex_ifinccsname:D
795  \__kernel_primitive:NN \leftmarginkern          \tex_leftmarginkern:D
796  \__kernel_primitive:NN \letterspacefont        \tex_letterspacefont:D
797  \__kernel_primitive:NN \lpcode                 \tex_lpcode:D
798  \__kernel_primitive:NN \quitvmode              \tex_quitvmode:D
799  \__kernel_primitive:NN \rightmarginkern         \tex_rightmarginkern:D
800  \__kernel_primitive:NN \rpcode                 \tex_rpcode:D
801  \__kernel_primitive:NN \synctex                \tex_synctex:D
802  \__kernel_primitive:NN \tagcode                \tex_tagcode:D

```

Post pdfTEX primitive availability gets more complex. Both X<sub>E</sub>T<sub>E</sub>X and LuaT<sub>E</sub>X have varying names for some primitives from pdfTEX. Particularly for LuaT<sub>E</sub>X tracking all of that would be hard. Instead, we now check that we only save primitives if they actually exist.

```

803  </initex | names | package>
804  <*initex | package>
805    \tex_long:D \tex_def:D \use_ii:nn #1#2 {#2}
806    \tex_long:D \tex_def:D \use_none:n #1 { }
807    \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
808    {
809      \tex_ifdefined:D #1
810      \tex_expandafter:D \use_ii:nn
811      \tex_fi:D
812      \use_none:n { \tex_global:D \tex_let:D #2 #1 }
813  <*initex>
814    \tex_global:D \tex_let:D #1 \tex_undefined:D
815  </initex>
816  }
817 </initex | package>
818 <*initex | names | package>

```

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.

```

819  \__kernel_primitive:NN \suppressfontnotfounderror
820      \tex_suppressfontnotfounderror:D
821  \__kernel_primitive:NN \XeTeXcharclass      \tex_XeTeXcharclass:D
822  \__kernel_primitive:NN \XeTeXcharglyph     \tex_XeTeXcharglyph:D
823  \__kernel_primitive:NN \XeTeXcountfeatures  \tex_XeTeXcountfeatures:D
824  \__kernel_primitive:NN \XeTeXcountglyphs   \tex_XeTeXcountglyphs:D
825  \__kernel_primitive:NN \XeTeXcountselectors \tex_XeTeXcountselectors:D
826  \__kernel_primitive:NN \XeTeXcountvariations \tex_XeTeXcountvariations:D
827  \__kernel_primitive:NN \XeTeXdefaultencoding \tex_XeTeXdefaultencoding:D
828  \__kernel_primitive:NN \XeTeXdashbreakstate \tex_XeTeXdashbreakstate:D
829  \__kernel_primitive:NN \XeTeXfeaturecode    \tex_XeTeXfeaturecode:D
830  \__kernel_primitive:NN \XeTeXfeaturename   \tex_XeTeXfeaturename:D
831  \__kernel_primitive:NN \XeTeXfindfeaturebyname
832      \tex_XeTeXfindfeaturebyname:D
833  \__kernel_primitive:NN \XeTeXfindselectorbyname
834      \tex_XeTeXfindselectorbyname:D
835  \__kernel_primitive:NN \XeTeXfindvariationbyname
836      \tex_XeTeXfindvariationbyname:D
837  \__kernel_primitive:NN \XeTeXfirstfontchar   \tex_XeTeXfirstfontchar:D
838  \__kernel_primitive:NN \XeTeXfonttype       \tex_XeTeXfonttype:D
839  \__kernel_primitive:NN \XeTeXgenerateactualtext
840      \tex_XeTeXgenerateactualtext:D
841  \__kernel_primitive:NN \XeTeXglyph          \tex_XeTeXglyph:D
842  \__kernel_primitive:NN \XeTeXglyphbounds    \tex_XeTeXglyphbounds:D
843  \__kernel_primitive:NN \XeTeXglyphindex     \tex_XeTeXglyphindex:D
844  \__kernel_primitive:NN \XeTeXglyphname      \tex_XeTeXglyphname:D
845  \__kernel_primitive:NN \XeTeXinputencoding  \tex_XeTeXinputencoding:D
846  \__kernel_primitive:NN \XeTeXinputnormalization
847      \tex_XeTeXinputnormalization:D
848  \__kernel_primitive:NN \XeTeXinterchartokenstate
849      \tex_XeTeXinterchartokenstate:D
850  \__kernel_primitive:NN \XeTeXinterchartoks   \tex_XeTeXinterchartoks:D
851  \__kernel_primitive:NN \XeTeXisdefaultselector
852      \tex_XeTeXisdefaultselector:D
853  \__kernel_primitive:NN \XeTeXisexclusivefeature
854      \tex_XeTeXisexclusivefeature:D
855  \__kernel_primitive:NN \XeTeXlastfontchar    \tex_XeTeXlastfontchar:D
856  \__kernel_primitive:NN \XeTeXlinebreakskip   \tex_XeTeXlinebreakskip:D
857  \__kernel_primitive:NN \XeTeXlinebreaklocale \tex_XeTeXlinebreaklocale:D
858  \__kernel_primitive:NN \XeTeXlinebreakpenalty \tex_XeTeXlinebreakpenalty:D
859  \__kernel_primitive:NN \XeTeXOTcountfeatures \tex_XeTeXOTcountfeatures:D
860  \__kernel_primitive:NN \XeTeXOTcountlanguages \tex_XeTeXOTcountlanguages:D
861  \__kernel_primitive:NN \XeTeXOTcountsheets  \tex_XeTeXOTcountsheets:D
862  \__kernel_primitive:NN \XeTeXOTfeaturetag    \tex_XeTeXOTfeaturetag:D
863  \__kernel_primitive:NN \XeTeXOTlanguageetag  \tex_XeTeXOTlanguageetag:D
864  \__kernel_primitive:NN \XeTeXOTscripttag    \tex_XeTeXOTscripttag:D
865  \__kernel_primitive:NN \XeTeXpdffile        \tex_XeTeXpdffile:D
866  \__kernel_primitive:NN \XeTeXpdfpagecount   \tex_XeTeXpdfpagecount:D
867  \__kernel_primitive:NN \XeTeXpicfile        \tex_XeTeXpicfile:D
868  \__kernel_primitive:NN \XeTeXrevision       \tex_XeTeXrevision:D
869  \__kernel_primitive:NN \XeTeXselectorname   \tex_XeTeXselectorname:D

```

```

870  \__kernel_primitive:NN \XeTeXtracingfonts      \tex_XeTeXtracingfonts:D
871  \__kernel_primitive:NN \XeTeXupwardsmode       \tex_XeTeXupwardsmode:D
872  \__kernel_primitive:NN \XeTeXuseglyphmetrics   \tex_XeTeXuseglyphmetrics:D
873  \__kernel_primitive:NN \XeTeXvariation        \tex_XeTeXvariation:D
874  \__kernel_primitive:NN \XeTeXvariationdefault  \tex_XeTeXvariationdefault:D
875  \__kernel_primitive:NN \XeTeXvariationmax     \tex_XeTeXvariationmax:D
876  \__kernel_primitive:NN \XeTeXvariationmin     \tex_XeTeXvariationmin:D
877  \__kernel_primitive:NN \XeTeXvariationname    \tex_XeTeXvariationname:D
878  \__kernel_primitive:NN \XeTeXversion         \tex_XeTeXversion:D

```

Primitives from pdftEX that X<sub>E</sub>T<sub>E</sub>X renames: also helps with LuaT<sub>E</sub>X.

```

879  \__kernel_primitive:NN \creationdate          \tex_creationdate:D
880  \__kernel_primitive:NN \elapsedtime         \tex_elapsedtime:D
881  \__kernel_primitive:NN \filedump            \tex_filedump:D
882  \__kernel_primitive:NN \filemoddate        \tex_filemoddate:D
883  \__kernel_primitive:NN \filesize             \tex_filesize:D
884  \__kernel_primitive:NN \mdfivesum          \tex_md5sum:D
885  \__kernel_primitive:NN \ifprimitive        \tex_ifprimitive:D
886  \__kernel_primitive:NN \primitive           \tex_primitive:D
887  \__kernel_primitive:NN \resettimer          \tex_resettimer:D
888  \__kernel_primitive:NN \shellescape         \tex_shellescape:D

```

Primitives from LuaT<sub>E</sub>X, some of which have been ported back to X<sub>E</sub>T<sub>E</sub>X.

```

889  \__kernel_primitive:NN \alignmark            \tex_alignmark:D
890  \__kernel_primitive:NN \aligntab            \tex_aligntab:D
891  \__kernel_primitive:NN \attribute           \tex_attribute:D
892  \__kernel_primitive:NN \attributedef        \tex_attributedef:D
893  \__kernel_primitive:NN \automaticdiscretionary
894  \tex_automaticdiscretionary:D
895  \__kernel_primitive:NN \automatichyphenmode \tex_automatichyphenmode:D
896  \__kernel_primitive:NN \automatichyphenpenalty
897  \tex_automatichyphenpenalty:D
898  \__kernel_primitive:NN \beginncsname        \tex_beginncsname:D
899  \__kernel_primitive:NN \bodydir            \tex_bodydir:D
900  \__kernel_primitive:NN \bodydirection      \tex_bodydirection:D
901  \__kernel_primitive:NN \boxdir              \tex_boxdir:D
902  \__kernel_primitive:NN \boxdirection       \tex_boxdirection:D
903  \__kernel_primitive:NN \breakafterdirmode  \tex_breakafterdirmode:D
904  \__kernel_primitive:NN \catcodetable       \tex_catcodetable:D
905  \__kernel_primitive:NN \clearmarks         \tex_clearmarks:D
906  \__kernel_primitive:NN \crampeddisplaystyle \tex_crampeddisplaystyle:D
907  \__kernel_primitive:NN \crampedscriptscriptstyle
908  \tex_crampedscriptscriptstyle:D
909  \__kernel_primitive:NN \crampedscriptstyle  \tex_crampedscriptstyle:D
910  \__kernel_primitive:NN \crampedtextstyle    \tex_crampedtextstyle:D
911  \__kernel_primitive:NN \csstring           \tex_csstring:D
912  \__kernel_primitive:NN \directlua          \tex_directlua:D
913  \__kernel_primitive:NN \dviextension       \tex_dviextension:D
914  \__kernel_primitive:NN \dvifeedback       \tex_dvifeedback:D
915  \__kernel_primitive:NN \dvivariable        \tex_dvivariable:D
916  \__kernel_primitive:NN \etoksapp           \tex_etoksapp:D
917  \__kernel_primitive:NN \etokspre           \tex_etokspre:D
918  \__kernel_primitive:NN \exceptionpenalty   \tex_exceptionpenalty:D
919  \__kernel_primitive:NN \explicithyphenpenalty \tex_explicithyphenpenalty:D
920  \__kernel_primitive:NN \expanded            \tex_expanded:D

```

```

921  \__kernel_primitive:NN \explicitdiscretionary \tex_explicitdiscretionary:D
922  \__kernel_primitive:NN \firstvalidlanguage \tex_firstvalidlanguage:D
923  \__kernel_primitive:NN \fontid \tex_fontid:D
924  \__kernel_primitive:NN \formatname \tex_formatname:D
925  \__kernel_primitive:NN \hjcode \tex_hjcode:D
926  \__kernel_primitive:NN \hpack \tex_hpack:D
927  \__kernel_primitive:NN \hyphenationbounds \tex_hyphenationbounds:D
928  \__kernel_primitive:NN \hyphenationmin \tex_hyphenationmin:D
929  \__kernel_primitive:NN \hyphenpenaltymode \tex_hyphenpenaltymode:D
930  \__kernel_primitive:NN \gleaders \tex_gleaders:D
931  \__kernel_primitive:NN \ifcondition \tex_ifcondition:D
932  \__kernel_primitive:NN \immediateassigned \tex_immediateassigned:D
933  \__kernel_primitive:NN \immediateassignment \tex_immediateassignment:D
934  \__kernel_primitive:NN \initcatcodetable \tex_initcatcodetable:D
935  \__kernel_primitive:NN \lastnamedcs \tex_lastnamedcs:D
936  \__kernel_primitive:NN \latelua \tex_latelua:D
937  \__kernel_primitive:NN \lateluafunction \tex_lateluafunction:D
938  \__kernel_primitive:NN \leftghost \tex_leftghost:D
939  \__kernel_primitive:NN \letcharcode \tex_letcharcode:D
940  \__kernel_primitive:NN \linedir \tex_linedir:D
941  \__kernel_primitive:NN \linedirection \tex_linedirection:D
942  \__kernel_primitive:NN \localbrokenpenalty \tex_localbrokenpenalty:D
943  \__kernel_primitive:NN \localinterlinepenalty \tex_localinterlinepenalty:D
944  \__kernel_primitive:NN \luabytecode \tex_luabytecode:D
945  \__kernel_primitive:NN \luabytecodecall \tex_luabytecodecall:D
946  \__kernel_primitive:NN \luacopyinputnodes \tex_luacopyinputnodes:D
947  \__kernel_primitive:NN \luadef \tex_luadef:D
948  \__kernel_primitive:NN \localleftbox \tex_localleftbox:D
949  \__kernel_primitive:NN \localrightbox \tex_localrightbox:D
950  \__kernel_primitive:NN \luaescapestring \tex_luaescapestring:D
951  \__kernel_primitive:NN \luafunction \tex_luafunction:D
952  \__kernel_primitive:NN \luafunctioncall \tex_luafunctioncall:D
953  \__kernel_primitive:NN \luatexbanner \tex_luatexbanner:D
954  \__kernel_primitive:NN \luatexrevision \tex_luatexrevision:D
955  \__kernel_primitive:NN \luatexversion \tex_luatexversion:D
956  \__kernel_primitive:NN \mathdelimitersmode \tex_mathdelimitersmode:D
957  \__kernel_primitive:NN \mathdir \tex_mathdir:D
958  \__kernel_primitive:NN \mathdirection \tex_mathdirection:D
959  \__kernel_primitive:NN \mathdisplayskipmode \tex_mathdisplayskipmode:D
960  \__kernel_primitive:NN \matheqnogapstep \tex_matheqnogapstep:D
961  \__kernel_primitive:NN \mathnolimitsmode \tex_mathnolimitsmode:D
962  \__kernel_primitive:NN \mathoption \tex_mathoption:D
963  \__kernel_primitive:NN \mathpenaltiesmode \tex_mathpenaltiesmode:D
964  \__kernel_primitive:NN \mathrulesfam \tex_mathrulesfam:D
965  \__kernel_primitive:NN \mathscriptsmode \tex_mathscriptsmode:D
966  \__kernel_primitive:NN \mathscriptboxmode \tex_mathscriptboxmode:D
967  \__kernel_primitive:NN \mathscriptcharchmode \tex_mathscriptcharchmode:D
968  \__kernel_primitive:NN \mathstyle \tex_mathstyle:D
969  \__kernel_primitive:NN \mathsurroundmode \tex_mathsurroundmode:D
970  \__kernel_primitive:NN \mathsurroundskip \tex_mathsurroundskip:D
971  \__kernel_primitive:NN \nohrule \tex_noahrule:D
972  \__kernel_primitive:NN \nokerns \tex_nokerns:D
973  \__kernel_primitive:NN \noligs \tex_noligs:D
974  \__kernel_primitive:NN \nospaces \tex_nospaces:D

```

```

975  \__kernel_primitive:NN \novrule           \tex_novrule:D
976  \__kernel_primitive:NN \outputbox          \tex_outputbox:D
977  \__kernel_primitive:NN \pagebottomoffset  \tex_pagebottomoffset:D
978  \__kernel_primitive:NN \pagedir            \tex_pagedir:D
979  \__kernel_primitive:NN \pagedirection     \tex_pagedirection:D
980  \__kernel_primitive:NN \pageleftoffset    \tex_pageleftoffset:D
981  \__kernel_primitive:NN \pagerightoffset   \tex_pagerightoffset:D
982  \__kernel_primitive:NN \pagetopoffset     \tex_pagetopoffset:D
983  \__kernel_primitive:NN \pardir             \tex_pardir:D
984  \__kernel_primitive:NN \pardirection      \tex_pardirection:D
985  \__kernel_primitive:NN \pdfextension       \tex_pdfextension:D
986  \__kernel_primitive:NN \pdffeedback        \tex_pdffeedback:D
987  \__kernel_primitive:NN \pdfvariable        \tex_pdfvariable:D
988  \__kernel_primitive:NN \postexhyphenchar   \tex_postexhyphenchar:D
989  \__kernel_primitive:NN \posthyphenchar     \tex_posthyphenchar:D
990  \__kernel_primitive:NN \prebinoppenalty    \tex_prebinoppenalty:D
991  \__kernel_primitive:NN \predisplaygapfactor \tex_predisplaygapfactor:D
992  \__kernel_primitive:NN \preexhyphenchar    \tex_preexhyphenchar:D
993  \__kernel_primitive:NN \prehyphenchar      \tex_prehyphenchar:D
994  \__kernel_primitive:NN \prerelpenalty      \tex_prerelpenalty:D
995  \__kernel_primitive:NN \rightghost         \tex_rightghost:D
996  \__kernel_primitive:NN \savecatcodetable  \tex_savecatcodetable:D
997  \__kernel_primitive:NN \scantexttokens    \tex_scantexttokens:D
998  \__kernel_primitive:NN \setfontid          \tex_setfontid:D
999  \__kernel_primitive:NN \shapemode          \tex_shapemode:D
1000 \__kernel_primitive:NN \suppressifcsnameerror \tex_suppressifcsnameerror:D
1001 \__kernel_primitive:NN \suppresslongerror   \tex_suppresslongerror:D
1002 \__kernel_primitive:NN \suppressmathparerror \tex_suppressmathparerror:D
1003 \__kernel_primitive:NN \suppressoutererror  \tex_suppressoutererror:D
1004 \__kernel_primitive:NN \suppressprimitiveerror \tex_suppressprimitiveerror:D
1005 \__kernel_primitive:NN \textdir              \tex_textdir:D
1006 \__kernel_primitive:NN \textdirection       \tex_textdirection:D
1007 \__kernel_primitive:NN \toksapp             \tex_toksapp:D
1008 \__kernel_primitive:NN \tokspre             \tex_tokspre:D
1009 \__kernel_primitive:NN \tpack               \tex_tpck:D
1010 \__kernel_primitive:NN \vpack               \tex_vpack:D
1011 \__kernel_primitive:NN \vpack

```

Primitives from pdfTeX that LuaTeX renames.

```

1012 \__kernel_primitive:NN \adjustspacing       \tex_adjustspacing:D
1013 \__kernel_primitive:NN \copyfont           \tex_copyfont:D
1014 \__kernel_primitive:NN \draftmode          \tex_draftmode:D
1015 \__kernel_primitive:NN \expandglyphsinfont \tex_fonthead:D
1016 \__kernel_primitive:NN \ifabsdim           \tex_ifabsdim:D
1017 \__kernel_primitive:NN \ifabsnum           \tex_ifabsnum:D
1018 \__kernel_primitive:NN \ignoreligaturesinfont \tex_ignoreligaturesinfont:D
1019 \__kernel_primitive:NN \insertht           \tex_insertht:D
1020 \__kernel_primitive:NN \lastsavedboxresourceindex
1021 \__kernel_primitive:NN \tex_pdflastxform:D
1022 \__kernel_primitive:NN \lastsavedimageresourceindex
1023 \__kernel_primitive:NN \tex_pdflastximage:D
1024 \__kernel_primitive:NN \lastsavedimageresourcepages
1025 \__kernel_primitive:NN \tex_pdflastximagepages:D
1026 \__kernel_primitive:NN \lastxpos            \tex_lastxpos:D
1027 \__kernel_primitive:NN \lastypos            \tex_lastypos:D

```

```

1028  \__kernel_primitive:NN \normaldeviate           \tex_normaldeviate:D
1029  \__kernel_primitive:NN \outputmode             \tex_pdfoutput:D
1030  \__kernel_primitive:NN \pageheight              \tex_pageheight:D
1031  \__kernel_primitive:NN \pagewidth               \tex_pagewith:D
1032  \__kernel_primitive:NN \protrudechars          \tex_protrudechars:D
1033  \__kernel_primitive:NN \pxdimen                 \tex_pxdimen:D
1034  \__kernel_primitive:NN \randomseed              \tex_randomseed:D
1035  \__kernel_primitive:NN \useboxresource          \tex_pdfrefxform:D
1036  \__kernel_primitive:NN \useimageresource        \tex_pdfrefximage:D
1037  \__kernel_primitive:NN \savepos                 \tex_savepos:D
1038  \__kernel_primitive:NN \saveboxresource          \tex_pdfform:D
1039  \__kernel_primitive:NN \saveimageresource        \tex_pdffimage:D
1040  \__kernel_primitive:NN \setrandomseed            \tex_setrandomseed:D
1041  \__kernel_primitive:NN \tracingfonts             \tex_tracingfonts:D
1042  \__kernel_primitive:NN \uniformdeviate           \tex_uniformdeviate:D

```

The set of Unicode math primitives were introduced by X<sub>E</sub>T<sub>E</sub>X and LuaT<sub>E</sub>X in a somewhat complex fashion: a few first as \XeTeX... which were then renamed with LuaT<sub>E</sub>X having a lot more. These names now all start \U... and mainly \Umath....

```

1043  \__kernel_primitive:NN \Uchar                  \tex_Uchar:D
1044  \__kernel_primitive:NN \Ucharcat                \tex_Ucharcat:D
1045  \__kernel_primitive:NN \Udelcode                \tex_Udelcode:D
1046  \__kernel_primitive:NN \Udelcodenum             \tex_Udelcodenum:D
1047  \__kernel_primitive:NN \Udelimiter               \tex_Udelimiter:D
1048  \__kernel_primitive:NN \Udelimiterover          \tex_Udelimiterover:D
1049  \__kernel_primitive:NN \Udelimiterunder         \tex_Udelimiterunder:D
1050  \__kernel_primitive:NN \Uhextensible            \tex_Uhextensible:D
1051  \__kernel_primitive:NN \Umathaccent              \tex_Umathaccent:D
1052  \__kernel_primitive:NN \Umathaxis                \tex_Umathaxis:D
1053  \__kernel_primitive:NN \Umathbinbinspacing       \tex_Umathbinbinspacing:D
1054  \__kernel_primitive:NN \Umathbinclosespacing     \tex_Umathbinclosespacing:D
1055  \__kernel_primitive:NN \Umathbininnerspacing      \tex_Umathbininnerspacing:D
1056  \__kernel_primitive:NN \Umathbinopenspacing       \tex_Umathbinopenspacing:D
1057  \__kernel_primitive:NN \Umathbinopspacing          \tex_Umathbinopspacing:D
1058  \__kernel_primitive:NN \Umathbinordspacing        \tex_Umathbinordspacing:D
1059  \__kernel_primitive:NN \Umathbinpunctspacing      \tex_Umathbinpunctspacing:D
1060  \__kernel_primitive:NN \Umathbinrelspacing        \tex_Umathbinrelspacing:D
1061  \__kernel_primitive:NN \Umathchar                 \tex_Umathchar:D
1062  \__kernel_primitive:NN \Umathcharclass            \tex_Umathcharclass:D
1063  \__kernel_primitive:NN \Umathchardef              \tex_Umathchardef:D
1064  \__kernel_primitive:NN \Umathcharfam              \tex_Umathcharfam:D
1065  \__kernel_primitive:NN \Umathcharnum              \tex_Umathcharnum:D
1066  \__kernel_primitive:NN \Umathcharnumdef            \tex_Umathcharnumdef:D
1067  \__kernel_primitive:NN \Umathcharslot              \tex_Umathcharslot:D
1068  \__kernel_primitive:NN \Umathclosebinspacing      \tex_Umathclosebinspacing:D
1069  \__kernel_primitive:NN \Umathcloseclosespacing    \tex_Umathcloseclosespacing:D
1070   \tex_Umathcloseclosespacing:D
1071  \__kernel_primitive:NN \Umathcloseinnerspacing     \tex_Umathcloseinnerspacing:D
1072   \tex_Umathcloseinnerspacing:D
1073  \__kernel_primitive:NN \Umathcloseopenspacing     \tex_Umathcloseopenspacing:D
1074  \__kernel_primitive:NN \Umathcloseopspacing        \tex_Umathcloseopspacing:D
1075  \__kernel_primitive:NN \Umathcloseordspacing      \tex_Umathcloseordspacing:D
1076  \__kernel_primitive:NN \Umathclosepunctspacing    \tex_Umathclosepunctspacing:D
1077   \tex_Umathclosepunctspacing:D

```

```

1078  \__kernel_primitive:NN \Umathcloserelspaceing  \tex_Umathcloserelspaceing:D
1079  \__kernel_primitive:NN \Umathcode          \tex_Umathcode:D
1080  \__kernel_primitive:NN \Umathcodenum        \tex_Umathcodenum:D
1081  \__kernel_primitive:NN \Umathconnectoroverlapmin
1082      \tex_Umathconnectoroverlapmin:D
1083  \__kernel_primitive:NN \Umathfractiondelsize  \tex_Umathfractiondelsize:D
1084  \__kernel_primitive:NN \Umathfractiondenomdown
1085      \tex_Umathfractiondenomdown:D
1086  \__kernel_primitive:NN \Umathfractiondenomvgap
1087      \tex_Umathfractiondenomvgap:D
1088  \__kernel_primitive:NN \Umathfractionnumup    \tex_Umathfractionnumup:D
1089  \__kernel_primitive:NN \Umathfractionnumvgap  \tex_Umathfractionnumvgap:D
1090  \__kernel_primitive:NN \Umathfractionrule     \tex_Umathfractionrule:D
1091  \__kernel_primitive:NN \Umathinnerbinspaceing \tex_Umathinnerbinspaceing:D
1092  \__kernel_primitive:NN \Umathinnerclosespaceing
1093      \tex_Umathinnerclosespaceing:D
1094  \__kernel_primitive:NN \Umathinnerinnerspaceing
1095      \tex_Umathinnerinnerspaceing:D
1096  \__kernel_primitive:NN \Umathinneropenspaceing \tex_Umathinneropenspaceing:D
1097  \__kernel_primitive:NN \Umathinneropspacing   \tex_Umathinneropspacing:D
1098  \__kernel_primitive:NN \Umathinnerordspacing  \tex_Umathinnerordspacing:D
1099  \__kernel_primitive:NN \Umathinnerpunctspaceing
1100      \tex_Umathinnerpunctspaceing:D
1101  \__kernel_primitive:NN \Umathinnerrelspaceing \tex_Umathinnerrelspaceing:D
1102  \__kernel_primitive:NN \Umathlimitabovebgap  \tex_Umathlimitabovebgap:D
1103  \__kernel_primitive:NN \Umathlimitabovekern  \tex_Umathlimitabovekern:D
1104  \__kernel_primitive:NN \Umathlimitabovevgap  \tex_Umathlimitabovevgap:D
1105  \__kernel_primitive:NN \Umathlimitbelowbgap  \tex_Umathlimitbelowbgap:D
1106  \__kernel_primitive:NN \Umathlimitbelowkern  \tex_Umathlimitbelowkern:D
1107  \__kernel_primitive:NN \Umathlimitbelowvgap  \tex_Umathlimitbelowvgap:D
1108  \__kernel_primitive:NN \Umathnolimitssubfactor \tex_Umathnolimitssubfactor:D
1109  \__kernel_primitive:NN \Umathnolimitsupfactor  \tex_Umathnolimitsupfactor:D
1110  \__kernel_primitive:NN \Umathopbinspaceing   \tex_Umathopbinspaceing:D
1111  \__kernel_primitive:NN \Umathopclosespaceing \tex_Umathopclosespaceing:D
1112  \__kernel_primitive:NN \Umathopenbinspaceing \tex_Umathopenbinspaceing:D
1113  \__kernel_primitive:NN \Umathopenclosespaceing \tex_Umathopenclosespaceing:D
1114  \__kernel_primitive:NN \Umathopeninnerspaceing \tex_Umathopeninnerspaceing:D
1115  \__kernel_primitive:NN \Umathopenopenspaceing \tex_Umathopenopenspaceing:D
1116  \__kernel_primitive:NN \Umathopenopspacing   \tex_Umathopenopspacing:D
1117  \__kernel_primitive:NN \Umathopenordspacing  \tex_Umathopenordspacing:D
1118  \__kernel_primitive:NN \Umathopenpunctspaceing \tex_Umathopenpunctspaceing:D
1119  \__kernel_primitive:NN \Umathopenrelspaceing \tex_Umathopenrelspaceing:D
1120  \__kernel_primitive:NN \Umathoperatorsize   \tex_Umathoperatorsize:D
1121  \__kernel_primitive:NN \Umathopinnerspaceing \tex_Umathopinnerspaceing:D
1122  \__kernel_primitive:NN \Umathopopenspaceing  \tex_Umathopopenspaceing:D
1123  \__kernel_primitive:NN \Umathopopspaceing   \tex_Umathopopspaceing:D
1124  \__kernel_primitive:NN \Umathopordspacing   \tex_Umathopordspacing:D
1125  \__kernel_primitive:NN \Umathoppunctspaceing \tex_Umathoppunctspaceing:D
1126  \__kernel_primitive:NN \Umathoprelspaceing  \tex_Umathoprelspaceing:D
1127  \__kernel_primitive:NN \Umathordbinspaceing \tex_Umathordbinspaceing:D
1128  \__kernel_primitive:NN \Umathordclosespaceing \tex_Umathordclosespaceing:D
1129  \__kernel_primitive:NN \Umathordinnerspaceing \tex_Umathordinnerspaceing:D
1130  \__kernel_primitive:NN \Umathordopenspaceing \tex_Umathordopenspaceing:D
1131  \__kernel_primitive:NN \Umathordopspaceing  \tex_Umathordopspaceing:D

```

```

1132  \__kernel_primitive:NN \Umathordordspacing      \tex_Umathordordspacing:D
1133  \__kernel_primitive:NN \Umathordpunctspacing   \tex_Umathordpunctspacing:D
1134  \__kernel_primitive:NN \Umathordrelspacing    \tex_Umathordrelspacing:D
1135  \__kernel_primitive:NN \Umathoverbarkern     \tex_Umathoverbarkern:D
1136  \__kernel_primitive:NN \Umathoverbarrule     \tex_Umathoverbarrule:D
1137  \__kernel_primitive:NN \Umathoverbarvgap     \tex_Umathoverbarvgap:D
1138  \__kernel_primitive:NN \Umathoverdelimitergap
1139          \tex_Umathoverdelimitergap:D
1140  \__kernel_primitive:NN \Umathoverdelimitervgap
1141          \tex_Umathoverdelimitervgap:D
1142  \__kernel_primitive:NN \Umathpunctbinspacing  \tex_Umathpunctbinspacing:D
1143  \__kernel_primitive:NN \Umathpunctclosespacing
1144          \tex_Umathpunctclosespacing:D
1145  \__kernel_primitive:NN \Umathpunctinnerspacing
1146          \tex_Umathpunctinnerspacing:D
1147  \__kernel_primitive:NN \Umathpunctopenspacing \tex_Umathpunctopenspacing:D
1148  \__kernel_primitive:NN \Umathpuncttopspacing   \tex_Umathpuncttopspacing:D
1149  \__kernel_primitive:NN \Umathpunctordspacing   \tex_Umathpunctordspacing:D
1150  \__kernel_primitive:NN \Umathpunctpunctspacing
1151          \tex_Umathpunctpunctspacing:D
1152  \__kernel_primitive:NN \Umathpunctrelspacing  \tex_Umathpunctrelspacing:D
1153  \__kernel_primitive:NN \Umathquad           \tex_Umathquad:D
1154  \__kernel_primitive:NN \Umathradicaldegreeafter
1155          \tex_Umathradicaldegreeafter:D
1156  \__kernel_primitive:NN \Umathradicaldegreebefore
1157          \tex_Umathradicaldegreebefore:D
1158  \__kernel_primitive:NN \Umathradicaldegreeraise
1159          \tex_Umathradicaldegreeraise:D
1160  \__kernel_primitive:NN \Umathradicalkern      \tex_Umathradicalkern:D
1161  \__kernel_primitive:NN \Umathradicalrule     \tex_Umathradicalrule:D
1162  \__kernel_primitive:NN \Umathradicalvgap     \tex_Umathradicalvgap:D
1163  \__kernel_primitive:NN \Umathrelbinspacing   \tex_Umathrelbinspacing:D
1164  \__kernel_primitive:NN \Umathrelclosespacing \tex_Umathrelclosespacing:D
1165  \__kernel_primitive:NN \Umathrelinnerspacing \tex_Umathrelinnerspacing:D
1166  \__kernel_primitive:NN \Umathrelopspacing    \tex_Umathrelopspacing:D
1167  \__kernel_primitive:NN \Umathrelopspacing    \tex_Umathrelopspacing:D
1168  \__kernel_primitive:NN \Umathrelordspacing  \tex_Umathrelordspacing:D
1169  \__kernel_primitive:NN \Umathrelpunctspacing \tex_Umathrelpunctspacing:D
1170  \__kernel_primitive:NN \Umathrelrelspacing   \tex_Umathrelrelspacing:D
1171  \__kernel_primitive:NN \Umathskewedfractionhgap
1172          \tex_Umathskewedfractionhgap:D
1173  \__kernel_primitive:NN \Umathskewedfractionvgap
1174          \tex_Umathskewedfractionvgap:D
1175  \__kernel_primitive:NN \Umathspaceafterscript \tex_Umathspaceafterscript:D
1176  \__kernel_primitive:NN \Umathstackdenomdown \tex_Umathstackdenomdown:D
1177  \__kernel_primitive:NN \Umathstacknumup      \tex_Umathstacknumup:D
1178  \__kernel_primitive:NN \Umathstackvgap       \tex_Umathstackvgap:D
1179  \__kernel_primitive:NN \Umathsubshiftdown   \tex_Umathsubshiftdown:D
1180  \__kernel_primitive:NN \Umathsubshiftdrop   \tex_Umathsubshiftdrop:D
1181  \__kernel_primitive:NN \Umathsubsupshiftdown \tex_Umathsubsupshiftdown:D
1182  \__kernel_primitive:NN \Umathsubsupvgap     \tex_Umathsubsupvgap:D
1183  \__kernel_primitive:NN \Umathsubtopmax     \tex_Umathsubtopmax:D
1184  \__kernel_primitive:NN \Umathsupbottommin   \tex_Umathsupbottommin:D
1185  \__kernel_primitive:NN \Umathsupshiftdrop   \tex_Umathsupshiftdrop:D

```

```

1186  \__kernel_primitive:NN \Umathsupshiftup      \tex_Umathsupshiftup:D
1187  \__kernel_primitive:NN \Umathsupsubbottommax \tex_Umathsupsubbottommax:D
1188  \__kernel_primitive:NN \Umathunderbarkern   \tex_Umathunderbarkern:D
1189  \__kernel_primitive:NN \Umathunderbarrule   \tex_Umathunderbarrule:D
1190  \__kernel_primitive:NN \Umathunderbarvgap   \tex_Umathunderbarvgap:D
1191  \__kernel_primitive:NN \Umathunderdelimterbgap
1192    \tex_Umathunderdelimterbgap:D
1193  \__kernel_primitive:NN \Umathunderdelimitervgap
1194    \tex_Umathunderdelimitervgap:D
1195  \__kernel_primitive:NN \Unosubscript        \tex_Unosubscript:D
1196  \__kernel_primitive:NN \Unosuperscript       \tex_Unosuperscript:D
1197  \__kernel_primitive:NN \Uoverdelimiter     \tex_Uoverdelimiter:D
1198  \__kernel_primitive:NN \Urational          \tex_Urational:D
1199  \__kernel_primitive:NN \Uroot              \tex_Uroot:D
1200  \__kernel_primitive:NN \Uskewed            \tex_Uskewed:D
1201  \__kernel_primitive:NN \Uskewedwithdelims \tex_Uskewedwithdelims:D
1202  \__kernel_primitive:NN \Ustack              \tex_Ustack:D
1203  \__kernel_primitive:NN \Ustartdisplaymath  \tex_Ustartdisplaymath:D
1204  \__kernel_primitive:NN \Ustartmath         \tex_Ustartmath:D
1205  \__kernel_primitive:NN \Ustopdisplaymath   \tex_Ustopdisplaymath:D
1206  \__kernel_primitive:NN \Ustopmath          \tex_Ustopmath:D
1207  \__kernel_primitive:NN \Usubscript         \tex_Usubscript:D
1208  \__kernel_primitive:NN \Usuperscript       \tex_Usuperscript:D
1209  \__kernel_primitive:NN \Uunderdelimiter   \tex_Uunderdelimiter:D
1210  \__kernel_primitive:NN \Uvextensible      \tex_Uvextensible:D

```

Primitives from plEX.

```

1211  \__kernel_primitive:NN \autospacing        \tex_autospacing:D
1212  \__kernel_primitive:NN \autoxspacing      \tex_autoxspacing:D
1213  \__kernel_primitive:NN \dtou              \tex_dtou:D
1214  \__kernel_primitive:NN \epTeXinputencoding \tex_epTeXinputencoding:D
1215  \__kernel_primitive:NN \epTeXversion      \tex_epTeXversion:D
1216  \__kernel_primitive:NN \euc                \tex_euc:D
1217  \__kernel_primitive:NN \ifdbox            \tex_ifdbox:D
1218  \__kernel_primitive:NN \ifddir            \tex_ifddir:D
1219  \__kernel_primitive:NN \ifmdir            \tex_ifmdir:D
1220  \__kernel_primitive:NN \iftbox             \tex_iftbox:D
1221  \__kernel_primitive:NN \iftdir             \tex_iftdir:D
1222  \__kernel_primitive:NN \ifybox             \tex_ifybox:D
1223  \__kernel_primitive:NN \ifydir             \tex_ifydir:D
1224  \__kernel_primitive:NN \inhibitglue       \tex_inhibitglue:D
1225  \__kernel_primitive:NN \inhibitxspcode   \tex_inhibitxspcode:D
1226  \__kernel_primitive:NN \jcharwidopenalty  \tex_jcharwidopenalty:D
1227  \__kernel_primitive:NN \jfam               \tex_jfam:D
1228  \__kernel_primitive:NN \jfont              \tex_jfont:D
1229  \__kernel_primitive:NN \jis                \tex_jis:D
1230  \__kernel_primitive:NN \kanjiskip         \tex_kanjiskip:D
1231  \__kernel_primitive:NN \kansuji           \tex_kansuji:D
1232  \__kernel_primitive:NN \kansujichar      \tex_kansujichar:D
1233  \__kernel_primitive:NN \kcatcode          \tex_kcatcode:D
1234  \__kernel_primitive:NN \kuten              \tex_kuten:D
1235  \__kernel_primitive:NN \noautospacing     \tex_noautospacing:D
1236  \__kernel_primitive:NN \noautoxspacing    \tex_noautoxspacing:D
1237  \__kernel_primitive:NN \postbreakpenalty  \tex_postbreakpenalty:D
1238  \__kernel_primitive:NN \prebreakpenalty   \tex_prebreakpenalty:D

```

```

1239  \__kernel_primitive:NN \ptexminorversion          \tex_ptexminorversion:D
1240  \__kernel_primitive:NN \ptexrevision             \tex_ptexrevision:D
1241  \__kernel_primitive:NN \ptexversion              \tex_ptexversion:D
1242  \__kernel_primitive:NN \showmode                \tex_showmode:D
1243  \__kernel_primitive:NN \sjis                   \tex_sjis:D
1244  \__kernel_primitive:NN \tate                    \tex_tate:D
1245  \__kernel_primitive:NN \tbaselineshift         \tex_tbaselineshift:D
1246  \__kernel_primitive:NN \tfont                  \tex_tfont:D
1247  \__kernel_primitive:NN \xkanjiskip            \tex_xkanjiskip:D
1248  \__kernel_primitive:NN \xspcode               \tex_xspcode:D
1249  \__kernel_primitive:NN \ybaselineshift         \tex_ybaselineshift:D
1250  \__kernel_primitive:NN \yoko                  \tex_yoko:D

```

Primitives from upTeX.

```

1251  \__kernel_primitive:NN \disablecjktoken        \tex_disablecjktoken:D
1252  \__kernel_primitive:NN \enablecjktoken          \tex_enablecjktoken:D
1253  \__kernel_primitive:NN \forcecjktoken           \tex_forcecjktoken:D
1254  \__kernel_primitive:NN \kchar                 \tex_kchar:D
1255  \__kernel_primitive:NN \kchardef              \tex_kchardef:D
1256  \__kernel_primitive:NN \kuten                 \tex_kuten:D
1257  \__kernel_primitive:NN \ucs                  \tex_UCS:D
1258  \__kernel_primitive:NN \uptexrevision         \tex_uptexrevision:D
1259  \__kernel_primitive:NN \uptexversion          \tex_uptexversion:D

```

End of the “just the names” part of the source.

```

1260  </initex | names | package>
1261  <*initex | package>

```

The job is done: close the group (using the primitive renamed!).

```
1262  \tex_endgroup:D
```

LATEX<sub>2ε</sub> moves a few primitives, so these are sorted out. A convenient test for LATEX<sub>2ε</sub> is the \@@end saved primitive.

```

1263  <*package>
1264  \tex_ifdefined:D \@@end
1265  \tex_let:D \tex_end:D                      \@@end
1266  \tex_let:D \tex_everydisplay:D            \frozen@everydisplay
1267  \tex_let:D \tex_everymath:D              \frozen@everymath
1268  \tex_let:D \tex_hyphen:D                 \@@hyph
1269  \tex_let:D \tex_input:D                  \@@input
1270  \tex_let:D \tex_italiccorrection:D     \@@italiccorr
1271  \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 LATEX<sub>2ε</sub> is in use, we use it’s \@tfor loop here.

```

1272  \tex_ifdefined:D \@@shipout
1273    \tex_let:D \tex_shipout:D \@@shipout
1274    \tex_fi:D
1275    \tex_begingroup:D
1276      \tex_eodef:D \l_tmpa_t1 { \tex_string:D \shipout }
1277      \tex_eodef:D \l_tmpb_t1 { \tex_meaning:D \shipout }

```

```

1278   \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1279   \tex_else:D
1280     \tex_expandafter:D \otfor \tex_expandafter:D \tempa \tex_string:D :=
1281       \CROP@shipout
1282       \dup@shipout
1283       \GPTorg@shipout
1284       \LL@shipout
1285       \mem@oldshipout
1286       \opem@shipout
1287       \pgfpages@originalshipout
1288       \pr@shipout
1289       \Shipout
1290       \verso@orig@shipout
1291       \do
1292         {
1293           \tex_edef:D \l_tmpb_t1
1294             { \tex_expandafter:D \tex_meaning:D \tempa }
1295           \tex_ifx:D \l_tmpa_t1 \l_tmpb_t1
1296             \tex_global:D \tex_expandafter:D \tex_let:D
1297               \tex_expandafter:D \tex_shipout:D \tempa
1298             \tex_fi:D
1299           }
1300         \tex_fi:D
1301       \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.)

```

1302   \tex_let:D \tex_tracingfonts:D \tex_undefined:D
1303   \tex_ifdefined:D \pdftracingfonts
1304     \tex_let:D \tex_tracingfonts:D \pdftracingfonts
1305   \tex_else:D
1306     \tex_ifdefined:D \tex_directlua:D
1307       \tex_directlua:D { tex.enableprimitives("@@", {"tracingfonts"}) }
1308     \tex_let:D \tex_tracingfonts:D \luatextracingfonts
1309   \tex_fi:D
1310 \tex_fi:D
1311 \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!

```

1312 \tex_ifdefined:D \luatexsuppressfontnotfounderror
1313   \tex_let:D \tex_alignmark:D          \luatexalignmark
1314   \tex_let:D \tex_aligntab:D          \luatexaligntab
1315   \tex_let:D \tex_attribute:D        \luatexattribute
1316   \tex_let:D \tex_attributedef:D      \luatexattributedef
1317   \tex_let:D \tex_catcodetable:D     \luatexcatcodetable
1318   \tex_let:D \tex_clearmarks:D       \luatexclearmarks
1319   \tex_let:D \tex_crampeddisplaystyle:D \luatexcrampeddisplaystyle
1320   \tex_let:D \tex_crampedscriptscriptstyle:D

```

```

1321     \luatexcrampedscriptscriptstyle
1322     \tex_let:D \tex_crampedscriptstyle:D \luatexcrampedscriptstyle
1323     \tex_let:D \tex_crampedtextstyle:D \luatexcrampedtextstyle
1324     \tex_let:D \tex_fontid:D \luatexfontid
1325     \tex_let:D \tex_formatname:D \luatexformatname
1326     \tex_let:D \tex_gleaders:D \luatexleaders
1327     \tex_let:D \tex_initcatcodetable:D \luatexinitcatcodetable
1328     \tex_let:D \tex_latelua:D \luatexlatelua
1329     \tex_let:D \tex_luaescapestring:D \luatexluaescapestring
1330     \tex_let:D \tex_luafunction:D \luatexluafunction
1331     \tex_let:D \tex_mathstyle:D \luatexmathstyle
1332     \tex_let:D \tex_nokerns:D \luatexnokerns
1333     \tex_let:D \tex_noligs:D \luatexnoligs
1334     \tex_let:D \tex_outputbox:D \luatexoutputbox
1335     \tex_let:D \tex_pageleftoffset:D \luatexpageleftoffset
1336     \tex_let:D \tex_pagetopoffset:D \luatexpagetopoffset
1337     \tex_let:D \tex_postexhyphenchar:D \luatexpostexhyphenchar
1338     \tex_let:D \tex_posthyphenchar:D \luatexposthyphenchar
1339     \tex_let:D \tex_preehyphenchar:D \luatexpreehyphenchar
1340     \tex_let:D \tex_prehyphenchar:D \luatexprehyphenchar
1341     \tex_let:D \tex_savecatcodetable:D \luatexsavecatcodetable
1342     \tex_let:D \tex_scantexttokens:D \luatexscantexttokens
1343     \tex_let:D \tex_suppressifcsnameerror:D
        \luatexsuppressifcsnameerror
1344     \tex_let:D \tex_suppresslongerror:D \luatexsuppresslongerror
1345     \tex_let:D \tex_suppressmathparerror:D
        \luatexsuppressmathparerror
1346     \tex_let:D \tex_suppressoutererror:D \luatexsuppressoutererror
1347     \tex_let:D \tex_Uchar:D \luatexUchar
1348     \tex_let:D \tex_suppressfontnotfounderror:D
        \luatexsuppressfontnotfounderror
1349     \tex_let:D \tex_suppressfontnotfounderror:D
        \luatexsuppressfontnotfounderror
1350
1351
1352     \tex_let:D \tex_bodydir:D \luatexbodydir
1353     \tex_let:D \tex_boxdir:D \luatexboxdir
1354     \tex_let:D \tex_leftghost:D \luatexleftghost
1355     \tex_let:D \tex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1356     \tex_let:D \tex_localinterlinepenalty:D
        \luatexlocalinterlinepenalty
1357     \tex_let:D \tex_localleftbox:D \luatexlocalleftbox
1358     \tex_let:D \tex_localrightbox:D \luatexlocalrightbox
1359     \tex_let:D \tex_mathdir:D \luatexmathdir
1360     \tex_let:D \tex_pagebottomoffset:D \luatexpagebottomoffset
1361     \tex_let:D \tex_pagedir:D \luatexpagedir
1362     \tex_let:D \tex_pageheight:D \luatexpageheight
1363     \tex_let:D \tex_pagerightoffset:D \luatexpagerightoffset
1364     \tex_let:D \tex_pagewidth:D \luatexpagewidth
1365     \tex_let:D \tex_pardir:D \luatexpardir
1366     \tex_let:D \tex_rightghost:D \luatexrightghost
1367     \tex_let:D \tex_textdir:D \luatextextdir
1368
1369 \tex_if:D

```

Which also covers those slightly odd ones.

```

1352     \tex_let:D \tex_bodydir:D \luatexbodydir
1353     \tex_let:D \tex_boxdir:D \luatexboxdir
1354     \tex_let:D \tex_leftghost:D \luatexleftghost
1355     \tex_let:D \tex_localbrokenpenalty:D \luatexlocalbrokenpenalty
1356     \tex_let:D \tex_localinterlinepenalty:D
        \luatexlocalinterlinepenalty
1357     \tex_let:D \tex_localleftbox:D \luatexlocalleftbox
1358     \tex_let:D \tex_localrightbox:D \luatexlocalrightbox
1359     \tex_let:D \tex_mathdir:D \luatexmathdir
1360     \tex_let:D \tex_pagebottomoffset:D \luatexpagebottomoffset
1361     \tex_let:D \tex_pagedir:D \luatexpagedir
1362     \tex_let:D \tex_pageheight:D \luatexpageheight
1363     \tex_let:D \tex_pagerightoffset:D \luatexpagerightoffset
1364     \tex_let:D \tex_pagewidth:D \luatexpagewidth
1365     \tex_let:D \tex_pardir:D \luatexpardir
1366     \tex_let:D \tex_rightghost:D \luatexrightghost
1367     \tex_let:D \tex_textdir:D \luatextextdir
1368
1369 \tex_if:D

```

Only pdf<sub>E</sub>X and Lu<sub>A</sub>T<sub>E</sub>X define \pdfmapfile and \pdfmapline: Tidy up the fact that some format-building processes leave a couple of questionable decisions about that!

```

1370 \tex_ifnum:D 0

```

```

1371   \tex_ifdefined:D \tex_pdftexversion:D 1 \tex_fi:D
1372   \tex_ifdefined:D \tex_luatexversion:D 1 \tex_fi:D
1373     = 0 %
1374   \tex_let:D \tex_mapfile:D \tex_undefined:D
1375   \tex_let:D \tex_mapline:D \tex_undefined:D
1376 \tex_fi:D
1377 </package>

```

Up to v0.80, LuaTeX defines the pdfTeX version data: rather confusing. Removing them means that \tex\_pdftexversion:D is a marker for pdfTeX alone: useful in engine-dependent code later.

```

1378 <*initex | package>
1379 \tex_ifdefined:D \tex_luatexversion:D
1380   \tex_let:D \tex_pdftexbanner:D \tex_undefined:D
1381   \tex_let:D \tex_pdftexrevision:D \tex_undefined:D
1382   \tex_let:D \tex_pdftexversion:D \tex_undefined:D
1383 \tex_fi:D
1384 </initex | package>

```

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.

```

1385 <*package>
1386 \tex_ifdefined:D \normalend
1387   \tex_let:D \tex_end:D \normalend
1388   \tex_let:D \tex_everyjob:D \normaleveryjob
1389   \tex_let:D \tex_input:D \normalinput
1390   \tex_let:D \tex_language:D \normallanguage
1391   \tex_let:D \tex_mathop:D \normalmathop
1392   \tex_let:D \tex_month:D \normalmonth
1393   \tex_let:D \tex_outer:D \normalouter
1394   \tex_let:D \tex_over:D \normalover
1395   \tex_let:D \tex_vcenter:D \normalvcenter
1396   \tex_let:D \tex_unexpanded:D \normalunexpanded
1397   \tex_let:D \tex_expanded:D \normalexpanded
1398 \tex_fi:D
1399 \tex_ifdefined:D \normalitaliccorrection
1400   \tex_let:D \tex_hoffset:D \normalhoffset
1401   \tex_let:D \tex_italiccorrection:D \normalitaliccorrection
1402   \tex_let:D \tex_voffset:D \normalvoffset
1403   \tex_let:D \tex_showtokens:D \normalshowtokens
1404   \tex_let:D \tex_bodydir:D \spac_directions_normal_body_dir
1405   \tex_let:D \tex_pagedir:D \spac_directions_normal_page_dir
1406 \tex_fi:D
1407 \tex_ifdefined:D \normalleft
1408   \tex_let:D \tex_left:D \normalleft
1409   \tex_let:D \tex_middle:D \normalmiddle
1410   \tex_let:D \tex_right:D \normalright
1411 \tex_fi:D
1412 </package>

```

## 2.1 Deprecated functions

Older versions of expl3 divided up primitives by “source”: that becomes very tricky with multiple parallel engine developments, so has been dropped. To cover the transition, we provide the older names here for a limited period (until the end of 2019).

To allow `\debug_on:n {<deprecation>}` to work we save the list of primitives into `\__kernel_primitives`:

```

1413  {*package}
1414  \tex_begingroup:D
1415    \tex_long:D \tex_def:D \use_i:nn #1#2 {#2}
1416    \tex_long:D \tex_def:D \use_none:n #1 { }
1417    \tex_long:D \tex_def:D \__kernel_primitive:NN #1#2
1418    {
1419      \tex_ifdefined:D #1
1420      \tex_expandafter:D \use_i:nn
1421      \tex_fi:D
1422      \use_none:n { \tex_global:D \tex_let:D #2 #1 }
1423    }
1424  \tex_xdef:D \__kernel_primitives:
1425  {
1426    \tex_unexpanded:D
1427    {
1428      \__kernel_primitive:NN \beginL
1429      \__kernel_primitive:NN \beginR
1430      \__kernel_primitive:NN \botmarks
1431      \__kernel_primitive:NN \clubpenalties
1432      \__kernel_primitive:NN \currentgrouplevel
1433      \__kernel_primitive:NN \currentgroupype
1434      \__kernel_primitive:NN \currentifbranch
1435      \__kernel_primitive:NN \currentiflevel
1436      \__kernel_primitive:NN \currentiftype
1437      \__kernel_primitive:NN \detokenize
1438      \__kernel_primitive:NN \dimexpr
1439      \__kernel_primitive:NN \displaywidowpenalties
1440        \etex_displaywidowpenalties:D
1441      \__kernel_primitive:NN \endL
1442      \__kernel_primitive:NN \endR
1443      \__kernel_primitive:NN \eTeXrevision
1444      \__kernel_primitive:NN \eTeXversion
1445      \__kernel_primitive:NN \everyeof
1446      \__kernel_primitive:NN \firstmarks
1447      \__kernel_primitive:NN \fontchardp
1448      \__kernel_primitive:NN \fontcharht
1449      \__kernel_primitive:NN \fontcharic
1450      \__kernel_primitive:NN \fontcharwd
1451      \__kernel_primitive:NN \glueexpr
1452      \__kernel_primitive:NN \glueshrink
1453      \__kernel_primitive:NN \glueshrinkorder
1454      \__kernel_primitive:NN \gluestretch
1455      \__kernel_primitive:NN \gluestretchorder
1456      \__kernel_primitive:NN \gluetomu
1457      \__kernel_primitive:NN \ifcsname
1458      \__kernel_primitive:NN \ifdefined
1459      \__kernel_primitive:NN \iffontchar
1460      \etex_beginL:D
1461      \etex_beginR:D
1462      \etex_botmarks:D
1463      \etex_clubpenalties:D
1464      \etex_currentgrouplevel:D
1465      \etex_currentgroupype:D
1466      \etex_currentifbranch:D
1467      \etex_currentiflevel:D
1468      \etex_currentiftype:D
1469      \etex_detokenize:D
1470      \etex_dimexpr:D
1471      \etex_endL:D
1472      \etex_endR:D
1473      \etex_eTeXrevision:D
1474      \etex_eTeXversion:D
1475      \etex_everyeof:D
1476      \etex_firstmarks:D
1477      \etex_fontchardp:D
1478      \etex_fontcharht:D
1479      \etex_fontcharic:D
1480      \etex_fontcharwd:D
1481      \etex_glueexpr:D
1482      \etex_glueshrink:D
1483      \etex_glueshrinkorder:D
1484      \etex_gluestretch:D
1485      \etex_gluestretchorder:D
1486      \etex_gluetomu:D
1487      \etex_ifcsname:D
1488      \etex_ifdefined:D
1489      \etex_iffontchar:D

```

```

1460  \__kernel_primitive:NN \interactionmode          \etex_interactionmode:D
1461  \__kernel_primitive:NN \interlinepenalties      \etex_interlinepenalties:D
1462  \__kernel_primitive:NN \lastlinefit            \etex_lastlinefit:D
1463  \__kernel_primitive:NN \lastnodetype           \etex_lastnodetype:D
1464  \__kernel_primitive:NN \marks                  \etex_marks:D
1465  \__kernel_primitive:NN \middle                 \etex_middle:D
1466  \__kernel_primitive:NN \muexpr                \etex_muexpr:D
1467  \__kernel_primitive:NN \mutoglu               \etex_mutoglu:D
1468  \__kernel_primitive:NN \numexpr               \etex_numexpr:D
1469  \__kernel_primitive:NN \pagediscards          \etex_pagediscards:D
1470  \__kernel_primitive:NN \parshapedimen         \etex_parshapedimen:D
1471  \__kernel_primitive:NN \parshapeindent        \etex_parshapeindent:D
1472  \__kernel_primitive:NN \parshapelen           \etex_parshapelen:D
1473  \__kernel_primitive:NN \predisplaydirection   \etex_predisplaydirection:D
1474  \__kernel_primitive:NN \protected              \etex_protected:D
1475  \__kernel_primitive:NN \readline               \etex_readline:D
1476  \__kernel_primitive:NN \savinghyphcodes       \etex_savinghyphcodes:D
1477  \__kernel_primitive:NN \savingvdiscards        \etex_savingvdiscards:D
1478  \__kernel_primitive:NN \scantokens             \etex_scantokens:D
1479  \__kernel_primitive:NN \showgroups            \etex_showgroups:D
1480  \__kernel_primitive:NN \showifs                \etex_showifs:D
1481  \__kernel_primitive:NN \showtokens            \etex_showtokens:D
1482  \__kernel_primitive:NN \splitbotmarks         \etex_splitbotmarks:D
1483  \__kernel_primitive:NN \spliddiscards         \etex_spliddiscards:D
1484  \__kernel_primitive:NN \splitfirstmarks       \etex_splitfirstmarks:D
1485  \__kernel_primitive:NN \TeXXeTstate           \etex_TeXXeTstate:D
1486  \__kernel_primitive:NN \topmarks               \etex_topmarks:D
1487  \__kernel_primitive:NN \tracingassigns        \etex_tracingassigns:D
1488  \__kernel_primitive:NN \tracinggroups          \etex_tracinggroups:D
1489  \__kernel_primitive:NN \tracingifs             \etex_tracingifs:D
1490  \__kernel_primitive:NN \tracingnesting         \etex_tracingnesting:D
1491  \__kernel_primitive:NN \tracingscantokens     \etex_tracingscantokens:D
1492  \__kernel_primitive:NN \unexpanded             \etex_unexpanded:D
1493  \__kernel_primitive:NN \unless                 \etex_unless:D
1494  \__kernel_primitive:NN \widowpenalties         \etex_widowpenalties:D
1495  \__kernel_primitive:NN \pdfannot              \pdftex_pdfannot:D
1496  \__kernel_primitive:NN \pdfcatalog            \pdftex_pdfcatalog:D
1497  \__kernel_primitive:NN \pdfcompresslevel      \pdftex_pdfcompresslevel:D
1498  \__kernel_primitive:NN \pdfcolorstack         \pdftex_pdfcolorstack:D
1499  \__kernel_primitive:NN \pdfcolorstackinit     \pdftex_pdfcolorstackinit:D
1500  \__kernel_primitive:NN \pdfcreationdate      \pdftex_pdfcreationdate:D
1501  \__kernel_primitive:NN \pdfdecimaldigits     \pdftex_pdfdecimaldigits:D
1502  \__kernel_primitive:NN \pdfdest                \pdftex_pdfdest:D
1503  \__kernel_primitive:NN \pdfdestmargin         \pdftex_pdfdestmargin:D
1504  \__kernel_primitive:NN \pdfendlink            \pdftex_pdfendlink:D
1505  \__kernel_primitive:NN \pdfendthread          \pdftex_pdfendthread:D
1506  \__kernel_primitive:NN \pdffontattr           \pdftex_pdffontattr:D
1507  \__kernel_primitive:NN \pdffontname           \pdftex_pdffontname:D
1508  \__kernel_primitive:NN \pdffontobjnum        \pdftex_pdffontobjnum:D
1509  \__kernel_primitive:NN \pdfgamma               \pdftex_pdfgamma:D
1510  \__kernel_primitive:NN \pdfimageapplygamma    \pdftex_pdfimageapplygamma:D
1511  \__kernel_primitive:NN \pdfimagegamma         \pdftex_pdfimagegamma:D
1512  \__kernel_primitive:NN \pdfgentounicode       \pdftex_pdfgentounicode:D
1513  \__kernel_primitive:NN \pdfglyphounicode      \pdftex_pdfglyphounicode:D

```

```

1514  \__kernel_primitive:NN \pdfhorigin          \pdftex_pdfhorigin:D
1515  \__kernel_primitive:NN \pdfimagehicolor    \pdftex_pdfimagehicolor:D
1516  \__kernel_primitive:NN \pdfimageresolution \pdftex_pdfimageresolution:D
1517  \__kernel_primitive:NN \pdfincludechars     \pdftex_pdfincludechars:D
1518  \__kernel_primitive:NN \pdfinclusioncopyfonts \pdftex_pdfinclusioncopyfonts:D
1519  \__kernel_primitive:NN \pdfinclusionerrorlevel \pdftex_pdfinclusionerrorlevel:D
1520  \__kernel_primitive:NN \pdflastinclusionerrorlevel \pdftex_pdflastinclusionerrorlevel:D
1521  \__kernel_primitive:NN \pdfinfo              \pdftex_pdfinfo:D
1522  \__kernel_primitive:NN \pdflastannot        \pdftex_pdflastannot:D
1523  \__kernel_primitive:NN \pdflastlink         \pdftex_pdflastlink:D
1524  \__kernel_primitive:NN \pdflastobj          \pdftex_pdflastobj:D
1525  \__kernel_primitive:NN \pdflastxform        \pdftex_pdflastxform:D
1526  \__kernel_primitive:NN \pdflastximage       \pdftex_pdflastximage:D
1527  \__kernel_primitive:NN \pdflastximagecolordepth \pdftex_pdflastximagecolordepth:D
1528  \__kernel_primitive:NN \pdflastximagepages   \pdftex_pdflastximagepages:D
1529  \__kernel_primitive:NN \pdflinkmargin       \pdftex_pdflinkmargin:D
1530  \__kernel_primitive:NN \pdfliteral           \pdftex_pdfliteral:D
1531  \__kernel_primitive:NN \pdfminorversion      \pdftex_pdfminorversion:D
1532  \__kernel_primitive:NN \pdfnames             \pdftex_pdfnames:D
1533  \__kernel_primitive:NN \pdfobj               \pdftex_pdfobj:D
1534  \__kernel_primitive:NN \pdfobjcompresslevel \pdftex_pdfobjcompresslevel:D
1535  \__kernel_primitive:NN \pdfoutline           \pdftex_pdfoutline:D
1536  \__kernel_primitive:NN \pdfoutput            \pdftex_pdfoutput:D
1537  \__kernel_primitive:NN \pdfpageattr          \pdftex_pdfpageattr:D
1538  \__kernel_primitive:NN \pdfpagebox           \pdftex_pdfpagebox:D
1539  \__kernel_primitive:NN \pdfpageref            \pdftex_pdfpageref:D
1540  \__kernel_primitive:NN \pdfpageresources     \pdftex_pdfpageresources:D
1541  \__kernel_primitive:NN \pdfpagesattr         \pdftex_pdfpagesattr:D
1542  \__kernel_primitive:NN \pdfrefobj            \pdftex_pdfrefobj:D
1543  \__kernel_primitive:NN \pdfrefxform          \pdftex_pdfrefxform:D
1544  \__kernel_primitive:NN \pdfrefximage         \pdftex_pdfrefximage:D
1545  \__kernel_primitive:NN \pdfrestore            \pdftex_pdfrestore:D
1546  \__kernel_primitive:NN \pdfretval             \pdftex_pdfretval:D
1547  \__kernel_primitive:NN \pdfsave               \pdftex_pdfsave:D
1548  \__kernel_primitive:NN \pdfsetmatrix          \pdftex_pdfsetmatrix:D
1549  \__kernel_primitive:NN \pdfstartlink         \pdftex_pdfstartlink:D
1550  \__kernel_primitive:NN \pdfstartthread       \pdftex_pdfstartthread:D
1551  \__kernel_primitive:NN \pdfsuppressptexinfo  \pdftex_pdfsuppressptexinfo:D
1552  \__kernel_primitive:NN \pdfthread            \pdftex_pdfthread:D
1553  \__kernel_primitive:NN \pdfthreadmargin      \pdftex_pdfthreadmargin:D
1554  \__kernel_primitive:NN \pdftrailer           \pdftex_pdftrailer:D
1555  \__kernel_primitive:NN \pdfuniqueresname    \pdftex_pdfuniqueresname:D
1556  \__kernel_primitive:NN \pdfvorigin           \pdftex_pdfvorigin:D
1557  \__kernel_primitive:NN \pdfxform             \pdftex_pdfxform:D
1558  \__kernel_primitive:NN \pdfxformattr         \pdftex_pdfxformattr:D
1559  \__kernel_primitive:NN \pdfxformname         \pdftex_pdfxformname:D
1560  \__kernel_primitive:NN \pdfxformresources    \pdftex_pdfxformresources:D
1561  \__kernel_primitive:NN \pdfximage            \pdftex_pdfximage:D
1562  \__kernel_primitive:NN \pdfximagebbox        \pdftex_pdfximagebbox:D
1563  \__kernel_primitive:NN \ifpdfabsdim         \pdftex_ifpdfabsdim:D

```

```

1568  \__kernel_primitive:NN \ifpdfabsnum
1569  \__kernel_primitive:NN \ifpdfprimitive
1570  \__kernel_primitive:NN \pdfadjustspacing
1571  \__kernel_primitive:NN \pdfcopyfont
1572  \__kernel_primitive:NN \pdfdraftmode
1573  \__kernel_primitive:NN \pdfeachlinedepth
1574  \__kernel_primitive:NN \pdfeachlineheight
1575  \__kernel_primitive:NN \pdffilemoddate
1576  \__kernel_primitive:NN \pdffilesize
1577  \__kernel_primitive:NN \pdffirstlineheight
1578  \__kernel_primitive:NN \pdffontexpand
1579  \__kernel_primitive:NN \pdffontsize
1580  \__kernel_primitive:NN \pdfignoreddimen
1581  \__kernel_primitive:NN \pdfinsertht
1582  \__kernel_primitive:NN \pdflastlinedepth
1583  \__kernel_primitive:NN \pdflastxpos
1584  \__kernel_primitive:NN \pdflastypos
1585  \__kernel_primitive:NN \pdfmapfile
1586  \__kernel_primitive:NN \pdfmapline
1587  \__kernel_primitive:NN \pdfmdfivesum
1588  \__kernel_primitive:NN \pdfnoligatures
1589  \__kernel_primitive:NN \pdfnormaldeviate
1590  \__kernel_primitive:NN \pdfpageheight
1591  \__kernel_primitive:NN \pdfpagewidth
1592  \__kernel_primitive:NN \pdfpkmode
1593  \__kernel_primitive:NN \pdfpkresolution
1594  \__kernel_primitive:NN \pdfprimitive
1595  \__kernel_primitive:NN \pdfprotrudechars
1596  \__kernel_primitive:NN \pdfpxdimen
1597  \__kernel_primitive:NN \pdfrandomseed
1598  \__kernel_primitive:NN \pdfsavepos
1599  \__kernel_primitive:NN \pdfstrcmp
1600  \__kernel_primitive:NN \pdfsetrandomseed
1601  \__kernel_primitive:NN \pdfshellescape
1602  \__kernel_primitive:NN \pdftracingfonts
1603  \__kernel_primitive:NN \pdfuniformdeviate
1604  \__kernel_primitive:NN \pdftexbanner
1605  \__kernel_primitive:NN \pdftexrevision
1606  \__kernel_primitive:NN \pdftexversion
1607  \__kernel_primitive:NN \efcode
1608  \__kernel_primitive:NN \ifincsname
1609  \__kernel_primitive:NN \leftmarginkern
1610  \__kernel_primitive:NN \letterspacefont
1611  \__kernel_primitive:NN \lpcode
1612  \__kernel_primitive:NN \quitvmode
1613  \__kernel_primitive:NN \rightmarginkern
1614  \__kernel_primitive:NN \rpcode
1615  \__kernel_primitive:NN \synctex
1616  \__kernel_primitive:NN \tagcode
1617  \__kernel_primitive:NN \mdfivesum
1618  \__kernel_primitive:NN \ifprimitive
1619  \__kernel_primitive:NN \primitive
1620  \__kernel_primitive:NN \shellescape
1621  \__kernel_primitive:NN \adjustspacing

```

\pdftex\_ifabsnum:D  
\pdftex\_ifprimitive:D  
\pdftex\_adjustspacing:D  
\pdftex\_copyfont:D  
\pdftex\_draftmode:D  
\pdftex\_eachlinedepth:D  
\pdftex\_eachlineheight:D  
\pdftex\_filemoddate:D  
\pdftex\_filesize:D  
\pdftex\_firstlineheight:D  
\pdftex\_fontexpand:D  
\pdftex fontsize:D  
\pdftex\_ignoreddimen:D  
\pdftex\_insertht:D  
\pdftex\_lastlinedepth:D  
\pdftex\_lastxpos:D  
\pdftex\_lastypos:D  
\pdftex\_mapfile:D  
\pdftex\_mapline:D  
\pdftex\_mdfivesum:D  
\pdftex\_noligatures:D  
\pdftex\_normaldeviate:D  
\pdftex\_pageheight:D  
\pdftex\_pagewidth:D  
\pdftex\_pkmode:D  
\pdftex\_pkresolution:D  
\pdftex\_primitive:D  
\pdftex\_protrudechars:D  
\pdftex\_pxdimen:D  
\pdftex\_randomseed:D  
\pdftex\_savepos:D  
\pdftex\_strcmp:D  
\pdftex\_setrandomseed:D  
\pdftex\_shellescape:D  
\pdftex\_tracingfonts:D  
\pdftex\_uniformdeviate:D  
\pdftex\_pdftexbanner:D  
\pdftex\_pdftexrevision:D  
\pdftex\_pdftexversion:D  
\pdftex\_efcode:D  
\pdftex\_ifincsname:D  
\pdftex\_leftmarginkern:D  
\pdftex\_letterspacefont:D  
\pdftex\_lpcode:D  
\pdftex\_quitvmode:D  
\pdftex\_rightmarginkern:D  
\pdftex\_rpcode:D  
\pdftex\_synctex:D  
\pdftex\_tagcode:D  
\pdftex\_mdfivesum:D  
\pdftex\_ifprimitive:D  
\pdftex\_primitive:D  
\pdftex\_shellescape:D  
\pdftex\_adjustspacing:D

```

1622  \__kernel_primitive:NN \copyfont           \pdftex_copyfont:D
1623  \__kernel_primitive:NN \draftmode         \pdftex_draftmode:D
1624  \__kernel_primitive:NN \expandglyphsinfont \pdftex_fontexpand:D
1625  \__kernel_primitive:NN \ifabsdim          \pdftex_ifabsdim:D
1626  \__kernel_primitive:NN \ifabsnum          \pdftex_ifabsnum:D
1627  \__kernel_primitive:NN \ignoreligaturesinfont \pdftex_ignoreligaturesinfont:D
1628  \__kernel_primitive:NN \insertht          \pdftex_insertht:D
1629  \__kernel_primitive:NN \lastsavedboxresourceindex \pdftex_lastsavedboxresourceindex
1630  \__kernel_primitive:NN \pdflastxform:D
1631  \__kernel_primitive:NN \lastsavedimageresourceindex \pdftex_pdflastximage:D
1632  \__kernel_primitive:NN \lastsavedimageresourcepages \pdftex_pdflastximagepages:D
1633  \__kernel_primitive:NN \pdftex_pdflastximagepages:D
1634  \__kernel_primitive:NN \lastsavedimageresourcepages \pdftex_pdflastximagepages:D
1635  \__kernel_primitive:NN \lastxpos           \pdftex_lastxpos:D
1636  \__kernel_primitive:NN \lastypos           \pdftex_lastypos:D
1637  \__kernel_primitive:NN \normaldeviate      \pdftex_normaldeviate:D
1638  \__kernel_primitive:NN \outputmode         \pdftex_pdfoutput:D
1639  \__kernel_primitive:NN \pageheight        \pdftex_pageheight:D
1640  \__kernel_primitive:NN \pagewidth          \pdftex_pagewidth:D
1641  \__kernel_primitive:NN \protrudechars     \pdftex_protrudechars:D
1642  \__kernel_primitive:NN \pxdimen            \pdftex_pxdimen:D
1643  \__kernel_primitive:NN \randomseed         \pdftex_randomseed:D
1644  \__kernel_primitive:NN \useboxresource     \pdftex_pdffrefxform:D
1645  \__kernel_primitive:NN \useimageresource   \pdftex_pdffrefximage:D
1646  \__kernel_primitive:NN \savepos            \pdftex_savepos:D
1647  \__kernel_primitive:NN \saveboxresource    \pdftex_pdffxform:D
1648  \__kernel_primitive:NN \saveimageresource  \pdftex_pdffximage:D
1649  \__kernel_primitive:NN \setrandomseed      \pdftex_setrandomseed:D
1650  \__kernel_primitive:NN \tracingfonts       \pdftex_tracingfonts:D
1651  \__kernel_primitive:NN \uniformdeviate     \pdftex_uniformdeviate:D
1652  \__kernel_primitive:NN \suppressfontnotfounderror \xetex_suppressfontnotfounderror:D
1653  \__kernel_primitive:NN \XeTeXcharclass      \xetex_charclass:D
1654  \__kernel_primitive:NN \XeTeXcharglyph     \xetex_charglyph:D
1655  \__kernel_primitive:NN \XeTeXcountfeatures \xetex_countfeatures:D
1656  \__kernel_primitive:NN \XeTeXcountglyphs   \xetex_countglyphs:D
1657  \__kernel_primitive:NN \XeTeXcountselectors \xetex_countselectors:D
1658  \__kernel_primitive:NN \XeTeXcountvariations \xetex_countvariations:D
1659  \__kernel_primitive:NN \XeTeXdefaultencoding \xetex_defaultencoding:D
1660  \__kernel_primitive:NN \XeTeXdashbreakstate \xetex_dashbreakstate:D
1661  \__kernel_primitive:NN \XeTeXfeaturecode    \xetex_featurecode:D
1662  \__kernel_primitive:NN \XeTeXfeaturename    \xetex_featurename:D
1663  \__kernel_primitive:NN \XeTeXfindfeaturebyname \xetex_findfeaturebyname:D
1664  \__kernel_primitive:NN \XeTeXfindselectorbyname \xetex_findselectorbyname:D
1665  \__kernel_primitive:NN \XeTeXfindvariationbyname \xetex_findvariationbyname:D
1666  \__kernel_primitive:NN \XeTeXfirstfontchar   \xetex_firstfontchar:D
1667  \__kernel_primitive:NN \XeTeXfonttype        \xetex_fonttype:D
1668  \__kernel_primitive:NN \XeTeXgenerateactualtext \xetex_generateactualtext:D
1669  \__kernel_primitive:NN \XeTeXglyph           \xetex_glyph:D

```

```

1676  \__kernel_primitive:NN \XeTeXglyphbounds           \xetex_glyphbounds:D
1677  \__kernel_primitive:NN \XeTeXglyphindex          \xetex_glyphindex:D
1678  \__kernel_primitive:NN \XeTeXglyphname           \xetex_glyphname:D
1679  \__kernel_primitive:NN \XeTeXinputencoding        \xetex_inputencoding:D
1680  \__kernel_primitive:NN \XeTeXinputnormalization   \xetex_inputnormalization:D
1681  \__kernel_primitive:NN \XeTeXinterchartokenstate  \xetex_interchartokenstate:D
1682  \__kernel_primitive:NN \XeTeXinterchartokots      \xetex_interchartokots:D
1683  \__kernel_primitive:NN \XeTeXisdefaultselector     \xetex_isdefaultselector:D
1684  \__kernel_primitive:NN \XeTeXlastfontchar         \xetex_lastfontchar:D
1685  \__kernel_primitive:NN \XeTeXlinebreakskip        \xetex_linebreakskip:D
1686  \__kernel_primitive:NN \XeTeXlinebreaklocale       \xetex_linebreaklocale:D
1687  \__kernel_primitive:NN \XeTeXlinebreakpenalty      \xetex_linebreakpenalty:D
1688  \__kernel_primitive:NN \XeTeXOTcountfeatures       \xetex_OTcountfeatures:D
1689  \__kernel_primitive:NN \XeTeXOTcountlanguages      \xetex_OTcountlanguages:D
1690  \__kernel_primitive:NN \XeTeXOTcountsscripts       \xetex_OTcountsscripts:D
1691  \__kernel_primitive:NN \XeTeXOTfeaturetag         \xetex_OTfeaturetag:D
1692  \__kernel_primitive:NN \XeTeXOTlanguagetag        \xetex_OTlanguagetag:D
1693  \__kernel_primitive:NN \XeTeXOTscripttag         \xetex_OTscripttag:D
1694  \__kernel_primitive:NN \XeTeXpdffile             \xetex_pdffile:D
1695  \__kernel_primitive:NN \XeTeXpdfpagecount        \xetex_pdfpagecount:D
1696  \__kernel_primitive:NN \XeTeXpicfile            \xetex_picfile:D
1697  \__kernel_primitive:NN \XeTeXselectorname        \xetex_selectorname:D
1698  \__kernel_primitive:NN \XeTeXtracingfonts        \xetex_tracingfonts:D
1699  \__kernel_primitive:NN \XeTeXupwardsmode         \xetex_upwardsmode:D
1700  \__kernel_primitive:NN \XeTeXuseglyphmetrics     \xetex_useglyphmetrics:D
1701  \__kernel_primitive:NN \XeTeXvariation           \xetex_variation:D
1702  \__kernel_primitive:NN \XeTeXvariationdefault    \xetex_variationdefault:D
1703  \__kernel_primitive:NN \XeTeXvariationmax        \xetex_variationmax:D
1704  \__kernel_primitive:NN \XeTeXvariationmin        \xetex_variationmin:D
1705  \__kernel_primitive:NN \XeTeXvariationname       \xetex_variationname:D
1706  \__kernel_primitive:NN \XeTeXrevision            \xetex_XeTeXrevision:D
1707  \__kernel_primitive:NN \XeTeXversion              \xetex_XeTeXversion:D
1708  \__kernel_primitive:NN \alignmark               \luatex_alignmark:D
1709  \__kernel_primitive:NN \aligntab                \luatex_aligntab:D
1710  \__kernel_primitive:NN \attribute               \luatex_attribute:D
1711  \__kernel_primitive:NN \attributedef           \luatex_attributedef:D
1712  \__kernel_primitive:NN \automaticdiscretionary \luatex_automaticdiscretionary:D
1713  \__kernel_primitive:NN \automatichyphenmode    \luatex_automatichyphenmode:D
1714  \__kernel_primitive:NN \beginncsname            \luatex_beginncsname:D
1715  \__kernel_primitive:NN \breakafterdirmode      \luatex_breakafterdirmode:D
1716  \__kernel_primitive:NN \catcodetable          \luatex_catcodetable:D
1717  \__kernel_primitive:NN \clearmarks             \luatex_clearmarks:D
1718  \__kernel_primitive:NN \crampeddisplaystyle     \luatex_crampeddisplaystyle:D
1719  \__kernel_primitive:NN \crampedscriptstyle      \luatex_crampedscriptstyle:D
1720  \__kernel_primitive:NN \crampeddisplaystyle     \luatex_crampeddisplaystyle:D
1721  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D
1722  \__kernel_primitive:NN \crampedscriptstyle      \luatex_crampedscriptstyle:D
1723  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D
1724  \__kernel_primitive:NN \crampedscriptstyle      \luatex_crampedscriptstyle:D
1725  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D
1726  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D
1727  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D
1728  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D
1729  \__kernel_primitive:NN \crampedscriptscriptstyle \luatex_crampedscriptscriptstyle:D

```

```

1730      \luatex_crampedscriptstyle:D
1731  \__kernel_primitive:NN \crampedscriptstyle
1732  \__kernel_primitive:NN \crampedtextstyle
1733  \__kernel_primitive:NN \directlua
1734  \__kernel_primitive:NN \dviextension
1735  \__kernel_primitive:NN \dvifeedback
1736  \__kernel_primitive:NN \dvivariable
1737  \__kernel_primitive:NN \etoksapp
1738  \__kernel_primitive:NN \etokspre
1739  \__kernel_primitive:NN \explicithyphenpenalty
1740          \luatex_explicithyphenpenalty:D
1741  \__kernel_primitive:NN \expanded
1742  \__kernel_primitive:NN \explicitdiscretionary
1743          \luatex_explicitdiscretionary:D
1744  \__kernel_primitive:NN \firstvalidlanguage
1745  \__kernel_primitive:NN \fontid
1746  \__kernel_primitive:NN \formatname
1747  \__kernel_primitive:NN \hjcode
1748  \__kernel_primitive:NN \hpack
1749  \__kernel_primitive:NN \hyphenationbounds
1750  \__kernel_primitive:NN \hyphenationmin
1751  \__kernel_primitive:NN \hyphenpenaltymode
1752  \__kernel_primitive:NN \gleaders
1753  \__kernel_primitive:NN \initcatcodetable
1754  \__kernel_primitive:NN \lastnamedcs
1755  \__kernel_primitive:NN \latelua
1756  \__kernel_primitive:NN \letcharcode
1757  \__kernel_primitive:NN \luaescapestring
1758  \__kernel_primitive:NN \luafunction
1759  \__kernel_primitive:NN \luatexbanner
1760  \__kernel_primitive:NN \luatexrevision
1761  \__kernel_primitive:NN \luatexversion
1762  \__kernel_primitive:NN \mathdelimitersmode
1763  \__kernel_primitive:NN \mathdisplayskipmode
1764          \luatex_mathdisplayskipmode:D
1765  \__kernel_primitive:NN \matheqnogapstep
1766  \__kernel_primitive:NN \mathnolimitsmode
1767  \__kernel_primitive:NN \mathoption
1768  \__kernel_primitive:NN \mathpenaltiesmode
1769  \__kernel_primitive:NN \mathrulesfam
1770  \__kernel_primitive:NN \mathscriptsmode
1771  \__kernel_primitive:NN \mathscriptboxmode
1772  \__kernel_primitive:NN \mathstyle
1773  \__kernel_primitive:NN \mathsurroundmode
1774  \__kernel_primitive:NN \mathsurroundskip
1775  \__kernel_primitive:NN \nohrule
1776  \__kernel_primitive:NN \nokerns
1777  \__kernel_primitive:NN \noligs
1778  \__kernel_primitive:NN \nospaces
1779  \__kernel_primitive:NN \novrule
1780  \__kernel_primitive:NN \outputbox
1781  \__kernel_primitive:NN \pagebottomoffset
1782  \__kernel_primitive:NN \pageleftoffset
1783  \__kernel_primitive:NN \pagerightoffset
\luatex_crampedscriptstyle:D
\luatex_crampedtextstyle:D
\luatex_directlua:D
\luatex_dviextension:D
\luatex_dvifeedback:D
\luatex_dvivariable:D
\luatex_etoksapp:D
\luatex_etokspre:D
\luatex_expanded:D
\luatex_firstvalidlanguage:D
\luatex_fontid:D
\luatex_formatname:D
\luatex_hjcode:D
\luatex_hpack:D
\luatex_hyphenationbounds:D
\luatex_hyphenationmin:D
\luatex_hyphenpenaltymode:D
\luatex_gleaders:D
\luatex_initcatcodetable:D
\luatex_lastnamedcs:D
\luatex_latelua:D
\luatex_letcharcode:D
\luatex_luaescapestring:D
\luatex_luafunction:D
\luatex_luatexbanner:D
\luatex_luatexrevision:D
\luatex_luatexversion:D
\luatex_mathdelimitersmode:D
\luatex_matheqnogapstep:D
\luatex_mathnolimitsmode:D
\luatex_mathoption:D
\luatex_mathpenaltiesmode:D
\luatex_mathrulesfam:D
\luatex_mathscriptsmode:D
\luatex_mathscriptboxmode:D
\luatex_mathstyle:D
\luatex_mathsurroundmode:D
\luatex_mathsurroundskip:D
\luatex_noctrl:D
\luatex_nokerns:D
\luatex_noligs:D
\luatex_nospaces:D
\luatex_novrule:D
\luatex_outputbox:D
\luatex_pagebottomoffset:D
\luatex_pageleftoffset:D
\luatex_pagerightoffset:D

```

```

1784  \__kernel_primitive:NN \pagetopoffset           \luatex_pagetopoffset:D
1785  \__kernel_primitive:NN \pdfextension           \luatex_pdfextension:D
1786  \__kernel_primitive:NN \pdffeedback           \luatex_pdffeedback:D
1787  \__kernel_primitive:NN \pdfvariable            \luatex_pdfvariable:D
1788  \__kernel_primitive:NN \postexhyphenchar       \luatex_postexhyphenchar:D
1789  \__kernel_primitive:NN \posthyphenchar          \luatex_posthyphenchar:D
1790  \__kernel_primitive:NN \prebinoppenalty         \luatex_prebinoppenalty:D
1791  \__kernel_primitive:NN \predisplaygapfactor    \luatex_predisplaygapfactor:D
1792  \__kernel_primitive:NN \luatex_predisplaygapfactor:D
1793  \__kernel_primitive:NN \preexhyphenchar          \luatex_preexhyphenchar:D
1794  \__kernel_primitive:NN \prehypenchar             \luatex_prehypenchar:D
1795  \__kernel_primitive:NN \prerelpenalty            \luatex_prerelpenalty:D
1796  \__kernel_primitive:NN \savecatcodetable        \luatex_savecatcodetable:D
1797  \__kernel_primitive:NN \scantexttokens           \luatex_scantexttokens:D
1798  \__kernel_primitive:NN \setfontid                \luatex_setfontid:D
1799  \__kernel_primitive:NN \shapemode                \luatex_shapemode:D
1800  \__kernel_primitive:NN \suppressifcsnameerror   \luatex_suppressifcsnameerror:D
1801  \__kernel_primitive:NN \luatex_suppressifcsnameerror:D
1802  \__kernel_primitive:NN \suppresslongerror        \luatex_suppresslongerror:D
1803  \__kernel_primitive:NN \suppressmathparerror     \luatex_suppressmathparerror:D
1804  \__kernel_primitive:NN \luatex_suppressmathparerror:D
1805  \__kernel_primitive:NN \suppressoutererror       \luatex_suppressoutererror:D
1806  \__kernel_primitive:NN \suppressprimitiveerror   \luatex_suppressprimitiveerror:D
1807  \__kernel_primitive:NN \luatex_suppressprimitiveerror:D
1808  \__kernel_primitive:NN \toksapp                 \luatex_toksapp:D
1809  \__kernel_primitive:NN \tokspre                 \luatex_tokspre:D
1810  \__kernel_primitive:NN \tpack                   \luatex_tpak:D
1811  \__kernel_primitive:NN \vpack                   \luatex_vpack:D
1812  \__kernel_primitive:NN \bodydir                 \luatex_bodydir:D
1813  \__kernel_primitive:NN \boxdir                  \luatex_boxdir:D
1814  \__kernel_primitive:NN \leftghost              \luatex_leftghost:D
1815  \__kernel_primitive:NN \linedir                 \luatex_linedir:D
1816  \__kernel_primitive:NN \localbrokenpenalty      \luatex_localbrokenpenalty:D
1817  \__kernel_primitive:NN \localinterlinepenalty    \luatex_localinterlinepenalty:D
1818  \__kernel_primitive:NN \luatex_localinterlinepenalty:D
1819  \__kernel_primitive:NN \localleftbox            \luatex_localleftbox:D
1820  \__kernel_primitive:NN \localrightbox           \luatex_localrightbox:D
1821  \__kernel_primitive:NN \mathdir                 \luatex_mathdir:D
1822  \__kernel_primitive:NN \pagedir                 \luatex_pagedir:D
1823  \__kernel_primitive:NN \pardir                 \luatex_pardir:D
1824  \__kernel_primitive:NN \rightghost              \luatex_rightghost:D
1825  \__kernel_primitive:NN \textdir                 \luatex_textdir:D
1826  \__kernel_primitive:NN \Uchar                  \utex_char:D
1827  \__kernel_primitive:NN \Ucharcat               \utex_charcat:D
1828  \__kernel_primitive:NN \Udelcode               \utex_delcode:D
1829  \__kernel_primitive:NN \Udelcodenum            \utex_delcodenum:D
1830  \__kernel_primitive:NN \Udelimiter              \utex_delimiter:D
1831  \__kernel_primitive:NN \Udelimiterover         \utex_delimiterover:D
1832  \__kernel_primitive:NN \Udelimiterunder        \utex_delimiterunder:D
1833  \__kernel_primitive:NN \Uhextensible           \utex_hextensible:D
1834  \__kernel_primitive:NN \Umathaccent            \utex_mathaccent:D
1835  \__kernel_primitive:NN \Umathaxis               \utex_mathaxis:D
1836  \__kernel_primitive:NN \Umathbinbinspacing     \utex_binbinspacing:D
1837  \__kernel_primitive:NN \Umathbinbinspacing     \utex_binbinspacing:D

```

```

1838  \__kernel_primitive:NN \Umathbininnerspacing \utex_bininnerspacing:D
1839  \__kernel_primitive:NN \Umathbinopenspacing \utex_binopenspacing:D
1840  \__kernel_primitive:NN \Umathbinopspacing \utex_binopspacing:D
1841  \__kernel_primitive:NN \Umathbinordspacing \utex_binordspacing:D
1842  \__kernel_primitive:NN \Umathbinpunctspacing \utex_binpunctspacing:D
1843  \__kernel_primitive:NN \Umathbinrelspacing \utex_binrelspacing:D
1844  \__kernel_primitive:NN \Umathchar \utex_mathchar:D
1845  \__kernel_primitive:NN \Umathcharclass \utex_mathcharclass:D
1846  \__kernel_primitive:NN \Umathchardef \utex_mathchardef:D
1847  \__kernel_primitive:NN \Umathcharfam \utex_mathcharfam:D
1848  \__kernel_primitive:NN \Umathcharnum \utex_mathcharnum:D
1849  \__kernel_primitive:NN \Umathcharnumdef \utex_mathcharnumdef:D
1850  \__kernel_primitive:NN \Umathcharslot \utex_mathcharslot:D
1851  \__kernel_primitive:NN \Umathclosebinspacing \utex_closebinspacing:D
1852  \__kernel_primitive:NN \Umathcloseclosespacing
   \utex_closeclosespacing:D
1853  \__kernel_primitive:NN \Umathcloseinnerspacing
   \utex_closeinnerspacing:D
1854  \__kernel_primitive:NN \Umathcloseopenspacing \utex_closeopenspacing:D
1855  \__kernel_primitive:NN \Umathcloseopspace \utex_closeopspace:D
1856  \__kernel_primitive:NN \Umathcloseordspacing \utex_closeordspacing:D
1857  \__kernel_primitive:NN \Umathclosepunctspacing
   \utex_closepunctspacing:D
1858  \__kernel_primitive:NN \Umathcloserelspace \utex_closerelspace:D
1859  \__kernel_primitive:NN \Umathcode \utex_mathcode:D
1860  \__kernel_primitive:NN \Umathcodenum \utex_mathcodenum:D
1861  \__kernel_primitive:NN \Umathconnectoroverlapmin
   \utex_connectoroverlapmin:D
1862  \__kernel_primitive:NN \Umathfractiondelsize \utex_fractiondelsize:D
1863  \__kernel_primitive:NN \Umathfractiondenomdown
   \utex_fractiondenomdown:D
1864  \__kernel_primitive:NN \Umathfractiondenomvgap
   \utex_fractiondenomvgap:D
1865  \__kernel_primitive:NN \Umathfractionnumup \utex_fractionnumup:D
1866  \__kernel_primitive:NN \Umathfractionnumvgap \utex_fractionnumvgap:D
1867  \__kernel_primitive:NN \Umathfractionrule \utex_fractionrule:D
1868  \__kernel_primitive:NN \Umathinnerbinspacing \utex_innerbinspacing:D
1869  \__kernel_primitive:NN \Umathinnerclosespacing
   \utex_innerclosespacing:D
1870  \__kernel_primitive:NN \Umathinnerinnerspacing
   \utex_innerinnerspacing:D
1871  \__kernel_primitive:NN \Umathinneropenspacing \utex_inneropenspacing:D
1872  \__kernel_primitive:NN \Umathinneropspace \utex_inneropspace:D
1873  \__kernel_primitive:NN \Umathinnerordspacing \utex_innerordspacing:D
1874  \__kernel_primitive:NN \Umathinnerpunctspacing
   \utex_innerpunctspacing:D
1875  \__kernel_primitive:NN \Umathinnerrelspacing \utex_innerrelspacing:D
1876  \__kernel_primitive:NN \Umathlimitabovebgap \utex_limitabovebgap:D
1877  \__kernel_primitive:NN \Umathlimitabovekern \utex_limitabovekern:D
1878  \__kernel_primitive:NN \Umathlimitabovevgap \utex_limitabovevgap:D
1879  \__kernel_primitive:NN \Umathlimitbelowbgap \utex_limitbelowbgap:D
1880  \__kernel_primitive:NN \Umathlimitbelowkern \utex_limitbelowkern:D
1881  \__kernel_primitive:NN \Umathlimitbelowvgap \utex_limitbelowvgap:D
1882  \__kernel_primitive:NN \Umathnolimitsfactor \utex_nolimitsfactor:D
1883  \__kernel_primitive:NN \Umathnolimitssubfactor \utex_nolimitssubfactor:D

```

```

1892 \__kernel_primitive:NN \Umathnolimitsupfactor \utex_nolimitsupfactor:D
1893 \__kernel_primitive:NN \Umathopbinspacing \utex_opbinspacing:D
1894 \__kernel_primitive:NN \Umathopclosespacing \utex_opclosespacing:D
1895 \__kernel_primitive:NN \Umathopenbinspacing \utex_openbinspacing:D
1896 \__kernel_primitive:NN \Umathopenclosespacing \utex_openclosespacing:D
1897 \__kernel_primitive:NN \Umathopeninnerspacing \utex_openinnerspacing:D
1898 \__kernel_primitive:NN \Umathopenopenspacing \utex_openopenspacing:D
1899 \__kernel_primitive:NN \Umathopenopsspacing \utex_openopsspacing:D
1900 \__kernel_primitive:NN \Umathopenordspacing \utex_openordspacing:D
1901 \__kernel_primitive:NN \Umathopenpunctspacing \utex_openpunctspacing:D
1902 \__kernel_primitive:NN \Umathopenrelspacing \utex_openrelspacing:D
1903 \__kernel_primitive:NN \Umathoperatorsize \utex_operatorsize:D
1904 \__kernel_primitive:NN \Umathopinnerspacing \utex_opinnerspacing:D
1905 \__kernel_primitive:NN \Umathopopenspacing \utex_opopenspacing:D
1906 \__kernel_primitive:NN \Umathopopspace \utex_opopspace:D
1907 \__kernel_primitive:NN \Umathopordspacing \utex_opordspacing:D
1908 \__kernel_primitive:NN \Umathoppunctspacing \utex_oppunctspacing:D
1909 \__kernel_primitive:NN \Umathoprelspacing \utex_oprelspacing:D
1910 \__kernel_primitive:NN \Umathordbinspacing \utex_ordbinspacing:D
1911 \__kernel_primitive:NN \Umathordclosespacing \utex_ordclosespacing:D
1912 \__kernel_primitive:NN \Umathordinnerspacing \utex_ordinnerspacing:D
1913 \__kernel_primitive:NN \Umathordopenspacing \utex_ordopenspacing:D
1914 \__kernel_primitive:NN \Umathorddspacing \utex_orddspacing:D
1915 \__kernel_primitive:NN \Umathordordspacing \utex_ordordspacing:D
1916 \__kernel_primitive:NN \Umathordpunctspacing \utex_ordpunctspacing:D
1917 \__kernel_primitive:NN \Umathordrelspacing \utex_ordrelspacing:D
1918 \__kernel_primitive:NN \Umathoverbarkern \utex_overbarkern:D
1919 \__kernel_primitive:NN \Umathoverbarrule \utex_overbarrule:D
1920 \__kernel_primitive:NN \Umathoverbarvgap \utex_overbarvgap:D
1921 \__kernel_primitive:NN \Umathoverdelimiterbgap
    \utex_overdelimiterbgap:D
1923 \__kernel_primitive:NN \Umathoverdelimitervgap
    \utex_overdelimitervgap:D
1925 \__kernel_primitive:NN \Umathpunctbinspacing \utex_punctbinspacing:D
1926 \__kernel_primitive:NN \Umathpunctclosespacing
    \utex_punctclosespacing:D
1928 \__kernel_primitive:NN \Umathpunctinnerspacing
    \utex_punctinnerspacing:D
1930 \__kernel_primitive:NN \Umathpunctopenspacing \utex_punctopenspacing:D
1931 \__kernel_primitive:NN \Umathpuncttopspacing \utex_puncttopspacing:D
1932 \__kernel_primitive:NN \Umathpunctordspacing \utex_punctordspacing:D
1933 \__kernel_primitive:NN \Umathpunctpunctspacing \utex_punctpunctspacing:D
1934 \__kernel_primitive:NN \Umathpunctrelspacing \utex_punctrelspacing:D
1935 \__kernel_primitive:NN \Umathquad \utex_quad:D
1936 \__kernel_primitive:NN \Umathradicaldegreeafter
    \utex_radicaldegreeafter:D
1938 \__kernel_primitive:NN \Umathradicaldegreebefore
    \utex_radicaldegreebefore:D
1940 \__kernel_primitive:NN \Umathradicaldegreeraise
    \utex_radicaldegreeraise:D
1942 \__kernel_primitive:NN \Umathradicalkern \utex_radicalkern:D
1943 \__kernel_primitive:NN \Umathradicalrule \utex_radicalrule:D
1944 \__kernel_primitive:NN \Umathradicalvgap \utex_radicalvgap:D
1945 \__kernel_primitive:NN \Umathrelbinspacing \utex_relbinspacing:D

```

```

1946  \__kernel_primitive:NN \Umathrelclosespacing \utex_relclosespacing:D
1947  \__kernel_primitive:NN \Umathrelinnerspacing \utex_relinnerspacing:D
1948  \__kernel_primitive:NN \Umathrelopenspacing \utex_relopenspacing:D
1949  \__kernel_primitive:NN \Umathrelopspacing \utex_rellopspacing:D
1950  \__kernel_primitive:NN \Umathrelordspacing \utex_relordspacing:D
1951  \__kernel_primitive:NN \Umathrelpunctspacing \utex_relpunctspacing:D
1952  \__kernel_primitive:NN \Umathrelrelopspacing \utex_relrelopspacing:D
1953  \__kernel_primitive:NN \Umathskewedfractionhgap
        \utex_skewedfractionhgap:D
1954  \__kernel_primitive:NN \Umathskewedfractionvgap
        \utex_skewedfractionvgap:D
1955  \__kernel_primitive:NN \Umathspaceafterscript \utex_spaceafterscript:D
1956  \__kernel_primitive:NN \Umathstackdenomdown \utex_stackdenomdown:D
1957  \__kernel_primitive:NN \Umathstacknumup \utex_stacknumup:D
1958  \__kernel_primitive:NN \Umathstackvgap \utex_stackvgap:D
1959  \__kernel_primitive:NN \Umathsubshiftdown \utex_subshiftdown:D
1960  \__kernel_primitive:NN \Umathsubshiftdrop \utex_subshiftdrop:D
1961  \__kernel_primitive:NN \Umathsubsupshiftdown \utex_subsupshiftdown:D
1962  \__kernel_primitive:NN \Umathsubsupvgap \utex_subsupvgap:D
1963  \__kernel_primitive:NN \Umathsubtopmax \utex_subtopmax:D
1964  \__kernel_primitive:NN \Umathsupbottommin \utex_supbottommin:D
1965  \__kernel_primitive:NN \Umathsupshiftdrop \utex_supshiftdrop:D
1966  \__kernel_primitive:NN \Umathsupshiftup \utex_supshiftup:D
1967  \__kernel_primitive:NN \Umathsupsubbottommax \utex_supsubbottommax:D
1968  \__kernel_primitive:NN \Umathunderbarkern \utex_underbarkern:D
1969  \__kernel_primitive:NN \Umathunderbarrule \utex_underbarrule:D
1970  \__kernel_primitive:NN \Umathunderbarvgap \utex_underbarvgap:D
1971  \__kernel_primitive:NN \Umathunderdelimiterbgap
        \utex_underdelimiterbgap:D
1972  \__kernel_primitive:NN \Umathunderdelimitervgap
        \utex_underdelimitervgap:D
1973  \__kernel_primitive:NN \Unosubscript \utex_nosubscript:D
1974  \__kernel_primitive:NN \Unosuperscript \utex_nosuperscript:D
1975  \__kernel_primitive:NN \Overdelimiter \utex_overdelimiter:D
1976  \__kernel_primitive:NN \Uradical \utex_radical:D
1977  \__kernel_primitive:NN \Uroot \utex_root:D
1978  \__kernel_primitive:NN \Uskewed \utex_skewed:D
1979  \__kernel_primitive:NN \Uskewedwithdelims \utex_skewedwithdelims:D
1980  \__kernel_primitive:NN \Ustack \utex_stack:D
1981  \__kernel_primitive:NN \Ustartdisplaymath \utex_startdisplaymath:D
1982  \__kernel_primitive:NN \Ustartmath \utex_startmath:D
1983  \__kernel_primitive:NN \Ustopdisplaymath \utex_stopdisplaymath:D
1984  \__kernel_primitive:NN \Ustopmath \utex_stopmath:D
1985  \__kernel_primitive:NN \Usubscript \utex_subscript:D
1986  \__kernel_primitive:NN \Usuperscript \utex_superscript:D
1987  \__kernel_primitive:NN \Underdelimiter \utex_underdelimiter:D
1988  \__kernel_primitive:NN \Uvextensible \utex_vextensible:D
1989  \__kernel_primitive:NN \autospacing \ptex_autospacing:D
1990  \__kernel_primitive:NN \autoxspacing \ptex_autoxspacing:D
1991  \__kernel_primitive:NN \dtou \ptex_dtou:D
1992  \__kernel_primitive:NN \epTeXinputencoding \ptex_inputencoding:D
1993  \__kernel_primitive:NN \epTeXversion \ptex_epTeXversion:D
1994  \__kernel_primitive:NN \euc \ptex_euc:D
1995  \__kernel_primitive:NN \ifdbox \ptex_ifdbox:D

```

```

2000  \__kernel_primitive:NN \ifddir          \ptex_ifddir:D
2001  \__kernel_primitive:NN \ifmdir          \ptex_ifmdir:D
2002  \__kernel_primitive:NN \iftbox           \ptex_iftbox:D
2003  \__kernel_primitive:NN \iftdir          \ptex_iftdir:D
2004  \__kernel_primitive:NN \ifybox           \ptex_ifybox:D
2005  \__kernel_primitive:NN \ifydir          \ptex_ifydir:D
2006  \__kernel_primitive:NN \inhibitglue     \ptex_inhibitglue:D
2007  \__kernel_primitive:NN \inhibitxspcode   \ptex_inhibitxspcode:D
2008  \__kernel_primitive:NN \jcharwidowpenalty \ptex_jcharwidowpenalty:D
2009  \__kernel_primitive:NN \jfam              \ptex_jfam:D
2010  \__kernel_primitive:NN \jfont              \ptex_jfont:D
2011  \__kernel_primitive:NN \jis                \ptex_jis:D
2012  \__kernel_primitive:NN \kanjiskip         \ptex_kanjiskip:D
2013  \__kernel_primitive:NN \kansaji           \ptex_kansaji:D
2014  \__kernel_primitive:NN \kansujichar       \ptex_kansujichar:D
2015  \__kernel_primitive:NN \kcatcode          \ptex_kcatcode:D
2016  \__kernel_primitive:NN \kuten              \ptex_kuten:D
2017  \__kernel_primitive:NN \noautospacing     \ptex_noautospacing:D
2018  \__kernel_primitive:NN \noautoxspacing    \ptex_noautoxspacing:D
2019  \__kernel_primitive:NN \postbreakpenalty   \ptex_postbreakpenalty:D
2020  \__kernel_primitive:NN \prebreakpenalty    \ptex_prebreakpenalty:D
2021  \__kernel_primitive:NN \ptexminorversion   \ptex_ptexminorversion:D
2022  \__kernel_primitive:NN \ptexrevision       \ptex_ptexrevision:D
2023  \__kernel_primitive:NN \ptexversion        \ptex_ptexversion:D
2024  \__kernel_primitive:NN \showmode          \ptex_showmode:D
2025  \__kernel_primitive:NN \sjis               \ptex_sjis:D
2026  \__kernel_primitive:NN \tate                \ptex_tate:D
2027  \__kernel_primitive:NN \tbaselineshift     \ptex_tbaselineshift:D
2028  \__kernel_primitive:NN \tfont               \ptex_tfont:D
2029  \__kernel_primitive:NN \xkanjiskip         \ptex_xkanjiskip:D
2030  \__kernel_primitive:NN \xspcode            \ptex_xspcode:D
2031  \__kernel_primitive:NN \ybaselineshift     \ptex_ybaselineshift:D
2032  \__kernel_primitive:NN \yoko                \ptex_yoko:D
2033  \__kernel_primitive:NN \disablecjktoken   \uptex_disablecjktoken:D
2034  \__kernel_primitive:NN \enablecjktoken    \uptex_enablecjktoken:D
2035  \__kernel_primitive:NN \forcecjktoken     \uptex_forcecjktoken:D
2036  \__kernel_primitive:NN \kchar               \uptex_kchar:D
2037  \__kernel_primitive:NN \kchardef           \uptex_kchardef:D
2038  \__kernel_primitive:NN \kuten              \uptex_kuten:D
2039  \__kernel_primitive:NN \ucs                \uptex_UCS:D
2040  \__kernel_primitive:NN \uptexrevision      \uptex_uptexrevision:D
2041  \__kernel_primitive:NN \uptexversion       \uptex_uptexversion:D
2042  }
2043  }
2044  \__kernel_primitives:
2045  \tex_endgroup:D
2046  </package>
2047  </initex | package>

```

### 3 Internal kernel functions

\\_\\_kernel\_chk\\_cs\\_exist:N  
\\_\\_kernel\_chk\\_cs\\_exist:c

\\_\\_kernel\_chk\\_cs\\_exist:N <cs>

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> {<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 {<expr>} <eval> {<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\_var\_exist:N

\\_\\_kernel\_chk\_var\_exist:N <var>

This function is only created if debugging is enabled. It checks that <var> is defined according to the criteria for \cs\_if\_exist\_p:N, and if not raises a kernel-level error.

\\_\\_kernel\_chk\_var\_scope>NN

\\_\\_kernel\_chk\_var\_scope>NN <scope> <var>

Checks the <var> has the correct <scope>, and if not raises a kernel-level error. This function is only created if debugging is enabled. The <scope> is a single letter l, g, c denoting local variables, global variables, or constants. More precisely, if the variable name starts with a letter and an underscore (normal expl3 convention) the function checks that this single letter matches the <scope>. Otherwise the function cannot know the scope <var> the first time: instead, it defines \\_\\_debug\_chk\_/<var name> to store that information for the next call. Thus, if a given <var> is subject to assignments of different scopes a kernel error will result.

\\_\\_kernel\_chk\_var\_local:N  
\\_\\_kernel\_chk\_var\_global:N

\\_\\_kernel\_chk\_var\_local:N <var>  
\\_\\_kernel\_chk\_var\_global:N <var>

Applies \\_\\_kernel\_chk\_var\_exist:N <var>, and assuming that is true applies \\_\\_kernel\_chk\_var\_scope>NN <scope> <var>, where <scope> is l or g.

\\_\\_kernel\_cs\_parm\_from\_arg\_count:nnF \\_\\_kernel\_cs\_parm\_from\_arg\_count:nnF {<follow-on>} {<args>} {<false code>}

Evaluates the number of <args> and leaves the <follow-on> code followed by a brace group containing the required number of primitive parameter markers (#1, etc.). If the number of <args> is outside the range [0, 9], the <false code> is inserted instead of the <follow-on>.

---

```
\__kernel_deprecation_code:nN \__kernel_deprecation_code:nN {⟨error code⟩} {⟨working code⟩}
```

Stores both an ⟨error⟩ and ⟨working⟩ definition for given material such that they can be exchanged by `\debug_on:` and `\debug_off:`.

---

```
\__kernel_if_debug:TF \__kernel_if_debug:TF {⟨true code⟩} {⟨false code⟩}
```

Runs the ⟨true code⟩ if debugging is enabled, namely only in L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> package mode with one of the options `check-declarations`, `enable-debug`, or `log-functions`. Otherwise runs the ⟨false code⟩. The T and F variants are not provided for this low-level conditional.

---

```
\__kernel_debug_log:x \__kernel_debug_log:x {⟨message text⟩}
```

If the `log-functions` option is active, this function writes the ⟨message text⟩ to the log file using `\iow_log:x`. Otherwise, the ⟨message text⟩ is ignored using `\use_none:n`. This function is only created if debugging is enabled.

---

```
\__kernel_exp_not:w * \__kernel_exp_not:w {⟨expandable tokens⟩} {⟨content⟩}
```

Carries out expansion on the ⟨expandable tokens⟩ before preventing further expansion of the ⟨content⟩ as for `\exp_not:n`. Typically, the ⟨expandable tokens⟩ will alter the nature of the ⟨content⟩, 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`.)

---

```
\__kernel_file_missing:n \__kernel_file_missing:n {⟨name⟩}
```

Expands the ⟨name⟩ as per `\__kernel_file_name_sanitze:nN` then produces an error message indicating that that file was not found.

---

```
\__kernel_file_name_sanitze:nN \__kernel_file_name_sanitze:nN {⟨name⟩} {str var}
```

For converting a ⟨name⟩ to a string where active characters are treated as strings.

---

```
\__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\_ior\_open:Nn  
\\_kernel\_ior\_open:No

\\_kernel\_ior\_open:Nn *stream* {*file name*}

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\_msg\_new:nnnn  
\\_kernel\_msg\_new:nnn

\\_kernel\_msg\_new:nnnn {*module*} {*message*} {*text*} {*more text*}

Creates a kernel *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 and expanded at the time the message is used. An error is raised if the *message* already exists.

---

\\_kernel\_msg\_set:nnnn  
\\_kernel\_msg\_set:nnn

\\_kernel\_msg\_set:nnnn {*module*} {*message*} {*text*} {*more text*}

Sets up the text for a kernel *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 and expanded at the time the message is used.

---

\\_kernel\_msg\_fatal:nnnnnn  
\\_kernel\_msg\_fatal:nnxxxx  
\\_kernel\_msg\_fatal:nnnnnn  
\\_kernel\_msg\_fatal:nnxxxx  
\\_kernel\_msg\_fatal:nnnn  
\\_kernel\_msg\_fatal:nnxx  
\\_kernel\_msg\_fatal:nnn  
\\_kernel\_msg\_fatal:nnx  
\\_kernel\_msg\_fatal:nn

\\_kernel\_msg\_fatal:nnnnnn {*module*} {*message*} {*arg one*} {*arg two*} {*arg three*} {*arg four*}

Issues kernel *module* error *message*, passing *arg one* to *arg four* to the text-creating functions. After issuing a fatal error the TeX run halts. Cannot be redirected.

---

\\_kernel\_msg\_error:nnnnnn  
\\_kernel\_msg\_error:nnxxxx  
\\_kernel\_msg\_error:nnnnnn  
\\_kernel\_msg\_error:nnxxxx  
\\_kernel\_msg\_error:nnnn  
\\_kernel\_msg\_error:nnxx  
\\_kernel\_msg\_error:nnn  
\\_kernel\_msg\_error:nnx  
\\_kernel\_msg\_error:nn

\\_kernel\_msg\_error:nnnnnn {*module*} {*message*} {*arg one*} {*arg two*} {*arg three*} {*arg four*}

Issues kernel *module* error *message*, passing *arg one* to *arg four* to the text-creating functions. The error stops processing and issues the text at the terminal. After user input, the run continues. Cannot be redirected.

---

```
\_\_kernel\_msg\_warning:nnnnnn \_\_kernel\_msg\_warning:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}
```

---

Issues kernel *<module>* warning *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The warning text is added to the log file, but the TeX run is not interrupted.

---

```
\_\_kernel\_msg\_info:nnnnnn \_\_kernel\_msg\_info:nnnnnnn {\<module>} {\<message>} {\<arg one>} {\<arg two>} {\<arg three>} {\<arg four>}
```

---

Issues kernel *<module>* information *<message>*, passing *<arg one>* to *<arg four>* to the text-creating functions. The information text is added to the log file.

---

```
\_\_kernel\_msg\_expandable\_error:nnnnnn * \_\_kernel\_msg\_expandable\_error:nnnnnnn {\<module>} {\<message>}
```

---

Issues an error, passing *<arg one>* to *<arg four>* to the text-creating functions. The resulting string must be much shorter than a line, otherwise it is cropped.

---

```
\_\_kernel\_patch:nnNNpn \_\_kernel\_patch:nnNNpn {\<before>} {\<after>}
```

---

If debugging is not enabled, this function ignores the *<before>* and *<after>* code and performs the *<definition>* with no patching. Otherwise it replaces *<code>* by *<before>* *<code>* *<after>* (which can involve #1 and so on) in the *<definition>* that follows. The *<definition>* must start with \cs\_new:Npn or \cs\_set:Npn or \cs\_gset:Npn or their \_protected counterparts. Other cases can be added as needed.

---

```
\_\_kernel\_patch\_conditional:nNNpnn \_\_kernel\_patch\_conditional:nNNpnn {\<before>}
```

---

Similar to \\_\\_kernel\\_patch:nnNNpn for conditionals, namely *<definition>* must be \prg\_new\_conditional:Npnn or its \_protected counterpart. There is no *<after>* code because that would interfere with the action of the conditional.

\\_kernel\_patch\_deprecation:nnNNpn \\_kernel\_patch\_deprecation:nnNNpn {<before>} {<after>} {definition} {function}{parameters} {type} {code}

Similar to \\_kernel\_patch:nnNNpn for deprecated functions.

```
\_kernel_patch_args:nNNpn          \_kernel_patch_args:nNNpn {<arguments>}\n\_\_kernel_patch_conditional_args:nNNppn  <definition> <function> <parameters> {<code>}
```

Like `\__kernel_patch:nnNpn`, this tweaks the following definition, but from the “inside out” (and if debugging is not enabled, the `<arguments>` are ignored). It replaces #1, #2 and so on in the `<code>` of the definition as indicated by the `<arguments>`. More precisely, a temporary function is defined using the `<definition>` with the `<parameters>` and `<code>`, then the result of expanding that function once in front of the `<arguments>` is used instead of the `<code>` when defining the actual function. For instance,

```
\__kernel_patch_args:nNnpn { { (#1) } }
\cs_new:Npn \int_eval:n #1
{ \int_value:w \__int_eval:w #1 \__int_eval_end: }
```

would replace #1 by (#1) in the definition of `\int_eval:n` when debugging is enabled. This fails if the `<code>` contains ##. The `\_kernel_patch_conditional_args:nNpnn` function is for use before `\prg_new_conditional:Npnn` or its `_protected` counterpart.

```
\_\_kernel\_patch\_args:nnnNNpn  
\_\_kernel\_patch\_conditional\_args:nnnNNpn
```

---

```
\_\_kernel\_patch\_args:nnnNNpn {{before}} {{after}}  
{{arguments}}  
(definition) (function) (parameters) {{code}}
```

A combination of \\_\\_kernel\\_patch:nnNNpn and \\_\\_kernel\\_patch\\_args:nNNpn.

`\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`.)

`\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  $\langle \text{min} \rangle$  and  $\langle \text{max} \rangle$  included. The  $\langle \text{min} \rangle$  and  $\langle \text{max} \rangle$  must be suitable for `\int_value:w` (and any `\int_eval:w` must be terminated by `\scan_stop:` or equivalent). For small ranges  $R = \langle \text{max} \rangle - \langle \text{min} \rangle + 1 \leq 2^{17} - 1$ ,  $\langle \text{min} \rangle - 1 + \_\_kernel\_randint:n\{R\}$  is faster.

---

```
\__kernel_register_show:N  
\__kernel_register_show:c
```

```
\__kernel_register_show:N <register>
```

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:c
```

```
\__kernel_register_log:N <register>
```

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.

## 4 13basics implementation

2048 `(*initex | package)`

### 4.1 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>5</sup>

```
\if_true:  
\if_false:  
  2049 \tex_let:D \if_true:  
  2050 \tex_let:D \if_false:  
  \or:  
  2051 \tex_let:D \or:  
  \else:  
  2052 \tex_let:D \else:  
  \reverse_if:N  
  2053 \tex_let:D \fi:  
  2054 \tex_let:D \reverse_if:N  
  2055 \tex_let:D \if:w  
  2056 \tex_let:D \if_charcode:w  
  2057 \tex_let:D \if_catcode:w  
  2058 \tex_let:D \if_meaning:w  
  Then some conditionals.  
  \tex_iftrue:D  
  \tex_iffalse:D  
  \tex_or:D  
  \tex_else:D  
  \tex_fi:D  
  \tex_unless:D  
  \tex_if:D  
  \tex_if:D  
  \tex_ifcat:D  
  \tex_ifx:D
```

(End definition for `\if_true:` and others. These functions are documented on page 20.)

<sup>5</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.

```

\if_mode_math: TEX lets us detect some if its modes.
\if_mode_horizontal:
2059 \tex_let:D \if_mode_math:           \tex_ifmmode:D
2060 \tex_let:D \if_mode_horizontal: \tex_ifhmode:D
2061 \tex_let:D \if_mode_vertical:   \tex_ifvmode:D
2062 \tex_let:D \if_mode_inner:     \tex_ifinner:D

(End definition for \if_mode_math: and others. These functions are documented on page 21.)
```

\if\_cs\_exist:N Building csnames and testing if control sequences exist.

```

\if_cs_exist:w 2063 \tex_let:D \if_cs_exist:N      \tex_ifdefined:D
    \cs:w       2064 \tex_let:D \if_cs_exist:w      \tex_ifcsname:D
\cs_end:        2065 \tex_let:D \cs:w            \tex_csname:D
                2066 \tex_let:D \cs_end:          \tex_endcsname:D

(End definition for \if_cs_exist:N and others. These functions are documented on page 21.)
```

\exp\_after:wN The five \exp\_ functions are used in the l3expan module where they are described.

```

\exp_not:N 2067 \tex_let:D \exp_after:wN      \tex_expandafter:D
\exp_not:n 2068 \tex_let:D \exp_not:N       \tex_noexpand:D
            2069 \tex_let:D \exp_not:n       \tex_unexpanded:D
            2070 \tex_let:D \exp:w         \tex_roman numeral:D
            2071 \tex_let:D \exp_end: = 0 ~

(End definition for \exp_after:wN, \exp_not:N, and \exp_not:n. These functions are documented on page 30.)
```

\token\_to\_meaning:N Examining a control sequence or token.

```

\cs_meaning:N 2072 \tex_let:D \token_to_meaning:N \tex_meaning:D
                2073 \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 122.)
```

\tl\_to\_str:n Making strings.

```

\token_to_str:N 2074 \tex_let:D \tl_to_str:n      \tex_detokenize:D
\__kernel_tl_to_str:w 2075 \tex_let:D \token_to_str:N      \tex_string:D
                        2076 \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 42.)
```

\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.

```

\group_begin: 2077 \tex_let:D \scan_stop:      \tex_relax:D
\group_end:   2078 \tex_let:D \group_begin:   \tex_begingroup:D
              2079 \tex_let:D \group_end:     \tex_endgroup:D

(End definition for \scan_stop:, \group_begin:, and \group_end:. These functions are documented on page 8.)
```

2080 ⟨@=int⟩

\if\_int\_compare:w For integers.

```

\__int_to_roman:w 2081 \tex_let:D \if_int_compare:w \tex_ifnum:D
                  2082 \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 91.)

`\group_insert_after:N` Adding material after the end of a group.

```
2083 \tex_let:D \group_insert_after:N \tex_aftergroup:D
```

(End definition for `\group_insert_after:N`. This function is documented on page 8.)

`\exp_args:Nc` Discussed in l3expan, but needed much earlier.

`\exp_args:cc`

```
2084 \tex_long:D \tex_def:D \exp_args:Nc #1#2
2085   { \exp_after:wN #1 \cs:w #2 \cs_end: }
2086 \tex_long:D \tex_def:D \exp_args:cc #1#2
2087   { \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 26.)

`\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:NNc`) 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.

```
2088 \tex_def:D \token_to_str:c { \exp_args:Nc \token_to_str:N }
2089 \tex_long:D \tex_def:D \cs_meaning:c #
2090   {
2091     \if_cs_exist:w #1 \cs_end:
2092       \exp_after:wN \use_i:nn
2093     \else:
2094       \exp_after:wN \use_ii:nn
2095     \fi:
2096     { \exp_args:Nc \cs_meaning:N {#1} }
2097     { \tl_to_str:n {undefined} }
2098   }
2099 \tex_let:D \token_to_meaning:c = \cs_meaning:c
```

(End definition for `\token_to_meaning:N`. This function is documented on page 122.)

## 4.2 Defining some constants

`\c_zero_int`

We need the constant `\c_zero_int` which is used by some functions in the l3alloc module. The rest are defined in the l3int module – at least for the ones that can be defined with `\tex_chardef:D` or `\tex_mathchardef:D`. For other constants the l3int module is required but it can't be used until the allocation has been set up properly!

```
2100 \tex_chardef:D \c_zero_int      = 0 ~
```

(End definition for `\c_zero_int`. This variable is documented on page 90.)

`\c_max_register_int`

This is here as this particular integer is needed both in package mode and to bootstrap l3alloc, and is documented in l3int.

```
2101 \tex_ifdefined:D \tex_luatexversion:D
2102   \tex_chardef:D \c_max_register_int = 65 535 ~
2103 \tex_else:D
2104   \tex_mathchardef:D \c_max_register_int = 32 767 ~
2105 \tex_fi:D
```

(End definition for `\c_max_register_int`. This variable is documented on page 90.)

## 4.3 Defining functions

We start by providing functions for the typical definition functions. First the local ones.

All assignment functions in L<sup>A</sup>T<sub>E</sub>X3 should be naturally protected; after all, the T<sub>E</sub>X primitives for assignments are and it can be a cause of problems if others aren't.

```

\cs_set_nopar:Npn
\cs_set_nopar:Npx
\cs_set:Npn
\cs_set:Npx
\cs_set_protected_nopar:Npn
\cs_set_protected_nopar:Npx
\cs_set_protected:Npn
\cs_set_protected:Npx
2106 \tex_let:D \cs_set_nopar:Npn           \tex_def:D
2107 \tex_let:D \cs_set_nopar:Npx           \tex_edef:D
2108 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npn
2109   { \tex_long:D \tex_def:D }
2110 \tex_protected:D \tex_long:D \tex_def:D \cs_set:Npx
2111   { \tex_long:D \tex_edef:D }
2112 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npn
2113   { \tex_protected:D \tex_def:D }
2114 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected_nopar:Npx
2115   { \tex_protected:D \tex_edef:D }
2116 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npn
2117   { \tex_protected:D \tex_long:D \tex_def:D }
2118 \tex_protected:D \tex_long:D \tex_def:D \cs_set_protected:Npx
2119   { \tex_protected:D \tex_long:D \tex_edef:D }

```

(End definition for `\cs_set_nopar:Npn` and others. These functions are documented on page 10.)

Global versions of the above functions.

```

\cs_gset_nopar:Npn
\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
2120 \tex_let:D \cs_gset_nopar:Npn           \tex_gdef:D
2121 \tex_let:D \cs_gset_nopar:Npx           \tex_xdef:D
2122 \cs_set_protected:Npn \cs_gset:Npn
2123   { \tex_long:D \tex_gdef:D }
2124 \cs_set_protected:Npn \cs_gset:Npx
2125   { \tex_long:D \tex_xdef:D }
2126 \cs_set_protected:Npn \cs_gset_protected_nopar:Npn
2127   { \tex_protected:D \tex_gdef:D }
2128 \cs_set_protected:Npn \cs_gset_protected_nopar:Npx
2129   { \tex_protected:D \tex_xdef:D }
2130 \cs_set_protected:Npn \cs_gset_protected:Npn
2131   { \tex_protected:D \tex_long:D \tex_gdef:D }
2132 \cs_set_protected:Npn \cs_gset_protected:Npx
2133   { \tex_protected:D \tex_long:D \tex_xdef:D }

```

(End definition for `\cs_gset_nopar:Npn` and others. These functions are documented on page 11.)

## 4.4 Selecting tokens

2134 `<@@=exp>`

`\l__exp_internal_tl` Scratch token list variable for `\l3expan`, used by `\use:x`, used in defining conditionals. We don't use `tl` methods because `l3basics` is loaded earlier.

2135 `\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.

2136 `\cs_set:Npn \use:c #1 { \cs:w #1 \cs_end: }`

(End definition for `\use:c`. This function is documented on page 15.)

**\use:x** Fully expands its argument and passes it to the input stream. Uses the reserved `\l__exp_internal_tl` which will be set up in `\3expan`.

```
2137 \cs_set_protected:Npn \use:x #1
2138 {
2139     \cs_set_nopar:Npx \l__exp_internal_tl {#1}
2140     \l__exp_internal_tl
2141 }
```

(End definition for `\use:x`. This function is documented on page 18.)

```
2142 ⟨@@=use⟩
```

**\use:e** Currently LuaTeX-only: emulated for older engines.

```
2143 \cs_set:Npn \use:e #1 { \tex_expanded:D {#1} }
2144 \tex_ifdefined:D \tex_expanded:D \tex_else:D
2145     \cs_set:Npn \use:e #1 { \exp_args:N \use:n {#1} }
2146 \tex_if:D
```

(End definition for `\use:e`. This function is documented on page 18.)

```
2147 ⟨@@=exp⟩
```

**\use:n** These macros grab their arguments and return them back to the input (with outer braces removed).

```
2148 \cs_set:Npn \use:n    #1      {#1}
2149 \cs_set:Npn \use:nn   #1#2    {#1#2}
2150 \cs_set:Npn \use:nnn #1#2#3 {#1#2#3}
2151 \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 17.)

**\use\_i:nn** The equivalent to L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub>'s `\@firstoftwo` and `\@secondoftwo`.

```
2152 \cs_set:Npn \use_i:nn  #1#2 {#1}
2153 \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 17.)

**\use\_i:nnn** We also need something for picking up arguments from a longer list.

```
2154 \cs_set:Npn \use_i:nnn  #1#2#3 {#1}
2155 \cs_set:Npn \use_i:nnn  #1#2#3 {#2}
2156 \cs_set:Npn \use_i:nnn  #1#2#3 {#3}
2157 \cs_set:Npn \use_i:nnn  #1#2#3 {#1#2}
2158 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#1}
2159 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#2}
2160 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#3}
2161 \cs_set:Npn \use_i:nnnn #1#2#3#4 {#4}
```

(End definition for `\use_i:nnn` and others. These functions are documented on page 17.)

**\use\_none\_delimit\_by\_q\_nil:w** Functions that gobble everything until they see either `\q_nil`, `\q_stop`, or `\q_recursion_stop`, respectively.

```
2162 \cs_set:Npn \use_none_delimit_by_q_nil:w #1 \q_nil { }
2163 \cs_set:Npn \use_none_delimit_by_q_stop:w #1 \q_stop { }
2164 \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 18.)

`\use_i_delimit_by_q_nil: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.

```
2165 \cs_set:Npn \use_i_delimit_by_q_nil:nw #1#2 \q_nil {#1}
2166 \cs_set:Npn \use_i_delimit_by_q_stop:nw #1#2 \q_stop {#1}
2167 \cs_set:Npn \use_i_delimit_by_q_recursion_stop:nw
2168 #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 19.)

## 4.5 Gobbling tokens from input

`\use_none:n` 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.

```
2169 \cs_set:Npn \use_none:n #1 { }
2170 \cs_set:Npn \use_none:nn #1#2 { }
2171 \cs_set:Npn \use_none:nnn #1#2#3 { }
2172 \cs_set:Npn \use_none:nnnn #1#2#3#4 { }
2173 \cs_set:Npn \use_none:nnnnn #1#2#3#4#5 { }
2174 \cs_set:Npn \use_none:nnnnnn #1#2#3#4#5#6 { }
2175 \cs_set:Npn \use_none:nnnnnnn #1#2#3#4#5#6#7 { }
2176 \cs_set:Npn \use_none:nnnnnnnn #1#2#3#4#5#6#7#8 { }
2177 \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 18.)

## 4.6 Debugging and patching later definitions

2178 `\@@=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. At present, debugging is disabled in the format and in generic mode, while in L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  mode it is enabled if one of the options `enable-debug`, `log-functions` or `check-declarations` was given.

```
2179 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#2}
2180 {*package}
2181 \tex_ifodd:D \l@expl@enable@debug@bool
2182 \cs_set_protected:Npn \__kernel_if_debug:TF #1#2 {#1}
2183 \fi:
2184 
```

(End definition for `\__kernel_if_debug:TF`.)

```
\debug_on:n
\debug_off:n
\__kernel_if_debug:TF
\__debug_all_on:
\__debug_all_off:
2185 \__kernel_if_debug:TF
2186 {
2187 \cs_set_protected:Npn \debug_on:n #1
```

```

2188 {
2189     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
2190     {
2191         \cs_if_exist_use:cF { __debug_ ##1 _on: }
2192         { \__kernel_msg_error:nnn { kernel } { debug } {##1} }
2193     }
2194 }
2195 \cs_set_protected:Npn \debug_off:n #1
2196 {
2197     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
2198     {
2199         \cs_if_exist_use:cF { __debug_ ##1 _off: }
2200         { \__kernel_msg_error:nnn { kernel } { debug } {##1} }
2201     }
2202 }
2203 \cs_set_protected:Npn \__debug_all_on:
2204 {
2205     \debug_on:n
2206     {
2207         check-declarations ,
2208         check-expressions ,
2209         deprecation ,
2210         log-functions ,
2211     }
2212 }
2213 \cs_set_protected:Npn \__debug_all_off:
2214 {
2215     \debug_off:n
2216     {
2217         check-declarations ,
2218         check-expressions ,
2219         deprecation ,
2220         log-functions ,
2221     }
2222 }
2223 }
2224 {
2225     \cs_set_protected:Npn \debug_on:n #1
2226     {
2227         \__kernel_msg_error:nnx { kernel } { enable-debug }
2228         { \tl_to_str:n { \debug_on:n {#1} } }
2229     }
2230     \cs_set_protected:Npn \debug_off:n #1
2231     {
2232         \__kernel_msg_error:nnx { kernel } { enable-debug }
2233         { \tl_to_str:n { \debug_off:n {#1} } }
2234     }
2235 }

```

(End definition for `\debug_on:n` and others. These functions are documented on page 242.)

`\debug_suspend:` Suspend and resume locally all debug-related errors and logging except deprecation errors.  
`\debug_resume:` The `\debug_suspend:` and `\debug_resume:` pairs can be nested. We keep track of

`\__debug_suspended:T`  
`\l__debug_suspended_tl`

nesting in a token list containing a number of periods. At first begin with the “non-suspended” version of `\_debug_suspended:T`.

```

2236 \_kernel_if_debug:TF
2237 {
2238     \cs_set_nopar:Npn \l__debug_suspended_tl { }
2239     \cs_set_protected:Npn \debug_suspend:
2240     {
2241         \tl_put_right:Nn \l__debug_suspended_tl { . }
2242         \cs_set_eq:NN \__debug_suspended:T \use:n
2243     }
2244     \cs_set_protected:Npn \debug_resume:
2245     {
2246         \tl_set:Nx \l__debug_suspended_tl
2247             { \tl_tail:N \l__debug_suspended_tl }
2248         \tl_if_empty:NT \l__debug_suspended_tl
2249             {
2250                 \cs_set_eq:NN \__debug_suspended:T \use_none:n
2251             }
2252         \cs_set:Npn \__debug_suspended:T #1 { }
2253     }
2254 }
2255 {
2256     \cs_set_protected:Npn \debug_suspend: { }
2257     \cs_set_protected:Npn \debug_resume: { }
2258 }
```

(End definition for `\debug_suspend:` and others. These functions are documented on page [242](#).)

`\_debug_check-declarations_on:`  
`\_debug_check-declarations_off:`  
`\_kernel_chk_var_exist:N`  
`\_kernel_chk_cs_exist:N`  
`\_kernel_chk_cs_exist:c`  
`\_kernel_chk_var_local:N`  
`\_kernel_chk_var_global:N`  
`\_kernel_chk_var_scope>NN`

When debugging is enabled these two functions set up functions that test their argument (when `check-declarations` is active)

- `\_kernel_chk_var_exist:N` and `\_kernel_chk_cs_exist:N`, two functions that test that their argument is defined;
- `\_kernel_chk_var_scope>NN` that checks that its argument #2 has scope #1.
- `\_kernel_chk_var_local:N` and `\_kernel_chk_var_global:N` that perform both checks.

```

2259 \_kernel_if_debug:TF
2260 {
2261     \exp_args:Nc \cs_set_protected:Npn { __debug_check-declarations_on: }
2262     {
2263         \cs_set_protected:Npn \_kernel_chk_var_exist:N ##1
2264         {
2265             \_debug_suspended:T \use_none:nnn
2266             \cs_if_exist:NF ##1
2267             {
2268                 \_kernel_msg_error:nnx { kernel } { non-declared-variable }
2269                 { \token_to_str:N ##1 }
2270             }
2271         }
2272         \cs_set_protected:Npn \_kernel_chk_cs_exist:N ##1
2273         {
2274             \_debug_suspended:T \use_none:nnn
```

```

2275   \cs_if_exist:NF ##1
2276   {
2277     \_kernel_msg_error:nnx { kernel } { command-not-defined }
2278     { \token_to_str:N ##1 }
2279   }
2280 }
2281 \cs_set_protected:Npn \_kernel_chk_var_scope:NN
2282 {
2283   \__debug_suspended:T \use_none:nnn
2284   \__debug_chk_var_scope_aux:NN
2285 }
2286 \cs_set_protected:Npn \_kernel_chk_var_local:N ##1
2287 {
2288   \__debug_suspended:T \use_none:nnnnn
2289   \_kernel_chk_var_exist:N ##1
2290   \__debug_chk_var_scope_aux:NN l ##1
2291 }
2292 \cs_set_protected:Npn \_kernel_chk_var_global:N ##1
2293 {
2294   \__debug_suspended:T \use_none:nnnnnn
2295   \_kernel_chk_var_exist:N ##1
2296   \__debug_chk_var_scope_aux:NN g ##1
2297 }
2298 }
2299 \exp_args:Nc \cs_set_protected:Npn { __debug_check-declarations_off: }
2300 {
2301   \cs_set_protected:Npn \_kernel_chk_var_exist:N ##1 { }
2302   \cs_set_protected:Npn \_kernel_chk_cs_exist:N ##1 { }
2303   \cs_set_protected:Npn \_kernel_chk_var_local:N ##1 { }
2304   \cs_set_protected:Npn \_kernel_chk_var_global:N ##1 { }
2305   \cs_set_protected:Npn \_kernel_chk_var_scope:NN ##1##2 { }
2306 }
2307 \cs_set_protected:Npn \_kernel_chk_cs_exist:c
2308 {
2309   \exp_args:Nc \_kernel_chk_cs_exist:N
2310   \tex_ifodd:D \l@expl@check@declarations@bool
2311   \use:c { __debug_check-declarations_on: }
2312   \else:
2313   \use:c { __debug_check-declarations_off: }
2314   \fi:
2315 }
2316 { }

```

(End definition for `\__debug_check-declarations_on:` and others.)

`\_debug_chk_var_scope_aux:NN` First check whether the name of the variable #2 starts with  $\langle letter \rangle$ . If it does then pass that letter, the  $\langle scope \rangle$ , and the variable name to `\__debug_chk_var_scope_aux:NN`.  
`\_debug_chk_var_scope_aux:Nn` That function compares the two letters and triggers an error if they differ (the `\scan_stop:` case is not reachable here). If the second character was not `_` then pass the same data to the same auxiliary, except for its first argument which is now a control sequence. That control sequence is actually a token list (but to avoid triggering the checking code we manipulate it using `\cs_set_nopar:Npn`) containing a single letter  $\langle scope \rangle$  according to what the first assignment to the given variable was.

```

2316 \_kernel_if_debug:TF
2317 {

```

```

2318   \cs_set_protected:Npn \__debug_chk_var_scope_aux:NN #1#2
2319     { \exp_args:NNf \__debug_chk_var_scope_aux:Nn #1 { \cs_to_str:N #2 } }
2320   \cs_set_protected:Npn \__debug_chk_var_scope_aux:Nn #1#2
2321   {
2322     \if:w _ \use_i:nn \use_i_delimit_by_q_stop:nw #2 ? ? \q_stop
2323       \exp_after:wN \__debug_chk_var_scope_aux:NNn
2324         \use_i_delimit_by_q_stop:nw #2 ? \q_stop
2325         #1 {#2}
2326     \else:
2327       \exp_args:Nc \__debug_chk_var_scope_aux:NNn
2328         { __debug_chk_ / #2 }
2329         #1 {#2}
2330     \fi:
2331   }
2332   \cs_set_protected:Npn \__debug_chk_var_scope_aux:NNn #1#2#3
2333   {
2334     \if:w #1 #2
2335     \else:
2336       \if:w #1 \scan_stop:
2337         \cs_gset_nopar:Npn #1 {#2}
2338       \else:
2339         \__kernel_msg_error:nnxxx { kernel } { local-global }
2340         {#1} {#2} { \iow_char:N \\ #3 }
2341       \fi:
2342     \fi:
2343   }
2344 }
2345 { }

```

(End definition for `\__debug_chk_var_scope_aux:NN`, `\__debug_chk_var_scope_aux:Nn`, and `\__debug_chk_var_scope_aux:NNn`.)

`\__debug_check-expressions_on:`  
`\__debug_check-expressions_off:`  
`\__kernel_chk_expr:nNnN`  
`\__debug_chk_expr_aux:nNnN`

When debugging is enabled these two functions set `\__kernel_chk_expr:nNnN` to test or not whether the given expression is valid. The idea is to evaluate the expression within a brace group (to catch trailing `\use_none:nn` or similar), then test that the result is what we expect. This is done by turning it to an integer and hitting that with `\tex_roman numeral:D` after replacing the first character by `-0`. If all goes well, that primitive finds a non-positive integer and gives an empty output. If the original expression evaluation stopped early it leaves a trailing `\tex_relax:D`, which stops the second evaluation (used to convert to integer) before it encounters the final `\tex_relax:D`. Since `\tex_roman numeral:D` does not absorb `\tex_relax:D` the output will be nonempty. Note that `#3` is empty except for mu expressions for which it is `\tex_mutoglu:D` to avoid an “incompatible glue units” error. Note also that if we had omitted the first `\tex_relax:D` then for instance `1+2\relax+3` would incorrectly be accepted as a valid integer expression.

```

2346 \__kernel_if_debug:TF
2347 {
2348   \exp_args:Nc \cs_set_protected:Npn { __debug_check-expressions_on: }
2349   {
2350     \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2
2351     {
2352       \__debug_suspended:T { ##1 \use_none:nnnnnn }
2353       \exp_after:wN \__debug_chk_expr_aux:nNnN

```

```

2354           \exp_after:wN { \tex_the:D ##2 ##1 \scan_stop: }
2355           ##2
2356       }
2357   }
2358 \exp_args:Nc \cs_set_protected:Npn { __debug_check-expressions_off: }
2359   { \cs_set:Npn \__kernel_chk_expr:nNnN ##1##2##3##4 {##1} }
2360 \use:c { __debug_check-expressions_off: }
2361 \cs_set:Npn \__debug_chk_expr_aux:nNnN #1#2#3#4
2362   {
2363     \tl_if_empty:oF
2364     {
2365       \tex_roman numeral:D - 0
2366       \exp_after:wN \use_none:n
2367       \int_value:w #3 #2 #1 \scan_stop:
2368     }
2369   {
2370     \__kernel_msg_expandable_error:nnnn
2371       { kernel } { expr } {#4} {#1}
2372   }
2373   #1
2374 }
2375 }
2376 { }

```

(End definition for `\__debug_check-expressions_on:` and others.)

`\__debug_log-functions_on:` These two functions (corresponding to the `expl3` option `log-functions`) control whether `\__kernel_debug_log:x` writes to the log file or not. Since `\iow_log:x` does not yet have its final definition we do not use `\cs_set_eq:NN` (not defined yet anyway). Once everything is defined, turn logging on or off depending on what option was given. When debugging is not enabled, simply produce an error.

```

2377 \__kernel_if_debug:TF
2378   {
2379     \exp_args:Nc \cs_set_protected:Npn { __debug_log-functions_on: }
2380     {
2381       \cs_set_protected:Npn \__kernel_debug_log:x
2382         { \__debug_suspended:T \use_none:nn \iow_log:x }
2383     }
2384     \exp_args:Nc \cs_set_protected:Npn { __debug_log-functions_off: }
2385       { \cs_set_protected:Npn \__kernel_debug_log:x { \use_none:n } }
2386     \tex_ifodd:D \l@expl@log@functions@bool
2387       \use:c { __debug_log-functions_on: }
2388     \else:
2389       \use:c { __debug_log-functions_off: }
2390     \fi:
2391   }
2392 { }

```

(End definition for `\__debug_log-functions_on:`, `\__debug_log-functions_off:`, and `\__kernel_debug_log:x`.)

`\__debug_deprecation_on:` Some commands were more recently deprecated and not yet removed; only make these into errors if the user requests it. This relies on two token lists, mostly filled up by calls to `\__kernel_patch_deprecation:nnNpn` in each module.

```

\__debug_deprecation_on:
\__debug_deprecation_off:
  \__kernel_deprecation_code:nn
\g__debug_deprecation_on_tl
\g__debug_deprecation_off_tl

```

```

2393 \__kernel_if_debug:TF
2394 {
2395     \cs_set_protected:Npn \__debug_deprecation_on:
2396         { \g__debug_deprecation_on_tl }
2397     \cs_set_protected:Npn \__debug_deprecation_off:
2398         { \g__debug_deprecation_off_tl }
2399     \cs_set_nopar:Npn \g__debug_deprecation_on_tl { }
2400     \cs_set_nopar:Npn \g__debug_deprecation_off_tl { }
2401     \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2
2402     {
2403         \tl_gput_right:Nn \g__debug_deprecation_on_tl {#1}
2404         \tl_gput_right:Nn \g__debug_deprecation_off_tl {#2}
2405     }
2406 }
2407 {
2408     \cs_set_protected:Npn \__kernel_deprecation_code:nn #1#2 { }
2409 }

```

(End definition for `\__debug_deprecation_on:` and others.)

```
\__kernel_patch_deprecation:nnNNpn
\__debug_deprecation_aux:nnNnn
```

Grab a definition (at present, must be `\cs_new_protected:Npn` or `\cs_new:Npn`). Add to `\g__debug_deprecation_on_tl` some code that makes the defined macro #3 outer (and defines it as an error). Add to `\g__debug_deprecation_off_tl` the definition itself. In both cases we undefine the token with `\tex_let:D` to avoid taking a potentially outer macro as the argument of some `expl3` function. Finally, define the macro itself: if it is protected, make it produce a warning then redefine and call itself. The macro initially takes no parameters: together with the x-expanding assignment and `\exp_not:n` this gives a convenient way of storing the macro's definition in itself in order to only produce the warning once for each macro. If debugging is disabled, `\__kernel_patch_deprecation:nnNNpn` lets the definition happen.

```

2410 \__kernel_if_debug:TF
2411 {
2412     \cs_set_protected:Npn \__kernel_patch_deprecation:nnNNpn #1#2#3#4#5#
2413     {
2414         \if_meaning:w \cs_new_protected:Npn #3
2415             \exp_after:wN \use_i:nn
2416         \else:
2417             \if_meaning:w \cs_new:Npn #3
2418                 \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
2419             \else:
2420                 \__kernel_msg_error:nnx { kernel } { debug-unpatchable }
2421                 { \token_to_str:N #3 ~ (for~deprecation) }
2422                 \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
2423             \fi:
2424         \fi:
2425         { \__debug_deprecation_aux:nnNnn {#1} {#2} #4 {#5} }
2426         { \__debug_deprecation_expandable:nnNnn {#1} {#2} #4 {#5} }
2427     }
2428     \cs_set_protected:Npn \__debug_deprecation_aux:nnNnn #1#2#3#4#5
2429     {
2430         \tl_gput_right:Nn \g__debug_deprecation_on_tl
2431         {
2432             \tex_let:D #3 \scan_stop:
2433             \__kernel_deprecation_error:Nnn #3 {#2} {#1}

```

```

2434     }
2435     \tl_gput_right:Nn \g__debug_deprecation_off_tl
2436     {
2437         \tex_let:D #3 \scan_stop:
2438         \cs_set_protected:Npn #3 #4 {#5}
2439     }
2440     \cs_new_protected:Npx #3
2441     {
2442         \exp_not:N \_kernel_msg_warning:nnxxx
2443         { kernel } { deprecated-command }
2444         {#1} { \token_to_str:N #3 } { \tl_to_str:n {#2} }
2445         \exp_not:n { \cs_gset_protected:Npn #3 #4 {#5} }
2446         \exp_not:N #3
2447     }
2448 }
2449 \cs_set_protected:Npn \_debug_deprecation_expandable:nnNnn #1#2#3#4#5
2450 {
2451     \tl_gput_right:Nn \g__debug_deprecation_on_tl
2452     {
2453         \tex_let:D #3 \scan_stop:
2454         \_kernel_deprecation_error:Nnn #3 {#2} {#1}
2455     }
2456     \tl_gput_right:Nn \g__debug_deprecation_off_tl
2457     {
2458         \tex_let:D #3 \scan_stop:
2459         \cs_set:Npn #3 #4 {#5}
2460     }
2461     \cs_new:Npn #3 #4 {#5}
2462 }
2463 }
2464 { \cs_set_protected:Npn \_kernel_patch_deprecation:nnNNpn #1#2 { } }

(End definition for \_kernel_patch_deprecation:nnNNpn and \_debug_deprecation_aux:nnNnn.)

```

\\_kernel\_patch:nnNNpn  
\\_kernel\_patch\_conditional:nNNpnn  
\\_debug\_patch\_aux:nnnn  
\\_debug\_patch\_auxii:nnnn

When debugging is not enabled, \\_kernel\_patch:nnNNpn and \\_kernel\_patch\_conditional:nNNpnn throw the patch away. Otherwise they can be followed by \cs\_new:Npn (or similar), and \prg\_new\_conditional:Npnn (or similar), respectively. In each case, grab the name of the function to be defined and its parameters then insert tokens before and/or after the definition.

```

2465 \_kernel_if_debug:TF
2466 {
2467     \cs_set_protected:Npn \_kernel_patch:nnNNpn #1#2#3#4#5#
2468     { \_debug_patch_aux:nnnn {#1} {#2} { #3 #4 #5 } }
2469     \cs_set_protected:Npn \_kernel_patch_conditional:nNNpnn #1#2#3#4#
2470     { \_debug_patch_auxii:nnnn {#1} { #2 #3 #4 } }
2471     \cs_set_protected:Npn \_debug_patch_aux:nnnn #1#2#3#4
2472     { #3 { #1 #4 #2 } }
2473     \cs_set_protected:Npn \_debug_patch_auxii:nnnn #1#2#3#4
2474     { #2 {#3} { #1 #4 } }
2475 }
2476 {
2477     \cs_set_protected:Npn \_kernel_patch:nnNNpn #1#2 { }
2478     \cs_set_protected:Npn \_kernel_patch_conditional:nNNpnn #1 { }
2479 }

```

(End definition for `\_kernel_patch:nNnpn` and others.)

`\_kernel_patch_args:nNnpn`  
`\_kernel_patch_conditional_args:nNnpnn`  
`\_kernel_patch_args:nnnNNpn`  
`\_kernel_patch_conditional_args:nnnNNpnn`

See `\_kernel_patch:nNnpn`. The first argument is something like `{#1}{(#2)}`. Define a temporary macro using the `<parameters>` and `<code>` of the definition that follows, then expand that temporary macro in front of the first argument to obtain new `<code>`. Then perform the definition as if that new `<code>` was directly typed in the file.

```

2480 \cs_set_protected:Npn \_kernel_patch_args:nNnpn
2481   { \_kernel_patch_args:nnnNNpn { } { } }
2482 \cs_set_protected:Npn \_kernel_patch_conditional_args:nNnpnn
2483   { \_kernel_patch_conditional_args:nnnNNpnn { } { } }
2484 \_kernel_if_debug:TF
2485   {
2486     \cs_set_protected:Npn \_kernel_patch_args:nnnNNpn #1#2#3#4#5#6#
2487       { \_debug_patch_args_aux:nnnNNnn {#1} {#2} {#3} #4 #5 {#6} }
2488     \cs_set_protected:Npn \_kernel_patch_conditional_args:nnnNNpnn
2489       #1#2#3#4#5#6#
2490       { \_debug_patch_args_aux:nnnNNnnn {#1} {#2} {#3} #4 #5 {#6} }
2491     \cs_set_protected:Npn \_debug_patch_args_aux:nnnNNnn #1#2#3#4#5#6#7
2492     {
2493       \cs_set:Npn \_debug_tmp:w #6 {#7}
2494       \exp_after:wN \_debug_patch_args_aux:nnnn \exp_after:wN
2495         { \_debug_tmp:w #3 } { #4 #5 #6 } {#1} {#2}
2496     }
2497     \cs_set_protected:Npn \_debug_patch_args_aux:nnnNNnnn #1#2#3#4#5#6#7#8
2498     {
2499       \cs_set:Npn \_debug_tmp:w #6 {#8}
2500       \exp_after:wN \_debug_patch_args_aux:nnnn \exp_after:wN
2501         { \_debug_tmp:w #3 } { #4 #5 #6 {#7} } {#1} {#2}
2502     }
2503     \cs_set_protected:Npn \_debug_patch_args_aux:nnnn #1#2#3#4
2504       { #2 { #3 #1 #4 } }
2505   }
2506   {
2507     \cs_set_protected:Npn \_kernel_patch_args:nnnNNpn #1#2#3 { }
2508     \cs_set_protected:Npn \_kernel_patch_conditional_args:nnnNNpnn
2509       #1#2#3 { }
2510   }

```

(End definition for `\_kernel_patch_args:nNnpn` and others.)

## 4.7 Conditional processing and definitions

2511 `<@@=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:` 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.
- `\prg_return_false:`

```
2512 \cs_set:Npn \prg_return_true:  
2513   { \exp_after:wN \use_i:nn \exp:w }  
2514 \cs_set:Npn \prg_return_false:  
2515   { \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 97.)

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.

```
2516 \cs_set_protected:Npn \prg_set_conditional:Npnn  
2517   { \__prg_generate_conditional_parm:NNNpnn \cs_set:Npn e }  
2518 \cs_set_protected:Npn \prg_new_conditional:Npnn  
2519   { \__prg_generate_conditional_parm:NNNpnn \cs_new:Npn e }  
2520 \cs_set_protected:Npn \prg_set_protected_conditional:Npnn  
2521   { \__prg_generate_conditional_parm:NNNpnn \cs_set_protected:Npn p }  
2522 \cs_set_protected:Npn \prg_new_protected_conditional:Npnn  
2523   { \__prg_generate_conditional_parm:NNNpnn \cs_new_protected:Npn p }  
2524 \cs_set_protected:Npn \__prg_generate_conditional_parm:NNNpnn #1#2#3#4#  
2525   {  
2526     \use:x  
2527     {  
2528       \__prg_generate_conditional:nnNNNnnn  
2529       \cs_split_function:N #3  
2530     }  
2531     #1 #2 {#4}  
2532   }
```

(End definition for `\prg_set_conditional:Npnn` and others. These functions are documented on page 95.)

```
\prg_set_conditional:Nnn
\prg_new_conditional:Nnn
\prg_set_protected_conditional:Nnn
\prg_new_protected_conditional:Nnn
\__prg_generate_conditional_count:NNNnn
\__prg_generate_conditional_count:nnNNNnn
```

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.

```
2533 \cs_set_protected:Npn \prg_set_conditional:Nnn
2534   { \__prg_generate_conditional_count:NNNnn \cs_set:Npn e }
2535 \cs_set_protected:Npn \prg_new_conditional:Nnn
2536   { \__prg_generate_conditional_count:NNNnn \cs_new:Npn e }
2537 \cs_set_protected:Npn \prg_set_protected_conditional:Nnn
2538   { \__prg_generate_conditional_count:NNNnn \cs_set_protected:Npn p }
2539 \cs_set_protected:Npn \prg_new_protected_conditional:Nnn
2540   { \__prg_generate_conditional_count:NNNnn \cs_new_protected:Npn p }
2541 \cs_set_protected:Npn \__prg_generate_conditional_count:NNNnn #1#2#3
2542   {
2543     \use:x
2544     {
2545       \__prg_generate_conditional_count:nnNNNnn
2546       \cs_split_function:N #3
2547     }
2548     #1 #2
2549   }
2550 \cs_set_protected:Npn \__prg_generate_conditional_count:nnNNNnn #1#2#3#4#5
2551   {
2552   \__kernel_cs_parm_from_arg_count:nnF
2553   { \__prg_generate_conditional:nnNNNnnn {#1} {#2} #3 #4 #5 }
2554   { \tl_count:n {#2} }
2555   {
2556     \__kernel_msg_error:nxxx { kernel } { bad-number-of-arguments }
2557     { \token_to_str:c { #1 : #2 } }
2558     { \tl_count:n {#2} }
2559     \use_none:nn
2560   }
2561 }
```

(End definition for `\prg_set_conditional:Nnn` and others. These functions are documented on page 95.)

```
\__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.

2562 \cs_set_protected:Npn \__prg_generate_conditional:nnNNnnn #1#2#3#4#5#6#7#8
2563 {
2564     \if_meaning:w \c_false_bool #3
2565         \__kernel_msg_error:nnx { kernel } { missing-colon }
2566             { \token_to_str:c {#1} }
2567             \exp_after:wN \use_none:nn
2568     \fi:
2569     \use:x
2570     {
2571         \exp_not:N \__prg_generate_conditional>NNnnnnNw
2572         \exp_not:n { #4 #5 {#1} {#2} {#6} }
2573         \__prg_generate_conditional_test:w
2574             #8 \q_mark
2575                 \__prg_generate_conditional_fast:nw
2576                 \prg_return_true: \else: \prg_return_false: \fi: \q_mark
2577                     \use_none:n
2578                     \exp_not:n { {#8} \use_i_ii:nnn }
2579                     \tl_to_str:n {#7}
2580                     \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
2581                 }
2582             }
2583 \cs_set:Npn \__prg_generate_conditional_test:w
2584     #1 \prg_return_true: \else: \prg_return_false: \fi: \q_mark #2
2585     { #2 {#1} }
2586 \cs_set:Npn \__prg_generate_conditional_fast:nw #1#2 \exp_not:n #3
2587     { \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.

```

2588 \cs_set_protected:Npn \__prg_generate_conditional>NNnnnnNw #1#2#3#4#5#6#7#8 ,
2589 {
2590     \if_meaning:w \q_recursion_tail #8
2591         \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2592     \fi:
2593     \use:c { __prg_generate_ #8 _form:wNNnnnnN }
2594         \tl_if_empty:nF {#8}
2595         {
2596             \__kernel_msg_error:nnxx
2597                 { kernel } { conditional-form-unknown }
2598                 {#8} { \token_to_str:c { #3 : #4 } }
2599         }
2600             \use_none:nnnnnnnn
2601             \q_stop
2602                 #1 #2 {#3} {#4} {#5} {#6} #7
2603                 \__prg_generate_conditional>NNnnnnNw #1 #2 {#3} {#4} {#5} {#6} #7
2604             }

```

(End definition for \\_\_prg\_generate\_conditional:nnNNnnn and others.)

```
\_\_prg\_generate\_p\_form:wNNnnnnN
\_\_prg\_generate\_TF\_form:wNNnnnnN
\_\_prg\_generate\_T\_form:wNNnnnnN
\_\_prg\_generate\_F\_form:wNNnnnnN
\_\_prg\_p\_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: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 could replace `\exp_after:wN \use_i:nnn` and similar by a single macro similar to `\_\_prg_p_true:w`. The drawback is that if the T or F arguments are actually missing, the recovery from the runaway argument would not insert `\fi:` back, messing up nesting of conditionals.

```
2605 \cs_set_protected:Npn \_\_prg_generate_p_form:wNNnnnnN
2606     #1 \q_stop #2#3#4#5#6#7#8
2607     {
2608         \if_meaning:w e #3
2609             \exp_after:wN \use_i:nn
2610         \else:
2611             \exp_after:wN \use_i:nn
2612         \fi:
2613         {
2614             #8
2615             { \exp_args:Nc #2 { #4 _p: #5 } #6 }
2616             { #7 \exp_end: \c_true_bool \c_false_bool } }
2617             { #7 \_\_prg_p_true:w \fi: \c_false_bool }
2618         }
2619         {
2620             \_\_kernel_msg_error:nnx { kernel } { protected-predicate }
2621             { \token_to_str:c { #4 _p: #5 } }
2622         }
2623     }
2624 \cs_set_protected:Npn \_\_prg_generate_T_form:wNNnnnnN
2625     #1 \q_stop #2#3#4#5#6#7#8
2626     {
2627         #8
2628             { \exp_args:Nc #2 { #4 : #5 T } #6 }
2629             { { #7 \exp_end: \use:n \use_none:n } }
2630             { #7 \exp_after:wN \use_i:nn \fi: \use_none:n }
2631         }
2632 \cs_set_protected:Npn \_\_prg_generate_F_form:wNNnnnnN
2633     #1 \q_stop #2#3#4#5#6#7#8
2634     {
2635         #8
2636             { \exp_args:Nc #2 { #4 : #5 F } #6 }
2637             { { #7 \exp_end: { } } }
2638             { #7 \exp_after:wN \use_none:nn \fi: \use:n }
2639         }
2640 \cs_set_protected:Npn \_\_prg_generate_TF_form:wNNnnnnN
2641     #1 \q_stop #2#3#4#5#6#7#8
2642     {
2643         #8
2644             { \exp_args:Nc #2 { #4 : #5 TF } #6 }
```

```

2645      { { #7 \exp_end: } }
2646      { #7 \exp_after:wN \use_ii:nnn \fi: \use_ii:nn }
2647    }
2648 \cs_set:Npn \__prg_p_true:w \fi: \c_false_bool { \fi: \c_true_bool }

(End definition for \__prg_generate_p_form:wNNnnnnN and others.)

```

\prg\_set\_eq\_conditional:NNn \prg\_new\_eq\_conditional:NNn \\_\_prg\_set\_eq\_conditional:NNNn The setting-equal functions. Split both functions and feed {\langle name<sub>1</sub> \rangle} {\langle signature<sub>1</sub> \rangle} {\langle boolean<sub>1</sub> \rangle} {\langle name<sub>2</sub> \rangle} {\langle signature<sub>2</sub> \rangle} {\langle boolean<sub>2</sub> \rangle} {\langle copying function \rangle} {\langle conditions \rangle}, \q\_recursion\_tail, \q\_recursion\_stop to a first auxiliary.

```

2649 \cs_set_protected:Npn \prg_set_eq_conditional:NNn
2650   { \__prg_set_eq_conditional:NNNn \cs_set_eq:cc }
2651 \cs_set_protected:Npn \prg_new_eq_conditional:NNn
2652   { \__prg_set_eq_conditional:NNNn \cs_new_eq:cc }
2653 \cs_set_protected:Npn \__prg_set_eq_conditional:NNNn #1#2#3#4
2654   {
2655     \use:x
2656     {
2657       \exp_not:N \__prg_set_eq_conditional:nnNnnNNw
2658       \cs_split_function:N #2
2659       \cs_split_function:N #3
2660       \exp_not:N #1
2661       \tl_to_str:n {#4}
2662       \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
2663     }
2664   }

```

(End definition for \prg\_set\_eq\_conditional:NNn, \prg\_new\_eq\_conditional:NNn, and \\_\_prg\_set\_eq\_conditional:NNNn. These functions are documented on page 96.)

\\_\_prg\_set\_eq\_conditional:nnNnnNNw \\_\_prg\_set\_eq\_conditional\_loop:mnnnnNw \\_\_prg\_set\_eq\_conditional\_p\_form:nnm \\_\_prg\_set\_eq\_conditional\_TF\_form:nnm \\_\_prg\_set\_eq\_conditional\_T\_form:nnm \\_\_prg\_set\_eq\_conditional\_F\_form:nnm 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 {\langle name<sub>1</sub> \rangle} {\langle signature<sub>1</sub> \rangle} {\langle name<sub>2</sub> \rangle} {\langle signature<sub>2</sub> \rangle} {\langle copying function \rangle} 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.

```

2665 \cs_set_protected:Npn \__prg_set_eq_conditional:nnNnnNNw #1#2#3#4#5#6
2666   {
2667     \if_meaning:w \c_false_bool #3
2668       \__kernel_msg_error:nnx { kernel } { missing-colon }
2669       { \token_to_str:c {#1} }
2670       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2671     \fi:
2672     \if_meaning:w \c_false_bool #6
2673       \__kernel_msg_error:nnx { kernel } { missing-colon }
2674       { \token_to_str:c {#4} }
2675       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2676     \fi:
2677     \__prg_set_eq_conditional_loop:mnnnnNw {#1} {#2} {#4} {#5}
2678   }
2679 \cs_set_protected:Npn \__prg_set_eq_conditional_loop:nnnnNw #1#2#3#4#5#6 ,
2680   {
2681     \if_meaning:w \q_recursion_tail #6

```

```

2682     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
2683     \fi:
2684     \use:c { __prg_set_eq_conditional_ #6 _form:wNnnnn }
2685     \tl_if_empty:nF {#6}
2686     {
2687         \__kernel_msg_error:nnxx
2688         { kernel } { conditional-form-unknown }
2689         {#6} { \token_to_str:c { #1 : #2 } }
2690     }
2691     \use_none:nnnnnn
2692     \q_stop
2693     #5 {#1} {#2} {#3} {#4}
2694     \__prg_set_eq_conditional_loop:nnnnNw {#1} {#2} {#3} {#4} #5
2695 }
2696 \__kernel_patch:nnNNpn
2697 { \__kernel_chk_cs_exist:c { #5 _p : #6 } } { }
2698 \cs_set:Npn \__prg_set_eq_conditional_p_form:wNnnnn #1 \q_stop #2#3#4#5#6
2699 { #2 { #3 _p : #4 } { #5 _p : #6 } }
2700 \__kernel_patch:nnNNpn
2701 { \__kernel_chk_cs_exist:c { #5 : #6 TF } } { }
2702 \cs_set:Npn \__prg_set_eq_conditional_TF_form:wNnnnn #1 \q_stop #2#3#4#5#6
2703 { #2 { #3 : #4 TF } { #5 : #6 TF } }
2704 \__kernel_patch:nnNNpn
2705 { \__kernel_chk_cs_exist:c { #5 : #6 T } } { }
2706 \cs_set:Npn \__prg_set_eq_conditional_T_form:wNnnnn #1 \q_stop #2#3#4#5#6
2707 { #2 { #3 : #4 T } { #5 : #6 T } }
2708 \__kernel_patch:nnNNpn
2709 { \__kernel_chk_cs_exist:c { #5 : #6 F } } { }
2710 \cs_set:Npn \__prg_set_eq_conditional_F_form:wNnnnn #1 \q_stop #2#3#4#5#6
2711 { #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**

```

2712 \tex_chardef:D \c_true_bool = 1 ~
2713 \tex_chardef:D \c_false_bool = 0 ~

```

(End definition for \c\_true\_bool and \c\_false\_bool. These variables are documented on page 20.)

## 4.8 Dissecting a control sequence

2714 ⟨@@=cs⟩

---

**\\_\_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
```

```
\__cs_to_str:N
```

```
\__cs_to_str:w
```

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 characters, 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 \ \`  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 \ \` , 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.

```
2715 \cs_set:Npn \cs_to_str:N  
2716 {
```

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.

```

2717     \tex_roman numeral:D
2718         \if:w \token_to_str:N \__cs_to_str:w \fi:
2719         \exp_after:wN \__cs_to_str:N \token_to_str:N
2720     }
2721 \cs_set:Npn \__cs_to_str:N #1 { \c_zero_int }
2722 \cs_set:Npn \__cs_to_str:w #1 \__cs_to_str:N
2723   { -\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 16.)*

### `\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 `\q_mark`, then `\c_true_bool` as #3, and #4 cleans up until `\q_stop`. Otherwise, the #1 contains the function name and `\q_mark \c_true_bool`, #2 is empty, #3 is `\c_false_bool`, and #4 cleans up. The second auxiliary trims the trailing `\q_mark` from the function name if present (that is, if the original function had no colon).

```

2724 \cs_set_protected:Npn \__cs_tmp:w #1
2725   {
2726     \cs_set:Npn \cs_split_function:N ##1
2727     {
2728       \exp_after:wN \exp_after:wN \exp_after:wN
2729       \__cs_split_function_auxi:w
2730       \cs_to_str:N ##1 \q_mark \c_true_bool
2731       ##1 \q_mark \c_false_bool \q_stop
2732     }
2733     \cs_set:Npn \__cs_split_function_auxi:w
2734     ##1 ##2 \q_mark ##3##4 \q_stop
2735     { \__cs_split_function_auxi:w ##1 \q_mark \q_stop {##2} ##3 }
2736     \cs_set:Npn \__cs_split_function_auxi:w ##1 \q_mark ##2 \q_stop
2737     { ##1 }
2738   }
2739 \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_auxi:w`. This function is documented on page 16.)*

## 4.9 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 `\cs_if_exist_p:c` 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:cTF 2740 \prg_set_conditional:Npnn \cs_if_exist:N #1 { p , T , F , TF }
2741   {
2742     \if_meaning:w #1 \scan_stop:
2743       \prg_return_false:
2744     \else:
2745       \if_cs_exist:N #1
2746         \prg_return_true:
2747       \else:
2748         \prg_return_false:
2749       \fi:
2750     \fi:
2751   }
```

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.

```

\cs_if_exist:c 2752 \prg_set_conditional:Npnn \cs_if_exist:c #1 { p , T , F , TF }
2753   {
2754     \if_cs_exist:w #1 \cs_end:
2755       \exp_after:wN \use_i:nn
2756     \else:
2757       \exp_after:wN \use_ii:nn
2758     \fi:
2759   {
2760     \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
2761       \prg_return_false:
2762     \else:
2763       \prg_return_true:
2764     \fi:
2765   }
2766   \prg_return_false:
2767 }
```

(End definition for `\cs_if_exist:NTF`. This function is documented on page 20.)

`\cs_if_free_p:N` The logical reversal of the above.

```

\cs_if_free_p:c 2768 \prg_set_conditional:Npnn \cs_if_free:N #1 { p , T , F , TF }
2769   {
2770     \if_meaning:w #1 \scan_stop:
2771       \prg_return_true:
2772     \else:
2773       \if_cs_exist:N #1
2774         \prg_return_false:
```

```

2775     \else:
2776         \prg_return_true:
2777     \fi:
2778 \fi:
2779 }
2780 \prg_set_conditional:Npnn \cs_if_free:c #1 { p , T , F , TF }
2781 {
2782     \if_cs_exist:w #1 \cs_end:
2783         \exp_after:wN \use_i:nn
2784     \else:
2785         \exp_after:wN \use_ii:nn
2786     \fi:
2787     {
2788         \exp_after:wN \if_meaning:w \cs:w #1 \cs_end: \scan_stop:
2789         \prg_return_true:
2790     \else:
2791         \prg_return_false:
2792     \fi:
2793 }
2794 { \prg_return_true: }
2795 }

```

(End definition for `\cs_if_free:NTF`. This function is documented on page 20.)

**\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 LaTeX we could use the `\lastnamedcs` primitive.

```

2796 \cs_set:Npn \cs_if_exist_use:NTF #1#2
2797   { \cs_if_exist:NTF #1 { #1 #2 } }
2798 \cs_set:Npn \cs_if_exist_use:NF #1
2799   { \cs_if_exist:NTF #1 { #1 } }
2800 \cs_set:Npn \cs_if_exist_use:NT #1 #2
2801   { \cs_if_exist:NTF #1 { #1 #2 } { } }
2802 \cs_set:Npn \cs_if_exist_use:N #1
2803   { \cs_if_exist:NTF #1 { #1 } { } }
2804 \cs_set:Npn \cs_if_exist_use:cTF #1#2
2805   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } }
2806 \cs_set:Npn \cs_if_exist_use:cF #1
2807   { \cs_if_exist:cTF {#1} { \use:c {#1} } }
2808 \cs_set:Npn \cs_if_exist_use:cT #1#2
2809   { \cs_if_exist:cTF {#1} { \use:c {#1} #2 } { } }
2810 \cs_set:Npn \cs_if_exist_use:c #1
2811   { \cs_if_exist:cTF {#1} { \use:c {#1} } { } }

```

(End definition for `\cs_if_exist_use:NTF`. This function is documented on page 15.)

## 4.10 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.

`\_kernel_msg_error:nnxx` If an internal error occurs before LATEX3 has loaded l3msg 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.

(End definition for `\_kernel\_msg_error:nxxx`, `\_kernel\_msg_error:rxxx`, `\_kernel\_msg_error:xxxx`)

Another one from l3msg which will be altered later.  
↳ [l3msg](#)

```
2830 { on~line~ \tex_the:D \tex_inputlineno:D }
```

`\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 re-  
2831 \cs_set_protected:Npn \iow_log:x  
2832   { \tex_immediate:D \tex_write:D -1 }  
2833 \cs_set_protected:Npn \iow_term:x
```

`\_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!`

```
2835 \__kernel_patch:nnNNpn { }
2836 {
2837     \__kernel_debug_log:x
2838     { Defining~\token_to_str:N #1~ \msg_line_context: }
2839 }
2840 \cs_set_protected:Npn \_kernel_chk_if_free_cs:N #1
```

```

2841      {
2842        \cs_if_free:NF #1
2843        {
2844          \__kernel_msg_error:nxxx { kernel } { command-already-defined }
2845          { \token_to_str:N #1 } { \token_to_meaning:N #1 }
2846        }
2847      }
2848 \cs_set_protected:Npn \__kernel_chk_if_free_cs:c
2849   { \exp_args:Nc \__kernel_chk_if_free_cs:N }

```

(End definition for `\__kernel_chk_if_free_cs:N`.)

## 4.11 Defining new functions

2850 `<@=cs>`

Function which check that the control sequence is free before defining it.

```

2851 \cs_set:Npn \__cs_tmp:w #1#2
2852   {
2853     \cs_set_protected:Npn #1 ##1
2854     {
2855       \__kernel_chk_if_free_cs:N ##1
2856       #2 ##1
2857     }
2858   }
2859 \__cs_tmp:w \cs_new_nopar:Npn           \cs_gset_nopar:Npn
2860 \__cs_tmp:w \cs_new_nopar:Npx          \cs_gset_nopar:Npx
2861 \__cs_tmp:w \cs_new:Npn                \cs_gset:Npn
2862 \__cs_tmp:w \cs_new:Npx                \cs_gset:Npx
2863 \__cs_tmp:w \cs_new_protected_nopar:Npn \cs_gset_protected_nopar:Npn
2864 \__cs_tmp:w \cs_new_protected_nopar:Npx \cs_gset_protected_nopar:Npx
2865 \__cs_tmp:w \cs_new_protected:Npn      \cs_gset_protected:Npn
2866 \__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 10.)

`\cs_set_nopar:cpn`  
`\cs_set_nopar:cpx`  
`\cs_gset_nopar:cpn`  
`\cs_gset_nopar:cpx`

Like `\cs_set_nopar:Npn` and `\cs_new_nopar:Npn`, 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.

`\cs_set_nopar:cpn<string><rep-text>` turns `<string>` into a csname and then assigns `<rep-text>` to it by using `\cs_set_nopar:Npn`. This means that there might be a parameter string between the two arguments.

```

2867 \cs_set:Npn \__cs_tmp:w #1#2
2868   { \cs_new_protected_nopar:Npn #1 { \exp_args:Nc #2 } }
2869 \__cs_tmp:w \cs_set_nopar:cpn \cs_set_nopar:Npn
2870 \__cs_tmp:w \cs_set_nopar:cpx \cs_set_nopar:Npx
2871 \__cs_tmp:w \cs_gset_nopar:cpn \cs_gset_nopar:Npn
2872 \__cs_tmp:w \cs_gset_nopar:cpx \cs_gset_nopar:Npx
2873 \__cs_tmp:w \cs_new_nopar:cpn \cs_new_nopar:Npn
2874 \__cs_tmp:w \cs_new_nopar:cpx \cs_new_nopar:Npx

```

(End definition for `\cs_set_nopar:Npn`. This function is documented on page 10.)

```

\cs_set:cpn Variants of the \cs_set:Npn versions which make a csname out of the first arguments.
\cs_set:cpx We may also do this globally.

2875 \__cs_tmp:w \cs_set:cpn \cs_set:Npn
2876 \__cs_tmp:w \cs_set:cpx \cs_set:Npx
2877 \__cs_tmp:w \cs_gset:cpn \cs_gset:Npn
2878 \__cs_tmp:w \cs_gset:cpx \cs_gset:Npx
2879 \__cs_tmp:w \cs_new:cpn \cs_new:Npn
2880 \__cs_tmp:w \cs_new:cpx \cs_new:Npx

```

(End definition for \cs\_set:Npn. This function is documented on page 10.)

\cs\_set\_protected\_nopar:cpn Variants of the \cs\_set\_protected\_nopar:Npn versions which make a csname out of the first arguments. We may also do this globally.

```

2881 \__cs_tmp:w \cs_set_protected_nopar:cpn \cs_set_protected_nopar:Npn
2882 \__cs_tmp:w \cs_set_protected_nopar:cpx \cs_set_protected_nopar:Npx
2883 \__cs_tmp:w \cs_gset_protected_nopar:cpn \cs_gset_protected_nopar:Npn
2884 \__cs_tmp:w \cs_gset_protected_nopar:cpx \cs_gset_protected_nopar:Npx
2885 \__cs_tmp:w \cs_new_protected_nopar:cpn \cs_new_protected_nopar:Npn
2886 \__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 11.)

\cs\_set\_protected:cpn Variants of the \cs\_set\_protected:Npn versions which make a csname out of the first arguments. We may also do this globally.

```

2887 \__cs_tmp:w \cs_set_protected:cpn \cs_set_protected:Npn
2888 \__cs_tmp:w \cs_set_protected:cpx \cs_set_protected:Npx
2889 \__cs_tmp:w \cs_gset_protected:cpn \cs_gset_protected:Npn
2890 \__cs_tmp:w \cs_gset_protected:cpx \cs_gset_protected:Npx
2891 \__cs_tmp:w \cs_new_protected:cpn \cs_new_protected:Npn
2892 \__cs_tmp:w \cs_new_protected:cpx \cs_new_protected:Npx

```

(End definition for \cs\_set\_protected:Npn. This function is documented on page 10.)

## 4.12 Copying definitions

**\cs\_set\_eq:NN** 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”.

```

2893 \cs_new_protected:Npn \cs_set_eq:NN #1 { \tex_let:D #1 =~ }
2894 \cs_new_protected:Npn \cs_set_eq:cN { \exp_args:Nc \cs_set_eq:NN }
2895 \cs_new_protected:Npn \cs_set_eq:Nc { \exp_args:NNc \cs_set_eq:NN }
2896 \cs_new_protected:Npn \cs_set_eq:cc { \exp_args:Ncc \cs_set_eq:NN }
2897 \cs_new_protected:Npn \cs_gset_eq:NN { \tex_global:D \cs_set_eq:NN }
2898 \cs_new_protected:Npn \cs_gset_eq:Nc { \exp_args:NNc \cs_gset_eq:NN }
2899 \cs_new_protected:Npn \cs_gset_eq:cN { \exp_args:Nc \cs_gset_eq:NN }
2900 \cs_new_protected:Npn \cs_gset_eq:cc { \exp_args:Ncc \cs_gset_eq:NN }
2901 \cs_new_protected:Npn \cs_new_eq:NN #1
2902 {
2903     \__kernel_chk_if_free_cs:N #1

```

```

2904     \tex_global:D \cs_set_eq:NN #1
2905   }
2906 \cs_new_protected:Npn \cs_new_eq:cN { \exp_args:Nc \cs_new_eq:NN }
2907 \cs_new_protected:Npn \cs_new_eq:Nc { \exp_args:NNc \cs_new_eq:NN }
2908 \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 14.)

### 4.13 Undefining functions

`\cs_undefine:N` 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.

```

2909 \cs_new_protected:Npn \cs_undefine:N #1
2910   { \cs_gset_eq:NN #1 \tex_undefined:D }
2911 \cs_new_protected:Npn \cs_undefine:c #1
2912   {
2913     \if_cs_exist:w #1 \cs_end:
2914       \exp_after:wN \use:n
2915     \else:
2916       \exp_after:wN \use_none:n
2917     \fi:
2918   { \cs_gset_eq:cN {#1} \tex_undefined:D }
2919 }

```

(End definition for `\cs_undefine:N`. This function is documented on page 14.)

### 4.14 Generating parameter text from argument count

2920  $\langle @@=cs \rangle$

`\_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.

```

2921 \cs_set_protected:Npn \_kernel_cs_parm_from_arg_count:nnF #1#2
2922   {
2923     \exp_args:Nx \_cs_parm_from_arg_count_test:nnF
2924     {
2925       \exp_after:wN \exp_not:n
2926       \if_case:w \int_eval:n {#2}
2927         {}
2928         \or: {##1}
2929         \or: {##1##2}
2930         \or: {##1##2##3}
2931         \or: {##1##2##3##4}
2932         \or: {##1##2##3##4##5}
2933         \or: {##1##2##3##4##5##6}

```

```

2934         \or: { ##1##2##3##4##5##6##7 }
2935         \or: { ##1##2##3##4##5##6##7##8 }
2936         \or: { ##1##2##3##4##5##6##7##8##9 }
2937         \else: { \c_false_bool }
2938         \fi:
2939     }
2940     {#1}
2941   }
2942 \cs_set_protected:Npn \__cs_parm_from_arg_count_test:nnF #1#2
2943   {
2944     \if_meaning:w \c_false_bool #1
2945       \exp_after:wN \use_i:nn
2946     \else:
2947       \exp_after:wN \use_i:nn
2948     \fi:
2949     { #2 {#1} }
2950   }

```

(End definition for `\__kernel_cs_parm_from_arg_count:nnF` and `\__cs_parm_from_arg_count_test:nnF`.)

## 4.15 Defining functions from a given number of arguments

2951 ⟨@@=cs⟩

`\__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.

```

2952 \cs_new:Npn \__cs_count_signature:N #1
2953   { \exp_args:Nf \__cs_count_signature:n { \cs_split_function:N #1 } }
2954 \cs_new:Npn \__cs_count_signature:n #1
2955   { \int_eval:n { \__cs_count_signature:nnN #1 } }
2956 \cs_new:Npn \__cs_count_signature:nnN #1#2#3
2957   {
2958     \if_meaning:w \c_true_bool #3
2959       \tl_count:n {#2}
2960     \else:
2961       -1
2962     \fi:
2963   }
2964 \cs_new:Npn \__cs_count_signature:c
2965   { \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

```

2966 \cs_new_protected:Npn \cs_generate_from_arg_count>NNnn #1#2#3#4
2967   {

```

```

2968     \__kernel_cs_parm_from_arg_count:nnF { \use:nnn #2 #1 } {#3}
2969     {
2970         \__kernel_msg_error:nnxx { kernel } { bad-number-of-arguments }
2971         { \token_to_str:N #1 } { \int_eval:n {#3} }
2972         \use_none:n
2973     }
2974     {#4}
2975 }
```

A variant form we need right away, plus one which is used elsewhere but which is most logically created here.

```

2976 \cs_new_protected:Npn \cs_generate_from_arg_count:cNnn
2977   { \exp_args:Nc \cs_generate_from_arg_count:NNnn }
2978 \cs_new_protected:Npn \cs_generate_from_arg_count:Ncnn
2979   { \exp_args:NNc \cs_generate_from_arg_count:NNnn }
```

(End definition for `\cs_generate_from_arg_count:NNnn`. This function is documented on page 13.)

## 4.16 Using the signature to define functions

```
2980 <@@=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.

```

2981 \cs_set:Npn \__cs_tmp:w #1#2#3
2982   {
2983     \cs_new_protected:cp { cs_ #1 : #2 }
2984     {
2985       \exp_not:N \__cs_generate_from_signature:NNn
2986       \exp_after:wN \exp_not:N \cs:w cs_ #1 : #3 \cs_end:
2987     }
2988   }
2989 \cs_new_protected:Npn \__cs_generate_from_signature:NNn #1#2
2990   {
2991     \use:x
2992     {
2993       \__cs_generate_from_signature:nnNNNn
2994       \cs_split_function:N #2
2995     }
2996     #1 #2
2997   }
2998 \cs_new_protected:Npn \__cs_generate_from_signature:nnNNNn #1#2#3#4#5#6
```

```

2999  {
3000      \bool_if:NTF #3
3001      {
3002          \str_if_eq:eeF { }
3003          { \tl_map_function:nN {#2} \__cs_generate_from_signature:n }
3004          {
3005              \__kernel_msg_error:nnx { kernel } { non-base-function }
3006              { \token_to_str:N #5 }
3007          }
3008          \cs_generate_from_arg_count>NNnn
3009          #5 #4 { \tl_count:n {#2} } {#6}
3010      }
3011      {
3012          \__kernel_msg_error:nnx { kernel } { missing-colon }
3013          { \token_to_str:N #5 }
3014      }
3015  }
3016 \cs_new:Npn \__cs_generate_from_signature:n #1
3017  {
3018      \if:w n #1 \else: \if:w N #1 \else:
3019      \if:w T #1 \else: \if:w F #1 \else: #1 \fi: \fi: \fi: \fi:
3020  }

```

Then we define the 24 variants beginning with N.

```

3021 \__cs_tmp:w { set } { Nn } { Npn }
3022 \__cs_tmp:w { set } { Nx } { Npx }
3023 \__cs_tmp:w { set_nopar } { Nn } { Npn }
3024 \__cs_tmp:w { set_nopar } { Nx } { Npx }
3025 \__cs_tmp:w { set_protected } { Nn } { Npn }
3026 \__cs_tmp:w { set_protected } { Nx } { Npx }
3027 \__cs_tmp:w { set_protected_nopar } { Nn } { Npn }
3028 \__cs_tmp:w { set_protected_nopar } { Nx } { Npx }
3029 \__cs_tmp:w { gset } { Nn } { Npn }
3030 \__cs_tmp:w { gset } { Nx } { Npx }
3031 \__cs_tmp:w { gset_nopar } { Nn } { Npn }
3032 \__cs_tmp:w { gset_nopar } { Nx } { Npx }
3033 \__cs_tmp:w { gset_protected } { Nn } { Npn }
3034 \__cs_tmp:w { gset_protected } { Nx } { Npx }
3035 \__cs_tmp:w { gset_protected_nopar } { Nn } { Npn }
3036 \__cs_tmp:w { gset_protected_nopar } { Nx } { Npx }
3037 \__cs_tmp:w { new } { Nn } { Npn }
3038 \__cs_tmp:w { new } { Nx } { Npx }
3039 \__cs_tmp:w { new_nopar } { Nn } { Npn }
3040 \__cs_tmp:w { new_nopar } { Nx } { Npx }
3041 \__cs_tmp:w { new_protected } { Nn } { Npn }
3042 \__cs_tmp:w { new_protected } { Nx } { Npx }
3043 \__cs_tmp:w { new_protected_nopar } { Nn } { Npn }
3044 \__cs_tmp:w { new_protected_nopar } { Nx } { Npx }

```

(End definition for `\cs_set:Nn` and others. These functions are documented on page 12.)

`\cs_set:cn` The 24 c variants simply use `\exp_args:Nc`.

`\cs_set:cx` `\cs_set:Npn \__cs_tmp:w #1#2`

```

\cs_set_nopar:cn
\cs_set_nopar:cx
3046  {
3047      \cs_new_protected:cp { cs_ #1 : c #2 }

```

`\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`

```

3048   {
3049     \exp_not:N \exp_args:Nc
3050     \exp_after:wN \exp_not:N \cs:w cs_ #1 : N #2 \cs_end:
3051   }
3052 }
3053 \__cs_tmp:w { set } { n }
3054 \__cs_tmp:w { set } { x }
3055 \__cs_tmp:w { set_nopar } { n }
3056 \__cs_tmp:w { set_nopar } { x }
3057 \__cs_tmp:w { set_protected } { n }
3058 \__cs_tmp:w { set_protected } { x }
3059 \__cs_tmp:w { set_protected_nopar } { n }
3060 \__cs_tmp:w { set_protected_nopar } { x }
3061 \__cs_tmp:w { gset } { n }
3062 \__cs_tmp:w { gset } { x }
3063 \__cs_tmp:w { gset_nopar } { n }
3064 \__cs_tmp:w { gset_nopar } { x }
3065 \__cs_tmp:w { gset_protected } { n }
3066 \__cs_tmp:w { gset_protected } { x }
3067 \__cs_tmp:w { gset_protected_nopar } { n }
3068 \__cs_tmp:w { gset_protected_nopar } { x }
3069 \__cs_tmp:w { new } { n }
3070 \__cs_tmp:w { new } { x }
3071 \__cs_tmp:w { new_nopar } { n }
3072 \__cs_tmp:w { new_nopar } { x }
3073 \__cs_tmp:w { new_protected } { n }
3074 \__cs_tmp:w { new_protected } { x }
3075 \__cs_tmp:w { new_protected_nopar } { n }
3076 \__cs_tmp:w { new_protected_nopar } { x }

```

(End definition for `\cs_set:Nn`. This function is documented on page 12.)

## 4.17 Checking control sequence equality

**\cs\_if\_eq\_p:NN** Check if two control sequences are identical.

```

\cs_if_eq_p:cN 3077 \prg_new_conditional:Npnn \cs_if_eq:NN #1#2 { p , T , F , TF }
\cs_if_eq_p:Nc 3078 {
\cs_if_eq_p:cc 3079   \if_meaning:w #1#2
\cs_if_eq:NNTF 3080     \prg_return_true: \else: \prg_return_false: \fi:
\cs_if_eq:cNTF 3081 }
\cs_if_eq:NcTF 3082 \cs_new:Npn \cs_if_eq_p:cN { \exp_args:Nc \cs_if_eq_p:NN }
\cs_if_eq:cNTF 3083 \cs_new:Npn \cs_if_eq:cNTF { \exp_args:Nc \cs_if_eq:NNTF }
\cs_if_eq:cCTF 3084 \cs_new:Npn \cs_if_eq:cNT { \exp_args:Nc \cs_if_eq:NNT }
3085 \cs_new:Npn \cs_if_eq:cNF { \exp_args:Nc \cs_if_eq:NNF }
3086 \cs_new:Npn \cs_if_eq_p:Nc { \exp_args:Nnc \cs_if_eq_p:NN }
3087 \cs_new:Npn \cs_if_eq:NcTF { \exp_args:Nnc \cs_if_eq:NNTF }
3088 \cs_new:Npn \cs_if_eq:NcT { \exp_args:Nnc \cs_if_eq:NNT }
3089 \cs_new:Npn \cs_if_eq:NcF { \exp_args:Nnc \cs_if_eq:NNF }
3090 \cs_new:Npn \cs_if_eq_p:cc { \exp_args:Ncc \cs_if_eq_p:NN }
3091 \cs_new:Npn \cs_if_eq:ccTF { \exp_args:Ncc \cs_if_eq:NNTF }
3092 \cs_new:Npn \cs_if_eq:ccT { \exp_args:Ncc \cs_if_eq:NNT }
3093 \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 20.)

## 4.18 Diagnostic functions

3094 ⟨@@=kernel⟩

\\_\_kernel\_chk\_defined:NT Error if the variable #1 is not defined.

```
3095 \cs_new_protected:Npn \__kernel_chk_defined:NT #1#2
3096 {
3097     \cs_if_exist:NTF #1
3098         {#2}
3099     {
3100         \__kernel_msg_error:n { kernel } { variable-not-defined }
3101         { \token_to_str:N #1 }
3102     }
3103 }
```

(End definition for \\_\_kernel\_chk\_defined:NT.)

\\_\_kernel\_register\_show:N 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.

\\_\_kernel\_register\_show:c  
\\_\_kernel\_register\_log:N  
\\_\_kernel\_register\_log:c  
\\_\_kernel\_register\_show\_aux:NN

```
3104 \cs_new_protected:Npn \__kernel_register_show:N
3105     { \__kernel_register_show_aux:NN \tl_show:n }
3106 \cs_new_protected:Npn \__kernel_register_show:c
3107     { \exp_args:Nc \__kernel_register_show:N }
3108 \cs_new_protected:Npn \__kernel_register_log:N
3109     { \__kernel_register_show_aux:NN \tl_log:n }
3110 \cs_new_protected:Npn \__kernel_register_log:c
3111     { \exp_args:Nc \__kernel_register_log:N }
3112 \cs_new_protected:Npn \__kernel_register_show_aux:NN #1#2
3113 {
3114     \__kernel_chk_defined:NT #2
3115     {
3116         \exp_args:No \__kernel_register_show_aux:nNN
3117         { \tex_the:D #2 } #2 #1
3118     }
3119 }
3120 \cs_new_protected:Npn \__kernel_register_show_aux:nNN #1#2#3
3121     { \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 TeX'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.

\\_\_kernel\_show>NN

```
3122 \cs_new_protected:Npn \cs_show:N { \__kernel_show>NN \tl_show:n }
3123 \cs_new_protected:Npn \cs_show:c
```

```

3124   { \group_begin: \exp_args:NNc \group_end: \cs_show:N }
3125   \cs_new_protected:Npn \cs_log:N { \_kernel_show:NN \tl_log:n }
3126   \cs_new_protected:Npn \cs_log:c
3127   { \group_begin: \exp_args:NNc \group_end: \cs_log:N }
3128   \cs_new_protected:Npn \_kernel_show:NN #1#2
3129   {
3130     \group_begin:
3131     \int_set:Nn \tex_escapechar:D { '\\" }
3132     \exp_args:NNx
3133     \group_end:
3134     #1 { \token_to_str:N #2 = \cs_meaning:N #2 }
3135   }

```

(End definition for `\cs_show:N`, `\cs_log:N`, and `\_kernel_show:NN`. These functions are documented on page 15.)

## 4.19 Doing nothing functions

`\prg_do_nothing:` This does not fit anywhere else!

```
3136 \cs_new:Npn \prg_do_nothing: { }
```

(End definition for `\prg_do_nothing:`. This function is documented on page 8.)

## 4.20 Breaking out of mapping functions

```
3137 ⟨@=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.

```

3138 \cs_new_eq:NN \prg_break_point:Nn \use_ii:nn
3139 \cs_new:Npn \prg_map_break:Nn #1#2#3 \prg_break_point:Nn #4#5
3140   {
3141     #5
3142     \if_meaning:w #1 #4
3143     \exp_after:wN \use_iii:nnn
3144     \fi:
3145     \prg_map_break:Nn #1 {#2}
3146   }

```

(End definition for `\prg_break_point:Nn` and `\prg_map_break:Nn`. These functions are documented on page 103.)

`\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.

```

3147 \cs_new_eq:NN \prg_break_point: \prg_do_nothing:
3148 \cs_new:Npn \prg_break: #1 \prg_break_point: { }
3149 \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 103.)

```
3150 ⟨/initex | package⟩
```

## 5 l3expan implementation

```
3151  {*initex | package}
3152  <@@=exp>
```

\l\_\_exp\_internal\_tl 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\_tl.)

\exp\_after:wN These are defined in l3basics, 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 30.)

### 5.1 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 5.3. In section 5.2 some common cases are coded by a more direct method for efficiency, typically using calls to \exp\_after:wN.

\l\_\_exp\_internal\_tl This scratch token list variable is defined in l3basics.

(End definition for \l\_\_exp\_internal\_tl.)

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.

```
3153 \cs_new:Npn \_exp_arg_next:nnn #1#2#3 { #2 \::: { #3 {#1} } }
```

```
3154 \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.

```
3155 \cs_new:Npn \::: {#1}
```

(End definition for \:::. This function is documented on page 34.)

**\:::n** This function is used to skip an argument that doesn't need to be expanded.

```
3156 \cs_new:Npn \:::n #1 \::: #2#3 { #1 \::: { #2 {#3} } }
```

(*End definition for \:::n. This function is documented on page 34.*)

**\:::N** This function is used to skip an argument that consists of a single token and doesn't need to be expanded.

```
3157 \cs_new:Npn \:::N #1 \::: #2#3 { #1 \::: {#2#3} }
```

(*End definition for \:::N. This function is documented on page 34.*)

**\:::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.

```
3158 \cs_new:Npn \:::p #1 \::: #2#3# { #1 \::: {#2#3} }
```

(*End definition for \:::p. This function is documented on page 34.*)

**\:::c** This function is used to skip an argument that is turned into a control sequence without expansion.

```
3159 \cs_new:Npn \:::c #1 \::: #2#3
3160   { \exp_after:wN \__exp_arg_next:Nnn \cs:w #3 \cs_end: {#1} {#2} }
```

(*End definition for \:::c. This function is documented on page 34.*)

**\:::o** This function is used to expand an argument once.

```
3161 \cs_new:Npn \:::o #1 \::: #2#3
3162   { \exp_after:wN \__exp_arg_next:nnn \exp_after:wN {#3} {#1} {#2} }
```

(*End definition for \:::o. This function is documented on page 34.*)

**\:::e** With the `\expanded` primitive available, just expand. Otherwise defer to `\exp_args:N` implemented later.

```
3163 \cs_if_exist:NTF \tex_expanded:D
3164   {
3165     \cs_new:Npn \:::e #1 \::: #2#3
3166       { \tex_expanded:D { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } } }
3167   }
3168   {
3169     \cs_new:Npn \:::e #1 \::: #2#3
3170       { \exp_args:N \__exp_arg_next:nnn {#3} {#1} {#2} }
3171   }
```

(*End definition for \:::e. This function is documented on page 34.*)

**\:::f** This function is used to expand a token list until the first unexpandable token is found. **\exp\_stop\_f:** 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.

```

3172 \cs_new:Npn \:::f #1 \::: #2#3
3173 {
3174     \exp_after:wN \__exp_arg_next:nnn
3175         \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
3176     {#1} {#2}
3177 }
3178 \use:nn { \cs_new_eq:NN \exp_stop:f: } { ~ }

```

(End definition for `\:::f` and `\exp_stop:f:`. These functions are documented on page 34.)

`\:::x` This function is used to expand an argument fully. We build in the expansion of `\__-exp_arg_next:nnn`.

```

3179 \cs_new_protected:Npn \:::x #1 \::: #2#3
3180 {
3181     \cs_set_nopar:Npx \l__exp_internal_tl
3182         { \exp_not:n { #1 \::: } { \exp_not:n {#2} {#3} } }
3183     \l__exp_internal_tl
3184 }

```

(End definition for `\:::x`. This function is documented on page 34.)

`\:::v` These functions return the value of a register, i.e., one of `tl`, `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.

```

3185 \cs_new:Npn \:::V #1 \::: #2#3
3186 {
3187     \exp_after:wN \__exp_arg_next:nnn
3188         \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
3189     {#1} {#2}
3190 }
3191 \cs_new:Npn \:::v # 1\::: #2#3
3192 {
3193     \exp_after:wN \__exp_arg_next:nnn
3194         \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
3195     {#1} {#2}
3196 }

```

(End definition for `\:::v` and `\:::V`. These functions are documented on page 34.)

`\__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:`.

```

3197 \cs_new:Npn \__exp_eval_register:N #1
3198 {
3199     \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.

```
3200      \if_meaning:w \scan_stop: #1
3201          \__exp_eval_error_msg:w
3202      \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`.

```
3203      \else:
3204          \exp_after:wN \use_i_i:nnn
3205      \fi:
3206      \exp_after:wN \exp_end: \tex_the:D #1
3207  }
3208 \cs_new:Npn \__exp_eval_register:c #1
3209 { \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}

3210 \cs_new:Npn \__exp_eval_error_msg:w #1 \tex_the:D #2
3211 {
3212     \fi:
3213     \fi:
3214     \__kernel_msg_expandable_error:nnn { kernel } { bad-variable } {#2}
3215     \exp_end:
3216 }
```

(End definition for `\__exp_eval_register:N` and `\__exp_eval_error_msg:w`.)

## 5.2 Hand-tuned definitions

One of the most important features of these functions is that they are fully expandable.

`\exp_args:Nc` In `l3basics`.

`\exp_args:cc`

(End definition for `\exp_args:Nc` and `\exp_args:cc`. These functions are documented on page 26.)

```

\exp_args:Nc
\exp_args:Ncc
\exp_args:Nccc
3217 \cs_new:Npn \exp_args:Nc #1#2#3
3218   { \exp_after:wN #1 \exp_after:wN #2 \cs:w # 3\cs_end: }
3219 \cs_new:Npn \exp_args:Ncc #1#2#3
3220   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: \cs:w #3 \cs_end: }
3221 \cs_new:Npn \exp_args:Nccc #1#2#3#4
3222   {
3223     \exp_after:wN #1
3224       \cs:w #2 \exp_after:wN \cs_end:
3225       \cs:w #3 \exp_after:wN \cs_end:
3226       \cs:w #4 \cs_end:
3227   }

```

(End definition for `\exp_args:Nc`, `\exp_args:Ncc`, and `\exp_args:Nccc`. These functions are documented on page 28.)

```

\exp_args:No
\exp_args:NNo
\exp_args:NNNo
3228 \cs_new:Npn \exp_args:No #1#2 { \exp_after:wN #1 \exp_after:wN {#2} }
3229 \cs_new:Npn \exp_args:NNo #1#2#3
3230   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN {#3} }
3231 \cs_new:Npn \exp_args:NNNo #1#2#3#4
3232   { \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 27.)

`\exp_args:Ne` When the `\expanded` primitive is available, use it. Otherwise use `\_exp_e:nn`, defined later, to fully expand tokens.

```

3233 \cs_if_exist:NTF \tex_expanded:D
3234   {
3235     \cs_new:Npn \exp_args:Ne #1#2
3236       { \exp_after:wN #1 \tex_expanded:D { {#2} } }
3237   }
3238   {
3239     \cs_new:Npn \exp_args:Ne #1#2
3240       {
3241         \exp_after:wN #1 \exp_after:wN
3242           { \exp:w \_exp_e:nn {#2} { } }
3243       }
3244   }

```

(End definition for `\exp_args:Ne`. This function is documented on page 27.)

```

\exp_args:Nf
\exp_args:NV
\exp_args:Nv
3245 \cs_new:Npn \exp_args:Nf #1#2
3246   { \exp_after:wN #1 \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } }
3247 \cs_new:Npn \exp_args:Nv #1#2
3248   {
3249     \exp_after:wN #1 \exp_after:wN
3250       { \exp:w \_exp_eval_register:c {#2} }
3251   }
3252 \cs_new:Npn \exp_args:NV #1#2
3253   {
3254     \exp_after:wN #1 \exp_after:wN
3255       { \exp:w \_exp_eval_register:N #2 }
3256   }

```

(End definition for `\exp_args:Nf`, `\exp_args:NV`, and `\exp_args:Nv`. These functions are documented on page 27.)

**`\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.

```

3257 \cs_new:Npn \exp_args:NNV #1#2#3
3258 {
3259     \exp_after:wN #1
3260     \exp_after:wN #2
3261     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
3262 }
3263 \cs_new:Npn \exp_args:NNv #1#2#3
3264 {
3265     \exp_after:wN #1
3266     \exp_after:wN #2
3267     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
3268 }
3269 \cs_if_exist:NTF \tex_expanded:D
3270 {
3271     \cs_new:Npn \exp_args:NNe #1#2#3
3272 {
3273     \exp_after:wN #1
3274     \exp_after:wN #2
3275     \tex_expanded:D { {#3} }
3276 }
3277 }
3278 { \cs_new:Npn \exp_args:NNe { \:::N \:::e \::: } }
3279 \cs_new:Npn \exp_args:NNf #1#2#3
3280 {
3281     \exp_after:wN #1
3282     \exp_after:wN #2
3283     \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
3284 }
3285 \cs_new:Npn \exp_args:Nco #1#2#3
3286 {
3287     \exp_after:wN #1
3288     \cs:w #2 \exp_after:wN \cs_end:
3289     \exp_after:wN {#3}
3290 }
3291 \cs_new:Npn \exp_args:NcV #1#2#3
3292 {
3293     \exp_after:wN #1
3294     \cs:w #2 \exp_after:wN \cs_end:
3295     \exp_after:wN { \exp:w \__exp_eval_register:N #3 }
3296 }
3297 \cs_new:Npn \exp_args:Ncv #1#2#3
3298 {
3299     \exp_after:wN #1
3300     \cs:w #2 \exp_after:wN \cs_end:
3301     \exp_after:wN { \exp:w \__exp_eval_register:c {#3} }
3302 }
3303 \cs_new:Npn \exp_args:Ncf #1#2#3
3304 {

```

```

3305      \exp_after:wN #1
3306      \cs:w #2 \exp_after:wN \cs_end:
3307      \exp_after:wN { \exp:w \exp_end_continue_f:w #3 }
3308  }
3309 \cs_new:Npn \exp_args:NVV #1#2#3
3310  {
3311      \exp_after:wN #1
3312      \exp_after:wN { \exp:w \exp_after:wN
3313          \_exp_eval_register:N \exp_after:wN #2 \exp_after:wN }
3314      \exp_after:wN { \exp:w \_exp_eval_register:N #3 }
3315  }

```

(End definition for `\exp_args:NNV` and others. These functions are documented on page 28.)

`\exp_args:NNNV` A few more that we can hand-tune.

```

\exp_args:NcNc 3316 \cs_new:Npn \exp_args:NNNV #1#2#3#4
\exp_args:NcNo 3317  {
\exp_args:Ncco 3318      \exp_after:wN #1
3319      \exp_after:wN #2
3320      \exp_after:wN #3
3321      \exp_after:wN { \exp:w \_exp_eval_register:N #4 }
3322  }
3323 \cs_new:Npn \exp_args:NcNc #1#2#3#4
3324  {
3325      \exp_after:wN #1
3326      \cs:w #2 \exp_after:wN \cs_end:
3327      \exp_after:wN #3
3328      \cs:w #4 \cs_end:
3329  }
3330 \cs_new:Npn \exp_args:NcNo #1#2#3#4
3331  {
3332      \exp_after:wN #1
3333      \cs:w #2 \exp_after:wN \cs_end:
3334      \exp_after:wN #3
3335      \exp_after:wN {#4}
3336  }
3337 \cs_new:Npn \exp_args:Ncco #1#2#3#4
3338  {
3339      \exp_after:wN #1
3340      \cs:w #2 \exp_after:wN \cs_end:
3341      \cs:w #3 \exp_after:wN \cs_end:
3342      \exp_after:wN {#4}
3343  }

```

(End definition for `\exp_args:NNNV` and others. These functions are documented on page 29.)

### 5.3 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:Nx
3344 \cs_new_protected:Npn \exp_args:Nx { \:::x \::: }

```

(End definition for `\exp_args:Nx`. This function is documented on page 28.)

\exp\_args:Nnc Here are the actual function definitions, using the helper functions above.

```
3345 \cs_new:Npn \exp_args:Nnc { \::n \::c \::: }  
3346 \cs_new:Npn \exp_args:Nno { \::n \::o \::: }  
3347 \cs_new:Npn \exp_args:NnV { \::n \::V \::: }  
3348 \cs_new:Npn \exp_args:Nnv { \::n \::v \::: }  
3349 \cs_new:Npn \exp_args:Nne { \::n \::e \::: }  
3350 \cs_new:Npn \exp_args:Nnf { \::n \::f \::: }  
3351 \cs_new:Npn \exp_args:Noc { \::o \::c \::: }  
3352 \cs_new:Npn \exp_args:Noo { \::o \::o \::: }  
3353 \cs_new:Npn \exp_args:Nof { \::o \::f \::: }  
3354 \cs_new:Npn \exp_args:NVo { \::V \::o \::: }  
3355 \cs_new:Npn \exp_args:Nfo { \::f \::o \::: }  
3356 \cs_new:Npn \exp_args:Nff { \::f \::f \::: }  
3357 \cs_new_protected:Npn \exp_args>NNx { \::N \::x \::: }  
3358 \cs_new_protected:Npn \exp_args:Ncx { \::c \::x \::: }  
3359 \cs_new_protected:Npn \exp_args:Nnx { \::n \::x \::: }  
3360 \cs_new_protected:Npn \exp_args:Nox { \::o \::x \::: }  
3361 \cs_new_protected:Npn \exp_args:Nxo { \::x \::o \::: }  
3362 \cs_new_protected:Npn \exp_args:Nxx { \::x \::x \::: }
```

(End definition for \exp\_args:Nnc and others. These functions are documented on page 28.)

\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:NNNx  
\exp\_args:NNnx  
\exp\_args:NNox  
\exp\_args:Nccx  
\exp\_args:Ncnx  
\exp\_args:Nnnx  
\exp\_args:Nnox  
\exp\_args:Noox

```
3363 \cs_new:Npn \exp_args:NNcf { \::N \::c \::f \::: }  
3364 \cs_new:Npn \exp_args:NNno { \::N \::n \::o \::: }  
3365 \cs_new:Npn \exp_args:NNnV { \::N \::n \::V \::: }  
3366 \cs_new:Npn \exp_args:NNoo { \::N \::o \::o \::: }  
3367 \cs_new:Npn \exp_args:NNVV { \::N \::V \::V \::: }  
3368 \cs_new:Npn \exp_args:Ncno { \::c \::n \::o \::: }  
3369 \cs_new:Npn \exp_args:NcnV { \::c \::n \::V \::: }  
3370 \cs_new:Npn \exp_args:Ncoo { \::c \::o \::o \::: }  
3371 \cs_new:Npn \exp_args:NcVV { \::c \::V \::V \::: }  
3372 \cs_new:Npn \exp_args:Nnnc { \::n \::n \::c \::: }  
3373 \cs_new:Npn \exp_args:Nnno { \::n \::n \::o \::: }  
3374 \cs_new:Npn \exp_args:Nnnf { \::n \::n \::f \::: }  
3375 \cs_new:Npn \exp_args:Nfff { \::n \::f \::f \::: }  
3376 \cs_new:Npn \exp_args:Nooo { \::o \::o \::o \::: }  
3377 \cs_new:Npn \exp_args:Noof { \::o \::o \::f \::: }  
3378 \cs_new:Npn \exp_args:Nffo { \::f \::f \::o \::: }  
3379 \cs_new_protected:Npn \exp_args>NNNx { \::N \::N \::x \::: }  
3380 \cs_new_protected:Npn \exp_args:NNnx { \::N \::n \::x \::: }  
3381 \cs_new_protected:Npn \exp_args:NNox { \::N \::o \::x \::: }  
3382 \cs_new_protected:Npn \exp_args:Nnnx { \::n \::n \::x \::: }  
3383 \cs_new_protected:Npn \exp_args:Nnox { \::n \::o \::x \::: }  
3384 \cs_new_protected:Npn \exp_args:Nccx { \::c \::c \::x \::: }  
3385 \cs_new_protected:Npn \exp_args:Ncnx { \::c \::n \::x \::: }  
3386 \cs_new_protected:Npn \exp_args:Noox { \::o \::o \::x \::: }
```

(End definition for \exp\_args:NNcf and others. These functions are documented on page 29.)

## 5.4 Last-unbraced versions

There are a few places where the last argument needs to be available unbraced. First some helper macros.

```
__exp_arg_last_unbraced:nn  
  \::o_unbraced  
  \::V_unbraced  
  \::v_unbraced  
  \::e_unbraced  
  \::f_unbraced  
  \::x_unbraced
```

```

3387 \cs_new:Npn \__exp_arg_last_unbraced:nn #1#2 { #2#1 }
3388 \cs_new:Npn \::o_unbraced \::: #1#2
3389   { \exp_after:wN \__exp_arg_last_unbraced:nn \exp_after:wN {#2} {#1} }
3390 \cs_new:Npn \::V_unbraced \::: #1#2
3391   {
3392     \exp_after:wN \__exp_arg_last_unbraced:nn
3393       \exp_after:wN { \exp:w \__exp_eval_register:N #2 } {#1}
3394   }
3395 \cs_new:Npn \::v_unbraced \::: #1#2
3396   {
3397     \exp_after:wN \__exp_arg_last_unbraced:nn
3398       \exp_after:wN { \exp:w \__exp_eval_register:c {#2} } {#1}
3399   }
3400 \cs_if_exist:NTF \tex_expanded:D
3401   {
3402     \cs_new:Npn \::e_unbraced \::: #1#2
3403       { \tex_expanded:D { \exp_not:n {#1} #2 } }
3404   }
3405   {
3406     \cs_new:Npn \::e_unbraced \::: #1#2
3407       { \exp:w \__exp_e:nn {#2} {#1} }
3408   }
3409 \cs_new:Npn \::f_unbraced \::: #1#2
3410   {
3411     \exp_after:wN \__exp_arg_last_unbraced:nn
3412       \exp_after:wN { \exp:w \exp_end_continue_f:w #2 } {#1}
3413   }
3414 \cs_new_protected:Npn \::x_unbraced \::: #1#2
3415   {
3416     \cs_set_nopar:Npx \l__exp_internal_tl { \exp_not:n {#1} #2 }
3417     \l__exp_internal_tl
3418   }

```

(End definition for `\__exp_arg_last_unbraced:nn` and others. These functions are documented on page 34.)

`\exp_last_unbraced:No`  
`\exp_last_unbraced:NV`

`\exp_last_unbraced:Nv`  
`\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:NNNo`

`\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.

```

3419 \cs_new:Npn \exp_last_unbraced:No #1#2 { \exp_after:wN #1 #2 }
3420 \cs_new:Npn \exp_last_unbraced:NV #1#2
3421   { \exp_after:wN #1 \exp:w \__exp_eval_register:N #2 }
3422 \cs_new:Npn \exp_last_unbraced:Nv #1#2
3423   { \exp_after:wN #1 \exp:w \__exp_eval_register:c {#2} }
3424 \cs_if_exist:NTF \tex_expanded:D
3425   {
3426     \cs_new:Npn \exp_last_unbraced:Ne #1#2
3427       { \exp_after:wN #1 \tex_expanded:D {#2} }
3428   }
3429   { \cs_new:Npn \exp_last_unbraced:Ne { \::e_unbraced \::: } }
3430 \cs_new:Npn \exp_last_unbraced:Nf #1#2
3431   { \exp_after:wN #1 \exp:w \exp_end_continue_f:w #2 }
3432 \cs_new:Npn \exp_last_unbraced:NNo #1#2#3
3433   { \exp_after:wN #1 \exp_after:wN #2 #3 }
3434 \cs_new:Npn \exp_last_unbraced:NNV #1#2#3

```

```

3435   {
3436     \exp_after:wN #1
3437     \exp_after:wN #2
3438     \exp:w \__exp_eval_register:N #3
3439   }
3440 \cs_new:Npn \exp_last_unbraced:NNf #1#2#3
3441   {
3442     \exp_after:wN #1
3443     \exp_after:wN #2
3444     \exp:w \exp_end_continue_f:w #3
3445   }
3446 \cs_new:Npn \exp_last_unbraced:Nco #1#2#3
3447   { \exp_after:wN #1 \cs:w #2 \exp_after:wN \cs_end: #3 }
3448 \cs_new:Npn \exp_last_unbraced:Ncv #1#2#3
3449   {
3450     \exp_after:wN #1
3451     \cs:w #2 \exp_after:wN \cs_end:
3452     \exp:w \__exp_eval_register:N #3
3453   }
3454 \cs_new:Npn \exp_last_unbraced:NNNo #1#2#3#4
3455   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 #4 }
3456 \cs_new:Npn \exp_last_unbraced:NNNV #1#2#3#4
3457   {
3458     \exp_after:wN #1
3459     \exp_after:wN #2
3460     \exp_after:wN #3
3461     \exp:w \__exp_eval_register:N #4
3462   }
3463 \cs_new:Npn \exp_last_unbraced:NNNf #1#2#3#4
3464   {
3465     \exp_after:wN #1
3466     \exp_after:wN #2
3467     \exp_after:wN #3
3468     \exp:w \exp_end_continue_f:w #4
3469   }
3470 \cs_new:Npn \exp_last_unbraced:Nno { \:::n \:::o_unbraced \::: }
3471 \cs_new:Npn \exp_last_unbraced:Noo { \:::o \:::o_unbraced \::: }
3472 \cs_new:Npn \exp_last_unbraced:Nfo { \:::f \:::o_unbraced \::: }
3473 \cs_new:Npn \exp_last_unbraced:NnNo { \:::n \:::N \:::o_unbraced \::: }
3474 \cs_new:Npn \exp_last_unbraced:NNNNo #1#2#3#4#5
3475   { \exp_after:wN #1 \exp_after:wN #2 \exp_after:wN #3 \exp_after:wN #4 #5 }
3476 \cs_new:Npn \exp_last_unbraced:NNNnf #1#2#3#4#5
3477   {
3478     \exp_after:wN #1
3479     \exp_after:wN #2
3480     \exp_after:wN #3
3481     \exp_after:wN #4
3482     \exp:w \exp_end_continue_f:w #5
3483   }
3484 \cs_new_protected:Npn \exp_last_unbraced:Nx { \:::x_unbraced \::: }

```

(End definition for `\exp_last_unbraced:No` and others. These functions are documented on page 30.)

`\exp_last_two_unbraced:Noo` If #2 is a single token then this can be implemented as  
`\__exp_last_two_unbraced:noN`

```

\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.

```

3485 \cs_new:Npn \exp_last_two_unbraced:Noo #1#2#3
3486   { \exp_after:wN \_exp_last_two_unbraced:noN \exp_after:wN {#3} {#2} #1 }
3487 \cs_new:Npn \_exp_last_two_unbraced:noN #1#2#3
3488   { \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 30.)

## 5.5 Preventing expansion

`\_kernel_exp_not:w` At the kernel level, we need the primitive behaviour to allow expansion *before* the brace group.

```

3489 \cs_new_eq:NN \_kernel_exp_not:w \tex_unexpanded:D

```

(End definition for `\_kernel_exp_not:w`.)

<code>\exp_not:c</code> <code>\exp_not:o</code> <code>\exp_not:e</code> <code>\exp_not:f</code> <code>\exp_not:v</code> <code>\exp_not:v</code>	All these except <code>\exp_not:c</code> call the kernel-internal <code>\_kernel_exp_not:w</code> namely <code>\tex_unexpanded:D</code> .  <code>\exp_not:c</code> 3490 <code>\cs_new:Npn \exp_not:c #1 { \exp_after:wN \exp_not:N \cs:w #1 \cs_end: }</code> <code>\exp_not:o</code> 3491 <code>\cs_new:Npn \exp_not:o #1 { \_kernel_exp_not:w \exp_after:wN {#1} }</code> <code>\exp_not:f</code> 3492 <code>\cs_if_exist:NTF \tex_expanded:D</code> <code>\exp_not:v</code> 3493 <code>{</code> 3494 <code>\cs_new:Npn \exp_not:e #1</code> 3495 <code>{ \_kernel_exp_not:w \tex_expanded:D { {#1} } }</code> 3496 <code>}</code> 3497 <code>{</code> 3498 <code>\cs_new:Npn \exp_not:e</code> 3499 <code>{ \_kernel_exp_not:w \exp_args:Ne \prg_do_nothing: }</code> 3500 <code>}</code> 3501 <code>\cs_new:Npn \exp_not:f #1</code> 3502 <code>{ \_kernel_exp_not:w \exp_after:wN { \exp:w \exp_end_continue_f:w #1 } }</code> 3503 <code>\cs_new:Npn \exp_not:V #1</code> 3504 <code>{</code> 3505 <code>\_kernel_exp_not:w \exp_after:wN</code> 3506 <code>{ \exp:w \_exp_eval_register:N #1 }</code> 3507 <code>}</code> 3508 <code>\cs_new:Npn \exp_not:v #1</code> 3509 <code>{</code> 3510 <code>\_kernel_exp_not:w \exp_after:wN</code> 3511 <code>{ \exp:w \_exp_eval_register:c {#1} }</code> 3512 <code>}</code>
--	--

(End definition for `\exp_not:c` and others. These functions are documented on page 31.)

## 5.6 Controlled expansion

`\exp:w` To trigger a sequence of “arbitrarily” many expansions we need a method to invoke  $\text{\TeX}$ ’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  $\text{\TeX}$ ; 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  $\text{\TeX}$ ’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.

```
3513 \group_begin:
3514   \tex_catcode:D '\^~@ = 13
3515   \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 `\xmltex.tex`.

```
3516   \if_cs_exist:N ^~@
3517   \else:
3518     \cs_new:Npn ^~@
3519     { \__kernel_msg_expandable_error:nn { kernel } { bad-exp-end-f } }
3520   \fi:
```

The same but grabbing an argument to remove spaces and braces.

```
3521   \cs_new:Npn \exp_end_continue_f:nw #1 { '^~@ #1 }
3522 \group_end:
```

(End definition for `\exp:w` and others. These functions are documented on page 33.)

## 5.7 Emulating e-type expansion

When the `\expanded` primitive is available it is used to implement e-type expansion; otherwise we emulate it.

```
3523 \cs_if_exist:NF \tex_expanded:D
3524   {
```

`\__exp_e: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`.

```

3525     \cs_new:Npn \__exp_e:nn #1
3526     {
3527         \if_false: { \fi:
3528             \tl_if_head_is_N_type:nTF {#1}
3529                 { \__exp_e:N }
3530                 {
3531                     \tl_if_head_is_group:nTF {#1}
3532                         { \__exp_e_group:n }
3533                         {
3534                             \tl_if_empty:nTF {#1}
3535                                 { \exp_after:wN \__exp_e_end:nn }
3536                                 { \exp_after:wN \__exp_e_space:nn }
3537                             \exp_after:wN { \if_false: } \fi:
3538                         }
3539                     }
3540                 #1
3541             }
3542         }
3543     \cs_new:Npn \__exp_e_end:nn #1#2 { \exp_end: #2 }
```

(End definition for `\__exp_e:nn`.)

`\__exp_e_space:nn` For an explicit space character, remove it by f-expansion and put it in the (future) output.

```

3544     \cs_new:Npn \__exp_e_space:nn #1#2
3545         { \exp_args:Nf \__exp_e:nn {#1} { #2 ~ } }
```

(End definition for `\__exp_e_space:nn`.)

`\__exp_e_group:n` 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.

```

3546     \cs_new:Npn \__exp_e_group:n #1
3547     {
3548         \exp_after:wN \__exp_e_put:nn
3549         \exp_after:wN { \exp_after:wN { \exp_after:wN {
3550             \exp:w \if_false: } \fi: \__exp_e:nn {#1} { } } }
3551     }
3552     \cs_new:Npn \__exp_e_put:nn #1
3553     {
3554         \exp_args:NNo \exp_args:No \__exp_e_put:nnn
3555             { \tl_head:n {#1} } {#1}
3556     }
3557     \cs_new:Npn \__exp_e_put:nnn #1#2#3
3558         { \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` 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`.

```

3559      \cs_new:Npn \_exp_e:N #1
3560      {
3561          \exp_after:wN \_exp_e:Nnn
3562          \exp_after:wN #1
3563          \exp_after:wN { \if_false: } \fi:
3564      }
3565      \cs_new:Npn \_exp_e:Nnn #1
3566      {
3567          \if_case:w
3568              \exp_after:wN \if_meaning:w \exp_not:N #1 #1 1 ~ \fi:
3569              \token_if_protected_macro:NT #1 { 1 ~ }
3570              \token_if_protected_long_macro:NT #1 { 1 ~ }
3571              \if_meaning:w \exp_not:n #1 2 ~ \fi:
3572              \if_meaning:w \exp_not:N #1 3 ~ \fi:
3573              \if_meaning:w \tex_the:D #1 4 ~ \fi:
3574              \if_meaning:w \tex_primitive:D #1 5 ~ \fi:
3575              0 ~
3576              \exp_after:wN \_exp_e_expandable:Nnn
3577              \or: \exp_after:wN \_exp_e_protected:Nnn
3578              \or: \exp_after:wN \_exp_e_unexpanded:Nnn
3579              \or: \exp_after:wN \_exp_e_noexpand:Nnn
3580              \or: \exp_after:wN \_exp_e_the:Nnn
3581              \or: \exp_after:wN \_exp_e_primitive:Nnn
3582              \fi:
3583              #1
3584      }
3585      \cs_new:Npn \_exp_e_protected:Nnn #1#2#3
3586      { \_exp_e:nn {#2} {#3 #1} }
3587      \cs_new:Npn \_exp_e_expandable:Nnn #1#
3588      { \exp_args:No \_exp_e:nn {#1 #2} }

```

(End definition for `\_exp_e:N`.)

`\_exp_e_primitive:Nnn` Quite rare. Will be implemented later.

```

3589      \cs_new:Npn \_exp_e_primitive:Nnn #1
3590      {
3591          \_kernel_msg_expandable_error:nnn { kernel } { e-type }
3592          { \primitive not-implemented }
3593          \_exp_e:nn
3594      }

```

(End definition for `\_exp_e_primitive:Nnn`.)

`\_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.

```

3595      \cs_new:Npn \_exp_e_noexpand:Nnn #1#2
3596      {
3597          \tl_if_head_is_N_type:nTF {#2}
3598          { \_exp_e_put:nn } { \_exp_e:nn } {#2}
3599      }

```

(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`. That primitive can be followed by an expandable N-type token, to be 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).

```
3600      \cs_new:Npn \_\_exp_e_unexpanded:Nnn #1 { \_\_exp_e_unexpanded:nn }
3601      \cs_new:Npn \_\_exp_e_unexpanded:nn #1
3602      {
3603          \tl_if_head_is_N_type:nTF {#1}
3604          {
3605              \exp_args:Nf \_\_exp_e_unexpanded:nn
3606              { \_\_exp_e_unexpanded:nN {#1} #1 }
3607          }
3608          {
3609              \tl_if_head_is_group:nTF {#1}
3610              { \_\_exp_e_put:nn }
3611              {
3612                  \tl_if_empty:nTF {#1}
3613                  {
3614                      \_kernel_msg_expandable_error:nnn
3615                      { kernel } { e-type }
3616                      { \unexpanded missing~brace }
3617                      \_\_exp_e_end:nn
3618                  }
3619                  { \exp_args:Nf \_\_exp_e_unexpanded:nn }
3620              }
3621              {#1}
3622          }
3623      }
3624      \cs_new:Npn \_\_exp_e_unexpanded:nN #1#2
3625      {
3626          \exp_after:wN \if_meaning:w \exp_not:N #2 #2
3627          \exp_after:wN \use_i:nn
3628          \else:
3629          \exp_after:wN \use_ii:nn
3630          \fi:
3631      {
3632          \token_if_eq_catcode:NNTF #2 \c_space_token
3633          { \exp_stop_f: }
3634      }
```

```

3635     \token_if_eq_meaning:NNTF #2 \scan_stop:
3636     { \exp_stop_f: }
3637     {
3638         \_kernel_msg_expandable_error:nnn
3639         { kernel } { e-type }
3640         { \unexpanded missing~brace }
3641         { }
3642     }
3643 }
3644 }
3645 {
3646     \token_if_eq_meaning:NNTF #2 \exp_not:N
3647     {
3648         \exp_args:No \tl_if_head_is_N_type:nT { \use_none:n #1 }
3649         { \_exp_e_unexpanded:N }
3650     }
3651     { \exp_after:wN \exp_stop_f: #2 }
3652 }
3653 }
3654 \cs_new:Npn \_exp_e_unexpanded:N #1
3655 {
3656     \exp_after:wN \if_meaning:w \exp_not:N #1 #1 \else:
3657         \exp_after:wN \use_i:nn
3658         \fi:
3659         \exp_stop_f: #1
3660 }

```

(End definition for `\_exp_e_unexpanded:Nnn` and others.)

`\_exp_e_the:Nnn` Finally implement `\the`. Followed by anything other than an N-type `<token>` this causes an error (we just let TeX make one), otherwise we test the `<token>`. If the `<token>` 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 `<token>` 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.

```

3661     \cs_new:Npn \_exp_e_the:Nnn #1#2
3662     {
3663         \tl_if_head_is_N_type:nTF {#2}
3664         { \if_false: { \fi: \_exp_e_the:N #2 } }
3665         { \exp_args:No \_exp_e:nn { \tex_the:D #2 } }
3666     }
3667 \cs_new:Npn \_exp_e_the:N #1
3668 {
3669     \exp_after:wN \if_meaning:w \exp_not:N #1 #1
3670         \exp_after:wN \use_i:nn
3671         \else:
3672             \exp_after:wN \use_ii:nn
3673             \fi:
3674             \if_meaning:w \tex_toks:D #1
3675                 \exp_after:wN \_exp_e_the_toks:wnn \int_value:w
3676

```

```

3677   \exp_after:wN \__exp_e_the_toks:n
3678   \exp_after:wN { \int_value:w \if_false: } \fi:
3679 \else:
3680   \__exp_e_if_toks_register:N#1
3681   { \exp_after:wN \__exp_e_the_toks_reg:N }
3682   {
3683     \exp_after:wN \__exp_e:nn \exp_after:wN {
3684       \tex_the:D \if_false: } \fi:
3685     }
3686   \exp_after:wN #1
3687 \fi:
3688 }
3689 {
3690   \exp_after:wN \__exp_e_the:Nnn \exp_after:wN ?
3691   \exp_after:wN { \exp:w \if_false: } \fi:
3692   \exp_after:wN \exp_end: #1
3693 }
3694 }
3695 \cs_new:Npn \__exp_e_the_toks_reg:N #1
3696 {
3697   \exp_after:wN \__exp_e_put:nn \exp_after:wN {
3698   \exp_after:wN {
3699     \tex_the:D \if_false: } \fi: #1 }
3700 }

```

(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`.

```

3701   \cs_new:Npn \__exp_e_the_toks:wnn #1; #2
3702   {
3703     \exp_args:No \__exp_e_put:nnn
3704     { \tex_the:D \tex_toks:D #1 } { ? #2 }
3705   }
3706 \cs_new:Npn \__exp_e_the_toks:n #1
3707   {
3708     \tl_if_head_is_N_type:nTF {#1}
3709     { \exp_after:wN \__exp_e_the_toks:N \if_false: { \fi: #1 } }
3710     { ; {#1} }
3711   }
3712 \cs_new:Npn \__exp_e_the_toks:N #1
3713   {
3714     \if_int_compare:w 10 < 9 \token_to_str:N #1 \exp_stop_f:
3715     \exp_after:wN \use_i:nn
3716   \else:
3717     \exp_after:wN \use_ii:nn
3718   \fi:
3719   }

```

```

3720      #1
3721      \exp_after:wN \__exp_e_the_toks:n
3722    }
3723    { \exp_after:wN ;
3724    \exp_after:wN { \if_false: } \fi:
3725  }

```

(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`

We need to detect both `\toks` registers like `\toks@` (in L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub>) and parameters such as `\everypar`, as the result of unpacking the register should not expand further. The list of parameters is finite so we just use a `\cs_if_exist:cTF` test to look up in a table. Registers are found by `\token_if_toks_register:NTF` by inspecting the meaning. We abuse `\cs_to_str:N`'s ability to remove a leading escape character whatever it is.

```

3726      \prg_new_conditional:Npnn \__exp_e_if_toks_register:N #1 { TF }
3727    {
3728      \token_if_toks_register:NTF #1 { \prg_return_true: }
3729    {
3730      \cs_if_exist:cTF
3731      {
3732        __exp_e_the_
3733        \exp_after:wN \cs_to_str:N
3734        \token_to_meaning:N #1
3735        :
3736      } { \prg_return_true: } { \prg_return_false: }
3737    }
3738  }
3739  \cs_new_eq:NN \__exp_e_the_XeTeXinterchartoks: ?
3740  \cs_new_eq:NN \__exp_e_the_errhelp: ?
3741  \cs_new_eq:NN \__exp_e_the_everycr: ?
3742  \cs_new_eq:NN \__exp_e_the_everydisplay: ?
3743  \cs_new_eq:NN \__exp_e_the_everyeof: ?
3744  \cs_new_eq:NN \__exp_e_the_everyhbox: ?
3745  \cs_new_eq:NN \__exp_e_the_everyjob: ?
3746  \cs_new_eq:NN \__exp_e_the_everymath: ?
3747  \cs_new_eq:NN \__exp_e_the_everypar: ?
3748  \cs_new_eq:NN \__exp_e_the_everyvbox: ?
3749  \cs_new_eq:NN \__exp_e_the_output: ?
3750  \cs_new_eq:NN \__exp_e_the_pdfpageattr: ?
3751  \cs_new_eq:NN \__exp_e_the_pdfpageresources: ?
3752  \cs_new_eq:NN \__exp_e_the_pdfpagesattr: ?
3753  \cs_new_eq:NN \__exp_e_the_pdffpemode: ?

```

(End definition for `\__exp_e_if_toks_register:NTF`.)

We are done emulating e-type argument expansion when `\expanded` is unavailable.

```
3754  }
```

## 5.8 Defining function variants

3755 ⟨@@=cs⟩

`\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.

```

3756 \__kernel_patch:nnNNpn { \__kernel_chk_cs_exist:N #1 } {
3757 \cs_new_protected:Npn \cs_generate_variant:Nn #1#2
3758 {
3759     \__cs_generate_variant:N #1
3760     \use:x
3761     {
3762         \__cs_generate_variant:nnNN
3763         \cs_split_function:N #1
3764         \exp_not:N #1
3765         \tl_to_str:n {#2} ,
3766         \exp_not:N \scan_stop: ,
3767         \exp_not:N \q_recursion_stop
3768     }
3769 }
3770 \cs_new_protected:Npn \cs_generate_variant:cn
3771 { \exp_args:Nc \cs_generate_variant:Nn }
```

(End definition for `\cs_generate_variant:Nn`. This function is documented on page 24.)

`\__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 `\q_mark` is taken as an argument of the `wwNw` auxiliary, and `#3` is `\cs_new_protected:Npx`, otherwise it is `\cs_new:Npx`.

```

3772 \cs_new_protected:Npx \__cs_generate_variant:N #1
3773 {
3774     \exp_not:N \exp_after:wN \exp_not:N \if_meaning:w
3775     \exp_not:N \exp_not:N #1 #1
3776     \cs_set_eq:NN \exp_not:N \__cs_tmp:w \cs_new_protected:Npx
3777     \exp_not:N \else:
3778     \exp_not:N \exp_after:wN \exp_not:N \__cs_generate_variant:ww
3779     \exp_not:N \token_to_meaning:N #1 \tl_to_str:n { ma }
3780     \exp_not:N \q_mark
3781     \exp_not:N \q_mark \cs_new_protected:Npx
3782     \tl_to_str:n { pr }
3783     \exp_not:N \q_mark \cs_new:Npx
```

```

3784           \exp_not:N \q_stop
3785       \exp_not:N \fi:
3786   }
3787 \exp_last_unbraced:NNNNo
3788   \cs_new_protected:Npn \__cs_generate_variant:ww
3789     #1 { \tl_to_str:n { ma } } #2 \q_mark
3790     { \__cs_generate_variant:wwNw #1 }
3791 \exp_last_unbraced:NNNNo
3792   \cs_new_protected:Npn \__cs_generate_variant:wwNw
3793     #1 { \tl_to_str:n { pr } } #2 \q_mark #3 #4 \q_stop
3794     { \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.

```

3795 \cs_new_protected:Npn \__cs_generate_variant:nnNN #1#2#3#4
3796   {
3797     \if_meaning:w \c_false_bool #3
3798       \_kernel_msg_error:nnx { kernel } { missing-colon }
3799       { \token_to_str:c {#1} }
3800       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
3801     \fi:
3802     \__cs_generate_variant:Nnnw #4 {#1}{#2}
3803   }

```

(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  $\langle\text{processed variant signature}\rangle \backslash q\_mark \langle\text{errors}\rangle \backslash q\_stop \langle\text{base function}\rangle \langle\text{new function}\rangle$ . If all went well,  $\langle\text{errors}\rangle$  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`.

```

3804 \cs_new_protected:Npn \_cs_generate_variant:Nnnw #1#2#3#4 ,
3805   {
3806     \if_meaning:w \scan_stop: #4
3807       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
3808     \fi:
3809     \use:x
3810     {
3811       \exp_not:N \_cs_generate_variant:wwNN
3812       \_cs_generate_variant_loop:nNwN { }
3813       #4
3814       \_cs_generate_variant_loop_end:nwwwNNnn
3815       \q_mark
3816       #3 ~
3817       { ~ { } \fi: \_cs_generate_variant_loop_long:wNNnn } ~
3818       { }
3819       \q_stop
3820       \exp_not:N #1 {#2} {#4}
3821     }
3822     \_cs_generate_variant:Nnnw #1 {#2} {#3}
3823   }

```

(End definition for `\_cs_generate_variant:Nnnw`.)

```

\_cs_generate_variant_loop:nNwN #1 : Last few consecutive letters common between the base and variant (more precisely,
\_cs_generate_variant_loop_base:N \_cs_generate_variant_same:N  $\langle\text{letter}\rangle$  for each letter).
\_cs_generate_variant_loop_same:w #2 : Next variant letter.
\_cs_generate_variant_loop_end:nwwwNNnn #3 : Remainder of variant form.
\_cs_generate_variant_loop_long:wNNnn #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.

```

3824 \cs_new:Npn \__cs_generate_variant_loop:nNwN #1#2#3 \q_mark #4
3825 {
3826   \if:w #2 #4
3827     \exp_after:wN \__cs_generate_variant_loop_same:w
3828   \else:
3829     \if:w #4 \__cs_generate_variant_loop_base:N #2 \else:
3830       \if:w 0
3831         \if:w N #4 \else: \if:w n #4 \else: 1 \fi: \fi:
3832         \if:w \scan_stop: \__cs_generate_variant_loop_base:N #2 1 \fi:
3833         0
3834         \__cs_generate_variant_loop_special:NNwNNnn #4#2
3835       \else:
3836         \__cs_generate_variant_loop_invalid>NNwNNnn #4#2
3837       \fi:
3838     \fi:
3839   \fi:
3840   #1
3841   \prg_do_nothing:
3842   #2
3843   \__cs_generate_variant_loop:nNwN { } #3 \q_mark
3844 }
3845 \cs_new:Npn \__cs_generate_variant_loop_base:N #1
3846 {
3847   \if:w c #1 N \else:
3848     \if:w o #1 n \else:
3849       \if:w V #1 n \else:
3850         \if:w v #1 n \else:
3851           \if:w f #1 n \else:
3852             \if:w e #1 n \else:
3853               \if:w x #1 n \else:
3854                 \if:w n #1 n \else:
3855                   \if:w N #1 N \else:

```

```

3856           \scan_stop:
3857           \fi:
3858           \fi:
3859           \fi:
3860           \fi:
3861           \fi:
3862           \fi:
3863           \fi:
3864           \fi:
3865           \fi:
3866       }
3867 \cs_new:Npn \__cs_generate_variant_loop_same:w
3868     #1 \prg_do_nothing: #2#3#4
3869     { #3 { #1 \__cs_generate_variant_same:N #2 } }
3870 \cs_new:Npn \__cs_generate_variant_loop_end:nwwwNNnn
3871     #1#2 \q_mark #3 ~ #4 \q_stop #5#6#7#8
3872 {
3873     \scan_stop: \scan_stop: \fi:
3874     \exp_not:N \q_mark
3875     \exp_not:N \q_stop
3876     \exp_not:N #6
3877     \exp_not:c { #7 : #8 #1 #3 }
3878 }
3879 \cs_new:Npn \__cs_generate_variant_loop_long:wNNnn #1 \q_stop #2#3#4#5
3880 {
3881     \exp_not:n
3882     {
3883         \q_mark
3884         \__kernel_msg_error:nxxx { kernel } { variant-too-long }
3885         {#5} { \token_to_str:N #3 }
3886         \use_none:nnn
3887         \q_stop
3888         #3
3889         #3
3890     }
3891 }
3892 \cs_new:Npn \__cs_generate_variant_loop_invalid>NNwNNnn
3893     #1#2 \fi: \fi: \fi: \q_stop #4#5#6#7
3894 {
3895     \fi: \fi: \fi:
3896     \exp_not:n
3897     {
3898         \q_mark
3899         \__kernel_msg_error:nnxxx { kernel } { invalid-variant }
3900         {#7} { \token_to_str:N #5 } {#1} {#2}
3901         \use_none:nnn
3902         \q_stop
3903         #5
3904         #5
3905     }
3906 }
3907 \cs_new:Npn \__cs_generate_variant_loop_special>NNwNNnn
3908     #1#2#3 \q_stop #4#5#6#7
3909 {

```

```

3910      #3 \q_stop #4 #5 {#6} {#7}
3911      \exp_not:n
3912      {
3913          \_kernel_msg_error:nxxxxx
3914          { kernel } { deprecated-variant }
3915          {#7} { \token_to_str:N #5 } {#1} {#2}
3916      }
3917  }

```

(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.

```

3918  \cs_new:Npn \_cs_generate_variant_same:N #1
3919  {
3920      \if:w N #1 N \else:
3921          \if:w p #1 p \else:
3922              n
3923              \if:w n #1 \else:
3924                  \_cs_generate_variant_loop_special:NNwNNnn #1#1
3925              \fi:
3926          \fi:
3927      \fi:
3928  }

```

(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.

```

3929  \_kernel_patch:nnNNpn
3930  {
3931      \cs_if_free:NF #4
3932      {
3933          \_kernel_debug_log:x
3934          {
3935              Variant~\token_to_str:N #4~%
3936              already~defined;~ not~ changing~ it~ \msg_line_context:
3937          }
3938      }
3939  }
3940  { }
3941  \cs_new_protected:Npn \_cs_generate_variant:wwNN
3942  #1 \q_mark #2 \q_stop #3#4
3943  {
3944      #2
3945      \cs_if_free:NT #4
3946      {
3947          \group_begin:
3948          \_cs_generate_internal_variant:n {#1}
3949          \_cs_tmp:w #4 { \exp_not:c { \exp_args:N #1 } \exp_not:N #3 }

```

```

3950           \group_end:
3951     }
3952   }

```

(End definition for `\_cs_generate_variant:wwNN.`)

`\_cs_generate_internal_variant:n`  
`\_cs_generate_internal_variant:wwnw`  
`\_cs_generate_internal_variant_loop:n`

Test if `\exp_args:N #1` is already defined and if not define it via the `\:::` commands using the chars in `#1`. If `#1` contains an `x` (this is the place where having converted the original comma-list argument to a string is very important), the result should be protected, and the next variant to be defined using that internal variant should be protected.

```

3953 \cs_new_protected:Npx \_cs_generate_internal_variant:n #1
3954   {
3955     \exp_not:N \_cs_generate_internal_variant:wwnNwnn
3956     #1 \exp_not:N \q_mark
3957     { \cs_set_eq:NN \exp_not:N \_cs_tmp:w \cs_new_protected:Npx }
3958     \cs_new_protected:cpx
3959     \token_to_str:N x \exp_not:N \q_mark
3960     {
3961       \cs_new:cpx
3962       \exp_not:N \q_stop
3963       { \exp_args:N #1 }
3964     {
3965       \exp_not:N \_cs_generate_internal_variant_loop:n #1
3966       { : \exp_not:N \use_i:nn }
3967     }
3968   }
3969 \exp_last_unbraced:NNNNo
3970   \cs_new_protected:Npn \_cs_generate_internal_variant:wwnNwnn #1
3971   { \token_to_str:N x } #2 \q_mark #3#4#5 \q_stop #6#7
3972   {
3973     #3
3974     \cs_if_free:cT {#6} { #4 {#6} {#7} }
3975   }

```

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.

```

3976 \cs_new:Npn \_cs_generate_internal_variant_loop:n #1
3977   {
3978     \exp_after:wN \exp_not:N \cs:w :: #1 \cs_end:
3979     \_cs_generate_internal_variant_loop:n
3980   }

```

(End definition for `\_cs_generate_internal_variant:n`, `\_cs_generate_internal_variant:wwnw`, 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

```

```

3981 \cs_new_protected:Npn \prg_generate_conditional_variant:Nnn #1
3982   {
3983     \use:x
3984     {
3985       \_cs_generate_variant:nnNnn
3986       \cs_split_function:N #1

```

```

3987     }
3988   }
3989 \cs_new_protected:Npn \__cs_generate_variant:nnNnn #1#2#3#4#5
3990   {
3991     \if_meaning:w \c_false_bool #3
3992       \_kernel_msg_error:nnx { kernel } { missing-colon }
3993       { \token_to_str:c {#1} }
3994       \use_i_delimit_by_q_stop:nw
3995     \fi:
3996     \exp_after:wN \__cs_generate_variant:w
3997     \tl_to_str:n {#5} , \scan_stop: , \q_recursion_stop
3998     \use_none_delimit_by_q_stop:w \q_mark {#1} {#2} {#4} \q_stop
3999   }
4000 \cs_new_protected:Npn \__cs_generate_variant:w
4001   #1 , #2 \q_mark #3#4#5
4002   {
4003     \if_meaning:w \scan_stop: #1 \scan_stop:
4004     \if_meaning:w \q_nil #1 \q_nil
4005       \use_i:nnn
4006     \fi:
4007     \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
4008   \else:
4009     \cs_if_exist_use:cTF { __cs_generate_variant_#1_form:nnn }
4010     { {#3} {#4} {#5} }
4011     {
4012       \_kernel_msg_error:nnxx
4013       { kernel } { conditional-form-unknown }
4014       {#1} { \token_to_str:c {#3 : #4} }
4015     }
4016   \fi:
4017   \__cs_generate_variant:w #2 \q_mark {#3} {#4} {#5}
4018 }
4019 \cs_new_protected:Npn \__cs_generate_variant_p_form:nnn #1#2
4020   { \cs_generate_variant:cn { #1 _p : #2 } }
4021 \cs_new_protected:Npn \__cs_generate_variant_T_form:nnn #1#2
4022   { \cs_generate_variant:cn { #1 : #2 T } }
4023 \cs_new_protected:Npn \__cs_generate_variant_F_form:nnn #1#2
4024   { \cs_generate_variant:cn { #1 : #2 F } }
4025 \cs_new_protected:Npn \__cs_generate_variant_TF_form:nnn #1#2
4026   { \cs_generate_variant:cn { #1 : #2 TF } }

```

(End definition for `\prg_generate_conditional_variant:Nnn` and others. This function is documented on page [97](#).)

**\exp\_args\_generate:n**  
`\__cs_args_generate:Nn`  
`\__cs_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 loop through the variant specifier and convert each letter to `\:::(variant letter)`, with a trailing `\:::::`.

```

4027 \cs_new_protected:Npn \exp_args_generate:n #1
4028   {
4029     \exp_args:No \clist_map_inline:nn { \tl_to_str:n {#1} }
4030     {
4031       \str_map_inline:nn {##1}

```

```

4032      {
4033          \str_if_in:nnF { NnpcofVvx } {####1}
4034          {
4035              \_kernel_msg_error:nnnn { kernel } { invalid-exp-args }
4036              {####1} {##1}
4037              \str_map_break:n { \use_none:nnnn }
4038          }
4039      }
4040      \exp_args:Nc \__cs_args_generate:Nn { exp_args:N ##1 } {##1}
4041  }
4042 }
4043 \cs_new_protected:Npn \__cs_args_generate:Nn #1#2
4044 {
4045     \cs_if_exist:NF #1
4046     {
4047         \str_if_in:nnTF {#2} { x } { \cs_new_protected:Npx } { \cs_new:Npx }
4048         #1 { \tl_map_function:nN {#2 : } \__cs_args_generate:n }
4049     }
4050 }
4051 \cs_new:Npn \__cs_args_generate:n #1 { \exp_not:c { :: #1 } }

(End definition for \exp_args_generate:n, \__cs_args_generate:Nn, and \__cs_args_generate:n.  

This function is documented on page 244.)
```

4052 ⟨/initex | package⟩

## 6 **l3tl** implementation

```

4053 (*initex | package)
4054 ⟨@@=tl⟩
```

A token list variable is a **TEX** macro that holds tokens. By using the  $\varepsilon$ -**TEX** primitive `\unexpanded` inside a **TEX** `\edef` it is possible to store any tokens, including `#`, in this way.

### 6.1 Functions

**\tl\_new:N** Creating new token list variables is a case of checking for an existing definition and doing the definition.

```

4055 \cs_new_protected:Npn \tl_new:N #1
4056 {
4057     \_kernel_chk_if_free_cs:N #1
4058     \cs_gset_eq:NN #1 \c_empty_tl
4059 }
4060 \cs_generate_variant:Nn \tl_new:N { c }
```

(End definition for \tl\_new:N. This function is documented on page 35.)

**\tl\_const:Nn** Constants are also easy to generate.

```

\tl_const:Nx 4061 \_kernel_patch:nnNNpn { \_kernel_chk_var_scope:NN c #1 } { }
\tl_const:cn 4062 \cs_new_protected:Npn \tl_const:Nn #1#2
\tl_const:cx 4063 {
4064     \_kernel_chk_if_free_cs:N #1
4065     \cs_gset_nopar:Npx #1 { \exp_not:n {#2} }
4066 }
```

```

4067 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
4068 \cs_new_protected:Npn \tl_const:Nx #1#2
4069 {
4070     \__kernel_chk_if_free_cs:N #1
4071     \cs_gset_nopar:Npx #1 {#2}
4072 }
4073 \cs_generate_variant:Nn \tl_const:Nn { c }
4074 \cs_generate_variant:Nn \tl_const:Nx { c }

```

(End definition for `\tl_const:Nn`. This function is documented on page 35.)

**\tl\_clear:N** Clearing a token list variable means setting it to an empty value. Error checking is sorted out by the parent function.

```

4075 \cs_new_protected:Npn \tl_clear:N #1
4076 { \tl_set_eq:NN #1 \c_empty_tl }
4077 \cs_new_protected:Npn \tl_gclear:N #1
4078 { \tl_gset_eq:NN #1 \c_empty_tl }
4079 \cs_generate_variant:Nn \tl_clear:N { c }
4080 \cs_generate_variant:Nn \tl_gclear:N { c }

```

(End definition for `\tl_clear:N` and `\tl_gclear:N`. These functions are documented on page 35.)

**\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.

```

4081 \cs_new_protected:Npn \tl_clear_new:N #1
4082 { \tl_if_exist:NTF #1 { \tl_clear:N #1 } { \tl_new:N #1 } }
4083 \cs_new_protected:Npn \tl_gclear_new:N #1
4084 { \tl_if_exist:NTF #1 { \tl_gclear:N #1 } { \tl_new:N #1 } }
4085 \cs_generate_variant:Nn \tl_clear_new:N { c }
4086 \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 36.)

**\tl\_set\_eq:NN** For setting token list variables equal to each other. When checking is turned on, make sure both variables exist.

```

4087 \__kernel_if_debug:TF
4088 {
4089     \cs_new_protected:Npn \tl_set_eq:NN #1#2
4090     {
4091         \__kernel_chk_var_local:N #1
4092         \__kernel_chk_var_exist:N #2
4093         \cs_set_eq:NN #1 #2
4094     }
4095     \cs_new_protected:Npn \tl_gset_eq:NN #1#2
4096     {
4097         \__kernel_chk_var_global:N #1
4098         \__kernel_chk_var_exist:N #2
4099         \cs_gset_eq:NN #1 #2
4100     }
4101 }
4102 {
4103     \cs_new_eq:NN \tl_set_eq:NN \cs_set_eq:NN
4104     \cs_new_eq:NN \tl_gset_eq:NN \cs_gset_eq:NN
4105 }

```

```

4106 \cs_generate_variant:Nn \tl_set_eq:NN { cN, Nc, cc }
4107 \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 36.)

**\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.

```

\tl_concat:ccc
4108 \__kernel_patch:nnNNpn
4109 {
4110     \__kernel_chk_var_exist:N #2
4111     \__kernel_chk_var_exist:N #3
4112 }
4113 {
4114 \cs_new_protected:Npn \tl_concat:NNN #1#2#3
4115 { \tl_set:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4116 \__kernel_patch:nnNNpn
4117 {
4118     \__kernel_chk_var_exist:N #2
4119     \__kernel_chk_var_exist:N #3
4120 }
4121 {
4122 \cs_new_protected:Npn \tl_gconcat:NNN #1#2#3
4123 { \tl_gset:Nx #1 { \exp_not:o {#2} \exp_not:o {#3} } }
4124 \cs_generate_variant:Nn \tl_concat:NNN { ccc }
4125 \cs_generate_variant:Nn \tl_gconcat:NNN { ccc }

```

(End definition for `\tl_concat:NNN` and `\tl_gconcat:NNN`. These functions are documented on page 36.)

**\tl\_if\_exist\_p:N** Copies of the `cs` functions defined in `l3basics`.

```

\tl_if_exist_p:c
4126 \prg_new_eq_conditional:NNn \tl_if_exist:N \cs_if_exist:N { TF , T , F , p }
\tl_if_exist:NTF
4127 \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 36.)

## 6.2 Constant token lists

**\c\_empty\_t1** Never full. We need to define that constant before using `\tl_new:N`.

```

4128 \tl_const:Nn \c_empty_t1 { }

```

(End definition for `\c_empty_t1`. This variable is documented on page 49.)

**\c\_novalue\_t1** A special marker: as we don't have `\char_generate:nn` yet, has to be created the old-fashioned way.

```

4129 \group_begin:
4130 \tex_lccode:D 'A = '-
4131 \tex_lccode:D 'N = 'N
4132 \tex_lccode:D 'V = 'V
4133 \tex_lowercase:D
4134 {
4135     \group_end:
4136     \tl_const:Nn \c_novalue_t1 { ANoValue- }
4137 }

```

(End definition for `\c_novalue_t1`. This variable is documented on page 49.)

**\c\_space\_tl** A space as a token list (as opposed to as a character).

```
4138 \tl_const:Nn \c_space_tl { ~ }
```

(End definition for **\c\_space\_tl**. This variable is documented on page 49.)

### 6.3 Adding to token list variables

**\tl\_set:Nn**

**\tl\_set:NV**

**\tl\_set:Nv**

**\tl\_set:No**

**\tl\_set:Nf**

**\tl\_set:Nx**

**\tl\_set:cn**

**\tl\_set:cV**

**\tl\_set:cv**

**\tl\_set:co**

**\tl\_set:cf**

**\tl\_set:cx**

**\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**

**\tl\_gset:cx**

```
4139 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4140 \cs_new_protected:Npn \tl_set:Nn #1#2
4141   { \cs_set_nopar:Npx #1 { \exp_not:n {#2} } }
4142 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4143 \cs_new_protected:Npn \tl_set:No #1#2
4144   { \cs_set_nopar:Npx #1 { \exp_not:o {#2} } }
4145 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4146 \cs_new_protected:Npn \tl_set:Nx #1#2
4147   { \cs_set_nopar:Npx #1 {#2} }
4148 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4149 \cs_new_protected:Npn \tl_gset:Nn #1#2
4150   { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} } }
4151 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4152 \cs_new_protected:Npn \tl_gset:No #1#2
4153   { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} } }
4154 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4155 \cs_new_protected:Npn \tl_gset:Nx #1#2
4156   { \cs_gset_nopar:Npx #1 {#2} }
4157 \cs_generate_variant:Nn \tl_set:Nn { NV , Nv , Nf }
4158 \cs_generate_variant:Nn \tl_set:Nx { c }
4159 \cs_generate_variant:Nn \tl_set:Nn { c, co , cV , cv , cf }
4160 \cs_generate_variant:Nn \tl_gset:Nn { NV , Nv , Nf }
4161 \cs_generate_variant:Nn \tl_gset:Nx { c }
4162 \cs_generate_variant:Nn \tl_gset:Nn { c, co , cV , cv , cf }
```

(End definition for **\tl\_set:Nn** and **\tl\_gset:Nn**. These functions are documented on page 36.)

**\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**

```
4163 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4164 \cs_new_protected:Npn \tl_put_left:Nn #1#2
4165   { \cs_set_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
4166 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4167 \cs_new_protected:Npn \tl_put_left:NV #1#2
4168   { \cs_set_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4169 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4170 \cs_new_protected:Npn \tl_put_left:No #1#2
4171   { \cs_set_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4172 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
4173 \cs_new_protected:Npn \tl_put_left:Nx #1#2
4174   { \cs_set_nopar:Npx #1 { #2 \exp_not:o #1 } }
4175 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4176 \cs_new_protected:Npn \tl_gput_left:Nn #1#2
4177   { \cs_gset_nopar:Npx #1 { \exp_not:n {#2} \exp_not:o #1 } }
```

```

4178 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4179 \cs_new_protected:Npn \tl_gput_left:NV #1#2
4180   { \cs_gset_nopar:Npx #1 { \exp_not:V #2 \exp_not:o #1 } }
4181 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4182 \cs_new_protected:Npn \tl_gput_left:No #1#2
4183   { \cs_gset_nopar:Npx #1 { \exp_not:o {#2} \exp_not:o #1 } }
4184 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
4185 \cs_new_protected:Npn \tl_gput_left:Nx #1#2
4186   { \cs_gset_nopar:Npx #1 { #2 \exp_not:o {#1} } }
4187 \cs_generate_variant:Nn \tl_put_left:Nn { c }
4188 \cs_generate_variant:Nn \tl_put_left:NV { c }
4189 \cs_generate_variant:Nn \tl_put_left:No { c }
4190 \cs_generate_variant:Nn \tl_put_left:Nx { c }
4191 \cs_generate_variant:Nn \tl_gput_left:Nn { c }
4192 \cs_generate_variant:Nn \tl_gput_left:NV { c }
4193 \cs_generate_variant:Nn \tl_gput_left:No { c }
4194 \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 36.)

`\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
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_put_right:Nn #1#2
{ \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_put_right:NV #1#2
{ \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_put_right:No #1#2
{ \cs_set_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:Nn #1#2
{ \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:Nx #1#2
{ \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:n {#2} } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:NV #1#2
{ \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:V #2 } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:No #1#2
{ \cs_gset_nopar:Npx #1 { \exp_not:o #1 \exp_not:o {#2} } }
\__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\cs_new_protected:Npn \tl_gput_right:Nx #1#2
{ \cs_gset_nopar:Npx #1 { \exp_not:o {#1} #2 } }
\cs_generate_variant:Nn \tl_put_right:Nn { c }
\cs_generate_variant:Nn \tl_put_right:NV { c }
\cs_generate_variant:Nn \tl_put_right:No { c }
\cs_generate_variant:Nn \tl_put_right:Nx { c }
\cs_generate_variant:Nn \tl_gput_right:Nn { c }
\cs_generate_variant:Nn \tl_gput_right:NV { c }
\cs_generate_variant:Nn \tl_gput_right:No { c }
\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 36.)

## 6.4 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.

4227 `\tl_const:Nx \c__tl_rescan_marker_tl { : \token_to_str:N : }`

(End definition for `\c__tl_rescan_marker_tl`.)

`\tl_set_rescan:Nnn`

`\tl_set_rescan:Nno`

`\tl_set_rescan:Nnx`

`\tl_set_rescan:cnn`

`\tl_set_rescan:cno`

`\tl_set_rescan:cnx`

`\tl_gset_rescan:Nnn`

`\tl_gset_rescan:Nno`

`\tl_gset_rescan:Nnx`

`\tl_gset_rescan:cnn`

`\tl_gset_rescan:cno`

`\tl_gset_rescan:cnx`

`\tl_rescan:nn`

`\_tl_set_rescan:NNnn`

`\_tl_set_rescan_multi:nNN`

`\_tl_rescan:NNw`

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 distinguishes single-line and multi-line files as explained below. 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: it 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 TeX 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_set_rescan:NNw`, that performs the assignment, then let TeX “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 `\oexpand` 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:NNnn \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.

The two `\if_false: ... \fi:` are there to prevent alignment tabs to cause a change of tabular cell while rescanning.

```
4228 \cs_new_protected:Npn \tl_set_rescan:Nnn
4229   { \_tl_set_rescan:NNnn \tl_set:No }
4230 \cs_new_protected:Npn \tl_gset_rescan:Nnn
4231   { \_tl_set_rescan:NNnn \tl_gset:No }
4232 \cs_new_protected:Npn \tl_rescan:nn #1#2
4233   {
4234     \tl_set_rescan:Nnn \l__tl_internal_a_tl {#1} {#2}
4235     \exp_after:wN \tl_clear:N \exp_after:wN \l__tl_internal_a_tl
4236     \l__tl_internal_a_tl
4237   }
4238 \cs_new_protected:Npn \_tl_set_rescan:NNnn #1#2#3#4
4239   {
4240     \if_false: { \fi:
4241     \group_begin:
4242       \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
```

```

4243   \exp_args:No \tex_everyeof:D { \c__tl_rescan_marker_tl }
4244   \int_compare:nNnT \tex_endlinechar:D = { 32 }
4245     { \int_set:Nn \tex_endlinechar:D { -1 } }
4246     \int_set_eq:NN \tex_newlinechar:D \tex_endlinechar:D
4247   #3 \scan_stop:
4248   \exp_args:No \__tl_set_rescan:nNN { \tl_to_str:n {#4} } #1 #2
4249   \if_false: } \fi:
4250 }
4251 \cs_new_protected:Npn \__tl_set_rescan_multi:nNN #1#2#3
4252 {
4253   \exp_after:wN \__tl_rescan>NNw
4254   \exp_after:wN #2
4255   \exp_after:wN #3
4256   \exp_after:wN \prg_do_nothing:
4257   \tex_scantokens:D {#1}
4258 }
4259 \exp_args:Nno \use:nn
4260 { \cs_new:Npn \__tl_rescan>NNw #1#2#3 } \c__tl_rescan_marker_tl
4261 {
4262   \group_end:
4263   #1 #2 {#3}
4264 }
4265 \cs_generate_variant:Nn \tl_set_rescan:Nnn { Nno , Nnx }
4266 \cs_generate_variant:Nn \tl_set_rescan:Nnn { c , cno , cnx }
4267 \cs_generate_variant:Nn \tl_gset_rescan:Nnn { Nno , Nnx }
4268 \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 38.)

`\__tl_set_rescan:nNN`  
`\__tl_set_rescan_single:nnNN`  
`\__tl_set_rescan_single_aux:nnNN`

This function 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.

The case of multiple lines is a straightforward application of `\scantokens` as described above. 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.

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). Once a valid character is found,

run some code very similar to `\_tl_set_rescan_multi:nNN` but with that character put at the start and end. The auxiliary `\_tl_rescan>NNw` must be redefined to also remove the additional character (with the appropriate catcode thanks to `\char_generate:nn`). If no valid character is found (very rare), fall-back on `\_tl_set_rescan_multi:nNN`.

```

4269 \cs_new_protected:Npn \_tl_set_rescan:nNN #1
4270   {
4271     \int_compare:nNnTF \tex_newlinechar:D < 0
4272       { \use_i:nn }
4273       {
4274         \exp_args:Nnf \tl_if_in:nnTF {#1}
4275           { \char_generate:nn { \tex_newlinechar:D } { 12 } }
4276       }
4277       { \_tl_set_rescan_multi:nNN }
4278       {
4279         \int_set:Nn \tex_endlinechar:D { -1 }
4280         \_tl_set_rescan_single:nnNN { ' }
4281       }
4282     {#1}
4283   }
4284 \cs_new_protected:Npn \_tl_set_rescan_single:nnNN #1
4285   {
4286     \int_compare:nNnTF
4287       { \char_value_catcode:n {#1} / 2 } = 6
4288     {
4289       \exp_args:Nf \_tl_set_rescan_single_aux:nnNN
4290         { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
4291     }
4292     {
4293       \int_compare:nNnTF {#1} < { '\~ }
4294         {
4295           \exp_args:Nf \_tl_set_rescan_single:nnNN
4296             { \int_eval:n { #1 + 1 } }
4297         }
4298       { \_tl_set_rescan_multi:nNN }
4299     }
4300   }
4301 \cs_new_protected:Npn \_tl_set_rescan_single_aux:nnNN #1#2#3#4
4302   {
4303     \exp_args:Nno \use:nn
4304       { \cs_set:Npn \_tl_rescan:NNw ##1##2##3 #1 }
4305       \c_tl_rescan_marker_tl
4306       {
4307         \group_end:
4308           ##1 ##2 { \use_none:n ##3 }
4309       }
4310       \exp_after:wN \_tl_rescan:NNw
4311       \exp_after:wN #3
4312       \exp_after:wN #4
4313       \tex_scantokens:D { #1 #2 #1 }
4314   }

```

(End definition for `\_tl_set_rescan:nNN`, `\_tl_set_rescan_single:nnNN`, and `\_tl_set_rescan_single_aux:nnNN`.)

## 6.5 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 `\tl_set:Nx` or `\tl_gset:Nx` for local or global replacements. Finally, the three arguments `\{tl var\} \{(pattern)\} \{(replacement)\}` provided by the user. When describing the auxiliary functions below, we denote the contents of the `\{tl var\}` by `\{token list\}`.

```
4315 \cs_new_protected:Npn \tl_replace_once:Nnn
4316   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_set:Nx }
4317 \cs_new_protected:Npn \tl_greplace_once:Nnn
4318   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_wrap:w \tl_gset:Nx }
4319 \cs_new_protected:Npn \tl_replace_all:Nnn
4320   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_set:Nx }
4321 \cs_new_protected:Npn \tl_greplace_all:Nnn
4322   { \__tl_replace:NnNNNnn \q_mark ? \__tl_replace_next:w \tl_gset:Nx }
4323 \cs_generate_variant:Nn \tl_replace_once:Nnn { c }
4324 \cs_generate_variant:Nn \tl_greplace_once:Nnn { c }
4325 \cs_generate_variant:Nn \tl_replace_all:Nnn { c }
4326 \cs_generate_variant:Nn \tl_greplace_all:Nnn { c }
```

(End definition for `\tl_replace_all:Nnn` and others. These functions are documented on page 37.)

```
\__tl_replace:NnNNNnn
\__tl_replace_auxi:NnnNNNnn
\__tl_replace_auxii:nNNNnn
\__tl_replace_next:w
\__tl_replace_wrap:w
```

To implement the actual replacement auxiliary `\__tl_replace_auxii:nNNNnn` we need a `\{delimiter\}` with the following properties:

- all occurrences of the `\{pattern\} #6` in “`\{token list\} \{delimiter\}`” belong to the `\{token list\}` and have no overlap with the `\{delimiter\}`,
- the first occurrence of the `\{delimiter\}` in “`\{token list\} \{delimiter\}`” is the trailing `\{delimiter\}`.

We first find the building blocks for the `\{delimiter\}`, namely two tokens `\{A\}` and `\{B\}` such that `\{A\}` does not appear in `#6` and `#6` is not `\{B\}` (this condition is trivial if `#6` has more than one token). Then we consider the delimiters “`\{A\}`” and “`\{A\} \{A\}^n \{B\} \{A\}^n \{B\}`”, for  $n \geq 1$ , where `\{A\}^n` denotes  $n$  copies of `\{A\}`, and we choose as our `\{delimiter\}` the first one which is not in the `\{token list\}`.

Every delimiter in the set obeys the first condition: `#6` does not contain `\{A\}` hence cannot be overlapping with the `\{token list\}` and the `\{delimiter\}`, and it cannot be within the `\{delimiter\}` since it would have to be in one of the two `\{B\}` 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 `\{delimiter\}` we choose does not appear in the `\{token list\}`. Additionally, the set of delimiters is such that a `\{token list\}` of  $n$  tokens can contain at most  $O(n^{1/2})$  of them, hence we find a `\{delimiter\}` 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 “`\{A\}`” in the list of delimiters to try, so that the `\{delimiter\}` is simply `\q_mark` in the most common situation where neither the `\{token list\}` nor the `\{pattern\}` contains `\q_mark`.

Let us now ahead, optimizing for this most common case. First, two special cases: an empty `\{pattern\} #6` is an error, and if `#1` is absent from both the `\{token list\} #5` and the `\{pattern\} #6` then we can use it as the `\{delimiter\}` through `\__tl_replace_auxii:nNNNnn`

{#1}. Otherwise, we end up calling `\__tl_replace:NnNNNnn` repeatedly with the first two arguments `\q_mark {?}`, `\? {??}`, `\?? {???`, 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 `\q_nil` or `\q_stop` such that it is not equal to #6.

The `\__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 `auxii` auxiliary.

```

4327 \cs_new_protected:Npn \__tl_replace:NnNNNnn #1#2#3#4#5#6#7
4328 {
4329   \tl_if_empty:nTF {#6}
4330   {
4331     \__kernel_msg_error:nnx { kernel } { empty-search-pattern }
4332     { \tl_to_str:n {#7} }
4333   }
4334   {
4335     \tl_if_in:onTF { #5 #6 } {#1}
4336     {
4337       \tl_if_in:nnTF {#6} {#1}
4338       { \exp_args:Nc \__tl_replace:NnNNNnn {#2} {#2?} }
4339       {
4340         \quark_if_nil:nTF {#6}
4341         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_stop } }
4342         { \__tl_replace_auxi:NnnNNNnn #5 {#1} { #1 \q_nil } }
4343       }
4344     }
4345     { \__tl_replace_auxii:nNNNnn {#1} }
4346     #3#4#5 {#6} {#7}
4347   }
4348 }
4349 \cs_new_protected:Npn \__tl_replace_auxi:NnnNNNnn #1#2#3
4350 {
4351   \tl_if_in:NnTF #1 { #2 #3 #3 }
4352   { \__tl_replace_auxi:NnnNNNnn #1 { #2 #3 } {#2?} }
4353   { \__tl_replace_auxii:nNNNnn { #2 #3 #3 } }
4354 }
```

The auxiliary `\__tl_replace_auxii:nNNNnn` receives the following arguments:

```

{⟨delimiter⟩} ⟨function⟩ ⟨assignment⟩
⟨tl var⟩ {⟨pattern⟩} {⟨replacement⟩}
```

All of its work is done between `\group_align_safe_begin:` and `\group_align_safe_end:` to avoid issues in alignments. It does the actual replacement within #3 #4 {...}, an x-expanding `⟨assignment⟩` #3 to the `⟨tl var⟩` #4. The auxiliary `\__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 `\__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 `\__tl_replace_wrap:w`

gets `##1` as its own argument `##1`, and protects it against the `x`-expanding assignment. It also finds `\exp_not:n` as `##2` and does nothing to it, thus letting through `\exp_not:n {⟨replacement⟩}` into the assignment. Note that `\_tl_replace_next:w` and `\_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⟩`).

```

4355 \cs_new_protected:Npn \_tl_replace_auxii:nNNNnn #1#2#3#4#5#6
4356   {
4357     \group_align_safe_begin:
4358     \cs_set:Npn \_tl_replace_wrap:w ##1 #1 ##2
4359       { \exp_not:o { \use_none:nn ##1 } ##2 }
4360     \cs_set:Npx \_tl_replace_next:w ##1 #5
4361       {
4362         \exp_not:N \_tl_replace_wrap:w ##1
4363         \exp_not:n { #1 }
4364         \exp_not:n { \exp_not:n {#6} }
4365         \exp_not:n { #2 { } { } }
4366       }
4367     #3 #4
4368   {
4369     \exp_after:wN \_tl_replace_next:w
4370     \exp_after:wN { \exp_after:wN }
4371     \exp_after:wN { \exp_after:wN }
4372     #4
4373     #1
4374   {
4375     \if_false: { \fi: }
4376     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4377   }
4378   #5
4379 }
4380 \group_align_safe_end:
4381 }
4382 \cs_new_eq:NN \_tl_replace_wrap:w ?
4383 \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
4384 \cs_new_protected:Npn \tl_remove_once:Nn #1#2
4385   { \tl_replace_once:Nnn #1 {#2} { } }
\tl_gremove_once:Nn
4386 \cs_new_protected:Npn \tl_gremove_once:Nn #1#2
4387   { \tl_greplace_once:Nnn #1 {#2} { } }

```

```

4388 \cs_generate_variant:Nn \tl_remove_once:Nn { c }
4389 \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 37.)

`\tl_remove_all:Nn` Removal is just a special case of replacement.

```

\tl_remove_all:cn 4390 \cs_new_protected:Npn \tl_remove_all:Nn #1#2
\tl_gremove_all:cn 4391 { \tl_replace_all:Nnn #1 {#2} { } }
\tl_gremove_all:cn 4392 \cs_new_protected:Npn \tl_gremove_all:Nn #1#2
4393 { \tl_greplace_all:Nnn #1 {#2} { } }
4394 \cs_generate_variant:Nn \tl_remove_all:Nn { c }
4395 \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 37.)

## 6.6 Token list conditionals

`\tl_if_blank_p:n` TeX skips spaces when reading a non-delimited arguments. Thus, a *<token list>* is blank if and only if `\use_none:n <token list>` ? 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_meaning:w \q_nil ... \q_nil`.

```

\tl_if_blank_p:V 4396 \prg_new_conditional:Npnn \tl_if_blank:n #1 { p , T , F , TF }
\tl_if_blank_p:oTF 4397 {
\__tl_if_blank_p:NNw 4398 \_tl_if_empty_if:o { \use_none:n #1 ? }
4399 \prg_return_true:
4400 \else:
4401 \prg_return_false:
4402 \fi:
4403 }
4404 \prg_generate_conditional_variant:Nnn \tl_if_blank:n
4405 { 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 38.)

`\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:NTF 4406 \prg_new_conditional:Npnn \tl_if_empty:N #1 { p , T , F , TF }
\tl_if_empty:cTF 4407 {
4408 \if_meaning:w #1 \c_empty_tl
4409 \prg_return_true:
4410 \else:
4411 \prg_return_false:
4412 \fi:
4413 }
4414 \prg_generate_conditional_variant:Nnn \tl_if_empty:N
4415 { c } { p , T , F , TF }

```

(End definition for `\tl_if_empty:NTF`. This function is documented on page 39.)

**\tl\_if\_empty\_p:n** Convert the argument to a string; this is empty if and only if the argument is. Then  
**\tl\_if\_empty\_p:v** **\if\_meaning:w \q\_nil ... \q\_nil** is true if and only if the string ... is empty. It  
**\tl\_if\_empty:nTF** could be tempting to use **\if\_meaning:w \q\_nil #1 \q\_nil** directly. This fails on a  
**\tl\_if\_empty:VTF** token list starting with **\q\_nil** of course but more troubling is the case where argument  
is a complete conditional such as **\if\_true: a \else: b \fi:** because then **\if\_true:**  
is used by **\if\_meaning:w**, the test turns out false, the **\else:** executes the false  
branch, the **\fi:** ends it and the **\q\_nil** at the end starts executing...

```

4416 \prg_new_conditional:Npnn \tl_if_empty:n #1 { p , TF , T , F }
4417   {
4418     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4419       \tl_to_str:n {#1} \q_nil
4420       \prg_return_true:
4421     \else:
4422       \prg_return_false:
4423     \fi:
4424   }
4425 \prg_generate_conditional_variant:Nnn \tl_if_empty:n
4426   { V } { p , TF , T , F }

```

(End definition for **\tl\_if\_empty:nTF**. This function is documented on page 39.)

**\tl\_if\_empty\_p:o** The auxiliary function **\\_tl\_if\_empty\_if:o** is for use in various token list conditionals  
**\tl\_if\_empty:oTF** which reduce to testing if a given token list is empty after applying a simple function to it.  
**\\_tl\_if\_empty\_if:o** 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.

```

4427 \cs_new:Npn \_tl_if_empty_if:o #1
4428   {
4429     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4430       \_kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
4431   }
4432 \prg_new_conditional:Npnn \tl_if_empty:o #1 { p , TF , T , F }
4433   {
4434     \_tl_if_empty_if:o {#1}
4435     \prg_return_true:
4436   \else:
4437     \prg_return_false:
4438   \fi:
4439 }

```

(End definition for **\tl\_if\_empty:nTF** and **\\_tl\_if\_empty\_if:o**. This function is documented on page 39.)

**\tl\_if\_eq\_p:NN** Returns **\c\_true\_bool** if and only if the two token list variables are equal.  
**\tl\_if\_eq\_p:Nc** **\prg\_new\_conditional:Npnn \tl\_if\_eq:NN #1#2 { p , T , F , TF }**  
**\tl\_if\_eq\_p:cN** 4440  
**\tl\_if\_eq\_p:cc** 4441  
**\tl\_if\_eq:NNTF** 4442 **\if\_meaning:w #1 #2**  
**\tl\_if\_eq:NcTF** 4443 **\prg\_return\_true:**  
**\tl\_if\_eq:cNTF** 4444 **\else:**  
**\tl\_if\_eq:cNTF** 4445 **\prg\_return\_false:**  
**\tl\_if\_eq:ccTF** 4446 **\fi:**  
4447 }

```

4448 \prg_generate_conditional_variant:Nnn \tl_if_eq:NN
4449   { Nc , c , cc } { p , TF , T , F }

```

(End definition for `\tl_if_eq:NNTF`. This function is documented on page 39.)

**\tl\_if\_eq:nnTF** A simple store and compare routine.

```

\l__tl_internal_a_tl 4450 \prg_new_protected_conditional:Npnn \tl_if_eq:nn #1#2 { T , F , TF }
\l__tl_internal_b_tl 4451 {
 4452   \group_begin:
 4453     \tl_set:Nn \l__tl_internal_a_tl {#1}
 4454     \tl_set:Nn \l__tl_internal_b_tl {#2}
 4455     \exp_after:wN
 4456   \group_end:
 4457   \if_meaning:w \l__tl_internal_a_tl \l__tl_internal_b_tl
 4458     \prg_return_true:
 4459   \else:
 4460     \prg_return_false:
 4461   \fi:
 4462 }
4463 \tl_new:N \l__tl_internal_a_tl
4464 \tl_new:N \l__tl_internal_b_tl

```

(End definition for `\tl_if_eq:nnTF`, `\l__tl_internal_a_tl`, and `\l__tl_internal_b_tl`. This function is documented on page 39.)

**\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`.

```

4465 \cs_new_protected:Npn \tl_if_in:NnT { \exp_args:No \tl_if_in:nnT }
4466 \cs_new_protected:Npn \tl_if_in:NnF { \exp_args:No \tl_if_in:nnF }
4467 \cs_new_protected:Npn \tl_if_in:NnTF { \exp_args:No \tl_if_in:nnTF }
4468 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn
  { c } { T , F , TF }

```

(End definition for `\tl_if_in:NnTF`. This function is documented on page 39.)

**\tl\_if\_in:nTF** Once more, the test relies on the emptiness test for robustness. The function `\_tl_tmpt:w` removes tokens until the first occurrence of `#2`. If this does not appear in `#1`, then `\tl_if_in:onTF` is removed, leaving an empty token list. Otherwise some tokens remain, and 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:`.

```

4470 \prg_new_protected_conditional:Npnn \tl_if_in:nn #1#2 { T , F , TF }
4471 {
 4472   \if_false: { \fi:
 4473     \cs_set:Npn \_tl_tmpt:w ##1 #2 { }
 4474     \tl_if_empty:otF { \_tl_tmpt:w #1 {} {} #2 }
 4475       { \prg_return_false: } { \prg_return_true: }
 4476     \if_false: } \fi:
 4477 }
4478 \prg_generate_conditional_variant:Nnn \tl_if_in:nn
  { V , o , no } { T , F , TF }

```

(End definition for `\tl_if_in:nTF`. This function is documented on page 39.)

`\tl_if_novalue_p:n` Tests for `-NoValue-`: this is similar to `\tl_if_in:n` but set up to be expandable and to check the value exactly. The question mark prevents the auxiliary from losing braces.

```
4480 \cs_set_protected:Npn \__tl_tmp:w #1
4481 {
4482     \prg_new_conditional:Npnn \tl_if_novalue:n ##1
4483         { p , T , F , TF }
4484     {
4485         \str_if_eq:onTF
4486             { \__tl_if_novalue:w ? ##1 { } #1 }
4487             { ? { } #1 }
4488             { \prg_return_true: }
4489             { \prg_return_false: }
4490     }
4491     \cs_new:Npn \__tl_if_novalue:w ##1 #1 {##1}
4492 }
4493 \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 39.)

`\tl_if_single_p:N` Expand the token list and feed it to `\tl_if_single:n`.

`\tl_if_single:NTF`

```
4494 \cs_new:Npn \tl_if_single_p:N { \exp_args:No \tl_if_single_p:n }
4495 \cs_new:Npn \tl_if_single:NT { \exp_args:No \tl_if_single:nT }
4496 \cs_new:Npn \tl_if_single:NF { \exp_args:No \tl_if_single:nF }
4497 \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 40.)

`\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, `\tl_to_str:n` 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:nw` picks the second token in front of it. If `#1` is empty, this token is the trailing `?` and the catcode test yields `false`. If `#1` has a single item, the token is `^` and the catcode test yields `true`. Otherwise, it is one of the characters resulting from `\tl_to_str:n`, and the catcode test yields `false`. Note that `\if_catcode:w` and `\_kernel_tl_to_str:w` are primitives that take care of expansion.

```
4498 \prg_new_conditional:Npnn \tl_if_single:n #1 { p , T , F , TF }
4499 {
4500     \if_catcode:w ^ \exp_after:wN \__tl_if_single:nnw
4501         \_kernel_tl_to_str:w
4502             \exp_after:wN { \use_none:nn #1 ?? } ^ ? \q_stop
4503             \prg_return_true:
4504         \else:
4505             \prg_return_false:
4506         \fi:
4507     }
4508 \cs_new:Npn \__tl_if_single:nnw #1#2#3 \q_stop {#2}
```

(End definition for `\tl_if_single:nTF` and `\__tl_if_single:nTF`. This function is documented on page 40.)

```

\tl_case:Nn
\tl_case:cn
\tl_case:NnTF
\tl_case:cNTF
\__tl_case:nNTF
\__tl_case:Nw
\__tl_case_end:nw
4509 \cs_new:Npn \tl_case:Nn #1#2
4510 {
4511     \exp:w
4512     \__tl_case:NnTF #1 {#2} { } { }
4513 }
4514 \cs_new:Npn \tl_case:NnT #1#2#3
4515 {
4516     \exp:w
4517     \__tl_case:NnTF #1 {#2} {#3} { }
4518 }
4519 \cs_new:Npn \tl_case:NnF #1#2#3
4520 {
4521     \exp:w
4522     \__tl_case:NnTF #1 {#2} { } {#3}
4523 }
4524 \cs_new:Npn \tl_case:NnTF #1#2
4525 {
4526     \exp:w
4527     \__tl_case:NnTF #1 {#2}
4528 }
4529 \cs_new:Npn \__tl_case:NnTF #1#2#3#4
4530 { \__tl_case:Nw #1 #2 #1 { } \q_mark {#3} \q_mark {#4} \q_stop }
4531 \cs_new:Npn \__tl_case:Nw #1#2#3
4532 {
4533     \tl_if_eq:NNTF #1 #2
4534     { \__tl_case_end:nw {#3} }
4535     { \__tl_case:Nw #1 }
4536 }
4537 \cs_generate_variant:Nn \tl_case:Nn { c }
4538 \prg_generate_conditional_variant:Nnn \tl_case:Nn
4539 { 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 \q\_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 \q\_mark and so #4 is the **false** code (the **true** code is mopped up by #3).

```

4540 \cs_new:Npn \__tl_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
4541 { \exp_end: #1 #4 }

```

(End definition for \tl\_case:NnTF and others. This function is documented on page 40.)

## 6.7 Mapping to token lists

```

\tl_map_function:nN
\tl_map_function:NN
\tl_map_function:cN
\__tl_map_function:Nn

```

Expandable loop macro for token lists. These have the advantage of not needing to test if the argument is empty, because if it is, the stop marker is read immediately and the

loop terminated.

```

4542 \cs_new:Npn \tl_map_function:nN #1#2
4543 {
4544     \__tl_map_function:Nn #2 #1
4545     \q_recursion_tail
4546     \prg_break_point:Nn \tl_map_break: { }
4547 }
4548 \cs_new:Npn \tl_map_function:NN
4549 { \exp_args:No \tl_map_function:nN }
4550 \cs_new:Npn \__tl_map_function:Nn #1#2
4551 {
4552     \quark_if_recursion_tail_break:nN {#2} \tl_map_break:
4553     #1 {#2} \__tl_map_function:Nn #1
4554 }
4555 \cs_generate_variant:Nn \tl_map_function:NN { c }

```

(End definition for `\tl_map_function:nN`, `\tl_map_function:NN`, and `\__tl_map_function:Nn`. These functions are documented on page 40.)

`\tl_map_inline:nn`  
`\tl_map_inline:Nn`  
`\tl_map_inline:cn`

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:Nn` from before.

```

4556 \cs_new_protected:Npn \tl_map_inline:nn #1#2
4557 {
4558     \int_gincr:N \g_kernel_prg_map_int
4559     \cs_gset_protected:cpn
4560     { \__tl_map_ \int_use:N \g_kernel_prg_map_int :w } ##1 {#2}
4561     \exp_args:NC \__tl_map_function:Nn
4562     { \__tl_map_ \int_use:N \g_kernel_prg_map_int :w }
4563     #1 \q_recursion_tail
4564     \prg_break_point:Nn \tl_map_break:
4565     { \int_gdecr:N \g_kernel_prg_map_int }
4566 }
4567 \cs_new_protected:Npn \tl_map_inline:Nn
4568 { \exp_args:No \tl_map_inline:nn }
4569 \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 41.)

`\tl_map_variable:nNn`  
`\tl_map_variable:NNn`  
`\tl_map_variable:cNn`  
`\__tl_map_variable:Nnn`

`\tl_map_variable:nNn` *<token list>* *<temp>* *<action>* assigns *<temp>* to each element and executes *<action>*.

```

4570 \cs_new_protected:Npn \tl_map_variable:nNn #1#2#3
4571 {
4572     \__tl_map_variable:Nnn #2 {#3} #1
4573     \q_recursion_tail
4574     \prg_break_point:Nn \tl_map_break: { }
4575 }
4576 \cs_new_protected:Npn \tl_map_variable:NNn
4577 { \exp_args:No \tl_map_variable:nNn }
4578 \cs_new_protected:Npn \__tl_map_variable:Nnn #1#2#3
4579 {
4580     \tl_set:Nn #1 {#3}
4581     \quark_if_recursion_tail_break:NN #1 \tl_map_break:

```

```

4582     \use:n {#2}
4583     \_tl_map_variable:Nnn #1 {#2}
4584   }
4585 \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 41.)
```

**\tl\_map\_break:** The break statements use the general \prg\_map\_break:Nn.

```

\tl_map_break:n
4586 \cs_new:Npn \tl_map_break:
4587   { \prg_map_break:Nn \tl_map_break: { } }
4588 \cs_new:Npn \tl_map_break:n
4589   { \prg_map_break:Nn \tl_map_break: }
```

(End definition for \tl\_map\_break: and \tl\_map\_break:n. These functions are documented on page 41.)

## 6.8 Using token lists

**\tl\_to\_str:n** Another name for a primitive: defined in l3basics.

```

\tl_to_str:V
4590 \cs_generate_variant:Nn \tl_to_str:n { V }
```

(End definition for \tl\_to\_str:n. This function is documented on page 42.)

**\tl\_to\_str:N** These functions return the replacement text of a token list as a string.

```

\tl_to_str:c
4591 \cs_new:Npn \tl_to_str:N #1 { \_kernel_tl_to_str:w \exp_after:wN {#1} }
4592 \cs_generate_variant:Nn \tl_to_str:N { c }
```

(End definition for \tl\_to\_str:N. This function is documented on page 42.)

**\tl\_use:N** Token lists which are simply not defined give a clear TeX error here. No such luck for  
**\tl\_use:c** ones equal to \scan\_stop: so instead a test is made and if there is an issue an error is forced.

```

\cs_new:Npn \tl_use:N #1
4594   {
4595     \tl_if_exist:NTF #1 {#1}
4596     {
4597       \_kernel_msg_expandable_error:nnn
4598         { kernel } { bad-variable } {#1}
4599     }
4600   }
4601 \cs_generate_variant:Nn \tl_use:N { c }
```

(End definition for \tl\_use:N. This function is documented on page 43.)

## 6.9 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.

```

4602 \cs_new:Npn \tl_count:n #1
4603 {
4604     \int_eval:n
4605     { 0 \tl_map_function:nN {#1} \_tl_count:n }
4606 }
4607 \cs_new:Npn \tl_count:N #1
4608 {
4609     \int_eval:n
4610     { 0 \tl_map_function:NN #1 \_tl_count:n }
4611 }
4612 \cs_new:Npn \_tl_count:n #1 { + 1 }
4613 \cs_generate_variant:Nn \tl_count:n { V , o }
4614 \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 43.)

**\tl\_reverse\_items:n** Reversal of a token list is done by taking one item at a time and putting it after **\q\_stop**.

```

\__tl_reverse_items:nwNwn
\__tl_reverse_items:wn
4615 \cs_new:Npn \tl_reverse_items:n #1
4616 {
4617     \_tl_reverse_items:nwNwn #1 ?
4618     \q_mark \_tl_reverse_items:nwNwn
4619     \q_mark \_tl_reverse_items:wn
4620     \q_stop { }
4621 }
4622 \cs_new:Npn \_tl_reverse_items:nwNwn #1 #2 \q_mark #3 #4 \q_stop #5
4623 {
4624     #3 #2
4625     \q_mark \_tl_reverse_items:nwNwn
4626     \q_mark \_tl_reverse_items:wn
4627     \q_stop { {#1} #5 }
4628 }
4629 \cs_new:Npn \_tl_reverse_items:wn #1 \q_stop #2
4630 { \exp_not:o { \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 43.)

**\tl\_trim\_spaces:n** 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 **\q\_mark**, and whose second argument is a *<continuation>*, which receives as a braced argument **\use\_none:n \q\_mark** (*trimmed token list*). In the case at hand, we take **\exp\_not:o** as our continuation, so that space trimming behaves correctly within an x-type expansion.

```

\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
4631 \cs_new:Npn \tl_trim_spaces:n #1
4632 { \_tl_trim_spaces:nn { \q_mark #1 } \exp_not:o }
4633 \cs_generate_variant:Nn \tl_trim_spaces:n { o }
4634 \cs_new:Npn \tl_trim_spaces_apply:nN #1#2
4635 { \_tl_trim_spaces:nn { \q_mark #1 } { \exp_args:No #2 } }
4636 \cs_generate_variant:Nn \tl_trim_spaces_apply:nN { o }

```

```

4637 \cs_new_protected:Npn \tl_trim_spaces:N #1
4638   { \tl_set:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4639 \cs_new_protected:Npn \tl_gtrim_spaces:N #1
4640   { \tl_gset:Nx #1 { \exp_args:No \tl_trim_spaces:n {#1} } }
4641 \cs_generate_variant:Nn \tl_trim_spaces:N { c }
4642 \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 `\q_nil`. Removing leading spaces is done with `\_tl_trim_spaces_auxi:w`, which loops until `\q_mark` matches the 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 `\q_nil` matches the one present in the definition of `\tl_trim_spacs:n`. Then `\_tl_trim_spaces_auxiv:w` puts the token list into a group, with `\use_none:n` placed there to gobble a lingering `\q_mark`, and feeds this to the *(continuation)*.

```

4643 \cs_set:Npn \_tl_tmp:w #1
4644   {
4645     \cs_new:Npn \_tl_trim_spaces:nn ##1
4646       {
4647         \_tl_trim_spaces_auxi:w
4648           ##
4649           \q_nil
4650           \q_mark #1 { }
4651           \q_mark \_tl_trim_spaces_auxii:w
4652           \_tl_trim_spaces_auxiii:w
4653           #1 \q_nil
4654           \_tl_trim_spaces_auxiv:w
4655             \q_stop
4656       }
4657     \cs_new:Npn \_tl_trim_spaces_auxi:w ##1 \q_mark #1 ##2 \q_mark ##3
4658       {
4659         ##
4660         \_tl_trim_spaces_auxi:w
4661         \q_mark
4662         ##
4663         \q_mark #1 {##1}
4664       }
4665     \cs_new:Npn \_tl_trim_spaces_auxii:w
4666       \_tl_trim_spaces_auxi:w \q_mark \q_mark ##1
4667       {
4668         \_tl_trim_spaces_auxiii:w
4669         ##
4670       }
4671     \cs_new:Npn \_tl_trim_spaces_auxiii:w ##1 #1 \q_nil ##
4672       {
4673         ##
4674         ##2
4675         ##1 \q_nil
4676         \_tl_trim_spaces_auxiii:w
4677       }
4678     \cs_new:Npn \_tl_trim_spaces_auxiv:w ##1 \q_nil ##2 \q_stop ##3
4679       { ##3 { \use_none:n ##1 } }

```

```

4679 }
4680 \__tl_tmp:w { ~ }
```

(End definition for `\tl_trim_spaces:n` and others. These functions are documented on page 44.)

`\tl_sort:Nn` Implemented in `l3sort`.

`\tl_sort:cn`

`\tl_gsort:Nn`

`\tl_gsort:cn`

`\tl_sort:nN`

## 6.10 Token by token changes

`\q__tl_act_mark`

`\q__tl_act_stop`

The `\__tl_act_...` functions may be applied to any token list. Hence, we use two private quarks, to allow any token, even quarks, in the token list. Only `\q__tl_act_mark` and `\q__tl_act_stop` may not appear in the token lists manipulated by `\__tl_act:NNNnn` functions. No quark module yet, so do things by hand.

```

4681 \cs_new_nopar:Npn \q__tl_act_mark { \q__tl_act_mark }
4682 \cs_new_nopar:Npn \q__tl_act_stop { \q__tl_act_stop }
```

(End definition for `\q__tl_act_mark` and `\q__tl_act_stop`.)

`\__tl_act:NNNnn`

`\__tl_act_output:n`

`\__tl_act_reverse_output:n`

`\__tl_act_loop:w`

`\__tl_act_normal:NwnNNN`

`\__tl_act_group:nwnNNN`

`\__tl_act_space:wwnNNN`

`\__tl_act_end:w`

To help control the expansion, `\__tl_act:NNNnn` should always be proceeded by `\exp:w` and ends by producing `\exp_end:` once the result has been obtained. Then loop over tokens, groups, and spaces in #5. The marker `\q__tl_act_mark` is used both to avoid losing outer braces and to detect the end of the token list more easily. The result is stored as an argument for the dummy function `\__tl_act_result:n`.

```

4683 \cs_new:Npn \__tl_act:NNNnn #1#2#3#4#5
4684 {
4685     \group_align_safe_begin:
4686     \__tl_act_loop:w #5 \q__tl_act_mark \q__tl_act_stop
4687     {#4} #1 #2 #3
4688     \__tl_act_result:n { }
4689 }
```

In the loop, we check how the token list begins and act accordingly. In the “normal” case, we may have reached `\q__tl_act_mark`, 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 a group or with a space. Some extra work is needed to make `\__tl_act_space:wwnNNN` gobble the space.

```

4690 \cs_new:Npn \__tl_act_loop:w #1 \q__tl_act_stop
4691 {
4692     \tl_if_head_is_N_type:nTF {#1}
4693     { \__tl_act_normal:NwnNNN }
4694     {
4695         \tl_if_head_is_group:nTF {#1}
4696         { \__tl_act_group:nwnNNN }
4697         { \__tl_act_space:wwnNNN }
4698     }
4699     #1 \q__tl_act_stop
4700 }
4701 \cs_new:Npn \__tl_act_normal:NwnNNN #1 #2 \q__tl_act_stop #3#4
4702 {
4703     \if_meaning:w \q__tl_act_mark #1
```

```

4704     \exp_after:wN \__tl_act_end:wn
4705     \fi:
4706     #4 {#3} #1
4707     \__tl_act_loop:w #2 \q__tl_act_stop
4708     {#3} #4
4709   }
4710 \cs_new:Npn \__tl_act_end:wn #1 \__tl_act_result:n #2
4711   { \group_align_safe_end: \exp_end: #2 }
4712 \cs_new:Npn \__tl_act_group:nwnNNN #1 #2 \q__tl_act_stop #3#4#5
4713   {
4714     #5 {#3} {#1}
4715     \__tl_act_loop:w #2 \q__tl_act_stop
4716     {#3} #4 #5
4717   }
4718 \exp_last_unbraced:NNo
4719   \cs_new:Npn \__tl_act_space:wwnNNN \c_space_tl #1 \q__tl_act_stop #2#3#4#5
4720   {
4721     #5 {#2}
4722     \__tl_act_loop:w #1 \q__tl_act_stop
4723     {#2} #3 #4 #5
4724   }

```

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.

```

4725 \cs_new:Npn \__tl_act_output:n #1 #2 \__tl_act_result:n #3
4726   { #2 \__tl_act_result:n { #3 #1 } }
4727 \cs_new:Npn \__tl_act_reverse_output:n #1 #2 \__tl_act_result:n #3
4728   { #2 \__tl_act_result:n { #1 #3 } }

```

(End definition for `\__tl_act:NNNnn` and others.)

**\tl\_reverse:n**   **\tl\_reverse:o**   **\tl\_reverse:v**  
**\\_\_tl\_reverse\_normal:nN**   **\\_\_tl\_reverse\_group\_preserve:nn**  
**\\_\_tl\_reverse\_space:n**

The goal here is to reverse without losing spaces nor braces. This is done using the general internal function `\__tl_act:NNNnn`. 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. All of the internal functions here drop one argument: this is needed by `\__tl_act:NNNnn` when changing case (to record which direction the change is in), but not when reversing the tokens.

```

4729 \cs_new:Npn \tl_reverse:n #1
4730   {
4731     \__kernel_exp_not:w \exp_after:wN
4732     {
4733       \exp:w
4734       \__tl_act:NNNnn
4735       \__tl_reverse_normal:nN
4736       \__tl_reverse_group_preserve:nn
4737       \__tl_reverse_space:n
4738       { }
4739       {#1}
4740     }
4741   }
4742 \cs_generate_variant:Nn \tl_reverse:n { o , v }
4743 \cs_new:Npn \__tl_reverse_normal:nN #1#2
4744   { \__tl_act_reverse_output:n {#2} }
4745 \cs_new:Npn \__tl_reverse_group_preserve:nn #1#2

```

```

4746   { \__tl_act_reverse_output:n { {#2} } }
4747 \cs_new:Npn \__tl_reverse_space:n #1
4748   { \__tl_act_reverse_output:n { ~ } }

```

(End definition for `\tl_reverse:n` and others. This function is documented on page 43.)

`\tl_reverse:N` This reverses the list, leaving `\exp_stop_f:` in front, which stops the f-expansion.

```

4749 \cs_new_protected:Npn \tl_reverse:N #1
4750   { \tl_set:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
4751 \cs_new_protected:Npn \tl_greverse:N #1
4752   { \tl_gset:Nx #1 { \exp_args:No \tl_reverse:n { #1 } } }
4753 \cs_generate_variant:Nn \tl_reverse:N { c }
4754 \cs_generate_variant:Nn \tl_greverse:N { c }

```

(End definition for `\tl_reverse:N` and `\tl_greverse:N`. These functions are documented on page 43.)

## 6.11 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 consistent with for example mapping to a list. The empty brace groups in `\tl_head:n` ensure that a blank argument gives an empty result. The result is returned within the `\unexpanded` primitive. The approach here is to use `\if_false:` to allow us to use `}` as the closing delimiter: this is the only safe choice, as any other token would not be able to parse its own code. Using a marker, we can see if what we are grabbing is exactly the marker, or there is anything else to deal with. Is there is, there is a loop. If not, tidy up and leave the item in the output stream. More detail in

<http://tex.stackexchange.com/a/70168>.

```

4755 \cs_new:Npn \tl_head:n #1
4756   {
4757     \__kernel_exp_not:w
4758     \if_false: { \fi: \__tl_head_auxi:nw #1 { } \q_stop }
4759   }
4760 \cs_new:Npn \__tl_head_auxi:nw #1#2 \q_stop
4761   {
4762     \exp_after:wN \__tl_head_auxii:n \exp_after:wN {
4763       \if_false: } \fi: {#1}
4764   }
4765 \cs_new:Npn \__tl_head_auxii:n #1
4766   {
4767     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
4768     \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #1 } \q_nil
4769     \exp_after:wN \use_i:nn
4770     \else:
4771       \exp_after:wN \use_ii:nn
4772     \fi:
4773     {#1}
4774     { \if_false: { \fi: \__tl_head_auxi:nw #1 } }
4775   }
4776 \cs_generate_variant:Nn \tl_head:n { V , v , f }
4777 \cs_new:Npn \tl_head:w #1#2 \q_stop {#1}
4778 \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.

```
4779 \cs_new:Npn \tl_tail:n #1
4780   {
4781     \__kernel_exp_not:w
4782     \tl_if_blank:nTF {#1}
4783     { {} }
4784     { \exp_after:wN { \use_none:n #1 } }
4785   }
4786 \cs_generate_variant:Nn \tl_tail:n { V , v , f }
4787 \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 45.)

`\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`

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 tokens: ? which is taken in the `\if_charcode:w` test, and `\use_none:nn`, which ensures that `\prg_return_false:` is returned regardless of whether the charcode test was `true` or `false`.

```
4788 \prg_new_conditional:Npnn \tl_if_head_eq_charcode:nN #1#2 { p , T , F , TF }
4789   {
4790     \if_charcode:w
4791       \exp_not:N #2
4792       \tl_if_head_is_N_type:nTF { #1 ? }
4793     {
4794       \exp_after:wN \exp_not:N
4795       \tl_head:w #1 { ? \use_none:nn } \q_stop
4796     }
4797     { \str_head:n {#1} }
4798     \prg_return_true:
4799   \else:
4800     \prg_return_false:
4801   \fi:
```

```

4802    }
4803 \prg_generate_conditional_variant:Nnn \tl_if_head_eq_catcode:nN
4804   { 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 `\prg_return_true:` and `\else:` with `\use_none:nn` in case the catcode test with the (arbitrarily chosen) ? is true.

```

4805 \prg_new_conditional:Npnn \tl_if_head_eq_catcode:nN #1 #2 { p , T , F , TF }
4806 {
4807   \if_catcode:w
4808     \exp_not:N #2
4809     \tl_if_head_is_N_type:nTF { #1 ? }
4810     {
4811       \exp_after:wN \exp_not:N
4812       \tl_head:w #1 { ? \use_none:nn } \q_stop
4813     }
4814     {
4815       \tl_if_head_is_group:nTF {#1}
4816       { \c_group_begin_token }
4817       { \c_space_token }
4818     }
4819     \prg_return_true:
4820   \else:
4821     \prg_return_false:
4822   \fi:
4823 }

```

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 the usual `\prg_return_true:` and `\else::`. 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.

```

4824 \prg_new_conditional:Npnn \tl_if_head_eq_meaning:nN #1#2 { p , T , F , TF }
4825 {
4826   \tl_if_head_is_N_type:nTF { #1 ? }
4827   { \__tl_if_head_eq_meaning_normal:nN }
4828   { \__tl_if_head_eq_meaning_special:nN }
4829   {#1} #2
4830 }
4831 \cs_new:Npn \__tl_if_head_eq_meaning_normal:nN #1 #2
4832 {
4833   \exp_after:wN \if_meaning:w
4834   \tl_head:w #1 { ?? \use_none:nnn } \q_stop #2
4835   \prg_return_true:
4836   \else:
4837     \prg_return_false:
4838   \fi:
4839 }
4840 \cs_new:Npn \__tl_if_head_eq_meaning_special:nN #1 #2

```

```

4841   {
4842     \if_charcode:w \str_head:n {#1} \exp_not:N #2
4843       \exp_after:wN \use:n
4844     \else:
4845       \prg_return_false:
4846       \exp_after:wN \use_none:n
4847     \fi:
4848   {
4849     \if_catcode:w \exp_not:N #2
4850       \tl_if_head_is_group:nTF {#1}
4851         { \c_group_begin_token }
4852         { \c_space_token }
4853       \prg_return_true:
4854     \else:
4855       \prg_return_false:
4856     \fi:
4857   }
4858 }
```

(End definition for `\tl_if_head_eq_meaning:nNTF` and others. These functions are documented on page 46.)

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:w` produces ^ (and otherwise nothing). In the third case (begin-group token), the lines involving `\exp_after:wN` 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 \* to the beginning: if #1 contains primitive conditionals, all of its occurrences must be dealt with before the `\if_catcode:w` tries to skip the `true` branch of the conditional.

```

4859 \prg_new_conditional:Npnn \tl_if_head_is_N_type:n #1 { p , T , F , TF }
4860   {
4861     \if_catcode:w
4862       \if_false: { \fi: \_tl_if_head_is_N_type:w ? #1 ~ }
4863       \exp_after:wN \use_none:n
4864       \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
4865       * *
4866       \prg_return_true:
4867     \else:
4868       \prg_return_false:
4869     \fi:
4870   }
4871 \cs_new:Npn \_tl_if_head_is_N_type:w #1 ~
4872   {
4873     \tl_if_empty:cTF { \use_none:n #1 } { ^ } { }
4874     \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4875   }
```

(End definition for `\tl_if_head_is_N_type:nTF` and `\_tl_if_head_is_N_type:w`. This function is documented on page 46.)

`\tl_if_head_is_group_p:n` Pass the first token of #1 through `\token_to_str:N`, then check for the brace balance.  
`\tl_if_head_is_group:nTF` The extra ? caters for an empty argument.<sup>6</sup>

---

<sup>6</sup>Bruno: this could be made faster, but we don't: if we hope to ever have an e-type argument, we

```

4876 \prg_new_conditional:Npnn \tl_if_head_is_group:n #1 { p , T , F , TF }
4877  {
4878      \if_catcode:w
4879          \exp_after:wN \use_none:n
4880          \exp_after:wN { \exp_after:wN { \token_to_str:N #1 ? } }
4881          * *
4882          \prg_return_false:
4883      \else:
4884          \prg_return_true:
4885      \fi:
4886  }

```

(End definition for `\tl_if_head_is_group:nTF`. This function is documented on page 46.)

`\tl_if_head_is_space_p:n`  
`\tl_if_head_is_space:nTF`  
`\_tl_if_head_is_space:w`

The auxiliary's argument is all that is before the first explicit space in `?#1?~`. If that is a single `?` 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 `\exp:w` and `\exp_end:` ensure that the result of a single step of expansion directly yields a balanced token list (no trailing closing brace).

```

4887 \prg_new_conditional:Npnn \tl_if_head_is_space:n #1 { p , T , F , TF }
4888  {
4889      \exp:w \if_false: { \fi:
4890          \_tl_if_head_is_space:w ? #1 ? ~ }
4891      }
4892 \cs_new:Npn \_tl_if_head_is_space:w #1 ~
4893  {
4894      \tl_if_empty:cTF { \use_none:n #1 }
4895      { \exp_after:wN \exp_end: \exp_after:wN \prg_return_true: }
4896      { \exp_after:wN \exp_end: \exp_after:wN \prg_return_false: }
4897      \exp_after:wN \use_none:n \exp_after:wN { \if_false: } \fi:
4898  }

```

(End definition for `\tl_if_head_is_space:nTF` and `\_tl_if_head_is_space:w`. This function is documented on page 46.)

## 6.12 Using a single item

`\tl_item:nn`  
`\tl_item:Nn`  
`\tl_item:cn`

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 `\quark_if_recursion_tail_stop:n` terminates the loop, and returns nothing at all.

```

4899 \cs_new:Npn \tl_item:nn #1#2
4900  {
4901      \exp_args:Nf \_tl_item:nn
4902      { \exp_args:Nf \_tl_item_aux:nn { \int_eval:n {#2} } {#1} }
4903      #1
4904      \q_recursion_tail
4905      \prg_break_point:
4906  }
4907 \cs_new:Npn \_tl_item_aux:nn #1#2

```

---

need all brace “tricks” to happen in one step of expansion, keeping the token list brace balanced at all times.

```

4908  {
4909    \int_compare:nNnTF {#1} < 0
4910    { \int_eval:n { \tl_count:n {#2} + 1 + #1 } }
4911    {#1}
4912  }
4913 \cs_new:Npn \__tl_item:nn #1#2
4914  {
4915    \quark_if_recursion_tail_break:nN {#2} \prg_break:
4916    \int_compare:nNnTF {#1} = 1
4917    { \prg_break:n { \exp_not:n {#2} } }
4918    { \exp_args:Nf \__tl_item:nn { \int_eval:n { #1 - 1 } } }
4919  }
4920 \cs_new:Npn \tl_item:Nn { \exp_args:No \tl_item:nn }
4921 \cs_generate_variant:Nn \tl_item:Nn { c }

```

(End definition for `\tl_item:nn` and others. These functions are documented on page 47.)

`\tl_rand_item:n` Importantly `\tl_item:nn` only evaluates its argument once.

```

4922 \cs_new:Npn \tl_rand_item:n #1
4923  {
4924    \tl_if_blank:nF {#1}
4925    { \tl_item:nn {#1} { \int_rand:nn { 1 } { \tl_count:n {#1} } } }
4926  }
4927 \cs_new:Npn \tl_rand_item:N { \exp_args:No \tl_rand_item:n }
4928 \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 47.)

`\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`.

```

4929 \cs_new:Npn \tl_range:Nnn { \exp_args:No \tl_range:nnn }
4930 \cs_generate_variant:Nn \tl_range:Nnn { c }
4931 \cs_new:Npn \tl_range:nnn { \__tl_range:Nnnn \__tl_range:w }
4932 \cs_new:Npn \__tl_range:Nnnn #1#2#3#4
4933  {
4934    \tl_head:f
4935    {
4936      \exp_args:Nf \__tl_range:nnnNn
4937      { \tl_count:n {#2} } {#3} {#4} #1 {#2}
4938    }
4939  }
4940 \cs_new:Npn \__tl_range:nnnNn #1#2#3

```

```

4941  {
4942    \exp_args:Nff \__tl_range:nnNn
4943    {
4944      \exp_args:Nf \__tl_range_normalize:nn
4945      { \int_eval:n { #2 - 1 } } {#1}
4946    }
4947    {
4948      \exp_args:Nf \__tl_range_normalize:nn
4949      { \int_eval:n {#3} } {#1}
4950    }
4951  }
4952 \cs_new:Npn \__tl_range:nnNn #1#2#3#4
4953  {
4954    \if_int_compare:w #2 > #1 \exp_stop_f: \else:
4955    \exp_after:wN { \exp_after:wN }
4956    \fi:
4957    \exp_after:wN #3
4958    \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
4959    \exp_after:wN { \exp:w \__tl_range_skip:w #1 ; { } #4 }
4960  }
4961 \cs_new:Npn \__tl_range_skip:w #1 ; #2
4962  {
4963    \if_int_compare:w #1 > 0 \exp_stop_f:
4964    \exp_after:wN \__tl_range_skip:w
4965    \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
4966  \else:
4967    \exp_after:wN \exp_end:
4968  \fi:
4969  }
4970 \cs_new:Npn \__tl_range:w #1 ; #2
4971  {
4972    \exp_args:Nf \__tl_range_collect:nn
4973    { \__tl_range_skip_spaces:n {#2} } {#1}
4974  }
4975 \cs_new:Npn \__tl_range_skip_spaces:n #1
4976  {
4977    \tl_if_head_is_space:nTF {#1}
4978    { \exp_args:Nf \__tl_range_skip_spaces:n {#1} }
4979    { { } #1 }
4980  }
4981 \cs_new:Npn \__tl_range_collect:nn #1#2
4982  {
4983    \int_compare:nNnTF {#2} = 0
4984    {#1}
4985    {
4986      \exp_args:No \tl_if_head_is_space:nTF { \use_none:n #1 }
4987      {
4988        \exp_args:Nf \__tl_range_collect:nn
4989        { \__tl_range_collect_space:nw #1 }
4990        {#2}
4991      }
4992      {
4993        \__tl_range_collect:ff
4994        {

```

```

4995     \exp_args:N \tl_if_head_is_N_type:nTF { \use_none:n #1 }
4996         { \__tl_range_collect_N:nN }
4997         { \__tl_range_collect_group:nn }
4998     #1
4999 }
5000     { \int_eval:n { #2 - 1 } }
5001 }
5002 }
5003 }
5004 \cs_new:Npn \__tl_range_collect_space:nw #1 ~ { { #1 ~ } }
5005 \cs_new:Npn \__tl_range_collect_N:nN #1#2 { { #1 #2 } }
5006 \cs_new:Npn \__tl_range_collect_group:nn #1#2 { { #1 {#2} } }
5007 \cs_generate_variant:Nn \__tl_range_collect:nn { ff }

```

(End definition for `\tl_range:Nnn` and others. These functions are documented on page 48.)

`\__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]`.

```

5008 \cs_new:Npn \__tl_range_normalize:nn #1#2
5009 {
5010     \int_eval:n
5011     {
5012         \if_int_compare:w #1 < 0 \exp_stop_f:
5013             \if_int_compare:w #1 < -#2 \exp_stop_f:
5014                 0
5015             \else:
5016                 #1 + #2 + 1
5017             \fi:
5018         \else:
5019             \if_int_compare:w #1 < #2 \exp_stop_f:
5020                 #1
5021             \else:
5022                 #2
5023             \fi:
5024         \fi:
5025     }
5026 }

```

(End definition for `\__tl_range_normalize:nn`.)

## 6.13 Viewing token lists

`\tl_show:N` Showing token list variables is done after checking that the variable is defined (see `\__kernel_register_show:N`).  
`\tl_show:c`  
`\tl_log:N`  
`\tl_log:c`  
`\__tl_show>NN`

```

5027 \cs_new_protected:Npn \tl_show:N { \__tl_show:NN \tl_show:n }
5028 \cs_generate_variant:Nn \tl_show:N { c }
5029 \cs_new_protected:Npn \tl_log:N { \__tl_show:NN \tl_log:n }
5030 \cs_generate_variant:Nn \tl_log:N { c }
5031 \cs_new_protected:Npn \__tl_show:NN #1#2
5032 {
5033     \__kernel_chk_defined:NT #2
5034     { \exp_args:Nx #1 { \token_to_str:N #2 = \exp_not:o {#2} } }

```

```
5035 }
```

(End definition for `\tl_show:N`, `\tl_log:N`, and `\_tl_show>NN`. These functions are documented on page 49.)

**\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.

```
5036 \cs_new_protected:Npn \tl_show:n #1
  { \iow_wrap:nnN { >~ \tl_to_str:n {#1} . } { } { } \_tl_show:n }
  \cs_new_protected:Npn \_tl_show:n #1
  {
    \tl_set:Nf \l__tl_internal_a_tl { \_tl_show:w #1 \q_stop }
    \_kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
    {
      \_kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
      {
        \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
        { \exp_after:wN \l__tl_internal_a_tl }
      }
    }
  }
\cs_new:Npn \_tl_show:w #1 > #2 . \q_stop {#2}
```

(End definition for `\tl_show:n`, `\_tl_show:n`, and `\_tl_show:w`. This function is documented on page 49.)

**\tl\_log:n** Logging is much easier, simply line-wrap. The `>~` and trailing period is there to match the output of `\tl_show:n`.

```
5051 \cs_new_protected:Npn \tl_log:n #1
  { \iow_wrap:nnN { >~ \tl_to_str:n {#1} . } { } { } \iow_log:n }
```

(End definition for `\tl_log:n`. This function is documented on page 49.)

## 6.14 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.

```
5053 \tl_new:N \g_tmpa_tl
  5054 \tl_new:N \g_tmpb_tl
```

(End definition for `\g_tmpa_tl` and `\g_tmpb_tl`. These variables are documented on page 50.)

**\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.

```
5055 \tl_new:N \l_tmpa_tl
  5056 \tl_new:N \l_tmpb_tl
```

(End definition for `\l_tmpa_t1` and `\l_tmpb_t1`. These variables are documented on page 50.)

5057 `</initex | package>`

## 7 l3str implementation

5058 `<*initex | package>`

5059 `<@@=str>`

### 7.1 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 hand.

`\str_use:c`

`\str_use:N`

`\str_clear:N`

`\str_clear:c`

`\str_gclear:N`

`\str_gclear:c`

`\str_clear_new:N`

`\str_clear_new:c`

`\str_gclear_new:N`

`\str_gclear_new:c`

`\str_set_eq>NN`

`\str_set_eq:cN`

`\str_set_eq:Nc`

`\str_set_eq:cc`

`\str_gset_eq>NN`

`\str_gset_eq:cN`

`\str_gset_eq:Nc`

`\str_gset_eq:cc`

`\str_concat:NNN`

`\str_concat:ccc`

`\str_gconcat:NNN`

`\str_gconcat:ccc`

5060 `\group_begin:`

5061   `\cs_set_protected:Npn \__str_tmp:n #1`

5062   `{`

5063     `\tl_if_blank:nF {#1}`

5064     `{`

5065       `\cs_new_eq:cc { str_ #1 :N } { tl_ #1 :N }`

5066       `\exp_args:Nc \cs_generate_variant:Nn { str_ #1 :N } { c }`

5067       `\__str_tmp:n`

5068     `}`

5069   `}`

5070   `\__str_tmp:n`

5071     `{ new }`

5072     `{ use }`

5073     `{ clear }`

5074     `{ gclear }`

5075     `{ clear_new }`

5076     `{ gclear_new }`

5077     `{ }`

5078 `\group_end:`

5079 `\cs_new_eq:NN \str_set_eq:NN \tl_set_eq:NN`

5080 `\cs_new_eq:NN \str_gset_eq:NN \tl_gset_eq:NN`

5081 `\cs_generate_variant:Nn \str_set_eq:NN { c , Nc , cc }`

5082 `\cs_generate_variant:Nn \str_gset_eq:NN { c , Nc , cc }`

5083 `\cs_new_eq:NN \str_concat:NNN \tl_concat:NNN`

5084 `\cs_new_eq:NN \str_gconcat:NNN \tl_gconcat:NNN`

5085 `\cs_generate_variant:Nn \str_concat:NNN { ccc }`

5086 `\cs_generate_variant:Nn \str_gconcat:NNN { ccc }`

(End definition for `\str_new:N` and others. These functions are documented on page 51.)

`\str_set:Nn` Simply convert the token list inputs to `<strings>`.

`\str_set:NV`

`\str_set:Nx`

`\str_set:cn`

`\str_set:cV`

`\str_set:cX`

`\str_gset:Nn`

`\str_gset:NV`

`\str_gset:Nx`

`\str_gset:cn`

`\str_gset:cV`

`\str_gset:cX`

`\str_const:Nn`

`\str_const:NV`

`\str_const:Nx`

`\str_const:cn`

`\str_const:cV`

`\str_const:cX`

`\str_put_left:Nn`

`\str_put_left:NV`

5087 `\group_begin:`

5088   `\cs_set_protected:Npn \__str_tmp:n #1`

5089   `{`

5090     `\tl_if_blank:nF {#1}`

5091     `{`

5092       `\cs_new_protected:cpx { str_ #1 :Nn } ##1##2`

5093       `{`

5094         `\exp_not:c { tl_ #1 :Nx } ##1`

5095         `{ \exp_not:N \tl_to_str:n {##2} }`

5096       `}`

```

5097         \cs_generate_variant:cn { str_ #1 :Nn } { NV , Nx , cn , cV , cx }
5098         \__str_tmp:n
5099     }
5100 }
5101 \__str_tmp:n
5102   { set }
5103   { gset }
5104   { const }
5105   { put_left }
5106   { gput_left }
5107   { put_right }
5108   { gput_right }
5109   {}
5110 \group_end:

```

(End definition for `\str_set:Nn` and others. These functions are documented on page 52.)

## 7.2 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 `\q_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.

```

\str_replace_all:Nnn
\str_replace_all:cnn
\str_greplace_all:Nnn
\str_greplace_all:cnn
\str_replace_once:Nnn
\str_replace_once:cnn
\str_greplace_once:Nnn
\str_greplace_once:cnn
\__str_replace>NNNN
\__str_replace_aux>NNNnnn
\__str_replace_next:w
\cs_new_protected:Npn \str_replace_once:Nnn
  { \__str_replace>NNNnn \prg_do_nothing: \tl_set:Nx }
\cs_new_protected:Npn \str_greplace_once:Nnn
  { \__str_replace>NNNnn \prg_do_nothing: \tl_gset:Nx }
\cs_new_protected:Npn \str_replace_all:Nnn
  { \__str_replace>NNNnn \__str_replace_next:w \tl_set:Nx }
\cs_new_protected:Npn \str_greplace_all:Nnn
  { \__str_replace>NNNnn \__str_replace_next:w \tl_gset:Nx }
\cs_generate_variant:Nn \str_replace_once:Nnn { c }
\cs_generate_variant:Nn \str_greplace_once:Nnn { c }
\cs_generate_variant:Nn \str_replace_all:Nnn { c }
\cs_generate_variant:Nn \str_greplace_all:Nnn { c }
\cs_new_protected:Npn \__str_replace>NNNnn #1#2#3#4#5
  {
    \tl_if_empty:nTF {#4}
    {
      \__kernel_msg_error:nnx { kernel } { empty-search-pattern } {#5}
    }
    {
      \use:x
      {
        \exp_not:n { \__str_replace_aux>NNNnnn #1 #2 #3 }
        { \tl_to_str:N #3 }
        { \tl_to_str:n {#4} } { \tl_to_str:n {#5} }
      }
    }
  }
}

```

```

5138 \cs_new_protected:Npn \__str_replace_aux:NNNnnn #1#2#3#4#5#6
5139  {
5140    \cs_set:Npn \__str_replace_next:w ##1 #5 { ##1 #6 #1 }
5141    #2 #3
5142    {
5143      \__str_replace_next:w
5144      #4
5145      \use_none_delimit_by_q_stop:w
5146      #5
5147      \q_stop
5148    }
5149  }
5150 \cs_new_eq:NN \__str_replace_next:w ?

```

(End definition for `\str_replace_all:Nnn` and others. These functions are documented on page 53.)

`\str_remove_once:Nn` Removal is just a special case of replacement.

```

5151 \cs_new_protected:Npn \str_remove_once:Nn #1#2
5152  { \str_replace_once:Nnn #1 {#2} { } }
5153 \cs_new_protected:Npn \str_gremove_once:Nn #1#2
5154  { \str_greplace_once:Nnn #1 {#2} { } }
5155 \cs_generate_variant:Nn \str_remove_once:Nn { c }
5156 \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 53.)

`\str_remove_all:Nn` Removal is just a special case of replacement.

```

5157 \cs_new_protected:Npn \str_remove_all:Nn #1#2
5158  { \str_replace_all:Nnn #1 {#2} { } }
5159 \cs_new_protected:Npn \str_gremove_all:Nn #1#2
5160  { \str_greplace_all:Nnn #1 {#2} { } }
5161 \cs_generate_variant:Nn \str_remove_all:Nn { c }
5162 \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 53.)

### 7.3 String comparisons

`\str_if_empty_p:N` More copy-paste!

```

5163 \prg_new_eq_conditional:NNn \str_if_exist:N \tl_if_exist:N
5164  { p , T , F , TF }
5165 \prg_new_eq_conditional:NNn \str_if_exist:c \tl_if_exist:c
5166  { p , T , F , TF }
5167 \prg_new_eq_conditional:NNn \str_if_empty:N \tl_if_empty:N
5168  { p , T , F , TF }
5169 \prg_new_eq_conditional:NNn \str_if_empty:c \tl_if_empty:c
5170  { p , T , F , TF }

```

(End definition for `\str_if_empty:NTF` and `\str_if_exist:NTF`. These functions are documented on page 54.)

\\_\\_str\\_if\\_eq:nn  
\\_\\_str\\_escape:n

String comparisons rely on the primitive `\(pdf)strcmp` if available: LuaTeX does not have it, so emulation is required. As the net result is that we do not *always* use the primitive, the correct approach is to wrap up in a function with defined behaviour. That's done by providing a wrapper and then redefining in the LuaTeX case. Note that the necessary Lua code is loaded in `l3bootstrap`. The need to detokenize and force expansion of input arises from the case where a # token is used in the input, *e.g.* `\_\_str_if_eq:nn {#} { \tl_to_str:n {#} }`, which otherwise would fail as `\tex_luaescapestring:D` does not double such tokens.

```

5171 \cs_new:Npn \_\_str_if_eq:nn #1#2 { \tex_strcmp:D {#1} {#2} }
5172 \cs_if_exist:NT \tex_luatexversion:D
5173   {
5174     \cs_set_eq:NN \lua_escape:e \tex_luaescapestring:D
5175     \cs_set_eq:NN \lua_now:e \tex_directlua:D
5176     \cs_set:Npn \_\_str_if_eq:nn #1#2
5177       {
5178         \lua_now:e
5179           {
5180             l3kernel.strptime
5181               (
5182                 " \_\_str_escape:n {#1} " ,
5183                 " \_\_str_escape:n {#2} "
5184               )
5185           }
5186       }
5187     \cs_new:Npn \_\_str_escape:n #1
5188       {
5189         \lua_escape:e
5190           { \_\_kernel_tl_to_str:w \use:e { {#1} } }
5191       }
5192   }

```

(End definition for `\_\_str_if_eq:nn` and `\_\_str_escape:n`.)

**\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 make life a bit clearer. The nn and xx versions are created directly as this is most efficient.

```

5193 \prg_new_conditional:Npnn \str_if_eq:nn #1#2 { p , T , F , TF }
5194   {
5195     \if_int_compare:w
5196       \_\_str_if_eq:nn { \exp_not:n {#1} } { \exp_not:n {#2} }
5197       = 0 \exp_stop_f:
5198       \prg_return_true: \else: \prg_return_false: \fi:
5199   }
5200 \prg_generate_conditional_variant:Nnn \str_if_eq:nn
5201   { V , v , o , nV , no , VV , nv } { p , T , F , TF }
5202 \prg_new_conditional:Npnn \str_if_eq:ee #1#2 { p , T , F , TF }
5203   {
5204     \if_int_compare:w \_\_str_if_eq:nn {#1} {#2} = 0 \exp_stop_f:
5205       \prg_return_true: \else: \prg_return_false: \fi:
5206   }

```

(End definition for `\str_if_eq:nnTF` and `\str_if_eq:eeTF`. These functions are documented on page 54.)

`\str_if_eq_p:NN` Note that `\str_if_eq:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.  
`\str_if_eq_p:Nc`

```

\str_if_eq_p:cN 5207 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
\str_if_eq_p:cc 5208 {
\str_if_eq:NNTF 5209   \if_int_compare:w
\str_if_eq:NcTF 5210     \_str_if_eq:nn { \tl_to_str:N #1 } { \tl_to_str:N #2 }
\str_if_eq:cNTF 5211     = 0 \exp_stop_f: \prg_return_true: \else: \prg_return_false: \fi:
\str_if_eq:ccTF 5212 }
\prg_generate_conditional_variant:Nnn \str_if_eq:NN
5213 { c , Nc , cc } { T , F , TF , p }
5214

```

(End definition for `\str_if_eq:NNTF`. This function is documented on page 54.)

`\str_if_in:NnTF` Everything here needs to be detokenized but beyond that it is a simple token list test.  
`\str_if_in:cNfTF` It would be faster to fine-tune the T, F, TF variants by calling the appropriate variant of  
`\str_if_in:nnTF` `\tl_if_in:nnTF` directly but that takes more code.

```

5215 \prg_new_protected_conditional:Npnn \str_if_in:Nn #1#2 { T , F , TF }
5216 {
5217     \use:x
5218     { \tl_if_in:nnTF { \tl_to_str:N #1 } { \tl_to_str:n {#2} } }
5219     { \prg_return_true: } { \prg_return_false: }
5220 }
5221 \prg_generate_conditional_variant:Nnn \str_if_in:Nn
5222 { c } { T , F , TF }
5223 \prg_new_protected_conditional:Npnn \str_if_in:nn #1#2 { T , F , TF }
5224 {
5225     \use:x
5226     { \tl_if_in:nnTF { \tl_to_str:n {#1} } { \tl_to_str:n {#2} } }
5227     { \prg_return_true: } { \prg_return_false: }
5228 }

```

(End definition for `\str_if_in:NnTF` and `\str_if_in:nnTF`. These functions are documented on page 54.)

`\str_case:nn` Much the same as `\tl_case:nn(TF)` here: just a change in the internal comparison.

```
\str_case:on      5229 \cs_new:Npn \str_case:nn #1#2
\str_case:nV      5230 {
\str_case:nnTF    5231   \exp:w
\str_case:onTF    5232     \_\str_case:nnTF {#1} {#2} { } { }
\str_case:nvTF    5233 }
\str_case:nnT #1#2#3
\str_case:nvTF    5234 \cs_new:Npn \str_case:nnT #1#2#3
\str_case:nnTF    5235 {
\str_case_e:nn     5236   \exp:w
\str_case_e:nnTF   5237     \_\str_case:nnTF {#1} {#2} {#3} { }
\_\str_case:nnTF   5238 }
\cs_new:Npn \str_case:nnF #1#2
\str_case_e:mnTF  5239 {
\str_case_e:nw     5240   \exp:w
\str_case_e:nw     5241     \_\str_case:nnTF {#1} {#2} { }
\str_case_end:nw   5242 }
\cs_new:Npn \str_case:nnTF #1#2
\str_case_end:nw   5243 {
\str_case_end:nw   5244   \exp:w
\str_case_end:nw   5245     \_\str_case:nnTF {#1} {#2}
\str_case_end:nw   5246 }
```

```

5248     }
5249     \cs_new:Npn \__str_case:nnTF #1#2#3#4
5250     { \__str_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5251     \cs_generate_variant:Nn \str_case:nn { o , nv , nv }
5252     \prg_generate_conditional_variant:Nnn \str_case:nn
5253     { o , nv , nv } { T , F , TF }
5254     \cs_new:Npn \__str_case:nw #1#2#3
5255     {
5256         \str_if_eq:nnTF {#1} {#2}
5257         { \__str_case_end:nw {#3} }
5258         { \__str_case:nw {#1} }
5259     }
5260     \cs_new:Npn \str_case_e:nn #1#2
5261     {
5262         \exp:w
5263         \__str_case_e:nnTF {#1} {#2} { } { }
5264     }
5265     \cs_new:Npn \str_case_e:nnT #1#2#3
5266     {
5267         \exp:w
5268         \__str_case_e:nnTF {#1} {#2} {#3} { }
5269     }
5270     \cs_new:Npn \str_case_e:nnF #1#2
5271     {
5272         \exp:w
5273         \__str_case_e:nnTF {#1} {#2} { }
5274     }
5275     \cs_new:Npn \str_case_e:nnTF #1#2
5276     {
5277         \exp:w
5278         \__str_case_e:nnTF {#1} {#2}
5279     }
5280     \cs_new:Npn \__str_case_e:nnTF #1#2#3#4
5281     { \__str_case_e:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
5282     \cs_new:Npn \__str_case_e:nw #1#2#3
5283     {
5284         \str_if_eq:eeTF {#1} {#2}
5285         { \__str_case_end:nw {#3} }
5286         { \__str_case_e:nw {#1} }
5287     }
5288     \cs_new:Npn \__str_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
5289     { \exp_end: #1 #4 }

```

(End definition for `\str_case:nnTF` and others. These functions are documented on page 55.)

## 7.4 Mapping to strings

`\str_map_function:NN` 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 `\str_map_variable>NNn`

`\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`

deal with the next space. The space before the braced space allows to optimize the \q\_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\_recursion\_tail (also by converting 9 spaces at a time in the \str\_map\_function:nN case).

```

5290 \cs_new:Npn \str_map_function:nN #1#2
5291 {
5292     \exp_after:wN \__str_map_function:w
5293     \exp_after:wN \__str_map_function:Nn \exp_after:wn #2
5294         \_kernel_tl_to_str:w {#1}
5295         \q_recursion_tail ? ~
5296     \prg_break_point:Nn \str_map_break: { }
5297 }
5298 \cs_new:Npn \str_map_function>NN
5299     { \exp_args:No \str_map_function:nN }
5300 \cs_new:Npn \__str_map_function:w #1 ~
5301     { #1 { ~ { ~ } \__str_map_function:w } }
5302 \cs_new:Npn \__str_map_function:Nn #1#2
5303 {
5304     \if_meaning:w \q_recursion_tail #2
5305         \exp_after:wN \str_map_break:
5306     \fi:
5307     #1 #2 \__str_map_function:Nn #1
5308 }
5309 \cs_generate_variant:Nn \str_map_function>NN { c }
5310 \cs_new_protected:Npn \str_map_inline:nn #1#2
5311 {
5312     \int_gincr:N \g__kernel_prg_map_int
5313     \cs_gset_protected:cpn
5314         { __str_map_ \int_use:N \g__kernel_prg_map_int :w } ##1 {#2}
5315     \use:x
5316     {
5317         \exp_not:N \__str_map_inline>NN
5318         \exp_not:c { __str_map_ \int_use:N \g__kernel_prg_map_int :w }
5319             \__kernel_str_to_other_fast:n {#1}
5320     }
5321     \q_recursion_tail
5322     \prg_break_point:Nn \str_map_break:
5323         { \int_gdecr:N \g__kernel_prg_map_int }
5324 }
5325 \cs_new_protected:Npn \str_map_inline:Nn
5326     { \exp_args:No \str_map_inline:nn }
5327 \cs_generate_variant:Nn \str_map_inline:Nn { c }
5328 \cs_new:Npn \__str_map_inline>NN #1#2
5329 {
5330     \quark_if_recursion_tail_break:NN #2 \str_map_break:
5331     \exp_args:No #1 { \token_to_str:N #2 }
5332     \__str_map_inline>NN #1
5333 }
5334 \cs_new_protected:Npn \str_map_variable:nNn #1#2#3
5335 {
5336     \use:x

```

```

5337 {
5338   \exp_not:n { \__str_map_variable:NnN #2 {#3} }
5339   \__kernel_str_to_other_fast:n {#1}
5340 }
5341 \q_recursion_tail
5342 \prg_break_point:Nn \str_map_break: { }
5343 }
5344 \cs_new_protected:Npn \str_map_variable:NNn
5345   { \exp_args:No \str_map_variable:nNn }
5346 \cs_new_protected:Npn \__str_map_variable:NnN #1#2#3
5347 {
5348   \quark_if_recursion_tail_break:NN #3 \str_map_break:
5349   \str_set:Nn #1 {#3}
5350   \use:n {#2}
5351   \__str_map_variable:NnN #1 {#2}
5352 }
5353 \cs_generate_variant:Nn \str_map_variable:NNn { c }
5354 \cs_new:Npn \str_map_break:
5355   { \prg_map_break:Nn \str_map_break: { } }
5356 \cs_new:Npn \str_map_break:n
5357   { \prg_map_break:Nn \str_map_break: }

```

(End definition for `\str_map_function:NN` and others. These functions are documented on page 55.)

## 7.5 Accessing specific characters in a string

`\__kernel_str_to_other:n`: 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 `\q_mark` and `\q_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 `\q_mark` and the first A (well, there is also the need to remove a space).

`\__str_to_other_loop:w`: The converted part of the string between the `\q_mark` and `\q_stop` markers.

`\__str_to_other_end:w`: Finds the result between `\q_mark` and the first A (well, there is also the need to remove a space).

```

5358 \cs_new:Npn \__kernel_str_to_other:n #1
5359 {
5360   \exp_after:wN \__str_to_other_loop:w
5361   \tl_to_str:n {#1} ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
5362 }
5363 \group_begin:
5364 \tex_lccode:D ‘\* = ‘\ %
5365 \tex_lccode:D ‘\A = ‘\A %
5366 \tex_lowercase:D
5367 {
5368   \group_end:
5369   \cs_new:Npn \__str_to_other_loop:w
5370     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
5371   {
5372     \if_meaning:w A #8
5373       \__str_to_other_end:w
5374     \fi:
5375     \__str_to_other_loop:w
5376       #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
5377   }
5378 \cs_new:Npn \__str_to_other_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
5379   { \fi: #2 }

```

```
5380 }
```

(End definition for `\_kernel_str_to_other:n`, `\_str_to_other_loop:w`, and `\_str_to_other_end:w`.)

```
\_kernel_str_to_other_fast:n  
\_kernel_str_to_other_fast_loop:w  
\_str_to_other_fast_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.

```
5381 \cs_new:Npn \_kernel_str_to_other_fast:n #1  
5382 {  
5383     \exp_after:wN \_str_to_other_fast_loop:w \tl_to_str:n {#1} ~  
5384     A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_stop  
5385 }  
5386 \group_begin:  
5387 \tex_lccode:D ‘\* = ‘\ %  
5388 \tex_lccode:D ‘\A = ‘\A %  
5389 \tex_lowercase:D  
5390 {  
5391     \group_end:  
5392     \cs_new:Npn \_str_to_other_fast_loop:w  
5393     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~  
5394     {  
5395         \if_meaning:w A #9  
5396             \_str_to_other_fast_end:w  
5397         \fi:  
5398         #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9  
5399         \_str_to_other_fast_loop:w *  
5400     }  
5401     \cs_new:Npn \_str_to_other_fast_end:w #1 * A #2 \q_stop {#1}  
5402 }
```

(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.

```
5403 \cs_new:Npn \str_item:Nn { \exp_args:No \str_item:nn }  
5404 \cs_generate_variant:Nn \str_item:Nn { c }  
5405 \cs_new:Npn \str_item:nn #1#2  
5406 {  
5407     \exp_args:Nf \tl_to_str:n  
5408     {  
5409         \exp_args:Nf \_str_item:nn  
5410         { \_kernel_str_to_other:n {#1} } {#2}  
5411     }  
5412 }
```

```

5413 \cs_new:Npn \str_item_ignore_spaces:nn #1
5414   { \exp_args:No \__str_item:nn { \tl_to_str:n {#1} } }
5415 \cs_new:Npn \__str_item:nn #1#2
5416   {
5417     \exp_after:wN \__str_item:w
5418     \int_value:w \int_eval:n {#2} \exp_after:wN ;
5419     \int_value:w \__str_count:n {#1} ;
5420     #1 \q_stop
5421   }
5422 \cs_new:Npn \__str_item:w #1; #2;
5423   {
5424     \int_compare:nNnTF {#1} < 0
5425     {
5426       \int_compare:nNnTF {#1} < {-#2}
5427         { \use_none_delimit_by_q_stop:w }
5428         {
5429           \exp_after:wN \use_i_delimit_by_q_stop:nw
5430           \exp:w \exp_after:wN \__str_skip_exp_end:w
5431             \int_value:w \int_eval:n { #1 + #2 } ;
5432         }
5433     }
5434   {
5435     \int_compare:nNnTF {#1} > {#2}
5436     { \use_none_delimit_by_q_stop:w }
5437     {
5438       \exp_after:wN \use_i_delimit_by_q_stop:nw
5439       \exp:w \__str_skip_exp_end:w #1 ; { }
5440     }
5441   }
5442 }

```

(End definition for `\str_item:Nn` and others. These functions are documented on page 58.)

```

\__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).

```

5443 \cs_new:Npn \__str_skip_exp_end:w #1;
5444   {
5445     \if_int_compare:w #1 > 8 \exp_stop_f:
5446       \exp_after:wN \__str_skip_loop:wNNNNNNNN
5447     \else:
5448       \exp_after:wN \__str_skip_end:w
5449       \int_value:w \int_eval:w
5450       \fi:
5451       #1 ;
5452   }
5453 \cs_new:Npn \__str_skip_loop:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
5454   {

```

```

5455     \exp_after:wN \__str_skip_exp_end:w
5456     \int_value:w \int_eval:n { #1 - 8 } ;
5457   }
5458 \cs_new:Npn \__str_skip_end:w #1 ;
5459   {
5460     \exp_after:wN \__str_skip_end:NNNNNNNN
5461     \if_case:w #1 \exp_stop_f: \or: \or: \or: \or: \or: \or: \or:
5462   }
5463 \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.

```

5464 \cs_new:Npn \str_range:Nnn { \exp_args:No \str_range:nnn }
5465 \cs_generate_variant:Nn \str_range:Nnn { c }
5466 \cs_new:Npn \str_range:nnn #1#2#3
5467   {
5468     \exp_args:Nf \tl_to_str:n
5469     {
5470       \exp_args:Nf \__str_range:nnn
5471       { \__kernel_str_to_other:n {#1} } {#2} {#3}
5472     }
5473   }
5474 \cs_new:Npn \str_range_ignore_spaces:nnn #1
5475   { \exp_args:No \__str_range:nnn { \tl_to_str:n {#1} } }
5476 \cs_new:Npn \__str_range:nnn #1#2#3
5477   {
5478     \exp_after:wN \__str_range:w
5479     \int_value:w \__str_count:n {#1} \exp_after:wN ;
5480     \int_value:w \int_eval:n { (#2) - 1 } \exp_after:wN ;
5481     \int_value:w \int_eval:n {#3} ;
5482     #1 \q_stop
5483   }
5484 \cs_new:Npn \__str_range:w #1; #2; #3;
5485   {
5486     \exp_args:Nf \__str_range:nnw
5487     { \__str_range_normalize:nn {#2} {#1} }
5488     { \__str_range_normalize:nn {#3} {#1} }
5489   }
5490 \cs_new:Npn \__str_range:nnw #1#2
5491   {
5492     \exp_after:wN \__str_collect_delimit_by_q_stop:w
5493     \int_value:w \int_eval:n { #2 - #1 } \exp_after:wN ;
5494     \exp:w \__str_skip_exp_end:w #1 ;
5495   }

```

(End definition for \str\_range:Nnn and others. These functions are documented on page 59.)

\\_\_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  $[0, \#2]$ .

```

5496 \cs_new:Npn \__str_range_normalize:nn #1#2
5497 {
5498     \int_eval:n
5499     {
5500         \if_int_compare:w #1 < 0 \exp_stop_f:
5501         \if_int_compare:w #1 < -#2 \exp_stop_f:
5502             0
5503         \else:
5504             #1 + #2 + 1
5505         \fi:
5506     \else:
5507         \if_int_compare:w #1 < #2 \exp_stop_f:
5508             #1
5509         \else:
5510             #2
5511         \fi:
5512     \fi:
5513 }
5514 }
```

(End definition for `\__str_range_normalize:nn`.)

`\__str_collect_delimit_by_q_stop:w`  
`\__str_collect_loop:wn`  
`\__str_collect_loop:wnNNNNNNN`  
`\__str_collect_end:wn`  
`\__str_collect_end:nnnnnnnnw`

Collects `max(#1, 0)` characters, and removes everything else until `\q_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:nnnnnnnnw` 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.

```

5515 \cs_new:Npn \__str_collect_delimit_by_q_stop:w #1;
5516     { \__str_collect_loop:wn #1 ; { } }
5517 \cs_new:Npn \__str_collect_loop:wn #1 ;
5518     {
5519         \if_int_compare:w #1 > 7 \exp_stop_f:
5520             \exp_after:wN \__str_collect_loop:wnNNNNNNN
5521         \else:
5522             \exp_after:wN \__str_collect_end:wn
5523         \fi:
5524         #1 ;
5525     }
5526 \cs_new:Npn \__str_collect_loop:wnNNNNNNN #1; #2 #3#4#5#6#7#8#9
5527     {
5528         \exp_after:wN \__str_collect_loop:wn
5529         \int_value:w \int_eval:n { #1 - 7 } ;
5530         { #2 #3#4#5#6#7#8#9 }
5531     }
5532 \cs_new:Npn \__str_collect_end:wn #1 ;
5533     {
5534         \exp_after:wN \__str_collect_end:nnnnnnnnw
5535         \if_case:w \if_int_compare:w #1 > 0 \exp_stop_f:
5536             #1 \else: 0 \fi: \exp_stop_f:
5537             \or: \or: \or: \or: \or: \or: \or: \fi:
5538     }
```

```

5539 \cs_new:Npn \__str_collect_end:nnnnnnnnw #1#2#3#4#5#6#7#8 #9 \q_stop
5540   { #1#2#3#4#5#6#7#8 }

```

(End definition for `\__str_collect_delimit_by_q_stop:w` and others.)

## 7.6 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  $X\langle number\rangle$ , and that  $\langle number\rangle$  is added to the sum of 9 that precedes, to adjust the result.

```

5541 \cs_new:Npn \str_count_spaces:N
5542   { \exp_args:No \str_count_spaces:n }
5543 \cs_generate_variant:Nn \str_count_spaces:N { c }
5544 \cs_new:Npn \str_count_spaces:n #1
5545   {
5546     \int_eval:n
5547     {
5548       \exp_after:wN \__str_count_spaces_loop:w
5549       \tl_to_str:n {#1} ~
5550       X 7 ~ X 6 ~ X 5 ~ X 4 ~ X 3 ~ X 2 ~ X 1 ~ X 0 ~ X -1 ~
5551       \q_stop
5552     }
5553   }
5554 \cs_new:Npn \__str_count_spaces_loop:w #1~#2~#3~#4~#5~#6~#7~#8~#9~
5555   {
5556     \if_meaning:w X #9
5557       \use_i_delimit_by_q_stop:nw
5558     \fi:
5559     9 + \__str_count_spaces_loop:w
5560   }

```

(End definition for `\str_count_spaces:N`, `\str_count_spaces:n`, and `\__str_count_spaces_loop:w`. These functions are documented on page 57.)

```

\str_count:N
\str_count:c
\str_count:n
\str_count_ignore_spaces:n
\__str_count:n
\__str_count_aux:n
\__str_count_loop:NNNNNNNNNN

```

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.

```

5561 \cs_new:Npn \str_count:N { \exp_args:No \str_count:n }
5562 \cs_generate_variant:Nn \str_count:N { c }
5563 \cs_new:Npn \str_count:n #1
5564   {
5565     \__str_count_aux:n
5566     {
5567       \str_count_spaces:n {#1}
5568       + \exp_after:wN \__str_count_loop:NNNNNNNNNN \tl_to_str:n {#1}

```

```

5569     }
5570   }
5571 \cs_new:Npn \__str_count:n #1
5572   {
5573     \__str_count_aux:n
5574       { \__str_count_loop:NNNNNNNNN #1 }
5575   }
5576 \cs_new:Npn \str_count_ignore_spaces:n #1
5577   {
5578     \__str_count_aux:n
5579       { \exp_after:wN \__str_count_loop:NNNNNNNNN \tl_to_str:n {#1} }
5580   }
5581 \cs_new:Npn \__str_count_aux:n #1
5582   {
5583     \int_eval:n
5584       {
5585         #1
5586         { X 8 } { X 7 } { X 6 }
5587         { X 5 } { X 4 } { X 3 }
5588         { X 2 } { X 1 } { X 0 }
5589         \q_stop
5590       }
5591   }
5592 \cs_new:Npn \__str_count_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
5593   {
5594     \if_meaning:w X #9
5595       \exp_after:wN \use_none_delimit_by_q_stop:w
5596     \fi:
5597     9 + \__str_count_loop:NNNNNNNNN
5598   }

```

(End definition for `\str_count:N` and others. These functions are documented on page 57.)

## 7.7 The first character in a string

`\str_head:N` The `_ignore_spaces` variant applies `\tl_to_str:n` then grabs the first item, thus skipping spaces. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`.  
`\str_head:c` To circumvent the fact that TeX skips spaces when grabbing undelimited macro parameters, `\__str_head:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `\use_i_delimit_by_q_stop:nw` leaves that in the input stream.  
`\str_head:n` On the other hand, if `#1` starts with a space, the `\__str_head:w` takes an empty argument, and the single (initially braced) space in the definition of `\__str_head:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `\use_i_delimit_by_q_stop:nw`.

```

5599 \cs_new:Npn \str_head:N { \exp_args:No \str_head:n }
5600 \cs_generate_variant:Nn \str_head:N { c }
5601 \cs_new:Npn \str_head:n #1
5602   {
5603     \exp_after:wN \__str_head:w
5604     \tl_to_str:n {#1}
5605     { { } } ~ \q_stop
5606   }

```

```

5607 \cs_new:Npn \__str_head:w #1 ~ %
5608   { \use_i_delimit_by_q_stop:nw #1 { ~ } }
5609 \cs_new:Npn \str_head_ignore_spaces:n #1
5610   {
5611     \exp_after:wN \use_i_delimit_by_q_stop:nw
5612     \tl_to_str:n {#1} { } \q_stop
5613   }

```

(End definition for `\str_head:N` and others. These functions are documented on page 58.)

`\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 `\q_mark`. One can check that an empty (or blank) string yields an empty tail.

```

5614 \cs_new:Npn \str_tail:N { \exp_args:No \str_tail:n }
5615 \cs_generate_variant:Nn \str_tail:N { c }
5616 \cs_new:Npn \str_tail:n #1
5617   {
5618     \exp_after:wN \__str_tail_auxi:w
5619     \reverse_if:N \if_charcode:w
5620     \scan_stop: \tl_to_str:n {#1} X X \q_stop
5621   }
5622 \cs_new:Npn \__str_tail_auxi:w #1 X #2 \q_stop { \fi: #1 }
5623 \cs_new:Npn \str_tail_ignore_spaces:n #1
5624   {
5625     \exp_after:wN \__str_tail_auxii:w
5626     \tl_to_str:n {#1} \q_mark \q_mark \q_stop
5627   }
5628 \cs_new:Npn \__str_tail_auxii:w #1 #2 \q_mark #3 \q_stop { #2 }

```

(End definition for `\str_tail:N` and others. These functions are documented on page 58.)

## 7.8 String manipulation

`\str_fold_case: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.

`\str_fold_case:V`

`\str_lower_case:n`

`\str_lower_case:f`

`\str_upper_case:n`

`\str_upper_case:f`

`\__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`

```

5629 \cs_new:Npn \str_fold_case:n #1 { \__str_change_case:nn {#1} { fold } }
5630 \cs_new:Npn \str_lower_case:n #1 { \__str_change_case:nn {#1} { lower } }
5631 \cs_new:Npn \str_upper_case:n #1 { \__str_change_case:nn {#1} { upper } }
5632 \cs_generate_variant:Nn \str_fold_case:n { V }
5633 \cs_generate_variant:Nn \str_lower_case:n { f }
5634 \cs_generate_variant:Nn \str_upper_case:n { f }
5635 \cs_new:Npn \__str_change_case:nn #1
5636   {
5637     \exp_after:wN \__str_change_case_aux:nn \exp_after:wN
5638       { \tl_to_str:n {#1} }

```

```

5639     }
5640 \cs_new:Npn \__str_change_case_aux:nn #1#2
5641 {
5642     \__str_change_case_loop:nw {#2} #1 \q_recursion_tail \q_recursion_stop
5643     \__str_change_case_result:n {}
5644 }
5645 \cs_new:Npn \__str_change_case_output:nw #1#2 \__str_change_case_result:n #3
5646 { #2 \__str_change_case_result:n { #3 #1 } }
5647 \cs_generate_variant:Nn \__str_change_case_output:nw { f }
5648 \cs_new:Npn \__str_change_case_end:wn #1 \__str_change_case_result:n #2
5649 { \tl_to_str:n {#2} }
5650 \cs_new:Npn \__str_change_case_loop:nw #1#2 \q_recursion_stop
5651 {
5652     \tl_if_head_is_space:nTF {#2}
5653     { \__str_change_case_space:n }
5654     { \__str_change_case_char:nN }
5655     {#1} #2 \q_recursion_stop
5656 }
5657 \exp_last_unbraced:NNNNo
5658 \cs_new:Npn \__str_change_case_space:n #1 \c_space_tl
5659 {
5660     \__str_change_case_output:nw { ~ }
5661     \__str_change_case_loop:nw {#1}
5662 }
5663 \cs_new:Npn \__str_change_case_char:nN #1#2
5664 {
5665     \quark_if_recursion_tail_stop_do:Nn #2
5666     { \__str_change_case_end:wn }
5667     \__str_change_case_output:fw
5668     { \use:c { char_ #1 _case:N } #2 }
5669     \__str_change_case_loop:nw {#1}
5670 }

```

(End definition for `\str_fold_case:n` and others. These functions are documented on page 61.)

`\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`

```

5671 \str_const:Nx \c_ampersand_str { \cs_to_str:N \& }
5672 \str_const:Nx \c_atsign_str { \cs_to_str:N \@ }
5673 \str_const:Nx \c_backslash_str { \cs_to_str:N \\ }
5674 \str_const:Nx \c_left_brace_str { \cs_to_str:N \{ }
5675 \str_const:Nx \c_right_brace_str { \cs_to_str:N \} }
5676 \str_const:Nx \c_circumflex_str { \cs_to_str:N \^ }
5677 \str_const:Nx \c_colon_str { \cs_to_str:N \: }
5678 \str_const:Nx \c_dollar_str { \cs_to_str:N \$ }
5679 \str_const:Nx \c_hash_str { \cs_to_str:N \# }
5680 \str_const:Nx \c_percent_str { \cs_to_str:N \% }
5681 \str_const:Nx \c_tilde_str { \cs_to_str:N \~ }
5682 \str_const:Nx \c_underscore_str { \cs_to_str:N \_ }

```

(End definition for `\c_ampersand_str` and others. These variables are documented on page 62.)

`\l_tmpa_str`    Scratch strings.  
`\l_tmpb_str`    `\str_new:N \l_tmpa_str`  
`\g_tmpa_str`  
`\g_tmpb_str`

```

5684 \str_new:N \l_tmpb_str
5685 \str_new:N \g_tmpa_str
5686 \str_new:N \g_tmpb_str

```

(End definition for `\l_tmpa_str` and others. These variables are documented on page 62.)

## 7.9 Viewing strings

`\str_show:n`

Displays a string on the terminal.

```

5687 \cs_new_eq:NN \str_show:n \tl_show:n
5688 \cs_new_eq:NN \str_show:N \tl_show:N
5689 \cs_generate_variant:Nn \str_show:N { c }

```

(End definition for `\str_show:n` and `\str_show:N`. These functions are documented on page 61.)

## 7.10 Deprecated functions

`\str_case_x:nn`

For removal after 2019-12-31.

```

5690 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_case_e:nn }
5691 \cs_new:Npn \str_case_x:nn { \str_case_e:nn }
5692 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_case_e:nnT }
5693 \cs_new:Npn \str_case_x:nnT { \str_case_e:nnT }
5694 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_case_e:nnF }
5695 \cs_new:Npn \str_case_x:nnF { \str_case_e:nnF }
5696 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_case_e:nnTF }
5697 \cs_new:Npn \str_case_x:nnTF { \str_case_e:nnTF }
5698 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_if_eq_p:ee }
5699 \cs_new:Npn \str_if_eq_x_p:nn { \str_if_eq_p:ee }
5700 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_if_eq:eeT }
5701 \cs_new:Npn \str_if_eq_x:nnT { \str_if_eq:eeT }
5702 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_if_eq:eeF }
5703 \cs_new:Npn \str_if_eq_x:nnF { \str_if_eq:eeF }
5704 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \str_if_eq:eeTF }
5705 \cs_new:Npn \str_if_eq_x:nnTF { \str_if_eq:eeTF }

```

(End definition for `\str_case_x:nnTF` and `\str_if_eq_x:nnTF`.)

5706 ⟨/initex | package⟩

# 8 I3quark implementation

The following test files are used for this code: `m3quark001.lvt`.

5707 ⟨\*initex | package⟩

## 8.1 Quarks

5708 ⟨@=quark⟩

`\quark_new:N` Allocate a new quark.

```

5709 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN q #1 } { }
5710 \cs_new_protected:Npn \quark_new:N #1
5711 {
5712     \__kernel_chk_if_free_cs:N #1
5713     \cs_gset_nopar:Npn #1 {#1}
5714 }

```

(End definition for \quark\_new:N. This function is documented on page 63.)

\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.

```
5715 \quark_new:N \q_nil
5716 \quark_new:N \q_mark
5717 \quark_new:N \q_no_value
5718 \quark_new:N \q_stop
```

(End definition for \q\_nil and others. These variables are documented on page 64.)

\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.

```
5719 \quark_new:N \q_recursion_tail
5720 \quark_new:N \q_recursion_stop
```

(End definition for \q\_recursion\_tail and \q\_recursion\_stop. These variables are documented on page 64.)

\quark\_if\_recursion\_tail\_stop:N 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.

```
5721 \cs_new:Npn \quark_if_recursion_tail_stop:N #1
5722 {
5723     \if_meaning:w \q_recursion_tail #1
5724         \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
5725     \fi:
5726 }
5727 \cs_new:Npn \quark_if_recursion_tail_stop_do:Nn #1
5728 {
5729     \if_meaning:w \q_recursion_tail #1
5730         \exp_after:wN \use_i_delimit_by_q_recursion_stop:nw
5731     \else:
5732         \exp_after:wN \use_none:n
5733     \fi:
5734 }
```

(End definition for \quark\_if\_recursion\_tail\_stop:N and \quark\_if\_recursion\_tail\_stop\_do:Nn. These functions are documented on page 65.)

\quark\_if\_recursion\_tail\_stop:n 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.

```
5735 \cs_new:Npn \quark_if_recursion_tail_stop:n #1
5736 {
5737     \tl_if_empty:cTF
5738         { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
5739         { \use_none_delimit_by_q_recursion_stop:w }
5740     { }
```

```

5741   }
5742 \cs_new:Npn \quark_if_recursion_tail_stop_do:nn #1
5743 {
5744   \tl_if_empty:cTF
5745   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
5746   { \use_i_delimit_by_q_recursion_stop:nw }
5747   { \use_none:n }
5748 }
5749 \cs_new:Npn \__quark_if_recursion_tail:w
5750   #1 \q_recursion_tail #2 ? #3 ?! { #1 #2 }
5751 \cs_generate_variant:Nn \quark_if_recursion_tail_stop:n { o }
5752 \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 65.)

`\quark_if_recursion_tail_break:NN`  
`\quark_if_recursion_tail_break:nN`

Analogues of the `\quark_if_recursion_tail_stop...` functions. Break the mapping using #2.

```

5753 \cs_new:Npn \quark_if_recursion_tail_break:NN #1#2
5754 {
5755   \if_meaning:w \q_recursion_tail #1
5756   \exp_after:wN #2
5757   \fi:
5758 }
5759 \cs_new:Npn \quark_if_recursion_tail_break:nN #1#2
5760 {
5761   \tl_if_empty:cT
5762   { \__quark_if_recursion_tail:w {} #1 {} ?! \q_recursion_tail ??! }
5763   {#2}
5764 }

```

(End definition for `\quark_if_recursion_tail_break:NN` and `\quark_if_recursion_tail_break:nN`. These functions are documented on page 65.)

`\quark_if_nil_p:N`  
`\quark_if_nil:NTF`  
`\quark_if_no_value_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>7</sup>

```

5765 \prg_new_conditional:Npnn \quark_if_nil:N #1 { p, T , F , TF }
5766 {
5767   \if_meaning:w \q_nil #1
5768   \prg_return_true:
5769   \else:
5770   \prg_return_false:
5771   \fi:
5772 }
5773 \prg_new_conditional:Npnn \quark_if_no_value:N #1 { p, T , F , TF }
5774 {
5775   \if_meaning:w \q_no_value #1
5776   \prg_return_true:
5777   \else:
5778   \prg_return_false:
5779   \fi:
580 }

```

---

<sup>7</sup>It may still loop in special circumstances however!

```

5781 \prg_generate_conditional_variant:Nnn \quark_if_no_value:N
5782   { c } { p , T , F , TF }

```

(End definition for `\quark_if_nil:nTF` and `\quark_if_no_value:NTF`. These functions are documented on page 64.)

`\quark_if_nil_p:n` Let us explain `\quark_if_nil:n(TF)`. 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 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.

```

5783 \prg_new_conditional:Npnn \quark_if_nil:n #1 { p , T , F , TF }
5784   {
5785     \_quark_if_empty_if:o
5786       { \_quark_if_nil:w {} #1 {} ? ! \q_nil ? ? ! }
5787       \prg_return_true:
5788     \else:
5789       \prg_return_false:
5790     \fi:
5791   }
5792 \cs_new:Npn \_quark_if_nil:w #1 \q_nil #2 ? #3 ? ! { #1 #2 }
5793 \prg_new_conditional:Npnn \quark_if_no_value:n #1 { p , T , F , TF }
5794   {
5795     \_quark_if_empty_if:o
5796       { \_quark_if_no_value:w {} #1 {} ? ! \q_no_value ? ? ! }
5797       \prg_return_true:
5798     \else:
5799       \prg_return_false:
5800     \fi:
5801   }
5802 \cs_new:Npn \_quark_if_no_value:w #1 \q_no_value #2 ? #3 ? ! { #1 #2 }
5803 \prg_generate_conditional_variant:Nnn \quark_if_nil:n
5804   { V , o } { p , TF , T , F }
5805 \cs_new:Npn \_quark_if_empty_if:o #1
5806   {
5807     \exp_after:wN \if_meaning:w \exp_after:wN \q_nil
5808       \_kernel_tl_to_str:w \exp_after:wN {#1} \q_nil
5809   }

```

(End definition for `\quark_if_nil:nTF` and others. These functions are documented on page 64.)

## 8.2 Scan marks

```

5810 <@=scan>

```

`\g__scan_marks_tl` The list of all scan marks currently declared.

```

5811 \tl_new:N \g__scan_marks_tl

```

(End definition for `\g__scan_marks_tl`.)

**\scan\_new:N** Check whether the variable is already a scan mark, then declare it to be equal to `\scan_stop:` globally.

```
5812 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN s #1 } { }
5813 \cs_new_protected:Npn \scan_new:N #1
5814 {
5815     \tl_if_in:NnTF \g__scan_marks_tl { #1 }
5816     {
5817         \__kernel_msg_error:nnx { kernel } { scanmark-already-defined }
5818         { \token_to_str:N #1 }
5819     }
5820     {
5821         \tl_gput_right:Nn \g__scan_marks_tl {#1}
5822         \cs_new_eq:NN #1 \scan_stop:
5823     }
5824 }
```

(End definition for `\scan_new:N`. This function is documented on page 66.)

**\s\_stop** We only declare one scan mark here, more can be defined by specific modules.

```
5825 \scan_new:N \s_stop
```

(End definition for `\s_stop`. This variable is documented on page 66.)

**\use\_none\_delimit\_by\_s\_stop:w** Similar to `\use_none_delimit_by_q_stop:w`.

```
5826 \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 67.)

```
5827 ⟨/initex | package⟩
```

## 9 **I3seq** implementation

The following test files are used for this code: `m3seq002,m3seq003`.

```
5828 ⟨/initex | package⟩
5829 ⟨@@=seq⟩
```

A sequence is a control sequence whose top-level expansion is of the form “`\s__-seq \__seq_item:n {⟨item1⟩} ... \__seq_item:n {⟨itemn⟩}`”, with a leading scan mark followed by  $n$  items of the same form. An earlier implementation used the structure “`\seq_elt:w ⟨item1⟩ \seq_elt_end: ... \seq_elt:w ⟨itemn⟩ \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 {⟨item⟩}`

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:x**

`\__seq_push_item_def:n {⟨code⟩}`

Saves the definition of `\__seq_item:n` and redefines it to accept one parameter and expand to `⟨code⟩`. This function should always be balanced by use of `\__seq_pop_item_def:..`

<code>\_\_seq\_pop\_item\_def:</code>	<code>\_\_seq\_pop\_item\_def:</code> Restores the definition of <code>\_\_seq\_item:n</code> most recently saved by <code>\_\_seq\_push\_item\_def:n</code> . This function should always be used in a balanced pair with <code>\_\_seq\_push\_item\_def:n</code> .
<code>\s\_seq</code>	This private scan mark.  5830 <code>\scan_new:N \s\_seq</code>  <i>(End definition for <code>\s\_seq</code>.)</i>
<code>\_\_seq\_item:n</code>	The delimiter is always defined, but when used incorrectly simply removes its argument and hits an undefined control sequence to raise an error.  5831 <code>\cs_new:Npn \_\_seq_item:n</code> 5832    { 5833 <code>\_kernel_msg_expandable_error:nn { kernel } { misused-sequence }</code> 5834 <code>\use_none:n</code> 5835    }  <i>(End definition for <code>\_\_seq_item:n</code>.)</i>
<code>\l\_seq\_internal\_a\_tl</code> <code>\l\_seq\_internal\_b\_tl</code>	Scratch space for various internal uses.  5836 <code>\tl_new:N \l\_seq_internal_a_tl</code> 5837 <code>\tl_new:N \l\_seq_internal_b_tl</code>  <i>(End definition for <code>\l\_seq_internal_a_tl</code> and <code>\l\_seq_internal_b_tl</code>.)</i>
<code>\_\_seq\_tmp:w</code>	Scratch function for internal use.  5838 <code>\cs_new_eq:NN \_\_seq_tmp:w ?</code>  <i>(End definition for <code>\_\_seq_tmp:w</code>.)</i>
<code>\c_empty_seq</code>	A sequence with no item, following the structure mentioned above.  5839 <code>\tl_const:Nn \c_empty_seq { \s\_seq }</code>  <i>(End definition for <code>\c_empty_seq</code>. This variable is documented on page 77.)</i>

## 9.1 Allocation and initialisation

<code>\seq_new:N</code>	Sequences are initialized to <code>\c_empty_seq</code> .
<code>\seq_new:c</code>	5840 <code>\cs_new_protected:Npn \seq_new:N #1</code> 5841    { 5842 <code>\_kernel_chk_if_free_cs:N #1</code> 5843 <code>\cs_gset_eq:NN #1 \c_empty_seq</code> 5844    } 5845 <code>\cs_generate_variant:Nn \seq_new:N { c }</code>  <i>(End definition for <code>\seq_new:N</code>. This function is documented on page 68.)</i>
<code>\seq_clear:N</code>	Clearing a sequence is similar to setting it equal to the empty one.
<code>\seq_clear:c</code>	5846 <code>\cs_new_protected:Npn \seq_clear:N #1</code> 5847    { <code>\seq_set_eq:NN #1 \c_empty_seq</code> }
<code>\seq_gclear:N</code>	5848 <code>\cs_generate_variant:Nn \seq_clear:N { c }</code>
<code>\seq_gclear:c</code>	5849 <code>\cs_new_protected:Npn \seq_gclear:N #1</code> 5850    { <code>\seq_gset_eq:NN #1 \c_empty_seq</code> } 5851 <code>\cs_generate_variant:Nn \seq_gclear:N { c }</code>

(End definition for `\seq_clear:N` and `\seq_gclear:N`. These functions are documented on page 68.)

`\seq_clear_new:N` Once again we copy code from the token list functions.

```
5852 \cs_new_protected:Npn \seq_clear_new:N #1
5853   { \seq_if_exist:NTF #1 { \seq_clear:N #1 } { \seq_new:N #1 } }
5854 \cs_generate_variant:Nn \seq_clear_new:N { c }
5855 \cs_new_protected:Npn \seq_gclear_new:N #1
5856   { \seq_if_exist:NTF #1 { \seq_gclear:N #1 } { \seq_new:N #1 } }
5857 \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 68.)

`\seq_set_eq:NN` Copying a sequence is the same as copying the underlying token list.

```
5858 \cs_new_eq:NN \seq_set_eq:NN \tl_set_eq:NN
5859 \cs_new_eq:NN \seq_set_eq:Nc \tl_set_eq:Nc
5860 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
5861 \cs_new_eq:NN \seq_set_eq:cc \tl_set_eq:cc
5862 \cs_new_eq:NN \seq_gset_eq:NN \tl_gset_eq:NN
5863 \cs_new_eq:NN \seq_gset_eq:Nc \tl_gset_eq:Nc
5864 \cs_new_eq:NN \seq_gset_eq:cc \tl_gset_eq:cc
5865 \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 68.)

`\seq_set_from_clist:NN` Setting a sequence from a comma-separated list is done using a simple mapping.

```
5866 \cs_new_protected:Npn \seq_set_from_clist:NN #1#2
5867   {
5868     \tl_set:Nx #1
5869     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
5870   }
5871 \cs_new_protected:Npn \seq_set_from_clist:Nn #1#2
5872   {
5873     \tl_set:Nx #1
5874     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
5875   }
5876 \cs_new_protected:Npn \seq_gset_from_clist:NN #1#2
5877   {
5878     \tl_gset:Nx #1
5879     { \s__seq \clist_map_function:NN #2 \__seq_wrap_item:n }
5880   }
5881 \cs_new_protected:Npn \seq_gset_from_clist:Nn #1#2
5882   {
5883     \tl_gset:Nx #1
5884     { \s__seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
5885   }
5886 \cs_generate_variant:Nn \seq_set_from_clist:NN { Nc }
5887 \cs_generate_variant:Nn \seq_set_from_clist:NN { c , cc }
5888 \cs_generate_variant:Nn \seq_set_from_clist:Nn { c      }
5889 \cs_generate_variant:Nn \seq_gset_from_clist:NN { Nc }
5890 \cs_generate_variant:Nn \seq_gset_from_clist:NN { c , cc }
5891 \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 68.)

```

\seq_set_split:Nnn
\seq_set_split:NnV
\seq_gset_split:Nnn
\seq_gset_split:NnV
\seq_set_split>NNnn
\seq_set_split_auxi:w
\seq_set_split_auxii:w
\seq_set_split_end:

```

When the separator is empty, everything is very simple, just map `\_seq_wrap_item:n` 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_tl` 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.

```

5892 \cs_new_protected:Npn \seq_set_split:Nnn
5893   { \_seq_set_split:NNnn \tl_set:Nx }
5894 \cs_new_protected:Npn \seq_gset_split:Nnn
5895   { \_seq_set_split:NNnn \tl_gset:Nx }
5896 \cs_new_protected:Npn \_seq_set_split:NNnn #1#2#3#4
5897   {
5898     \tl_if_empty:nTF {#3}
5899     {
5900       \tl_set:Nn \l_\_seq_internal_a_tl
5901         { \tl_map_function:nN {#4} \_seq_wrap_item:n }
5902     }
5903     {
5904       \tl_set:Nn \l_\_seq_internal_a_tl
5905         {
5906           \_seq_set_split_auxi:w \prg_do_nothing:
5907             #4
5908           \_seq_set_split_end:
5909         }
5910       \tl_replace_all:Nnn \l_\_seq_internal_a_tl { #3 }
5911       {
5912         \_seq_set_split_end:
5913           \_seq_set_split_auxi:w \prg_do_nothing:
5914         }
5915       \tl_set:Nx \l_\_seq_internal_a_tl { \l_\_seq_internal_a_tl }
5916     }
5917   #1 #2 { \s_\_seq \l_\_seq_internal_a_tl }
5918 }
5919 \cs_new:Npn \_seq_set_split_auxi:w #1 \_seq_set_split_end:
5920   {
5921     \exp_not:N \_seq_set_split_auxii:w
5922     \exp_args:No \tl_trim_spaces:n {#1}
5923     \exp_not:N \_seq_set_split_end:
5924   }
5925 \cs_new:Npn \_seq_set_split_auxii:w #1 \_seq_set_split_end:
5926   { \_seq_wrap_item:n {#1} }
5927 \cs_generate_variant:Nn \seq_set_split:Nnn { NnV }
5928 \cs_generate_variant:Nn \seq_gset_split:Nnn { NnV }

```

(End definition for `\seq_set_split:Nnn` and others. These functions are documented on page 69.)

```

\seq_concat:NNN
\seq_concat:ccc
\seq_gconcat:NNN
\seq_gconcat:ccc

```

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.

```

5929 \cs_new_protected:Npn \seq_concat:NNN #1#2#3
5930   { \tl_set:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
5931 \cs_new_protected:Npn \seq_gconcat:NNN #1#2#3
5932   { \tl_gset:Nf #1 { \exp_after:wN \use_i:nn \exp_after:wN #2 #3 } }
5933 \cs_generate_variant:Nn \seq_concat:NNN { ccc }
5934 \cs_generate_variant:Nn \seq_gconcat:NNN { ccc }

```

(End definition for `\seq_concat:NNN` and `\seq_gconcat:NNN`. These functions are documented on page 69.)

**\seq\_if\_exist\_p:N** Copies of the `cs` functions defined in `!3basics`.

```

5935 \prg_new_eq_conditional>NNn \seq_if_exist:N \cs_if_exist:N
5936   { TF , T , F , p }
5937 \prg_new_eq_conditional>NNn \seq_if_exist:c \cs_if_exist:c
5938   { TF , T , F , p }

```

(End definition for `\seq_if_exist:NTF`. This function is documented on page 69.)

## 9.2 Appending data to either end

**\seq\_put\_left:Nn**

**\seq\_put\_left:NV**

**\seq\_put\_left:Nv**

**\seq\_put\_left:No**

**\seq\_put\_left:Nx**

**\seq\_put\_left:cn**

**\seq\_put\_left:cV**

**\seq\_put\_left:cv**

**\seq\_put\_left:co**

**\seq\_put\_left:cx**

**\seq\_gput\_left:Nn**

**\seq\_gput\_left:NV**

**\seq\_gput\_left:Nv**

**\seq\_gput\_left:No**

**\seq\_gput\_left:Nx**

**\seq\_gput\_left:cn**

**\seq\_gput\_left:cV**

**\seq\_gput\_left:cv**

**\seq\_gput\_left:co**

**\seq\_gput\_left:cx**

**\\_seq\_put\_left\_aux:w**

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.

```

5939 \cs_new_protected:Npn \seq_put_left:Nn #1#2
5940   {
5941     \tl_set:Nx #1
5942     {
5943       \exp_not:n { \s__seq \_seq_item:n {#2} }
5944       \exp_not:f { \exp_after:wN \_seq_put_left_aux:w #1 }
5945     }
5946   }
5947 \cs_new_protected:Npn \seq_gput_left:Nn #1#2
5948   {
5949     \tl_gset:Nx #1
5950     {
5951       \exp_not:n { \s__seq \_seq_item:n {#2} }
5952       \exp_not:f { \exp_after:wN \_seq_put_left_aux:w #1 }
5953     }
5954   }
5955 \cs_new:Npn \_seq_put_left_aux:w \s__seq { \exp_stop_f: }
5956 \cs_generate_variant:Nn \seq_put_left:Nn { NV , Nv , No , Nx }
5957 \cs_generate_variant:Nn \seq_put_left:Nn { c , cV , cv , co , cx }
5958 \cs_generate_variant:Nn \seq_gput_left:Nn { NV , Nv , No , Nx }
5959 \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 69.)

**\seq\_put\_right:Nn**

**\seq\_put\_right:NV**

**\seq\_put\_right:Nv**

**\seq\_put\_right:No**

**\seq\_put\_right:Nx**

**\seq\_put\_right:cn**

**\seq\_put\_right:cV**

**\seq\_put\_right:cv**

**\seq\_put\_right:co**

**\seq\_put\_right:cx**

**\seq\_gput\_right:Nn**

**\seq\_gput\_right:NV**

**\seq\_gput\_right:Nv**

**\seq\_gput\_right:No**

**\seq\_gput\_right:Nx**

**\seq\_gput\_right:cn**

Since there is no trailing marker, adding an item to the right of a sequence simply means wrapping it in `\_seq_item:n`.

```

5960 \cs_new_protected:Npn \seq_put_right:Nn #1#2
5961   { \tl_put_right:Nn #1 { \_seq_item:n {#2} } }
5962 \cs_new_protected:Npn \seq_gput_right:Nn #1#2
5963   { \tl_gput_right:Nn #1 { \_seq_item:n {#2} } }

```

```

5964 \cs_generate_variant:Nn \seq_gput_right:Nn { NV , Nv , No , Nx }
5965 \cs_generate_variant:Nn \seq_gput_right:Nn { c , cV , cv , co , cx }
5966 \cs_generate_variant:Nn \seq_put_right:Nn { NV , Nv , No , Nx }
5967 \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 69.)

### 9.3 Modifying sequences

`\__seq_wrap_item:n` This function converts its argument to a proper sequence item in an x-expansion context.

```
5968 \cs_new:Npn \__seq_wrap_item:n #1 { \exp_not:n { \__seq_item:n {#1} } }
```

(End definition for `\__seq_wrap_item:n`.)

`\l__seq_remove_seq` An internal sequence for the removal routines.

```
5969 \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.

`\seq_remove_duplicates:c`

`\seq_gremove_duplicates:N`

`\seq_gremove_duplicates:c`

`\__seq_remove_duplicates>NN`

```

5970 \cs_new_protected:Npn \seq_remove_duplicates:N
5971   { \__seq_remove_duplicates>NN \seq_set_eq:NN }
5972 \cs_new_protected:Npn \seq_gremove_duplicates:N
5973   { \__seq_remove_duplicates>NN \seq_gset_eq:NN }
5974 \cs_new_protected:Npn \__seq_remove_duplicates>NN #1#2
5975   {
5976     \seq_clear:N \l__seq_remove_seq
5977     \seq_map_inline:Nn #2
5978     {
5979       \seq_if_in:NnF \l__seq_remove_seq {##1}
5980         { \seq_put_right:Nn \l__seq_remove_seq {##1} }
5981     }
5982     #1 #2 \l__seq_remove_seq
5983   }
5984 \cs_generate_variant:Nn \seq_remove_duplicates:N { c }
5985 \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 72.)

`\seq_remove_all:Nn`

`\seq_remove_all:cn`

`\seq_gremove_all:Nn`

`\seq_gremove_all:cn`

`\__seq_remove_all_aux>NNn`

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.

```

5986 \cs_new_protected:Npn \seq_remove_all:Nn
5987   { \__seq_remove_all_aux>NNn \tl_set:Nx }
5988 \cs_new_protected:Npn \seq_gremove_all:Nn
5989   { \__seq_remove_all_aux>NNn \tl_gset:Nx }

```

```
5990 \cs_new_protected:Npn \__seq_remove_all_aux:NNn #1#2#3
5991 {
5992     \__seq_push_item_def:n
5993     {
5994         \str_if_eq:nnT {##1} {#3}
5995         {
5996             \if_false: { \fi: }
5997             \tl_set:Nn \l__seq_internal_b_tl {##1}
5998             #1 #2
5999             { \if_false: } \fi:
6000             \exp_not:o {#2}
6001             \tl_if_eq:NNT \l__seq_internal_a_tl \l__seq_internal_b_tl
6002             { \use_none:nn }
6003         }
6004         \__seq_wrap_item:n {##1}
6005     }
6006     \tl_set:Nn \l__seq_internal_a_tl {#3}
6007     #1 #2 {#2}
6008     \__seq_pop_item_def:
6009 }
6010 \cs_generate_variant:Nn \seq_remove_all:Nn { c }
6011 \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 [72](#).)

`\seq_reverse:N` Previously, `\seq_reverse:N` was coded by collecting the items in reverse order after an `\seq_reverse:c \exp_stop_f:` marker.

```
\seq_greverse:N \seq_greverse:c _seq_reverse:NN reverse_item:nwn
\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_
{
    #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.

```
6012 \cs_new_protected:Npn \seq_reverse:N  
6013 { \__seq_reverse:NN \tl_set:Nx }
```

```

6014 \cs_new_protected:Npn \seq_greverse:N
6015   { \__seq_reverse:NN \tl_gset:Nx }
6016 \cs_new_protected:Npn \__seq_reverse:NN #1 #2
6017   {
6018     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6019     \cs_set_eq:NN \__seq_item:n \__seq_reverse_item:nwn
6020     #1 #2 { #2 \exp_not:n { } }
6021     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6022   }
6023 \cs_new:Npn \__seq_reverse_item:nwn #1 #2 \exp_not:n #3
6024   {
6025     #2
6026     \exp_not:n { \__seq_item:n {#1} #3 }
6027   }
6028 \cs_generate_variant:Nn \seq_reverse:N { c }
6029 \cs_generate_variant:Nn \seq_greverse:N { c }

```

(End definition for `\seq_reverse:N` and others. These functions are documented on page 72.)

`\seq_sort:Nn` Implemented in `l3sort`.

`\seq_sort:cn`

`\seq_gsort:Nn`

`\seq_gsort:cn`

## 9.4 Sequence conditionals

`\seq_if_empty_p:N` Similar to token lists, we compare with the empty sequence.

```

6030 \prg_new_conditional:Npnn \seq_if_empty:N #1 { p , T , F , TF }
6031   {
6032     \if_meaning:w #1 \c_empty_seq
6033     \prg_return_true:
6034     \else:
6035     \prg_return_false:
6036     \fi:
6037   }
6038 \prg_generate_conditional_variant:Nnn \seq_if_empty:N
6039   { c } { p , T , F , TF }

```

(End definition for `\seq_if_empty:NTF`. This function is documented on page 72.)

`\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.

```

6040 \prg_new_protected_conditional:Npnn \seq_if_in:Nn #1#2
6041   { T , F , TF }
6042   {
6043     \group_begin:
6044     \tl_set:Nn \l__seq_internal_a_tl {##2}
6045     \cs_set_protected:Npn \__seq_item:n ##1
6046     {
6047       \tl_set:Nn \l__seq_internal_b_tl {##1}
6048       \if_meaning:w \l__seq_internal_a_tl \l__seq_internal_b_tl
6049       \exp_after:wN \__seq_if_in:

```

```

6050           \fi:
6051       }
6052     #1
6053   \group_end:
6054   \prg_return_false:
6055   \prg_break_point:
6056 }
6057 \cs_new:Npn \__seq_if_in:
6058   { \prg_break:n { \group_end: \prg_return_true: } }
6059 \prg_generate_conditional_variant:Nnn \seq_if_in:Nn
6060   { 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 73.)

```

## 9.5 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.

```

6061 \cs_new_protected:Npn \__seq_pop:NNNN #1#2#3#4
6062 {
6063   \if_meaning:w #3 \c_empty_seq
6064     \tl_set:Nn #4 { \q_no_value }
6065   \else:
6066     #1#2#3#4
6067   \fi:
6068 }
6069 \cs_new_protected:Npn \__seq_pop_TF:NNNN #1#2#3#4
6070 {
6071   \if_meaning:w #3 \c_empty_seq
6072     % \tl_set:Nn #4 { \q_no_value }
6073     \prg_return_false:
6074   \else:
6075     #1#2#3#4
6076     \prg_return_true:
6077   \fi:
6078 }

```

(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

```

6079 \cs_new_protected:Npn \seq_get_left:NN #1#2
6080 {
6081   \tl_set:Nx #2
6082   {
6083     \exp_after:wN \__seq_get_left:wnw
6084     #1 \__seq_item:n { \q_no_value } \q_stop
6085   }
6086 }
6087 \cs_new:Npn \__seq_get_left:wnw #1 \__seq_item:n #2#3 \q_stop
6088   { \exp_not:n {#2} }
6089 \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 69.)

`\seq_pop_left:NN` 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.

```

6090 \cs_new_protected:Npn \seq_pop_left:NN
6091   { \_seq_pop>NNNN \_seq_pop_left>NNNN \tl_set:Nn }
6092   \cs_new_protected:Npn \seq_gpop_left:NN
6093   { \_seq_pop>NNNN \_seq_pop_left>NNNN \tl_gset:Nn }
6094   \cs_new_protected:Npn \_seq_pop_left>NNNN #1#2#3
6095   { \exp_after:wN \_seq_pop_left:wnwNNN #2 \q_stop #1#2#3 }
6096   \cs_new_protected:Npn \_seq_pop_left:wnwNNN
6097   #1 \_seq_item:n #2#3 \q_stop #4#5#6
6098   {
6099     #4 #5 { #1 #3 }
6100     \tl_set:Nn #6 {#2}
6101   }
6102 \cs_generate_variant:Nn \seq_pop_left:NN { c }
6103 \cs_generate_variant:Nn \seq_gpop_left:NN { c }
```

(End definition for `\seq_pop_left:NN` and others. These functions are documented on page 70.)

`\seq_get_right:NN` 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.

```

6104 \cs_new_protected:Npn \seq_get_right:NN #1#2
6105   {
6106     \tl_set:Nx #2
6107     {
6108       \exp_after:wN \use_i_i:nnn
6109       \exp_after:wN \_seq_get_right_loop:nw
6110       \exp_after:wN \q_no_value
6111       #
6112       \_seq_get_right_end:Nnn \_seq_item:n
6113     }
6114   }
6115 \cs_new:Npn \_seq_get_right_loop:nw #1#2 \_seq_item:n
6116   {
6117     #2 \use_none:n {#1}
6118     \_seq_get_right_loop:nw
6119   }
6120 \cs_new:Npn \_seq_get_right_end:Nnn #1#2#3 { \exp_not:n {#2} }
6121 \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 70.)

`\seq_pop_right: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.

```

6122 \cs_new_protected:Npn \seq_pop_right:NN
6123   { \__seq_pop>NNNN \__seq_pop_right>NNN \tl_set:Nx }
6124 \cs_new_protected:Npn \seq_gpop_right:NN
6125   { \__seq_pop>NNNN \__seq_pop_right>NNN \tl_gset:Nx }
6126 \cs_new_protected:Npn \__seq_pop_right>NNN #1#2#3
6127   {
6128     \cs_set_eq:NN \__seq_tmp:w \__seq_item:n
6129     \cs_set_eq:NN \__seq_item:n \scan_stop:
6130     #1 #2
6131     { \if_false: } \fi: \s_seq
6132     \exp_after:wN \use_i:nnn
6133     \exp_after:wN \__seq_pop_right_loop:nn
6134     #2
6135     {
6136       \if_false: { \fi: }
6137       \tl_set:Nx #3
6138     }
6139     { } \use_none:nn
6140     \cs_set_eq:NN \__seq_item:n \__seq_tmp:w
6141   }
6142 \cs_new:Npn \__seq_pop_right_loop:nn #1#2
6143   {
6144     #2 { \exp_not:n {#1} }
6145     \__seq_pop_right_loop:nn
6146   }
6147 \cs_generate_variant:Nn \seq_pop_right:NN { c }
6148 \cs_generate_variant:Nn \seq_gpop_right:NN { c }

```

(End definition for `\seq_pop_right:NN` and others. These functions are documented on page 70.)

`\seq_get_left:NNTF` 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_left:cNTF
\seq_get_right:cNTF
6149 \prg_new_protected_conditional:Npnn \seq_get_left:NN #1#2 { T , F , TF }
6150   { \__seq_pop_TF>NNNN \prg_do_nothing: \seq_get_left:NN #1#2 }
6151 \prg_new_protected_conditional:Npnn \seq_get_right:NN #1#2 { T , F , TF }
6152   { \__seq_pop_TF>NNNN \prg_do_nothing: \seq_get_right:NN #1#2 }
6153 \prg_generate_conditional_variant:Nnn \seq_get_left:NN
6154   { c } { T , F , TF }
6155 \prg_generate_conditional_variant:Nnn \seq_get_right:NN
6156   { c } { T , F , TF }

```

(End definition for `\seq_get_left:NNTF` and `\seq_get_right:NNTF`. These functions are documented on page 71.)

`\seq_pop_left:NNTF` More or less the same for popping.

```

\seq_pop_left:cNTF
\seq_gpop_left:NNTF
\seq_gpop_left:cNTF
\seq_pop_right:NNTF
\seq_pop_right:cNTF
\seq_gpop_right:NNTF
\seq_gpop_right:cNTF
6157 \prg_new_protected_conditional:Npnn \seq_pop_left:NN #1#2
6158   { T , F , TF }
6159   { \__seq_pop_TF>NNNN \__seq_pop_left>NNN \tl_set:Nn #1 #2 }
6160 \prg_new_protected_conditional:Npnn \seq_gpop_left:NN #1#2
6161   { T , F , TF }

```

```

6162 { \__seq_pop_TF:NNNN \__seq_pop_left:NNN \tl_gset:Nn #1 #2 }
6163 \prg_new_protected_conditional:Npnn \seq_pop_right:NN #1#2
6164 { T , F , TF }
6165 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_set:Nx #1 #2 }
6166 \prg_new_protected_conditional:Npnn \seq_gpop_right:NN #1#2
6167 { T , F , TF }
6168 { \__seq_pop_TF:NNNN \__seq_pop_right:NNN \tl_gset:Nx #1 #2 }
6169 \prg_generate_conditional_variant:Nnn \seq_pop_left:NN { c }
6170 { T , F , TF }
6171 \prg_generate_conditional_variant:Nnn \seq_gpop_left:NN { c }
6172 { T , F , TF }
6173 \prg_generate_conditional_variant:Nnn \seq_pop_right:NN { c }
6174 { T , F , TF }
6175 \prg_generate_conditional_variant:Nnn \seq_gpop_right:NN { c }
6176 { T , F , TF }

```

(End definition for `\seq_pop_left:NNTF` and others. These functions are documented on page 71.)

`\seq_item:Nn`  
`\seq_item:cn`  
`\__seq_item:wNn`  
`\__seq_item:nN`  
`\__seq_item:nwn`

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.

```

6177 \cs_new:Npn \seq_item:Nn #1
6178 { \exp_after:wN \__seq_item:wNn #1 \q_stop #1 }
6179 \cs_new:Npn \__seq_item:wNn \s__seq #1 \q_stop #2#3
6180 {
6181     \exp_args:Nf \__seq_item:nwn
6182     { \exp_args:Nf \__seq_item:nN { \int_eval:n {#3} } #2 }
6183     #1
6184     \prg_break: \__seq_item:n { }
6185     \prg_break_point:
6186 }
6187 \cs_new:Npn \__seq_item:nN #1#2
6188 {
6189     \int_compare:nNnTF {#1} < 0
6190     { \int_eval:n { \seq_count:N #2 + 1 + #1 } }
6191     {#1}
6192 }
6193 \cs_new:Npn \__seq_item:nwn #1#2 \__seq_item:n #3
6194 {
6195     #2
6196     \int_compare:nNnTF {#1} = 1
6197     { \prg_break:n { \exp_not:n {#3} } }
6198     { \exp_args:Nf \__seq_item:nwn { \int_eval:n { #1 - 1 } } }
6199 }
6200 \cs_generate_variant:Nn \seq_item:Nn { c }

```

(End definition for `\seq_item:Nn` and others. This function is documented on page 70.)

`\seq_rand_item:N`  
`\seq_rand_item:c`

Importantly, `\seq_item:Nn` only evaluates its argument once.

```

6201 \cs_new:Npn \seq_rand_item:N #1
6202 {
6203     \seq_if_empty:NF #1
6204     { \seq_item:Nn #1 { \int_rand:nn { 1 } { \seq_count:N #1 } } }

```

```

6205   }
6206 \cs_generate_variant:Nn \seq_rand_item:N { c }

```

(End definition for `\seq_rand_item:N`. This function is documented on page 70.)

## 9.6 Mapping to 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.

```

6207 \cs_new:Npn \seq_map_break:
6208   { \prg_map_break:Nn \seq_map_break: { } }
6209 \cs_new:Npn \seq_map_break:n
6210   { \prg_map_break:Nn \seq_map_break: }

```

(End definition for `\seq_map_break:` and `\seq_map_break:n`. These functions are documented on page 73.)

`\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 argument delimited by `\_seq_item:n` is 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.

```

6211 \cs_new:Npn \seq_map_function:NN #1#
6212   {
6213     \exp_after:wN \use_i_i:nnn
6214     \exp_after:wN \_seq_map_function:Nw
6215     \exp_after:wN #2
6216     #1
6217     \prg_break: \_seq_item:n { } \prg_break_point:
6218     \prg_break_point:Nn \seq_map_break: { }
6219   }
6220 \cs_new:Npn \_seq_map_function:Nw #1#2 \_seq_item:n #3
6221   {
6222     #2
6223     #1 {#3}
6224     \_seq_map_function:Nw #1
6225   }
6226 \cs_generate_variant:Nn \seq_map_function:NN { c }

```

(End definition for `\seq_map_function>NN` and `\_seq_map_function>NNn`. This function is documented on page 73.)

`\_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.

```

6227 \cs_new_protected:Npn \_seq_push_item_def:n
6228   {
6229     \_seq_push_item_def:
6230     \cs_gset:Npn \_seq_item:n ##1
6231   }
6232 \cs_new_protected:Npn \_seq_push_item_def:x
6233   {
6234     \_seq_push_item_def:
6235     \cs_gset:Npx \_seq_item:n ##1

```

```

6236     }
6237 \cs_new_protected:Npn \__seq_push_item_def:
6238 {
6239     \int_gincr:N \g__kernel_prg_map_int
6240     \cs_gset_eq:cN { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
6241     \__seq_item:n
6242 }
6243 \cs_new_protected:Npn \__seq_pop_item_def:
6244 {
6245     \cs_gset_eq:Nc \__seq_item:n
6246     { __seq_map_ \int_use:N \g__kernel_prg_map_int :w }
6247     \int_gdecr:N \g__kernel_prg_map_int
6248 }

```

(End definition for `\__seq_push_item_def:n`, `\__seq_push_item_def:`, and `\__seq_pop_item_def:.`)

`\seq_map_inline:Nn`  
`\seq_map_inline:cn`

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`.

```

6249 \cs_new_protected:Npn \seq_map_inline:Nn #1#2
6250 {
6251     \__seq_push_item_def:n {#2}
6252     #1
6253     \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6254 }
6255 \cs_generate_variant:Nn \seq_map_inline:Nn { c }

```

(End definition for `\seq_map_inline:Nn`. This function is documented on page 73.)

`\seq_map_variable>NNn`  
`\seq_map_variable:Ncn`  
`\seq_map_variable:cNn`  
`\seq_map_variable:ccn`

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.

```

6256 \cs_new_protected:Npn \seq_map_variable>NNn #1#2#3
6257 {
6258     \__seq_push_item_def:x
6259     {
6260         \tl_set:Nn \exp_not:N #2 {##1}
6261         \exp_not:n {#3}
6262     }
6263     #1
6264     \prg_break_point:Nn \seq_map_break: { \__seq_pop_item_def: }
6265 }
6266 \cs_generate_variant:Nn \seq_map_variable>NNn { Nc }
6267 \cs_generate_variant:Nn \seq_map_variable>NNn { c , cc }

```

(End definition for `\seq_map_variable>NNn`. This function is documented on page 73.)

`\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.

```

6268 \cs_new:Npn \seq_count:N #1
6269 {
6270     \int_eval:n
6271     {

```

```

6272     \exp_after:wN \use_i:nn
6273     \exp_after:wN \__seq_count:w
6274     #1
6275     \__seq_count_end:w \__seq_item:n 7
6276     \__seq_count_end:w \__seq_item:n 6
6277     \__seq_count_end:w \__seq_item:n 5
6278     \__seq_count_end:w \__seq_item:n 4
6279     \__seq_count_end:w \__seq_item:n 3
6280     \__seq_count_end:w \__seq_item:n 2
6281     \__seq_count_end:w \__seq_item:n 1
6282     \__seq_count_end:w \__seq_item:n 0
6283     \prg_break_point:
6284   }
6285 }
6286 \cs_new:Npn \__seq_count:w
6287   #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4 \__seq_item:n
6288   #5 \__seq_item:n #6 \__seq_item:n #7 \__seq_item:n #8 #9 \__seq_item:n
6289   { #9 8 + \__seq_count:w }
6290 \cs_new:Npn \__seq_count_end:w #1#2 \prg_break_point: {#1}
6291 \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 74.)

## 9.7 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`.

```

6292 \cs_new:Npn \seq_use:Nnnn #1#2#3#4
6293 {
6294   \seq_if_exist:NTF #1
6295   {
6296     \int_case:nnF { \seq_count:N #1 }
6297     {
6298       { 0 } { }
6299       { 1 } { \exp_after:wN \__seq_use>NNnNnn #1 ? { } { } }
6300       { 2 } { \exp_after:wN \__seq_use>NNnNnn #1 {#2} }
6301     }
6302     {
6303       \exp_after:wN \__seq_use_setup:w #1 \__seq_item:n
6304       \q_mark { \__seq_use:nwwwnwn {#3} }
6305       \q_mark { \__seq_use:nwwn {#4} }
6306       \q_stop { }
6307     }
6308   }
6309   {
6310     \__kernel_msg_expandable_error:nnn
6311     { kernel } { bad-variable } {#1}
6312   }
6313 }
6314 \cs_generate_variant:Nn \seq_use:Nnnn { c }
6315 \cs_new:Npn \__seq_use>NNnNnn #1#2#3#4#5#6 { \exp_not:n { #3 #6 #5 } }
6316 \cs_new:Npn \__seq_use_setup:w \s_seq { \__seq_use:nwwwnwn { } }

```

```

6317 \cs_new:Npn \__seq_use:nwwwnwn
6318     #1 \__seq_item:n #2 \__seq_item:n #3 \__seq_item:n #4#5
6319     \q_mark #6#7 \q_stop #8
6320     {
6321         #6 \__seq_item:n {#3} \__seq_item:n {#4} #5
6322         \q_mark {#6} #7 \q_stop { #8 #1 #2 }
6323     }
6324 \cs_new:Npn \__seq_use:nwnn #1 \__seq_item:n #2 #3 \q_stop #4
6325     { \exp_not:n { #4 #1 #2 } }
6326 \cs_new:Npn \seq_use:Nn #1#2
6327     { \seq_use:Nnnn #1 {#2} {#2} {#2} }
6328 \cs_generate_variant:Nn \seq_use:Nn { c }

```

(End definition for `\seq_use:Nnnn` and others. These functions are documented on page 74.)

## 9.8 Sequence stacks

The same functions as for sequences, but with the correct naming.

<code>\seq_push:Nn</code>	Pushing to a sequence is the same as adding on the left.
<code>\seq_push:NV</code>	6329 \cs_new_eq:NN \seq_push:Nn \seq_put_left:Nn
<code>\seq_push:Nv</code>	6330 \cs_new_eq:NN \seq_push:NV \seq_put_left:NV
<code>\seq_push:No</code>	6331 \cs_new_eq:NN \seq_push:Nv \seq_put_left:Nv
<code>\seq_push:Nx</code>	6332 \cs_new_eq:NN \seq_push:No \seq_put_left:No
<code>\seq_push:cn</code>	6333 \cs_new_eq:NN \seq_push:Nx \seq_put_left:Nx
<code>\seq_push:cV</code>	6334 \cs_new_eq:NN \seq_push:cn \seq_put_left:cn
<code>\seq_push:cV</code>	6335 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
<code>\seq_push:co</code>	6336 \cs_new_eq:NN \seq_push:cV \seq_put_left:cV
<code>\seq_push:cx</code>	6337 \cs_new_eq:NN \seq_push:co \seq_put_left:co
<code>\seq_push:cx</code>	6338 \cs_new_eq:NN \seq_push:cX \seq_put_left:cX
<code>\seq_gpush:Nn</code>	6339 \cs_new_eq:NN \seq_gpush:Nn \seq_gput_left:Nn
<code>\seq_gpush:NV</code>	6340 \cs_new_eq:NN \seq_gpush:NV \seq_gput_left:NV
<code>\seq_gpush:Nv</code>	6341 \cs_new_eq:NN \seq_gpush:Nv \seq_gput_left:Nv
<code>\seq_gpush:No</code>	6342 \cs_new_eq:NN \seq_gpush:No \seq_gput_left:No
<code>\seq_gpush:Nx</code>	6343 \cs_new_eq:NN \seq_gpush:Nx \seq_gput_left:Nx
<code>\seq_gpush:cn</code>	6344 \cs_new_eq:NN \seq_gpush:cn \seq_gput_left:cn
<code>\seq_gpush:cV</code>	6345 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
<code>\seq_gpush:cV</code>	6346 \cs_new_eq:NN \seq_gpush:cV \seq_gput_left:cV
<code>\seq_gpush:co</code>	6347 \cs_new_eq:NN \seq_gpush:co \seq_gput_left:co
<code>\seq_gpush:cx</code>	6348 \cs_new_eq:NN \seq_gpush:cX \seq_gput_left:cX

(End definition for `\seq_push:Nn` and `\seq_gpush:Nn`. These functions are documented on page 76.)

<code>\seq_get&gt;NN</code>	In most cases, getting items from the stack does not need to specify that this is from the left. So alias are provided.
<code>\seq_get:cN</code>	
<code>\seq_pop&gt;NN</code>	6349 \cs_new_eq:NN \seq_get>NN \seq_get_left>NN
<code>\seq_pop:cN</code>	6350 \cs_new_eq:NN \seq_get:cN \seq_get_left:cN
<code>\seq_gpop&gt;NN</code>	6351 \cs_new_eq:NN \seq_pop>NN \seq_pop_left>NN
<code>\seq_gpop:cN</code>	6352 \cs_new_eq:NN \seq_pop:cN \seq_pop_left:cN
	6353 \cs_new_eq:NN \seq_gpop>NN \seq_gpop_left>NN
	6354 \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 75.)

`\seq_get:NNTF` More copies.

```
6355 \prg_new_eq_conditional:NNn \seq_get:NN \seq_get_left:NN { T , F , TF }
6356 \prg_new_eq_conditional:NNn \seq_get:cN \seq_get_left:cN { T , F , TF }
6357 \prg_new_eq_conditional:NNn \seq_pop:NN \seq_pop_left:NN { T , F , TF }
6358 \prg_new_eq_conditional:NNn \seq_pop:cN \seq_pop_left:cN { T , F , TF }
6359 \prg_new_eq_conditional:NNn \seq_gpop:NN \seq_gpop_left:NN { T , F , TF }
6360 \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 75.)

## 9.9 Viewing sequences

`\seq_show:N` Apply the general `\msg_show:nnnnnn`.

```
6361 \cs_new_protected:Npn \seq_show:N { \__seq_show:NN \msg_show:nnxxxxx }
6362 \cs_generate_variant:Nn \seq_show:N { c }
6363 \cs_new_protected:Npn \seq_log:N { \__seq_show:NN \msg_log:nnxxxxx }
6364 \cs_generate_variant:Nn \seq_log:N { c }
6365 \cs_new_protected:Npn \__seq_show:NN #1#2
6366 {
6367     \__kernel_chk_defined:NT #2
6368     {
6369         #1 { LaTeX/kernel } { show-seq }
6370         { \token_to_str:N #2 }
6371         { \seq_map_function:NN #2 \msg_show_item:n }
6372         { } { }
6373     }
6374 }
```

(End definition for `\seq_show:N`, `\seq_log:N`, and `\__seq_show:NN`. These functions are documented on page 78.)

## 9.10 Scratch sequences

`\l_tmpa_seq` Temporary comma list variables.

```
6375 \seq_new:N \l_tmpa_seq
6376 \seq_new:N \l_tmpb_seq
6377 \seq_new:N \g_tmpa_seq
6378 \seq_new:N \g_tmpb_seq
```

(End definition for `\l_tmpa_seq` and others. These variables are documented on page 78.)

6379 ⟨\*initex | package⟩

# 10 l3int implementation

6380 ⟨\*initex | package⟩

6381 ⟨@@=int⟩

The following test files are used for this code: `m3int001`, `m3int002`, `m3int03`.

`\c_max_register_int` Done in l3basics.

(End definition for `\c_max_register_int`. This variable is documented on page 90.)

`\_int_to_roman:w` Done in `l3basics`.  
`\if_int_compare:w` (*End definition for `\_int_to_roman:w` and `\if_int_compare:w`. This function is documented on page 91.*)

`\or:` Done in `l3basics`.

(*End definition for `\or:`. This function is documented on page 91.*)

`\int_value:w` Here are the remaining primitives for number comparisons and expressions.

```

\_\_int\_eval:w      6382 \cs_new_eq:NN \int_value:w      \tex_number:D
\_\_int\_eval_end:  6383 \cs_new_eq:NN \_\_int\_eval:w      \tex_numexpr:D
\if_int_odd:w       6384 \cs_new_eq:NN \_\_int\_eval_end:   \tex_relax:D
\if_case:w          6385 \cs_new_eq:NN \if_int_odd:w    \tex_ifodd:D
\if_case:w          6386 \cs_new_eq:NN \if_case:w     \tex_ifcase:D

```

(*End definition for `\int_value:w` and others. These functions are documented on page 91.*)

## 10.1 Integer expressions

`\int_eval:n` Wrapper for `\_int_eval:w`: can be used in an integer expression or directly in the input stream. When debugging, use parentheses to catch early termination.

```

6387 \_\_kernel_patch_args:nNnpn
6388 { { \_\_kernel_chk_expr:nNn {#1} \_\_int_eval:w { } \int_eval:n } }
6389 \cs_new:Npn \int_eval:n #1
6390 { \int_value:w \_\_int_eval:w #1 \_\_int_eval_end: }
6391 \cs_new:Npn \int_eval:w { \int_value:w \_\_int_eval:w }

```

(*End definition for `\int_eval:n` and `\int_eval:w`. These functions are documented on page 79.*)

`\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.

```

\_\_int_abs:N      6392 \_\_kernel_patch_args:nNnpn
6393 { { \_\_kernel_chk_expr:nNn {#1} \_\_int_eval:w { } \int_abs:n } }
6394 \cs_new:Npn \int_abs:n #1
6395 {
6396     \int_value:w \exp_after:wN \_\_int_abs:N
6397     \int_value:w \_\_int_eval:w #1 \_\_int_eval_end:
6398     \exp_stop_f:
6399 }
6400 \cs_new:Npn \_\_int_abs:N #1
6401 { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
6402 \_\_kernel_patch_args:nNnpn
6403 {
6404     { \_\_kernel_chk_expr:nNn {#1} \_\_int_eval:w { } \int_max:nn }
6405     { \_\_kernel_chk_expr:nNn {#2} \_\_int_eval:w { } \int_max:nn }
6406 }
6407 \cs_set:Npn \int_max:nn #1#2
6408 {
6409     \int_value:w \exp_after:wN \_\_int_maxmin:wwN
6410     \int_value:w \_\_int_eval:w #1 \exp_after:wN ;
6411     \int_value:w \_\_int_eval:w #2 ;
6412     >
6413     \exp_stop_f:
6414 }

```

```

6415 \__kernel_patch_args:nNNpn
6416 {
6417   { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_min:nn }
6418   { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_min:nn }
6419 }
6420 \cs_set:Npn \int_min:nn #1#2
6421 {
6422   \int_value:w \exp_after:wN \__int_maxmin:wwN
6423   \int_value:w \__int_eval:w #1 \exp_after:wN ;
6424   \int_value:w \__int_eval:w #2 ;
6425   <
6426   \exp_stop_f:
6427 }
6428 \cs_new:Npn \__int_maxmin:wwN #1 ; #2 ; #3
6429 {
6430   \if_int_compare:w #1 #3 ~
6431     #1
6432   \else:
6433     #2
6434   \fi:
6435 }

```

(End definition for `\int_abs:n` and others. These functions are documented on page 79.)

`\int_div_truncate:nn` 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.

```

6436 \__kernel_patch_args:nNNpn
6437 {
6438   { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_div_truncate:nn }
6439   { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_div_truncate:nn }
6440 }
6441 \cs_new:Npn \int_div_truncate:nn #1#2
6442 {
6443   \int_value:w \__int_eval:w
6444   \exp_after:wN \__int_div_truncate:NwNw
6445   \int_value:w \__int_eval:w #1 \exp_after:wN ;
6446   \int_value:w \__int_eval:w #2 ;
6447   \__int_eval_end:
6448 }
6449 \cs_new:Npn \__int_div_truncate:NwNw #1#2; #3#4;
6450 {
6451   \if_meaning:w 0 #1
6452   0
6453   \else:
6454   (
6455     #1#2
6456     \if_meaning:w - #1 + \else: - \fi:

```

```

6457      ( \if_meaning:w - #3 - \fi: #3#4 - 1 ) / 2
6458      )
6459      \fi:
6460      / #3#4
6461  }

```

For the sake of completeness:

```

6462  \cs_new:Npn \int_div_round:nn #1#2
6463    { \int_value:w \__int_eval:w ( #1 ) / ( #2 ) \__int_eval_end: }

```

Finally there's the modulus operation.

```

6464  \__kernel_patch_args:nNnpn
6465  {
6466    { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_mod:nn }
6467    { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_mod:nn }
6468  }
6469  \cs_new:Npn \int_mod:nn #1#2
6470  {
6471    \int_value:w \__int_eval:w \exp_after:wN \__int_mod:ww
6472    \int_value:w \__int_eval:w #1 \exp_after:wN ;
6473    \int_value:w \__int_eval:w #2 ;
6474    \__int_eval_end:
6475  }
6476  \cs_new:Npn \__int_mod:ww #1; #2;
6477  { #1 - ( \__int_div_truncate:NwNw #1 ; #2 ; ) * #2 }

```

(End definition for `\int_div_truncate:nn` and others. These functions are documented on page 80.)

`\__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.

```

6478  \cs_new:Npn \__kernel_int_add:nnn #1#2#3
6479  {
6480    \int_value:w \__int_eval:w #1
6481    \if_int_compare:w #2 < \c_zero_int \exp_after:wN \reverse_if:N \fi:
6482    \if_int_compare:w #1 < \c_zero_int + #2 + #3 \else: + #3 + #2 \fi:
6483    \__int_eval_end:
6484  }

```

(End definition for `\__kernel_int_add:nnn`.)

## 10.2 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, `\int_new:c` `\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.)

```

6485  (*package)
6486  \cs_new_protected:Npn \int_new:N #1
6487  {

```

```

6488      \__kernel_chk_if_free_cs:N #1
6489      \cs:w newcount \cs_end: #1
6490    }
6491  </package>
6492 \cs_generate_variant:Nn \int_new:N { c }

```

(End definition for `\int_new:N`. This function is documented on page 80.)

```

\int_const:Nn
\int_const:cN
\__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.

```

6493 \__kernel_patch_args:nnNNpn
6494   { \__kernel_chk_var_scope:NN c #1 }
6495   {
6496     { {#1} { \__kernel_chk_expr:nNnN {#2} \__int_eval:w { } \int_const:Nn } }
6497     \cs_new_protected:Npn \int_const:Nn #1#2
6498     {
6499       \int_compare:nNnTF {#2} < \c_zero_int
6500       {
6501         \int_new:N #1
6502         \tex_global:D
6503       }
6504       {
6505         \int_compare:nNnTF {#2} > \c__int_max_constdef_int
6506         {
6507           \int_new:N #1
6508           \tex_global:D
6509         }
6510         {
6511           \__kernel_chk_if_free_cs:N #1
6512           \tex_global:D \__int_constdef:Nw
6513         }
6514       }
6515       #1 = \__int_eval:w #2 \__int_eval_end:
6516     }
6517 \cs_generate_variant:Nn \int_const:Nn { c }
6518 \if_int_odd:w 0
6519   \cs_if_exist:NT \tex_luatexversion:D { 1 }
6520   \cs_if_exist:NT \tex_disablecjktoken:D
6521   { \if_int_compare:w \tex_jis:D "2121 = "3000 ~ 1 \fi: }
6522   \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
6523   \cs_if_exist:NTF \tex_disablecjktoken:D
6524   { \cs_new_eq:NN \__int_constdef:Nw \tex_kchardef:D }
6525   { \cs_new_eq:NN \__int_constdef:Nw \tex_chardef:D }
6526   \__int_constdef:Nw \c__int_max_constdef_int 1114111 ~
6527 \else:
6528   \cs_new_eq:NN \__int_constdef:Nw \tex_mathchardef:D
6529   \tex_mathchardef:D \c__int_max_constdef_int 32767 ~
6530 \fi:

```

(End definition for `\int_const:Nn`, `\__int_constdef:Nw`, and `\c__int_max_constdef_int`. This function is documented on page 80.)

```

\int_zero:N Functions that reset an <integer> register to zero.
\int_zero:c
\int_gzero:N
\int_gzero:c
6531 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
6532 \cs_new_protected:Npn \int_zero:N #1 { #1 = \c_zero_int }
6533 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
6534 \cs_new_protected:Npn \int_gzero:N #1 { \tex_global:D #1 = \c_zero_int }
6535 \cs_generate_variant:Nn \int_zero:N { c }
6536 \cs_generate_variant:Nn \int_gzero:N { c }

(End definition for \int_zero:N and \int_gzero:N. These functions are documented on page 80.)
```

```

\int_zero_new:N Create a register if needed, otherwise clear it.
\int_zero_new:c
\int_gzero_new:N
\int_gzero_new:c
6537 \cs_new_protected:Npn \int_zero_new:N #1
6538 { \int_if_exist:NTF #1 { \int_zero:N #1 } { \int_new:N #1 } }
6539 \cs_new_protected:Npn \int_gzero_new:N #1
6540 { \int_if_exist:NTF #1 { \int_gzero:N #1 } { \int_new:N #1 } }
6541 \cs_generate_variant:Nn \int_zero_new:N { c }
6542 \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 80.)
```

```

\int_set_eq:NN Setting equal means using one integer inside the set function of another. Check that
\int_set_eq:cN assigned integer is local/global. No need to check that the other one is defined as TEX
\int_set_eq:Nc does it for us.
\int_set_eq:cc
\int_gset_eq:NN
\int_gset_eq:cN
\int_gset_eq:Nc
\int_gset_eq:cc
6543 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
6544 \cs_new_protected:Npn \int_set_eq:NN #1#2 { #1 = #2 }
6545 \cs_generate_variant:Nn \int_set_eq:NN { c , Nc , cc }
6546 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
6547 \cs_new_protected:Npn \int_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
6548 \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 81.)
```

```

\int_if_exist_p:N Copies of the cs functions defined in l3basics.
\int_if_exist_p:c
\int_if_exist:NTF
\int_if_exist:cTF
6549 \prg_new_eq_conditional:NNNn \int_if_exist:N \cs_if_exist:N
6550 { TF , T , F , p }
6551 \prg_new_eq_conditional:NNNn \int_if_exist:c \cs_if_exist:c
6552 { TF , T , F , p }

(End definition for \int_if_exist:NTF. This function is documented on page 81.)
```

### 10.3 Setting and incrementing integers

Several functions here have a signature :Nn and are such that when debugging, the first argument should be checked to be a local/global variable and the second should be wrapped in code for an expression. The temporary function \\_\_int\_tmp:w finds the name #3 of the function being redefined and writes the appropriate patch.

```

6553 \cs_set_protected:Npn \__int_tmp:w #1#2#3
6554 {
6555   \__kernel_patch_args:nnnNNpn
6556   { #1 ##1 }
6557   { }
6558   { {##1} { \__kernel_chk_expr:nNnN {##2} \__int_eval:w { } #3 } }
6559   #2 #3
6560 }
```

```

\int_add:Nn Adding and subtracting to and from a counter. For each function, the debugging code
\int_add:cn produced by \__int_tmp:w checks that the assigned variable is correctly local/global and
\int_gadd:Nn wraps the expression in some checking code.

\int_gadd:cn
  \int_sub:Nn
  \int_sub:cn
\int_gsub:Nn
\int_gsub:cn
  6561 \__int_tmp:w \__kernel_chk_var_local:N
  6562 \cs_new_protected:Npn \int_add:Nn #1#2
  6563 { \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
  6564 \__int_tmp:w \__kernel_chk_var_local:N
  6565 \cs_new_protected:Npn \int_sub:Nn #1#2
  6566 { \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
  6567 \__int_tmp:w \__kernel_chk_var_global:N
  6568 \cs_new_protected:Npn \int_gadd:Nn #1#2
  6569 { \tex_global:D \tex_advance:D #1 by \__int_eval:w #2 \__int_eval_end: }
  6570 \__int_tmp:w \__kernel_chk_var_global:N
  6571 \cs_new_protected:Npn \int_gsub:Nn #1#2
  6572 { \tex_global:D \tex_advance:D #1 by - \__int_eval:w #2 \__int_eval_end: }
  6573 \cs_generate_variant:Nn \int_add:Nn { c }
  6574 \cs_generate_variant:Nn \int_gadd:Nn { c }
  6575 \cs_generate_variant:Nn \int_sub:Nn { c }
  6576 \cs_generate_variant:Nn \int_gsub:Nn { c }

```

(End definition for \int\_add:Nn and others. These functions are documented on page 81.)

```

\int_incr:N Incrementing and decrementing of integer registers is done with the following functions.
\int_incr:c
  6577 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\int_gincr:N
  6578 \cs_new_protected:Npn \int_incr:N #1
\int_gincr:c
  6579 { \tex_advance:D #1 \c_one_int }
\int_decr:N
  6580 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\int_decr:c
  6581 \cs_new_protected:Npn \int_decr:N #1
  6582 { \tex_advance:D #1 - \c_one_int }
\int_gdecr:N
  6583 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\int_gdecr:c
  6584 \cs_new_protected:Npn \int_gincr:N #1
  6585 { \tex_global:D \tex_advance:D #1 \c_one_int }
  6586 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
  6587 \cs_new_protected:Npn \int_gdecr:N #1
  6588 { \tex_global:D \tex_advance:D #1 - \c_one_int }
  6589 \cs_generate_variant:Nn \int_incr:N { c }
  6590 \cs_generate_variant:Nn \int_decr:N { c }
  6591 \cs_generate_variant:Nn \int_gincr:N { c }
  6592 \cs_generate_variant:Nn \int_gdecr:N { c }

```

(End definition for \int\_incr:N and others. These functions are documented on page 81.)

```

\int_set:Nn As integers are register-based TeX issues an error if they are not defined. Thus there is
\int_set:cn no need to check their existence as for token list variables. However, the code that checks
\int_gset:Nn whether the assignment is local or global is still needed.

\int_gset:cn
  6593 \__int_tmp:w \__kernel_chk_var_local:N
  6594 \cs_new_protected:Npn \int_set:Nn #1#2
  6595 { #1 ~ \__int_eval:w #2 \__int_eval_end: }
  6596 \__int_tmp:w \__kernel_chk_var_global:N
  6597 \cs_new_protected:Npn \int_gset:Nn #1#2
  6598 { \tex_global:D #1 ~ \__int_eval:w #2 \__int_eval_end: }
  6599 \cs_generate_variant:Nn \int_set:Nn { c }
  6600 \cs_generate_variant:Nn \int_gset:Nn { c }

```

(End definition for \int\_set:Nn and \int\_gset:Nn. These functions are documented on page 81.)

## 10.4 Using integers

\int\_use:N Here is how counters are accessed:  
\int\_use:c 6601 \cs\_new\_eq:NN \int\_use:N \tex\_the:D

We hand-code this for some speed gain:

```
6602 \%cs_generate_variant:Nn \int_use:N { c }
6603 \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 82.)

## 10.5 Integer expression conditionals

\\_int\_compare\_error: 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 T<sub>E</sub>X 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.

```
6604 \cs_new_protected:Npn \_int_compare_error:
6605 {
6606     \if_int_compare:w \c_zero_int \c_zero_int \fi:
6607     =
6608     \_int_compare_error:
6609 }
6610 \cs_new:Npn \_int_compare_error:Nw
6611     #1#2 \q_stop
6612 {
6613     {
6614         \c_zero_int \fi:
6615         \_kernel_msg_expandable_error:nnn
6616         { kernel } { unknown-comparison } {#1}
6617         \prg_return_false:
6618 }
```

(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 T<sub>E</sub>X conditional is taken (because of \reverse\_if:N), immediately returning false as the result of the test. There is no T<sub>E</sub>X 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  $\text{\TeX}$  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 `\q_stop` used to grab the entire argument when necessary.

```

6619 \prg_new_conditional:Npnn \int_compare:n #1 { p , T , F , TF }
6620   {
6621     \exp_after:wN \_int_compare:w
6622     \int_value:w \_int_eval:w #1 \_int_compare_error:
6623   }
6624 \cs_new:Npn \_int_compare:w #1 \_int_compare_error:
6625   {
6626     \exp_after:wN \if_false: \int_value:w
6627     \_int_compare:Nw #1 e { = nd_ } \q_stop
6628   }

```

The goal here is to find an *`\langle operand\rangle`* and a *`\langle comparison\rangle`*. The *`\langle operand\rangle`* 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  $\text{\TeX}$  into `\scan_stop:`, ignored thanks to `\unexpanded`, and `\_int_compare_error:Nw` raises an error.

```

6629 \cs_new:Npn \_int_compare:Nw #1#2 \q_stop
6630   {
6631     \exp_after:wN \_int_compare:NNw
6632     \_int_to_roman:w - 0 #2 \q_mark
6633     #1#2 \q_stop
6634   }
6635 \cs_new:Npn \_int_compare:NNw #1#2#3 \q_mark
6636   {
6637     \_kernel_exp_not:w
6638     \use:c
6639     {
6640       \_int_compare_ \token_to_str:N #1
6641       \if_meaning:w = #2 = \fi:
6642       :NNw
6643     }
6644     \_int_compare_error:Nw #1
6645   }

```

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 *operand*, and #3 is one of <, =, or >. As announced earlier, we leave the *operand* 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 *operand* #2 and the comparison #3, and call \\_\\_int\_compare:Nw to look for additional operands, after evaluating the following expression.

```

6646 \cs_new:cpn { __int_compare_end_=:NNw } #1#2#3 e #4 \q_stop
6647   {
6648     {#3} \exp_stop_f:
6649     \prg_return_false: \else: \prg_return_true: \fi:
6650   }
6651 \cs_new:Npn \__int_compare:nnN #1#2#3
6652   {
6653     {#2} \exp_stop_f:
6654     \prg_return_false: \exp_after:wN \use_none_delimit_by_q_stop:w
6655     \fi:
6656     #1 #2 #3 \exp_after:wN \__int_compare:Nw \int_value:w \__int_eval:w
6657   }

```

The actual comparisons are then simple function calls, using the relation as delimiter for a delimited argument and discarding \\_\\_int\_compare\_error:Nw *(token)* responsible for error detection.

```

6658 \cs_new:cpn { __int_compare_=:NNw } #1#2#3 =
6659   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
6660 \cs_new:cpn { __int_compare_<:NNw } #1#2#3 <
6661   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} < }
6662 \cs_new:cpn { __int_compare_>:NNw } #1#2#3 >
6663   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} > }
6664 \cs_new:cpn { __int_compare_==:NNw } #1#2#3 ==
6665   { \__int_compare:nnN { \reverse_if:N \if_int_compare:w } {#3} = }
6666 \cs_new:cpn { __int_compare_!=:NNw } #1#2#3 !=
6667   { \__int_compare:nnN { \if_int_compare:w } {#3} = }
6668 \cs_new:cpn { __int_compare_<=:NNw } #1#2#3 <=
6669   { \__int_compare:nnN { \if_int_compare:w } {#3} > }
6670 \cs_new:cpn { __int_compare_>=:NNw } #1#2#3 >=
6671   { \__int_compare:nnN { \if_int_compare:w } {#3} < }

```

(End definition for \int\_compare:nTF and others. This function is documented on page 83.)

`\int_compare_p:nNn` More efficient but less natural in typing.

```

\int_compare:nNnTF
6672 \__kernel_patch_conditional_args:nNnpnn
6673   {
6674     { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_compare:nNn }
6675     { \__int_eval_end: #2 }
6676     { \__kernel_chk_expr:nNnN {#3} \__int_eval:w { } \int_compare:nNn }
6677   }
6678 \prg_new_conditional:Npnn \int_compare:nNn #1#2#3 { p , T , F , TF }
6679   {
6680     \if_int_compare:w \__int_eval:w #1 #2 \__int_eval:w #3 \__int_eval_end:
6681       \prg_return_true:
6682     \else:
6683       \prg_return_false:
6684     \fi:
6685   }

```

(End definition for `\int_compare:nNnTF`. This function is documented on page 82.)

`\int_case:nn` For integer cases, the first task to fully expand the check condition. The over all idea is  
`\int_case:nnTF` then much the same as for `\tl_case:nn(TF)` as described in l3tl.

```
6686 \cs_new:Npn \int_case:nnTF #1
6687 {
6688     \exp:w
6689     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} }
6690 }
6691 \cs_new:Npn \int_case:nnT #1#2#3
6692 {
6693     \exp:w
6694     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} {#3} { }
6695 }
6696 \cs_new:Npn \int_case:nnF #1#2
6697 {
6698     \exp:w
6699     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { }
6700 }
6701 \cs_new:Npn \int_case:nn #1#2
6702 {
6703     \exp:w
6704     \exp_args:Nf \__int_case:nnTF { \int_eval:n {#1} } {#2} { } { }
6705 }
6706 \cs_new:Npn \__int_case:nnTF #1#2#3#4
6707 { \__int_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
6708 \cs_new:Npn \__int_case:nw #1#2#3
6709 {
6710     \int_compare:nNnTF {#1} = {#2}
6711     { \__int_case_end:nw {#3} }
6712     { \__int_case:nw {#1} }
6713 }
6714 \cs_new:Npn \__int_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
6715 { \exp_end: #1 #4 }
```

(End definition for `\int_case:nnTF` and others. This function is documented on page 84.)

`\int_if_odd_p:n` A predicate function.

```
6716 \__kernel_patch_conditional_args:nNNpnn
6717 { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_if_odd:n } }
6718 \prg_new_conditional:Npnn \int_if_odd:n #1 { p , T , F , TF }
6719 {
6720     \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
6721     \prg_return_true:
6722     \else:
6723     \prg_return_false:
6724     \fi:
6725 }
6726 \__kernel_patch_conditional_args:nNNpnn
6727 { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_if_even:n } }
6728 \prg_new_conditional:Npnn \int_if_even:n #1 { p , T , F , TF }
6729 {
6730     \reverse_if:N \if_int_odd:w \__int_eval:w #1 \__int_eval_end:
6731     \prg_return_true:
```

```

6732     \else:
6733         \prg_return_false:
6734     \fi:
6735 }
```

(End definition for `\int_if_odd:nTF` and `\int_if_even:nTF`. These functions are documented on page 84.)

## 10.6 Integer expression loops

`\int_while_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.

```

6736 \cs_new:Npn \int_while_do:nn #1#2
6737 {
6738     \int_compare:nT {#1}
6739     {
6740         #2
6741         \int_while_do:nn {#1} {#2}
6742     }
6743 }
6744 \cs_new:Npn \int_until_do:nn #1#2
6745 {
6746     \int_compare:nF {#1}
6747     {
6748         #2
6749         \int_until_do:nn {#1} {#2}
6750     }
6751 }
6752 \cs_new:Npn \int_do_while:nn #1#2
6753 {
6754     #2
6755     \int_compare:nT {#1}
6756     { \int_do_while:nn {#1} {#2} }
6757 }
6758 \cs_new:Npn \int_do_until:nn #1#2
6759 {
6760     #2
6761     \int_compare:nF {#1}
6762     { \int_do_until:nn {#1} {#2} }
6763 }
```

(End definition for `\int_while_do:nn` and others. These functions are documented on page 85.)

`\int_while_do:nNnn` As above but not using the more natural syntax.

```

6764 \cs_new:Npn \int_while_do:nNnn #1#2#3#4
6765 {
6766     \int_compare:nNnT {#1} #2 {#3}
6767     {
6768         #4
6769         \int_while_do:nNnn {#1} #2 {#3} {#4}
6770     }
6771 }
6772 \cs_new:Npn \int_until_do:nNnn #1#2#3#4
6773 {
```

```

6774     \int_compare:nNnF {#1} #2 {#3}
6775     {
6776         #4
6777         \int_until_do:nNnn {#1} #2 {#3} {#4}
6778     }
6779 }
6780 \cs_new:Npn \int_do_while:nNnn #1#2#3#4
6781 {
6782     #4
6783     \int_compare:nNnT {#1} #2 {#3}
6784     { \int_do_while:nNnn {#1} #2 {#3} {#4} }
6785 }
6786 \cs_new:Npn \int_do_until:nNnn #1#2#3#4
6787 {
6788     #4
6789     \int_compare:nNnF {#1} #2 {#3}
6790     { \int_do_until:nNnn {#1} #2 {#3} {#4} }
6791 }

```

(End definition for `\int_while_do:nNnn` and others. These functions are documented on page 85.)

## 10.7 Integer step functions

```
\int_step_function:nnN
\__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).

```

6792 \__kernel_patch_args:nNNpn
6793 {
6794 {
6795     \__kernel_chk_expr:nNnN {#1} \__int_eval:w { }
6796     \int_step_function:nnN
6797 }
6798 {
6799     \__kernel_chk_expr:nNnN {#2} \__int_eval:w { }
6800     \int_step_function:nnN
6801 }
6802 {
6803     \__kernel_chk_expr:nNnN {#3} \__int_eval:w { }
6804     \int_step_function:nnN
6805 }
6806 }
6807 \cs_new:Npn \int_step_function:nnnN #1#2#3
6808 {
6809     \exp_after:wN \__int_step:wwwN
6810     \int_value:w \__int_eval:w #1 \exp_after:wN ;
6811     \int_value:w \__int_eval:w #2 \exp_after:wN ;
6812     \int_value:w \__int_eval:w #3 ;
6813 }
6814 \cs_new:Npn \__int_step:wwwN #1; #2; #3; #4
6815 {
6816     \int_compare:nNnTF {#2} > \c_zero_int
6817     { \__int_step:NwnnN > }
```

```

6818 {
6819     \int_compare:nNnTF {#2} = \c_zero_int
6820     {
6821         \__kernel_msg_expandable_error:nnn
6822             { kernel } { zero-step } {#4}
6823             \prg_break:
6824     }
6825     { \__int_step:NwnnN < }
6826 }
6827 #1 ; {#2} {#3} #4
6828 \prg_break_point:
6829 }
6830 \cs_new:Npn \__int_step:NwnnN #1#2 ; #3#4#5
6831 {
6832     \if_int_compare:w #2 #1 #4 \exp_stop_f:
6833         \prg_break:n
6834     \fi:
6835     #5 {#2}
6836     \exp_after:wN \__int_step:NwnnN
6837     \exp_after:wN #1
6838     \int_value:w \__int_eval:w #2 + #3 ; {#3} {#4} #5
6839 }
6840 \cs_new:Npn \int_step_function:nN
6841     { \int_step_function:nnnN { 1 } { 1 } }
6842 \cs_new:Npn \int_step_function:nnN #1
6843     { \int_step_function:nnnN {#1} { 1 } }

```

(End definition for `\int_step_function:nnnN` and others. These functions are documented on page 86.)

```

\int_step_inline:nn
\int_step_inline:nnn
\int_step_inline:nnnn
\int_step_variable:nNn
\int_step_variable:nnNn
\int_step_variable:nnnNn
\int_step_variable: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 `\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.

```

6844 \cs_new_protected:Npn \int_step_inline:nn
6845     { \int_step_inline:nnnn { 1 } { 1 } }
6846 \cs_new_protected:Npn \int_step_inline:nnn #1
6847     { \int_step_inline:nnnn {#1} { 1 } }
6848 \cs_new_protected:Npn \int_step_inline:nnnn
6849     {
6850         \int_gincr:N \g_kernel_prg_map_int
6851         \exp_args:NNc \__int_step:NNnnnn
6852         \cs_gset_protected:Npn
6853             { __int_map_ \int_use:N \g_kernel_prg_map_int :w }
6854     }
6855 \cs_new_protected:Npn \int_step_variable:nNn
6856     { \int_step_variable:nnnNn { 1 } { 1 } }
6857 \cs_new_protected:Npn \int_step_variable:nnNn #1
6858     { \int_step_variable:nnnNn {#1} { 1 } }
6859 \cs_new_protected:Npn \int_step_variable:nnnNn #1#2#3#4#5
6860     {
6861         \int_gincr:N \g_kernel_prg_map_int
6862         \exp_args:NNc \__int_step:NNnnnn

```

```

6863   \cs_gset_protected:Npx
6864   { _int_map_ \int_use:N \g__kernel_prg_map_int :w }
6865   {#1}{#2}{#3}
6866   {
6867     \tl_set:Nn \exp_not:N #4 {##1}
6868     \exp_not:n {#5}
6869   }
6870 }
6871 \cs_new_protected:Npn \__int_step:NNnnnn #1#2#3#4#5#6
6872 {
6873   #1 #2 ##1 {#6}
6874   \int_step_function:nnnN {#3} {#4} {#5} #2
6875   \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
6876 }

```

(End definition for `\int_step_inline:nn` and others. These functions are documented on page 86.)

## 10.8 Formatting integers

`\int_to_arabic:n` Nothing exciting here.

```
6877 \cs_new_eq:NN \int_to_arabic:n \int_eval:n
```

(End definition for `\int_to_arabic:n`. This function is documented on page 87.)

`\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.

```

6878 \cs_new:Npn \int_to_symbols:nnn #1#2#3
6879 {
6880   \int_compare:nNnTF {#1} > {#2}
6881   {
6882     \exp_args:NNo \exp_args:No \__int_to_symbols:nnnn
6883     {
6884       \int_case:nn
6885         { 1 + \int_mod:nn { #1 - 1 } {#2} }
6886         {#3}
6887     }
6888     {#1} {#2} {#3}
6889   }
6890   { \int_case:nn {#1} {#3} }
6891 }
6892 \cs_new:Npn \__int_to_symbols:nnnn #1#2#3#4
6893 {
6894   \exp_args:Nf \int_to_symbols:nnn
6895   { \int_div_truncate:nn { #2 - 1 } {#3} } {#3} {#4}
6896   #1
6897 }

```

(End definition for `\int_to_symbols:nnn` and `\__int_to_symbols:nnnn`. This function is documented on page 87.)

**\int\_to\_alpha:n** These both use the above function with input functions that make sense for the alphabet  
**\int\_to\_Alph:n** in English.

```
6898 \cs_new:Npn \int_to_alpha:n #1
6899 {
6900     \int_to_symbols:nnn {#1} { 26 }
6901     {
6902         { 1 } { a }
6903         { 2 } { b }
6904         { 3 } { c }
6905         { 4 } { d }
6906         { 5 } { e }
6907         { 6 } { f }
6908         { 7 } { g }
6909         { 8 } { h }
6910         { 9 } { i }
6911         { 10 } { j }
6912         { 11 } { k }
6913         { 12 } { l }
6914         { 13 } { m }
6915         { 14 } { n }
6916         { 15 } { o }
6917         { 16 } { p }
6918         { 17 } { q }
6919         { 18 } { r }
6920         { 19 } { s }
6921         { 20 } { t }
6922         { 21 } { u }
6923         { 22 } { v }
6924         { 23 } { w }
6925         { 24 } { x }
6926         { 25 } { y }
6927         { 26 } { z }
6928     }
6929 }
6930 \cs_new:Npn \int_to_Alph:n #1
6931 {
6932     \int_to_symbols:nnn {#1} { 26 }
6933     {
6934         { 1 } { A }
6935         { 2 } { B }
6936         { 3 } { C }
6937         { 4 } { D }
6938         { 5 } { E }
6939         { 6 } { F }
6940         { 7 } { G }
6941         { 8 } { H }
6942         { 9 } { I }
6943         { 10 } { J }
6944         { 11 } { K }
6945         { 12 } { L }
6946         { 13 } { M }
6947         { 14 } { N }
6948         { 15 } { O }
6949         { 16 } { P }
```

```

6950      { 17 } { Q }
6951      { 18 } { R }
6952      { 19 } { S }
6953      { 20 } { T }
6954      { 21 } { U }
6955      { 22 } { V }
6956      { 23 } { W }
6957      { 24 } { X }
6958      { 25 } { Y }
6959      { 26 } { Z }
6960  }
6961 }

```

(End definition for `\int_to_alpha:n` and `\int_to_Alpha:n`. These functions are documented on page 87.)

**\int\_to\_base:nn** 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.

```

\__int_to_base:nn 6962 \cs_new:Npn \int_to_base:nn #1
\__int_to_Base:nn 6963  { \exp_args:Nf \__int_to_base:nn { \int_eval:n {#1} } }
\__int_to_base:nnN 6964 \cs_new:Npn \int_to_Base:nn #1
\__int_to_Base:nnN 6965  { \exp_args:Nf \__int_to_Base:nn { \int_eval:n {#1} } }
\__int_to_base:nnnN 6966 \cs_new:Npn \__int_to_base:nn #1#2
\__int_to_Base:nnnN 6967  {
\__int_to_letter:n 6968      \int_compare:nNnTF {#1} < 0
\__int_to_Letter:n 6969      { \exp_args:No \__int_to_base:nnN { \use_none:n #1 } {#2} - }
\__int_to_base:n 6970      { \__int_to_base:nnN {#1} {#2} \c_empty_t1 }
\__int_to_Letter:n 6971  }
\__int_to_Base:nn 6972 \cs_new:Npn \__int_to_Base:nn #1#2
\__int_to_base:n 6973  {
\__int_to_Base:nnN 6974      \int_compare:nNnTF {#1} < 0
\__int_to_base:n 6975      { \exp_args:No \__int_to_Base:nnN { \use_none:n #1 } {#2} - }
\__int_to_Base:nnN 6976      { \__int_to_Base:nnN {#1} {#2} \c_empty_t1 }
\__int_to_base:n 6977  }

```

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.

```

\__int_to_base:nnN 6978 \cs_new:Npn \__int_to_base:nnN #1#2#3
\__int_to_base:nnN 6979  {
\__int_to_base:nnN 6980      \int_compare:nNnTF {#1} < {#2}
\__int_to_base:nnN 6981      { \exp_last_unbraced:Nf #3 { \__int_to_letter:n {#1} } }
\__int_to_base:nnN 6982  {
\__int_to_base:nnN 6983      \exp_args:Nf \__int_to_base:nnnN
\__int_to_base:nnN 6984      { \__int_to_letter:n { \int_mod:nn {#1} {#2} } }
\__int_to_base:nnN 6985      {#1}
\__int_to_base:nnN 6986      {#2}
\__int_to_base:nnN 6987      #3
\__int_to_base:nnN 6988  }
\__int_to_base:nnN 6989  }
\__int_to_base:nnN 6990 \cs_new:Npn \__int_to_base:nnnN #1#2#3#4

```

```

6991   {
6992     \exp_args:Nf \__int_to_base:nnN
6993     { \int_div_truncate:nn {#2} {#3} }
6994     {#3}
6995     #4
6996     #1
6997   }
6998 \cs_new:Npn \__int_to_Base:nnN #1#2#3
6999   {
7000     \int_compare:nNnTF {#1} < {#2}
7001     { \exp_last_unbraced:Nf #3 { \__int_to_Letter:n {#1} } }
7002     {
7003       \exp_args:Nf \__int_to_Base:nnnN
7004       { \__int_to_Letter:n { \int_mod:nn {#1} {#2} } }
7005       {#1}
7006       {#2}
7007       #3
7008     }
7009   }
7010 \cs_new:Npn \__int_to_Base:nnnN #1#2#3#4
7011   {
7012     \exp_args:Nf \__int_to_Base:nnN
7013     { \int_div_truncate:nn {#2} {#3} }
7014     {#3}
7015     #4
7016     #1
7017   }

```

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:`.

```

7018 \cs_new:Npn \__int_to_letter:n #1
7019   {
7020     \exp_after:wN \exp_after:wN
7021     \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
7022       a
7023       \or: b
7024       \or: c
7025       \or: d
7026       \or: e
7027       \or: f
7028       \or: g
7029       \or: h
7030       \or: i
7031       \or: j
7032       \or: k
7033       \or: l
7034       \or: m
7035       \or: n
7036       \or: o
7037       \or: p

```

```

7038     \or: q
7039     \or: r
7040     \or: s
7041     \or: t
7042     \or: u
7043     \or: v
7044     \or: w
7045     \or: x
7046     \or: y
7047     \or: z
7048     \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
7049     \fi:
7050   }
7051 \cs_new:Npn \__int_to_Letter:n #1
7052 {
7053   \exp_after:wN \exp_after:wN
7054   \if_case:w \__int_eval:w #1 - 10 \__int_eval_end:
7055     A
7056     \or: B
7057     \or: C
7058     \or: D
7059     \or: E
7060     \or: F
7061     \or: G
7062     \or: H
7063     \or: I
7064     \or: J
7065     \or: K
7066     \or: L
7067     \or: M
7068     \or: N
7069     \or: O
7070     \or: P
7071     \or: Q
7072     \or: R
7073     \or: S
7074     \or: T
7075     \or: U
7076     \or: V
7077     \or: W
7078     \or: X
7079     \or: Y
7080     \or: Z
7081   \else: \int_value:w \__int_eval:w #1 \exp_after:wN \__int_eval_end:
7082   \fi:
7083 }

```

(End definition for `\int_to_base:nn` and others. These functions are documented on page 88.)

`\int_to_bin:n` Wrappers around the generic function.  
`\int_to_hex:n` 7084 `\cs_new:Npn \int_to_bin:n #1`  
`\int_to_Hex:n` 7085 `{ \int_to_base:nn {#1} { 2 } }`  
`\int_to_oct:n` 7086 `\cs_new:Npn \int_to_hex:n #1`  
 7087 `{ \int_to_base:nn {#1} { 16 } }`

```

7088 \cs_new:Npn \int_to_Hex:n #1
7089   { \int_to_Base:nn {#1} { 16 } }
7090 \cs_new:Npn \int_to_oct:n #1
7091   { \int_to_base:nn {#1} { 8 } }

```

(End definition for `\int_to_bin:n` and others. These functions are documented on page 88.)

<code>\int_to_roman:n</code> <code>\int_to_Roman:n</code> <code>\_int_to_roman:N</code> <code>\_int_to_roman:N</code> <code>\_int_to_roman_i:w</code> <code>\_int_to_roman_v:w</code> <code>\_int_to_roman_x:w</code> <code>\_int_to_roman_l:w</code> <code>\_int_to_roman_c:w</code> <code>\_int_to_roman_d:w</code> <code>\_int_to_roman_m:w</code> <code>\_int_to_roman_Q:w</code> <code>\_int_to_Roman_i:w</code> <code>\_int_to_Roman_v:w</code> <code>\_int_to_Roman_x:w</code> <code>\_int_to_Roman_l:w</code> <code>\_int_to_Roman_c:w</code> <code>\_int_to_Roman_d:w</code> <code>\_int_to_Roman_m:w</code> <code>\_int_to_Roman_Q:w</code>	The <code>\_int_to_roman:w</code> primitive creates tokens of category code 12 (other). Usually, what is actually wanted is letters. The approach here is to convert the output of the primitive into letters using appropriate control sequence names. That keeps everything expandable. The loop is terminated by the conversion of the Q.  <pre> 7092 \cs_new:Npn \int_to_roman:n #1 7093   { 7094     \exp_after:wN \_int_to_roman:N 7095       \_int_to_roman:w \int_eval:n {#1} Q 7096   } 7097 \cs_new:Npn \_int_to_roman:N #1 7098   { 7099     \use:c { _int_to_roman_ #1 :w } 7100     \_int_to_roman:N 7101   } 7102 \cs_new:Npn \int_to_Roman:n #1 7103   { 7104     \exp_after:wN \_int_to_Roman_aux:N 7105       \_int_to_roman:w \int_eval:n {#1} Q 7106   } 7107 \cs_new:Npn \_int_to_Roman_aux:N #1 7108   { 7109     \use:c { _int_to_Roman_ #1 :w } 7110     \_int_to_Roman_aux:N 7111   } 7112 \cs_new:Npn \_int_to_roman_i:w { i } 7113 \cs_new:Npn \_int_to_roman_v:w { v } 7114 \cs_new:Npn \_int_to_roman_x:w { x } 7115 \cs_new:Npn \_int_to_roman_l:w { l } 7116 \cs_new:Npn \_int_to_roman_c:w { c } 7117 \cs_new:Npn \_int_to_roman_d:w { d } 7118 \cs_new:Npn \_int_to_roman_m:w { m } 7119 \cs_new:Npn \_int_to_roman_Q:w { } 7120 \cs_new:Npn \_int_to_Roman_i:w { I } 7121 \cs_new:Npn \_int_to_Roman_v:w { V } 7122 \cs_new:Npn \_int_to_Roman_x:w { X } 7123 \cs_new:Npn \_int_to_Roman_l:w { L } 7124 \cs_new:Npn \_int_to_Roman_c:w { C } 7125 \cs_new:Npn \_int_to_Roman_d:w { D } 7126 \cs_new:Npn \_int_to_Roman_m:w { M } 7127 \cs_new:Npn \_int_to_Roman_Q:w { } </pre>
--	--

(End definition for `\int_to_roman:n` and others. These functions are documented on page 88.)

## 10.9 Converting from other formats to integers

`\_int_pass_signs:wn`  
`\_int_pass_signs_end:wn` Called as `\_int_pass_signs:wn signs and digits \q_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 `\q_stop`). More precisely, it deletes any `+` and passes any `-` to the input stream, hence should be called in an integer expression.

```

7128 \cs_new:Npn \__int_pass_signs:wn #1
7129 {
7130     \if:w + \if:w - \exp_not:N #1 + \fi: \exp_not:N #1
7131     \exp_after:wN \__int_pass_signs:wn
7132     \else:
7133         \exp_after:wN \__int_pass_signs_end:wn
7134         \exp_after:wN #1
7135     \fi:
7136 }
7137 \cs_new:Npn \__int_pass_signs_end:wn #1 \q_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.

```

7138 \cs_new:Npn \int_from_alpha:n #1
7139 {
7140     \int_eval:n
7141     {
7142         \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
7143         \q_stop { \__int_from_alpha:nN { 0 } }
7144         \q_recursion_tail \q_recursion_stop
7145     }
7146 }
7147 \cs_new:Npn \__int_from_alpha:nN #1#2
7148 {
7149     \quark_if_recursion_tail_stop_do:Nn #2 {#1}
7150     \exp_args:Nf \__int_from_alpha:nN
7151     { \int_eval:n { #1 * 26 + \__int_from_alpha:N #2 } }
7152 }
7153 \cs_new:Npn \__int_from_alpha:N #1
7154 { '#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 88.)

**\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.

```

7155 \cs_new:Npn \int_from_base:nn #1#2
7156 {
7157     \int_eval:n
7158     {
7159         \exp_after:wN \__int_pass_signs:wn \tl_to_str:n {#1}
7160         \q_stop { \__int_from_base:nnN { 0 } {#2} }
7161         \q_recursion_tail \q_recursion_stop
7162     }
7163 }
```

```

7164 \cs_new:Npn \__int_from_base:nnN #1#2#3
7165   {
7166     \quark_if_recursion_tail_stop_do:Nn #3 {#1}
7167     \exp_args:Nf \__int_from_base:nnN
7168       { \int_eval:n { #1 * #2 + \__int_from_base:N #3 } }
7169       {#2}
7170   }
7171 \cs_new:Npn \__int_from_base:N #1
7172   {
7173     \int_compare:nNnTF { '#1 } < { 58 }
7174     {#1}
7175     { '#1 - \int_compare:nNnTF { '#1 } < { 91 } { 55 } { 87 } }
7176   }

```

(End definition for `\int_from_base:nn`, `\__int_from_base:nnN`, and `\__int_from_base:N`. This function is documented on page 89.)

`\int_from_bin:n` Wrappers around the generic function.

```

7177 \cs_new:Npn \int_from_bin:n #1
7178   { \int_from_base:nn {#1} { 2 } }
7179 \cs_new:Npn \int_from_hex:n #1
7180   { \int_from_base:nn {#1} { 16 } }
7181 \cs_new:Npn \int_from_oct:n #1
7182   { \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 88.)

`\c_int_from_roman_i_int` Constants used to convert from Roman numerals to integers.

```

7183 \int_const:cn { c_int_from_roman_i_int } { 1 }
7184 \int_const:cn { c_int_from_roman_v_int } { 5 }
7185 \int_const:cn { c_int_from_roman_x_int } { 10 }
7186 \int_const:cn { c_int_from_roman_l_int } { 50 }
7187 \int_const:cn { c_int_from_roman_c_int } { 100 }
7188 \int_const:cn { c_int_from_roman_d_int } { 500 }
7189 \int_const:cn { c_int_from_roman_m_int } { 1000 }
7190 \int_const:cn { c_int_from_roman_I_int } { 1 }
7191 \int_const:cn { c_int_from_roman_V_int } { 5 }
7192 \int_const:cn { c_int_from_roman_X_int } { 10 }
7193 \int_const:cn { c_int_from_roman_L_int } { 50 }
7194 \int_const:cn { c_int_from_roman_C_int } { 100 }
7195 \int_const:cn { c_int_from_roman_D_int } { 500 }
7196 \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` 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`.

```

7197 \cs_new:Npn \int_from_roman:n #1
7198   {
7199     \int_eval:n
7200       {
7201         (

```

```

7202          0
7203          \exp_after:wN \__int_from_roman:NN \tl_to_str:n {#1}
7204          \q_recursion_tail \q_recursion_tail \q_recursion_stop
7205      )
7206  }
7207 }
7208 \cs_new:Npn \__int_from_roman:NN #1#2
7209 {
7210     \quark_if_recursion_tail_stop:N #1
7211     \int_if_exist:cF { c__int_from_roman_ #1 _int }
7212     { \__int_from_roman_error:w }
7213     \quark_if_recursion_tail_stop_do:Nn #2
7214     { + \use:c { c__int_from_roman_ #1 _int } }
7215     \int_if_exist:cF { c__int_from_roman_ #2 _int }
7216     { \__int_from_roman_error:w }
7217     \int_compare:nNnTF
7218     { \use:c { c__int_from_roman_ #1 _int } }
7219     <
7220     { \use:c { c__int_from_roman_ #2 _int } }
7221     {
7222         + \use:c { c__int_from_roman_ #2 _int }
7223         - \use:c { c__int_from_roman_ #1 _int }
7224         \__int_from_roman:NN
7225     }
7226     {
7227         + \use:c { c__int_from_roman_ #1 _int }
7228         \__int_from_roman:NN #2
7229     }
7230 }
7231 \cs_new:Npn \__int_from_roman_error:w #1 \q_recursion_stop #2
7232 { #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 89.)

## 10.10 Viewing integer

`\int_show:N` Diagnostics.

```

\int_show:c 7233 \cs_new_eq:NN \int_show:N \__kernel_register_show:N
\__int_show:nN 7234 \cs_generate_variant:Nn \int_show:N { c }

```

(End definition for `\int_show:N` and `\__int_show:nN`. This function is documented on page 89.)

`\int_show:n` We don't use the TeX primitive `\showthe` to show integer expressions: this gives a more unified output.

```

7235 \cs_new_protected:Npn \int_show:n
7236 { \msg_show_eval:Nn \int_eval:n }

```

(End definition for `\int_show:n`. This function is documented on page 90.)

`\int_log:N` Diagnostics.

```

\int_log:c 7237 \cs_new_eq:NN \int_log:N \__kernel_register_log:N
\__int_log:nN 7238 \cs_generate_variant:Nn \int_log:N { c }

```

(End definition for `\int_log:N`. This function is documented on page 90.)

`\int_log:n` Similar to `\int_show:n`.

```
7239 \cs_new_protected:Npn \int_log:n
7240   { \msg_log_eval:Nn \int_eval:n }
```

(End definition for `\int_log:n`. This function is documented on page 90.)

## 10.11 Random integers

`\int_rand:nn` Defined in `l3fp-random`.

(End definition for `\int_rand:nn`. This function is documented on page 89.)

## 10.12 Constant integers

`\c_zero_int` The zero is defined in `l3basics`.

```
7241 \int_const:Nn \c_zero_int { 1 }
```

(End definition for `\c_zero_int` and `\c_one_int`. These variables are documented on page 90.)

`\c_max_int` The largest number allowed is  $2^{31} - 1$

```
7242 \int_const:Nn \c_max_int { 2 147 483 647 }
```

(End definition for `\c_max_int`. This variable is documented on page 90.)

`\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].

```
7243 \int_const:Nn \c_max_char_int
7244   {
7245     \if_int_odd:w 0
7246       \cs_if_exist:NT \tex_luatexversion:D { 1 }
7247       \cs_if_exist:NT \tex_XeTeXversion:D { 1 } ~
7248       "10FFFF
7249     \else:
7250       "FF
7251     \fi:
7252 }
```

(End definition for `\c_max_char_int`. This variable is documented on page 90.)

## 10.13 Scratch integers

`\l_tmpa_int` We provide two local and two global scratch counters, maybe we need more or less.

```
7253 \int_new:N \l_tmpa_int
7254 \int_new:N \l_tmpb_int
7255 \int_new:N \g_tmpa_int
7256 \int_new:N \g_tmpb_int
```

(End definition for `\l_tmpa_int` and others. These variables are documented on page 90.)

## 10.14 Deprecated

```

\c_zero  Constants that are now deprecated. By default define them with \int_const:Nn.
\c_one   To deprecate them call for instance \__kernel_deprecation_error:Nnn \c_zero {0}
\c_two
\c_three {2019-12-31}. To redefine them (locally), use \__int_constdef:Nw, with an \exp-
\c_four  not:N construction because the constants themselves are outer at that point.
\c_five
\c_six
\c_seven
\c_eight
\c_nine
\c_ten
\c_eleven
\c_twelve
\c_thirteen
\c_fourteen
\c_fifteen
\c_sixteen
\c_thirty_two
\c_one_hundred
\c_two_hundred_fifty_five
\c_two_hundred_fifty_six
\c_one_thousand
\c_ten_thousand
\_int_deprecated_constants:nn
7257 \cs_new_protected:Npn \__int_DEPRECATED_CONSTANTS:nn #1#2
7258 {
7259     #1 \c_zero          { 0 } #2
7260     #1 \c_one           { 1 } #2
7261     #1 \c_two           { 2 } #2
7262     #1 \c_three         { 3 } #2
7263     #1 \c_four          { 4 } #2
7264     #1 \c_five          { 5 } #2
7265     #1 \c_six           { 6 } #2
7266     #1 \c_seven         { 7 } #2
7267     #1 \c_eight         { 8 } #2
7268     #1 \c_nine          { 9 } #2
7269     #1 \c_ten           { 10 } #2
7270    #1 \c_eleven         { 11 } #2
7271    #1 \c_twelve         { 12 } #2
7272    #1 \c_thirteen        { 13 } #2
7273    #1 \c_fourteen        { 14 } #2
7274    #1 \c_fifteen         { 15 } #2
7275    #1 \c_sixteen         { 16 } #2
7276    #1 \c_thirty_two       { 32 } #2
7277    #1 \c_one_hundred      { 100 } #2
7278    #1 \c_two_hundred_fifty_five { 255 } #2
7279    #1 \c_two_hundred_fifty_six { 256 } #2
7280    #1 \c_one_thousand      { 1000 } #2
7281    #1 \c_ten_thousand      { 10000 } #2
7282 }
7283 \__int_DEPRECATED_CONSTANTS:nn { \int_const:Nn } { }
7284 \__kernel_deprecation_code:nn
7285 {
7286     \__int_DEPRECATED_CONSTANTS:nn
7287     { \__kernel_deprecation_error:Nnn } { { 2019-12-31 } }
7288 }
7289 {
7290     \__int_DEPRECATED_CONSTANTS:nn
7291     {
7292         \exp_after:wN \use:nnn
7293         \exp_after:wN \__int_constdef:Nw \exp_not:N
7294     }
7295     { \exp_stop_f: }
7296 }

```

(End definition for \c\_zero and others.)

```

\_int_value:w Made public.
7297 \cs_new_eq:NN \__int_value:w \int_value:w
(End definition for \_int_value:w.)
7298 </initex | package>

```

## 11 l3flag implementation

```
7299  {*initex | package}
7300  <@@=flag>
```

The following test files are used for this code: *m3flag001*.

### 11.1 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!

```
7301  \cs_new_protected:Npn \flag_new:n #1
7302  {
7303    \cs_new:cpn { flag~#1 } ##1 ;
7304    { \exp_after:wN \use_none:n \cs:w flag~#1~##1 \cs_end: }
7305 }
```

(End definition for  $\text{\flag\_new:n}$ . This function is documented on page 93.)

**\flag\_clear:n**  $\_\_ \text{\flag\_clear:wn}$  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  $\text{\flag_new:n}$ .

```
7306  \_kernel_patch:nnNNpn
7307  { \exp_args:Nc \_kernel_chk_var_exist:N { flag~#1 } } { }
7308  \cs_new_protected:Npn \flag_clear:n #1 { \_flag_clear:wn 0 ; {#1} }
7309  \cs_new_protected:Npn \_flag_clear:wn #1 ; #2
7310  {
7311    \if_cs_exist:w flag~#2~#1 \cs_end:
7312      \cs_set_eq:cN { flag~#2~#1 } \tex_undefined:D
7313      \exp_after:wN \_flag_clear:wn
7314      \int_value:w \int_eval:w 1 + #1
7315    \else:
7316      \use_i:nnn
7317    \fi:
7318    ; {#2}
7319 }
```

(End definition for  $\text{\flag\_clear:n}$  and  $\text{\_flag\_clear:wn}$ . This function is documented on page 93.)

**\flag\_clear\_new:n** As for other datatypes, clear the  $\langle\text{flag}\rangle$  or create a new one, as appropriate.

```
7320  \cs_new_protected:Npn \flag_clear_new:n #1
7321  { \flag_if_exist:nTF {#1} { \flag_clear:n } { \flag_new:n } {#1} }
```

(End definition for  $\text{\flag\_clear\_new:n}$ . This function is documented on page 93.)

**\flag\_show:n** Show the height (terminal or log file) using appropriate l3msg auxiliaries.  
**\flag\_log:n**  $\_\_ \text{\flag\_show:Nn}$

```
7322  \cs_new_protected:Npn \flag_show:n { \_flag_show:Nn \tl_show:n }
7323  \cs_new_protected:Npn \flag_log:n { \_flag_show:Nn \tl_log:n }
7324  \cs_new_protected:Npn \_flag_show:Nn #1#2
7325  {
```

```

7326     \exp_args:Nc \_kernel_chk_defined:NT { flag~#2 }
7327     {
7328         \exp_args:Nx #1
7329         { \tl_to_str:n { flag~#2-height } = \flag_height:n {#2} }
7330     }
7331 }
```

(End definition for `\flag_show:n`, `\flag_log:n`, and `\_flag_show:Nn`. These functions are documented on page 93.)

## 11.2 Expandable flag commands

`\_flag_chk_exist:n`

Analogue of `\_kernel_chk_var_exist:N` for flags, and with an expandable error. We need to add checks by hand because flags are not implemented in terms of other variables. Not all functions need to be patched since some are defined in terms of others.

```

7332 (*package)
7333 \_kernel_if_debug:TF
7334 {
7335     \cs_new:Npn \_flag_chk_exist:n #1
7336     {
7337         \flag_if_exist:nF {#1}
7338         {
7339             \_kernel_msg_expandable_error:nnn
7340             { kernel } { bad-variable } { flag~#1~ }
7341         }
7342     }
7343 }
```

{ }

7345

(End definition for `\_flag_chk_exist:n`.)

`\flag_if_exist:p:n`

A flag exist if the corresponding trap `\flag {flag name}:n` is defined.

`\flag_if_exist:nTF`

```

7346 \prg_new_conditional:Npnn \flag_if_exist:n #1 { p , T , F , TF }
7347 {
7348     \cs_if_exist:cTF { flag~#1 }
7349     { \prg_return_true: } { \prg_return_false: }
7350 }
```

(End definition for `\flag_if_exist:nTF`. This function is documented on page 94.)

`\flag_if_raised:p:n`

Test if the flag has a non-zero height, by checking the 0 control sequence.

`\flag_if_raised:nTF`

```

7351 \_kernel_patch_conditional:nNNpnn { \_flag_chk_exist:n {#1} }
7352 \prg_new_conditional:Npnn \flag_if_raised:n #1 { p , T , F , TF }
7353 {
7354     \if_cs_exist:w flag~#1~0 \cs_end:
7355     \prg_return_true:
7356     \else:
7357     \prg_return_false:
7358     \fi:
7359 }
```

(End definition for `\flag_if_raised:nTF`. This function is documented on page 94.)

```

\flag_height:n Extract the value of the flag by going through all of the control sequences starting from
\__flag_height_loop:wn 0.
\__flag_height_end:wn

7360 \__kernel_patch:nnNNpn { \__flag_chk_exist:n {#1} } { }
7361 \cs_new:Npn \flag_height:n #1 { \__flag_height_loop:wn 0; {#1} }
7362 \cs_new:Npn \__flag_height_loop:wn #1 ; #2
7363 {
7364     \if_cs_exist:w flag~#2~#1 \cs_end:
7365         \exp_after:wN \__flag_height_loop:wn \int_value:w \int_eval:w 1 +
7366     \else:
7367         \exp_after:wN \__flag_height_end:wn
7368     \fi:
7369     #1 ; {#2}
7370 }
7371 \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 94.)

```

\flag\_raise:n Simply apply the trap to the height, after expanding the latter.

```

7372 \cs_new:Npn \flag_raise:n #1
7373 {
7374     \cs:w flag~#1 \exp_after:wN \cs_end:
7375     \int_value:w \flag_height:n {#1} ;
7376 }

```

(End definition for \flag\_raise:n. This function is documented on page 94.)

```
7377 
```

## 12 l3prg implementation

The following test files are used for this code: m3prg001.lvt, m3prg002.lvt, m3prg003.lvt.

```
7378 
```

### 12.1 Primitive conditionals

\if\_bool:N Those two primitive TeX conditionals are synonyms.

```

\if_predicate:w
7379 \cs_new_eq:NN \if_bool:N      \tex_ifodd:D
7380 \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 103.)

### 12.2 Defining a set of conditional functions

These are all defined in l3basics, as they are needed “early”. This is just a reminder!

(End definition for \prg\_set\_conditional:Npnn and others. These functions are documented on page 95.)

```

\prg_set_conditional:Npnn
\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
\prg_set_eq_conditional:NNn
\prg_new_eq_conditional:NNn
  \prg_return_true:
  \prg_return_false:

```

### 12.3 The boolean data type

7381 `\@C=bool`

`\bool_new:N` Boolean variables have to be initiated when they are created. Other than that there is not much to say here.

```
7382 \cs_new_protected:Npn \bool_new:N #1 { \cs_new_eq:NN #1 \c_false_bool }
7383 \cs_generate_variant:Nn \bool_new:N { c }
```

(End definition for `\bool_new:N`. This function is documented on page 97.)

`\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.

```
7384 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
7385 \cs_new_protected:Npn \bool_set_true:N #1
7386 { \cs_set_eq:NN #1 \c_true_bool }
7387 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
7388 \cs_new_protected:Npn \bool_set_false:N #1
7389 { \cs_set_eq:NN #1 \c_false_bool }
7390 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
7391 \cs_new_protected:Npn \bool_gset_true:N #1
7392 { \cs_gset_eq:NN #1 \c_true_bool }
7393 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
7394 \cs_new_protected:Npn \bool_gset_false:N #1
7395 { \cs_gset_eq:NN #1 \c_false_bool }
7396 \cs_generate_variant:Nn \bool_set_true:N { c }
7397 \cs_generate_variant:Nn \bool_set_false:N { c }
7398 \cs_generate_variant:Nn \bool_gset_true:N { c }
7399 \cs_generate_variant:Nn \bool_gset_false:N { c }
```

(End definition for `\bool_set_true:N` and others. These functions are documented on page 98.)

`\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.

```
7400 \cs_new_eq:NN \bool_set_eq:NN \tl_set_eq:NN
7401 \cs_new_eq:NN \bool_gset_eq:NN \tl_gset_eq:NN
7402 \cs_generate_variant:Nn \bool_set_eq:NN { Nc, cN, cc }
7403 \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 98.)

`\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.

```
7404 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
7405 \cs_new_protected:Npn \bool_set:Nn #1#2
7406 {
7407     \exp_last_unbraced:NNNF
7408     \tex_chardef:D #1 = { \bool_if_p:n {#2} }
7409 }
7410 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
7411 \cs_new_protected:Npn \bool_gset:Nn #1#2
```

```

7412   {
7413     \exp_last_unbraced:NNNNf
7414       \tex_global:D \tex_chardef:D #1 = { \bool_if_p:n {#2} }
7415   }
7416 \cs_generate_variant:Nn \bool_set:Nn { c }
7417 \cs_generate_variant:Nn \bool_gset:Nn { c }

```

(End definition for `\bool_set:Nn` and `\bool_gset:Nn`. These functions are documented on page 98.)

**\bool\_if\_p:N** Straight forward here. We could optimize here if we wanted to as the boolean can just be input directly.

```

7418 \prg_new_conditional:Npnn \bool_if:N #1 { p , T , F , TF }
\bool_if:cTF
7419   {
7420     \if_bool:N #1
7421       \prg_return_true:
7422     \else:
7423       \prg_return_false:
7424     \fi:
7425   }
7426 \prg_generate_conditional_variant:Nnm \bool_if:N { c } { p , T , F , TF }

```

(End definition for `\bool_if:NTF`. This function is documented on page 98.)

**\bool\_show:n** Show the truth value of the boolean, as `true` or `false`.

```

\bool_log:n
\__bool_to_str:n
7427 \cs_new_protected:Npn \bool_show:n
7428   { \msg_show_eval:Nn \__bool_to_str:n }
7429 \cs_new_protected:Npn \bool_log:n
7430   { \msg_log_eval:Nn \__bool_to_str:n }
7431 \cs_new:Npn \__bool_to_str:n #1
7432   { \bool_if:nTF {#1} { true } { false } }

```

(End definition for `\bool_show:n`, `\bool_log:n`, and `\__bool_to_str:n`. These functions are documented on page 98.)

**\bool\_show:N** Show the truth value of the boolean, as `true` or `false`.

```

\bool_show:c
\bool_log:N
\bool_log:c
\__bool_show:NN
7433 \cs_new_protected:Npn \bool_show:N { \__bool_show>NN \tl_show:n }
7434 \cs_generate_variant:Nn \bool_show:N { c }
7435 \cs_new_protected:Npn \bool_log:N { \__bool_show>NN \tl_log:n }
7436 \cs_generate_variant:Nn \bool_log:N { c }
7437 \cs_new_protected:Npn \__bool_show>NN #1#2
7438   {
7439     \__kernel_chk_defined:NT #2
7440     { \exp_args:Nx #1 { \token_to_str:N #2 = \__bool_to_str:n {#2} } }
7441   }

```

(End definition for `\bool_show:N`, `\bool_log:N`, and `\__bool_show:NN`. These functions are documented on page 98.)

**\l\_tmpa\_bool** A few booleans just if you need them.

```

\l_tmpb_bool
\g_tmpa_bool
\g_tmpb_bool
7442 \bool_new:N \l_tmpa_bool
7443 \bool_new:N \l_tmpb_bool
7444 \bool_new:N \g_tmpa_bool
7445 \bool_new:N \g_tmpb_bool

```

(End definition for `\l_tmpa_bool` and others. These variables are documented on page 99.)

```

\bool_if_exist_p:N Copies of the cs functions defined in !3basics.
\bool_if_exist_p:c
7446 \prg_new_eq_conditional:Nnn \bool_if_exist:N \cs_if_exist:N
7447 { TF , T , F , p }
\bool_if_exist:NTF
7448 \prg_new_eq_conditional:Nnn \bool_if_exist:c \cs_if_exist:c
7449 { TF , T , F , p }

```

(End definition for \bool\_if\_exist:NTF. This function is documented on page 99.)

## 12.4 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*.

```

7450 \prg_new_conditional:Npnn \bool_if:n #1 { T , F , TF }
7451 {
7452     \if_predicate:w \bool_if_p:n {#1}
7453         \prg_return_true:
7454     \else:
7455         \prg_return_false:
7456     \fi:
7457 }

```

(End definition for \bool\_if:nTF. This function is documented on page 100.)

```
\bool_if_p:n
\__bool_if_p:n
\__bool_if_p_aux:w
```

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. 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.

```
7458 \cs_new:Npn \bool_if_p:n { \exp_args:Nf \__bool_if_p:n }
7459 \cs_new:Npn \__bool_if_p:n #1
7460 {
7461     \tl_if_empty:oT { \use_none:nn #1 . } { \__bool_if_p_aux:w }
7462     \group_align_safe_begin:
7463     \exp_after:wN
7464     \group_align_safe_end:
7465     \exp:w \exp_end_continue_f:w % (
7466     \__bool_get_next:NN \use_i:nnnn #1 )
7467 }
7468 \cs_new:Npn \__bool_if_p_aux:w #1 \use_i:nnnn #2#3 {#2}
```

(End definition for \bool\_if\_p:n, \\_\_bool\_if\_p:n, and \\_\_bool\_if\_p\_aux:w. This function is documented on page 100.)

```
\__bool_get_next:NN
```

The GetNext operation. Its first argument is \use\_i:nnnn, \use\_i:i:nnnn, \use\_i: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.

```
7469 \cs_new:Npn \__bool_get_next:NN #1#2
7470 {
7471     \use:c
7472     {
7473         __bool_
7474         \if_meaning:w !#2 ! \else: \if_meaning:w (#2 ( \else: p \fi: \fi:
7475         :Nw
7476     }
7477     #1 #2
7478 }
```

(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.

```
7479 \cs_new:cpn { __bool_! :Nw } #1#2
7480 {
```

```

7481     \exp_after:wN \__bool_get_next:NN
7482     #1 \use_ii:nnnn \use_i:nnnn \use_iii:nnnn \use_iv:nnnn
7483 }

```

(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.

```

7484 \cs_new:cpn { __bool_( :Nw } #1#2
7485 {
7486     \exp_after:wN \__bool_choose:NNN \exp_after:wN #1
7487     \int_value:w \__bool_get_next:NN \use_i:nnnn
7488 }

```

(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.

```

7489 \cs_new:cpn { __bool_p:Nw } #1
7490     { \exp_after:wN \__bool_choose:NNN \exp_after:wN #1 \int_value:w }

```

(End definition for `\__bool_p:Nw.`)

`\__bool_choose:NNN` 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 &&`.

`\__bool_|_0:` When seeing ) the current subexpression is done, leave the appropriate boolean.  
`\__bool_|_1:` When seeing & in case 0 go into state 4, equivalent to having seen `\c_false_bool &&`.  
`\__bool_|_2:` 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 ||`.

```

7491 \cs_new:Npn \__bool_choose:NNN #1#2#3
7492 {
7493     \use:c
7494     {
7495         __bool_ \token_to_str:N #3 -
7496         #1 #2 { \if_meaning:w 0 #2 1 \else: 0 \fi: } 2 0 :
7497     }
7498 }
7499 \cs_new:cpn { __bool_|_0: } { \c_false_bool }
7500 \cs_new:cpn { __bool_|_1: } { \c_true_bool }
7501 \cs_new:cpn { __bool_|_2: } { \c_true_bool }
7502 \cs_new:cpn { __bool_&_0: } & { \__bool_get_next:NN \use_iv:nnnn }
7503 \cs_new:cpn { __bool_&_1: } & { \__bool_get_next:NN \use_i:nnnn }
7504 \cs_new:cpn { __bool_&_2: } & { \__bool_get_next:NN \use_iii:nnnn }

```

```

7505 \cs_new:cpn { __bool_|_0: } | { \__bool_get_next:NN \use_i:nnnn }
7506 \cs_new:cpn { __bool_|_1: } | { \__bool_get_next:NN \use_iii:nnnn }
7507 \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` 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`.

`\bool_lazy_all:nTF`

`\__bool_lazy_all:n`

```

7508 \cs_new:Npn \bool_lazy_all_p:n #1
7509   { \__bool_lazy_all:n #1 \q_recursion_tail \q_recursion_stop }
7510 \prg_new_conditional:Npnn \bool_lazy_all:n #1 { T , F , TF }
7511   {
7512     \if_predicate:w \bool_lazy_all_p:n {#1}
7513       \prg_return_true:
7514     \else:
7515       \prg_return_false:
7516     \fi:
7517   }
7518 \cs_new:Npn \__bool_lazy_all:n #1
7519   {
7520     \quark_if_recursion_tail_stop_do:nn {#1} { \c_true_bool }
7521     \bool_if:nF {#1}
7522       { \use_i_delimit_by_q_recursion_stop:nw { \c_false_bool } }
7523     \__bool_lazy_all:n
7524   }

```

(End definition for `\bool_lazy_all:nTF` and `\__bool_lazy_all:n`. This function is documented on page 100.)

`\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 TeX conditionals.

`\bool_lazy_and:nnTF`

```

7525 \prg_new_conditional:Npnn \bool_lazy_and:nn #1#2 { p , T , F , TF }
7526   {
7527     \if_predicate:w
7528       \bool_if:nTF {#1} { \bool_if_p:n {#2} } { \c_false_bool }
7529       \prg_return_true:
7530     \else:
7531       \prg_return_false:
7532     \fi:
7533   }

```

(End definition for `\bool_lazy_and:nnTF`. This function is documented on page 100.)

`\bool_lazy_any:p:n`

`\bool_lazy_any:nTF`

`\__bool_lazy_any: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`.

```

7534 \cs_new:Npn \bool_lazy_any_p:n #1
7535   { \__bool_lazy_any:n #1 \q_recursion_tail \q_recursion_stop }
7536 \prg_new_conditional:Npnn \bool_lazy_any:n #1 { T , F , TF }
7537   {
7538     \if_predicate:w \bool_lazy_any_p:n {#1}
7539       \prg_return_true:
7540     \else:
7541       \prg_return_false:

```

```

7542     \fi:
7543   }
7544 \cs_new:Npn \__bool_lazy_any:n #1
7545 {
7546   \quark_if_recursion_tail_stop_do:nn {#1} { \c_false_bool }
7547   \bool_if:nT {#1}
7548   { \use_i_delimit_by_q_recursion_stop:nw { \c_true_bool } }
7549   \__bool_lazy_any:n
7550 }
```

(End definition for `\bool_lazy_any:nTF` and `\__bool_lazy_any:n`. This function is documented on page 100.)

`\bool_lazy_or_p:nn` Only evaluate the second expression if the first is `false`.

`\bool_lazy_or:nnTF`

```

7551 \prg_new_conditional:Npnn \bool_lazy_or:nn #1#2 { p , T , F , TF }
7552 {
7553   \if_predicate:w
7554     \bool_if:nTF {#1} { \c_true_bool } { \bool_if_p:n {#2} }
7555     \prg_return_true:
7556   \else:
7557     \prg_return_false:
7558   \fi:
7559 }
```

(End definition for `\bool_lazy_or:nnTF`. This function is documented on page 101.)

`\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.

```
7560 \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 101.)

`\bool_xor_p:nn` Exclusive or. If the boolean expressions have same truth value, return `false`, otherwise return `true`.

```

7561 \prg_new_conditional:Npnn \bool_xor:nn #1#2 { p , T , F , TF }
7562 {
7563   \bool_if:nT {#1} \reverse_if:N
7564   \if_predicate:w \bool_if_p:n {#2}
7565     \prg_return_true:
7566   \else:
7567     \prg_return_false:
7568   \fi:
7569 }
```

(End definition for `\bool_xor:nnTF`. This function is documented on page 101.)

## 12.5 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_while_do:cn`

```

7570 \cs_new:Npn \bool_while_do:Nn #1#2
7571   { \bool_if:NT #1 { #2 \bool_while_do:Nn #1 {#2} } }
```

```

7572 \cs_new:Npn \bool_until_do:Nn #1#2
7573   { \bool_if:NF #1 { #2 \bool_until_do:Nn #1 {#2} } }
7574 \cs_generate_variant:Nn \bool_while_do:Nn { c }
7575 \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 101.)

`\bool_do_while:Nn`  
`\bool_do_while:cn` 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.

```

7576 \cs_new:Npn \bool_do_while:Nn #1#2
7577   { #2 \bool_if:NT #1 { \bool_do_while:Nn #1 {#2} } }
7578 \cs_new:Npn \bool_do_until:Nn #1#2
7579   { #2 \bool_if:NF #1 { \bool_do_until:Nn #1 {#2} } }
7580 \cs_generate_variant:Nn \bool_do_while:Nn { c }
7581 \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 101.)

`\bool_while_do:nn`  
`\bool_do_while:nn`  
`\bool_until_do:nn`  
`\bool_do_until:nn` Loop functions with the test either before or after the first body expansion.

```

7582 \cs_new:Npn \bool_while_do:nn #1#2
7583   {
7584     \bool_if:nT {#1}
7585     {
7586       #2
7587       \bool_while_do:nn {#1} {#2}
7588     }
7589   }
7590 \cs_new:Npn \bool_do_while:nn #1#2
7591   {
7592     #2
7593     \bool_if:nT {#1} { \bool_do_while:nn {#1} {#2} }
7594   }
7595 \cs_new:Npn \bool_until_do:nn #1#2
7596   {
7597     \bool_if:nF {#1}
7598     {
7599       #2
7600       \bool_until_do:nn {#1} {#2}
7601     }
7602   }
7603 \cs_new:Npn \bool_do_until:nn #1#2
7604   {
7605     #2
7606     \bool_if:nF {#1} { \bool_do_until:nn {#1} {#2} }
7607   }

```

(End definition for `\bool_while_do:nn` and others. These functions are documented on page 102.)

## 12.6 Producing multiple copies

7608 `<@=prg>`

`\prg_replicate:nn` This function uses a cascading csname technique by David Kastrup (who else :-)

```

\__prg_replicate:N
\__prg_replicate_first:N
  \__prg_replicate_
  \__prg_replicate_0:n
  \__prg_replicate_1:n
  \__prg_replicate_2:n
  \__prg_replicate_3:n
  \__prg_replicate_4:n
  \__prg_replicate_5:n
  \__prg_replicate_6:n
  \__prg_replicate_7:n
  \__prg_replicate_8:n
  \__prg_replicate_9:n

```

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} {\langle code\rangle} }. 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.

```

7609 \cs_new:Npn \prg_replicate:nn #1
7610   {
7611     \exp:w
7612       \exp_after:wN \__prg_replicate_first:N
7613       \int_value:w \int_eval:n {#1}
7614       \cs_end:
7615     }
7616 \cs_new:Npn \__prg_replicate:N #1
7617   { \cs:w __prg_replicate_#1 :n \__prg_replicate:N }
7618 \cs_new:Npn \__prg_replicate_first:N #1
7619   { \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.

```

7620 \cs_new:Npn \__prg_replicate_ :n #1 { \cs_end: }
7621 \cs_new:cpn { __prg_replicate_0:n } #1
7622   { \cs_end: {#1#1#1#1#1#1#1#1#1} }
7623 \cs_new:cpn { __prg_replicate_1:n } #1
7624   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1 }
7625 \cs_new:cpn { __prg_replicate_2:n } #1
7626   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1 }
7627 \cs_new:cpn { __prg_replicate_3:n } #1
7628   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1 }
7629 \cs_new:cpn { __prg_replicate_4:n } #1
7630   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1 }
7631 \cs_new:cpn { __prg_replicate_5:n } #1
7632   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1 }
7633 \cs_new:cpn { __prg_replicate_6:n } #1
7634   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1 }
7635 \cs_new:cpn { __prg_replicate_7:n } #1
7636   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1 }
7637 \cs_new:cpn { __prg_replicate_8:n } #1
7638   { \cs_end: {#1#1#1#1#1#1#1#1#1} #1#1#1#1#1#1#1#1 }

```

```

7639 \cs_new:cpn { __prg_replicate_9:n } #1
7640   { \cs_end: {#1#1#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...

```

7641 \cs_new:cpn { __prg_replicate_first_:-n } #1
7642   {
7643     \exp_end:
7644     \__kernel_msg_expandable_error:nn { kernel } { negative-replication }
7645   }
7646 \cs_new:cpn { __prg_replicate_first_0:n } #1 { \exp_end: }
7647 \cs_new:cpn { __prg_replicate_first_1:n } #1 { \exp_end: #1 }
7648 \cs_new:cpn { __prg_replicate_first_2:n } #1 { \exp_end: #1#1 }
7649 \cs_new:cpn { __prg_replicate_first_3:n } #1 { \exp_end: #1#1#1 }
7650 \cs_new:cpn { __prg_replicate_first_4:n } #1 { \exp_end: #1#1#1#1 }
7651 \cs_new:cpn { __prg_replicate_first_5:n } #1 { \exp_end: #1#1#1#1#1 }
7652 \cs_new:cpn { __prg_replicate_first_6:n } #1 { \exp_end: #1#1#1#1#1#1 }
7653 \cs_new:cpn { __prg_replicate_first_7:n } #1 { \exp_end: #1#1#1#1#1#1#1 }
7654 \cs_new:cpn { __prg_replicate_first_8:n } #1 { \exp_end: #1#1#1#1#1#1#1#1 }
7655 \cs_new:cpn { __prg_replicate_first_9:n } #1
7656   { \exp_end: #1#1#1#1#1#1#1#1#1#1 }

```

(End definition for `\prg_replicate:nn` and others. This function is documented on page 102.)

## 12.7 Detecting TeX's mode

`\mode_if_vertical:p`: For testing vertical mode. Strikes me here on the bus with David, that as long as we `\mode_if_vertical:TF` 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.

```

7657 \prg_new_conditional:Npnn \mode_if_vertical: { p , T , F , TF }
7658   { \if_mode_vertical: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_vertical:TF`. This function is documented on page 102.)

`\mode_if_horizontal:p`: For testing horizontal mode.

```

7659 \prg_new_conditional:Npnn \mode_if_horizontal: { p , T , F , TF }
7660   { \if_mode_horizontal: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_horizontal:TF`. This function is documented on page 102.)

`\mode_if_inner:p`: For testing inner mode.

```

7661 \prg_new_conditional:Npnn \mode_if_inner: { p , T , F , TF }
7662   { \if_mode_inner: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_inner:TF`. This function is documented on page 102.)

`\mode_if_math:p`: For testing math mode. At the beginning of an alignment cell, this should be used only `\mode_if_math:TF` inside a non-expandable function.

```

7663 \prg_new_conditional:Npnn \mode_if_math: { p , T , F , TF }
7664   { \if_mode_math: \prg_return_true: \else: \prg_return_false: \fi: }

```

(End definition for `\mode_if_math:TF`. This function is documented on page 102.)

## 12.8 Internal programming functions

\group\_align\_safe\_begin:  
\group\_align\_safe\_end:

TeX's alignment structures present many problems. As Knuth says himself in *TeX: The Program*: "It's sort of a miracle whenever \halign or \valign work, [...]" One problem relates to commands that internally issues 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 TeX 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 code documented only in Appendix D of *The TeXbook*... 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.

```
7665 \cs_new:Npn \group_align_safe_begin:
7666   { \if_int_compare:w \if_false: { \fi: } = \c_zero_int \fi: }
7667 \cs_new:Npn \group_align_safe_end:
7668   { \if_int_compare:w '{ = \c_zero_int } \fi: }
```

(End definition for \group\_align\_safe\_begin: and \group\_align\_safe\_end:. These functions are documented on page 104.)

7669 ⟨@=prg⟩

\g\_kernel\_prg\_map\_int A nesting counter for mapping.

```
7670 \int_new:N \g_kernel_prg_map_int
```

(End definition for \g\_kernel\_prg\_map\_int.)

\prg\_break\_point:Nn These are defined in l3basics, 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 103.)

\prg\_break\_point: Also done in l3basics as in format mode these are needed within l3alloc.

\prg\_break:  
\prg\_break:n  
(End definition for \prg\_break\_point:, \prg\_break:, and \prg\_break:n. These functions are documented on page 103.)

## 12.9 Deprecated functions

\\_\_prg\_break\_point:Nn  
\\_\_prg\_break\_point:  
\\_\_prg\_map\_break:Nn  
\\_\_prg\_map\_break:  
\\_\_prg\_break:  
\\_\_prg\_break:n

Made public, but used by a few third-parties. It's not possible to perfectly support a mixture of \\_\_prg\_map\_break:Nn and \prg\_map\_break:Nn because they use different delimiters. The following code only breaks if someone tries to break from two "old-style" \\_\_prg\_map\_break:Nn ... \\_\_prg\_break\_point:Nn mappings in one go. Basically, the \\_\_prg\_map\_break:Nn converts a single \\_\_prg\_break\_point:Nn to \prg\_break\_point:Nn, and that delimiter had better be the right one. Then we call \prg\_map\_break:Nn which may end up breaking intermediate looks in the (unbraced) argument #1. It is essential to define the break\_point functions before the corresponding break functions: otherwise \debug\_on:n {deprecation} \debug\_off:n {deprecation} would break when trying to restore the definitions because they would involve deprecated commands whose definition has not yet been restored.

```

7671 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break_point:Nn }
7672 \cs_new:Npn \__prg_break_point:Nn { \prg_break_point:Nn }
7673 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break_point: }
7674 \cs_new:Npn \__prg_break_point: { \prg_break_point: }
7675 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_map_break:Nn }
7676 \cs_new:Npn \__prg_map_break:Nn #1 \__prg_break_point:Nn
7677 { \prg_map_break:Nn #1 \prg_break_point:Nn }
7678 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break: }
7679 \cs_new:Npn \__prg_break: #1 \__prg_break_point: { }
7680 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \prg_break:n }
7681 \cs_new:Npn \__prg_break:n #1#2 \__prg_break_point: {#1}

(End definition for \__prg_break_point:Nn and others.)

```

7682 ⟨/initex | package⟩

## 13 I3sys implementation

```

7683 ⟨*initex | package⟩
7684 ⟨@@=sys⟩

```

### 13.1 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 needs to actually contain the text of the job name rather than the name of the primitive, of course.

```

7685 ⟨*initex⟩
7686 \tex_everyjob:D \exp_after:wN
7687 {
7688   \tex_the:D \tex_everyjob:D
7689   \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
7690 }
7691 ⟨/initex⟩
7692 ⟨*package⟩
7693 \str_const:Nx \c_sys_jobname_str { \tex_jobname:D }
7694 ⟨/package⟩

```

(End definition for \c\_sys\_jobname\_str. This variable is documented on page 105.)

### 13.2 Time and date

**\c\_sys\_minute\_int** Copies of the information provided by T<sub>E</sub>X

```

7695 \int_const:Nn \c_sys_minute_int
7696 { \int_mod:nn { \tex_time:D } { 60 } }
7697 \int_const:Nn \c_sys_hour_int
7698 { \int_div_truncate:nn { \tex_time:D } { 60 } }
7699 \int_const:Nn \c_sys_day_int { \tex_day:D }
7700 \int_const:Nn \c_sys_month_int { \tex_month:D }
7701 \int_const:Nn \c_sys_year_int { \tex_year:D }

```

(End definition for \c\_sys\_minute\_int and others. These variables are documented on page 105.)

### 13.3 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.

```

7702 \cs_new_protected:Npn \__sys_const:nn #1#2
7703 {
7704     \bool_if:nTF {#2}
7705     {
7706         \cs_new_eq:cN { #1 :T } \use:n
7707         \cs_new_eq:cN { #1 :F } \use_none:n
7708         \cs_new_eq:cN { #1 :TF } \use_i:nn
7709         \cs_new_eq:cN { #1 _p: } \c_true_bool
7710     }
7711     {
7712         \cs_new_eq:cN { #1 :T } \use_none:n
7713         \cs_new_eq:cN { #1 :F } \use:n
7714         \cs_new_eq:cN { #1 :TF } \use_ii:nn
7715         \cs_new_eq:cN { #1 _p: } \c_false_bool
7716     }
7717 }
```

(End definition for `\__sys_const:nn`.)

`\sys_if_engine_luatex_p:` `\sys_if_engine_luatex:TF`  
`\sys_if_engine_pdftex_p:` `\sys_if_engine_pdftex:TF` Set up the engine tests on the basis exactly one test should be true. Mainly a case of looking for the appropriate marker primitive. For upTeX, there is a complexity in that setting `-kanji-internal=sjis` or `-kanji-internal=euc` effective makes it more like pTeX. In those cases we therefore report pTeX rather than upTeX.

```

7718 \str_const:Nx \c_sys_engine_str
7719 {
7720     \cs_if_exist:NT \tex_luatexversion:D { luatex }
7721     \cs_if_exist:NT \tex_pdftexversion:D { pdftex }
7722     \cs_if_exist:NT \tex_kanjiskip:D
7723     {
7724         \bool_lazy_and:nnTF
7725             { \cs_if_exist_p:N \tex_disablecjktoken:D }
7726             { \int_compare_p:nNn { \tex_jis:D "2121 } = { "3000 } }
7727             { uptex }
7728             { ptex }
7729     }
7730     \cs_if_exist:NT \tex_XeTeXversion:D { xetex }
7731 }
7732 \tl_map_inline:nn { { luatex } { pdftex } { ptex } { uptex } { xetex } }
7733 {
7734     \__sys_const:nn { sys_if_engine_ #1 }
7735     { \str_if_eq_p:Vn \c_sys_engine_str {#1} }
7736 }
```

(End definition for `\sys_if_engine_luatex:TF` and others. These functions are documented on page 105.)

### 13.4 Detecting the output

`\sys_if_output_dvi_p:` `\sys_if_output_dvi:TF` This is a simple enough concept: the two views here are complementary.

`\sys_if_output_pdf_p:` `\sys_if_output_pdf:TF`

`\c_sys_output_str`

```

7738   {
7739     \int_compare:nNnTF
7740       { \cs_if_exist_use:N \tex_pdfoutput:D { 0 } } > { 0 }
7741         { pdf }
7742         { dvi }
7743     }
7744   \__sys_const:nn { sys_if_output_dvi }
7745     { \str_if_eq_p:Vn \c_sys_output_str { dvi } }
7746   \__sys_const:nn { sys_if_output_pdf }
7747     { \str_if_eq_p:Vn \c_sys_output_str { pdf } }

```

(End definition for `\sys_if_output_dvi:TF`, `\sys_if_output_pdf:TF`, and `\c_sys_output_str`. These functions are documented on page [106](#).)

### 13.5 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**

```

7748   \__sys_const:nn { sys_if_rand_exist }
7749     { \cs_if_exist_p:N \tex_uniformdeviate:D }

```

(End definition for `\sys_if_rand_exist:TF`. This function is documented on page [252](#).)

```
7750  ⟨/initex | package⟩
```

## 14 l3clist implementation

The following test files are used for this code: `m3clist002`.

```
7751  ⟨*initex | package⟩
```

```
7752  ⟨@=clist⟩
```

**\c\_empty\_clist** An empty comma list is simply an empty token list.

```
7753  \cs_new_eq:NN \c_empty_clist \c_empty_tl
```

(End definition for `\c_empty_clist`. This variable is documented on page [116](#).)

**\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`

```
7754  \tl_new:N \l__clist_internal_clist
```

(End definition for `\l__clist_internal_clist`.)

**\\_\_clist\_tmp:w** A temporary function for various purposes.

```
7755  \cs_new_protected:Npn \__clist_tmp:w { }
```

(End definition for `\__clist_tmp:w`.)

## 14.1 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.

```
7756 \cs_new:Npn \__clist_trim_next:w #1 ,
7757 {
7758     \tl_if_empty:oTF { \use_none:nn #1 ? }
7759     { \__clist_trim_next:w \prg_do_nothing: }
7760     { \tl_trim_spaces_apply:oN {#1} \exp_end: }
7761 }
```

(End definition for `\__clist_trim_next:w`.)

`\__clist_sanitize:n`

`\__clist_sanitize:Nn`

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.

```
7762 \cs_new:Npn \__clist_sanitize:n #1
7763 {
7764     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN \c_empty_tl
7765     \exp:w \__clist_trim_next:w \prg_do_nothing:
7766     #1 , \q_recursion_tail , \q_recursion_stop
7767 }
7768 \cs_new:Npn \__clist_sanitize:Nn #1#2
7769 {
7770     \quark_if_recursion_tail_stop:n {#2}
7771     #1 \__clist_wrap_item:w #2 ,
7772     \exp_after:wN \__clist_sanitize:Nn \exp_after:wN ,
7773     \exp:w \__clist_trim_next:w \prg_do_nothing:
7774 }
```

(End definition for `\__clist_sanitize:n` and `\__clist_sanitize:Nn`.)

`\__clist_if_wrap:nTF`

`\__clist_if_wrap:w`

True if the argument must be wrapped to avoid getting altered by some clist operations. 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.

All `\clist` functions go through the same test when they need to determine whether to brace an item, so it is not a problem that this test has false positives such as “`\q_mark ?`”. If the argument starts or end 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.

```

7775 \prg_new_conditional:Npnn \__clist_if_wrap:n #1 { TF }
7776   {
7777     \tl_if_empty:oTF
7778     {
7779       \__clist_if_wrap:w
7780       \q_mark ? #1 ~ \q_mark ? ~ #1 \q_mark , ~ \q_mark #1 ,
7781     }
7782     {
7783       \tl_if_head_is_group:nTF { #1 { } }
7784       {
7785         \tl_if_empty:nTF {#1}
7786         { \prg_return_true: }
7787         {
7788           \tl_if_empty:oTF { \use_none:n #1}
7789           { \prg_return_true: }
7790           { \prg_return_false: }
7791         }
7792       }
7793       { \prg_return_false: }
7794     }
7795     { \prg_return_true: }
7796   }
7797 \cs_new:Npn \__clist_if_wrap:w #1 \q_mark ? ~ #2 ~ \q_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.

```

7798 \cs_new:Npn \__clist_wrap_item:w #1 ,
7799   { \__clist_if_wrap:nTF {#1} { \exp_not:n { {#1} } } { \exp_not:n {#1} } }

```

(End definition for `\__clist_wrap_item:w`.)

## 14.2 Allocation and initialisation

`\clist_new:N` Internally, comma lists are just token lists.

```

7800 \cs_new_eq:NN \clist_new:N \tl_new:N
7801 \cs_new_eq:NN \clist_new:c \tl_new:c

```

(End definition for `\clist_new:N`. This function is documented on page 107.)

`\clist_const:Nn` Creating and initializing a constant comma list is done by sanitizing all items (stripping spaces and braces).

```

7802 \cs_new_protected:Npn \clist_const:Nn #1#2
7803   { \tl_const:Nx #1 { \__clist_sanitize:n {#2} } }
7804 \cs_generate_variant:Nn \clist_const:Nn { c , Nx , cx }

```

(End definition for `\clist_const:Nn`. This function is documented on page 108.)

`\clist_clear:N` Clearing comma lists is just the same as clearing token lists.

```

7805 \cs_new_eq:NN \clist_clear:N \tl_clear:N
7806 \cs_new_eq:NN \clist_clear:c \tl_clear:c
7807 \cs_new_eq:NN \clist_gclear:N \tl_gclear:N
7808 \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 108.)

`\clist_clear_new:N` Once again a copy from the token list functions.  
`\clist_clear_new:c` 7809 `\cs_new_eq:NN \clist_clear_new:N \tl_clear_new:N`  
`\clist_gclear_new:N` 7810 `\cs_new_eq:NN \clist_clear_new:c \tl_clear_new:c`  
`\clist_gclear_new:c` 7811 `\cs_new_eq:NN \clist_gclear_new:N \tl_gclear_new:N`  
7812 `\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 108.)

`\clist_set_eq:NN` Once again, these are simple copies from the token list functions.  
`\clist_set_eq:cN` 7813 `\cs_new_eq:NN \clist_set_eq:NN \tl_set_eq:NN`  
`\clist_set_eq:Nc` 7814 `\cs_new_eq:NN \clist_set_eq:Nc \tl_set_eq:Nc`  
`\clist_set_eq:cc` 7815 `\cs_new_eq:NN \clist_set_eq:cc \tl_set_eq:cc`  
`\clist_gset_eq:NN` 7816 `\cs_new_eq:NN \clist_gset_eq:NN \tl_gset_eq:NN`  
`\clist_gset_eq:cN` 7817 `\cs_new_eq:NN \clist_gset_eq:Nc \tl_gset_eq:Nc`  
`\clist_gset_eq:Nc` 7818 `\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc`  
`\clist_gset_eq:cc` 7819 `\cs_new_eq:NN \clist_gset_eq:cc \tl_gset_eq:cc`  
7820 `\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 108.)

`\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.

7821 `\cs_new_protected:Npn \clist_set_from_seq:NN`  
7822 { `\_clist_set_from_seq:NNNN \clist_clear:N \tl_set:Nx }`  
7823 `\cs_new_protected:Npn \clist_gset_from_seq:NN`  
7824 { `\_clist_set_from_seq:NNNN \clist_gclear:N \tl_gset:Nx }`  
7825 `\cs_new_protected:Npn \_clist_set_from_seq:NNNN #1#2#3#4`  
7826 {  
7827 `\seq_if_empty:NTF #4`  
7828 { `#1 #3` }  
7829 {  
7830 `#2 #3`  
7831 {  
7832 `\exp_after:wN \use_none:n \exp:w \exp_end_continue_f:w`  
7833 `\seq_map_function:NN #4 \_clist_set_from_seq:n`  
7834 }  
7835 }  
7836 }  
7837 `\cs_new:Npn \_clist_set_from_seq:n #1`  
7838 {  
7839 ,  
7840 `\_clist_if_wrap:nTF {#1}`  
7841 { `\exp_not:n { {#1} }`  }  
7842 { `\exp_not:n { {#1} }`  }  
7843 }  
7844 `\cs_generate_variant:Nn \clist_set_from_seq:NN { Nc }`  
7845 `\cs_generate_variant:Nn \clist_set_from_seq:NN { c , cc }`  
7846 `\cs_generate_variant:Nn \clist_gset_from_seq:NN { Nc }`  
7847 `\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 108.)

`\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.

```
7848 \cs_new_protected:Npn \clist_concat:NNN
7849   { \__clist_concat:NNNN \tl_set:Nx }
7850 \cs_new_protected:Npn \clist_gconcat:NNN
7851   { \__clist_concat:NNNN \tl_gset:Nx }
7852 \cs_new_protected:Npn \__clist_concat:NNNN #1#2#3#4
7853   {
7854     #1 #2
7855   {
7856     \exp_not:o #3
7857     \clist_if_empty:NF #3 { \clist_if_empty:NF #4 { , } }
7858     \exp_not:o #4
7859   }
7860 }
7861 \cs_generate_variant:Nn \clist_concat:NNN { ccc }
7862 \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 108.)

`\clist_if_exist_p:N` Copies of the `cs` functions defined in l3basics.

```
7863 \prg_new_eq_conditional:Nnn \clist_if_exist:N \cs_if_exist:N
7864   { TF , T , F , p }
7865 \prg_new_eq_conditional:Nnn \clist_if_exist:c \cs_if_exist:c
7866   { TF , T , F , p }
```

(End definition for `\clist_if_exist:NTF`. This function is documented on page 108.)

### 14.3 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_set:NW`  
`\clist_lpat_gset:NW`  
`\clist_lpat_gset:Nx`  
`\clist_lpat_gset:Nn`  
`\clist_lpat_gset:cW`  
`\clist_lpat_gset:cN`  
`\clist_lpat_gset:co`  
`\clist_lpat_gset:cx`  
`\clist_put_left:Nn`  
`\clist_gput_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 109.)

Everything is based on concatenation after storing in `\l__clist_internal_clist`. This avoids having to worry here about space-trimming and so on.

```
7873 \cs_new_protected:Npn \clist_put_left:Nn
7874   { \__clist_put_left:NNNn \clist_concat:NNN \clist_set:Nn }
7875 \cs_new_protected:Npn \clist_gput_left:Nn
7876   { \__clist_put_left:NNNn \clist_gconcat:NNN \clist_set:Nn }
7877 \cs_new_protected:Npn \__clist_put_left:NNNn #1#2#3#4
7878   {
7879     #2 \l__clist_internal_clist {#4}
7880     #1 #3 \l__clist_internal_clist #3
7881 }
```

```

7882 \cs_generate_variant:Nn \clist_put_left:Nn { NV , No , Nx }
7883 \cs_generate_variant:Nn \clist_put_left:Nn { c , cV , co , cx }
7884 \cs_generate_variant:Nn \clist_gput_left:Nn { NV , No , Nx }
7885 \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 109.)

```

\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

```

```

7886 \cs_new_protected:Npn \clist_put_right:Nn
7887 { \_clist_put_right:NNNn \clist_concat:NNN \clist_set:Nn }
7888 \cs_new_protected:Npn \clist_gput_right:Nn
7889 { \_clist_put_right:NNNn \clist_gconcat:NNN \clist_set:Nn }
7890 \cs_new_protected:Npn \_clist_put_right:NNNn #1#2#3#4
7891 {
7892 #2 \l__clist_internal_clist {#4}
7893 #1 #3 #3 \l__clist_internal_clist
7894 }
7895 \cs_generate_variant:Nn \clist_put_right:Nn { NV , No , Nx }
7896 \cs_generate_variant:Nn \clist_put_right:Nn { c , cV , co , cx }
7897 \cs_generate_variant:Nn \clist_gput_right:Nn { NV , No , Nx }
7898 \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 109.)

## 14.4 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.)

```

7899 \cs_new_protected:Npn \clist_get:NN #1#2
7900 {
7901 \if_meaning:w #1 \c_empty_clist
7902 \tl_set:Nn #2 { \q_no_value }
7903 \else:
7904 \exp_after:wN \__clist_get:wN #1 , \q_stop #2
7905 \fi:
7906 }
7907 \cs_new_protected:Npn \__clist_get:wN #1 , #2 \q_stop #3
7908 { \tl_set:Nn #3 {#1} }
7909 \cs_generate_variant:Nn \clist_get:NN { c }

```

(End definition for `\clist_get:NN` and `\__clist_get:wN`. This function is documented on page 114.)

```

\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 `\q_mark`, unless the original clist contained exactly one item: then the argument is just `\q_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.

```

7910 \cs_new_protected:Npn \clist_pop:NN
7911 { \__clist_pop:NNN \tl_set:Nx }
7912 \cs_new_protected:Npn \clist_gpop:NN

```

```

7913   { \__clist_pop:NNN \tl_gset:Nx }
7914   \cs_new_protected:Npn \__clist_pop:NNN #1#2#3
7915   {
7916     \if_meaning:w #2 \c_empty_clist
7917       \tl_set:Nn #3 { \q_no_value }
7918     \else:
7919       \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
7920     \fi:
7921   }
7922   \cs_new_protected:Npn \__clist_pop:wwNNN #1 , #2 \q_stop #3#4#5
7923   {
7924     \tl_set:Nn #5 {#1}
7925     #3 #4
7926     {
7927       \__clist_pop:wN \prg_do_nothing:
7928         #2 \exp_not:o
7929           , \q_mark \use_none:n
7930             \q_stop
7931       }
7932   }
7933   \cs_new:Npn \__clist_pop:wN #1 , \q_mark #2 #3 \q_stop { #2 {#1} }
7934   \cs_generate_variant:Nn \clist_pop:NN { c }
7935   \cs_generate_variant:Nn \clist_gpop:NN { c }

```

(End definition for `\clist_pop:NN` and others. These functions are documented on page 114.)

`\clist_get:NNTF` The same, as branching code: very similar to the above.

```

\clist_get:cNTF
7936 \prg_new_protected_conditional:Npnn \clist_get:NN #1#2 { T , F , TF }
\clist_pop:NNTF
7937   {
7938     \if_meaning:w #1 \c_empty_clist
7939       \prg_return_false:
7940     \else:
7941       \exp_after:wN \__clist_get:wN #1 , \q_stop #2
7942         \prg_return_true:
7943     \fi:
7944   }
7945 \prg_generate_conditional_variant:Nnn \clist_get:NN { c } { T , F , TF }
7946 \prg_new_protected_conditional:Npnn \clist_pop:NN #1#2 { T , F , TF }
7947   { \__clist_pop_TF:NNN \tl_set:Nx #1 #2 }
7948 \prg_new_protected_conditional:Npnn \clist_gpop:NN #1#2 { T , F , TF }
7949   { \__clist_pop_TF:NNN \tl_gset:Nx #1 #2 }
7950 \cs_new_protected:Npn \__clist_pop_TF:NNN #1#2#3
7951   {
7952     \if_meaning:w #2 \c_empty_clist
7953       \prg_return_false:
7954     \else:
7955       \exp_after:wN \__clist_pop:wwNNN #2 , \q_mark \q_stop #1#2#3
7956         \prg_return_true:
7957     \fi:
7958   }
7959 \prg_generate_conditional_variant:Nnn \clist_pop:NN { c } { T , F , TF }
7960 \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 114.)

```

\clist_push:Nn Pushing to a comma list is the same as adding on the left.
\clist_push:NV
\clist_push:No
\clist_push:Nx
\clist_push:cn
\clist_push:cV
\clist_push:co
\clist_push:cx
\clist_gpush:Nn
\clist_gpush:NV
\clist_gpush:No
\clist_gpush:Nx
\clist_gpush:cn
\clist_gpush:cV
\clist_gpush:co
\clist_gpush:cx

```

(End definition for `\clist_push:Nn` and `\clist_gpush:Nn`. These functions are documented on page 114.)

## 14.5 Modifying comma lists

`\l__clist_internal_remove_clist`  
`\l__clist_internal_remove_seq`

```

7977 \clist_new:N \l__clist_internal_remove_clist
7978 \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`  
`\clist_remove_duplicates:c`  
`\clist_gremove_duplicates:N`  
`\clist_gremove_duplicates:c`

```

7979 \cs_new_protected:Npn \clist_remove_duplicates:N
7980   { \__clist_remove_duplicates>NN \clist_set_eq:NN }
7981 \cs_new_protected:Npn \clist_gremove_duplicates:N
7982   { \__clist_remove_duplicates>NN \clist_gset_eq:NN }
7983 \cs_new_protected:Npn \__clist_remove_duplicates>NN #1#2
7984   {
7985     \clist_clear:N \l__clist_internal_remove_clist
7986     \clist_map_inline:Nn #2
7987     {
7988       \clist_if_in:NnF \l__clist_internal_remove_clist {##1}
7989         { \clist_put_right:Nn \l__clist_internal_remove_clist {##1} }
7990     }
7991     #1 #2 \l__clist_internal_remove_clist
7992   }
7993 \cs_generate_variant:Nn \clist_remove_duplicates:N { c }
7994 \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 110.)

`\clist_remove_all:Nn`  
`\clist_remove_all:cn`  
`\clist_gremove_all:Nn`  
`\clist_gremove_all:cn`  
`\__clist_remove_all>NNNn`  
`\__clist_remove_all:w`  
`\__clist_remove_all:`

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 `\q_mark` delimiter used is the one inserted by `\__clist_tmp:w`, and `\use_none_delimit_by_q_stop:w` is deleted. At the end, the final *<item>* is grabbed, and the argument of `\__clist_tmp:w` contains `\q_mark`: in that case, `\__clist_remove_all:w` removes the second `\q_mark` (inserted by `\__clist_tmp:w`), and lets `\use_none_delimit_by_q_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.

```

7995 \cs_new_protected:Npn \clist_remove_all:Nn
7996   { \__clist_remove_all:NNNn \clist_set_from_seq:NN \tl_set:Nx }
7997 \cs_new_protected:Npn \clist_gremove_all:Nn
7998   { \__clist_remove_all:NNNn \clist_gset_from_seq:NN \tl_gset:Nx }
7999 \cs_new_protected:Npn \__clist_remove_all:NNNn #1#2#3#4
8000   {
8001     \__clist_if_wrap:nTF {#4}
8002     {
8003       \seq_set_from_clist:NN \l__clist_internal_remove_seq #3
8004       \seq_remove_all:Nn \l__clist_internal_remove_seq {#4}
8005       #1 #3 \l__clist_internal_remove_seq
8006     }
8007     {
8008       \cs_set:Npn \__clist_tmp:w ##1 , #4 ,
8009       {
8010         ##1
8011         , \q_mark , \use_none_delimit_by_q_stop:w ,
8012         \__clist_remove_all:
8013       }
8014     #2 #3
8015     {
8016       \exp_after:wN \__clist_remove_all:
8017       #3 , \q_mark , #4 , \q_stop
8018     }
8019     \clist_if_empty:NF #3
8020     {
8021       #2 #3
8022       {
8023         \exp_args:No \exp_not:o
8024         { \exp_after:wN \use_none:n #3 }
8025       }
8026     }
8027   }
8028 }
8029 \cs_new:Npn \__clist_remove_all:
8030   { \exp_after:wN \__clist_remove_all:w \__clist_tmp:w , }
8031 \cs_new:Npn \__clist_remove_all:w #1 , \q_mark , #2 , { \exp_not:n {#1} }
```

```

8032 \cs_generate_variant:Nn \clist_remove_all:Nn { c }
8033 \cs_generate_variant:Nn \clist_gremove_all:Nn { c }

```

(End definition for `\clist_remove_all:Nn` and others. These functions are documented on page 110.)

**\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.

```

8034 \cs_new_protected:Npn \clist_reverse:N #1
8035   { \tl_set:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
8036 \cs_new_protected:Npn \clist_greverse:N #1
8037   { \tl_gset:Nx #1 { \exp_args:No \clist_reverse:n {#1} } }
8038 \cs_generate_variant:Nn \clist_reverse:N { c }
8039 \cs_generate_variant:Nn \clist_greverse:N { c }

```

(End definition for `\clist_reverse:N` and `\clist_greverse:N`. These functions are documented on page 110.)

**\clist\_reverse:n** The reversed token list is built one item at a time, and stored between `\q_stop` and `\q_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 `\q_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 “`\q_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 `\q_stop`), and leaves its argument #1 within `\exp_not:o`. There is also a need to remove a leading comma, hence `\exp_not:o` and `\use_none:n`.

```

8040 \cs_new:Npn \clist_reverse:n #1
8041   {
8042     \__clist_reverse:wwNww ? #1 ,
8043     \q_mark \__clist_reverse:wwNww ! ,
8044     \q_mark \__clist_reverse_end:ww
8045     \q_stop ? \q_mark
8046   }
8047 \cs_new:Npn \__clist_reverse:wwNww
8048   #1 , #2 \q_mark #3 #4 \q_stop ? #5 \q_mark
8049   { #3 ? #2 \q_mark #3 #4 \q_stop #1 , #5 \q_mark }
8050 \cs_new:Npn \__clist_reverse_end:ww #1 ! #2 , \q_mark
8051   { \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 110.)

**\clist\_sort:Nn** Implemented in l3sort.

**\clist\_sort:cn**

**\clist\_gsort:Nn** (End definition for `\clist_sort:Nn` and `\clist_gsort:Nn`. These functions are documented on page 110.)

**\clist\_gsort:cn**

## 14.6 Comma list conditionals

```
\clist_if_empty_p:N Simple copies from the token list variable material.
\clist_if_empty_p:c
8052 \prg_new_eq_conditional:NNN \clist_if_empty:N \tl_if_empty:N
8053 { p , T , F , TF }
8054 \prg_new_eq_conditional:NNN \clist_if_empty:c \tl_if_empty:c
8055 { p , T , F , TF }
```

(End definition for `\clist_if_empty:NTF`. This function is documented on page 110.)

```
\clist_if_empty_p:n As usual, we insert a token (here ?) before grabbing any argument: this avoids losing
\clist_if_empty:nTF braces. The argument of \tl_if_empty:oTF is empty if #1 is ? followed by blank spaces
\_clist_if_empty_n:w (besides, this particular variant of the emptiness test is optimized). If the item of the
\_clist_if_empty_n:wNw 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 \q_mark \prg_return_false: item.
```

```
8056 \prg_new_conditional:Npnn \clist_if_empty:n #1 { p , T , F , TF }
8057 {
8058     \_clist_if_empty_n:w ? #1
8059     , \q_mark \prg_return_false:
8060     , \q_mark \prg_return_true:
8061     \q_stop
8062 }
8063 \cs_new:Npn \_clist_if_empty_n:w #1 ,
8064 {
8065     \tl_if_empty:oTF { \use_none:nn #1 ? }
8066     { \_clist_if_empty_n:w ? }
8067     { \_clist_if_empty_n:wNw }
8068 }
8069 \cs_new:Npn \_clist_if_empty_n:wNw #1 \q_mark #2#3 \q_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 111.)

```
\clist_if_in:NnTF For “safe” items, we simply surround the comma list, and the item, with commas, then
\clist_if_in:NvTF use the same code as for \tl_if_in:Nn. For “unsafe” items we follow the same route as
\clist_if_in:NtTF \seq_if_in:Nn, mapping through the list a comparison function. If found, return true
\clist_if_in:CnTF and remove \prg_return_false:.
\clist_if_in:cVTF
\clist_if_in:cOTF
\clist_if_in:nNTF
\clist_if_in:nVTF
\clist_if_in:notTF
\__clist_if_in_return:nN For “safe” items, we simply surround the comma list, and the item, with commas, then
use the same code as for \tl_if_in:Nn. For “unsafe” items we follow the same route as
\seq_if_in:Nn, mapping through the list a comparison function. If found, return true
and remove \prg_return_false:.
8070 \prg_new_protected_conditional:Npnn \clist_if_in:Nn #1#2 { T , F , TF }
8071 {
8072     \exp_args:No \_clist_if_in_return:nN #1 {#2} #1
8073 }
8074 \prg_new_protected_conditional:Npnn \clist_if_in:nN #1#2 { T , F , TF }
8075 {
8076     \clist_set:Nn \l__clist_internal_clist {#1}
8077     \exp_args:No \_clist_if_in_return:nN \l__clist_internal_clist {#2}
8078     \l__clist_internal_clist
8079 }
8080 \cs_new_protected:Npn \_clist_if_in_return:nN #1#2#3
8081 {
8082     \_clist_if_wrap:nTF {#2}
8083 {
8084     \cs_set:Npx \_clist_tmp:w ##1
8085 }
```

```

8086     \exp_not:N \tl_if_eq:nnT {##1}
8087     \exp_not:n
8088     {
8089         {#2}
8090         { \clist_map_break:n { \prg_return_true: \use_none:n } }
8091     }
8092 }
8093 \clist_map_function:NN #3 \__clist_tmp:w
8094 \prg_return_false:
8095 }
8096 {
8097     \cs_set:Npn \__clist_tmp:w ##1 ,#2, { }
8098     \tl_if_empty:oTF
8099     { \__clist_tmp:w ,#1, {} {} ,#2, }
8100     { \prg_return_false: } { \prg_return_true: }
8101 }
8102 }
8103 \prg_generate_conditional_variant:Nnn \clist_if_in:Nn
8104 { NV , No , c , cV , co } { T , F , TF }
8105 \prg_generate_conditional_variant:Nnn \clist_if_in:nn
8106 { 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 111.)

## 14.7 Mapping to comma lists

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 one comma-delimited item at a time. The end is marked by `\q_recursion_tail`. The auxiliary function `\__clist_map_function:Nw` is also used in `\clist_map_inline:Nn`.

```

8107 \cs_new:Npn \clist_map_function:NN #1#2
8108 {
8109     \clist_if_empty:NF #1
8110     {
8111         \exp_last_unbraced:NNo \__clist_map_function:Nw #2 #1
8112         , \q_recursion_tail ,
8113         \prg_break_point:Nn \clist_map_break: { }
8114     }
8115 }
8116 \cs_new:Npn \__clist_map_function:Nw #1#2 ,
8117 {
8118     \quark_if_recursion_tail_break:nN {#2} \clist_map_break:
8119     #1 {#2}
8120     \__clist_map_function:Nw #1
8121 }
8122 \cs_generate_variant:Nn \clist_map_function:NN { c }

```

(End definition for `\clist_map_function:NN` and `\__clist_map_function:Nw`. This function is documented on page 111.)

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_unbrace:Nw`

`\__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:Nw`.

```

8123 \cs_new:Npn \clist_map_function:nN #1#2
8124 {
8125   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #2
8126   \exp:w \__clist_trim_next:w \prg_do_nothing: #1 , \q_recursion_tail ,
8127   \prg_break_point:Nn \clist_map_break: { }
8128 }
8129 \cs_new:Npn \__clist_map_function_n:Nn #1 #2
8130 {
8131   \quark_if_recursion_tail_break:nN {#2} \clist_map_break:
8132   \__clist_map_unbrace:Nw #1 #2,
8133   \exp_after:wN \__clist_map_function_n:Nn \exp_after:wN #1
8134   \exp:w \__clist_trim_next:w \prg_do_nothing:
8135 }
8136 \cs_new:Npn \__clist_map_unbrace:Nw #1 #2, { #1 {#2} }
```

(End definition for `\clist_map_function:nN`, `\__clist_map_function_n:Nn`, and `\__clist_map_unbrace:Nw`. This function is documented on page 111.)

`\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 TeX’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.

```

8137 \cs_new_protected:Npn \clist_map_inline:Nn #1#2
8138 {
8139   \clist_if_empty:NF #1
8140   {
8141     \int_gincr:N \g_kernel_prg_map_int
8142     \cs_gset_protected:cpn
8143     { \__clist_map_ \int_use:N \g_kernel_prg_map_int :w } ##1 {#2}
8144     \exp_last_unbraced:Nco \__clist_map_function:Nw
8145     { \__clist_map_ \int_use:N \g_kernel_prg_map_int :w }
8146     #1 , \q_recursion_tail ,
8147     \prg_break_point:Nn \clist_map_break:
8148     { \int_gdecr:N \g_kernel_prg_map_int }
8149   }
8150 }
8151 \cs_new_protected:Npn \clist_map_inline:nn #1
8152 {
8153   \clist_set:Nn \l__clist_internal_clist {#1}
8154   \clist_map_inline:Nn \l__clist_internal_clist
8155 }
8156 \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 111.)

`\clist_map_variable>NNn`  
`\clist_map_variable:cNn`  
`\clist_map_variable:nNn`  
`\__clist_map_variable:Nnw`

As for other comma-list mappings, filter out the case of an empty list. Same approach as `\clist_map_function:Nn`, additionally we store each item in the given variable. As for inline mappings, space trimming for the `n` variant is done by storing the comma

list in a variable. The strange `\use:n` avoids unlikely problems when #2 would contain `\q_recursion_stop`.

```

8157 \cs_new_protected:Npn \clist_map_variable:NNn #1#2#3
8158   {
8159     \clist_if_empty:NF #1
8160     {
8161       \exp_args:Nno \use:nn
8162         { \__clist_map_variable:Nnw #2 {#3} }
8163         #1
8164         , \q_recursion_tail , \q_recursion_stop
8165         \prg_break_point:Nn \clist_map_break: { }
8166     }
8167   }
8168 \cs_new_protected:Npn \clist_map_variable:nNn #1
8169   {
8170     \clist_set:Nn \l__clist_internal_clist {#1}
8171     \clist_map_variable:NNn \l__clist_internal_clist
8172   }
8173 \cs_new_protected:Npn \__clist_map_variable:Nnw #1#2#3,
8174   {
8175     \tl_set:Nn #1 {#3}
8176     \quark_if_recursion_tail_stop:N #1
8177     \use:n {#2}
8178     \__clist_map_variable:Nnw #1 {#2}
8179   }
8180 \cs_generate_variant:Nn \clist_map_variable:NNn { c }
```

(End definition for `\clist_map_variable:NNn`, `\clist_map_variable:nNn`, and `\__clist_map_variable:Nnw`. These functions are documented on page [112](#).)

**\clist\_map\_break:** The break statements use the general `\prg_map_break:Nn` mechanism.

**\clist\_map\_break:n**

```

8181 \cs_new:Npn \clist_map_break:
8182   { \prg_map_break:Nn \clist_map_break: { } }
8183 \cs_new:Npn \clist_map_break:n
8184   { \prg_map_break:Nn \clist_map_break: }
```

(End definition for `\clist_map_break:` and `\clist_map_break:n`. These functions are documented on page [112](#).)

**\clist\_count:N** **\clist\_count:c** **\clist\_count:n** **\\_\_clist\_count:n** **\\_\_clist\_count:w** 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).

```

8185 \cs_new:Npn \clist_count:N #1
8186   {
8187     \int_eval:n
8188     {
8189       0
8190       \clist_map_function:NN #1 \__clist_count:n
8191     }
8192   }
8193 \cs_generate_variant:Nn \clist_count:N { c }
8194 \cs_new:Npx \clist_count:n #1
```

```

8195   {
8196     \exp_not:N \int_eval:n
8197   {
8198     0
8199     \exp_not:N \__clist_count:w \c_space_tl
8200     #1 \exp_not:n { , \q_recursion_tail , \q_recursion_stop }
8201   }
8202 }
8203 \cs_new:Npn \__clist_count:n #1 { + 1 }
8204 \cs_new:Npx \__clist_count:w #1 ,
8205 {
8206   \exp_not:n { \exp_args:Nf \quark_if_recursion_tail_stop:n } {#1}
8207   \exp_not:N \tl_if_blank:nF {#1} { + 1 }
8208   \exp_not:N \__clist_count:w \c_space_tl
8209 }

```

(End definition for `\clist_count:N` and others. These functions are documented on page 112.)

## 14.8 Using comma lists

```

\clist_use:Nnnn
\clist_use:cnnn
\__clist_use:wwn
\__clist_use:nwwwnwn
\__clist_use:nwwn
\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 `\q_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 `\q_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, `\q_mark` is taken as a third item, and now the second `\q_mark` serves as a delimiter to `use_ii`, switching to the other *<continuation>*, `use_iii`, which uses the *<separator between final two>*.

```

8210 \cs_new:Npn \clist_use:Nnnn #1#2#3#4
8211 {
8212   \clist_if_exist:NTF #1
8213   {
8214     \int_case:nnF { \clist_count:N #1 }
8215     {
8216       { 0 } { }
8217       { 1 } { \exp_after:wN \__clist_use:wwn #1 , { } }
8218       { 2 } { \exp_after:wN \__clist_use:wwn #1 , {#2} }
8219     }
8220   {
8221     \exp_after:wN \__clist_use:nwwwnwn
8222     \exp_after:wN { \exp_after:wN } #1 ,
8223     \q_mark , { \__clist_use:nwwwnwn {#3} }
8224     \q_mark , { \__clist_use:nwwn {#4} }
8225     \q_stop { }
8226   }
8227 }

```

```

8228     {
8229         \__kernel_msg_expandable_error:nnn
8230         { kernel } { bad-variable } {#1}
8231     }
8232 }
8233 \cs_generate_variant:Nn \clist_use:Nnnn { c }
8234 \cs_new:Npn \__clist_use:wwn #1 , #2 , #3 { \exp_not:n { #1 #3 #2 } }
8235 \cs_new:Npn \__clist_use:nwwwnw
8236     #1#2 , #3 , #4 , #5 \q_mark , #6#7 \q_stop #8
8237     { #6 {#3} , {#4} , #5 \q_mark , {#6} #7 \q_stop { #8 #1 #2 } }
8238 \cs_new:Npn \__clist_use:nwwn #1#2 , #3 \q_stop #4
8239     { \exp_not:n { #4 #1 #2 } }
8240 \cs_new:Npn \clist_use:Nn #1#2
8241     { \clist_use:Nnn #1 {#2} {#2} {#2} }
8242 \cs_generate_variant:Nn \clist_use:Nn { c }

```

(End definition for `\clist_use:Nnnn` and others. These functions are documented on page 113.)

## 14.9 Using a single item

`\clist_item:Nn`

`\clist_item:cn`

`\__clist_item:nnN`

`\__clist_item:ffoN`

`\__clist_item:ffnN`

`\__clist_item_N_loop:nw`

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.

```

8243 \cs_new:Npn \clist_item:Nn #1#2
8244     {
8245         \__clist_item:ffoN
8246         { \clist_count:N #1 }
8247         { \int_eval:n {#2} }
8248         #1
8249         \__clist_item_N_loop:nw
8250     }
8251 \cs_new:Npn \__clist_item:nnN #1#2#3#4
8252     {
8253         \int_compare:nNnTF {#2} < 0
8254         {
8255             \int_compare:nNnTF {#2} < { - #1 }
8256             { \use_none_delimit_by_q_stop:w }
8257             { \exp_args:Nf #4 { \int_eval:n { #2 + 1 + #1 } } }
8258         }
8259         {
8260             \int_compare:nNnTF {#2} > {#1}
8261             { \use_none_delimit_by_q_stop:w }
8262             { #4 {#2} }
8263         }
8264         { } , #3 , \q_stop
8265     }
8266 \cs_generate_variant:Nn \__clist_item:nnN { ffo, ff }
8267 \cs_new:Npn \__clist_item_N_loop:nw #1 #2,
8268     {
8269         \int_compare:nNnTF {#1} = 0
8270             { \use_i_delimit_by_q_stop:nw { \exp_not:n {#2} } }
8271             { \exp_args:Nf \__clist_item_N_loop:nw { \int_eval:n { #1 - 1 } } }

```

```

8272   }
8273 \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 115.)

```

\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.

```

8274 \cs_new:Npn \clist_item:nn #1#2
8275   {
8276     \__clist_item:ffnN
8277       { \clist_count:n {#1} }
8278       { \int_eval:n {#2} }
8279       {#1}
8280     \__clist_item_n:nw
8281   }
8282 \cs_new:Npn \__clist_item_n:nw #1
8283   { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
8284 \cs_new:Npn \__clist_item_n_loop:nw #1 #2,
8285   {
8286     \exp_args:No \tl_if_blank:nTF {#2}
8287       { \__clist_item_n_loop:nw {#1} \prg_do_nothing: }
8288       {
8289         \int_compare:nNnTF {#1} = 0
8290           { \exp_args:No \__clist_item_n_end:n {#2} }
8291           {
8292             \exp_args:Nf \__clist_item_n_loop:nw
8293               { \int_eval:n {#1 - 1} }
8294             \prg_do_nothing:
8295           }
8296         }
8297       }
8298 \cs_new:Npn \__clist_item_n_end:n #1 #2 \q_stop
8299   { \tl_trim_spaces_apply:nN {#1} \__clist_item_n_strip:n }
8300 \cs_new:Npn \__clist_item_n_strip:n #1 { \__clist_item_n_strip:w #1 , }
8301 \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 115.)

```

\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.

```

8302 \cs_new:Npn \clist_rand_item:n #1
8303   { \exp_args:Nf \__clist_rand_item:nn { \clist_count:n {#1} } {#1} }
8304 \cs_new:Npn \__clist_rand_item:nn #1#2
8305   {
8306     \int_compare:nNnF {#1} = 0
8307       { \clist_item:nn {#2} { \int_rand:nn { 1 } {#1} } }
8308   }
8309 \cs_new:Npn \clist_rand_item:N #1
8310   {

```

```

8311     \clist_if_empty:NF #1
8312         { \clist_item:Nn #1 { \int_rand:nn { 1 } { \clist_count:N #1 } } }
8313     }
8314 \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 115.)

## 14.10 Viewing comma lists

`\clist_show:N` Apply the general `\_kernel_chk_defined:NT` and `\msg_show:nnnnnn`.

```

\clist_show:c
  \clist_log:N
  \clist_log:c
\__clist_show>NN
8315 \cs_new_protected:Npn \clist_show:N { \__clist_show>NN \msg_show:nnxxxx }
8316 \cs_generate_variant:Nn \clist_show:N { c }
8317 \cs_new_protected:Npn \clist_log:N { \__clist_show>NN \msg_log:nnxxxx }
8318 \cs_generate_variant:Nn \clist_log:N { c }
8319 \cs_new_protected:Npn \__clist_show>NN #1#2
8320 {
8321     \__kernel_chk_defined:NT #2
8322     {
8323         #1 { LaTeX/kernel } { show-clist }
8324         { \token_to_str:N #2 }
8325         { \clist_map_function:NN #2 \msg_show_item:n }
8326         { } { }
8327     }
8328 }

```

(End definition for `\clist_show:N`, `\clist_log:N`, and `\_clist_show>NN`. These functions are documented on page 115.)

`\clist_show:n` A variant of the above: no existence check, empty first argument for the message.

```

\clist_log:n
\__clist_show:Nn
8329 \cs_new_protected:Npn \clist_show:n { \__clist_show:Nn \msg_show:nnxxxx }
8330 \cs_new_protected:Npn \clist_log:n { \__clist_show:Nn \msg_log:nnxxxx }
8331 \cs_new_protected:Npn \__clist_show:Nn #1#2
8332 {
8333     #1 { LaTeX/kernel } { show-clist }
8334     { } { \clist_map_function:nN {#2} \msg_show_item:n } { } { }
8335 }

```

(End definition for `\clist_show:n`, `\clist_log:n`, and `\_clist_show:Nn`. These functions are documented on page 115.)

## 14.11 Scratch comma lists

`\l_tmpa_clist` Temporary comma list variables.

```

\l_tmpb_clist
\g_tmpa_clist
\g_tmpb_clist
8336 \clist_new:N \l_tmpa_clist
8337 \clist_new:N \l_tmpb_clist
8338 \clist_new:N \g_tmpa_clist
8339 \clist_new:N \g_tmpb_clist

```

(End definition for `\l_tmpa_clist` and others. These variables are documented on page 116.)

8340 ⟨/initex | package⟩

## 15 I3token implementation

```
8341  {*initex | package}
8342  <@=char>
```

### 15.1 Manipulating and interrogating character tokens

\char\_set\_catcode:nn

\char\_value\_catcode:n

\char\_show\_value\_catcode:n

```
8343  \cs_new_protected:Npn \char_set_catcode:nn #1#2
8344    { \tex_catcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
8345  \cs_new:Npn \char_value_catcode:n #1
8346    { \tex_the:D \tex_catcode:D \int_eval:n {#1} \exp_stop_f: }
8347  \cs_new_protected:Npn \char_show_value_catcode:n #1
8348    { \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 120.)

\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

```
8349  \cs_new_protected:Npn \char_set_catcode_escape:N #1
8350    { \char_set_catcode:nn { '#1 } { 0 } }
8351  \cs_new_protected:Npn \char_set_catcode_group_begin:N #1
8352    { \char_set_catcode:nn { '#1 } { 1 } }
8353  \cs_new_protected:Npn \char_set_catcode_group_end:N #1
8354    { \char_set_catcode:nn { '#1 } { 2 } }
8355  \cs_new_protected:Npn \char_set_catcode_math_toggle:N #1
8356    { \char_set_catcode:nn { '#1 } { 3 } }
8357  \cs_new_protected:Npn \char_set_catcode_alignment:N #1
8358    { \char_set_catcode:nn { '#1 } { 4 } }
8359  \cs_new_protected:Npn \char_set_catcode_end_line:N #1
8360    { \char_set_catcode:nn { '#1 } { 5 } }
8361  \cs_new_protected:Npn \char_set_catcode_parameter:N #1
8362    { \char_set_catcode:nn { '#1 } { 6 } }
8363  \cs_new_protected:Npn \char_set_catcode_math_superscript:N #1
8364    { \char_set_catcode:nn { '#1 } { 7 } }
8365  \cs_new_protected:Npn \char_set_catcode_math_subscript:N #1
8366    { \char_set_catcode:nn { '#1 } { 8 } }
8367  \cs_new_protected:Npn \char_set_catcode_ignore:N #1
8368    { \char_set_catcode:nn { '#1 } { 9 } }
8369  \cs_new_protected:Npn \char_set_catcode_space:N #1
8370    { \char_set_catcode:nn { '#1 } { 10 } }
8371  \cs_new_protected:Npn \char_set_catcode_letter:N #1
8372    { \char_set_catcode:nn { '#1 } { 11 } }
8373  \cs_new_protected:Npn \char_set_catcode_other:N #1
8374    { \char_set_catcode:nn { '#1 } { 12 } }
8375  \cs_new_protected:Npn \char_set_catcode_active:N #1
8376    { \char_set_catcode:nn { '#1 } { 13 } }
8377  \cs_new_protected:Npn \char_set_catcode_comment:N #1
8378    { \char_set_catcode:nn { '#1 } { 14 } }
8379  \cs_new_protected:Npn \char_set_catcode_invalid:N #1
8380    { \char_set_catcode:nn { '#1 } { 15 } }
```

(End definition for \char\_set\_catcode\_escape:N and others. These functions are documented on page 119.)

```

\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

8381  \cs_new_protected:Npn \char_set_catcode_escape:n #1
8382    { \char_set_catcode:nn {#1} { 0 } }
8383  \cs_new_protected:Npn \char_set_catcode_group_begin:n #1
8384    { \char_set_catcode:nn {#1} { 1 } }
8385  \cs_new_protected:Npn \char_set_catcode_group_end:n #1
8386    { \char_set_catcode:nn {#1} { 2 } }
8387  \cs_new_protected:Npn \char_set_catcode_math_toggle:n #1
8388    { \char_set_catcode:nn {#1} { 3 } }
8389  \cs_new_protected:Npn \char_set_catcode_alignment:n #1
8390    { \char_set_catcode:nn {#1} { 4 } }
8391  \cs_new_protected:Npn \char_set_catcode_end_line:n #1
8392    { \char_set_catcode:nn {#1} { 5 } }
8393  \cs_new_protected:Npn \char_set_catcode_parameter:n #1
8394    { \char_set_catcode:nn {#1} { 6 } }
8395  \cs_new_protected:Npn \char_set_catcode_math_superscript:n #1
8396    { \char_set_catcode:nn {#1} { 7 } }
8397  \cs_new_protected:Npn \char_set_catcode_math_subscript:n #1
8398    { \char_set_catcode:nn {#1} { 8 } }
8399  \cs_new_protected:Npn \char_set_catcode_ignore:n #1
8400    { \char_set_catcode:nn {#1} { 9 } }
8401  \cs_new_protected:Npn \char_set_catcode_space:n #1
8402    { \char_set_catcode:nn {#1} { 10 } }
8403  \cs_new_protected:Npn \char_set_catcode_letter:n #1
8404    { \char_set_catcode:nn {#1} { 11 } }
8405  \cs_new_protected:Npn \char_set_catcode_other:n #1
8406    { \char_set_catcode:nn {#1} { 12 } }
8407  \cs_new_protected:Npn \char_set_catcode_active:n #1
8408    { \char_set_catcode:nn {#1} { 13 } }
8409  \cs_new_protected:Npn \char_set_catcode_comment:n #1
8410    { \char_set_catcode:nn {#1} { 14 } }
8411  \cs_new_protected:Npn \char_set_catcode_invalid:n #1
8412    { \char_set_catcode:nn {#1} { 15 } }


```

(End definition for `\char_set_catcode_escape:n` and others. These functions are documented on page 119.)

```

\char_set_mathcode:nn
\char_value_mathcode:n
\char_show_value_mathcode:n
  \char_set_lccode:nn
  \char_value_lccode:n
\char_show_value_lccode:n
  \char_set_uccode:nn
  \char_value_uccode:n
\char_show_value_uccode:n
  \char_set_sfcode:nn
  \char_value_sfcode:n
\char_show_value_sfcode:n

8413  \cs_new_protected:Npn \char_set_mathcode:nn #1#2
8414    { \tex_mathcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
8415  \cs_new:Npn \char_value_mathcode:n #1
8416    { \tex_the:D \tex_mathcode:D \int_eval:n {#1} \exp_stop_f: }
8417  \cs_new_protected:Npn \char_show_value_mathcode:n #1
8418    { \exp_args:Nf \tl_show:n { \char_value_mathcode:n {#1} } }
8419  \cs_new_protected:Npn \char_set_lccode:nn #1#2
8420    { \tex_lccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
8421  \cs_new:Npn \char_value_lccode:n #1
8422    { \tex_the:D \tex_lccode:D \int_eval:n {#1} \exp_stop_f: }
8423  \cs_new_protected:Npn \char_show_value_lccode:n #1
8424    { \exp_args:Nf \tl_show:n { \char_value_lccode:n {#1} } }
8425  \cs_new_protected:Npn \char_set_uccode:nn #1#2
8426    { \tex_uccode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
8427  \cs_new:Npn \char_value_uccode:n #1
8428    { \tex_the:D \tex_uccode:D \int_eval:n {#1} \exp_stop_f: }


```

```

8429 \cs_new_protected:Npn \char_show_value_uccode:n #1
8430   { \exp_args:Nf \tl_show:n { \char_value_uccode:n {#1} } }
8431 \cs_new_protected:Npn \char_set_sfcode:nn #1#2
8432   { \tex_sfcode:D \int_eval:n {#1} = \int_eval:n {#2} \exp_stop_f: }
8433 \cs_new:Npn \char_value_sfcode:n #1
8434   { \tex_the:D \tex_sfcode:D \int_eval:n {#1} \exp_stop_f: }
8435 \cs_new_protected:Npn \char_show_value_sfcode:n #1
8436   { \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 121.)

`\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`).

```

8437 \seq_new:N \l_char_special_seq
8438 \seq_set_split:Nnn \l_char_special_seq { }
8439   { \ " \# \$ \% & \\ ^ _ { } ~ }
8440 \seq_new:N \l_char_active_seq
8441 \seq_set_split:Nnn \l_char_active_seq { }
8442   { " \$ & ^ _ ~ }

```

(End definition for `\l_char_active_seq` and `\l_char_special_seq`. These variables are documented on page 121.)

## 15.2 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_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.

```

8443 \group_begin:
8444   \char_set_catcode_active:N \^@ 
8445 \cs_set_protected:Npn \__char_tmp:nN #1#2
8446   {
8447     \cs_new_protected:cpn { #1 :nN } ##1
8448     {
8449       \group_begin:
8450         \char_set_lccode:nn { '\^@ } { ##1 }
8451         \tex_lowercase:D { \group_end: #2 ^@ }
8452     }
8453     \cs_new_protected:cpx { #1 :NN } ##1
8454     { \exp_not:c { #1 : nN } { ##1 } }
8455   }
8456   \__char_tmp:nN { char_set_active_eq } \cs_set_eq:NN
8457   \__char_tmp:nN { char_gset_active_eq } \cs_gset_eq:NN
8458 \group_end:
8459 \cs_generate_variant:Nn \char_set_active_eq:NN { Nc }
8460 \cs_generate_variant:Nn \char_gset_active_eq:NN { Nc }
8461 \cs_generate_variant:Nn \char_set_active_eq:nN { nc }
8462 \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 117.)

`\__char_int_to_roman:w`

For efficiency in 8-bit engines, we use the faster primitive approach to making roman numerals.

```
8463 \cs_new_eq:NN \__char_int_to_roman:w \tex_roman numeral:D
```

(End definition for `\_char_int_to_roman:w`.)

```
\char_generate:nn
\__char_generate_aux:nn
\__char_generate_aux:nnw
\__char_generate_auxii:nnw
  \l__char_tmp_tl
\__char_generate_invalid_catcode:
```

The aim here is to generate characters of (broadly) arbitrary category code. Where possible, that is done using engine support (X<sub>E</sub>T<sub>E</sub>X, LuaT<sub>E</sub>X). 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.

```
8464 \cs_new:Npn \char_generate:nn #1#2
8465   {
8466     \exp:w \exp_after:wN \__char_generate_aux:w
8467       \int_value:w \int_eval:n {#1} \exp_after:wN ;
8468       \int_value:w \int_eval:n {#2} ;
8469   }
```

Before doing any actual conversion, first some special case filtering. Spaces are out here as LuaT<sub>E</sub>X emulation only makes normal (charcode 32 spaces). However, `^@` 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.

```
8470 \cs_new:Npn \__char_generate_aux:w #1 ; #2 ;
8471   {
8472     \if_int_compare:w #2 = 10 \exp_stop_f:
8473       \if_int_compare:w #1 = 0 \exp_stop_f:
8474         \__kernel_msg_expandable_error:nn { kernel } { char-null-space }
8475       \else:
8476         \__kernel_msg_expandable_error:nn { kernel } { char-space }
8477       \fi:
8478     \else:
8479       \if_int_odd:w 0
8480         \if_int_compare:w #2 < 1 \exp_stop_f: 1 \fi:
8481         \if_int_compare:w #2 = 5 \exp_stop_f: 1 \fi:
8482         \if_int_compare:w #2 = 9 \exp_stop_f: 1 \fi:
8483         \if_int_compare:w #2 > 13 \exp_stop_f: 1 \fi: \exp_stop_f:
8484         \__kernel_msg_expandable_error:nn { kernel }
8485           { char-invalid-catcode }
8486     \else:
8487       \if_int_odd:w 0
8488         \if_int_compare:w #1 < 0 \exp_stop_f: 1 \fi:
8489         \if_int_compare:w #1 > \c_max_char_int 1 \fi: \exp_stop_f:
8490         \__kernel_msg_expandable_error:nn { kernel }
8491           { char-out-of-range }
8492       \else:
8493         \__char_generate_aux:nnw {#1} {#2}
8494       \fi:
8495     \fi:
8496   \fi:
8497   \exp_end:
8498 }
8499 \tl_new:N \l__char_tmp_tl
```

Engine-dependent definitions are now needed for the implementation. For LuaT<sub>E</sub>X 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.

```

8500 \group_begin:
8501 (*package)
8502   \char_set_catcode_active:N `^`L
8503   \cs_set:Npn `^`L { }
8504 
```

`</package>`

```

8505   \char_set_catcode_other:n { 0 }
8506   \if_int_odd:w 0
8507     \sys_if_engine_luatex:T { 1 }
8508     \sys_if_engine_xetex:T { 1 } \exp_stop_f:
8509     \sys_if_engine_luatex:TF
8510     {
8511       \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
8512       {
8513         #3
8514         \exp_after:wN \exp_after:wN \exp_after:wN \exp_end:
8515         \lua_now:e { 13kernel.charcat(#1, #2) }
8516       }
8517     }
8518   {
8519     \cs_new:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
8520     {
8521       #3
8522       \exp_after:wN \exp_end:
8523       \tex_Ucharcat:D #1 \exp_stop_f: #2 \exp_stop_f:
8524     }
8525     \cs_if_exist:NF \tex_expanded:D
8526     {
8527       \cs_new_eq:NN \__char_generate_auxii:nnw \__char_generate_aux:nnw
8528       \cs_gset:Npn \__char_generate_aux:nnw #1#2#3 \exp_end:
8529       {
8530         #3
8531         \if_int_compare:w #2 = 13 \exp_stop_f:
8532           \__kernel_msg_expandable_error:nn { kernel } { char-active }
8533         \else:
8534           \__char_generate_auxii:nnw {#1} {#2}
8535         \fi:
8536         \exp_end:
8537       }
8538     }
8539   }
8540 }
```

`\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. There are a few things to notice here. As `^^L` is `\outer` we need to locally set it to avoid a problem. To get open/close braces into the list, they are set up using `\if_false:` pairing and are then x-type expanded together into the desired form.

```

8541   \tl_set:Nn \l__char_tmp_tl { \exp_not:N \or: }
8542   \char_set_catcode_group_begin:n { 0 } % {
8543   \tl_put_right:Nn \l__char_tmp_tl { `^`@ \if_false: } }
```

```

8544   \char_set_catcode_group_end:n { 0 }
8545   \tl_put_right:Nn \l__char_tmp_tl { { \fi: \exp_not:N \or: ^@ } % }
8546   \tl_set:Nx \l__char_tmp_tl { \l__char_tmp_tl }
8547   \char_set_catcode_math_toggle:n { 0 }
8548   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8549   \char_set_catcode_alignment:n { 0 }
8550   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8551   \tl_put_right:Nn \l__char_tmp_tl { \or: }
8552   \char_set_catcode_parameter:n { 0 }
8553   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8554   \char_set_catcode_math_superscript:n { 0 }
8555   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8556   \char_set_catcode_math_subscript:n { 0 }
8557   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8558   \tl_put_right:Nn \l__char_tmp_tl { \or: }

```

For making spaces, there needs to be an o-type expansion of a \use:n (or some other tokenization) to avoid dropping the space. We also set up active tokens although they are (currently) filtered out by the interface layer (\Ucharcat cannot make active tokens).

```

8559   \char_set_catcode_space:n { 0 }
8560   \tl_put_right:No \l__char_tmp_tl { \use:n { \or: } ^@ }
8561   \char_set_catcode_letter:n { 0 }
8562   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8563   \char_set_catcode_other:n { 0 }
8564   \tl_put_right:Nn \l__char_tmp_tl { \or: ^@ }
8565   \char_set_catcode_active:n { 0 }
8566   \tl_put_right:Nn \l__char_tmp_tl { \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. In package mode, ^L is awkward hence this is done in three parts. Notice that at this stage ^@ is active.

```

8567   \cs_set_protected:Npn \__char_tmp:n #1
8568   {
8569     \char_set_lccode:nn { 0 } {#1}
8570     \char_set_lccode:nn { 32 } {#1}
8571     \exp_args:Nx \tex_lowercase:D
8572     {
8573       \tl_const:Nn
8574         \exp_not:c { c__char_ \__char_int_to_roman:w #1 _tl }
8575         { \exp_not:o \l__char_tmp_tl }
8576     }
8577   }
8578 (*package)
8579   \int_step_function:nnN { 0 } { 11 } \__char_tmp:n
8580   \group_begin:
8581     \tl_replace_once:Nnn \l__char_tmp_tl { ^@ } { \ERROR }
8582     \__char_tmp:n { 12 }
8583   \group_end:
8584   \int_step_function:nnN { 13 } { 255 } \__char_tmp:n
8585 
```

```

8586   /*\package
8587   /*\initex
8588   \int_step_function:nnN { 0 } { 255 } \__char_tmp:n
8589 
```

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:`.

```

8589     \cs_new:Npn \__char_generate_aux:nw #1#2#3 \exp_end:
8590     {
8591         #3
8592         \if_false: { \fi:
8593             \exp_after:wN \exp_after:wN
8594             \exp_after:wN \exp_end:
8595             \exp_after:wN \exp_after:wN
8596             \if_case:w #2
8597                 \exp_last_unbraced:Nv \exp_stop_f:
8598                 { c__char_ \__char_int_to_roman:w #1 _tl }
8599             \or:
8600             \fi:
8601         }
8602         \fi:
8603     \group_end:

```

(End definition for `\char_generate:nn` and others. This function is documented on page 118.)

`\c_catcode_other_space_tl` Create a space with category code 12: an “other” space.

```

8604 \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 118.)

### 15.3 Generic tokens

8605 `<@=token>`

`\token_to_meaning:N` These are all defined in `l3basics`, as they are needed “early”. This is just a reminder!

`\token_to_meaning:c`  
`\token_to_meaning:c`  
`\token_to_str:N`  
`\token_to_str:c`

(End definition for `\token_to_meaning:N` and `\token_to_str:N`. These functions are documented on page 122.)

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.

```

8606 \group_begin:
8607     \__kernel_chk_if_free_cs:N \c_group_begin_token
8608     \tex_global:D \tex_let:D \c_group_begin_token {
8609         \__kernel_chk_if_free_cs:N \c_group_end_token
8610         \tex_global:D \tex_let:D \c_group_end_token }
8611         \char_set_catcode_math_toggle:N *
8612         \cs_new_eq:NN \c_math_toggle_token *
8613         \char_set_catcode_alignment:N *
8614         \cs_new_eq:NN \c_alignment_token *
8615         \cs_new_eq:NN \c_parameter_token #
8616         \cs_new_eq:NN \c_math_superscript_token ^
8617         \char_set_catcode_math_subscript:N *
8618         \cs_new_eq:NN \c_math_subscript_token *

```

```

8619   \__kernel_chk_if_free_cs:N \c_space_token
8620   \use:n { \tex_global:D \tex_let:D \c_space_token = ~ } ~
8621   \cs_new_eq:NN \c_catcode_letter_token a
8622   \cs_new_eq:NN \c_catcode_other_token 1
8623 \group_end:

```

(End definition for `\c_group_begin_token` and others. These functions are documented on page 122.)

`\c_catcode_active_tl` Not an implicit token!

```

8624 \group_begin:
8625   \char_set_catcode_active:N \*
8626   \tl_const:Nn \c_catcode_active_tl { \exp_not:N * }
8627 \group_end:

```

(End definition for `\c_catcode_active_tl`. This variable is documented on page 122.)

## 15.4 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.

```

8628 \prg_new_conditional:Npnn \token_if_group_begin:N #1 { p , T , F , TF }
8629   {
8630     \if_catcode:w \exp_not:N #1 \c_group_begin_token
8631     \prg_return_true: \else: \prg_return_false: \fi:
8632   }

```

(End definition for `\token_if_group_begin:NTF`. This function is documented on page 123.)

`\token_if_group_end_p:N` Check if token is a end group token. We use the constant `\c_group_end_token` for this.

`\token_if_group_end:NTF`

```

8633 \prg_new_conditional:Npnn \token_if_group_end:N #1 { p , T , F , TF }
8634   {
8635     \if_catcode:w \exp_not:N #1 \c_group_end_token
8636     \prg_return_true: \else: \prg_return_false: \fi:
8637   }

```

(End definition for `\token_if_group_end:NTF`. This function is documented on page 123.)

`\token_if_math_toggle_p:N` Check if token is a math shift token. We use the constant `\c_math_toggle_token` for this.

```

8638 \prg_new_conditional:Npnn \token_if_math_toggle:N #1 { p , T , F , TF }
8639   {
8640     \if_catcode:w \exp_not:N #1 \c_math_toggle_token
8641     \prg_return_true: \else: \prg_return_false: \fi:
8642   }

```

(End definition for `\token_if_math_toggle:NTF`. This function is documented on page 123.)

`\token_if_alignment_p:N` Check if token is an alignment tab token. We use the constant `\c_alignment_token` for this.

```

8643 \prg_new_conditional:Npnn \token_if_alignment:N #1 { p , T , F , TF }
8644   {
8645     \if_catcode:w \exp_not:N #1 \c_alignment_token
8646     \prg_return_true: \else: \prg_return_false: \fi:
8647   }

```

(End definition for `\token_if_alignment:NTF`. This function is documented on page 123.)

`\token_if_parameter_p:N` Check if token is a parameter token. We use the constant `\c_parameter_token` for this.  
`\token_if_parameter:NTF` 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.

```
8648 \group_begin:
8649   \cs_set_eq:NN \c_parameter_token \scan_stop:
8650   \prg_new_conditional:Npnn \token_if_parameter:N #1 { p , T , F , TF }
8651   {
8652     \if_catcode:w \exp_not:N #1 \c_parameter_token
8653       \prg_return_true: \else: \prg_return_false: \fi:
8654     }
8655 \group_end:
```

(End definition for `\token_if_parameter:NTF`. This function is documented on page 123.)

`\token_if_math_superscript_p:N` Check if token is a math superscript token. We use the constant `\c_math_superscript_token` for this.  
`\token_if_math_superscript:NTF`

```
8656 \prg_new_conditional:Npnn \token_if_math_superscript:N #1
8657   { p , T , F , TF }
8658   {
8659     \if_catcode:w \exp_not:N #1 \c_math_superscript_token
8660       \prg_return_true: \else: \prg_return_false: \fi:
8661     }
```

(End definition for `\token_if_math_superscript:NTF`. This function is documented on page 123.)

`\token_if_math_subscript_p:N` Check if token is a math subscript token. We use the constant `\c_math_subscript_token` for this.  
`\token_if_math_subscript:NTF`

```
8662 \prg_new_conditional:Npnn \token_if_math_subscript:N #1 { p , T , F , TF }
8663   {
8664     \if_catcode:w \exp_not:N #1 \c_math_subscript_token
8665       \prg_return_true: \else: \prg_return_false: \fi:
8666     }
```

(End definition for `\token_if_math_subscript:NTF`. This function is documented on page 123.)

`\token_if_space_p:N` Check if token is a space token. We use the constant `\c_space_token` for this.  
`\token_if_space:NTF`

```
8667 \prg_new_conditional:Npnn \token_if_space:N #1 { p , T , F , TF }
8668   {
8669     \if_catcode:w \exp_not:N #1 \c_space_token
8670       \prg_return_true: \else: \prg_return_false: \fi:
8671     }
```

(End definition for `\token_if_space:NTF`. This function is documented on page 123.)

`\token_if_letter_p:N` Check if token is a letter token. We use the constant `\c_catcode_letter_token` for this.  
`\token_if_letter:NTF`

```
8672 \prg_new_conditional:Npnn \token_if_letter:N #1 { p , T , F , TF }
8673   {
8674     \if_catcode:w \exp_not:N #1 \c_catcode_letter_token
8675       \prg_return_true: \else: \prg_return_false: \fi:
8676     }
```

(End definition for `\token_if_letter:NTF`. This function is documented on page 124.)

`\token_if_other_p:N` Check if token is an other char token. We use the constant `\c_catcode_other_token` for this.

```
8677 \prg_new_conditional:Npnn \token_if_other:N #1 { p , T , F , TF }
8678 {
8679     \if_catcode:w \exp_not:N #1 \c_catcode_other_token
8680         \prg_return_true: \else: \prg_return_false: \fi:
8681 }
```

(End definition for `\token_if_other:NTF`. This function is documented on page 124.)

`\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.

```
8682 \prg_new_conditional:Npnn \token_if_active:N #1 { p , T , F , TF }
8683 {
8684     \if_catcode:w \exp_not:N #1 \c_catcode_active_tl
8685         \prg_return_true: \else: \prg_return_false: \fi:
8686 }
```

(End definition for `\token_if_active:NTF`. This function is documented on page 124.)

`\token_if_eq_meaning_p:NN` Check if the tokens #1 and #2 have same meaning.

```
8687 \prg_new_conditional:Npnn \token_if_eq_meaning:NN #1#2 { p , T , F , TF }
8688 {
8689     \if_meaning:w #1 #2
8690         \prg_return_true: \else: \prg_return_false: \fi:
8691 }
```

(End definition for `\token_if_eq_meaning:NNTF`. This function is documented on page 124.)

`\token_if_eq_catcode_p:NN` Check if the tokens #1 and #2 have same category code.

```
8692 \prg_new_conditional:Npnn \token_if_eq_catcode:NN #1#2 { p , T , F , TF }
8693 {
8694     \if_catcode:w \exp_not:N #1 \exp_not:N #2
8695         \prg_return_true: \else: \prg_return_false: \fi:
8696 }
```

(End definition for `\token_if_eq_catcode:NNTF`. This function is documented on page 124.)

`\token_if_eq_charcode_p:NN` Check if the tokens #1 and #2 have same character code.

```
8697 \prg_new_conditional:Npnn \token_if_eq_charcode:NN #1#2 { p , T , F , TF }
8698 {
8699     \if_charcode:w \exp_not:N #1 \exp_not:N #2
8700         \prg_return_true: \else: \prg_return_false: \fi:
8701 }
```

(End definition for `\token_if_eq_charcode:NNTF`. This function is documented on page 124.)

`\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 `->`. 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.

```

8702 \use:x
8703 {
8704   \prg_new_conditional:Npn \exp_not:N \token_if_macro:N ##1
8705     { p , T , F , TF }
8706   {
8707     \exp_not:N \exp_after:wN \exp_not:N \__token_if_macro_p:w
8708     \exp_not:N \token_to_meaning:N ##1 \tl_to_str:n { ma : }
8709     \exp_not:N \q_stop
8710   }
8711   \cs_new:Npn \exp_not:N \__token_if_macro_p:w
8712     ##1 \tl_to_str:n { ma } ##2 \cColonStr ##3 \exp_not:N \q_stop
8713 }
8714 {
8715   \str_if_eq:nnTF { #2 } { cro }
8716   { \prg_return_true: }
8717   { \prg_return_false: }
8718 }
```

(End definition for `\token_if_macro:NTF` and `\__token_if_macro_p:w`. This function is documented on page 124.)

**`\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.

```

8719 \prg_new_conditional:Npn \token_if_cs:N #1 { p , T , F , TF }
8720 {
8721   \if_catcode:w \exp_not:N #1 \scan_stop:
8722   \prg_return_true: \else: \prg_return_false: \fi:
8723 }
```

(End definition for `\token_if_cs:NTF`. This function is documented on page 124.)

**`\token_if_expandable_p:N`** Check if token is expandable. We use the fact that T<sub>E</sub>X 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 T<sub>E</sub>X's conditional apparatus).

```

8724 \prg_new_conditional:Npn \token_if_expandable:N #1 { p , T , F , TF }
8725 {
8726   \exp_after:wN \if_meaning:w \exp_not:N #1 #1
8727   \prg_return_false:
8728 \else:
8729   \if cs_exist:N #1
8730     \prg_return_true:
8731   \else:
8732     \prg_return_false:
8733 \fi:
```

```

8734     \fi:
8735 }

```

(End definition for `\token_if_expandable:NTF`. This function is documented on page 124.)

```

\__token_delimit_by_char":w
\__token_delimit_by_count:w
\__token_delimit_by_dimen: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 `\q_stop`, and returns the first one and its delimiter. This result is eventually compared to another string.

```

8736 \group_begin:
8737 \cs_set_protected:Npn \__token_tmp:w #1
8738 {
8739     \use:x
8740     {
8741         \cs_new:Npn \exp_not:c { __token_delimit_by_ #1 :w }
8742             #####1 \tl_to_str:n {#1} #####2 \exp_not:N \q_stop
8743             { #####1 \tl_to_str:n {#1} }
8744     }
8745 }
8746 \__token_tmp:w { char" }
8747 \__token_tmp:w { count }
8748 \__token_tmp:w { dimen }
8749 \__token_tmp:w { macro }
8750 \__token_tmp:w { muskip }
8751 \__token_tmp:w { skip }
8752 \__token_tmp:w { toks }
8753 \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_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 five conditionals, `\cs_if_exist:cT` turns out to be `false`, 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).

```

8754 \group_begin:
8755 \cs_set_protected:Npn \__token_tmp:w #1#2#3
8756 {
8757     \use:x
8758     {
8759         \prg_new_conditional:Npnn \exp_not:c { token_if_ #1 :N } #####
8760             { p , T , F , TF }
8761         {
8762             \cs_if_exist:cT { tex_ #2 :D }
8763             {
8764                 \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 :D }
8765                 \exp_not:N \prg_return_false:
8766                 \exp_not:N \else:
8767                 \exp_not:N \if_meaning:w #####1 \exp_not:c { tex_ #2 def:D }
8768                 \exp_not:N \prg_return_false:
8769                 \exp_not:N \else:
8770             }
8771             \exp_not:N \str_if_eq:eeTF
8772             {
8773                 \exp_not:N \exp_after:wN
8774                 \exp_not:c { __token_delimit_by_ #2 :w }
8775                 \exp_not:N \token_to_meaning:N #####
8776                 ? \tl_to_str:n {#2} \exp_not:N \q_stop
8777             }
8778             { \exp_not:n {#3} }
8779             { \exp_not:N \prg_return_true: }
8780             { \exp_not:N \prg_return_false: }
8781             \cs_if_exist:cT { tex_ #2 :D }
8782             {
8783                 \exp_not:N \fi:
8784                 \exp_not:N \fi:
8785             }
8786         }
8787     }
8788 }
8789 \__token_tmp:w { chardef } { char" } { \token_to_str:N \char" }
8790 \__token_tmp:w { mathchardef } { char" } { \token_to_str:N \mathchar" }
8791 \__token_tmp:w { long_macro } { macro } { \tl_to_str:n { \long } macro }
8792 \__token_tmp:w { protected_macro } { macro }
8793     { \tl_to_str:n { \protected } macro }
8794 \__token_tmp:w { protected_long_macro } { macro }
8795     { \token_to_str:N \protected \tl_to_str:n { \long } macro }
8796 \__token_tmp:w { dim_register } { dimen } { \token_to_str:N \dimen }
8797 \__token_tmp:w { int_register } { count } { \token_to_str:N \count }
8798 \__token_tmp:w { muskip_register } { muskip } { \token_to_str:N \muskip }
8799 \__token_tmp:w { skip_register } { skip } { \token_to_str:N \skip }
8800 \__token_tmp:w { toks_register } { toks } { \token_to_str:N \toks }
8801 \group_end:
```

(End definition for `\token_if_chardef:NTF` and others. These functions are documented on page 125.)

```
\token_if_primitive_p:N
\token_if_primitive:NTF
\__token_if_primitive:NNw
  \__token_if_primitive_space:w
  \__token_if_primitive_nullfont:N
\__token_if_primitive_loop:N
\__token_if_primitive:Nw
  \__token_if_primitive_undefined: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 `\q_stop`.

The meaning of each one of the five `\...mark` primitives has the form  $\langle\text{letters}\rangle:\langle\text{user material}\rangle$ . 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.

```
8802 \tex_chardef:D \c__token_A_int = 'A ~ %
8803 \use:x
8804 {
8805   \prg_new_conditional:Npnn \exp_not:N \token_if_primitive:N ##1
8806     { p , T , F , TF }
8807   {
8808     \exp_not:N \token_if_macro:NTF ##1
8809     \exp_not:N \prg_return_false:
8810   {
8811     \exp_not:N \exp_after:wn \exp_not:N \__token_if_primitive:NNw
8812     \exp_not:N \token_to_meaning:N ##1
8813       \tl_to_str:n { : : : } \exp_not:N \q_stop ##1
8814   }
8815 }
8816 \cs_new:Npn \exp_not:N \__token_if_primitive:NNw
8817   ##1##2 ##3 \c_colon_str ##4 \exp_not:N \q_stop
8818 {
8819   \exp_not:N \tl_if_empty:oTF
8820     { \exp_not:N \__token_if_primitive_space:w ##3 ~ }
8821   {
8822     \exp_not:N \__token_if_primitive_loop:N ##3
8823       \c_colon_str \exp_not:N \q_stop
8824   }
8825   { \exp_not:N \__token_if_primitive_nullfont:N }
8826 }
```

```

8827    }
8828 \cs_new:Npn \__token_if_primitive_space:w #1 ~ { }
8829 \cs_new:Npn \__token_if_primitive_nullfont:N #1
8830 {
8831     \if_meaning:w \tex_nullfont:D #1
8832         \prg_return_true:
8833     \else:
8834         \prg_return_false:
8835     \fi:
8836 }
8837 \cs_new:Npn \__token_if_primitive_loop:N #1
8838 {
8839     \if_int_compare:w '#1 < \c_token_A_int %
8840         \exp_after:wN \__token_if_primitive:Nw
8841         \exp_after:wN #1
8842     \else:
8843         \exp_after:wN \__token_if_primitive_loop:N
8844     \fi:
8845 }
8846 \cs_new:Npn \__token_if_primitive:Nw #1 #2 \q_stop
8847 {
8848     \if:w : #1
8849         \exp_after:wN \__token_if_primitive_undefined:N
8850     \else:
8851         \prg_return_false:
8852         \exp_after:wN \use_none:n
8853     \fi:
8854 }
8855 \cs_new:Npn \__token_if_primitive_undefined:N #1
8856 {
8857     \if_cs_exist:N #1
8858         \prg_return_true:
8859     \else:
8860         \prg_return_false:
8861     \fi:
8862 }

```

(End definition for `\token_if_primitive:NTF` and others. This function is documented on page [126](#).)

## 15.5 Peeking ahead at the next token

8863 `<@@=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.  
\g\_peek\_token

```
8864 \cs_new_eq:NN \l_peek_token ?
8865 \cs_new_eq:NN \g_peek_token ?
```

(End definition for \l\_peek\_token and \g\_peek\_token. These variables are documented on page 126.)

\l\_\_peek\_search\_token The token to search for as an implicit token: cf. \l\_\_peek\_search\_tl.

```
8866 \cs_new_eq:NN \l__peek_search_token ?
```

(End definition for \l\_\_peek\_search\_token.)

\l\_\_peek\_search\_tl The token to search for as an explicit token: cf. \l\_\_peek\_search\_token.

```
8867 \tl_new:N \l__peek_search_tl
```

(End definition for \l\_\_peek\_search\_tl.)

\\_\\_peek\_true:w Functions used by the branching and space-stripping code.

\\_\\_peek\_true\_aux:w  
\\_\\_peek\_false:w  
\\_\\_peek\_tmp:w

```
8868 \cs_new:Npn \_\_peek_true:w { }
8869 \cs_new:Npn \_\_peek_true_aux:w { }
8870 \cs_new:Npn \_\_peek_false:w { }
8871 \cs_new:Npn \_\_peek_tmp:w { }
```

(End definition for \\_\\_peek\_true:w and others.)

\peek\_after:Nw Simple wrappers for \futurelet: no arguments absorbed here.  
\peek\_gafter:Nw

```
8872 \cs_new_protected:Npn \peek_after:Nw
8873   { \tex_futurelet:D \l_peek_token }
8874 \cs_new_protected:Npn \peek_gafter:Nw
8875   { \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 126.)

\\_\\_peek\_true\_remove:w A function to remove the next token and then regain control.

```
8876 \cs_new_protected:Npn \_\_peek_true_remove:w
8877   {
8878     \tex_afterassignment:D \_\_peek_true_aux:w
8879     \cs_set_eq:NN \_\_peek_tmp:w
8880   }
```

(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.

```
8881 \cs_new_protected:Npn \peek_remove_spaces:n #1
8882   {
8883     \cs_set:Npx \_\_peek_false:w { \exp_not:n {\#1} }
8884     \group_align_safe_begin:
8885     \cs_set:Npn \_\_peek_true_aux:w { \peek_after:Nw \_\_peek_remove_spaces: }
8886     \_\_peek_true_aux:w
8887   }
8888 \cs_new_protected:Npn \_\_peek_remove_spaces:
8889   {
8890     \if_meaning:w \l_peek_token \c_space_token
8891       \exp_after:wN \_\_peek_true_remove:w
```

```

8892     \else:
8893         \group_align_safe_end:
8894             \exp_after:wN \__peek_false:w
8895     \fi:
8896 }

```

(End definition for `\peek_remove_spaces:n` and `\__peek_remove_spaces:..`. This function is documented on page 260.)

#### `\__peek_token_generic_aux:NNTF`

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.

```

8897 \cs_new_protected:Npn \__peek_token_generic_aux:NNTF #1#2#3#4#5
8898 {
8899     \group_align_safe_begin:
8900         \cs_set_eq:NN \l__peek_search_token #3
8901         \tl_set:Nn \l__peek_search_tl {#3}
8902         \cs_set:Npx \__peek_true_aux:w
8903         {
8904             \exp_not:N \group_align_safe_end:
8905             \exp_not:n {#4}
8906         }
8907         \cs_set_eq:NN \__peek_true:w #1
8908         \cs_set:Npx \__peek_false:w
8909         {
8910             \exp_not:N \group_align_safe_end:
8911             \exp_not:n {#5}
8912         }
8913         \peek_after:Nw #2
8914     }

```

(End definition for `\__peek_token_generic_aux:NNTF`.)

#### `\__peek_token_generic:NNTF`

For token removal there needs to be a call to the auxiliary function which does the work.

```

8915 \cs_new_protected:Npn \__peek_token_generic:NNTF
8916     { \__peek_token_generic_aux:NNTF \__peek_true_aux:w }
8917 \cs_new_protected:Npn \__peek_token_generic:NNT #1#2#3
8918     { \__peek_token_generic:NNTF #1 #2 {#3} { } }
8919 \cs_new_protected:Npn \__peek_token_generic:NNF #1#2#3
8920     { \__peek_token_generic:NNTF #1 #2 { } {#3} }
8921 \cs_new_protected:Npn \__peek_token_remove_generic:NNTF
8922     { \__peek_token_generic_aux:NNTF \__peek_true_remove:w }
8923 \cs_new_protected:Npn \__peek_token_remove_generic:NNT #1#2#3
8924     { \__peek_token_remove_generic:NNTF #1 #2 {#3} { } }
8925 \cs_new_protected:Npn \__peek_token_remove_generic:NNF #1#2#3
8926     { \__peek_token_remove_generic:NNTF #1 #2 { } {#3} }

```

(End definition for `\__peek_token_generic:NNTF` and `\__peek_token_remove_generic:NNTF`.)

#### `\__peek_execute_branches_meaning:`

The meaning test is straight forward.

```

8927 \cs_new:Npn \__peek_execute_branches_meaning:
8928 {
8929     \if_meaning:w \l__peek_token \l__peek_search_token
8930         \exp_after:wN \__peek_true:w

```

```

8931     \else:
8932         \exp_after:wN \__peek_false:w
8933     \fi:
8934 }

```

(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_tl`. The `\exp_not:N` prevents outer macros (coming from non-L<sup>A</sup>T<sub>E</sub>X3 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.

```

8935 \cs_new:Npn \__peek_execute_branches_catcode:
8936     { \if_catcode:w \__peek_execute_branches_catcode_aux: }
8937 \cs_new:Npn \__peek_execute_branches_charcode:
8938     { \if_charcode:w \__peek_execute_branches_catcode_aux: }
8939 \cs_new:Npn \__peek_execute_branches_catcode_aux:
8940     {
8941         \if_catcode:w \exp_not:N \l__peek_token \scan_stop:
8942             \exp_after:wN \exp_after:wN
8943                 \exp_after:wN \__peek_execute_branches_catcode_auxii:N
8944                 \exp_after:wN \exp_not:N
8945             \else:
8946                 \exp_after:wN \__peek_execute_branches_catcode_auxiii:
8947             \fi:
8948     }
8949 \cs_new:Npn \__peek_execute_branches_catcode_auxii:N #1
8950     {
8951         \exp_not:N #1
8952         \exp_after:wN \exp_not:N \l__peek_search_tl
8953             \exp_after:wN \__peek_true:w
8954             \else:
8955                 \exp_after:wN \__peek_false:w
8956             \fi:
8957             #1
8958     }
8959 \cs_new:Npn \__peek_execute_branches_catcode_auxiii:

```

```

8960    {
8961        \exp_not:N \l_peek_token
8962        \exp_after:wN \exp_not:N \l__peek_search_tl
8963        \exp_after:wN \__peek_true:w
8964    \else:
8965        \exp_after:wN \__peek_false:w
8966    \fi:
8967 }

```

(End definition for `\__peek_execute_branches_catcode:` and others.)

`\peek_catcode:NTF` The public functions themselves cannot be defined using `\prg_new_conditional:Npnn`.  
`\peek_catcode_remove:NTF` Instead, the TF, T, F variants are defined in terms of corresponding variants of `\__-`  
`\peek_charcode:NTF` `\peek_token_generic:NNTF` or `\__peek_token_remove_generic:NNTF`, with first ar-  
`\peek_charcode_remove:NTF` gument one of `\__peek_execute_branches_catcode:`, `\__peek_execute_branches_-`  
`\peek_meaning:NTF` `\__peek_meaning:NNTF`  
`\peek_meaning_remove:NTF`

```

8968 \tl_map_inline:nn { { catcode } { charcode } { meaning } }
8969 {
8970     \tl_map_inline:nn { { } { _remove } }
8971     {
8972         \tl_map_inline:nn { { TF } { T } { F } }
8973         {
8974             \cs_new_protected:cp { peek_ #1 ##1 :N #####1 }
8975             {
8976                 \exp_not:c { __peek_token ##1 _generic:NN #####1 }
8977                 \exp_not:c { __peek_execute_branches_ #1 : }
8978             }
8979         }
8980     }
8981 }

```

(End definition for `\peek_catcode:NTF` and others. These functions are documented on page 126.)

To ignore spaces, remove them using `\peek_remove_spaces:n` before running the tests.

```

8982 \tl_map_inline:nn
8983 {
8984     { catcode } { catcode_remove }
8985     { charcode } { charcode_remove }
8986     { meaning } { meaning_remove }
8987 }
8988 {
8989     \cs_new_protected:cp { peek_#1_ignore_spaces:NTF } ##1##2##3
8990     {
8991         \peek_remove_spaces:n
8992         { \exp_not:c { peek_#1:NTF } ##1 {##2} {##3} }
8993     }
8994     \cs_new_protected:cp { peek_#1_ignore_spaces:NT } ##1##2
8995     {
8996         \peek_remove_spaces:n
8997         { \exp_not:c { peek_#1:NT } ##1 {##2} }
8998     }
8999     \cs_new_protected:cp { peek_#1_ignore_spaces:NF } ##1##2
9000     {
9001         \peek_remove_spaces:n

```

```

9002         { \exp_not:c { peek_#1:NF } ##1 {##2} }
9003     }
9004 }
```

(End definition for `\peek_catcode_ignore_spaces:NTF` and others. These functions are documented on page 127.)

## 15.6 Decomposing a macro definition

```

\token_get_prefix_spec:N
\token_get_arg_spec:N
\token_get_replacement_spec:N
```

```
\_peek_get_prefix_arg_replacement:wN
```

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 argument 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.

```

9005 \exp_args:Nno \use:nn
9006   { \cs_new:Npn \_peek_get_prefix_arg_replacement:wN #1 }
9007   { \tl_to_str:n { macro : } #2 -> #3 \q_stop #4 }
9008   { #4 {#1} {#2} {#3} }
9009 \cs_new:Npn \token_get_prefix_spec:N #1
9010   {
9011     \token_if_macro:NTF #1
9012     {
9013       \exp_after:wN \_peek_get_prefix_arg_replacement:wN
9014       \token_to_meaning:N #1 \q_stop \use_i:nnn
9015     }
9016     { \scan_stop: }
9017   }
9018 \cs_new:Npn \token_get_arg_spec:N #1
9019   {
9020     \token_if_macro:NTF #1
9021     {
9022       \exp_after:wN \_peek_get_prefix_arg_replacement:wN
9023       \token_to_meaning:N #1 \q_stop \use_ii:nnn
9024     }
9025     { \scan_stop: }
9026   }
9027 \cs_new:Npn \token_get_replacement_spec:N #1
9028   {
9029     \token_if_macro:NTF #1
9030     {
9031       \exp_after:wN \_peek_get_prefix_arg_replacement:wN
9032       \token_to_meaning:N #1 \q_stop \use_iii:nnn
9033     }
9034     { \scan_stop: }
9035   }
```

(End definition for `\token_get_prefix_spec:N` and others. These functions are documented on page 129.)

```
9036 </initex | package>
```

## 16 I3prop implementation

The following test files are used for this code: *m3prop001*, *m3prop002*, *m3prop003*, *m3prop004*, *m3show001*.

```
9037  {*initex | package}
9038  <@@=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_t1` 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_t1`.)

---

`\__prop_split:NnTF` `\__prop_split:NnTF <property list> {<key>} {<true code>} {<false code>}`

Updated: 2013-01-08  
Splits the `<property list>` at the `<key>`, giving three token lists: the `<extract>` of `<property list>` before the `<key>`, the `<value>` associated with the `<key>` and the `<extract>` of the `<property list>` after the `<value>`. Both `<extracts>` retain the internal structure of a property list, and the concatenation of the two `<extracts>` is a property list. If the `<key>` is present in the `<property list>` then the `<true code>` is left in the input stream, with #1, #2, and #3 replaced by the first `<extract>`, the `<value>`, and the second extract. If the `<key>` is not present in the `<property list>` then the `<false code>` is left in the input stream, with no trailing material. Both `<true code>` and `<false code>` 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 `<true code>` for the three extracts from the property list. The `<key>` comparison takes place as described for `\str_if_eq:nn`.

`\s__prop` A private scan mark is used as a marker after each key, and at the very beginning of the property list.

```
9039  \scan_new:N \s__prop
```

(End definition for `\s__prop`.)

<code>\__prop_pair:wn</code>	The delimiter is always defined, but when misused simply triggers an error and removes its argument.
	<pre>9040 \cs_new:Npn \__prop_pair:wn #1 \s__prop #2   { \__kernel_msg_expandable_error:nn { kernel } { misused-prop } }</pre> <p>(End definition for <code>\__prop_pair:wn</code>.)</p>
<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.
	<pre>9042 \tl_new:N \l__prop_internal_tl</pre> <p>(End definition for <code>\l__prop_internal_tl</code>.)</p>

`\c_empty_prop` An empty prop.

```
9043 \tl_const:Nn \c_empty_prop { \s__prop }
```

(End definition for `\c_empty_prop`. This variable is documented on page [137](#).)

## 16.1 Allocation and initialisation

`\prop_new:N` Property lists are initialized with the value `\c_empty_prop`.

```
9044 \cs_new_protected:Npn \prop_new:N #1
  {
    \__kernel_chk_if_free_cs:N #1
    \cs_gset_eq:NN #1 \c_empty_prop
  }
\cs_generate_variant:Nn \prop_new:N { c }
```

(End definition for `\prop_new:N`. This function is documented on page [132](#).)

`\prop_clear:N` The same idea for clearing.

```
9050 \cs_new_protected:Npn \prop_clear:N #1
  { \prop_set_eq:NN #1 \c_empty_prop }
\prop_gclear:N
\prop_gclear:c
  \cs_generate_variant:Nn \prop_clear:N { c }
  \cs_new_protected:Npn \prop_gclear:N #1
  { \prop_gset_eq:NN #1 \c_empty_prop }
  \cs_generate_variant:Nn \prop_gclear:N { c }
```

(End definition for `\prop_clear:N` and `\prop_gclear:N`. These functions are documented on page [132](#).)

`\prop_clear_new:N` Once again a simple variation of the token list functions.

```
9056 \cs_new_protected:Npn \prop_clear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_clear:N #1 } { \prop_new:N #1 } }
\prop_gclear_new:N
\prop_gclear_new:c
  \cs_generate_variant:Nn \prop_clear_new:N { c }
  \cs_new_protected:Npn \prop_gclear_new:N #1
  { \prop_if_exist:NTF #1 { \prop_gclear:N #1 } { \prop_new:N #1 } }
  \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 [132](#).)

```

\prop_set_eq:NN These are simply copies from the token list functions.
\prop_set_eq:cN 9062 \cs_new_eq:NN \prop_set_eq:NN \tl_set_eq:NN
\prop_set_eq:Nc 9063 \cs_new_eq:NN \prop_set_eq:Nc \tl_set_eq:Nc
\prop_set_eq:cc 9064 \cs_new_eq:NN \prop_set_eq:cN \tl_set_eq:cN
\prop_gset_eq:NN 9065 \cs_new_eq:NN \prop_set_eq:cc \tl_set_eq:cc
\prop_gset_eq:cN 9066 \cs_new_eq:NN \prop_gset_eq:NN \tl_gset_eq:NN
\prop_gset_eq:Nc 9067 \cs_new_eq:NN \prop_gset_eq:Nc \tl_gset_eq:Nc
\prop_gset_eq:cc 9068 \cs_new_eq:NN \prop_gset_eq:cN \tl_gset_eq:cN
9069 \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 132.)

```

\l_tmpa_prop We can now initialize the scratch variables.
\l_tmpb_prop 9070 \prop_new:N \l_tmpa_prop
\g_tmpa_prop 9071 \prop_new:N \l_tmpb_prop
\g_tmpb_prop 9072 \prop_new:N \g_tmpa_prop
9073 \prop_new:N \g_tmpb_prop

```

(End definition for `\l_tmpa_prop` and others. These variables are documented on page 137.)

```

\prop_set_from_keyval:Nn Loop through items separated by commas, with \q_mark to avoid losing braces. After
\prop_set_from_keyval:cn checking for termination, split the item at the first then at the second = (which ought
\prop_gset_from_keyval:Nn to be the first of the trailing =). At both splits, trim spaces and call \__prop_from_
\prop_gset_from_keyval:cn \__prop_from_keyval_key:w, then \__prop_from_keyval_value:w, followed by the trimmed material,
\prop_const_from_keyval:Nn \q_nil, the subsequent part of the item, and the trailing =’s and \q_stop. After finding
\prop_const_from_keyval:cn the <key> just store it after \q_stop. After finding the <value> ignore completely empty
\__prop_from_keyval:n items (both trailing = were used as delimiters and all parts are empty); if the remaining
\__prop_from_keyval_loop:w part #2 consists exactly of the second trailing = (namely there was exactly one = in the
\__prop_from_keyval_split:Nw item) then output one key–value pair for the property list; otherwise complain about a
\__prop_from_keyval_key:n missing or extra =.
\__prop_from_keyval_key:w 9074 \cs_new_protected:Npn \prop_set_from_keyval:Nn #1#2
\__prop_from_keyval_value:n 9075 { \tl_set:Nx #1 { \__prop_from_keyval:n {#2} } }
\__prop_from_keyval_value:w 9076 \cs_generate_variant:Nn \prop_set_from_keyval:Nn { c }
\__prop_from_keyval_value:w 9077 \cs_new_protected:Npn \prop_gset_from_keyval:Nn #1#2
\__prop_from_keyval_value:n 9078 { \tl_gset:Nx #1 { \__prop_from_keyval:n {#2} } }
\__prop_from_keyval_value:w 9079 \cs_generate_variant:Nn \prop_gset_from_keyval:Nn { c }
9080 \cs_new_protected:Npn \prop_const_from_keyval:Nn #1#2
9081 { \tl_const:Nx #1 { \__prop_from_keyval:n {#2} } }
9082 \cs_generate_variant:Nn \prop_const_from_keyval:Nn { c }
9083 \cs_new:Npn \__prop_from_keyval:n #1
9084 {
9085     \s__prop
9086     \__prop_from_keyval_loop:w \q_mark #1 ,
9087         \q_recursion_tail , \q_recursion_stop
9088 }
9089 \cs_new:Npn \__prop_from_keyval_loop:w #1 ,
9090 {
9091     \quark_if_recursion_tail_stop:o { \use_none:n #1 }
9092     \__prop_from_keyval_split:Nw \__prop_from_keyval_key:n
9093         #1 = = \q_stop { \use_none:n #1 }
9094     \__prop_from_keyval_loop:w \q_mark
9095 }
9096 \cs_new:Npn \__prop_from_keyval_split:Nw #1#2 =

```

```

9097      {
9098        \tl_trim_spaces_apply:oN { \use_none:n #2 } #1
9099        \q_nil
9100      }
9101 \cs_new:Npn \__prop_from_keyval_key:n #1
9102   { \__prop_from_keyval_key:w #1 }
9103 \cs_new:Npn \__prop_from_keyval_key:w #1 \q_nil #2 \q_stop
9104   {
9105     \__prop_from_keyval_split:Nw \__prop_from_keyval_value:n
9106     \q_mark #2 \q_stop {#1}
9107   }
9108 \cs_new:Npn \__prop_from_keyval_value:n #1
9109   { \__prop_from_keyval_value:w #1 }
9110 \cs_new:Npn \__prop_from_keyval_value:w #1 \q_nil #2 \q_stop #3#4
9111   {
9112     \tl_if_empty:nF { #3 #1 #2 }
9113     {
9114       \str_if_eq:nnTF {#2} { = }
9115       {
9116         \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
9117         \s__prop { \exp_not:n {#1} }
9118       }
9119       {
9120         \__kernel_msg_expandable_error:nnf
9121         { kernel } { prop-keyval }
9122         { \exp_after:wN \exp_stop_f: #4 }
9123       }
9124     }
9125   }

```

(End definition for `\prop_set_from_keyval:Nn` and others. These functions are documented on page 249.)

## 16.2 Accessing data in property lists

`\__prop_split:NnTF`, `\__prop_split_aux:NnTF` and `\__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 \q_mark #4 #5 \q_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_i:nn`, which keeps the *false code*.

```

9126 \cs_new_protected:Npn \__prop_split:NnTF #1#2
9127   { \exp_args:NNo \__prop_split_aux:NnTF #1 { \tl_to_str:n {#2} } }
9128 \cs_new_protected:Npn \__prop_split_aux:NnTF #1#2#3#4
9129   {
9130     \cs_set:Npn \__prop_split_aux:w ##1
9131       \__prop_pair:wn #2 \s__prop ##2 ##3 \q_mark ##4 ##5 \q_stop
9132       { ##4 {#3} {#4} }
9133       \exp_after:wN \__prop_split_aux:w #1 \q_mark \use_i:nn
9134         \__prop_pair:wn #2 \s__prop { } \q_mark \use_i:nn \q_stop
9135   }
9136 \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.

```

9137 \cs_new_protected:Npn \prop_remove:Nn #1#2
9138   {
9139     \__prop_split:NnTF #1 {#2}
9140     { \tl_set:Nn #1 { ##1 ##3 } }
9141     { }
9142   }
9143 \cs_new_protected:Npn \prop_gremove:Nn #1#2
9144   {
9145     \__prop_split:NnTF #1 {#2}
9146     { \tl_gset:Nn #1 { ##1 ##3 } }
9147     { }
9148   }
9149 \cs_generate_variant:Nn \prop_remove:Nn { NV }
9150 \cs_generate_variant:Nn \prop_remove:Nn { c , cV }
9151 \cs_generate_variant:Nn \prop_gremove:Nn { NV }
9152 \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 134.)

```

**\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.

```

9153 \cs_new_protected:Npn \prop_get:NnN #1#2#3
9154   {
9155     \__prop_split:NnTF #1 {#2}
9156     { \tl_set:Nn #3 {##2} }
9157     { \tl_set:Nn #3 { \q_no_value } }
9158   }
9159 \cs_generate_variant:Nn \prop_get:NnN { NV , No }
9160 \cs_generate_variant:Nn \prop_get:NnN { c , cV , co }

(End definition for \prop_get:NnN. This function is documented on page 133.)

```

**\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.

```

9161 \cs_new_protected:Npn \prop_pop:NnN #1#2#3
9162   {
9163     \__prop_split:NnTF #1 {#2}
9164     { \tl_set:Nn #3 {##2} }
9165     { \tl_set:Nn #3 { \q_no_value } }
9166   }
9167 \cs_generate_variant:Nn \prop_pop:NnN { NV , No }
9168 \cs_generate_variant:Nn \prop_pop:NnN { c , cV , co }

(End definition for \prop_pop:NnN. This function is documented on page 133.)

```

```

9163     \__prop_split:NnTF #1 {##2}
9164     {
9165         \tl_set:Nn #3 {##2}
9166         \tl_set:Nn #1 { ##1 ##3 }
9167     }
9168     { \tl_set:Nn #3 { \q_no_value } }
9169 }
9170 \cs_new_protected:Npn \prop_gpop:Nnn #1#2#3
9171 {
9172     \__prop_split:NnTF #1 {##2}
9173     {
9174         \tl_set:Nn #3 {##2}
9175         \tl_gset:Nn #1 { ##1 ##3 }
9176     }
9177     { \tl_set:Nn #3 { \q_no_value } }
9178 }
9179 \cs_generate_variant:Nn \prop_pop:NnN { No }
9180 \cs_generate_variant:Nn \prop_pop:NnN { c , co }
9181 \cs_generate_variant:Nn \prop_gpop:NnN { No }
9182 \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 133.)

**\prop\_item:Nn**  
**\prop\_item:cn**  
**\\_\_prop\_item\_Nn:nwwn**

Getting the value corresponding to a key in a property list in an expandable fashion is similar to mapping some tokens. Go through the property list one  $\langle key \rangle - \langle value \rangle$  pair at a time: the arguments of `\__prop_item_Nn:nwwn` are the  $\langle key \rangle$  we are looking for, a  $\langle key \rangle$  of the property list, and its associated value. The  $\langle keys \rangle$  are compared (as strings). If they match, the  $\langle value \rangle$  is returned, within `\exp_not:n`. The loop terminates even if the  $\langle key \rangle$  is missing, and yields an empty value, because we have appended the appropriate  $\langle key \rangle - \langle empty value \rangle$  pair to the property list.

```

9183 \cs_new:Npn \prop_item:Nn #1#2
9184 {
9185     \exp_last_unbraced:Noo \__prop_item_Nn:nwwn { \tl_to_str:n {##2} } #1
9186     \__prop_pair:wn \tl_to_str:n {##2} \s__prop { }
9187     \prg_break_point:
9188 }
9189 \cs_new:Npn \__prop_item_Nn:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
9190 {
9191     \str_if_eq:eeTF {##1} {##3}
9192     { \prg_break:n { \exp_not:n {##4} } }
9193     { \__prop_item_Nn:nwwn {##1} }
9194 }
9195 \cs_generate_variant:Nn \prop_item:Nn { c }

```

(End definition for `\prop_item:Nn` and `\__prop_item_Nn:nwwn`. This function is documented on page 134.)

**\prop\_pop:NnTF**  
**\prop\_pop:cnNTF**  
**\prop\_gpop:NnTF**  
**\prop\_gpop:cnNTF**

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.

```

9196 \prg_new_protected_conditional:Npnn \prop_pop:NnN #1#2#3 { T , F , TF }
9197 {
9198     \__prop_split:NnTF #1 {##2}
9199 }

```

```

9200      \tl_set:Nn ##2}
9201      \tl_set:Nn #1 { ##1 ##3 }
9202      \prg_return_true:
9203  }
9204  { \prg_return_false:
9205  }
9206 \prg_new_protected_conditional:Npnn \prop_gpop:NnN #1#2#3 { T , F , TF }
9207  {
9208      \__prop_split:NnTF #1 {#2}
9209  {
9210      \tl_set:Nn ##2}
9211      \tl_gset:Nn #1 { ##1 ##3 }
9212      \prg_return_true:
9213  }
9214  { \prg_return_false:
9215  }
9216 \prg_generate_conditional_variant:Nnn \prop_pop:NnN { c } { T , F , TF }
9217 \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 [135](#).)

**\prop\_put:Nnn** Since the branches of `\__prop_split:NnTF` are used as the replacement text of an internal macro, and since the `<key>` and new `<value>` 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 `<key>` 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 `<key>` in the property list, preserving the order of entries.

```

9218 \cs_new_protected:Npn \prop_put:Nnn { \__prop_put:NNnn \tl_set:Nx }
9219 \cs_new_protected:Npn \prop_gput:Nnn { \__prop_put:NNnn \tl_gset:Nx }
9220 \cs_new_protected:Npn \__prop_put:NNnn #1#2#3#4
9221  {
9222      \tl_set:Nn \l__prop_internal_tl
9223  {
9224      \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
9225      \s__prop { \exp_not:n {#4} }
9226  }
9227      \__prop_split:NnTF #2 {#3}
9228      { #1 #2 { \exp_not:n {##1} \l__prop_internal_tl \exp_not:n {##3} } }
9229      { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
9230  }
9231 \cs_generate_variant:Nn \prop_put:Nnn
9232  { Nv , Nno , Nnx , NV , Nvv , No , Noo }
9233 \cs_generate_variant:Nn \prop_put:Nnn
9234  { c , cnv , cno , cnx , cv , cV , co , coo }
9235 \cs_generate_variant:Nn \prop_gput:Nnn
9236  { Nv , Nno , Nnx , NV , Nvv , No , Noo }
9237 \cs_generate_variant:Nn \prop_gput:Nnn
9238  { c , cnv , cno , cnx , cv , cV , co , coo }

(End definition for \prop_put:Nnn, \prop_gput:Nnn, and \__prop_put:NNnn. These functions are documented on page 133.)
```

```

\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
9239 \cs_new_protected:Npn \prop_put_if_new:Nnn
9240   { \__prop_put_if_new:NNnn \tl_set:Nx }
9241 \cs_new_protected:Npn \prop_gput_if_new:Nnn
9242   { \__prop_put_if_new:NNnn \tl_gset:Nx }
9243 \cs_new_protected:Npn \__prop_put_if_new:NNnn #1#2#3#4
9244   {
9245     \tl_set:Nn \l__prop_internal_tl
9246     {
9247       \exp_not:N \__prop_pair:wn \tl_to_str:n {#3}
9248       \s__prop \exp_not:n {#4}
9249     }
9250     \__prop_split:NnTF #2 {#3}
9251     {
9252       { #1 #2 { \exp_not:o {#2} \l__prop_internal_tl } }
9253     }
9254 \cs_generate_variant:Nn \prop_put_if_new:Nnn { c }
9255 \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 [133](#).)

### 16.3 Property list conditionals

```

\prop_if_exist_p:N Copies of the cs functions defined in l3basics.
\prop_if_exist_p:c
\prop_if_exist:NTF
\prop_if_exist:cTF
9256 \prg_new_eq_conditional>NNNn \prop_if_exist:N \cs_if_exist:N
9257   { TF , T , F , p }
9258 \prg_new_eq_conditional>NNNn \prop_if_exist:c \cs_if_exist:c
9259   { TF , T , F , p }

```

(End definition for `\prop_if_exist:NTF`. This function is documented on page [134](#).)

`\prop_if_empty_p:N` Same test as for token lists.

```

\prop_if_empty_p:c
\prop_if_empty:NTF
\prop_if_empty:cTF
9260 \prg_new_conditional:Npnn \prop_if_empty:N #1 { p , T , F , TF }
9261   {
9262     \tl_if_eq:NNTF #1 \c_empty_prop
9263     \prg_return_true: \prg_return_false:
9264   }
9265 \prg_generate_conditional_variant:Nnn \prop_if_empty:N
9266   { c } { p , T , F , TF }

```

(End definition for `\prop_if_empty:NTF`. This function is documented on page [134](#).)

`\prop_if_in_p:Nn` Testing expandably if a key is in a property list requires to go through the key–value pairs one by one. This is rather slow, and a faster test would be

```

\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:NvTF
\prop_if_in:NotF
\prop_if_in:cNTF
\prop_if_in:cVTF
\prop_if_in:cOTF
\_prop_if_in:nwvn
\_prop_if_in:N
9267 \prg_new_protected_conditional:Npnn \prop_if_in:Nn #1 #2
9268   {
9269     \__split:NnTF #1 {#2}
9270     {
9271       { \prg_return_true: }
9272       { \prg_return_false: }
9273     }
9274   }

```

but `\_prop_split:NnTF` is non-expandable.

Instead, the key is compared to each key in turn using `\str_if_eq:ee`, which is expandable. To terminate the mapping, we append to the property list the key that is searched for. This second `\tl_to_str:n` is not expanded at the start, but only when included in the `\str_if_eq:ee`. It cannot make the breaking mechanism choke, because the arbitrary token list material is enclosed in braces. The second argument of `\_prop_if_in:nwwn` is most often empty. When the `\langle key` is found in the list, `\_prop_if_in:N` receives `\_prop_pair:wn`, and if it is found as the extra item, the function receives `\q_recursion_tail`, easily recognizable.

Here, `\prop_map_function>NN` is not sufficient for the mapping, since it can only map a single token, and cannot carry the key that is searched for.

```

9267 \prg_new_conditional:Npnn \prop_if_in:Nn #1#2 { p , T , F , TF }
9268 {
9269   \exp_last_unbraced:Noo \_prop_if_in:nwwn { \tl_to_str:n {#2} } #1
9270   \_prop_pair:wn \tl_to_str:n {#2} \s__prop { }
9271   \q_recursion_tail
9272   \prg_break_point:
9273 }
9274 \cs_new:Npn \_prop_if_in:nwwn #1#2 \_prop_pair:wn #3 \s__prop #4
9275 {
9276   \str_if_eq:eeTF {#1} {#3}
9277   { \_prop_if_in:N }
9278   { \_prop_if_in:nwwn {#1} }
9279 }
9280 \cs_new:Npn \_prop_if_in:N #1
9281 {
9282   \if_meaning:w \q_recursion_tail #1
9283   \prg_return_false:
9284   \else:
9285   \prg_return_true:
9286   \fi:
9287   \prg_break:
9288 }
9289 \prg_generate_conditional_variant:Nnn \prop_if_in:Nn
9290 { NV , No , c , cv , co } { p , T , F , TF }

```

(End definition for `\prop_if_in:NnTF`, `\_prop_if_in:nwwn`, and `\_prop_if_in:N`. This function is documented on page 134.)

## 16.4 Recovering values from property lists with branching

`\prop_get:NnTF` 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.

`\prop_get:NVNTF`

`\prop_get:NoNTF`

`\prop_get:cnNTF`

`\prop_get:cVNTF`

`\prop_get:coNTF`

```

9291 \prg_new_protected_conditional:Npnn \prop_get:NnN #1#2#3 { T , F , TF }
9292 {
9293   \_prop_split:NnTF #1 {#2}
9294   {
9295     \tl_set:Nn #3 {##2}
9296     \prg_return_true:
9297   }
9298   { \prg_return_false: }
9299 }
```

```

9300 \prg_generate_conditional_variant:Nnn \prop_get:NnN
9301   { NV , No , c , cV , co } { T , F , TF }

```

(End definition for `\prop_get:NnNTF`. This function is documented on page 135.)

## 16.5 Mapping to property lists

`\prop_map_function:NN` The argument delimited by `\__prop_pair:wn` is empty except at the end of the loop where it is `\prg_break:..` No need for any quark test.

```

9302 \cs_new:Npn \prop_map_function:NN #1#2
9303   {
9304     \exp_after:wN \use_i_i:nnn
9305     \exp_after:wN \__prop_map_function:Nwwn
9306     \exp_after:wN #2
9307     #1
9308     \prg_break: \__prop_pair:wn \s__prop { } \prg_break_point:
9309     \prg_break_point:Nn \prop_map_break: { }
9310   }
9311 \cs_new:Npn \__prop_map_function:Nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
9312   {
9313     #2
9314     #1 {#3} {#4}
9315     \__prop_map_function:Nwwn #1
9316   }
9317 \cs_generate_variant:Nn \prop_map_function:NN { Nc , c , cc }

```

(End definition for `\prop_map_function:NN` and `\__prop_map_function:Nwwn`. This function is documented on page 135.)

`\prop_map_inline:Nn` 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.

```

9318 \cs_new_protected:Npn \prop_map_inline:Nn #1#2
9319   {
9320     \cs_gset_eq:cN
9321       { __prop_map_ \int_use:N \g_kernel_prg_map_int :wn } \__prop_pair:wn
9322       \int_gincr:N \g_kernel_prg_map_int
9323     \cs_gset_protected:Npn \__prop_pair:wn ##1 \s__prop ##2 {#2}
9324     #1
9325     \prg_break_point:Nn \prop_map_break:
9326     {
9327       \int_gdecr:N \g_kernel_prg_map_int
9328       \cs_gset_eq:Nc \__prop_pair:wn
9329         { __prop_map_ \int_use:N \g_kernel_prg_map_int :wn }
9330     }
9331   }
9332 \cs_generate_variant:Nn \prop_map_inline:Nn { c }

```

(End definition for `\prop_map_inline:Nn`. This function is documented on page 135.)

`\prop_map_break:` The break statements are based on the general `\prg_map_break:Nn`.

`\prop_map_break:n`

```
9333 \cs_new:Npn \prop_map_break:
9334   { \prg_map_break:Nn \prop_map_break: { } }
9335 \cs_new:Npn \prop_map_break:n
9336   { \prg_map_break:Nn \prop_map_break: }
```

(End definition for `\prop_map_break:` and `\prop_map_break:n`. These functions are documented on page 136.)

## 16.6 Viewing property lists

`\prop_show:N` Apply the general `\_kernel_chk_defined:NT` and `\msg_show:nnnnnn`. Contrarily to sequences and comma lists, we use `\msg_show_item:nn` to format both the key and the value for each pair.

`\prop_log:c`

```
9337 \cs_new_protected:Npn \prop_show:N { \_prop_show:NN \msg_show:nnxxxx }
9338 \cs_generate_variant:Nn \prop_show:N { c }
9339 \cs_new_protected:Npn \prop_log:N { \_prop_show:NN \msg_log:nnxxxx }
9340 \cs_generate_variant:Nn \prop_log:N { c }
9341 \cs_new_protected:Npn \_prop_show:NN #1#2
9342   {
9343     \_kernel_chk_defined:NT #2
9344     {
9345       #1 { LaTeX/kernel } { show-prop }
9346       { \token_to_str:N #2 }
9347       { \prop_map_function:NN #2 \msg_show_item:nn }
9348       { } { }
9349     }
9350   }
```

(End definition for `\prop_show:N` and `\prop_log:N`. These functions are documented on page 136.)

9351 `</initex | package>`

## 17 l3msg implementation

```
9352 <*initex | package>
9353 <@=msg>
```

`\l__msg_tmp_tl` A general scratch for the module.

```
9354 \tl_new:N \l__msg_tmp_tl
```

(End definition for `\l__msg_tmp_tl`.)

`\l__msg_name_str` Used to save module info when creating messages.

`\l__msg_text_str`

```
9355 \str_new:N \l__msg_name_str
9356 \str_new:N \l__msg_text_str
```

(End definition for `\l__msg_name_str` and `\l__msg_text_str`.)

## 17.1 Creating messages

Messages are created and used separately, so there are 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.

```
9357 \tl_const:Nn \c__msg_text_prefix_tl { msg-text->~ }
9358 \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 Test whether the control sequence containing the message text exists or not.

```
9359 \prg_new_conditional:Npnm \msg_if_exist:nn #1#2 { p , T , F , TF }
9360 {
  9361   \cs_if_exist:cTF { \c__msg_text_prefix_tl #1 / #2 }
  9362   { \prg_return_true: } { \prg_return_false: }
  9363 }
```

(End definition for \msg\_if\_exist:nnTF. This function is documented on page 139.)

\\_\_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.

```
9364 \__kernel_patch:nnNpn { }
9365 {
  9366   \__kernel_debug_log:x
   { Defining-message~ #1 / #2 ~\msg_line_context: }
  9368 }
9369 \cs_new_protected:Npn \__msg_chk_free:nn #1#2
9370 {
  9371   \msg_if_exist:nnT {#1} {#2}
  9372   {
    9373     \__kernel_msg_error:nnxx { kernel } { message-already-defined }
    9374     {#1} {#2}
  9375   }
  9376 }
```

(End definition for \\_\_msg\_chk\_if\_free:nn.)

\msg\_new:nnnn Setting a message simply means saving the appropriate text into two functions. A sanity check first.

```
9377 \cs_new_protected:Npn \msg_new:nnnn #1#2
9378 {
  9379   \__msg_chk_free:nn {#1} {#2}
  9380   \msg_gset:nnnn {#1} {#2}
  9381 }
9382 \cs_new_protected:Npn \msg_new:nnn #1#2#3
9383 { \msg_new:nnnn {#1} {#2} {#3} { } }
9384 \cs_new_protected:Npn \msg_set:nnnn #1#2#3#4
9385 {
  9386   \cs_set:cpn { \c__msg_text_prefix_tl #1 / #2 }
  9387   ##1##2##3##4 {#3}
  9388   \cs_set:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
  9389   ##1##2##3##4 {#4}
  9390 }
```

```

9391 \cs_new_protected:Npn \msg_set:n {#1} {#2} {#3} { }
9392   { \msg_set:nnnn {#1} {#2} {#3} { } }
9393 \cs_new_protected:Npn \msg_gset:nnnn #1#2#3#4
9394   {
9395     \cs_gset:cpn { \c__msg_text_prefix_tl #1 / #2 }
9396     ##1##2##3##4 {#3}
9397     \cs_gset:cpn { \c__msg_more_text_prefix_tl #1 / #2 }
9398     ##1##2##3##4 {#4}
9399   }
9400 \cs_new_protected:Npn \msg_gset:nnn #1#2#3
9401   { \msg_gset:nnnn {#1} {#2} {#3} { } }

```

(End definition for `\msg_new:nnnn` and others. These functions are documented on page 138.)

## 17.2 Messages: support functions and text

Simple pieces of text for messages.

```

\c__msg_coding_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

9402 \tl_const:Nn \c__msg_coding_error_text_tl
9403   {
9404     This-is-a-coding-error.
9405     \\ \\
9406   }
9407 \tl_const:Nn \c__msg_continue_text_tl
9408   { Type-<return>-to-continue }
9409 \tl_const:Nn \c__msg_critical_text_tl
9410   { Reading-the-current-file-'g_file_curr_name_str'-will-stop. }
9411 \tl_const:Nn \c__msg_fatal_text_tl
9412   { This-is-a-fatal-error:-LaTeX-will-abort. }
9413 \tl_const:Nn \c__msg_help_text_tl
9414   { For-immediate-help-type-H-<return> }
9415 \tl_const:Nn \c__msg_no_info_text_tl
9416   {
9417     LaTeX-does-not-know-anything-more-about-this-error,-sorry.
9418     \c__msg_return_text_tl
9419   }
9420 \tl_const:Nn \c__msg_on_line_text_tl { on-line }
9421 \tl_const:Nn \c__msg_return_text_tl
9422   {
9423     \\ \\
9424     Try-typing-<return>-to-proceed.
9425     \\
9426     If-that-doesn't-work,-type-X-<return>-to-quit.
9427   }
9428 \tl_const:Nn \c__msg_trouble_text_tl
9429   {
9430     \\ \\
9431     More-errors-will-almost-certainly-follow: \\
9432     the-LaTeX-run-should-be-aborted.
9433   }

```

(End definition for `\c__msg_coding_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.

```

9434 \cs_new:Npn \msg_line_number: { \int_use:N \tex_inlineno:D }
9435 \cs_gset:Npn \msg_line_context:
9436 {
9437     \c__msg_on_line_text_tl
9438     \c_space_tl
9439     \msg_line_number:
9440 }

```

(End definition for `\msg_line_number:` and `\msg_line_context:`. These functions are documented on page 139.)

### 17.3 Showing messages: low level mechanism

`\__msg_interrupt: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`.

```

9441 \cs_new_protected:Npn \__msg_interrupt:Nnnn #1#2#3#4
9442 {
9443     \str_set:Nx \l__msg_text_str { #1 {#2} }
9444     \str_set:Nx \l__msg_name_str { \msg_module_name:n {#2} }
9445     \tl_if_empty:nTF {#4}
9446     {
9447         \__msg_interrupt_wrap:nnn {#3}
9448         { \c__msg_continue_text_tl }
9449         { \c__msg_no_info_text_tl }
9450     }
9451     {
9452         \__msg_interrupt_wrap:nnn {#3}
9453         { \c__msg_help_text_tl }
9454         {#4}
9455     }
9456 }

```

(End definition for `\__msg_interrupt:Nnnn`.)

`\__msg_interrupt_wrap:nnn`

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.

```

9457 \cs_new_protected:Npn \__msg_interrupt_wrap:nnn #1#2#3
9458 {
9459     \iow_wrap:nnN { \\ #3 } { } { } \__msg_interrupt_more_text:n
9460     \group_begin:
9461     \int_sub:Nn \l_iow_line_count_int { 2 }
9462     \iow_wrap:nxnN { \l__msg_text_str : ~ #1 }
9463     {
9464         ( \l__msg_name_str )
9465         \prg_replicate:nn
9466         {
9467             \str_count:N \l__msg_text_str

```

```

9468     - \str_count:N \l__msg_name_str
9469     + 2
9470   }
9471   { ~ }
9472 }
9473 { } \__msg_interrupt_text:n
9474 \iow_wrap:nnnN { \l__msg_tmp_tl \\ \\ #2 } { } { }
9475   \__msg_interrupt:n
9476 }
9477 \cs_new_protected:Npn \__msg_interrupt_text:n #1
9478 {
9479   \group_end:
9480   \tl_set:Nn \l__msg_tmp_tl {#1}
9481 }
9482 \cs_new_protected:Npn \__msg_interrupt_more_text:n #1
9483 { \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.

```

9484 \group_begin:
9485   \char_set_lccode:nn { 38 } { 32 } % &
9486   \char_set_lccode:nn { 46 } { 32 } % .
9487   \char_set_lccode:nn { 123 } { 32 } % {
9488   \char_set_lccode:nn { 125 } { 32 } % }
9489   \char_set_catcode_active:N \&
9490 \tex_lowercase:D
9491 {
9492   \group_end:
9493 \cs_new_protected:Npn \__msg_interrupt:n #1
9494 {
9495   \iow_term:n { }
9496   \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^J }
9497   {
9498     \__kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9499   {

```

```

9500   \group_begin:
9501     \cs_set_protected:Npn &
9502     {
9503       \tex_errmessage:D
9504       {
9505         #1
9506         \use_none:n
9507         { ..... }
9508       }
9509     }
9510     \exp_after:wN
9511     \group_end:
9512     &
9513   }
9514 }
9515 }
9516 }
```

(End definition for `\_msg_interrupt:n`.)

## 17.4 Displaying messages

LATEX is handling error messages and so the TEX ones are disabled. This is already done by the LATEX 2 <sub>$\varepsilon$</sub>  kernel, so to avoid messing up any deliberate change by a user this is only set in format mode.

```

9517 <*initex>
9518 \int_gset:Nn \tex_errorcontextlines:D { -1 }
9519 </initex>
```

<code>\msg_fatal_text:n</code> <code>\msg_critical_text:n</code> <code>\msg_error_text:n</code> <code>\msg_warning_text:n</code> <code>\msg_info_text:n</code> <code>\_msg_text:nn</code> <code>\_msg_text:n</code>	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.  <code>\cs_new:Npn \msg_fatal_text:n #1</code> <code>9521 {</code> <code>9522   Fatal ~</code> <code>9523   \msg_error_text:n {#1}</code> <code>9524 }</code> <code>9525 \cs_new:Npn \msg_critical_text:n #1</code> <code>9526 {</code> <code>9527   Critical ~</code> <code>9528   \msg_error_text:n {#1}</code> <code>9529 }</code> <code>9530 \cs_new:Npn \msg_error_text:n #1</code> <code>9531 { \_msg_text:nn {#1} { Error } }</code> <code>9532 \cs_new:Npn \msg_warning_text:n #1</code> <code>9533 { \_msg_text:nn {#1} { Warning } }</code> <code>9534 \cs_new:Npn \msg_info_text:n #1</code> <code>9535 { \_msg_text:nn {#1} { Info } }</code> <code>9536 \cs_new:Npn \_msg_text:nn #1#2</code> <code>9537 {</code> <code>9538   \exp_args:Nf \_msg_text:n { \msg_module_type:n {#1} }</code> <code>9539   \msg_module_name:n {#1} ~</code> <code>9540 #2</code>
---	---

```

9541     }
9542 \cs_new:Npn \__msg_text:n #1
9543 {
9544     \tl_if_blank:nF {#1}
9545     { #1 ~ }
9546 }
```

(End definition for `\msg_fatal_text:n` and others. These functions are documented on page 139.)

`\g_msg_module_name_prop`  
`\g_msg_module_type_prop`

For storing public module information: the kernel data is set up in advance.

```

9547 \prop_new:N \g_msg_module_name_prop
9548 \prop_gput:Nnn \g_msg_module_name_prop { LaTeX } { LaTeX3 }
9549 \prop_new:N \g_msg_module_type_prop
9550 \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 140.)

`\msg_module_type:n` Contextual footer information, with the potential to give modules an alternative name.

```

9551 \cs_new:Npn \msg_module_type:n #1
9552 {
9553     \prop_if_in:NnTF \g_msg_module_type_prop {#1}
9554     { \prop_item:Nn \g_msg_module_type_prop {#1} }
9555     {*initex}
9556     { Module }
9557     {/initex}
9558     {*package}
9559     { Package }
9560     {/package}
9561 }
```

(End definition for `\msg_module_type:n`. This function is documented on page 140.)

`\msg_module_name:n` Contextual footer information, with the potential to give modules an alternative name.

`\msg_see_documentation_text:n`

```

9562 \cs_new:Npn \msg_module_name:n #1
9563 {
9564     \prop_if_in:NnTF \g_msg_module_name_prop {#1}
9565     { \prop_item:Nn \g_msg_module_name_prop {#1} }
9566     {#1}
9567 }
9568 \cs_new:Npn \msg_see_documentation_text:n #1
9569 {
9570     See-the- \msg_module_name:n {#1} ~
9571     documentation-for-further-information.
9572 }
```

(End definition for `\msg_module_name:n` and `\msg_see_documentation_text:n`. These functions are documented on page 140.)

`\__msg_class_new:nn`

```

9573 \group_begin:
9574     \cs_set_protected:Npn \__msg_class_new:nn #1#2
9575     {
9576         \prop_new:c { l__msg_redirect_ #1 _prop }
9577         \cs_new_protected:cpx { __msg_ #1 _code:nnnnnn }
```

```

9578      ##1##2##3##4##5##6 {#2}
9579      \cs_new_protected:cpn { msg_ #1 :nnnnnn } ##1##2##3##4##5##6
9580      {
9581          \use:x
9582          {
9583              \exp_not:n { \__msg_use:nnnnnn {#1} {##1} {##2} }
9584              { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9585              { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9586          }
9587      }
9588      \cs_new_protected:cpx { msg_ #1 :nnnnn } ##1##2##3##4##5
9589          { \exp_not:c { msg_ #1 :nnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9590      \cs_new_protected:cpx { msg_ #1 :nnnn } ##1##2##3##4
9591          { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} {##4} { } { } }
9592      \cs_new_protected:cpx { msg_ #1 :nnn } ##1##2##3
9593          { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} {##3} { } { } { } }
9594      \cs_new_protected:cpx { msg_ #1 :nn } ##1##2
9595          { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} { } { } { } { } }
9596      \cs_new_protected:cpx { msg_ #1 :nnxxxx } ##1##2##3##4##5##6
9597          {
9598              \use:x
9599              {
9600                  \exp_not:N \exp_not:n
9601                      { \exp_not:c { msg_ #1 :nnnnnn } {##1} {##2} }
9602                      {##3} {##4} {##5} {##6}
9603              }
9604          }
9605      \cs_new_protected:cpx { msg_ #1 :nnxxx } ##1##2##3##4##5
9606          { \exp_not:c { msg_ #1 :nnxxx } {##1} {##2} {##3} {##4} {##5} { } }
9607      \cs_new_protected:cpx { msg_ #1 :nnxx } ##1##2##3##4
9608          { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} {##4} { } { } }
9609      \cs_new_protected:cpx { msg_ #1 :nnx } ##1##2##3
9610          { \exp_not:c { msg_ #1 :nnxxxx } {##1} {##2} {##3} { } { } { } }
9611      }

```

(End definition for `\__msg_class_new:nn`.)

`\msg_fatal:nnnnnn`

For fatal errors, after the error message TeX bails out.

```

9612      \__msg_class_new:nn { fatal }
9613      {
9614          \__msg_interrupt:Nnnn
9615              \msg_fatal_text:n
9616              {#1}
9617              { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9618              { \c__msg_fatal_text_tl }
9619              \tex_end:D
9620      }

```

(End definition for `\msg_fatal:nnnnnn` and others. These functions are documented on page 141.)

`\msg_critical:nnnnnn`

Not quite so bad: just end the current file.

```

9621      \__msg_class_new:nn { critical }
9622      {
9623          \__msg_interrupt:Nnnn
9624              \msg_critical_text:n

```

```

9625     {#1}
9626     { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9627     { \c__msg_critical_text_tl }
9628     \tex_endinput:D
9629 }
```

(End definition for `\msg_critical:nnnnnn` and others. These functions are documented on page 141.)

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

```

9630     \__msg_class_new:nn { error }
9631     {
9632         \__msg_error:cnnnnn
9633         { \c__msg_more_text_prefix_tl #1 / #2 }
9634         {#3} {#4} {#5} {#6}
9635         {
9636             \__msg_interrupt:Nnnn
9637             \msg_error_text:n
9638             {#1}
9639             { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9640         }
9641     }
9642     \cs_new_protected:Npn \__msg_error:cnnnnn #1#2#3#4#5#6
9643     {
9644         \cs_if_eq:cNTF {#1} \__msg_no_more_text:nnnn
9645         { #6 { } }
9646         { #6 { \use:c {#1} {#2} {#3} {#4} {#5} } }
9647     }
9648     \cs_new:Npn \__msg_no_more_text:nnnn #1#2#3#4 { }
```

(End definition for `\msg_error:nnnnnn` and others. These functions are documented on page 141.)

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

```

9649     \__msg_class_new:nn { warning }
9650     {
9651         \str_set:Nx \l__msg_text_str { \msg_warning_text:n {#1} }
9652         \str_set:Nx \l__msg_name_str { \msg_module_name:n {#1} }
9653         \iow_term:n { }
9654         \iow_wrap:nxnN
9655         {
9656             \l__msg_text_str : ~
9657             \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9658         }
9659     {
9660         ( \l__msg_name_str )
9661         \prg_replicate:nn
9662         {
9663             \str_count:N \l__msg_text_str
9664             - \str_count:N \l__msg_name_str
9665         }
9666         { ~ }
9667     }
9668     { } \iow_term:n
9669     \iow_term:n { }
970 }
```

(End definition for `\msg_warning:nnnnnn` and others. These functions are documented on page [141](#).)

`\msg_info:nnnnnn` Information only goes into the log.

```
9671     \__msg_class_new:nn { info }
9672     {
9673         \str_set:Nx \l__msg_text_str { \msg_info_text:n {#1} }
9674         \str_set:Nx \l__msg_name_str { \msg_module_name:n {#1} }
9675         \iow_log:n {}
9676         \iow_wrap:nxnN
9677         {
9678             \l__msg_text_str : ~
9679             \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6}
9680         }
9681         {
9682             ( \l__msg_name_str )
9683             \prg_replicate:nn
9684             {
9685                 \str_count:N \l__msg_text_str
9686                 - \str_count:N \l__msg_name_str
9687             }
9688             { ~ }
9689         }
9690         { } \iow_log:n
9691         \iow_log:n {}
9692     }
```

(End definition for `\msg_info:nnnnnn` and others. These functions are documented on page [142](#).)

`\msg_log:nnnnnn` “Log” data is very similar to information, but with no extras added.

```
9693     \__msg_class_new:nn { log }
9694     {
9695         \iow_wrap:nmnN
9696         { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
9697         { } { } \iow_log:n
9698     }
```

(End definition for `\msg_log:nnnnnn` and others. These functions are documented on page [142](#).)

`\msg_none:nnnnnn` The `none` message type is needed so that input can be gobbled.

```
9699     \__msg_class_new:nn { none } { }
```

(End definition for `\msg_none:nnnnnn` and others. These functions are documented on page [142](#).)

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 that 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 `>~`.

```
9700     \__msg_class_new:nn { show }
9701     {
9702         \iow_wrap:nmnN
9703         { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
```

`\__msg_show_dot:w`  
`\__msg_show:nn`

```

9704     { } { } \__msg_show:n
9705   }
9706 \cs_new_protected:Npn \__msg_show:n #1
9707   {
9708     \tl_if_in:nnTF { ^^J #1 } { ^^J > ~ }
9709     {
9710       \tl_if_in:nnTF { #1 \q_mark } { . \q_mark }
9711         { \__msg_show_dot:w } { \__msg_show:w }
9712         ^^J #1 \q_stop
9713     }
9714     { \__msg_show:nn { ? #1 } { } }
9715   }
9716 \cs_new:Npn \__msg_show_dot:w #1 ^^J > ~ #2 . \q_stop
9717   { \__msg_show:nn {#1} {#2} }
9718 \cs_new:Npn \__msg_show:w #1 ^^J > ~ #2 \q_stop
9719   { \__msg_show:nn {#1} {#2} }
9720 \cs_new_protected:Npn \__msg_show:nn #1#2
9721   {
9722     \tl_if_empty:nF {#1}
9723       { \exp_args:No \iow_term:n { \use_none:n #1 } }
9724     \tl_set:Nn \l__msg_tmp_tl {#2}
9725     \kernel_iow_with:Nnn \tex_newlinechar:D { 10 }
9726       {
9727         \kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
9728       {
9729         \tex_showtokens:D \exp_after:wN \exp_after:wN \exp_after:wN
9730           { \exp_after:wN \l__msg_tmp_tl }
9731       }
9732     }
9733   }

```

(End definition for `\msg_show:nnnnnn` and others. These functions are documented on page 248.)

End the group to eliminate `\__msg_class_new:nn`.

```
9734 \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.

```

9735 \cs_new:Npn \__msg_class_chk_exist:nT #1
9736   {
9737     \cs_if_free:cTF { __msg_ #1 _code:nnnnnn }
9738       { \__kernel_msg_error:nnx { kernel } { message-class-unknown } {#1} }
9739   }

```

(End definition for `\__msg_class_chk_exist:nT`.)

`\l__msg_class_tl` Support variables needed for the redirection system.

```

9740 \tl_new:N \l__msg_class_tl
9741 \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

```
9742 \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 {}.

```
9743 \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.

```
9744 \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.

```
9745 \cs_new_protected:Npn \_\_msg_use:nnnnnn #1#2#3#4#5#6#7
9746   {
9747     \use:c { conditionally@traceoff }
9748     \msg_if_exist:nTF {#2} {#3}
9749     {
9750       \_\_msg_class_chk_exist:nT {#1}
9751       {
9752         \tl_set:Nn \l__msg_current_class_tl {#1}
9753         \cs_set_protected:Npx \_\_msg_use_code:
9754         {
9755           \exp_not:n
9756           {
9757             \use:c { __msg_ \l__msg_class_tl _code:nnnnnn }
9758             {#2} {#3} {#4} {#5} {#6} {#7}
9759           }
9760           }
9761           \_\_msg_use_redirect_name:n { #2 / #3 }
9762         }
9763       }
9764       { \_\_kernel_msg_error:nnxx { kernel } { message-unknown } {#2} {#3} }
9765     \use:c { conditionally@traceon }
9766   }
9767 \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.

```
9768 \cs_new_protected:Npn \_\_msg_use_redirect_name:n #1
9769   {
9770     \prop_get:NnNTF \l__msg_redirect_prop { / #1 } \l__msg_class_tl
9771     { \_\_msg_use_code: }
9772     {
9773       \seq_clear:N \l__msg_hierarchy_seq
```

```

9774         \_\_msg\_use\_hierarchy:nwwN { }
9775             #1 \q\_mark \_\_msg\_use\_hierarchy:nwwN
9776                 / \q\_mark \use\_none\_delimit\_by\_q\_stop:w
9777                     \q\_stop
9778                         \_\_msg\_use\_redirect\_module:n { }
9779                     }
9780     }
9781 \cs\_new\_protected:Npn \_\_msg\_use\_hierarchy:nwwN #1#2 / #3 \q\_mark #4
9782     {
9783         \seq\_put\_left:Nn \l\_\_msg\_hierarchy\_seq {\#1}
9784             #4 { #1 / #2 } #3 \q\_mark #4
9785     }

```

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`.

```

9786 \cs\_new\_protected:Npn \_\_msg\_use\_redirect\_module:n #1
9787     {
9788         \seq\_map\_inline:Nn \l\_\_msg\_hierarchy\_seq
9789             {
9790                 \prop\_get:cnNTF { l\_\_msg\_redirect\_ \l\_\_msg\_current\_class\_tl _prop }
9791                     {\#\#1} \l\_\_msg\_class\_tl
9792                     {
9793                         \seq\_map\_break:n
9794                         {
9795                             \tl\_if\_eq:NNTF \l\_\_msg\_current\_class\_tl \l\_\_msg\_class\_tl
9796                             { \_\_msg\_use\_code: }
9797                             {
9798                                 \tl\_set\_eq:NN \l\_\_msg\_current\_class\_tl \l\_\_msg\_class\_tl
9799                                     \_\_msg\_use\_redirect\_module:n {\#\#1}
9800                             }
9801                         }
9802                     }
9803             {
9804                 \str\_if\_eq:nnT {\#\#1} {\#1}
9805                 {
9806                     \tl\_set\_eq:NN \l\_\_msg\_class\_tl \l\_\_msg\_current\_class\_tl
9807                     \seq\_map\_break:n { \_\_msg\_use\_code: }
9808                 }
9809             }
9810         }
9811     }

```

(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.

```

9812 \cs_new_protected:Npn \msg_redirect_name:nnn #1#2#3
9813 {
9814     \tl_if_empty:nTF {#3}
9815         { \prop_remove:Nn \l__msg_redirect_prop { / #1 / #2 } }
9816     {
9817         \__msg_class_chk_exist:nT {#3}
9818             { \prop_put:Nnm \l__msg_redirect_prop { / #1 / #2 } {#3} }
9819     }
9820 }

```

(End definition for `\msg_redirect_name:nnn`. This function is documented on page 143.)

```

\msg_redirect_class:nn
\msg_redirect_module:nnn
\__msg_redirect:nnn

```

```

\__msg_redirect_loop_chk:nnn
\__msg_redirect_loop_list:n

```

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`.

```

9821 \cs_new_protected:Npn \msg_redirect_class:nn
9822     { \__msg_redirect:nnn { } }
9823 \cs_new_protected:Npn \msg_redirect_module:nnn #1
9824     { \__msg_redirect:nnn { / #1 } }
9825 \cs_new_protected:Npn \__msg_redirect:nnn #1#2#3
9826     {
9827         \__msg_class_chk_exist:nT {#2}
9828             {
9829                 \tl_if_empty:nTF {#3}
9830                     { \prop_remove:cn { l__msg_redirect_ #2 _prop } {#1} }
9831                 {
9832                     \__msg_class_chk_exist:nT {#3}
9833                         {
9834                             \prop_put:cnn { l__msg_redirect_ #2 _prop } {#1} {#3}
9835                             \tl_set:Nn \l__msg_current_class_tl {#2}
9836                             \seq_clear:N \l__msg_class_loop_seq
9837                             \__msg_redirect_loop_chk:nnn {#2} {#3} {#1}
9838                         }
9839                     }
9840                 }
9841             }

```

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.

```

9842 \cs_new_protected:Npn \__msg_redirect_loop_chk:nnn #1#2#3
9843     {
9844         \seq_put_right:Nn \l__msg_class_loop_seq {#1}
9845         \prop_get:cnNT { l__msg_redirect_ #1 _prop } {#3} \l__msg_current_class_tl
9846             {
9847                 \str_if_eq:VnF \l__msg_current_class_tl {#1}

```

```

9848 {
9849   \tl_if_eq:NNTF \l__msg_class_tl \l__msg_current_class_tl
9850   {
9851     \prop_put:cnn { l__msg_redirect_ #2 _prop } {#3} {#2}
9852     \__kernel_msg_warning:nxxxxx
9853     { kernel } { message-redirect-loop }
9854     { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9855     { \seq_item:Nn \l__msg_class_loop_seq { 2 } }
9856     {#3}
9857     {
9858       \seq_map_function:NN \l__msg_class_loop_seq
9859         \__msg_redirect_loop_list:n
9860         { \seq_item:Nn \l__msg_class_loop_seq { 1 } }
9861       }
9862     }
9863     { \__msg_redirect_loop_chk:onn \l__msg_class_tl {#2} {#3} }
9864   }
9865 }
9866 }
9867 \cs_generate_variant:Nn \__msg_redirect_loop_chk:nnn { o }
9868 \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 143.)

## 17.5 Kernel-specific functions

```

\__kernel_msg_new:nnnn
\__kernel_msg_new:nnn
\__kernel_msg_set:nnnn
\__kernel_msg_set:nnn

```

The kernel needs some messages of its own. These are created using pre-built functions. Two functions are provided: one more general and one which only has the short text part.

```

9869 \cs_new_protected:Npn \__kernel_msg_new:nnnn #1#2
9870   { \msg_new:nnnn { LaTeX } { #1 / #2 } }
9871 \cs_new_protected:Npn \__kernel_msg_new:nnn #1#2
9872   { \msg_new:nnn { LaTeX } { #1 / #2 } }
9873 \cs_new_protected:Npn \__kernel_msg_set:nnnn #1#2
9874   { \msg_set:nnnn { LaTeX } { #1 / #2 } }
9875 \cs_new_protected:Npn \__kernel_msg_set:nnn #1#2
9876   { \msg_set:nnn { LaTeX } { #1 / #2 } }

```

(End definition for `\__kernel_msg_new:nnnn` and others.)

```

\__msg_kernel_class_new:nN
\__msg_kernel_class_new_aux:nN

```

All the functions for kernel messages come in variants ranging from 0 to 4 arguments. Those with less than 4 arguments are defined in terms of the 4-argument variant, in a way very similar to `\msg_class_new:nn`. This auxiliary is destroyed at the end of the group.

```

9877 \group_begin:
9878   \cs_set_protected:Npn \__msg_kernel_class_new:nN #1
9879     { \__msg_kernel_class_new_aux:nN { __kernel_msg_ #1 } }
9880   \cs_set_protected:Npn \__msg_kernel_class_new_aux:nN #1#2
9881   {
9882     \cs_new_protected:cpn { #1 :nnnnnn } ##1##2##3##4##5##6
9883     {
9884       \use:x
9885       {
9886         \exp_not:n { #2 { LaTeX } { ##1 / ##2 } }

```

```

9887      { \tl_to_str:n {##3} } { \tl_to_str:n {##4} }
9888      { \tl_to_str:n {##5} } { \tl_to_str:n {##6} }
9889    }
9890  }
9891 \cs_new_protected:cp { #1 :nnnnn } ##1##2##3##4##5
9892   { \exp_not:c { #1 :nnnnn } {##1} {##2} {##3} {##4} {##5} { } }
9893 \cs_new_protected:cp { #1 :nnnn } ##1##2##3##4
9894   { \exp_not:c { #1 :nnnnn } {##1} {##2} {##3} {##4} { } { } }
9895 \cs_new_protected:cp { #1 :nnn } ##1##2##3
9896   { \exp_not:c { #1 :nnnnn } {##1} {##2} {##3} { } { } { } }
9897 \cs_new_protected:cp { #1 :nn } ##1##2
9898   { \exp_not:c { #1 :nnnnn } {##1} {##2} { } { } { } { } }
9899 \cs_new_protected:cp { #1 :nxxxxx } ##1##2##3##4##5##6
9900   {
9901     \use:x
9902   {
9903     \exp_not:N \exp_not:n
9904       { \exp_not:c { #1 :nnnnn } {##1} {##2} }
9905       {##3} {##4} {##5} {##6}
9906   }
9907 }
9908 \cs_new_protected:cp { #1 :nxxxx } ##1##2##3##4##5
9909   { \exp_not:c { #1 :nxxxx } {##1} {##2} {##3} {##4} {##5} { } }
9910 \cs_new_protected:cp { #1 :nnxx } ##1##2##3##4
9911   { \exp_not:c { #1 :nxxxx } {##1} {##2} {##3} {##4} { } { } }
9912 \cs_new_protected:cp { #1 :nx } ##1##2##3
9913   { \exp_not:c { #1 :nxxxx } {##1} {##2} {##3} { } { } { } }
9914 }

```

(End definition for `\_msg_kernel_class_new:nN` and `\_msg_kernel_class_new_aux:nN`.)

`\_kernel_msg_fatal:nnnnn`  
`\_kernel_msg_fatal:nnxxx`  
`\_kernel_msg_fatal:nnnn`  
`\_kernel_msg_fatal:nnxx`  
`\_kernel_msg_fatal:nnn`  
`\_kernel_msg_fatal:nnx`  
`\_kernel_msg_fatal:nn`  
`\_kernel_msg_error:nnnnn`

Neither fatal kernel errors nor kernel errors can be redirected. We directly use the code for (non-kernel) fatal errors and errors, adding the “`LATEX`” module name. Three functions are already defined by `I3basics`; we need to undefine them to avoid errors.

```

9915   \_msg_kernel_class_new:nN { fatal } \_msg_fatal_code:nnnnn
9916   \cs_undefine:N \_kernel_msg_error:nnxx
9917   \cs_undefine:N \_kernel_msg_error:nnx
9918   \cs_undefine:N \_kernel_msg_error:nn
9919   \_msg_kernel_class_new:nN { error } \_msg_error_code:nnnnnn

```

(End definition for `\_kernel_msg_fatal:nnnnn` and others.)

`\_kernel_msg_warning:nnnnn`  
`\_kernel_msg_warning:nnxxx`  
`\_kernel_msg_warning:nnnn`  
`\_kernel_msg_warning:nnxx`  
`\_kernel_msg_warning:nnn`  
`\_kernel_msg_warning:nnx`  
`\_kernel_msg_warning:nn`  
`\_kernel_msg_info:nnnnn`  
`\_kernel_msg_info:nnxxx`  
`\_kernel_msg_info:nnnnn`  
`\_kernel_msg_info:nnxx`  
`\_kernel_msg_info:nnn`  
`\_kernel_msg_info:nnx`  
`\_kernel_msg_info:nn`

Kernel messages which can be redirected simply use the machinery for normal messages, with the module name “`LATEX`”.

```

9920   \_msg_kernel_class_new:nN { warning } \msg_warning:nnxxx
9921   \_msg_kernel_class_new:nN { info } \msg_info:nnxxx

```

(End definition for `\_kernel_msg_warning:nnnnn` and others.)

End the group to eliminate `\_msg_kernel_class_new:nN`.

```
9922 \group_end:
```

Error messages needed to actually implement the message system itself.

```

9923 \_kernel_msg_new:nnnn { kernel } { message-already-defined }
9924   { Message~'#2'~for~module~'#1'~already~defined. }
9925   {

```

```

9926     \c__msg_coding_error_text_tl
9927     LaTeX~was~asked~to~define~a~new~message~called~'#2'\\
9928     by~the~module~'#1':~this~message~already~exists.
9929     \c__msg_return_text_tl
9930   }
9931 \__kernel_msg_new:nnn { kernel } { message-unknown }
9932   { Unknown~message~'#2'~for~module~'#1'. }
9933   {
9934     \c__msg_coding_error_text_tl
9935     LaTeX~was~asked~to~display~a~message~called~'#2'\\
9936     by~the~module~'#1':~this~message~does~not~exist.
9937     \c__msg_return_text_tl
9938   }
9939 \__kernel_msg_new:nnn { kernel } { message-class-unknown }
9940   { Unknown~message~class~'#1'. }
9941   {
9942     LaTeX~has~been~asked~to~redirect~messages~to~a~class~'#1':\\
9943     this~was~never~defined.
9944     \c__msg_return_text_tl
9945   }
9946 \__kernel_msg_new:nnn { kernel } { message-redirect-loop }
9947   {
9948     Message~redirection~loop~caused~by~ {#1} ~=>~ {#2}
9949     \tl_if_empty:nF {#3} { ~for~module~' \use_none:n #3 ' } .
9950   }
9951   {
9952     Adding~the~message~redirection~ {#1} ~=>~ {#2}
9953     \tl_if_empty:nF {#3} { ~for~the~module~' \use_none:n #3 ' } ~
9954     created~an~infinite~loop\\\\\
9955     \iow_indent:n { #4 \\\\ }
9956   }

```

Messages for earlier kernel modules plus a few for l3keys which cover coding errors.

```

9957 \__kernel_msg_new:nnn { kernel } { bad-number-of-arguments }
9958   { Function~'#1'~cannot~be~defined~with~#2~arguments. }
9959   {
9960     \c__msg_coding_error_text_tl
9961     LaTeX~has~been~asked~to~define~a~function~'#1'~with~
9962     #2~arguments.~
9963     TeX~allows~between~0~and~9~arguments~for~a~single~function.
9964   }
9965 \__kernel_msg_new:nnn { kernel } { char-active }
9966   { Cannot~generate~active~chars. }
9967 \__kernel_msg_new:nnn { kernel } { char-invalid-catcode }
9968   { Invalid~catcode~for~char~generation. }
9969 \__kernel_msg_new:nnn { kernel } { char-null-space }
9970   { Cannot~generate~null~char~as~a~space. }
9971 \__kernel_msg_new:nnn { kernel } { char-out-of-range }
9972   { Charcode~requested~out~of~engine~range. }
9973 \__kernel_msg_new:nnn { kernel } { char-space }
9974   { Cannot~generate~space~chars. }
9975 \__kernel_msg_new:nnn { kernel } { command-already-defined }
9976   { Control~sequence~'#1'~already~defined. }
9977   {
9978     \c__msg_coding_error_text_tl

```

```

9979      LaTeX~has~been~asked~to~create~a~new~control~sequence~'#1'~
9980      but~this~name~has~already~been~used~elsewhere. \\ \\
9981      The~current~meaning~is:\\
9982      \\ #2
9983  }
9984 \_\_kernel\_msg\_new:nnnn { kernel } { command-not-defined }
9985   { Control-sequence~#1~undefined. }
9986   {
9987     \c\_msg\_coding\_error\_text_tl
9988     LaTeX~has~been~asked~to~use~a~control~sequence~'#1':\\
9989     this~has~not~been~defined~yet.
9990   }
9991 \_\_kernel\_msg\_new:nnn { kernel } { deprecated-command }
9992   {
9993     The~deprecated~command~'#2'~has~been~or~will~be~removed~on~#1.
9994     \tl_if_empty:nF {#3} { ~Use~instead~'#3'. }
9995   }
9996 \_\_kernel\_msg\_new:nnnn { kernel } { empty-search-pattern }
9997   { Empty~search~pattern. }
9998   {
9999     \c\_msg\_coding\_error\_text_tl
10000    LaTeX~has~been~asked~to~replace~an~empty~pattern~by~'#1':~that~
10001    would~lead~to~an~infinite~loop!
10002  }
10003 \_\_kernel\_msg\_new:nnnn { kernel } { out-of-registers }
10004   { No~room~for~a~new~#1. }
10005   {
10006     TeX~only~supports~\int_use:N \c_max_register_int \ %
10007     of~each~type.~All~the~#1~registers~have~been~used.~
10008     This~run~will~be~aborted~now.
10009   }
10010 \_\_kernel\_msg\_new:nnnn { kernel } { non-base-function }
10011   { Function~'#1'~is~not~a~base~function }
10012   {
10013     \c\_msg\_coding\_error\_text_tl
10014     Functions~defined~through~\iow_char:N\cs_new:Nn~must~have~
10015     a~signature~consisting~of~only~normal~arguments~'N'~and~'n'.~
10016     To~define~variants~use~\iow_char:N\cs_generate_variant:Nn~
10017     and~to~define~other~functions~use~\iow_char:N\cs_new:Npn.
10018   }
10019 \_\_kernel\_msg\_new:nnnn { kernel } { missing-colon }
10020   { Function~'#1'~contains~no~':'. }
10021   {
10022     \c\_msg\_coding\_error\_text_tl
10023     Code-level~functions~must~contain~':~to~separate~the~
10024     argument~specification~from~the~function~name.~This~is~
10025     needed~when~defining~conditionals~or~variants,~or~when~building~a~
10026     parameter~text~from~the~number~of~arguments~of~the~function.
10027   }
10028 \_\_kernel\_msg\_new:nnnn { kernel } { overflow }
10029   { Integers~larger~than~2^{30}-1~cannot~be~stored~in~arrays. }
10030   {
10031     An~attempt~was~made~to~store~#3~
10032     \tl_if_empty:nF {#2} { at~position~#2~ } in~the~array~'#1'~.

```

```

10033     The~largest~allowed~value~#4~will~be~used~instead.
10034   }
10035 \__kernel_msg_new:nnnn { kernel } { out-of-bounds }
10036   { Access~to~an~entry~beyond~an~array's~bounds. }
10037   {
10038     An~attempt~was~made~to~access~or~store~data~at~position~#2~of~the~
10039     array~'#1',~but~this~array~has~entries~at~positions~from~1~to~#3.
10040   }
10041 \__kernel_msg_new:nnnn { kernel } { protected-predicate }
10042   { Predicate~'#1'~must~be~expandable. }
10043   {
10044     \c__msg_coding_error_text_tl
10045     LaTeX~has~been~asked~to~define~'#1'~as~a~protected~predicate.~
10046     Only~expandable~tests~can~have~a~predicate~version.
10047   }
10048 \__kernel_msg_new:nnn { kernel } { randint-backward-range }
10049   { Bounds~ordered~backwards~in~\int_rand:nn {#1}~{#2}. }
10050 \__kernel_msg_new:nnnn { kernel } { conditional-form-unknown }
10051   { Conditional-form~'#1'~for~function~'#2'~unknown. }
10052   {
10053     \c__msg_coding_error_text_tl
10054     LaTeX~has~been~asked~to~define~the~conditional~form~'#1'~of~
10055     the~function~'#2',~but~only~'TF',~'T',~'F',~and~'p'~forms~exist.
10056   }
10057 \__kernel_msg_new:nnnn { kernel } { key-no-property }
10058   { No~property~given~in~definition~of~key~'#1'. }
10059   {
10060     \c__msg_coding_error_text_tl
10061     Inside~\keys_define:nn each~key~name~
10062     needs~a~property: \\ \\
10063     \iow_indent:n { #1 .<property>} \\ \\
10064     LaTeX~did~not~find~a~'.~to~indicate~the~start~of~a~property.
10065   }
10066 \__kernel_msg_new:nnnn { kernel } { key-property-boolean-values-only }
10067   { The~property~'#1'~accepts~boolean~values~only. }
10068   {
10069     \c__msg_coding_error_text_tl
10070     The~property~'#1'~only~accepts~the~values~'true'~and~'false'.
10071   }
10072 \__kernel_msg_new:nnnn { kernel } { key-property-requires-value }
10073   { The~property~'#1'~requires~a~value. }
10074   {
10075     \c__msg_coding_error_text_tl
10076     LaTeX~was~asked~to~set~property~'#1'~for~key~'#2'.\\
10077     No~value~was~given~for~the~property,~and~one~is~required.
10078   }
10079 \__kernel_msg_new:nnnn { kernel } { key-property-unknown }
10080   { The~key~property~'#1'~is~unknown. }
10081   {
10082     \c__msg_coding_error_text_tl
10083     LaTeX~has~been~asked~to~set~the~property~'#1'~for~key~'#2':~
10084     this~property~is~not~defined.
10085   }
10086 \__kernel_msg_new:nnnn { kernel } { scanmark-already-defined }

```

```

10087 { Scan~mark~#1~already~defined. }
10088 {
10089   \c__msg_coding_error_text_tl
10090   LaTeX~has~been~asked~to~create~a~new~scan~mark~'#1'~
10091   but~this~name~has~already~been~used~for~a~scan~mark.
10092 }
10093 \_kernel_msg_new:nnnn { kernel } { variable-not-defined }
10094 { Variable~#1~undefined. }
10095 {
10096   \c__msg_coding_error_text_tl
10097   LaTeX~has~been~asked~to~show~a~variable~#1,~but~this~has~not~
10098   been~defined~yet.
10099 }
10100 \_kernel_msg_new:nnnn { kernel } { variant-too-long }
10101 { Variant~form~'#1'~longer~than~base~signature~of~'#2'. }
10102 {
10103   \c__msg_coding_error_text_tl
10104   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
10105   with~a~signature~starting~with~'#1',~but~that~is~longer~than~
10106   the~signature~(part~after~the~colon)~of~'#2'.
10107 }
10108 \_kernel_msg_new:nnnn { kernel } { invalid-variant }
10109 { Variant~form~'#1'~invalid~for~base~form~'#2'. }
10110 {
10111   \c__msg_coding_error_text_tl
10112   LaTeX~has~been~asked~to~create~a~variant~of~the~function~'#2'~
10113   with~a~signature~starting~with~'#1',~but~cannot~change~an~argument~
10114   from~type~'#3'~to~type~'#4'.
10115 }
10116 \_kernel_msg_new:nnnn { kernel } { invalid-exp-args }
10117 { Invalid~variant~specifier~'#1'~in~'#2'. }
10118 {
10119   \c__msg_coding_error_text_tl
10120   LaTeX~has~been~asked~to~create~an~\iow_char:N\\exp_args:N...~
10121   function~with~signature~'N#2'~but~'#1'~is~not~a~valid~argument~
10122   specifier.
10123 }
10124 \_kernel_msg_new:nnn { kernel } { deprecated-variant }
10125 {
10126   Variant~form~'#1'~deprecated~for~base~form~'#2'..
10127   One~should~not~change~an~argument~from~type~'#3'~to~type~'#4'
10128   \str_case:nnF {#3}
10129   {
10130     { n } { :~use~a~'\token_if_eq_charcode:NNTF #4 c v V'~variant? }
10131     { N } { :~base~form~only~accepts~a~single~token~argument. }
10132     {#4} { :~base~form~is~already~a~variant. }
10133   } { . }
10134 }

```

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.

```

10135 {*package}
10136 \_kernel_if_debug:TF

```

```

10137  {
10138  \_kernel_msg_new:nnnn { kernel } { debug }
10139  { The~debugging~option~'#1'~does~not~exist~\msg_line_context:.. }
10140  {
10141  The~functions~'\iow_char:N\debug_on:n'~and~
10142  '\iow_char:N\debug_off:n'~only~accept~the~arguments~
10143  'check-declarations',~'deprecation',~'log-functions',~not~'#1'.
10144  }
10145  \_kernel_msg_new:nnn { kernel } { expr } { '#2'~in~#1 }
10146  \_kernel_msg_new:nnnn { kernel } { local-global }
10147  { Inconsistent~local/global~assignment }
10148  {
10149  \c_msg_coding_error_text_tl
10150  \if:w 1 #2 Local \else: Global \fi: \ %
10151  assignment~to~a~
10152  \if:w 1 #1 local
10153  \else:
10154  \if:w g #1 global \else: constant \fi:
10155  \fi:
10156  \ %
10157  variable~'#3'.
10158  }
10159  \_kernel_msg_new:nnnn { kernel } { non-declared-variable }
10160  { The~variable~#1~has~not~been~declared~\msg_line_context:.. }
10161  {
10162  \c_msg_coding_error_text_tl
10163  Checking~is~active,~and~you~have~tried~do~so~something~like: \\
10164  \ \ \tl_set:Nn ~ #1 ~ \{ ~ ... ~ \} \\
10165  without~first~having: \\
10166  \ \ \tl_new:N ~ #1 \\
10167  \\
10168  LaTeX~will~create~the~variable~and~continue.
10169  }
10170  }
10171  {
10172  \_kernel_msg_new:nnnn { kernel } { enable-debug }
10173  { To~use~'#1'~load~expl3~with~the~'enable-debug'~option. }
10174  {
10175  The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
10176  some~internal~functions~in~expl3~have~been~appropriately~
10177  defined.~This~only~happens~if~one~of~the~options~
10178  'enable-debug',~'check-declarations'~or~'log-functions'~was~
10179  given~when~loading~expl3.
10180  }
10181  }
10182  </package>
10183  <*initex>
10184  \_kernel_msg_new:nnnn { kernel } { enable-debug }
10185  { '#1'~cannot~be~used~in~format~mode. }
10186  {
10187  The~function~'#1'~will~be~ignored~because~it~can~only~work~if~
10188  some~internal~functions~in~expl3~have~been~appropriately~
10189  defined.~This~only~happens~in~package~mode~(and~only~if~one~of~
the~options~'enable-debug',~'check-declarations'~or~'log-functions'~
10190

```

```

10191      was~given~when~loading~expl3.
10192      }
10193  </initex>

```

Some errors only appear in expandable settings, hence don't need a "more-text" argument.

```

10194  \_\_kernel\_msg\_new:nnn { kernel } { bad-exp-end-f }
10195      { Misused~\exp_end_continue_f:w or~:nw }
10196  \_\_kernel\_msg\_new:nnn { kernel } { bad-variable }
10197      { Erroneous~variable~#1 used! }
10198  \_\_kernel\_msg\_new:nnn { kernel } { misused-sequence }
10199      { A-sequence~was~misused. }
10200  \_\_kernel\_msg\_new:nnn { kernel } { misused-prop }
10201      { A-property-list~was~misused. }
10202  \_\_kernel\_msg\_new:nnn { kernel } { negative-replication }
10203      { Negative~argument~for~\prg_replicate:nn. }
10204  \_\_kernel\_msg\_new:nnn { kernel } { prop-keyval }
10205      { Missing/extra~'=~,~in~,~#1~,~(in~,~...~keyval:Nn') }
10206  \_\_kernel\_msg\_new:nnn { kernel } { unknown-comparison }
10207      { Relation~'#1~unknown:~use~=~,~<,~>,~==,~!=,~<=,~>=. }
10208  \_\_kernel\_msg\_new:nnn { kernel } { zero-step }
10209      { Zero~step~size~for~step~function~#1. }
10210 \cs_if_exist:NF \tex_expanded:D
10211      {
10212          \_\_kernel\_msg\_new:nnn { kernel } { e-type }
10213          { #1 ~ in-e-type~argument }
10214      }

```

Messages used by the "show" functions.

```

10215  \_\_kernel\_msg\_new:nnn { kernel } { show-clist }
10216      {
10217          The~comma~list~ \tl_if_empty:nF {#1} { #1 ~ }
10218          \tl_if_empty:nTF {#2}
10219              { is~empty \\>~ . }
10220              { contains~the~items~(without~outer~braces): #2 . }
10221      }
10222  \_\_kernel\_msg\_new:nnn { kernel } { show-intarray }
10223      { The~integer~array~#1~contains~#2~items: \\ #3 . }
10224  \_\_kernel\_msg\_new:nnn { kernel } { show-prop }
10225      {
10226          The~property~list~#1~
10227          \tl_if_empty:nTF {#2}
10228              { is~empty \\>~ . }
10229              { contains~the~pairs~(without~outer~braces): #2 . }
10230      }
10231  \_\_kernel\_msg\_new:nnn { kernel } { show-seq }
10232      {
10233          The~sequence~#1~
10234          \tl_if_empty:nTF {#2}
10235              { is~empty \\>~ . }
10236              { contains~the~items~(without~outer~braces): #2 . }
10237      }
10238  \_\_kernel\_msg\_new:nnn { kernel } { show-streams }
10239      {
10240          \tl_if_empty:nTF {#2} { No~ } { The~following~ }

```

```

10241     \str_case:nn {#1}
10242     {
10243         { ior } { input ~ }
10244         { iow } { output ~ }
10245     }
10246     streams~are~
10247     \tl_if_empty:nTF {#2} { open } { in-use: #2 . }
10248 }
```

## 17.6 Expandable errors

`\_msg_expandable_error:n` In expansion only context, we cannot use the normal means of reporting errors. Instead, `\_msg_expandable_error:w` we feed TeX an undefined control sequence, `\LaTeX3 error:`. It is thus interrupted, and shows the context, which thanks to the odd-looking `\use:n` is

```

<argument> \LaTeX3 error:
The error message.
```

In other words, TeX is processing the argument of `\use:n`, which is `\LaTeX3 error: <error message>`. Then `\_msg_expandable_error:w` cleans up. In fact, there is an extra subtlety: if the user inserts tokens for error recovery, they should be kept. Thus we also use an odd space character (with category code 7) and keep tokens until that space character, dropping everything else until `\q_stop`. The `\exp_end:` prevents losing braces around the user-inserted text if any, and stops the expansion of `\exp:w`. The group is used to prevent `\LaTeX3-error:` from being globally equal to `\scan_stop:`.

```

10249 \group_begin:
10250 \cs_set_protected:Npn \_msg_tmp:w #1#2
10251 {
10252     \cs_new:Npn \_msg_expandable_error:n ##1
10253     {
10254         \exp:w
10255         \exp_after:wN \exp_after:wN
10256         \exp_after:wN \_msg_expandable_error:w
10257         \exp_after:wN \exp_after:wN
10258         \exp_after:wN \exp_end:
10259         \use:n { #1 #2 ##1 } #2
10260     }
10261     \cs_new:Npn \_msg_expandable_error:w ##1 #2 ##2 #2 {##1}
10262 }
10263 \exp_args:Nc \_msg_tmp:w { LaTeX3-error: }
10264 { \char_generate:nn { '\ } { 7 } }
10265 \group_end:
```

(End definition for `\_msg_expandable_error:n` and `\_msg_expandable_error:w`.)

The command built from the csname `\c_msg_text_prefix_tl` LaTeX / #1 / #2 takes four arguments and builds the error text, which is fed to `\_msg_expandable_error:n`.

```

10266 \cs_new:Npn \_kernel_msg_expandable_error:nnnnn #1#2#3#4#5#6
10267 {
10268     \exp_args:Nf \_msg_expandable_error:n
10269     {
10270         \exp_args:NNc \exp_after:wN \exp_stop_f:
10271             { \c_msg_text_prefix_tl LaTeX / #1 / #2 }
```

```

10272     {#3} {#4} {#5} {#6}
10273     }
10274   }
10275 \cs_new:Npn \__kernel_msg_expandable_error:nnnnn #1#2#3#4#5
10276   {
10277     \__kernel_msg_expandable_error:nnnnnn
10278     {#1} {#2} {#3} {#4} {#5} { }
10279   }
10280 \cs_new:Npn \__kernel_msg_expandable_error:nnnn #1#2#3#4
10281   {
10282     \__kernel_msg_expandable_error:nnnnnn
10283     {#1} {#2} {#3} {#4} { } { }
10284   }
10285 \cs_new:Npn \__kernel_msg_expandable_error:nnn #1#2#3
10286   {
10287     \__kernel_msg_expandable_error:nnnnnn
10288     {#1} {#2} {#3} { } { } { }
10289   }
10290 \cs_new:Npn \__kernel_msg_expandable_error:nn #1#2
10291   {
10292     \__kernel_msg_expandable_error:nnnnnn
10293     {#1} {#2} { } { } { } { }
10294   }
10295 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnnnn { nnffff }
10296 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnn { nnfff }
10297 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnnn { nnff }
10298 \cs_generate_variant:Nn \__kernel_msg_expandable_error:nnn { nnf }

(End definition for \__kernel_msg_expandable_error:nnnnnn and others.)

```

```

\msg_log:n
\msg_term:n
10299 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \iow_log:n }
10300 \cs_new_protected:Npn \msg_log:n #1
10301   {
10302     \iow_log:n { ..... }
10303     \iow_wrap:nnnN { . ~ #1} { . ~ } { } \iow_log:n
10304     \iow_log:n { ..... }
10305   }
10306 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { \iow_term:n }
10307 \cs_new_protected:Npn \msg_term:n #1
10308   {
10309     \iow_term:n { **** }
10310     \iow_wrap:nnnN { * ~ #1} { * ~ } { } \iow_term:n
10311     \iow_term:n { **** }
10312   }

```

(End definition for \msg\_log:n and \msg\_term:n.)

```

\msg_interrupt:nnn
10313 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 } { [Defined~error~message] }
10314 \cs_new_protected:Npn \msg_interrupt:nnn #1#2#3
10315   {
10316     \tl_if_empty:nTF {#3}
10317     {
10318       \__msg_old_interrupt_wrap:nn { \\ \c__msg_no_info_text_tl }

```

```

10319      {#1 \\\\ #2 \\\\ \c__msg_continue_text_t1 }
10320  }
10321  {
10322      \_\_msg\_old\_interrupt\_wrap:nn { \\ #3 }
10323      {#1 \\\\ #2 \\\\ \c__msg_help_text_t1 }
10324  }
10325  }
10326 \cs_new_protected:Npn \_\_msg_old_interrupt_wrap:nn #1#2
10327  {
10328      \iow_wrap:nnnN {#1} { | ~ } { } \_\_msg_old_interrupt_more_text:n
10329      \iow_wrap:nnnN {#2} { ! ~ } { } \_\_msg_old_interrupt_text:n
10330  }
10331 \cs_new_protected:Npn \_\_msg_old_interrupt_more_text:n #1
10332  {
10333      \exp_args:Nx \tex_errhelp:D
10334  {
10335      | , , , , , , , , , , , , , , , , , , , , , ,
10336      #1 \iow_newline:
10337      | ..... .
10338  }
10339  }
10340 \group_begin:
10341     \char_set_lccode:nn {'\{} {'\}
10342     \char_set_lccode:nn {'\}} {'\}
10343     \char_set_lccode:nn {'\&} {'\!}
10344     \char_set_catcode_active:N \&
10345 \tex_lowercase:D
10346  {
10347      \group_end:
10348      \cs_new_protected:Npn \_\_msg_old_interrupt_text:n #1
10349  {
10350      \iow_term:x
10351  {
10352      \iow_newline:
10353      !!!!!!!!
10354      \iow_newline:
10355      !
10356  }
10357  \_\_kernel_iow_with:Nnn \tex_newlinechar:D { '\^J }
10358  {
10359      \_\_kernel_iow_with:Nnn \tex_errorcontextlines:D { -1 }
10360  {
10361      \group_begin:
10362      \cs_set_protected:Npn &
10363  {
10364      \tex_errmessage:D
10365  {
10366      #1
10367      \use_none:n
10368      { ..... }
10369  }
10370  }
10371  \exp_after:wN
10372  \group_end:

```

```

10373           &
10374       }
10375   }
10376 }
10377 }

(End definition for \msg_interrupt:nnn.)
10378 </initex | package>

```

## 18 I3file implementation

The following test files are used for this code: *m3file001*.

```
10379 <*initex | package>
```

### 18.1 Input operations

```
10380 <@=ior>
```

#### 18.1.1 Variables and constants

`\l__ior_internal_tl` Used as a short-term scratch variable.

```
10381 \tl_new:N \l__ior_internal_tl
```

(End definition for `\l__ior_internal_tl`.)

`\c_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.

```
10382 \int_const:Nn \c_term_ior { 16 }
```

(End definition for `\c_term_ior`. This variable is documented on page 151.)

`\g__ior_streams_seq` A list of the currently-available input streams to be used as a stack. In format mode, all streams (from 0 to 15) are available, while the package requests streams to L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> as they are needed (initially none are needed), so the starting point varies!

```

10383 \seq_new:N \g__ior_streams_seq
10384 <*initex>
10385 \seq_gset_split:Nnn \g__ior_streams_seq { , }
10386 { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15 }
10387 </initex>

```

(End definition for `\g__ior_streams_seq`.)

`\l__ior_stream_tl` Used to recover the raw stream number from the stack.

```
10388 \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>$\varepsilon$</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>$\varepsilon$</sub>  mechanism it holds the value of the *last* stream allocated.

```

10389 \prop_new:N \g__ior_streams_prop
10390 (*package)
10391 \int_step_inline:nnn
10392 { 0 }
10393 {
10394     \cs_if_exist:NTF \normalend
10395         { \tex_count:D 38 ~ }
10396         {
10397             \tex_count:D 16 ~ %
10398             \cs_if_exist:NT \locount { - 1 }
10399         }
10400     }
10401 {
10402     \prop_gput:Nnn \g__ior_streams_prop {#1} { Reserved-by~format }
10403 }
10404 
```

(End definition for \g\_\_ior\_streams\_prop.)

### 18.1.2 Stream management

\ior\_new:N Reserving a new stream is done by defining the name as equal to using the terminal.

```

\ior_new:c 10405 \cs_new_protected:Npn \ior_new:N #1 { \cs_new_eq:NN #1 \c_term_ior }
10406 \cs_generate_variant:Nn \ior_new:N { c }
```

(End definition for \ior\_new:N. This function is documented on page 144.)

\g\_tmpa\_ior The usual scratch space.

```

\g_tmpb_ior 10407 \ior_new:N \g_tmpa_ior
10408 \ior_new:N \g_tmpb_ior
```

(End definition for \g\_tmpa\_ior and \g\_tmpb\_ior. These variables are documented on page 151.)

\ior\_open:Nn Use the conditional version, with an error if the file is not found.

```

\ior_open:cn 10409 \cs_new_protected:Npn \ior_open:Nn #1#2
10410 { \ior_open:NnF #1 {#2} { \__kernel_file_missing:n {#2} } }
10411 \cs_generate_variant:Nn \ior_open:Nn { c }
```

(End definition for \ior\_open:Nn. This function is documented on page 144.)

\l\_\_ior\_file\_name\_str Data storage.

```

10412 \str_new:N \l__ior_file_name_str
```

(End definition for \l\_\_ior\_file\_name\_str.)

**\ior\_open:NnTF** An auxiliary searches for the file in the T<sub>E</sub>X, L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> and L<sup>A</sup>T<sub>E</sub>X3 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.

```

10413 \prg_new_protected_conditional:Npnn \ior_open:Nn #1#2 { T , F , TF }
10414 {
10415   \file_get_full_name:nN {#2} \l__ior_file_name_str
10416   \str_if_empty:NTF \l__ior_file_name_str
10417   { \prg_return_false: }
10418   {
10419     \__kernel_ior_open:No #1 \l__ior_file_name_str
10420     \prg_return_true:
10421   }
10422 }
10423 \prg_generate_conditional_variant:Nnn \ior_open:Nn { c } { T , F , TF }
```

(End definition for `\ior_open:NnTF`. This function is documented on page 145.)

**\\_\_ior\_new:N** In package mode, 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 T<sub>E</sub>X's `\newread` being `\outer`. For ConT<sub>E</sub>Xt, we have to deal with the fact that `\newread` works like our own: it actually checks before altering definition.

```

10424 (*package)
10425 \exp_args:NNf \cs_new_protected:Npn \__ior_new:N
10426   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newread } }
10427 \cs_if_exist:NT \normalend
10428   {
10429     \cs_new_eq:NN \__ior_new_aux:N \__ior_new:N
10430     \cs_set_protected:Npn \__ior_new:N #1
10431     {
10432       \cs_undefine:N #1
10433       \__ior_new_aux:N #1
10434     }
10435   }
10436 
```

(End definition for `\__ior_new:N`.)

**\\_\_kernel\_ior\_open:Nn** The stream allocation itself uses the fact that there is a list of all of those available, so allocation is simply a question of using the number at the top of the list. In package mode, 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 T<sub>E</sub>X or L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> for a new stream and use that number (after a bit of conversion).

```

10437 \cs_new_protected:Npn \__kernel_ior_open:Nn #1#2
10438   {
10439     \ior_close:N #1
10440     \seq_gpop:NNTF \g__ior_streams_seq \l__ior_stream_tl
10441     { \__ior_open_stream:Nn #1 {#2} }
10442 (*initex)
10443   { \__kernel_msg_fatal:nn { kernel } { input-streams-exhausted } }
10444 
```

```

10445  {*package}
10446    {
10447      \__ior_new:N #1
10448      \tl_set:Nx \l__ior_stream_tl { \int_eval:n {#1} }
10449      \__ior_open_stream:Nn #1 {#2}
10450    }
10451  
```

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

`\cs_new_protected:Npn \_ior_open_stream:Nn #1#2`

{

`\tex_global:D \tex_chardef:D #1 = \l__ior_stream_tl \scan_stop:`

`\prop_gput:NVn \g__ior_streams_prop #1 {#2}`

`\tex_openin:D #1 #2 \scan_stop:`

}

(End definition for `\_kernel_iор_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.

```

10460 \cs_new_protected:Npn \ior_close:N #1
10461  {
10462    \int_compare:nT { -1 < #1 < \c_term_ior }
10463    {
10464      \tex_closein:D #1
10465      \prop_gremove:NV \g__ior_streams_prop #1
10466      \seq_if_in:NVF \g__ior_streams_seq #1
10467      { \seq_gpush:NV \g__ior_streams_seq #1 }
10468      \cs_gset_eq:NN #1 \c_term_ior
10469    }
10470  }
10471 \cs_generate_variant:Nn \ior_close:N { c }

```

(End definition for `\ior_close:N`. This function is documented on page 145.)

**\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

**\ior\_log\_list:** `\_ior_list:N` 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.

```

10472 \cs_new_protected:Npn \ior_show_list: { \_ior_list:N \msg_show:nnxxxx }
10473 \cs_new_protected:Npn \ior_log_list: { \_ior_list:N \msg_log:nnxxxx }
10474 \cs_new_protected:Npn \_ior_list:N #1
10475  {
10476    #1 { LaTeX / kernel } { show-streams }
10477    { ior }
10478    {
10479      \prop_map_function:NN \g__ior_streams_prop
10480      \msg_show_item_unbraced:nn
10481    }
10482    { } { }
10483  }

```

(End definition for `\ior_show_list:`, `\ior_log_list:`, and `\_ior_list:N`. These functions are documented on page 145.)

### 18.1.3 Reading input

`\if_eof:w` The primitive conditional

```
10484 \cs_new_eq:NN \if_eof:w \tex_ifeof:D
```

(End definition for `\if_eof:w`. This function is documented on page 151.)

`\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).

```
10485 \prg_new_conditional:Npnn \ior_if_eof:N #1 { p , T , F , TF }
10486   {
10487     \cs_if_exist:NTF #1
10488     {
10489       \int_compare:nTF { -1 < #1 < \c_term_ior }
10490       {
10491         \if_eof:w #1
10492           \prg_return_true:
10493         \else:
10494           \prg_return_false:
10495         \fi:
10496       }
10497       { \prg_return_true: }
10498     }
10499     { \prg_return_true: }
10500   }
```

(End definition for `\ior_if_eof:NTF`. This function is documented on page 148.)

`\ior_get:NN` And here we read from files.

```
10501 \cs_new_protected:Npn \ior_get:NN #1#2
10502   { \tex_read:D #1 to #2 }
```

(End definition for `\ior_get:NN`. This function is documented on page 146.)

`\ior_str_get:NN` Reading as strings is a more complicated wrapper, as we wish to remove the endline character and restore it afterwards.

```
10503 \cs_new_protected:Npn \ior_str_get:NN #1#2
10504   {
10505     \exp_args:Nno \use:n
10506     {
10507       \int_set:Nn \tex_endlinechar:D { -1 }
10508       \tex_readline:D #1 to #2
10509       \int_set:Nn \tex_endlinechar:D
10510     } { \int_use:N \tex_endlinechar:D }
10511   }
```

(End definition for `\ior_str_get:NN`. This function is documented on page 146.)

**\ior\_map\_break:**

**\ior\_map\_break:n**

Usual map breaking functions.

```

10512 \cs_new:Npn \ior_map_break:
10513   { \prg_map_break:Nn \ior_map_break: { } }
10514 \cs_new:Npn \ior_map_break:n
10515   { \prg_map_break:Nn \ior_map_break: }
```

(End definition for **\ior\_map\_break:** and **\ior\_map\_break:n**. These functions are documented on page 147.)

**\ior\_map\_inline:Nn**

**\ior\_str\_map\_inline:Nn**

**\\_\_ior\_map\_inline:NNn**

**\\_\_ior\_map\_inline:NNNn**

**\\_\_ior\_map\_inline\_loop:NNN**

Mapping to 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”.

```

10516 \cs_new_protected:Npn \ior_map_inline:Nn
10517   { \__ior_map_inline:NNn \ior_get:NN }
10518 \cs_new_protected:Npn \ior_str_map_inline:Nn
10519   { \__ior_map_inline:NNn \ior_str_get:NN }
10520 \cs_new_protected:Npn \__ior_map_inline:NNn
10521   {
10522     \int_gincr:N \g__kernel_prg_map_int
10523     \exp_args:Nc \__ior_map_inline:NNNn
10524     { \__ior_map_ \int_use:N \g__kernel_prg_map_int :n }
10525   }
10526 \cs_new_protected:Npn \__ior_map_inline:NNNn #1#2#3#4
10527   {
10528     \cs_gset_protected:Npn #1 ##1 {#4}
10529     \ior_if_eof:NF #3 { \__ior_map_inline_loop:NNN #1#2#3 }
10530     \prg_break_point:Nn \ior_map_break:
10531     { \int_gdecr:N \g__kernel_prg_map_int }
10532   }
10533 \cs_new_protected:Npn \__ior_map_inline_loop:NNN #1#2#3
10534   {
10535     #2 #3 \l__ior_internal_tl
10536     \if_eof:w #3
10537       \exp_after:wN \ior_map_break:
10538     \fi:
10539     \exp_args:No #1 \l__ior_internal_tl
10540     \__ior_map_inline_loop:NNN #1#2#3
10541   }
```

(End definition for **\ior\_map\_inline:Nn** and others. These functions are documented on page 147.)

**\ior\_map\_variable:NNn**

**\ior\_str\_map\_variable:NNn**

**\\_\_ior\_map\_variable:NNNn**

**\\_\_ior\_map\_variable\_loop:NNNn**

```

10542 \cs_new_protected:Npn \ior_map_variable:NNn
10543   { \__ior_map_variable:NNNn \ior_get:NN }
10544 \cs_new_protected:Npn \ior_str_map_variable:NNn
10545   { \__ior_map_variable:NNNn \ior_str_get:NN }
10546 \cs_new_protected:Npn \__ior_map_variable:NNNn #1#2#3#4
10547   {
10548     \ior_if_eof:NF #2 { \__ior_map_variable_loop:NNNn #1#2#3 {#4} }
10549     \prg_break_point:Nn \ior_map_break: { }
10550   }
```

```

10551 \cs_new_protected:Npn \__ior_map_variable_loop:NNNn #1#2#3#4
10552 {
10553     #1 #2 #3
10554     \if_eof:w #2
10555         \exp_after:wN \ior_map_break:
10556     \fi:
10557     #4
10558     \__ior_map_variable_loop:NNNn #1#2#3 {#4}
10559 }

```

(End definition for `\ior_map_variable:NNn` and others. These functions are documented on page 246.)

## 18.2 Output operations

```
10560 ⟨@=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.

### 18.2.1 Variables and constants

`\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.

```

10561 \int_const:Nn \c_log_iow { -1 }
10562 \int_const:Nn \c_term_iow
10563 {
10564     \bool_lazy_and:nnTF
10565         { \sys_if_engine_luatex_p: }
10566         { \int_compare_p:nNn \tex_luatexversion:D > { 80 } }
10567         { 128 }
10568         { 16 }
10569 }

```

(End definition for `\c_log_iow` and `\c_term_iow`. These variables are documented on page 151.)

`\g__iow_streams_seq`

A list of the currently-available output streams to be used as a stack. The stream 18 is special, as `\write18` is used to denote commands to be sent to the OS.

```

10570 \seq_new:N \g__iow_streams_seq
10571 ⟨*initex⟩
10572 \exp_args:Nnx \use:n
10573     { \seq_gset_split:Nnn \g__iow_streams_seq { } }
10574     {
10575         \int_step_function:nnN { 0 } { \c_term_iow }
10576         \prg_do_nothing:
10577     }
10578 \int_compare:nNnF \c_term_iow < { 18 }
10579     { \seq_gremove_all:Nn \g__iow_streams_seq { 18 } }
10580 ⟨/initex⟩

```

(End definition for `\g__iow_streams_seq`.)

`\l__iow_stream_tl` Used to recover the raw stream number from the stack.

```
10581 \tl_new:N \l__iow_stream_tl
```

(End definition for `\l_iow_stream_t1`.)

`\g_iow_streams_prop` As for reads with the appropriate adjustment of the register numbers to check on.

```
10582 \prop_new:N \g_iow_streams_prop
10583 (*package)
10584 \int_step_inline:nnn
10585   { 0 }
10586   {
10587     \cs_if_exist:NTF \normalend
10588     { \tex_count:D 39 ~ }
10589     {
10590       \tex_count:D 17 ~
10591       \cs_if_exist:NT \loccount { - 1 }
10592     }
10593   }
10594   {
10595     \prop_gput:Nnn \g_iow_streams_prop {#1} { Reserved~by~format }
10596   }
10597 
```

(End definition for `\g_iow_streams_prop`.)

### 18.3 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.

```
10598 \cs_new_protected:Npn \iow_new:N #1 { \cs_new_eq:NN #1 \c_term_iow }
10599 \cs_generate_variant:Nn \iow_new:N { c }
```

(End definition for `\iow_new:N`. This function is documented on page 144.)

`\g_tmpa_iow` The usual scratch space.

```
10600 \iow_new:N \g_tmpa_iow
10601 \iow_new:N \g_tmpb_iow
```

(End definition for `\g_tmpa_iow` and `\g_tmpb_iow`. These variables are documented on page 151.)

`\_iow_new:N` As for read streams, copy `\newwrite` in package mode, making sure that it is not `\outer`.

```
10602 (*package)
10603 \exp_args:Nnf \cs_new_protected:Npn \_iow_new:N
10604   { \exp_args:NNc \exp_after:wN \exp_stop_f: { newwrite } }
10605 
```

(End definition for `\_iow_new:N`.)

`\l_iow_file_name_str` Data storage.

```
10606 \str_new:N \l_iow_file_name_str
```

(End definition for `\l_iow_file_name_str`.)

**\iow\_open:Nn**  
**\iow\_open:cn**

\\_\\_iow\\_open\\_stream:Nn  
\\_\\_iow\\_open\\_stream:NV

```
10607 \cs_new_protected:Npn \iow_open:Nn #1#2
10608 {
10609     \__kernel_file_name_sanitize:nN {#2} \l_iow_file_name_str
10610     \iow_close:N #1
10611     \seq_gpop:NNTF \g_iow_streams_seq \l_iow_stream_tl
10612     { \_\_iow_open_stream:NV #1 \l_iow_file_name_str }
10613 (*initex)
10614     { \__kernel_msg_fatal:nn { kernel } { output-streams-exhausted } }
10615 (/initex)
10616 (*package)
10617 {
10618     \__iow_new:N #1
10619     \tl_set:Nx \l_iow_stream_tl { \int_eval:n {#1} }
10620     \_\_iow_open_stream:NV #1 \l_iow_file_name_str
10621 }
10622 (/package)
10623 }
10624 \cs_generate_variant:Nn \iow_open:Nn { c }
10625 \cs_new_protected:Npn \_\_iow_open_stream:Nn #1#2
10626 {
10627     \tex_global:D \tex_chardef:D #1 = \l_iow_stream_tl \scan_stop:
10628     \prop_gput:NVn \g_iow_streams_prop #1 {#2}
10629     \tex_immediate:D \tex_openout:D #1 #2 \scan_stop:
10630 }
10631 \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 145.)

**\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.

```
10632 \cs_new_protected:Npn \iow_close:N #1
10633 {
10634     \int_compare:nT { - \c_log_iow < #1 < \c_term_iow }
10635     {
10636         \tex_immediate:D \tex_closeout:D #1
10637         \prop_gremove:NV \g_iow_streams_prop #1
10638         \seq_if_in:NVF \g_iow_streams_seq #1
10639         { \seq_gpush:NV \g_iow_streams_seq #1 }
10640         \cs_gset_eq:NN #1 \c_term_iow
10641     }
10642 }
10643 \cs_generate_variant:Nn \iow_close:N { c }
```

(End definition for `\iow_close:N`. This function is documented on page 145.)

**\iow\_show\_list:**  
**\iow\_log\_list:**  
\\_\\_iow\\_list:N

Done as for input, but with a copy of the auxiliary so the name is correct.

```
10644 \cs_new_protected:Npn \iow_show_list: { \_\_iow_list:N \msg_show:nnxxxx }
10645 \cs_new_protected:Npn \iow_log_list: { \_\_iow_list:N \msg_log:nnxxxx }
10646 \cs_new_protected:Npn \_\_iow_list:N #1
10647 {
```

```

10648     #1 { LaTeX / kernel } { show-streams }
10649     { iow }
10650     {
10651         \prop_map_function:NN \g_iow_streams_prop
10652             \msg_show_item_unbraced:nn
10653     }
10654     { } { }
10655 }
```

(End definition for `\iow_show_list:`, `\iow_log_list:`, and `\_iow_list:N`. These functions are documented on page 145.)

### 18.3.1 Deferred writing

`\iow_shipout_x:Nn`  
`\iow_shipout_x:Nx`  
`\iow_shipout_x:cn`  
`\iow_shipout_x:cx`

```

10656 \cs_new_protected:Npn \iow_shipout_x:Nn #1#2
10657     { \tex_write:D #1 {#2} }
10658 \cs_generate_variant:Nn \iow_shipout_x:Nn { c, Nx, cx }
```

(End definition for `\iow_shipout_x:Nn`. This function is documented on page 149.)

`\iow_shipout:Nn`  
`\iow_shipout:Nx`  
`\iow_shipout:cn`  
`\iow_shipout:cx`

```

10659 \cs_new_protected:Npn \iow_shipout:Nn #1#2
10660     { \tex_write:D #1 { \exp_not:n {#2} } }
10661 \cs_generate_variant:Nn \iow_shipout:Nn { c, Nx, cx }
```

(End definition for `\iow_shipout:Nn`. This function is documented on page 149.)

### 18.3.2 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.

```

10662 \cs_new_protected:Npn \_kernel_iow_with:Nnn #1#2
10663     {
10664         \int_compare:nNnTF {#1} = {#2}
10665             { \use:n }
10666             { \exp_args:No \_iow_with:nNnn { \int_use:N #1 } #1 {#2} }
10667     }
10668 \cs_new_protected:Npn \_iow_with:nNnn #1#2#3#4
10669     {
10670         \int_set:Nn #2 {#3}
10671         #4
10672         \int_set:Nn #2 {#1}
10673     }
```

(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 T<sub>E</sub>X: 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.

```
10674 \cs_new_protected:Npn \iow_now:Nn #1#2
10675 {
10676     \__kernel_iow_with:Nnn \tex_newlinechar:D { '\^J }
10677     { \tex_immediate:D \tex_write:D #1 { \exp_not:n {#2} } }
10678 }
10679 \cs_generate_variant:Nn \iow_now:Nn { c, Nx, cx }
```

(End definition for `\iow_now:Nn`. This function is documented on page 148.)

`\iow_log:n` Writing to the log and the terminal directly are relatively easy.

```
10680 \cs_set_protected:Npn \iow_log:x { \iow_now:Nx \c_log_iow }
10681 \cs_new_protected:Npn \iow_log:n { \iow_now:Nn \c_log_iow }
10682 \cs_set_protected:Npn \iow_term:x { \iow_now:Nx \c_term_iow }
10683 \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 148.)

### 18.3.3 Special characters for writing

`\iow_newline`: Global variable holding the character that forces a new line when something is written to an output stream.

```
10684 \cs_new:Npn \iow_newline: { ^J }
```

(End definition for `\iow_newline`. This function is documented on page 149.)

`\iow_char:N` Function to write any escaped char to an output stream.

```
10685 \cs_new_eq:NN \iow_char:N \cs_to_str:N
```

(End definition for `\iow_char:N`. This function is documented on page 149.)

### 18.3.4 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 TeXLive and MiKTeX.

```
10686 \int_new:N \l_iow_line_count_int
10687 \int_set:Nn \l_iow_line_count_int { 78 }
```

(End definition for `\l_iow_line_count_int`. This variable is documented on page 150.)

`\l__iow_newline_tl` The token list inserted to produce a new line, with the *<run-on text>*.

```
10688 \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.

```
10689 \int_new:N \l__iow_line_target_int
```

(End definition for `\l__iow_line_target_int`.)

\\_\\_iow\\_set\\_indent:n  
\\_\\_iow\\_unindent:w  
\l\\_iow\\_one\\_indent\\_tl  
\l\\_iow\\_one\\_indent\\_int

The `one_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.

```

10690 \tl_new:N \l\_iow\_one\_indent\_tl
10691 \int_new:N \l\_iow\_one\_indent\_int
10692 \cs_new:Npn \_\_iow\_unindent:w { }
10693 \cs_new_protected:Npn \_\_iow\_set\_indent:n #1
10694 {
10695     \tl_set:Nx \l\_iow\_one\_indent\_tl
10696         { \exp_args:No \_\_kernel_str_to_other_fast:n { \tl_to_str:n {#1} } }
10697     \int_set:Nn \l\_iow\_one\_indent\_int
10698         { \str_count:N \l\_iow\_one\_indent\_tl }
10699     \exp_last_unbraced:NNo
10700         \cs_set:Npn \_\_iow\_unindent:w \l\_iow\_one\_indent\_tl { }
10701     }
10702 \exp_args:Nx \_\_iow\_set\_indent:n { \prg_replicate:nn { 4 } { ~ } }
```

(End definition for `\_\_iow\_set\_indent:n` and others.)

\l\\_iow\\_indent\\_tl  
\l\\_iow\\_indent\\_int

The current indentation (some copies of `\l\_iow\_one\_indent\_tl`) and its number of characters.

```

10703 \tl_new:N \l\_iow\_indent\_tl
10704 \int_new:N \l\_iow\_indent\_int
```

(End definition for `\l\_iow\_indent\_tl` and `\l\_iow\_indent\_int`.)

\l\\_iow\\_line\\_tl  
\l\\_iow\\_line\\_part\\_tl

These hold the current line of text and a partial line to be added to it, respectively.

```

10705 \tl_new:N \l\_iow\_line\_tl
10706 \tl_new:N \l\_iow\_line\_part\_tl
```

(End definition for `\l\_iow\_line\_tl` and `\l\_iow\_line\_part\_tl`.)

\l\\_iow\\_line\\_break\\_bool

```
10707 \bool_new:N \l\_iow\_line\_break\_bool
```

(End definition for `\l\_iow\_line\_break\_bool`.)

\l\\_iow\\_wrap\\_tl

Used for the expansion step before detokenizing, and for the output from wrapping text: fully expanded and with lines which are not overly long.

```
10708 \tl_new:N \l\_iow\_wrap\_tl
```

(End definition for `\l\_iow\_wrap\_tl`.)

\c\\_iow\\_wrap\\_marker\\_tl  
\c\\_iow\\_wrap\\_end\\_marker\\_tl  
\c\\_iow\\_wrap\\_newline\\_marker\\_tl  
\c\\_iow\\_wrap\\_allow\\_break\\_marker\\_tl

Every special action of the wrapping code starts with the same recognizable string, `\c\_iow\_wrap\_marker\_tl`. Upon seeing that “word”, the wrapping code reads one space-delimited argument to know what operation to perform. The setting of `\escapechar` here is not very important, but makes `\c\_iow\_wrap\_marker\_tl` look marginally nicer.

```

10709 \group_begin:
10710     \int_set:Nn \tex_escapechar:D { -1 }
10711     \tl_const:Nx \c\_iow\_wrap\_marker\_tl
10712         { \tl_to_str:n { \^I \^O \^W \^_ \^W \^R \^A \^P } }
10713 \group_end:
10714 \tl_map_inline:nn
```

```

10715 { { end } { newline } { allow_break } { indent } { unindent } }
10716 {
10717   \tl_const:cx { c__iow_wrap_ #1 _marker_tl }
10718   {
10719     \c__iow_wrap_marker_tl
10720     #1
10721     \c_catcode_other_space_tl
10722   }
10723 }
```

(End definition for `\c__iow_wrap_marker_tl` and others.)

**\iow\_allow\_break:**  
`\_\_iow_allow_break:`  
`\_\_iow_allow_break_error:`

We set `\iow_allow_break:n` to produce an error when outside messages. Within wrapped message, it is set to `\_\_iow_allow_break:` when valid and otherwise to `\_\_iow_allow_break_error:`. The second produces an error expandably.

```

10724 \cs_new_protected:Npn \iow_allow_break:
10725   {
10726     \_\_kernel_msg_error:nnnm { kernel } { iow-indent }
10727     { \iow_wrap:nnN } { \iow_allow_break: }
10728   }
10729 \cs_new:Npx \_\_iow_allow_break: { \c__iow_wrap_allow_break_marker_tl }
10730 \cs_new:Npn \_\_iow_allow_break_error:
10731   {
10732     \_\_kernel_msg_expandable_error:nnnn { kernel } { iow-indent }
10733     { \iow_wrap:nnN } { \iow_allow_break: }
10734 }
```

(End definition for `\iow_allow_break:`, `\_\_iow_allow_break:`, and `\_\_iow_allow_break_error:`. This function is documented on page 246.)

**\iow\_indent:n**  
`\_\_iow_indent:n`  
`\_\_iow_indent_error: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.

```

10735 \cs_new_protected:Npn \iow_indent:n #1
10736   {
10737     \_\_kernel_msg_error:nnnnn { kernel } { iow-indent }
10738     { \iow_wrap:nnN } { \iow_indent:n } {#1}
10739     #1
10740   }
10741 \cs_new:Npx \_\_iow_indent:n #1
10742   {
10743     \c__iow_wrap_indent_marker_tl
10744     #1
10745     \c__iow_wrap_unindent_marker_tl
10746   }
10747 \cs_new:Npn \_\_iow_indent_error:n #1
10748   {
10749     \_\_kernel_msg_expandable_error:nnnnn { kernel } { iow-indent }
10750     { \iow_wrap:nnN } { \iow_indent:n } {#1}
10751     #1
10752 }
```

(End definition for `\iow_indent:n`, `\_iow_indent:n`, and `\_iow_indent_error:n`. This function is documented on page 150.)

- `\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.  
`\iow_wrap:nxnN` 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.

```

10753 \cs_new_protected:Npn \iow_wrap:nnnN #1#2#3#4
10754   {
10755     \group_begin:
10756     \use:c { conditionally@traceoff }
10757     \int_set:Nn \tex_escapechar:D { -1 }
10758     \cs_set:Npx \{ { \token_to_str:N \{ }
10759     \cs_set:Npx \# { \token_to_str:N \# }
10760     \cs_set:Npx \} { \token_to_str:N \} }
10761     \cs_set:Npx \% { \token_to_str:N \% }
10762     \cs_set:Npx \~ { \token_to_str:N \~ }
10763     \int_set:Nn \tex_escapechar:D { 92 }
10764     \cs_set_eq:NN \\ \iow_newline:
10765     \cs_set_eq:NN \c_catcode_other_space_tl
10766     \cs_set_eq:NN \iow_allow_break: \_iow_allow_break:
10767     \cs_set_eq:NN \iow_indent:n \_iow_indent:n
10768   #3

```

Then fully-expand the input: in package mode, the expansion uses L<sup>A</sup>T<sub>E</sub>X 2<sub><</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:nnnN`.

```

10769 \package \cs_set_eq:NN \protect \token_to_str:N
10770   \tl_set:Nx \l_iow_wrap_tl {#1}
10771   \cs_set_eq:NN \iow_allow_break: \_iow_allow_break_error:
10772   \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).

```

10773   \tl_set:Nx \l_iow_newline_tl { \iow_newline: #2 }
10774   \tl_set:Nx \l_iow_newline_tl { \tl_to_str:N \l_iow_newline_tl }
10775   \int_set:Nn \l_iow_line_target_int
10776     { \l_iow_line_count_int - \str_count:N \l_iow_newline_tl + 1 }

```

Sanity check.

```

10777   \int_compare:nNnT { \l_iow_line_target_int } < 0
10778   {
10779     \tl_set:Nn \l_iow_newline_tl { \iow_newline: }
10780     \int_set:Nn \l_iow_line_target_int
10781       { \l_iow_line_count_int + 1 }
10782   }

```

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`.

```

10783      \__iow_wrap_do:
10784      \exp_args:Nnf \group_end:
10785      #4 { \tl_to_str:N \l__iow_wrap_tl }
10786  }
10787 \cs_generate_variant:Nn \iow_wrap:nnN { nx }

```

(End definition for `\iow_wrap:nnN`. This function is documented on page ??.)

`\__iow_wrap_do:`  
`\__iow_wrap_fix_newline:w`  
`\__iow_wrap_start:w`

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.

```

10788 \cs_new_protected:Npn \__iow_wrap_do:
10789  {
10790      \tl_set:Nx \l__iow_wrap_tl
10791      {
10792          \exp_args:No \__kernel_str_to_other_fast:n \l__iow_wrap_tl
10793          \c__iow_wrap_end_marker_tl
10794      }
10795      \tl_set:Nx \l__iow_wrap_tl
10796      {
10797          \exp_after:wN \__iow_wrap_fix_newline:w \l__iow_wrap_tl
10798          ^^J \q_nil ^^J \q_stop
10799      }
10800      \exp_after:wN \__iow_wrap_start:w \l__iow_wrap_tl
10801  }
10802 \cs_new:Npn \__iow_wrap_fix_newline:w #1 ^^J #2 ^^J
10803  {
10804      #1
10805      \if_meaning:w \q_nil #2
10806          \use_i_delimit_by_q_stop:nw
10807      \fi:
10808      \c__iow_wrap_newline_marker_tl
10809      \__iow_wrap_fix_newline:w #2 ^^J
10810  }
10811 \cs_new_protected:Npn \__iow_wrap_start:w
10812  {
10813      \bool_set_false:N \l__iow_line_break_bool
10814      \tl_clear:N \l__iow_line_tl
10815      \tl_clear:N \l__iow_line_part_tl
10816      \tl_set:Nn \l__iow_wrap_tl { ~ \use_none:n }
10817      \int_zero:N \l__iow_indent_int
10818      \tl_clear:N \l__iow_indent_tl
10819      \__iow_wrap_chunk:nw { \l__iow_line_count_int }
10820  }

```

(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.

```

10821 \cs_set_protected:Npn \__iow_tmp:w #1#2
10822 {
10823   \cs_new_protected:Npn \__iow_wrap_chunk:nw ##1##2 #2
10824   {
10825     \tl_if_empty:nTF {##2}
10826     {
10827       \tl_clear:N \l__iow_line_part_tl
10828       \__iow_wrap_next:nw {##1}
10829     }
10830   {
10831     \tl_if_empty:NTF \l__iow_line_tl
10832     {
10833       \__iow_wrap_line:nw
10834       { \l__iow_indent_tl }
10835       ##1 - \l__iow_indent_int ;
10836     }
10837     { \__iow_wrap_line:nw { } ##1 ; }
10838     ##2 #1
10839     \__iow_wrap_end_chunk:w 7 6 5 4 3 2 1 0 \q_stop
10840   }
10841 }
10842 \cs_new_protected:Npn \__iow_wrap_next:nw ##1##2 #1
10843 { \use:c { __iow_wrap_##2:n } {##1} }
10844 }
10845 \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:nnnnnnn
  \__iow_wrap_line_end:NnnnnnnnN
\__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:nnnnn` 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:nnnnn` ensures that the required data is in the right place.

```

10846 \cs_new_protected:Npn \__iow_wrap_line:nw #
10847 {
10848   \tex_eodef:D \l__iow_line_part_tl { \if_false: } \fi:
10849   #1

```

```

10850      \exp_after:wN \__iow_wrap_line_loop:w
10851      \int_value:w \int_eval:w
10852  }
10853 \cs_new:Npn \__iow_wrap_line_loop:w #1 ; #2#3#4#5#6#7#8#9
10854  {
10855      \if_int_compare:w #1 < 8 \exp_stop_f:
10856          \__iow_wrap_line_aux:Nw #1
10857      \fi:
10858      #2 #3 #4 #5 #6 #7 #8 #9
10859      \exp_after:wN \__iow_wrap_line_loop:w
10860      \int_value:w \int_eval:w #1 - 8 ;
10861  }
10862 \cs_new:Npn \__iow_wrap_line_aux:Nw #1#2#3 \exp_after:wN #4 ;
10863  {
10864      #2
10865      \exp_after:wN \__iow_wrap_line_end:NnnnnnnnN
10866      \exp_after:wN #1
10867      \exp:w \exp_end_continue_f:w
10868      \exp_after:wN \exp_after:wN
10869      \if_case:w #1 \exp_stop_f:
10870          \prg_do_nothing:
10871          \or: \use_none:n
10872          \or: \use_none:nn
10873          \or: \use_none:nnn
10874          \or: \use_none:nnnn
10875          \or: \use_none:nnnnn
10876          \or: \use_none:nnnnnn
10877          \or: \__iow_wrap_line_seven:nnnnnnn
10878          \fi:
10879          { } { } { } { } { } { } { } #3
10880  }
10881 \cs_new:Npn \__iow_wrap_line_seven:nnnnnnn #1#2#3#4#5#6#7 { \exp_stop_f: }
10882 \cs_new:Npn \__iow_wrap_line_end:NnnnnnnnN #1#2#3#4#5#6#7#8#9
10883  {
10884      #2 #3 #4 #5 #6 #7 #8
10885      \use_none:nnnnn \int_eval:w 8 - ; #9
10886      \token_if_eq_charcode:NNTF \c_space_token #9
10887          { \__iow_wrap_line_end:nw { } }
10888          { \if_false: { \fi: } \__iow_wrap_break:w #9 }
10889      }
10890 \cs_new:Npn \__iow_wrap_line_end:nw #1
10891  {
10892      \if_false: { \fi: }
10893      \__iow_wrap_store_do:n {#1}
10894      \__iow_wrap_next_line:w
10895  }
10896 \cs_new:Npn \__iow_wrap_end_chunk:w
10897      #1 \int_eval:w #2 - #3 ; #4#5 \q_stop
10898  {
10899      \if_false: { \fi: }
10900      \exp_args:Nf \__iow_wrap_next:nw { \int_eval:n { #2 - #4 } }
10901  }

```

(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).

```

10902 \cs_set_protected:Npn \__iow_tmp:w #1
10903   {
10904     \cs_new:Npn \__iow_wrap_break:w
10905       {
10906         \tex_eodef:D \l__iow_line_part_tl
10907           { \if_false: } \fi:
10908             \exp_after:wN \__iow_wrap_break_first:w
10909               \l__iow_line_part_tl
10910                 #1
10911                 { ? \__iow_wrap_break_end:w }
10912                   \q_mark
10913                 }
10914     \cs_new:Npn \__iow_wrap_break_first:w ##1 #1 ##2
10915       {
10916         \use_none:nn ##2 \__iow_wrap_break_none:w
10917           \__iow_wrap_break_loop:w ##1 #1 ##2
10918         }
10919     \cs_new:Npn \__iow_wrap_break_none:w ##1##2 #1 ##3 \q_mark ##4 #1
10920       {
10921         \tl_if_empty:NTF \l__iow_line_tl
10922           { ##2 ##4 \__iow_wrap_line_end:nw { } }
10923             { \__iow_wrap_line_end:nw { \__iow_wrap_trim:N } ##2 ##4 #1 }
10924         }
10925     \cs_new:Npn \__iow_wrap_break_loop:w ##1 #1 ##2 #1 ##3
10926       {
10927         \use_none:n ##3
10928           ##1 #1
10929             \__iow_wrap_break_loop:w ##2 #1 ##3
10930           }
10931     \cs_new:Npn \__iow_wrap_break_end:w ##1 #1 ##2 ##3 #1 ##4 \q_mark
10932       { ##1 \__iow_wrap_line_end:nw { } ##3 }
10933     }
10934 \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).

```

10935 \cs_new_protected:Npn \__iow_wrap_next_line:w #1#2 \q_stop
10936   {

```

```

10937   \tl_clear:N \l__iow_line_tl
10938   \token_if_eq_meaning:NNTF #1 \__iow_wrap_end_chunk:w
10939   {
10940     \tl_clear:N \l__iow_line_part_tl
10941     \bool_set_true:N \l__iow_line_break_bool
10942     \__iow_wrap_next:nw { \l__iow_line_target_int }
10943   }
10944   {
10945     \__iow_wrap_line:nw
10946     { \l__iow_indent_tl }
10947     \l__iow_line_target_int - \l__iow_indent_int ;
10948     #1 #2 \q_stop
10949   }
10950 }
```

(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.

```

10951 \cs_new_protected:Npn \__iow_wrap_allow_break:n #1
10952   {
10953     \tl_set:Nx \l__iow_line_tl
10954     { \l__iow_line_tl \__iow_wrap_trim:N \l__iow_line_part_tl }
10955     \bool_set_false:N \l__iow_line_break_bool
10956     \tl_if_empty:NTF \l__iow_line_part_tl
10957     { \__iow_wrap_chunk:nw {#1} }
10958     { \exp_args:Nf \__iow_wrap_chunk:nw { \int_eval:n { #1 + 1 } } }
10959   }
```

(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.

```

10960 \cs_new_protected:Npn \__iow_wrap_indent:n #1
10961   {
10962     \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
10963     \bool_set_false:N \l__iow_line_break_bool
10964     \int_add:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10965     \tl_put_right:No \l__iow_indent_tl { \l__iow_one_indent_tl }
10966     \__iow_wrap_chunk:nw {#1}
10967   }
10968 \cs_new_protected:Npn \__iow_wrap_unindent:n #1
10969   {
10970     \tl_put_right:Nx \l__iow_line_tl { \l__iow_line_part_tl }
10971     \bool_set_false:N \l__iow_line_break_bool
10972     \int_sub:Nn \l__iow_indent_int { \l__iow_one_indent_int }
10973     \tl_set:Nx \l__iow_indent_tl
10974     { \exp_after:wN \__iow_unindent:w \l__iow_indent_tl }
10975     \__iow_wrap_chunk:nw {#1}
10976   }
```

(End definition for `\_iow_wrap_indent:n` and `\_iow_wrap_unindent:n`.)

`\_iow_wrap_newline: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.

```
10977 \cs_new_protected:Npn \_iow_wrap_newline:n #1
10978 {
10979     \bool_if:NF \l_iow_line_break_bool
10980         { \_iow_wrap_store_do:n { \_iow_wrap_trim:N } }
10981     \bool_set_false:N \l_iow_line_break_bool
10982     \_iow_wrap_chunk:nw { \l_iow_line_target_int }
10983 }
10984 \cs_new_protected:Npn \_iow_wrap_end:n #1
10985 {
10986     \bool_if:NF \l_iow_line_break_bool
10987         { \_iow_wrap_store_do:n { \_iow_wrap_trim:N } }
10988     \bool_set_false:N \l_iow_line_break_bool
10989 }
```

(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`).

```
10990 \cs_new_protected:Npn \_iow_wrap_store_do:n #1
10991 {
10992     \tl_set:Nx \l_iow_line_tl
10993         { \l_iow_line_tl \l_iow_line_part_tl }
10994     \tl_set:Nx \l_iow_wrap_tl
10995         {
10996             \l_iow_wrap_tl
10997             \l_iow_newline_tl
10998             #1 \l_iow_line_tl
10999         }
11000     \tl_clear:N \l_iow_line_tl
11001 }
```

(End definition for `\_iow_wrap_store_do:n`.)

`\_iow_wrap_trim:N` Remove one trailing “other” space from the argument if present.

```
11002 \cs_set_protected:Npn \_iow_tmp:w #1
11003 {
11004     \cs_new:Npn \_iow_wrap_trim:N ##1
11005         { \exp_after:wN \_iow_wrap_trim:w ##1 \q_mark #1 \q_stop }
11006     \cs_new:Npn \_iow_wrap_trim:w ##1 #1 \q_mark
11007         { \_iow_wrap_trim_aux:w ##1 \q_mark }
11008     \cs_new:Npn \_iow_wrap_trim_aux:w ##1 \q_mark ##2 \q_stop {##1}
11009 }
11010 \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`.)

11011 ⟨@=file⟩

## 18.4 File operations

\l\_file\_internal\_tl Used as a short-term scratch variable.  
 11012 \tl\_new:N \l\_file\_internal\_tl  
*(End definition for \l\_file\_internal\_tl.)*

\g\_file\_internal\_ior A reserved stream to test for file existence.  
 11013 \ior\_new:N \g\_file\_internal\_ior  
*(End definition for \g\_file\_internal\_ior.)*

\g\_file\_curr\_dir\_str \g\_file\_curr\_ext\_str \g\_file\_curr\_name\_str The name of the current file should be available at all times. For the format the file name needs to be picked up at the start of the run. In L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  package mode the current file name is collected from \currname.  
 11014 \str\_new:N \g\_file\_curr\_dir\_str  
 11015 \str\_new:N \g\_file\_curr\_ext\_str  
 11016 \str\_new:N \g\_file\_curr\_name\_str  
 11017 (\*initex)  
 11018 \tex\_everyjob:D \exp\_after:wN  
 11019 {  
 11020 \tex\_the:D \tex\_everyjob:D  
 11021 \str\_gset:Nx \g\_file\_curr\_name\_str { \tex\_jobname:D }  
 11022 }  
 11023   
 11024 (\*package)  
 11025 \cs\_if\_exist:NT \currname  
 11026 { \str\_gset\_eq:NN \g\_file\_curr\_name\_str \currname }  
 11027   
*(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 151.)*

\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.  
 11028 \seq\_new:N \g\_file\_stack\_seq  
 11029 (\*package)  
 11030 \group\_begin:  
 11031 \cs\_set\_protected:Npn \\_file\_tmp:w #1#2#3  
 11032 {  
 11033 \tl\_if\_blank:nTF {#1}  
 11034 {  
 11035 \cs\_set:Npn \\_file\_tmp:w ##1 " ##2 " ##3 \q\_stop  
 11036 { { } {#2} { } }  
 11037 \seq\_gput\_right:Nx \g\_file\_stack\_seq  
 11038 {  
 11039 \exp\_after:wN \\_file\_tmp:w \tex\_jobname:D  
 11040 " \tex\_jobname:D " \q\_stop  
 11041 }  
 11042 }  
 11043 {  
 11044 \seq\_gput\_right:Nn \g\_file\_stack\_seq { { } {#1} {#2} }

```

11045           \_\_file\_tmp:w
11046       }
11047   }
11048   \cs_if_exist:NT \currnamestack
11049     { \exp_after:wN \_\_file\_tmp:w \currnamestack }
11050 \group_end:
11051 
```

(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! In format mode, this is done at the very start of the TeX run. In package mode we will eventually copy the contents of `\@filelist`.

```

11052 \seq_new:N \g_file_record_seq
11053 <*initex>
11054 \tex_everyjob:D \exp_after:wN
11055   {
11056     \tex_the:D \tex_everyjob:D
11057     \seq_gput_right:NV \g_file_record_seq \g_file_curr_name_str
11058   }
11059 
```

(End definition for `\g_file_record_seq`.)

`\l_file_base_name_str` For storing the basename and full path whilst passing data internally.

`\l_file_full_name_str`

```

11060 \str_new:N \l_file_base_name_str
11061 \str_new:N \l_file_full_name_str

```

(End definition for `\l_file_base_name_str` and `\l_file_full_name_str`.)

`\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.

```

11062 \str_new:N \l_file_dir_str
11063 \str_new:N \l_file_ext_str
11064 \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.

```

11065 \seq_new:N \l_file_search_path_seq

```

(End definition for `\l_file_search_path_seq`. This variable is documented on page 152.)

`\l_file_tmp_seq` Scratch space for comma list conversion in package mode.

```

11066 <*package>
11067 \seq_new:N \l_file_tmp_seq
11068 
```

(End definition for `\l_file_tmp_seq`.)

```
\_\_kernel_file_name_sanitize:nN
\_\_file_name_quote:nN
\_\_file_name_sanitize_aux:n
```

For converting a token list to a string where active characters are treated as strings from the start. The logic to the quoting normalisation is the same as used by `\ualatexquotejobname`: check for balanced ", and assuming they balance strip all of them out before quoting the entire name if it contains spaces.

```
11069 \cs_new_protected:Npn \_\_kernel_file_name_sanitize:nN #1#2
11070  {
11071    \group_begin:
11072    \seq_map_inline:Nn \l_char_active_seq
11073    {
11074      \tl_set:Nx \l__file_internal_tl { \iow_char:N ##1 }
11075      \char_set_active_eq:NN ##1 \l__file_internal_tl
11076    }
11077    \tl_set:Nx \l__file_internal_tl {#1}
11078    \tl_set:Nx \l__file_internal_tl
11079    { \tl_to_str:N \l__file_internal_tl }
11080    \exp_args:NNNV \group_end:
11081    \str_set:Nn #2 \l__file_internal_tl
11082  }
11083 \cs_new_protected:Npn \_\_file_name_quote:nN #1#2
11084  {
11085    \str_set:Nx #2 {#1}
11086    \int_if_even:nF
11087    { 0 \tl_map_function:NN #2 \_\_file_name_quote_aux:n }
11088    {
11089      \_\_kernel_msg_error:nnx
11090      { kernel } { unbalanced-quote-in-filename } {#2}
11091    }
11092    \tl_remove_all:Nn #2 { " }
11093    \tl_if_in:NnT #2 { ~ }
11094    { \str_set:Nx #2 { " \exp_not:V #2 " } }
11095  }
11096 \cs_new:Npn \_\_file_name_quote_aux:n #1
11097  { \token_if_eq_charcode:NNT #1 " { + 1 } }
```

(End definition for `\_\_kernel_file_name_sanitize:nN`, `\_\_file_name_quote:nN`, and `\_\_file_name_sanitize_aux:n`.)

`\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.

```
11098 \tl_const:Nx \c\_file_marker_tl { : \token_to_str:N : }
```

(End definition for `\c\_file_marker_tl`.)

```
\file_get:nnNTF
\file_get:nnN
\_\_file_get_aux:nnN
\_\_file_get_do:Nw
```

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.

```
11099 \cs_new_protected:Npn \file_get:nnN #1#2#3
11100  {
11101    \file_get:nnNF {#1} {#2} #3
11102    { \_\_kernel_file_missing:n {#1} }
11103  }
11104 \prg_new_protected_conditional:Npnn \file_get:nnN #1#2#3 { T , F , TF }
```

```

11105  {
11106    \file_get_full_name:nN {#1} \l_file_full_name_str
11107    \str_if_empty:NTF \l_file_full_name_str
11108      { \prg_return_false: }
11109      {
11110        \exp_args:NV \__file_get_aux:nnN
11111          \l_file_full_name_str
11112            {#2} #3
11113          \prg_return_true:
11114        }
11115      }
11116 \cs_new_protected:Npn \__file_get_aux:nnN #1#2#3
11117  {
11118    \if_false: { \fi:
11119    \group_begin:
11120      \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
11121      \exp_args:No \tex_everyeof:D { \c_file_marker_tl }
11122      #2 \scan_stop:
11123      \exp_after:wN \__file_get_do:Nw
11124      \exp_after:wN #3
11125      \exp_after:wN \prg_do_nothing:
11126        \tex_input:D #1 \scan_stop:
11127      \if_false: } \fi:
11128    }
11129  \exp_args:Nno \use:nn
11130    { \cs_new_protected:Npn \__file_get_do:Nw #1#2 }
11131    { \c_file_marker_tl }
11132  {
11133    \group_end:
11134    \tl_set:No #1 {#2}
11135  }

```

(End definition for `\file_get:nnNTF` and others. These functions are documented on page 152.)

**\file\_get\_full\_name:nN** 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.

```

11136 \cs_new_protected:Npn \file_get_full_name:nN #1#2
11137  {
11138    \__kernel_file_name_sanitize:nN {#1} \l_file_base_name_str
11139    \__file_get_full_name_search:nN { } \use:n
11140    \seq_map_inline:Nn \l_file_search_path_seq
11141      { \__file_get_full_name_search:nN { ##1 / } \seq_map_break:n }
11142  (*package)
11143    \cs_if_exist:NT \input@path
11144    {
11145      \tl_map_inline:Nn \input@path
11146        { \__file_get_full_name_search:nN { ##1 } \tl_map_break:n }
11147    }
11148  (/package)

```

```

11149     \str_clear:N \l__file_full_name_str
11150     \prg_break_point:
11151     \str_if_empty:NF \l__file_full_name_str
11152     {
11153         \exp_args:NV \file_parse_full_name:nNNN \l__file_full_name_str
11154         \l__file_dir_str \l__file_name_str \l__file_ext_str
11155         \str_if_empty:NT \l__file_ext_str
11156         {
11157             \__kernel_iор_open:No \g__file_internal_iор
11158             { \l__file_full_name_str .tex }
11159             \ior_if_eof:NF \g__file_internal_iор
11160             { \str_put_right:Nn \l__file_full_name_str { .tex } }
11161         }
11162     }
11163     \str_set_eq:NN #2 \l__file_full_name_str
11164     \ior_close:N \g__file_internal_iор
11165 }
11166 \cs_generate_variant:Nn \file_get_full_name:nN { V }
11167 \cs_new_protected:Npn \__file_get_full_name_search:nN #1#2
11168 {
11169     \__file_name_quote:nN
11170     { \tl_to_str:n {#1} \l__file_base_name_str }
11171     \l__file_full_name_str
11172     \__kernel_iор_open:No \g__file_internal_iор \l__file_full_name_str
11173     \ior_if_eof:NF \g__file_internal_iор { #2 { \prg_break: } }
11174 }

```

(End definition for `\file_get_full_name:nN` and `\__file_get_full_name_search:nN`. This function is documented on page 152.)

**\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.

```

11175 \prg_new_protected_conditional:Npnn \file_if_exist:n #1 { T , F , TF }
11176 {
11177     \file_get_full_name:nN {#1} \l__file_full_name_str
11178     \str_if_empty:NTF \l__file_full_name_str
11179     { \prg_return_false: }
11180     { \prg_return_true: }
11181 }

```

(End definition for `\file_if_exist:nTF`. This function is documented on page 152.)

**\\_\_kernel\_file\_missing:n** An error message for a missing file, also used in `\ior_open:Nn`.

```

11182 \cs_new_protected:Npn \__kernel_file_missing:n #1
11183 {
11184     \__kernel_file_name_sanitize:nN {#1} \l__file_base_name_str
11185     \__kernel_msg_error:nnx { kernel } { file-not-found }
11186     { \l__file_base_name_str }
11187 }

```

(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
\__file_input:n if it does. Push the file name on the \g__file_stack_seq, and add it to the file list,
\__file_input:V either \g__file_record_seq, or \@filelist in package mode.

\__file_input_push:n 11188 \cs_new_protected:Npn \file_input:n #1
\__kernel_file_input_push:n 11189 {
\__file_input_pop: 11190   \file_get_full_name:nN {#1} \l__file_full_name_str
\__kernel_file_input_pop: 11191   \str_if_empty:NTF \l__file_full_name_str
\__file_input_pop:nnn 11192   { \__kernel_file_missing:n {#1} }
11193   { \__file_input:V \l__file_full_name_str }

}
11194
11195 \cs_new_protected:Npn \__file_input:n #1
11196 {
11197  {*initex}
11198  \seq_gput_right:Nn \g__file_record_seq {#1}
11199  (/initex)
11200  (*package)
11201  \clist_if_exist:NTF \@filelist
11202  { \addtofilelist {#1} }
11203  { \seq_gput_right:Nn \g__file_record_seq {#1} }
11204  (/package)
11205  \__file_input_push:n {#1}
11206  \tex_input:D #1 \c_space_tl
11207  \__file_input_pop:
11208 }
11209 \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.

```

11210 \cs_new_protected:Npn \__file_input_push:n #1
11211 {
11212  \seq_gpush:Nx \g__file_stack_seq
11213  {
11214    { \g_file_curr_dir_str }
11215    { \g_file_curr_name_str }
11216    { \g_file_curr_ext_str }
11217  }
11218  \file_parse_full_name:nNNN {#1}
11219  \l__file_dir_str \l__file_name_str \l__file_ext_str
11220  \str_gset_eq:NN \g_file_curr_dir_str \l__file_dir_str
11221  \str_gset_eq:NN \g_file_curr_name_str \l__file_name_str
11222  \str_gset_eq:NN \g_file_curr_ext_str \l__file_ext_str
11223 }
11224 (*package)
11225 \cs_new_eq:NN \__kernel_file_input_push:n \__file_input_push:n
11226 (/package)
11227 \cs_new_protected:Npn \__file_input_pop:
11228 {
11229  \seq_gpop>NN \g__file_stack_seq \l__file_internal_tl
11230  \exp_after:wN \__file_input_pop:nnn \l__file_internal_tl
11231 }
11232 (*package)
11233 \cs_new_eq:NN \__kernel_file_input_pop: \__file_input_pop:
11234 (/package)
11235 \cs_new_protected:Npn \__file_input_pop:nnn #1#2#3


```

```

11236      {
11237          \str_gset:Nn \g_file_curr_dir_str  {#1}
11238          \str_gset:Nn \g_file_curr_name_str {#2}
11239          \str_gset:Nn \g_file_curr_ext_str  {#3}
11240      }

```

(End definition for `\file_input:n` and others. This function is documented on page 153.)

### `\file_parse_full_name:nNNN`

```

\__file_parse_full_name_auxi:w
\__file_parse_full_name_split:nNNNTF

```

Parsing starts by stripping off any surrounding quotes. Then find the directory #4 by splitting at the last /. (The auxiliary returns `true/false` depending on whether it found the delimiter.) We correct for the case of a file in the root /, as in that case we wish to keep the trailing (and only) slash. Then split the base name #5 at the last dot. If there was indeed a dot, #5 contains the name and #6 the extension without the dot, which we add back for convenience. In the special case of no extension given, the auxiliary stored the name into #6, we just have to move it to #5.

```

11241 \cs_new_protected:Npn \file_parse_full_name:nNNN #1#2#3#4
11242 {
11243     \exp_after:wN \__file_parse_full_name_auxi:w
11244         \tl_to_str:n { #1 " #1 " } \q_stop #2#3#4
11245     }
11246 \cs_new_protected:Npn \__file_parse_full_name_auxi:w
11247     #1 " #2 " #3 \q_stop #4#5#6
11248 {
11249     \__file_parse_full_name_split:nNNNTF {#2} / #4 #5
11250         { \str_if_empty:NT #4 { \str_set:Nn #4 { / } } }
11251         {
11252             \exp_args:No \__file_parse_full_name_split:nNNNTF {#5} . #5 #6
11253                 { \str_put_left:Nn #6 { . } }
11254                 {
11255                     \str_set_eq:NN #5 #6
11256                     \str_clear:N #6
11257                 }
11258             }
11259 \cs_new_protected:Npn \__file_parse_full_name_split:nNNNTF #1#2#3#4
11260 {
11261     \cs_set_protected:Npn \__file_tmp:w ##1 ##2 #2 ##3 \q_stop
11262     {
11263         \tl_if_empty:nTF {##3}
11264         {
11265             \str_set:Nn #4 {##2}
11266             \tl_if_empty:nTF {##1}
11267             {
11268                 \str_clear:N #3
11269                 \use_i:nn
11270             }
11271             {
11272                 \str_set:Nx #3 { \str_tail:n {##1} }
11273                 \use_i:nn
11274             }
11275         }
11276         { \__file_tmp:w { ##1 #2 ##2 } ##3 \q_stop }
11277     }
11278     \__file_tmp:w { } #1 #2 \q_stop
11279 }

```

(End definition for `\file_parse_full_name:nNNN`, `\_file_parse_full_name_aux:w`, and `\_file_parse_full_name_split:nNNNTF`. This function is documented on page 152.)

`\file_show_list:` A function to list all files used to the log, without duplicates. In package mode, if `\@filelist` is still defined, we need to take this list of file names into account (we capture it `\AtBeginDocument` into `\g_file_record_seq`), turning it to a string (this does not affect the commas of this comma list).

```

11280 \cs_new_protected:Npn \file_show_list: { \_file_list:N \msg_show:nnxxxx }
11281 \cs_new_protected:Npn \file_log_list: { \_file_list:N \msg_log:nnxxxx }
11282 \cs_new_protected:Npn \_file_list:N #1
11283 {
11284     \seq_clear:N \l__file_tmp_seq
11285     (*package)
11286     \clist_if_exist:NT \@filelist
11287     {
11288         \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
11289         { \tl_to_str:N \@filelist }
11290     }
11291 
```

```

11292     \seq_concat:NNN \l__file_tmp_seq \l__file_tmp_seq \g_file_record_seq
11293     \seq_remove_duplicates:N \l__file_tmp_seq
11294     #1 { LaTeX/kernel } { file-list }
11295     { \seq_map_function:NN \l__file_tmp_seq \_file_list_aux:n }
11296     { } { } { }
11297 }
11298 \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 153.)

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`.

```

11299 (*package)
11300 \AtBeginDocument
11301 {
11302     \exp_args:NNx \seq_set_from_clist:Nn \l__file_tmp_seq
11303     { \tl_to_str:N \@filelist }
11304     \seq_gconcat:NNN
11305     \g_file_record_seq
11306     \g_file_record_seq
11307     \l__file_tmp_seq
11308 }
11309 
```

## 18.5 Messages

```

11310 \_kernel_msg_new:nnnn { kernel } { file-not-found }
11311     { File~'#1'~not~found. }
11312     {
11313         The~requested~file~could~not~be~found~in~the~current~directory,~
11314         in~the~TeX~search~path~or~in~the~LaTeX~search~path.
11315     }
11316 \_kernel_msg_new:nnn { kernel } { file-list }
11317     { }
```

```

11318     >-File~List-<
11319     #1 \\
11320     .....
11321   }
11322 \__kernel_msg_new:nnnn { kernel } { input-streams-exhausted }
11323   { Input-streams-exhausted }
11324   {
11325     TeX-can-only-open-up-to-16-input-streams-at-one-time.\\
11326     All-16-are-currently-in-use,-and-something-wanted-to-open-
11327     another-one.
11328   }
11329 \__kernel_msg_new:nnnn { kernel } { output-streams-exhausted }
11330   { Output-streams-exhausted }
11331   {
11332     TeX-can-only-open-up-to-16-output-streams-at-one-time.\\
11333     All-16-are-currently-in-use,-and-something-wanted-to-open-
11334     another-one.
11335   }
11336 \__kernel_msg_new:nnnn { kernel } { unbalanced-quote-in-filename }
11337   { Unbalanced-quotes-in-file-name-'#1'. }
11338   {
11339     File-names-must-contain-balanced-numbers-of-quotes-(").
11340   }
11341 \__kernel_msg_new:nnnn { kernel } { iow-indent }
11342   { Only-#1 (arg-1)-allows-#2 }
11343   {
11344     The-command-#2 can-only-be-used-in-messages-
11345     which-will-be-wrapped-using-#1.
11346     \tl_if_empty:nF {#3} { ~ It-was-called-with-argument-'#3'. }
11347   }
11348 
```

## 19 I3skip implementation

```

11349  <*initex | package>
11350  <@=dim>

```

### 19.1 Length primitives renamed

**\if\_dim:w** Primitives renamed.

```

\__dim_eval:w    \cs_new_eq:NN \if_dim:w      \tex_ifdim:D
\__dim_eval_end: \cs_new_eq:NN \__dim_eval:w    \tex_dimexpr:D
\__dim_eval_end: \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 168.)

### 19.2 Creating and initialising dim variables

**\dim\_new:N** Allocating  $\langle dim \rangle$  registers ...

```

\dim_new:c 11354  <*package>
                \cs_new_protected:Npn \dim_new:N #1
11355  {
11356    \__kernel_chk_if_free_cs:N #1

```

```

11358      \cs:w newdimen \cs_end: #1
11359    }
11360  </package>
11361 \cs_generate_variant:Nn \dim_new:N { c }

(End definition for \dim_new:N. This function is documented on page 154.)
```

**\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.

```

11362 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
11363 \cs_new_protected:Npn \dim_const:Nn #1#2
11364  {
11365    \dim_new:N #1
11366    \tex_global:D #1 ~ \dim_eval:n {#2} \scan_stop:
11367  }
11368 \cs_generate_variant:Nn \dim_const:Nn { c }

(End definition for \dim_const:Nn. This function is documented on page 154.)
```

**\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).

```

11369 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
11370 \cs_new_protected:Npn \dim_zero:N #1 { #1 \c_zero_skip }
11371 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
11372 \cs_new_protected:Npn \dim_gzero:N #1
11373  { \tex_global:D #1 \c_zero_skip }
11374 \cs_generate_variant:Nn \dim_zero:N { c }
11375 \cs_generate_variant:Nn \dim_gzero:N { c }

(End definition for \dim_zero:N and \dim_gzero:N. These functions are documented on page 154.)
```

**\dim\_zero\_new:N** Create a register if needed, otherwise clear it.

```

11376 \cs_new_protected:Npn \dim_zero_new:N #1
11377  { \dim_if_exist:NTF #1 { \dim_zero:N #1 } { \dim_new:N #1 } }
11378 \cs_new_protected:Npn \dim_gzero_new:N #1
11379  { \dim_if_exist:NTF #1 { \dim_gzero:N #1 } { \dim_new:N #1 } }
11380 \cs_generate_variant:Nn \dim_zero_new:N { c }
11381 \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 154.)
```

**\dim\_if\_exist\_p:N** Copies of the cs functions defined in l3basics.

```

11382 \prg_new_eq_conditional:NNn \dim_if_exist:N \cs_if_exist:N
11383  { TF , T , F , p }
11384 \prg_new_eq_conditional:NNn \dim_if_exist:c \cs_if_exist:c
11385  { TF , T , F , p }

(End definition for \dim_if_exist:NTF. This function is documented on page 154.)
```

### 19.3 Setting dim variables

Several functions here have a signature :Nn and are such that when debugging, the first argument should be checked to be a local/global variable and the second should be wrapped in code for an expression. The temporary function `\__dim_tmp:w` finds the name #3 of the function being redefined and writes the appropriate patch.

```

11386 \cs_set_protected:Npn \__dim_tmp:w #1#2#3
11387   {
11388     \__kernel_patch_args:nnnNNpn
11389     { #1 ##1 }
11390     { }
11391     { {##1} { \__kernel_chk_expr:nNn {##2} \__dim_eval:w { } #3 } }
11392     #2 #3
11393   }

```

**\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).

```

11394 \__dim_tmp:w \__kernel_chk_var_local:N
11395 \cs_new_protected:Npn \dim_set:Nn #1#2
11396   { #1 ~ \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
11397 \__dim_tmp:w \__kernel_chk_var_global:N
11398 \cs_new_protected:Npn \dim_gset:Nn #1#2
11399   { \tex_global:D #1 ~ \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
11400 \cs_generate_variant:Nn \dim_set:Nn { c }
11401 \cs_generate_variant:Nn \dim_gset:Nn { c }

```

(End definition for `\dim_set:Nn` and `\dim_gset:Nn`. These functions are documented on page 155.)

**\dim\_set\_eq:NN** All straightforward, with a `\scan_stop:` to deal with the case where #1 is (incorrectly) a skip.

```

11402 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
11403 \cs_new_protected:Npn \dim_set_eq:NN #1#2
11404   { #1 = #2 \scan_stop: }
11405 \cs_generate_variant:Nn \dim_set_eq:NN { c , Nc , cc }
11406 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
11407 \cs_new_protected:Npn \dim_gset_eq:NN #1#2
11408   { \tex_global:D #1 = #2 \scan_stop: }
11409 \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 155.)

**\dim\_add:Nn** Using by here deals with the (incorrect) case `\dimen123`. 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. The debugging code is inserted by `\__dim_tmp:w`.

```

11410 \__dim_tmp:w \__kernel_chk_var_local:N
11411 \cs_new_protected:Npn \dim_add:Nn #1#2
11412   { \tex_advance:D #1 by \__dim_eval:w #2 \__dim_eval_end: \scan_stop: }
11413 \__dim_tmp:w \__kernel_chk_var_global:N
11414 \cs_new_protected:Npn \dim_gadd:Nn #1#2
11415   {

```

```

11416      \tex_global:D \tex_advance:D #1 by
11417          \_dim_eval:w #2 \_dim_eval_end: \scan_stop:
11418      }
11419 \cs_generate_variant:Nn \dim_add:Nn { c }
11420 \cs_generate_variant:Nn \dim_gadd:Nn { c }
11421 \_dim_tmp:w \_kernel_chk_var_local:N
11422 \cs_new_protected:Npn \dim_sub:Nn #1#2
11423     { \tex_advance:D #1 by - \_dim_eval:w #2 \_dim_eval_end: \scan_stop: }
11424 \_dim_tmp:w \_kernel_chk_var_global:N
11425 \cs_new_protected:Npn \dim_gsub:Nn #1#2
11426     {
11427         \tex_global:D \tex_advance:D #1 by
11428             -\_dim_eval:w #2 \_dim_eval_end: \scan_stop:
11429     }
11430 \cs_generate_variant:Nn \dim_sub:Nn { c }
11431 \cs_generate_variant:Nn \dim_gsub:Nn { c }

```

(End definition for `\dim_add:Nn` and others. These functions are documented on page 155.)

## 19.4 Utilities for dimension calculations

`\dim_abs:n`  
`\_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.

```

11432 \_kernel_patch_args:nNNpn
11433     { { \_kernel_chk_expr:nNnN {#1} \_dim_eval:w { } \dim_abs:n } }
11434 \cs_new:Npn \dim_abs:n #1
11435     {
11436         \exp_after:wN \_dim_abs:N
11437         \dim_use:N \_dim_eval:w #1 \_dim_eval_end:
11438     }
11439 \cs_new:Npn \_dim_abs:N #1
11440     { \if_meaning:w - #1 \else: \exp_after:wN #1 \fi: }
11441 \_kernel_patch_args:nNNpn
11442     {
11443         { \_kernel_chk_expr:nNnN {#1} \_dim_eval:w { } \dim_max:nn }
11444         { \_kernel_chk_expr:nNnN {#2} \_dim_eval:w { } \dim_max:nn }
11445     }
11446 \cs_new:Npn \dim_max:nn #1#2
11447     {
11448         \dim_use:N \_dim_eval:w \exp_after:wN \_dim_maxmin:wwN
11449             \dim_use:N \_dim_eval:w #1 \exp_after:wN ;
11450             \dim_use:N \_dim_eval:w #2 ;
11451         >
11452         \_dim_eval_end:
11453     }
11454 \_kernel_patch_args:nNNpn
11455     {
11456         { \_kernel_chk_expr:nNnN {#1} \_dim_eval:w { } \dim_min:nn }
11457         { \_kernel_chk_expr:nNnN {#2} \_dim_eval:w { } \dim_min:nn }
11458     }
11459 \cs_new:Npn \dim_min:nn #1#2
11460     {
11461         \dim_use:N \_dim_eval:w \exp_after:wN \_dim_maxmin:wwN
11462             \dim_use:N \_dim_eval:w #1 \exp_after:wN ;

```

```

11463      \dim_use:N \__dim_eval:w #2 ;
11464      <
11465      \__dim_eval_end:
11466  }
11467 \cs_new:Npn \__dim_maxmin:wwN #1 ; #2 ; #3
11468  {
11469      \if_dim:w #1 #3 #2 ~
11470          #1
11471      \else:
11472          #2
11473      \fi:
11474  }

```

(End definition for `\dim_abs:n` and others. These functions are documented on page 155.)

**\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.

```

11475 \cs_new:Npn \dim_ratio:nn #1#2
11476  { \__dim_ratio:n {#1} / \__dim_ratio:n {#2} }
11477 \cs_new:Npn \__dim_ratio:n #1
11478  { \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 156.)

## 19.5 Dimension expression conditionals

**\dim\_compare\_p:nNn**  
**\dim\_compare:nNnTF**

Simple comparison.

```

11479 \__kernel_patch_conditional_args:nNNpnn
11480  {
11481      { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_compare:nNn }
11482      { \__dim_eval_end: #2 }
11483      { \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { } \dim_compare:nNn }
11484  }
11485 \prg_new_conditional:Npnn \dim_compare:nNn #1#2#3 { p , T , F , TF }
11486  {
11487      \if_dim:w \__dim_eval:w #1 #2 \__dim_eval:w #3 \__dim_eval_end:
11488          \prg_return_true: \else: \prg_return_false: \fi:
11489  }

```

(End definition for `\dim_compare:nNnTF`. This function is documented on page 156.)

**\dim\_compare\_p:n**  
**\dim\_compare:nTF**  
**\\_\_dim\_compare:w**  
**\\_\_dim\_compare:wnN**  
**\\_\_dim\_compare\_=:w**  
**\\_\_dim\_compare\_!:>:w**  
**\\_\_dim\_compare\_<:w**  
**\\_\_dim\_compare\_>:w**  
**\\_\_dim\_compare\_error:**

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 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 `>`.

```

11490 \prg_new_conditional:Npnn \dim_compare:n #1 { p , T , F , TF }
11491  {
11492      \exp_after:wN \__dim_compare:w
11493      \dim_use:N \__dim_eval:w #1 \__dim_compare_error:
11494  }

```

```

11495 \cs_new:Npn \__dim_compare:w #1 \__dim_compare_error:
11496 {
11497     \exp_after:wN \if_false: \exp:w \exp_end_continue_f:w
11498         \__dim_compare:wNN #1 ? { = \__dim_compare_end:w \else: } \q_stop
11499 }
11500 \exp_args:Nno \use:nn
11501 { \cs_new:Npn \__dim_compare:wNN #1 } { \tl_to_str:n {pt} #2#3 }
11502 {
11503     \if_meaning:w = #3
11504         \use:c { __dim_compare_#2:w }
11505     \fi:
11506         #1 pt \exp_stop_f:
11507     \prg_return_false:
11508     \exp_after:wN \use_none_delimit_by_q_stop:w
11509 \fi:
11510 \reverse_if:N \if_dim:w #1 pt #2
11511     \exp_after:wN \__dim_compare:wNN
11512     \dim_use:N \__dim_eval:w #3
11513 }
11514 \cs_new:cpn { __dim_compare_ ! :w }
11515     #1 \reverse_if:N #2 ! #3 = { #1 #2 = #3 }
11516 \cs_new:cpn { __dim_compare_ = :w }
11517     #1 \__dim_eval:w = { #1 \__dim_eval:w }
11518 \cs_new:cpn { __dim_compare_ < :w }
11519     #1 \reverse_if:N #2 < #3 = { #1 #2 > #3 }
11520 \cs_new:cpn { __dim_compare_ > :w }
11521     #1 \reverse_if:N #2 > #3 = { #1 #2 < #3 }
11522 \cs_new:Npn \__dim_compare_end:w #1 \prg_return_false: #2 \q_stop
11523 { #1 \prg_return_false: \else: \prg_return_true: \fi: }
11524 \cs_new_protected:Npn \__dim_compare_error:
11525 {
11526     \if_int_compare:w \c_zero_int \c_zero_int \fi:
11527     =
11528     \__dim_compare_error:
11529 }

```

(End definition for `\dim_compare:nTF` and others. This function is documented on page [157](#).)

### `\dim_case:nn` `\dim_case:nnTF`

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:nn(TF)` as described in [l3basics](#).

```

11530 \cs_new:Npn \dim_case:nnTF #1
11531 {
11532     \exp:w
11533     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} }
11534 }
11535 \cs_new:Npn \dim_case:nnT #1#2#3
11536 {
11537     \exp:w
11538     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} {#3} { }
11539 }
11540 \cs_new:Npn \dim_case:nnF #1#2
11541 {
11542     \exp:w
11543     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { }

```

```

11544     }
11545   \cs_new:Npn \dim_case:nn #1#2
11546   {
11547     \exp:w
11548     \exp_args:Nf \__dim_case:nnTF { \dim_eval:n {#1} } {#2} { } { }
11549   }
11550 \cs_new:Npn \__dim_case:nnTF #1#2#3#4
11551   { \__dim_case:nw {#1} #2 {#1} { } \q_mark {#3} \q_mark {#4} \q_stop }
11552 \cs_new:Npn \__dim_case:nw #1#2#3
11553   {
11554     \dim_compare:nNnTF {#1} = {#2}
11555     { \__dim_case_end:nw {#3} }
11556     { \__dim_case:nw {#1} }
11557   }
11558 \cs_new:Npn \__dim_case_end:nw #1#2#3 \q_mark #4#5 \q_stop
11559   { \exp_end: #1 #4 }

```

(End definition for `\dim_case:nnTF` and others. This function is documented on page 158.)

## 19.6 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.

```

11560 \cs_new:Npn \dim_while_do:nn #1#2
11561   {
11562     \dim_compare:nT {#1}
11563     {
11564       #2
11565       \dim_while_do:nn {#1} {#2}
11566     }
11567   }
11568 \cs_new:Npn \dim_until_do:nn #1#2
11569   {
11570     \dim_compare:nF {#1}
11571     {
11572       #2
11573       \dim_until_do:nn {#1} {#2}
11574     }
11575   }
11576 \cs_new:Npn \dim_do_while:nn #1#2
11577   {
11578     #2
11579     \dim_compare:nT {#1}
11580     { \dim_while_do:nn {#1} {#2} }
11581   }
11582 \cs_new:Npn \dim_do_until:nn #1#2
11583   {
11584     #2
11585     \dim_compare:nF {#1}
11586     { \dim_do_until:nn {#1} {#2} }
11587   }

```

(End definition for `\dim_while_do:nn` and others. These functions are documented on page 159.)

```

\dim_while_do:nNnn while_do and do_while functions for dimensions. Same as for the int type only the
\dim_until_do:nNnn names have changed.

11588 \cs_new:Npn \dim_while_do:nNnn #1#2#3#4
11589 {
11590     \dim_compare:nNnT {#1} #2 {#3}
11591     {
11592         #4
11593         \dim_while_do:nNnn {#1} #2 {#3} {#4}
11594     }
11595 }
11596 \cs_new:Npn \dim_until_do:nNnn #1#2#3#4
11597 {
11598     \dim_compare:nNnF {#1} #2 {#3}
11599     {
11600         #4
11601         \dim_until_do:nNnn {#1} #2 {#3} {#4}
11602     }
11603 }
11604 \cs_new:Npn \dim_do_while:nNnn #1#2#3#4
11605 {
11606     #4
11607     \dim_compare:nNnT {#1} #2 {#3}
11608     { \dim_while:nNnn {#1} #2 {#3} {#4} }
11609 }
11610 \cs_new:Npn \dim_do_until:nNnn #1#2#3#4
11611 {
11612     #4
11613     \dim_compare:nNnF {#1} #2 {#3}
11614     { \dim_do_until:nNnn {#1} #2 {#3} {#4} }
11615 }

```

(End definition for `\dim_while_do:nNnn` and others. These functions are documented on page 159.)

## 19.7 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).

```

11616 \__kernel_patch_args:nNNpn
11617 {
11618 {
11619     \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { }
11620     \dim_step_function:nnnN
11621 }
11622 {
11623     \__kernel_chk_expr:nNnN {#2} \__dim_eval:w { }
11624     \dim_step_function:nnnN
11625 }
11626 {
11627     \__kernel_chk_expr:nNnN {#3} \__dim_eval:w { }
11628     \dim_step_function:nnnN

```

```

11629      }
11630    }
11631 \cs_new:Npn \dim_step_function:nnnN #1#2#3
11632  {
11633    \exp_after:wN \__dim_step:wwwN
11634    \tex_the:D \__dim_eval:w #1 \exp_after:wN ;
11635    \tex_the:D \__dim_eval:w #2 \exp_after:wN ;
11636    \tex_the:D \__dim_eval:w #3 ;
11637  }
11638 \cs_new:Npn \__dim_step:wwwN #1; #2; #3; #4
11639  {
11640    \dim_compare:nNnTF {#2} > \c_zero_dim
11641    { \__dim_step:NnnnN > }
11642    {
11643      \dim_compare:nNnTF {#2} = \c_zero_dim
11644      {
11645        \__kernel_msg_expandable_error:nnn { kernel } { zero-step } {#4}
11646        \use_none:nnnn
11647      }
11648      { \__dim_step:NnnnN < }
11649    }
11650    {#1} {#2} {#3} #4
11651  }
11652 \cs_new:Npn \__dim_step:NnnnN #1#2#3#4#5
11653  {
11654    \dim_compare:nNnF {#2} #1 {#4}
11655    {
11656      #5 {#2}
11657      \exp_args:NNf \__dim_step:NnnnN
11658      #1 { \dim_eval:n { #2 + #3 } } {#3} {#4} #5
11659    }
11660  }

```

(End definition for `\dim_step_function:nnnN`, `\__dim_step:wwwN`, and `\__dim_step:NnnnN`. This function is documented on page 159.)

```

\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.

```

11661 \cs_new_protected:Npn \dim_step_inline:nnn
11662  {
11663    \int_gincr:N \g_kernel_prg_map_int
11664    \exp_args:NNc \__dim_step>NNnnnn
11665    \cs_gset_protected:Npn
11666    { \__dim_map_ \int_use:N \g_kernel_prg_map_int :w }
11667  }
11668 \cs_new_protected:Npn \dim_step_variable:nnnNn #1#2#3#4#5
11669  {
11670    \int_gincr:N \g_kernel_prg_map_int
11671    \exp_args:NNc \__dim_step>NNnnnn
11672    \cs_gset_protected:Npx

```

```

11673 { __dim_map_ \int_use:N \g__kernel_prg_map_int :w }
11674 {##1}{##2}{##3}
11675 {
11676     \tl_set:Nn \exp_not:N #4 {##1}
11677     \exp_not:n {##5}
11678 }
11679 }
11680 \cs_new_protected:Npn \__dim_step:NNnnnn #1#2#3#4#5#6
11681 {
11682     #1 #2 ##1 {##6}
11683     \dim_step_function:nnnN {##3} {##4} {##5} #2
11684     \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g__kernel_prg_map_int }
11685 }

```

(End definition for `\dim_step_inline:nnnn`, `\dim_step_variable:nnnNn`, and `\__dim_step:NNnnnn`. These functions are documented on page 159.)

## 19.8 Using dim expressions and variables

**\dim\_eval:n** Evaluating a dimension expression expandably.

```

11686 \__kernel_patch_args:nNNpn
11687 { { \__kernel_chk_expr:nNnN {##1} \__dim_eval:w { } \dim_eval:n } }
11688 \cs_new:Npn \dim_eval:n #1
11689 { \dim_use:N \__dim_eval:w #1 \__dim_eval_end: }

```

(End definition for `\dim_eval:n`. This function is documented on page 160.)

**\dim\_use:N** Accessing a  $\langle\dim\rangle$ .

**\dim\_use:c** `\cs_new_eq:NN \dim_use:N \tex_the:D`

We hand-code this for some speed gain:

```

11691 \%cs_generate_variant:Nn \dim_use:N { c }
11692 \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 160.)

**\dim\_to\_decimal:n** 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.

```

11693 \__kernel_patch_args:nNNpn
11694 { { \__kernel_chk_expr:nNnN {##1} \__dim_eval:w { } \dim_to_decimal:n } }
11695 \cs_new:Npn \dim_to_decimal:n #1
11696 {
11697     \exp_after:wN
11698     \__dim_to_decimal:w \dim_use:N \__dim_eval:w #1 \__dim_eval_end:
11699 }
11700 \use:x
11701 {
11702     \cs_new:Npn \exp_not:N \__dim_to_decimal:w
11703     ##1 . ##2 \tl_to_str:n { pt }
11704 }
11705 {
11706     \int_compare:nNnTF {##2} > { 0 }
11707     { #1 . #2 }

```

```

11708     { #1 }
11709 }
```

(End definition for `\dim_to_decimal:n` and `\_dim_to_decimal:w`. This function is documented on page 160.)

**\dim\_to\_decimal\_in\_bp:n** Conversion to big points is done using a scaling inside `\_dim_eval:w` as ε-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).

```

11710 \cs_new:Npn \dim_to_decimal_in_bp:n #1
11711   { \dim_to_decimal:n { ( #1 ) * 800 / 803 } }
```

(End definition for `\dim_to_decimal_in_bp:n`. This function is documented on page 161.)

**\dim\_to\_decimal\_in\_sp:n** Another hard-coded conversion: this one is necessary to avoid things going off-scale.

```

11712 \_kernel_patch_args:nNnpn
11713   {
11714     {
11715       \_kernel_chk_expr:nNnN {#1} \_dim_eval:w { }
11716       \dim_to_decimal_in_sp:n
11717     }
11718   }
11719 \cs_new:Npn \dim_to_decimal_in_sp:n #1
11720   { \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 161.)

**\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.

```

11721 \cs_new:Npn \dim_to_decimal_in_unit:nn #1#2
11722   {
11723     \dim_to_decimal:n
11724     {
11725       1pt *
11726       \dim_ratio:nn {#1} {#2}
11727     }
11728   }
```

(End definition for `\dim_to_decimal_in_unit:nn`. This function is documented on page 161.)

**\dim\_to\_fp:n** Defined in `l3fp-convert`, documented here.

(End definition for `\dim_to_fp:n`. This function is documented on page 161.)

## 19.9 Viewing dim variables

**\dim\_show:N** Diagnostics.

```

\dim_show:c 11729 \cs_new_eq:NN \dim_show:N \_kernel_register_show:N
11730 \cs_generate_variant:Nn \dim_show:N { c }
```

(End definition for `\dim_show:N`. This function is documented on page 161.)

**\dim\_show:n** Diagnostics. We don't use the TEx primitive `\showthe` to show dimension expressions: this gives a more unified output.

```

11731 \cs_new_protected:Npn \dim_show:n
11732   { \msg_show_eval:Nn \dim_eval:n }
```

(End definition for `\dim_show:n`. This function is documented on page 162.)

`\dim_log:N` Diagnostics. Redirect output of `\dim_show:n` to the log.  
`\dim_log:c` 11733 `\cs_new_eq:NN \dim_log:N \__kernel_register_log:N`  
`\dim_log:n` 11734 `\cs_new_eq:NN \dim_log:c \__kernel_register_log:c`  
11735 `\cs_new_protected:Npn \dim_log:n`  
11736 { `\msg_log_eval:Nn \dim_eval:n` }

(End definition for `\dim_log:N` and `\dim_log:n`. These functions are documented on page 162.)

## 19.10 Constant dimensions

`\c_zero_dim` Constant dimensions.  
`\c_max_dim` 11737 `\dim_const:Nn \c_zero_dim { 0 pt }`  
11738 `\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 162.)

## 19.11 Scratch dimensions

We provide two local and two global scratch registers, maybe we need more or less.  
`\l_tmpa_dim` 11739 `\dim_new:N \l_tmpa_dim`  
`\l_tmpb_dim` 11740 `\dim_new:N \l_tmpb_dim`  
`\g_tmpa_dim` 11741 `\dim_new:N \g_tmpa_dim`  
`\g_tmpb_dim` 11742 `\dim_new:N \g_tmpb_dim`

(End definition for `\l_tmpa_dim` and others. These variables are documented on page 162.)

## 19.12 Creating and initialising skip variables

11743 `<@=skip>`  
`\skip_new:N` Allocation of a new internal registers.  
`\skip_new:c` 11744 `(*package)`  
11745 `\cs_new_protected:Npn \skip_new:N #1`  
11746 {  
11747 `\__kernel_chk_if_free_cs:N #1`  
11748 `\cs:w newskip \cs_end: #1`  
11749 }  
11750 `</package>`  
11751 `\cs_generate_variant:Nn \skip_new:N { c }`

(End definition for `\skip_new:N`. This function is documented on page 162.)

`\skip_const:Nn` Contrarily to integer constants, we cannot avoid using a register, even for constants. See  
`\skip_const:cn` `\dim_const:Nn` for why we cannot use `\skip_gset:Nn`.  
11752 `\__kernel_patch:nnNpn { \__kernel_chk_var_scope>NN c #1 } { }`  
11753 `\cs_new_protected:Npn \skip_const:Nn #1#2`  
11754 {  
11755 `\skip_new:N #1`  
11756 `\tex_global:D #1 ~ \skip_eval:n {#2} \scan_stop:`  
11757 }  
11758 `\cs_generate_variant:Nn \skip_const:Nn { c }`

(End definition for `\skip_const:Nn`. This function is documented on page 162.)

```

\skip_zero:N Reset the register to zero.
\skip_zero:c 11759 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\skip_gzero:N 11760 \cs_new_protected:Npn \skip_zero:N #1 { #1 \c_zero_skip }
\skip_gzero:c 11761 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
11762 \cs_new_protected:Npn \skip_gzero:N #1 { \tex_global:D #1 \c_zero_skip }
11763 \cs_generate_variant:Nn \skip_zero:N { c }
11764 \cs_generate_variant:Nn \skip_gzero:N { c }

(End definition for \skip_zero:N and \skip_gzero:N. These functions are documented on page 163.)

```

```

\skip_zero_new:N Create a register if needed, otherwise clear it.
\skip_zero_new:c 11765 \cs_new_protected:Npn \skip_zero_new:N #1
\skip_gzero_new:N 11766 { \skip_if_exist:NTF #1 { \skip_zero:N #1 } { \skip_new:N #1 } }
\skip_gzero_new:c 11767 \cs_new_protected:Npn \skip_gzero_new:N #1
11768 { \skip_if_exist:NTF #1 { \skip_gzero:N #1 } { \skip_new:N #1 } }
11769 \cs_generate_variant:Nn \skip_zero_new:N { c }
11770 \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 163.)

```

```

\skip_if_exist_p:N Copies of the cs functions defined in l3basics.
\skip_if_exist_p:c 11771 \prg_new_eq_conditional:NNn \skip_if_exist:N \cs_if_exist:N
\skip_if_exist:NTF 11772 { TF , T , F , p }
\skip_if_exist:cTF 11773 \prg_new_eq_conditional:NNn \skip_if_exist:c \cs_if_exist:c
11774 { TF , T , F , p }

(End definition for \skip_if_exist:NTF. This function is documented on page 163.)

```

## 19.13 Setting skip variables

Much as for dim variables, `\__skip_tmp:w` prepares a patch for :Nn function definitions in which the first argument should be checked to be a local/global variable and the second should be wrapped in code for an expression.

```

11775 \cs_set_protected:Npn \__skip_tmp:w #1#2#3
11776 {
11777   \__kernel_patch_args:nnnNNpn
11778   { #1 ##1 }
11779   { }
11780   { ##1 { \__kernel_chk_expr:nNnN {##2} \tex_glueexpr:D { } #3 } }
11781   #2 #3
11782 }


```

```

\skip_set:Nn Much the same as for dimensions.
\skip_set:cn 11783 \__skip_tmp:w \__kernel_chk_var_local:N
\skip_gset:Nn 11784 \cs_new_protected:Npn \skip_set:Nn #1#2
\skip_gset:cn 11785 { #1 ~ \tex_glueexpr:D #2 \scan_stop: }
11786 \__skip_tmp:w \__kernel_chk_var_global:N
11787 \cs_new_protected:Npn \skip_gset:Nn #1#2
11788 { \tex_global:D #1 ~ \tex_glueexpr:D #2 \scan_stop: }
11789 \cs_generate_variant:Nn \skip_set:Nn { c }
11790 \cs_generate_variant:Nn \skip_gset:Nn { c }

(End definition for \skip_set:Nn and \skip_gset:Nn. These functions are documented on page 163.)

```

```

\skip_set_eq:NN All straightforward.
\skip_set_eq:cN
\skip_set_eq:Nc
\skip_set_eq:cc
\skip_gset_eq:NN
\skip_gset_eq:cN
\skip_gset_eq:Nc
\skip_gset_eq:cc
(End definition for \skip_set_eq:NN and \skip_gset_eq:NN. These functions are documented on page 163.)
\skip_add:Nn Using by here deals with the (incorrect) case \skip123.
\skip_add:cn
\skip_gadd:Nn
\skip_gadd:cn
\skip_sub:Nn
\skip_sub:cn
\skip_gsub:Nn
\skip_gsub:cn
11795 \__skip_tmp:w \__kernel_chk_var_local:N
11796 \cs_new_protected:Npn \skip_add:Nn #1#2
11797 { \tex_advance:D #1 by \tex_glueexpr:D #2 \scan_stop: }
11798 \__skip_tmp:w \__kernel_chk_var_global:N
11799 \cs_new_protected:Npn \skip_gadd:Nn #1#2
11800 { \tex_global:D \tex_advance:D #1 by \tex_glueexpr:D #2 \scan_stop: }
11801 \cs_generate_variant:Nn \skip_add:Nn { c }
11802 \cs_generate_variant:Nn \skip_gadd:Nn { c }
11803 \__skip_tmp:w \__kernel_chk_var_local:N
11804 \cs_new_protected:Npn \skip_sub:Nn #1#2
11805 { \tex_advance:D #1 by - \tex_glueexpr:D #2 \scan_stop: }
11806 \__skip_tmp:w \__kernel_chk_var_global:N
11807 \cs_new_protected:Npn \skip_gsub:Nn #1#2
11808 { \tex_global:D \tex_advance:D #1 by - \tex_glueexpr:D #2 \scan_stop: }
11809 \cs_generate_variant:Nn \skip_sub:Nn { c }
11810 \cs_generate_variant:Nn \skip_gsub:Nn { c }

(End definition for \skip_add:Nn and others. These functions are documented on page 163.)

```

## 19.14 Skip expression conditionals

\skip\_if\_eq\_p:nn Comparing skips means doing two expansions to make strings, and then testing them. As a result, only equality is tested.

```

11811 \prg_new_conditional:Npnn \skip_if_eq:nn #1#2 { p , T , F , TF }
11812 {
11813   \str_if_eq:eeTF { \skip_eval:n { #1 } } { \skip_eval:n { #2 } }
11814   { \prg_return_true: }
11815   { \prg_return_false: }
11816 }

```

(End definition for \skip\_if\_eq:nnTF. This function is documented on page 164.)

\skip\_if\_finite\_p:n With  $\epsilon$ -TeX, we have an easy access to the order of infinities 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.

```

11817 \cs_set_protected:Npn \__skip_tmp:w #1
11818 {
11819   \__kernel_patch_conditional_args:nNNpnn
11820   {
11821     {
11822       \__kernel_chk_expr:nNnN
11823       {##1} \tex_glueexpr:D { } \skip_if_finite:n

```

```

11824     }
11825   }
11826 \prg_new_conditional:Npnn \skip_if_finite:n ##1 { p , T , F , TF }
11827   {
11828     \exp_after:wN \__skip_if_finite:wwNw
11829     \skip_use:N \tex_glueexpr:D ##1 ; \prg_return_false:
11830     #1 ; \prg_return_true: \q_stop
11831   }
11832   \cs_new:Npn \__skip_if_finite:wwNw ##1 #1 ##2 ; ##3 ##4 \q_stop {##3}
11833 }
11834 \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 164.)

## 19.15 Using skip expressions and variables

`\skip_eval:n` Evaluating a skip expression expandably.

```

11835 \__kernel_patch_args:nNNpn
11836   { { \__kernel_chk_expr:nNnN {#1} \tex_glueexpr:D { } \skip_eval:n } }
11837   \cs_new:Npn \skip_eval:n #1
11838     { \skip_use:N \tex_glueexpr:D #1 \scan_stop: }
```

(End definition for `\skip_eval:n`. This function is documented on page 164.)

`\skip_use:N` Accessing a `\langle skip\rangle`.

```

\skip_use:c
11839 \cs_new_eq:NN \skip_use:N \tex_the:D
11840 \%cs_generate_variant:Nn \skip_use:N { c }
11841 \cs_new:Npn \skip_use:c #1 { \tex_the:D \cs:w #1 \cs_end: }
```

(End definition for `\skip_use:N`. This function is documented on page 164.)

## 19.16 Inserting skips into the output

`\skip_horizontal:N` Inserting skips.

```

\skip_horizontal:c
11842 \cs_new_eq:NN \skip_horizontal:N \tex_hskip:D
\skip_horizontal:n
11843 \__kernel_patch_args:nNNpn
11844   {
11845     {
11846       \__kernel_chk_expr:nNnN {#1} \tex_glueexpr:D { }
11847       \skip_horizontal:n
11848     }
11849   }
11850 \cs_new:Npn \skip_horizontal:n #1
11851   { \skip_horizontal:N \tex_glueexpr:D #1 \scan_stop: }
11852 \cs_new_eq:NN \skip_vertical:N \tex_vskip:D
11853 \__kernel_patch_args:nNNpn
11854   {
11855     {
11856       \__kernel_chk_expr:nNnN {#1} \tex_glueexpr:D { }
11857       \skip_vertical:n
11858     }
11859   }
11860 \cs_new:Npn \skip_vertical:n #1
```

```

11861   { \skip_vertical:N \tex_glueexpr:D #1 \scan_stop: }
11862   \cs_generate_variant:Nn \skip_horizontal:N { c }
11863   \cs_generate_variant:Nn \skip_vertical:N { c }

```

(End definition for `\skip_horizontal:N` and others. These functions are documented on page 165.)

## 19.17 Viewing skip variables

`\skip_show:N` Diagnostics.

```

\skip_show:c 11864 \cs_new_eq:NN \skip_show:N \__kernel_register_show:N
              11865 \cs_generate_variant:Nn \skip_show:N { c }

```

(End definition for `\skip_show:N`. This function is documented on page 164.)

`\skip_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show skip expressions: this gives a more unified output.

```

11866 \cs_new_protected:Npn \skip_show:n
      11867 { \msg_show_eval:Nn \skip_eval:n }

```

(End definition for `\skip_show:n`. This function is documented on page 164.)

`\skip_log:N` Diagnostics. Redirect output of `\skip_show:n` to the log.

```

\skip_log:c 11868 \cs_new_eq:NN \skip_log:N \__kernel_register_log:N
\skip_log:n 11869 \cs_new_eq:NN \skip_log:c \__kernel_register_log:c
              11870 \cs_new_protected:Npn \skip_log:n
              11871 { \msg_log_eval:Nn \skip_eval:n }

```

(End definition for `\skip_log:N` and `\skip_log:n`. These functions are documented on page 165.)

## 19.18 Constant skips

`\c_zero_skip` Skips with no rubber component are just dimensions but need to terminate correctly.

```

\c_max_skip 11872 \skip_const:Nn \c_zero_skip { \c_zero_dim }
              11873 \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 165.)

## 19.19 Scratch skips

`\l_tmpa_skip` We provide two local and two global scratch registers, maybe we need more or less.

```

\l_tmpb_skip 11874 \skip_new:N \l_tmpa_skip
\g_tmpa_skip 11875 \skip_new:N \l_tmpb_skip
\g_tmpb_skip 11876 \skip_new:N \g_tmpa_skip
              11877 \skip_new:N \g_tmpb_skip

```

(End definition for `\l_tmpa_skip` and others. These variables are documented on page 165.)

## 19.20 Creating and initialising muskip variables

**\muskip\_new:N** And then we add muskips.

```

\muskip_new:c 11878 {*package}
               11879 \cs_new_protected:Npn \muskip_new:N #1
               11880 {
               11881     \__kernel_chk_if_free_cs:N #1
               11882     \cs:w newmuskip \cs_end: #1
               11883 }
               11884 
```

(End definition for `\muskip_new:N`. This function is documented on page 166.)

**\muskip\_const:Nn** See `\skip_const:Nn`.

```

\muskip_const:c 11886 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
                  11887 \cs_new_protected:Npn \muskip_const:Nn #1#2
                  11888 {
                  11889     \muskip_new:N #1
                  11890     \tex_global:D #1 ~ \muskip_eval:n {#2} \scan_stop:
                  11891 }
                  11892 
```

(End definition for `\muskip_const:Nn`. This function is documented on page 166.)

**\muskip\_zero:N** Reset the register to zero.

```

\muskip_zero:c 11893 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\muskip_gzero:N 11894 \cs_new_protected:Npn \muskip_zero:N #1
\muskip_gzero:c 11895 { #1 \c_zero_muskip }
                  11896 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
                  11897 \cs_new_protected:Npn \muskip_gzero:N #1
                  11898 { \tex_global:D #1 \c_zero_muskip }
                  11899 
```

(End definition for `\muskip_zero:N` and `\muskip_gzero:N`. These functions are documented on page 166.)

**\muskip\_zero\_new:N** Create a register if needed, otherwise clear it.

```

\muskip_zero_new:c 11901 \cs_new_protected:Npn \muskip_zero_new:N #1
\muskip_gzero_new:N 11902 { \muskip_if_exist:NTF #1 { \muskip_zero:N #1 } { \muskip_new:N #1 } }
\muskip_gzero_new:c 11903 \cs_new_protected:Npn \muskip_gzero_new:N #1
                  11904 { \muskip_if_exist:NTF #1 { \muskip_gzero:N #1 } { \muskip_new:N #1 } }
                  11905 
```

(End definition for `\muskip_zero_new:N` and `\muskip_gzero_new:N`. These functions are documented on page 166.)

**\muskip\_if\_exist\_p:N** Copies of the `cs` functions defined in `l3basics`.

```

\muskip_if_exist_p:c 11907 \prg_new_eq_conditional:NNn \muskip_if_exist:N \cs_if_exist:N
\muskip_if_exist:NTF 11908 { TF , T , F , p }
\muskip_if_exist:cTF 11909 \prg_new_eq_conditional:NNn \muskip_if_exist:c \cs_if_exist:c
                  11910 { TF , T , F , p }
```

(End definition for `\muskip_if_exist:NTF`. This function is documented on page 166.)

## 19.21 Setting `muskip` variables

See `skip` case.

```

11911 \cs_set_protected:Npn \__skip_tmp:w #1#2#3
11912 {
11913   \__kernel_patch_args:nnnNNpn
11914   { #1 ##1 }
11915   { }
11916   {
11917     {##1}
11918     {
11919       \__kernel_chk_expr:nNnN {##2}
11920       \tex_muexpr:D { \tex_mutoglu:D } #3
11921     }
11922   }
11923   #2 #3
11924 }
```

`\muskip_set:Nn` This should be pretty familiar.

```

\muskip_set:cn
\muskip_gset:Nn
\muskip_gset:cn
11925 \__skip_tmp:w \__kernel_chk_var_local:N
11926 \cs_new_protected:Npn \muskip_set:Nn #1#2
11927 { #1 ~ \tex_muexpr:D #2 \scan_stop: }
11928 \__skip_tmp:w \__kernel_chk_var_global:N
11929 \cs_new_protected:Npn \muskip_gset:Nn #1#2
11930 { \tex_global:D #1 ~ \tex_muexpr:D #2 \scan_stop: }
11931 \cs_generate_variant:Nn \muskip_set:Nn { c }
11932 \cs_generate_variant:Nn \muskip_gset:Nn { c }
```

(End definition for `\muskip_set:Nn` and `\muskip_gset:Nn`. These functions are documented on page 166.)

`\muskip_set_eq:NN` All straightforward.

```

\muskip_set_eq:cN
\muskip_set_eq:Nc
\muskip_set_eq:cc
\muskip_gset_eq:NN
\muskip_gset_eq:cN
\muskip_gset_eq:Nc
\muskip_gset_eq:cc
11933 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
11934 \cs_new_protected:Npn \muskip_set_eq:NN #1#2 { #1 = #2 }
11935 \cs_generate_variant:Nn \muskip_set_eq:NN { c , Nc , cc }
11936 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
11937 \cs_new_protected:Npn \muskip_gset_eq:NN #1#2 { \tex_global:D #1 = #2 }
11938 \cs_generate_variant:Nn \muskip_gset_eq:NN { c , Nc , cc }
```

(End definition for `\muskip_set_eq:NN` and `\muskip_gset_eq:NN`. These functions are documented on page 166.)

`\muskip_add:Nn` Using `by` here deals with the (incorrect) case `\muskip123`.

```

\muskip_add:cn
\muskip_gadd:Nn
\muskip_gadd:cn
\muskip_sub:Nn
\muskip_sub:cn
\muskip_gsub:Nn
\muskip_gsub:cn
11939 \__skip_tmp:w \__kernel_chk_var_local:N
11940 \cs_new_protected:Npn \muskip_add:Nn #1#2
11941 { \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: }
11942 \__skip_tmp:w \__kernel_chk_var_global:N
11943 \cs_new_protected:Npn \muskip_gadd:Nn #1#2
11944 { \tex_global:D \tex_advance:D #1 by \tex_muexpr:D #2 \scan_stop: }
11945 \cs_generate_variant:Nn \muskip_add:Nn { c }
11946 \cs_generate_variant:Nn \muskip_gadd:Nn { c }
11947 \__skip_tmp:w \__kernel_chk_var_local:N
11948 \cs_new_protected:Npn \muskip_sub:Nn #1#2
11949 { \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: }
11950 \__skip_tmp:w \__kernel_chk_var_global:N
```

```

11951 \cs_new_protected:Npn \muskip_gsub:Nn #1#2
11952   { \tex_global:D \tex_advance:D #1 by - \tex_muexpr:D #2 \scan_stop: }
11953 \cs_generate_variant:Nn \muskip_sub:Nn { c }
11954 \cs_generate_variant:Nn \muskip_gsub:Nn { c }

```

(End definition for `\muskip_add:Nn` and others. These functions are documented on page 166.)

## 19.22 Using muskip expressions and variables

`\muskip_eval:n` Evaluating a muskip expression expandably.

```

11955 \__kernel_patch_args:nNnpn
11956   {
11957     {
11958       \__kernel_chk_expr:nNnN {#1} \tex_muexpr:D
11959         { \tex_mutoglu:D } \muskip_eval:n
11960       }
11961     }
11962 \cs_new:Npn \muskip_eval:n #1
11963   { \muskip_use:N \tex_muexpr:D #1 \scan_stop: }

```

(End definition for `\muskip_eval:n`. This function is documented on page 167.)

`\muskip_use:N` Accessing a *(muskip)*.

```

\muskip_use:c 11964 \cs_new_eq:NN \muskip_use:N \tex_the:D
11965 \cs_generate_variant:Nn \muskip_use:N { c }

```

(End definition for `\muskip_use:N`. This function is documented on page 167.)

## 19.23 Viewing muskip variables

`\muskip_show:N` Diagnostics.

```

\muskip_show:c 11966 \cs_new_eq:NN \muskip_show:N \__kernel_register_show:N
11967 \cs_generate_variant:Nn \muskip_show:N { c }

```

(End definition for `\muskip_show:N`. This function is documented on page 167.)

`\muskip_show:n` Diagnostics. We don't use the TeX primitive `\showthe` to show muskip expressions: this gives a more unified output.

```

11968 \cs_new_protected:Npn \muskip_show:n
11969   { \msg_show_eval:Nn \muskip_eval:n }

```

(End definition for `\muskip_show:n`. This function is documented on page 167.)

`\muskip_log:N` Diagnostics. Redirect output of `\muskip_show:n` to the log.

```

\muskip_log:c 11970 \cs_new_eq:NN \muskip_log:N \__kernel_register_log:N
\muskip_log:n 11971 \cs_new_eq:NN \muskip_log:c \__kernel_register_log:c
11972 \cs_new_protected:Npn \muskip_log:n
11973   { \msg_log_eval:Nn \muskip_eval:n }

```

(End definition for `\muskip_log:N` and `\muskip_log:n`. These functions are documented on page 167.)

## 19.24 Constant muskips

```
\c_zero_muskip Constant muskips given by their value.  
\c_max_muskip  
11974 \muskip_const:Nn \c_zero_muskip { 0 mu }  
11975 \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 168.)
```

## 19.25 Scratch muskips

```
\l_tmpa_muskip We provide two local and two global scratch registers, maybe we need more or less.  
\l_tmpb_muskip  
\g_tmpa_muskip  
\g_tmpb_muskip  
11976 \muskip_new:N \l_tmpa_muskip  
11977 \muskip_new:N \l_tmpb_muskip  
11978 \muskip_new:N \g_tmpa_muskip  
11979 \muskip_new:N \g_tmpb_muskip  
(End definition for \l_tmpa_muskip and others. These variables are documented on page 168.)  
11980 ⟨/initex | package⟩
```

# 20 I3keys Implementation

```
11981 ⟨*initex | package⟩
```

## 20.1 Low-level interface

The low-level key parser is based heavily on `keyval`, but with a number of additional “safety” requirements and with the idea that the parsed list of key–value pairs can be processed in a variety of ways. The net result is that this code needs around twice 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.

```
11982 ⟨@@=keyval⟩
```

```
\l__keyval_key_tl The current key name and value.  
\l__keyval_value_tl  
11983 \tl_new:N \l__keyval_key_tl  
11984 \tl_new:N \l__keyval_value_tl  
(End definition for \l__keyval_key_tl and \l__keyval_value_tl.)  
\l__keyval_sanitise_tl A token list variable for dealing with awkward category codes in the input.  
11985 \tl_new:N \l__keyval_sanitise_tl  
(End definition for \l__keyval_sanitise_tl.)
```

`\keyval_parse:NNn` The main function starts off by normalising category codes in package mode. That’s relatively “expensive” so is skipped (hopefully) in format mode. We then hand off to the parser. The use of `\q_mark` here prevents loss of braces from the key argument. Notice that by passing the two processor commands along the input stack we avoid the need to track these at all.

```
11986 \cs_new_protected:Npn \keyval_parse:NNn #1#2#3  
11987 {  
11988 ⟨*initex⟩  
11989     \__keyval_loop>NNw #1#2 \q_mark #3 , \q_recursion_tail ,
```

```

11990  </initex>
11991  (*package)
11992      \tl_set:Nn \l__keyval_sanitise_tl {#3}
11993      \__keyval_sanitise_equals:
11994      \__keyval_sanitise_comma:
11995      \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
11996      \exp_after:wN \q_mark \l__keyval_sanitise_tl , \q_recursion_tail ,
11997  (/package)
11998  }

```

(End definition for `\keyval_parse:NNn`. This function is documented on page 181.)

`\__keyval_sanitise_equals:`  
`\__keyval_sanitise_comma:`  
`\__keyval_sanitise_equals_auxi:w`  
`\__keyval_sanitise_equals_auxii:w`  
`\__keyval_sanitise_comma_auxi:w`  
`\__keyval_sanitise_comma_auxii:w`  
`\__keyval_sanitise_aux:w`

```

11999 (*package)
12000 \group_begin:
12001     \char_set_catcode_active:n { '= }
12002     \char_set_catcode_active:n { '\, }
12003     \cs_new_protected:Npn \__keyval_sanitise_equals:
12004     {
12005         \exp_after:wN \__keyval_sanitise_equals_auxi:w \l__keyval_sanitise_tl
12006         \q_mark = \q_nil =
12007         \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
12008     }
12009     \cs_new_protected:Npn \__keyval_sanitise_equals_auxi:w #1 =
12010     {
12011         \tl_set:Nn \l__keyval_sanitise_tl {#1}
12012         \__keyval_sanitise_equals_auxii:w
12013     }
12014     \cs_new_protected:Npn \__keyval_sanitise_equals_auxii:w #1 =
12015     {
12016         \if_meaning:w \q_nil #1 \scan_stop:
12017         \else:
12018             \tl_set:Nx \l__keyval_sanitise_tl
12019             {
12020                 \exp_not:o \l__keyval_sanitise_tl
12021                 \token_to_str:N =
12022                 \exp_not:n {#1}
12023             }
12024             \exp_after:wN \__keyval_sanitise_equals_auxii:w
12025         \fi:
12026     }
12027     \cs_new_protected:Npn \__keyval_sanitise_comma:
12028     {
12029         \exp_after:wN \__keyval_sanitise_comma_auxi:w \l__keyval_sanitise_tl
12030         \q_mark , \q_nil ,
12031         \exp_after:wN \__keyval_sanitise_aux:w \l__keyval_sanitise_tl
12032     }
12033     \cs_new_protected:Npn \__keyval_sanitise_comma_auxi:w #1 ,
12034     {
12035         \tl_set:Nn \l__keyval_sanitise_tl {#1}
12036         \__keyval_sanitise_comma_auxii:w
12037     }

```

```

12038     \cs_new_protected:Npn \__keyval_sanitise_comma_auxii:w #1 ,
12039     {
12040         \if_meaning:w \q_nil #1 \scan_stop:
12041         \else:
12042             \tl_set:Nx \l__keyval_sanitise_tl
12043             {
12044                 \exp_not:o \l__keyval_sanitise_tl
12045                 \token_to_str:N ,
12046                 \exp_not:n {#1}
12047             }
12048             \exp_after:wN \__keyval_sanitise_comma_auxii:w
12049             \fi:
12050         }
12051     \group_end:
12052     \cs_new_protected:Npn \__keyval_sanitise_aux:w #1 \q_mark
12053     { \tl_set:Nn \l__keyval_sanitise_tl {#1} }
12054 
```

(End definition for `\__keyval_sanitise_equals:` and others.)

`\__keyval_loop>NNw`

A fast test for the end of the loop, remembering to remove the leading quark first. Assuming that is not the case, look for a key and value then loop around, re-inserting a leading quark in front of the next position.

```

12055     \cs_new_protected:Npn \__keyval_loop:NNw #1#2#3 ,
12056     {
12057         \exp_after:wN \if_meaning:w \exp_after:wN \q_recursion_tail
12058         \use_none:n #3 \prg_do_nothing:
12059         \else:
12060             \__keyval_split>NNw #1#2#3 == \q_stop
12061             \exp_after:wN \__keyval_loop:NNw \exp_after:wN #1 \exp_after:wN #2
12062             \exp_after:wN \q_mark
12063         \fi:
12064     }

```

(End definition for `\__keyval_loop>NNw`.)

`\__keyval_split>NNw`

`\__keyval_split_value>NNw`

`\__keyval_split_tidy:w`

`\__keyval_action:`

The value is picked up separately from the key so there can be another quark inserted at the front, keeping braces and allowing both parts to share the same code paths. The key is found first then there's a check that there is something there: this is biased to the common case of there actually being a key. For the value, we first need to see if there is anything to do: if there is, extract it. The appropriate action is then inserted in front of the key and value. Doing this using an assignment is marginally faster than an an expansion chain.

```

12065     \cs_new_protected:Npn \__keyval_split>NNw #1#2#3 =
12066     {
12067         \__keyval_def:Nn \l__keyval_key_tl {#3}
12068         \if_meaning:w \l__keyval_key_tl \c_empty_tl
12069             \exp_after:wN \__keyval_split_tidy:w
12070         \else:
12071             \exp_after:wN \__keyval_split_value>NNw
12072             \exp_after:wN #1
12073             \exp_after:wN #2
12074             \exp_after:wN \q_mark
12075         \fi:

```

```

12076    }
12077 \cs_new_protected:Npn \__keyval_split_value:NNw #1#2#3 = #4 \q_stop
12078 {
12079     \if:w \scan_stop: \tl_to_str:n {#4} \scan_stop:
12080         \cs_set:Npx \__keyval_action:
12081             { \exp_not:N #1 { \exp_not:o \l__keyval_key_tl } }
12082     \else:
12083         \if:w
12084             \scan_stop:
12085             \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #4 }
12086             \scan_stop:
12087             \__keyval_def:Nn \l__keyval_value_tl {#3}
12088             \cs_set:Npx \__keyval_action:
12089                 {
12090                     \exp_not:N #2
12091                     { \exp_not:o \l__keyval_key_tl }
12092                     { \exp_not:o \l__keyval_value_tl }
12093                 }
12094     \else:
12095         \cs_set:Npn \__keyval_action:
12096             {
12097                 \__kernel_msg_error:nn { kernel }
12098                 { misplaced-equals-sign }
12099             }
12100     \fi:
12101 \fi:
12102     \__keyval_action:
12103 }
12104 \cs_new_protected:Npn \__keyval_split_tidy:w #1 \q_stop
12105 {
12106     \if:w
12107         \scan_stop:
12108         \__kernel_tl_to_str:w \exp_after:wN { \use_none:n #1 }
12109         \scan_stop:
12110     \else:
12111         \exp_after:wN \__keyval_empty_key:
12112     \fi:
12113 }
12114 \cs_new:Npn \__keyval_action: { }
12115 \cs_new_protected:Npn \__keyval_empty_key:
12116     { \__kernel_msg_error:nn { kernel } { misplaced-equals-sign } }

(End definition for \__keyval_split:NNw and others.)

```

\\_\_keyval\_def:Nn First remove the leading quark, then trim spaces off, and finally remove a set of braces.

```

\__keyval_def_aux:n
\__keyval_def_aux:w
12117 \cs_new_protected:Npn \__keyval_def:Nn #1#2
12118 {
12119     \tl_set:Nx #1
12120     { \tl_trim_spaces_apply:oN { \use_none:n #2 } \__keyval_def_aux:n }
12121 }
12122 \cs_new:Npn \__keyval_def_aux:n #1
12123     { \__keyval_def_aux:w #1 \q_stop }
12124 \cs_new:Npn \__keyval_def_aux:w #1 \q_stop { \exp_not:n {#1} }

(End definition for \__keyval_def:Nn, \__keyval_def_aux:n, and \__keyval_def_aux:w.)

```

One message for the low level parsing system.

```
12125 \__kernel_msg_new:nnnn { kernel } { misplaced-equals-sign }
12126   { Misplaced~equals~sign~in~key~value~input~\msg_line_number: }
12127   {
12128     LaTeX~is~attempting~to~parse~some~key~value~input~but~found~
12129     two~equals~signs~not~separated~by~a~comma.
12130 }
```

## 20.2 Constants and variables

12131 ⟨@@=keys⟩

Various storage areas for the different data which make up keys.

```
12132 \tl_const:Nn \c_keys_code_root_tl { key~code~>~ }
12133 \tl_const:Nn \c_keys_default_root_tl { key~default~>~ }
12134 \tl_const:Nn \c_keys_groups_root_tl { key~groups~>~ }
12135 \tl_const:Nn \c_keys_inherit_root_tl { key~inherit~>~ }
12136 \tl_const:Nn \c_keys_type_root_tl { key~type~>~ }
12137 \tl_const:Nn \c_keys_validate_root_tl { key~validate~>~ }
```

(*End definition for \c\_keys\_code\_root\_tl and others.*)

\c\_keys\_props\_root\_tl The prefix for storing properties.

```
12138 \tl_const:Nn \c_keys_props_root_tl { key~prop~>~ }
```

(*End definition for \c\_keys\_props\_root\_tl.*)

\l\_keys\_choice\_int Publicly accessible data on which choice is being used when several are generated as a set.  
\l\_keys\_choice\_tl

```
12139 \int_new:N \l_keys_choice_int
12140 \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 175.*)

\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.

```
12141 \clist_new:N \l_keys_groups_clist
```

(*End definition for \l\_keys\_groups\_clist.*)

\l\_keys\_key\_tl The name of a key itself: needed when setting keys.

```
12142 \tl_new:N \l_keys_key_tl
```

(*End definition for \l\_keys\_key\_tl. This variable is documented on page 177.*)

\l\_keys\_module\_tl The module for an entire set of keys.

```
12143 \tl_new:N \l_keys_module_tl
```

(*End definition for \l\_keys\_module\_tl.*)

\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.

```
12144 \bool_new:N \l_keys_no_value_bool
```

(*End definition for \l\_keys\_no\_value\_bool.*)

<code>\l__keys_only_known_bool</code>	Used to track if only “known” keys are being set.  <code>12145 \bool_new:N \l__keys_only_known_bool</code>  <i>(End definition for <code>\l__keys_only_known_bool</code>.)</i>
<code>\l_keys_path_tl</code>	The “path” of the current key is stored here: this is available to the programmer and so is public.  <code>12146 \tl_new:N \l_keys_path_tl</code>  <i>(End definition for <code>\l_keys_path_tl</code>. This variable is documented on page 177.)</i>
<code>\l__keys_inherit_tl</code>	 <code>12147 \tl_new:N \l__keys_inherit_tl</code>  <i>(End definition for <code>\l__keys_inherit_tl</code>.)</i>
<code>\l__keys_property_tl</code>	The “property” begin set for a key at definition time is stored here.  <code>12148 \tl_new:N \l__keys_property_tl</code>  <i>(End definition for <code>\l__keys_property_tl</code>.)</i>
<code>\l__keys_selective_bool</code> <code>\l__keys_filtered_bool</code>	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”).  <code>12149 \bool_new:N \l__keys_selective_bool</code> <code>12150 \bool_new:N \l__keys_filtered_bool</code>  <i>(End definition for <code>\l__keys_selective_bool</code> and <code>\l__keys_filtered_bool</code>.)</i>
<code>\l__keys_selective_seq</code>	The list of key groups being filtered in or out during selective setting.  <code>12151 \seq_new:N \l__keys_selective_seq</code>  <i>(End definition for <code>\l__keys_selective_seq</code>.)</i>
<code>\l__keys_unused_clist</code>	Used when setting only some keys to store those left over.  <code>12152 \tl_new:N \l__keys_unused_clist</code>  <i>(End definition for <code>\l__keys_unused_clist</code>.)</i>
<code>\l_keys_value_tl</code>	The value given for a key: may be empty if no value was given.  <code>12153 \tl_new:N \l_keys_value_tl</code>  <i>(End definition for <code>\l_keys_value_tl</code>. This variable is documented on page 177.)</i>
<code>\l__keys_tmp_bool</code>	Scratch space.  <code>12154 \bool_new:N \l__keys_tmp_bool</code>  <i>(End definition for <code>\l__keys_tmp_bool</code>.)</i>

### 20.3 The key defining mechanism

\keys\_define:nn  
\\_keys\_define:nnn  
\\_keys\_define:onn

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).

```

12155 \cs_new_protected:Npn \keys_define:nn
12156   { \_keys_define:onn \l__keys_module_tl }
12157 \cs_new_protected:Npn \_keys_define:nnn #1#2#3
12158   {
12159     \tl_set:Nx \l__keys_module_tl { \_keys_remove_spaces:n {#2} }
12160     \keyval_parse:NNn \_keys_define:n \_keys_define:nn {#3}
12161     \tl_set:Nn \l__keys_module_tl {#1}
12162   }
12163 \cs_generate_variant:Nn \_keys_define:nnn { o }
```

(End definition for \keys\_define:nn and \\_keys\_define:nnn. This function is documented on page 170.)

\\_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.

```

12164 \cs_new_protected:Npn \_keys_define:n #1
12165   {
12166     \bool_set_true:N \l__keys_no_value_bool
12167     \_keys_define_aux:nn {#1} { }
12168   }
12169 \cs_new_protected:Npn \_keys_define:nn #1#2
12170   {
12171     \bool_set_false:N \l__keys_no_value_bool
12172     \_keys_define_aux:nn {#1} {#2}
12173   }
12174 \cs_new_protected:Npn \_keys_define_aux:nn #1#2
12175   {
12176     \_keys_property_find:n {#1}
12177     \cs_if_exist:cTF { \c__keys_props_root_tl \l__keys_property_tl }
12178       { \_keys_define_code:n {#2} }
12179       {
12180         \tl_if_empty:NF \l__keys_property_tl
12181           {
12182             \_kernel_msg_error:nxxx { kernel } { key-property-unknown }
12183             { \l__keys_property_tl } { \l__keys_path_tl }
12184           }
12185       }
12186   }
```

(End definition for \\_keys\_define:n, \\_keys\_define:nn, and \\_keys\_define\_aux:nn.)

\\_keys\_property\_find:n  
\\_keys\_property\_find: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.

```

12187 \cs_new_protected:Npn \_keys_property_find:n #1
12188   {
12189     \tl_set:Nx \l__keys_property_tl { \_keys_remove_spaces:n {#1} }
12190     \exp_after:wN \_keys_property_find:w \l__keys_property_tl . .
```

```

12191      \q_stop {#1}
12192    }
12193 \cs_new_protected:Npn \__keys_property_find:w #1 . #2 . #3 \q_stop #4
12194  {
12195    \tl_if_blank:nTF {#3}
12196    {
12197      \tl_clear:N \l__keys_property_tl
12198      \__kernel_msg_error:nnn { kernel } { key-no-property } {#4}
12199    }
12200    {
12201      \str_if_eq:nnTF {#3} { . }
12202      {
12203        \tl_set:Nx \l__keys_path_tl
12204        {
12205          \tl_if_empty:NF \l__keys_module_tl
12206            { \l__keys_module_tl / }
12207            #1
12208        }
12209        \tl_set:Nn \l__keys_property_tl { . #2 }
12210      }
12211      {
12212        \tl_set:Nx \l__keys_path_tl { \l__keys_module_tl / #1 . #2 }
12213        \__keys_property_search:w #3 \q_stop
12214      }
12215    }
12216  }
12217 \cs_new_protected:Npn \__keys_property_search:w #1 . #2 \q_stop
12218  {
12219    \str_if_eq:nnTF {#2} { . }
12220    {
12221      \tl_set:Nx \l__keys_path_tl { \l__keys_path_tl }
12222      \tl_set:Nn \l__keys_property_tl { . #1 }
12223    }
12224    {
12225      \tl_set:Nx \l__keys_path_tl { \l__keys_path_tl . #1 }
12226      \__keys_property_search:w #2 \q_stop
12227    }
12228  }

```

(End definition for `\__keys_property_find:n` and `\__keys_property_find:w`.)

`\__keys_define_code:n` 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.

```

12229 \cs_new_protected:Npn \__keys_define_code:n #1
12230  {
12231    \bool_if:NTF \l__keys_no_value_bool
12232    {
12233      \exp_after:wN \__keys_define_code:w
12234      \l__keys_property_tl \q_stop
12235      { \use:c { \c__keys_props_root_tl \l__keys_property_tl } }
12236    {
12237      \__kernel_msg_error:nnxx { kernel }

```

```

12238          { key-property-requires-value } { \l__keys_property_tl }
12239          { \l_keys_path_tl }
12240      }
12241      }
12242      { \use:c { \c__keys_props_root_tl \l__keys_property_tl } {#1} }
12243  }
12244 \exp_last_unbraced:NNNN
12245   \cs_new:Npn \__keys_define_code:w #1 \cColonStr #2 \q_stop
12246   { \tl_if_empty:nTF {#2} }

```

(End definition for `\__keys_define_code:n` and `\__keys_define_code:w`.)

## 20.4 Turning properties into actions

`\__keys_bool_set:Nn` Boolean keys are really just choices, but all done by hand. The second argument here is the scope: either empty or `g` for global.

```

12247 \cs_new_protected:Npn \__keys_bool_set:Nn #1#2
12248  {
12249    \bool_if_exist:NF #1 { \bool_new:N #1 }
12250    \__keys_choice_make:
12251    \__keys_cmd_set:nx { \l_keys_path_tl / true }
12252    { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
12253    \__keys_cmd_set:nx { \l_keys_path_tl / false }
12254    { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
12255    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
12256    {
12257      \__kernel_msg_error:nnx { kernel } { boolean-values-only }
12258      { \l_keys_key_tl }
12259    }
12260    \__keys_default_set:n { true }
12261  }
12262 \cs_generate_variant:Nn \__keys_bool_set:Nn { c }

```

(End definition for `\__keys_bool_set:Nn`.)

`\__keys_bool_set_inverse:Nn` Inverse boolean setting is much the same.

```

12263 \cs_new_protected:Npn \__keys_bool_set_inverse:Nn #1#2
12264  {
12265    \bool_if_exist:NF #1 { \bool_new:N #1 }
12266    \__keys_choice_make:
12267    \__keys_cmd_set:nx { \l_keys_path_tl / true }
12268    { \exp_not:c { bool_ #2 set_false:N } \exp_not:N #1 }
12269    \__keys_cmd_set:nx { \l_keys_path_tl / false }
12270    { \exp_not:c { bool_ #2 set_true:N } \exp_not:N #1 }
12271    \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
12272    {
12273      \__kernel_msg_error:nnx { kernel } { boolean-values-only }
12274      { \l_keys_key_tl }
12275    }
12276    \__keys_default_set:n { true }
12277  }
12278 \cs_generate_variant:Nn \__keys_bool_set_inverse:Nn { c }

```

(End definition for `\__keys_bool_set_inverse:Nn`.)

`\__keys_choice_make:`  
`\__keys_multichoice_make:`  
`\__keys_choice_make:N`  
`\__keys_choice_make_aux:N`

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.

```

12279 \cs_new_protected:Npn \__keys_choice_make:
1280   { \__keys_choice_make:N \__keys_choice_find:n }
1281 \cs_new_protected:Npn \__keys_multichoice_make:
1282   { \__keys_choice_make:N \__keys_multichoice_find:n }
1283 \cs_new_protected:Npn \__keys_choice_make:N #1
1284   {
1285     \cs_if_exist:cTF
1286       { \c__keys_type_root_tl \__keys_parent:o \l_keys_path_tl }
1287       {
1288         \str_if_eq:vnTF
1289           { \c__keys_type_root_tl \__keys_parent:o \l_keys_path_tl }
1290           { choice }
1291           {
1292             \__kernel_msg_error:nxxx { kernel } { nested-choice-key }
1293               { \l_keys_path_tl } { \__keys_parent:o \l_keys_path_tl }
1294             }
1295             { \__keys_choice_make_aux:N #1 }
1296           }
1297           { \__keys_choice_make_aux:N #1 }
1298       }
1299 \cs_new_protected:Npn \__keys_choice_make_aux:N #1
1300   {
1301     \cs_set_nopar:cpn { \c__keys_type_root_tl \l_keys_path_tl }
1302     { choice }
1303     \__keys_cmd_set:nn { \l_keys_path_tl } { #1 {##1} }
1304     \__keys_cmd_set:nn { \l_keys_path_tl / unknown }
1305     {
1306       \__kernel_msg_error:nxxx { kernel } { key-choice-unknown }
1307         { \l_keys_path_tl } {##1}
1308     }
1309   }

```

(End definition for `\__keys_choice_make:` and others.)

`\__keys_choices_make:nn`  
`\__keys_multichoices_make:nn`  
`\__keys_choices_make:Nnn`

Auto-generating choices means setting up the root key as a choice, then defining each choice in turn.

```

12310 \cs_new_protected:Npn \__keys_choices_make:nn
12311   { \__keys_choices_make:Nnn \__keys_choice_make: }
12312 \cs_new_protected:Npn \__keys_multichoices_make:nn
12313   { \__keys_choices_make:Nnn \__keys_multichoice_make: }
12314 \cs_new_protected:Npn \__keys_choices_make:Nnn #1#2#3
12315   {
12316     #1
12317     \int_zero:N \l_keys_choice_int
12318     \clist_map_inline:nn {#2}
12319     {
12320       \int_incr:N \l_keys_choice_int
12321       \__keys_cmd_set:nx
12322         { \l_keys_path_tl / \__keys_remove_spaces:n {##1} }
12323         {
12324           \tl_set:Nn \exp_not:N \l_keys_choice_tl {##1}

```

```

12325     \int_set:Nn \exp_not:N \l_keys_choice_int
12326         { \int_use:N \l_keys_choice_int }
12327     \exp_not:n {#3}
12328 }
12329 }
12330 }
```

(End definition for `\__keys_choices_make:nn`, `\__keys_multichoices_make:nn`, and `\__keys_choices-make:Nnn`.)

`\__keys_cmd_set:nn` Setting the code for a key first logs if appropriate that we are defining a new key, then saves the code.

```

12331 \__kernel_patch:nnNNpn
12332 {
12333     \cs_if_exist:cF { \c_keys_code_root_tl #1 }
12334         { \__kernel_debug_log:x { Defining~key~#1~\msg_line_context: } }
12335     }
12336     { }
12337 \cs_new_protected:Npn \__keys_cmd_set:nn #1#2
12338     { \cs_set_protected:cpn { \c_keys_code_root_tl #1 } ##1 {#2} }
12339 \cs_generate_variant:Nn \__keys_cmd_set:nn { nx , Vn , Vo }
```

(End definition for `\__keys_cmd_set:nn`.)

`\__keys_default_set:n` Setting a default value is easy. These are stored using `\cs_set:cpx` as this avoids any worries about whether a token list exists.

```

12340 \cs_new_protected:Npn \__keys_default_set:n #1
12341 {
12342     \tl_if_empty:nTF {#1}
12343     {
12344         \cs_set_eq:cN
12345             { \c_keys_default_root_tl \l_keys_path_tl }
12346             \tex_undefined:D
12347     }
12348     {
12349         \cs_set:cpx
12350             { \c_keys_default_root_tl \l_keys_path_tl }
12351             { \exp_not:n {#1} }
12352     }
12353 }
```

(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.

```

12354 \cs_new_protected:Npn \__keys_groups_set:n #1
12355 {
12356     \clist_set:Nn \l__keys_groups_clist {#1}
12357     \clist_if_empty:NTF \l__keys_groups_clist
12358     {
12359         \cs_set_eq:cN { \c_keys_groups_root_tl \l_keys_path_tl }
12360         \tex_undefined:D
```

```

12361      }
12362      {
12363          \cs_set_eq:cN { \c_keys_groups_root_tl } \l_keys_path_tl }
12364          \l_keys_groups_clist
12365      }
12366  }

```

(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.

```

12367  \cs_new_protected:Npn \_keys_inherit:n #1
12368  {
12369      \_keys_undefine:
12370      \cs_set_nopar:cpn { \c_keys_inherit_root_tl } \l_keys_path_tl } {#1}
12371  }

```

(End definition for `\_keys_inherit:n`.)

`\_keys_initialise:n` A set up for initialisation: just run the code if it exists.

```

12372  \cs_new_protected:Npn \_keys_initialise:n #1
12373  {
12374      \cs_if_exist_use:cT { \c_keys_code_root_tl } \l_keys_path_tl } { {#1} }
12375  }

```

(End definition for `\_keys_initialise:n`.)

`\_keys_meta_make:n` To create a meta-key, simply set up to pass data through.

```

\_keys_meta_make:nn
12376  \cs_new_protected:Npn \_keys_meta_make:n #1
12377  {
12378      \_keys_cmd_set:Vo \l_keys_path_tl
12379      {
12380          \exp_after:wN \keys_set:nn
12381          \exp_after:wN { \l_keys_module_tl } {#1}
12382      }
12383  }
12384  \cs_new_protected:Npn \_keys_meta_make:nn #1#2
12385  { \_keys_cmd_set:Vn \l_keys_path_tl { \keys_set:nn {#1} {#2} } }

```

(End definition for `\_keys_meta_make:n` and `\_keys_meta_make:nn`.)

`\_keys_undefine:` Undefining a key has to be done without `\cs_undefine:c` as that function acts globally.

```

12386  \cs_new_protected:Npn \_keys_undefine:
12387  {
12388      \clist_map_inline:nn
12389      { code , default , groups , inherit , type , validate }
12390      {
12391          \cs_set_eq:cN
12392          { \tl_use:c { \c_keys_ ##1 _root_tl } } \l_keys_path_tl }
12393          \tex_undefined:D
12394      }
12395  }

```

(End definition for `\_keys_undefine:..`)

\\_\\_keys\\_value\\_requirement:nn  
\\_\\_keys\\_validate\\_forbidden:  
\\_\\_keys\\_validate\\_required:  
\\_\\_keys\\_validate\\_cleanup:w

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.

```

12396 \cs_new_protected:Npn \_\_keys_value_requirement:nn #1#2
12397 {
12398     \str_case:nnF {#2}
12399     {
12400         { true }
12401         {
12402             \cs_set_eq:cc
12403             { \c_\_keys_validate_root_tl \l_\_keys_path_tl }
12404             { \_\_keys_validate_ #1 : }
12405         }
12406         { false }
12407         {
12408             \cs_if_eq:ccT
12409             { \c_\_keys_validate_root_tl \l_\_keys_path_tl }
12410             { \_\_keys_validate_ #1 : }
12411             {
12412                 \cs_set_eq:cN
12413                 { \c_\_keys_validate_root_tl \l_\_keys_path_tl }
12414                 \tex_undefined:D
12415             }
12416         }
12417     }
12418     {
12419         \_\_kernel_msg_error:nnx { kernel }
12420         { key-property-boolean-values-only }
12421         { .value_ #1 :n }
12422     }
12423 }
12424 \cs_new_protected:Npn \_\_keys_validate_forbidden:
12425 {
12426     \bool_if:NF \l_\_keys_no_value_bool
12427     {
12428         \_\_kernel_msg_error:nnxx { kernel } { value-forbidden }
12429         { \l_\_keys_path_tl } { \l_\_keys_value_tl }
12430         \_\_keys_validate_cleanup:w
12431     }
12432 }
12433 \cs_new_protected:Npn \_\_keys_validate_required:
12434 {
12435     \bool_if:NT \l_\_keys_no_value_bool
12436     {
12437         \_\_kernel_msg_error:nnx { kernel } { value-required }
12438         { \l_\_keys_path_tl }
12439         \_\_keys_validate_cleanup:w
12440     }
12441 }
12442 \cs_new_protected:Npn \_\_keys_validate_cleanup:w #1 \cs_end: #2#3 { }
```

(End definition for \\_\\_keys\_value\_requirement:nn and others.)

\\_\\_keys\\_variable\\_set:NnnN  
\\_\\_keys\\_variable\\_set:cnnN Setting a variable takes the type and scope separately so that it is easy to make a new variable if needed.

```

12443 \cs_new_protected:Npn \_\_keys_variable_set:NnnN #1#2#3#4
12444 {
12445   \use:c { #2_if_exist:NF } #1 { \use:c { #2 _new:N } #1 }
12446   \_\_keys_cmd_set:nx { \l_keys_path_tl }
12447   {
12448     \exp_not:c { #2 _ #3 set:N #4 }
12449     \exp_not:N #1
12450     \exp_not:n { ##1 }
12451   }
12452 }
12453 \cs_generate_variant:Nn \_\_keys_variable_set:NnnN { c }
```

(End definition for \\_\\_keys\\_variable\\_set:NnnN.)

## 20.5 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.  
.bool\\_set:c  
12454 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_set:N } #1  
.bool\_gset:N  
12455 { \\_\\_keys\_bool\_set:Nn #1 { } }  
.bool\_gset:c  
12456 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_set:c } #1  
12457 { \\_\\_keys\_bool\_set:cn {#1} { } }  
12458 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_gset:N } #1  
12459 { \\_\\_keys\_bool\_set:Nn #1 { g } }  
12460 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_gset:c } #1  
12461 { \\_\\_keys\_bool\_set:cn {#1} { g } }

(End definition for .bool\\_set:N and .bool\\_gset:N. These functions are documented on page 171.)

.bool\\_set\\_inverse:N One function for this.  
.bool\\_set\\_inverse:c  
12462 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_set\_inverse:N } #1  
.bool\_gset\_inverse:N  
12463 { \\_\\_keys\_bool\_set\_inverse:Nn #1 { } }  
.bool\_gset\_inverse:c  
12464 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_set\_inverse:c } #1  
12465 { \\_\\_keys\_bool\_set\_inverse:cn {#1} { } }  
12466 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_gset\_inverse:N } #1  
12467 { \\_\\_keys\_bool\_set\_inverse:Nn #1 { g } }  
12468 \cs\_new\_protected:cpn { \c\\_keys\_props\_root\_tl .bool\_gset\_inverse:c } #1  
12469 { \\_\\_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 171.)

.choice: Making a choice is handled internally, as it is also needed by .generate\_choices:n.

```

12470 \cs_new_protected:cpn { \c\_keys_props_root_tl .choice: }
12471 { \_\_keys_choice_make: }
```

(End definition for `.choice::`. This function is documented on page 171.)

**.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.

```
12472 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:nn } #1
12473   { \__keys_choices_make:nn #1 }
12474 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:Vn } #1
12475   { \exp_args:NV \__keys_choices_make:nn #1 }
12476 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:on } #1
12477   { \exp_args:No \__keys_choices_make:nn #1 }
12478 \cs_new_protected:cpn { \c__keys_props_root_tl .choices:xn } #1
12479   { \exp_args:Nx \__keys_choices_make:nn #1 }
```

(End definition for `.choices:nn`. This function is documented on page 171.)

**.code:n** Creating code is simply a case of passing through to the underlying `set` function.

```
12480 \cs_new_protected:cpn { \c__keys_props_root_tl .code:n } #1
12481   { \__keys_cmd_set:nn { \l_keys_path_tl } {#1} }
```

(End definition for `.code:n`. This function is documented on page 171.)

**.clist\_set:N**  
**.clist\_set:c**  
**.clist\_gset:N**  
**.clist\_gset:c**

```
12482 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:N } #1
12483   { \__keys_variable_set:NnnN #1 { clist } { } n }
12484 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_set:c } #1
12485   { \__keys_variable_set:cnnN {#1} { clist } { } n }
12486 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:N } #1
12487   { \__keys_variable_set:NnnN #1 { clist } { g } n }
12488 \cs_new_protected:cpn { \c__keys_props_root_tl .clist_gset:c } #1
12489   { \__keys_variable_set:cnnN {#1} { clist } { g } n }
```

(End definition for `.clist_set:N` and `.clist_gset:N`. These functions are documented on page 171.)

**.default:n** Expansion is left to the internal functions.

```
12490 \cs_new_protected:cpn { \c__keys_props_root_tl .default:n } #1
12491   { \__keys_default_set:n {#1} }
12492 \cs_new_protected:cpn { \c__keys_props_root_tl .default:V } #1
12493   { \exp_args:NV \__keys_default_set:n #1 }
12494 \cs_new_protected:cpn { \c__keys_props_root_tl .default:o } #1
12495   { \exp_args:No \__keys_default_set:n {#1} }
12496 \cs_new_protected:cpn { \c__keys_props_root_tl .default:x } #1
12497   { \exp_args:Nx \__keys_default_set:n {#1} }
```

(End definition for `.default:n`. This function is documented on page 172.)

**.dim\_set:N** Setting a variable is very easy: just pass the data along.

```
12498 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:N } #1
12499   { \__keys_variable_set:NnnN #1 { dim } { } n }
12500 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_set:c } #1
12501   { \__keys_variable_set:cnnN {#1} { dim } { } n }
12502 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:N } #1
12503   { \__keys_variable_set:NnnN #1 { dim } { g } n }
12504 \cs_new_protected:cpn { \c__keys_props_root_tl .dim_gset:c } #1
12505   { \__keys_variable_set:cnnN {#1} { dim } { g } n }
```

(End definition for `.dim_set:N` and `.dim_gset:N`. These functions are documented on page 172.)

**`.fp_set:N`** Setting a variable is very easy: just pass the data along.

```
12506 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_set:N } #1
. .fp_set:c   { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:N   { \__keys_variable_set:NnnN #1 { fp } { } n }
.fp_gset:c   { \__keys_variable_set:cnnN {#1} { fp } { } n }
12509   { \__keys_variable_set:cnnN {#1} { fp } { } n }
12510 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:N } #1
12511   { \__keys_variable_set:NnnN #1 { fp } { g } n }
12512 \cs_new_protected:cpn { \c__keys_props_root_tl .fp_gset:c } #1
12513   { \__keys_variable_set:cnnN {#1} { fp } { g } n }
```

(End definition for `.fp_set:N` and `.fp_gset:N`. These functions are documented on page 172.)

**`.groups:n`** A single property to create groups of keys.

```
12514 \cs_new_protected:cpn { \c__keys_props_root_tl .groups:n } #1
12515   { \__keys_groups_set:n {#1} }
```

(End definition for `.groups:n`. This function is documented on page 172.)

**`.inherit:n`** Nothing complex: only one variant at the moment!

```
12516 \cs_new_protected:cpn { \c__keys_props_root_tl .inherit:n } #1
12517   { \__keys_inherit:n {#1} }
```

(End definition for `.inherit:n`. This function is documented on page 172.)

**`.initial:n`** The standard hand-off approach.

```
12518 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:n } #1
. .initial:V   { \__keys_initialise:n {#1} }
.initial:o   { \__keys_initialise:n {#1} }
.initial:x   { \__keys_initialise:n {#1} }
12520 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:V } #1
12521   { \exp_args:NV \__keys_initialise:n {#1} }
12522 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:o } #1
12523   { \exp_args:No \__keys_initialise:n {#1} }
12524 \cs_new_protected:cpn { \c__keys_props_root_tl .initial:x } #1
12525   { \exp_args:Nx \__keys_initialise:n {#1} }
```

(End definition for `.initial:n`. This function is documented on page 173.)

**`.int_set:N`** Setting a variable is very easy: just pass the data along.

```
12526 \cs_new_protected:cpn { \c__keys_props_root_tl .int_set:N } #1
. .int_set:c   { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:N   { \__keys_variable_set:NnnN #1 { int } { } n }
.int_gset:c   { \__keys_variable_set:cnnN {#1} { int } { } n }
12529   { \__keys_variable_set:cnnN {#1} { int } { } n }
12530 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:N } #1
12531   { \__keys_variable_set:NnnN #1 { int } { g } n }
12532 \cs_new_protected:cpn { \c__keys_props_root_tl .int_gset:c } #1
12533   { \__keys_variable_set:cnnN {#1} { int } { g } n }
```

(End definition for `.int_set:N` and `.int_gset:N`. These functions are documented on page 173.)

**`.meta:n`** Making a meta is handled internally.

```
12534 \cs_new_protected:cpn { \c__keys_props_root_tl .meta:n } #1
12535   { \__keys_meta_make:n {#1} }
```

(End definition for `.meta:n`. This function is documented on page 173.)

**.meta:nn** Meta with path: potentially lots of variants, but for the moment no so many defined.

```
12536 \cs_new_protected:cpn { \c_keys_props_root_tl .meta:nn } #1
12537   { \_keys_meta:nn #1 }
```

(End definition for **.meta:nn**. This function is documented on page [173](#).)

**.multichoice:** The same idea as **.choice:** and **.choices:nn**, but where more than one choice is allowed.

```
12538 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice: }
12539   { \_keys_multichoice_make: }
12540 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:nn } #1
12541   { \_keys_multichoice:nn #1 }
12542 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:Vn } #1
12543   { \exp_args:N \_keys_multichoice:nn #1 }
12544 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:on } #1
12545   { \exp_args:No \_keys_multichoice:nn #1 }
12546 \cs_new_protected:cpn { \c_keys_props_root_tl .multichoice:xn } #1
12547   { \exp_args:Nx \_keys_multichoice:nn #1 }
```

(End definition for **.multichoice:** and **.multichoice:nn**. These functions are documented on page [173](#).)

**.skip\_set:N** Setting a variable is very easy: just pass the data along.

```
12548 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_set:N } #1
12549   { \_keys_variable_set:NnnN #1 { skip } { } n }
12550 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_set:c } #1
12551   { \_keys_variable_set:cnnN {#1} { skip } { } n }
12552 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_gset:N } #1
12553   { \_keys_variable_set:NnnN #1 { skip } { g } n }
12554 \cs_new_protected:cpn { \c_keys_props_root_tl .skip_gset:c } #1
12555   { \_keys_variable_set:cnnN {#1} { skip } { g } n }
```

(End definition for **.skip\_set:N** and **.skip\_gset:N**. These functions are documented on page [173](#).)

**.tl\_set:N** Setting a variable is very easy: just pass the data along.

```
12556 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set:N } #1
12557   { \_keys_variable_set:NnnN #1 { tl } { } n }
12558 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set:c } #1
12559   { \_keys_variable_set:cnnN {#1} { tl } { } n }
12560 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set_x:N } #1
12561   { \_keys_variable_set:NnnN #1 { tl } { } x }
12562 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_set_x:c } #1
12563   { \_keys_variable_set:cnnN {#1} { tl } { } x }
12564 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset:N } #1
12565   { \_keys_variable_set:NnnN #1 { tl } { g } n }
12566 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset:c } #1
12567   { \_keys_variable_set:cnnN {#1} { tl } { g } n }
12568 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset_x:N } #1
12569   { \_keys_variable_set:NnnN #1 { tl } { g } x }
12570 \cs_new_protected:cpn { \c_keys_props_root_tl .tl_gset_x:c } #1
12571   { \_keys_variable_set:cnnN {#1} { tl } { g } x }
```

(End definition for **.tl\_set:N** and others. These functions are documented on page [173](#).)

**.undefine:** Another simple wrapper.

```
12572 \cs_new_protected:cpn { \c_keys_props_root_tl .undefine: }
12573   { \_keys_undefine: }
```

(End definition for `.undefined`. This function is documented on page 174.)

### `.value_forbidden:n`

### `.value_required:n`

```
12574 \cs_new_protected:cpn { \c__keys_props_root_tl .value_forbidden:n } #1
12575   { \__keys_value_requirement:nn { forbidden } {#1} }
12576 \cs_new_protected:cpn { \c__keys_props_root_tl .value_required:n } #1
12577   { \__keys_value_requirement:nn { required } {#1} }
```

(End definition for `.value_forbidden:n` and `.value_required:n`. These functions are documented on page 174.)

## 20.6 Setting keys

### `\keys_set:nn`

```
\keys_set:nV
\keys_set:nv
\keys_set:no
\__keys_set:nnn
\__keys_set:onn
12578 \cs_new_protected:Npn \keys_set:nn
12579   { \__keys_set:onn { \l__keys_module_tl } }
12580 \cs_new_protected:Npn \__keys_set:nnn #1#2#3
12581   {
12582     \tl_set:Nx \l__keys_module_tl { \__keys_remove_spaces:n {#2} }
12583     \keyval_parse:NNn \__keys_set:n \__keys_set:nn {#3}
12584     \tl_set:Nn \l__keys_module_tl {#1}
12585   }
12586 \cs_generate_variant:Nn \keys_set:nn { nV , nv , no }
12587 \cs_generate_variant:Nn \__keys_set:nnn { o }
```

(End definition for `\keys_set:nn` and `\__keys_set:nnn`. This function is documented on page 177.)

### `\keys_set_known:nnN`

### `\keys_set_known:nVN`

### `\keys_set_known:nvN`

### `\keys_set_known:noN`

### `\__keys_set_known:nnnN`

### `\__keys_set_known:onnN`

### `\keys_set_known:nn`

### `\keys_set_known:nV`

### `\keys_set_known:nv`

### `\keys_set_known:no`

### `\__keys_keys_set_known:nn`

```
12588 \cs_new_protected:Npn \keys_set_known:nnN
12589   { \__keys_set_known:onnN \l__keys_unused_clist }
12590 \cs_generate_variant:Nn \keys_set_known:nnN { nV , nv , no }
12591 \cs_new_protected:Npn \__keys_set_known:nnnN #1#2#3#4
12592   {
12593     \clist_clear:N \l__keys_unused_clist
12594     \keys_set_known:nn {#2} {#3}
12595     \tl_set:Nx #4 { \exp_not:o { \l__keys_unused_clist } }
12596     \tl_set:Nn \l__keys_unused_clist {#1}
12597   }
12598 \cs_generate_variant:Nn \__keys_set_known:nnnN { o }
12599 \cs_new_protected:Npn \keys_set_known:nn #1#2
12600   {
12601     \bool_if:NTF \l__keys_only_known_bool
12602       { \keys_set:nn }
12603       { \__keys_set_known:nn }
12604       {#1} {#2}
12605   }
12606 \cs_generate_variant:Nn \keys_set_known:nn { nV , nv , no }
12607 \cs_new_protected:Npn \__keys_set_known:nn #1#2
12608   {
12609     \bool_set_true:N \l__keys_only_known_bool
12610     \keys_set:nn {#1} {#2}
```

```

12611     \bool_set_false:N \l__keys_only_known_bool
12612 }

```

(End definition for `\keys_set_known:nnN` and others. These functions are documented on page 178.)

```

\keys_set_filter:nnN
\keys_set_filter:nnVN
\keys_set_filter:nnvN
\keys_set_filter:nnnN
\__keys_set_filter:nnnnN
\__keys_set_filter:onnnN
\keys_set_filter:nnN
\keys_set_filter:nnV
\keys_set_filter:nnV
\keys_set_filter:nnn
\__keys_set_filter:nnn
\keys_set_groups:nnN
\keys_set_groups:nnV
\keys_set_groups:nnV
\keys_set_groups:nnn
\keys_set_groups:nnn
\__keys_set_groups:nnn
\__keys_set_selective:nnN
\__keys_set_selective:nnnnN
\__keys_set_selective:onnN
\__keys_set_selective:nnN

```

```

12613 \cs_new_protected:Npn \keys_set_filter:nnN
12614     { \__keys_set_filter:onnnN \l__keys_unused_clist }
12615 \cs_generate_variant:Nn \keys_set_filter:nnN { nnV , nnv , nno }
12616 \cs_new_protected:Npn \__keys_set_filter:nnnnN #1#2#3#4#5
12617     {
12618         \clist_clear:N \l__keys_unused_clist
12619         \keys_set_filter:nnn {#2} {#3} {#4}
12620         \tl_set:Nx #5 { \exp_not:o { \l__keys_unused_clist } }
12621         \tl_set:Nn \l__keys_unused_clist {#1}
12622     }
12623 \cs_generate_variant:Nn \__keys_set_filter:nnnnN { o }
12624 \cs_new_protected:Npn \keys_set_filter:nnn #1#2#3
12625     {
12626         \bool_if:NTF \l__keys_filtered_bool
12627             { \__keys_set_selective:nnN }
12628             { \__keys_set_filter:nnN }
12629             {#1} {#2} {#3}
12630     }
12631 \cs_generate_variant:Nn \keys_set_filter:nnn { nnV , nnv , nno }
12632 \cs_new_protected:Npn \__keys_set_filter:nnn #1#2#3
12633     {
12634         \bool_set_true:N \l__keys_filtered_bool
12635         \__keys_set_selective:nnN {#1} {#2} {#3}
12636         \bool_set_false:N \l__keys_filtered_bool
12637     }
12638 \cs_new_protected:Npn \keys_set_groups:nnn #1#2#3
12639     {
12640         \bool_if:NTF \l__keys_filtered_bool
12641             { \__keys_set_groups:nnN }
12642             { \__keys_set_selective:nnN }
12643             {#1} {#2} {#3}
12644     }
12645 \cs_generate_variant:Nn \keys_set_groups:nnn { nnV , nnv , nno }
12646 \cs_new_protected:Npn \__keys_set_groups:nnn #1#2#3
12647     {
12648         \bool_set_false:N \l__keys_filtered_bool
12649         \__keys_set_selective:nnN {#1} {#2} {#3}
12650         \bool_set_true:N \l__keys_filtered_bool
12651     }
12652 \cs_new_protected:Npn \__keys_set_selective:nnN
12653     { \__keys_set_selective:onnN \l__keys_selective_seq }
12654 \cs_new_protected:Npn \__keys_set_selective:nnnn #1#2#3#4
12655     {
12656         \seq_set_from_clist:Nn \l__keys_selective_seq {#3}
12657         \bool_if:NTF \l__keys_selective_bool
12658             { \keys_set:nn }

```

```

12659      { \__keys_set_selective:nn }
12660      {#2} {#4}
12661      \tl_set:Nn \l__keys_selective_seq {#1}
12662    }
12663 \cs_generate_variant:Nn \__keys_set_selective:nnnn { o }
12664 \cs_new_protected:Npn \__keys_set_selective:nn #1#2
12665  {
12666    \bool_set_true:N \l__keys_selective_bool
12667    \keys_set:nn {#1} {#2}
12668    \bool_set_false:N \l__keys_selective_bool
12669  }

```

(End definition for `\keys_set_filter:nnN` and others. These functions are documented on page 179.)

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.

```

12670 \cs_new_protected:Npn \__keys_set:n #1
12671  {
12672    \bool_set_true:N \l__keys_no_value_bool
12673    \__keys_set_aux:onn \l__keys_module_tl {#1} { }
12674  }
12675 \cs_new_protected:Npn \__keys_set:nn #1#2
12676  {
12677    \bool_set_false:N \l__keys_no_value_bool
12678    \__keys_set_aux:onn \l__keys_module_tl {#1} {#2}
12679  }

```

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.

```

12680 \cs_new_protected:Npn \__keys_set_aux:nnn #1#2#3
12681  {
12682    \tl_set:Nx \l__keys_path_tl
12683    {
12684      \tl_if_blank:nF {#1}
12685      { #1 / }
12686      \__keys_remove_spaces:n {#2}
12687    }
12688    \tl_clear:N \l__keys_module_tl
12689    \exp_after:wN \__keys_find_key_module:w \l__keys_path_tl / \q_stop
12690    \__keys_value_or_default:n {#3}
12691    \bool_if:NTF \l__keys_selective_bool
12692      { \__keys_set_selective: }
12693      { \__keys_execute: }
12694    \tl_set:Nn \l__keys_module_tl {#1}
12695  }
12696 \cs_generate_variant:Nn \__keys_set_aux:nnn { o }
12697 \cs_new_protected:Npn \__keys_find_key_module:w #1 / #2 \q_stop
12698  {
12699    \tl_if_blank:nTF {#2}
12700      { \tl_set:Nn \l__keys_key_tl {#1} }
12701      {

```

```

12702     \tl_put_right:Nx \l__keys_module_tl
12703     {
12704         \tl_if_empty:NF \l__keys_module_tl { / }
12705         #1
12706     }
12707     \__keys_find_key_module:w #2 \q_stop
12708 }
12709 }
```

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.

```

12710 \cs_new_protected:Npn \__keys_set_selective:
12711 {
12712     \cs_if_exist:cTF { \c__keys_groups_root_tl \l_keys_path_tl }
12713     {
12714         \clist_set_eq:Nc \l__keys_groups_clist
12715         { \c__keys_groups_root_tl \l_keys_path_tl }
12716         \__keys_check_groups:
12717     }
12718     {
12719         \bool_if:NTF \l__keys_filtered_bool
12720         { \__keys_execute: }
12721         { \__keys_store_unused: }
12722     }
12723 }
```

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.

```

12724 \cs_new_protected:Npn \__keys_check_groups:
12725 {
12726     \bool_set_false:N \l__keys_tmp_bool
12727     \seq_map_inline:Nn \l__keys_selective_seq
12728     {
12729         \clist_map_inline:Nn \l__keys_groups_clist
12730         {
12731             \str_if_eq:nnT {##1} {####1}
12732             {
12733                 \bool_set_true:N \l__keys_tmp_bool
12734                 \clist_map_break:n { \seq_map_break: }
12735             }
12736         }
12737     }
12738     \bool_if:NTF \l__keys_tmp_bool
12739     {
12740         \bool_if:NTF \l__keys_filtered_bool
12741         { \__keys_store_unused: }
12742         { \__keys_execute: }
12743     }
12744     {
12745         \bool_if:NTF \l__keys_filtered_bool
12746         { \__keys_execute: }
12747         { \__keys_store_unused: }
12748     }
12749 }
```

```
12749 }
```

(End definition for `\_keys_set:n` and others.)

`\_keys_value_or_default:n` If a value is given, return it as #1, otherwise send a default if available.

```
12750 \cs_new_protected:Npn \_keys_value_or_default:n #1
12751 {
12752     \bool_if:NTF \l__keys_no_value_bool
12753     {
12754         \cs_if_exist:cTF { \c__keys_default_root_tl \l_keys_path_tl }
12755         {
12756             \tl_set_eq:Nc
12757             \l_keys_value_tl
12758             { \c__keys_default_root_tl \l_keys_path_tl }
12759         }
12760         { \tl_clear:N \l_keys_value_tl }
12761     }
12762     { \tl_set:Nn \l_keys_value_tl {#1} }
12763 }
```

(End definition for `\_keys_value_or_default:n`.)

`\_keys_execute:` 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.

```
12764 \cs_new_protected:Npn \_keys_execute:
12765 {
12766     \cs_if_exist:cTF { \c__keys_code_root_tl \l_keys_path_tl }
12767     {
12768         \cs_if_exist_use:c { \c__keys_validate_root_tl \l_keys_path_tl }
12769         \cs:w \c__keys_code_root_tl \l_keys_path_tl \exp_after:wN \cs_end:
12770         \exp_after:wN { \l_keys_value_tl }
12771     }
12772     {
12773         \bool_if:NTF \l__keys_only_known_bool
12774         { \_keys_store_unused: }
12775         {
12776             \cs_if_exist:cTF
12777             { \c__keys_inherit_root_tl \_keys_parent:o \l_keys_path_tl }
12778             { \_keys_execute_inherit: }
12779             { \_keys_execute_unknown: }
12780         }
12781     }
12782 }
```

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.

```
12783 \cs_new_protected:Npn \_keys_execute_inherit:
12784 {
12785     \clist_map_inline:cn
12786     { \c__keys_inherit_root_tl \_keys_parent:o \l_keys_path_tl }
12787     {
12788         \cs_if_exist:cT
```

```

12789 { \c_keys_code_root_t1 ##1 / \l_keys_key_t1 }
12790 {
12791     \tl_set:Nn \l_keys_inherit_t1 {##1}
12792     \cs:w \c_keys_code_root_t1 ##1 / \l_keys_key_t1
12793         \exp_after:wN \cs_end: \exp_after:wN
12794             { \l_keys_value_t1 }
12795             \clist_map_break:n { \use_none:n }
12796     }
12797 }
12798 \__keys_execute_unknown:
12799 }
12800 \cs_new_protected:Npn \__keys_execute_unknown:
12801 {
12802     \cs_if_exist:cTF
12803         { \c_keys_code_root_t1 \l_keys_module_t1 / unknown }
12804     {
12805         \cs:w \c_keys_code_root_t1 \l_keys_module_t1 / unknown
12806             \exp_after:wN \cs_end: \exp_after:wN { \l_keys_value_t1 }
12807     }
12808     {
12809         \__kernel_msg_error:nnxx { kernel } { key-unknown }
12810             { \l_keys_path_t1 } { \l_keys_module_t1 }
12811     }
12812 }
12813 \cs_new:Npn \__keys_execute:nn #1#2
12814 {
12815     \cs_if_exist:cTF { \c_keys_code_root_t1 #1 }
12816     {
12817         \cs:w \c_keys_code_root_t1 #1 \exp_after:wN \cs_end:
12818             \exp_after:wN { \l_keys_value_t1 }
12819     }
12820     {#2}
12821 }
12822 \cs_new_protected:Npn \__keys_store_unused:
12823 {
12824     \clist_put_right:Nx \l_keys_unused_clist
12825     {
12826         \exp_not:o \l_keys_key_t1
12827         \bool_if:NF \l_keys_no_value_bool
12828             { = { \exp_not:o \l_keys_value_t1 } }
12829     }
12830 }

```

(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.

```

12831 \cs_new:Npn \__keys_choice_find:n #1
12832 {
12833     \tl_if_empty:NTF \l_keys_inherit_t1
12834         { \__keys_choice_find:nn { \l_keys_path_t1 } {#1} }
12835

```

```

12836      \__keys_choice_find:nn
12837          { \l__keys_inherit_tl / \l_keys_key_tl } {#1}
12838      }
12839  }
12840 \cs_new:Npn \__keys_choice_find:nn #1#2
12841  {
12842      \cs_if_exist:cTF { \c__keys_code_root_tl #1 / \__keys_remove_spaces:n {#2} }
12843          { \use:c { \c__keys_code_root_tl #1 / \__keys_remove_spaces:n {#2} } {#2} }
12844          { \use:c { \c__keys_code_root_tl #1 / unknown } {#2} }
12845      }
12846 \cs_new:Npn \__keys_multichoice_find:n #1
12847      { \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.)

```

## 20.7 Utilities

\\_\_keys\_parent:n Used to strip off the ending part of the key path after the last /.

```

12848 \cs_new:Npn \__keys_parent:n #1
12849     { \__keys_parent:w #1 / \q_stop { } }
12850 \cs_generate_variant:Nn \__keys_parent:n { o }
12851 \cs_new:Npn \__keys_parent:w #1 / #2 / #3 \q_stop #4
12852     {
12853         \tl_if_blank:nTF {#2}
12854             { \use_none:n #4 }
12855             {
12856                 \__keys_parent:w #2 / #3 \q_stop { #4 / #1 }
12857             }
12858     }

```

(End definition for \\_\_keys\_parent:n and \\_\_keys\_parent:w.)

\\_\_keys\_remove\_spaces:n Used in a few places so worth handling as a dedicated function.

```

12859 \cs_new:Npn \__keys_remove_spaces:n #1
12860     { \tl_trim_spaces:o { \tl_to_str:n {#1} } }

```

(End definition for \\_\_keys\_remove\_spaces:n.)

**\keys\_if\_exist\_p:nn** A utility for others to see if a key exists.

```

12861 \prg_new_conditional:Npnn \keys_if_exist:nn #1#2 { p , T , F , TF }
12862     {
12863         \cs_if_exist:cTF
12864             { \c__keys_code_root_tl \__keys_remove_spaces:n { #1 / #2 } }
12865             { \prg_return_true: }
12866             { \prg_return_false: }
12867     }

```

(End definition for \keys\_if\_exist:nnTF. This function is documented on page 179.)

**\keys\_if\_choice\_exist\_p:nnn** Just an alternative view on \keys\_if\_exist:nnTF.

```

12868 \prg_new_conditional:Npnn \keys_if_choice_exist:nnn #1#2#3
12869     { p , T , F , TF }
12870     {
12871         \cs_if_exist:cTF

```

```

12872   { \c_keys_code_root_tl \_keys_remove_spaces:n { #1 / #2 / #3 } }
12873   { \prg_return_true: }
12874   { \prg_return_false: }
12875 }

```

(End definition for `\keys_if_choice_exist:nnnTF`. This function is documented on page 179.)

`\keys_show:nn` To show a key, show its code using a message.

```

\keys_log:nn
\_keys_show:Nnn
12876 \cs_new_protected:Npn \keys_show:nn
12877   { \_keys_show:Nnn \msg_show:nnxxxx }
12878 \cs_new_protected:Npn \keys_log:nn
12879   { \_keys_show:Nnn \msg_log:nnxxxx }
12880 \cs_new_protected:Npn \_keys_show:Nnn #1#2#3
12881   {
12882     #1 { LaTeX / kernel } { show-key }
12883     { \_keys_remove_spaces:n { #2 / #3 } }
12884     {
12885       \keys_if_exist:nnT {#2} {#3}
12886       {
12887         \exp_args:Nnf \msg_show_item_unbraced:nn { code }
12888         {
12889           \exp_args:Nc \token_get_replacement_spec:N
12890           {
12891             \c_keys_code_root_tl
12892             \_keys_remove_spaces:n { #2 / #3 }
12893           }
12894         }
12895       }
12896     }
12897   { } { }
12898 }

```

(End definition for `\keys_show:nn`, `\keys_log:nn`, and `\_keys_show:Nnn`. These functions are documented on page 179.)

## 20.8 Messages

For when there is a need to complain.

```

\_kernel_msg_new:nnnn { kernel } { boolean-values-only }
12900   { Key~'#1'~accepts~boolean~values~only. }
12901   { The~key~'#1'~only~accepts~the~values~'true'~and~'false'. }
12902 \_kernel_msg_new:nnnn { kernel } { key-choice-unknown }
12903   { Key~'#1'~accepts~only~a~fixed~set~of~choices. }
12904   {
12905     The~key~'#1'~only~accepts~predefined~values~,~
12906     and~'#2'~is~not~one~of~these.
12907   }
12908 \_kernel_msg_new:nnnn { kernel } { key-unknown }
12909   { The~key~'#1'~is~unknown~and~is~being~ignored. }
12910   {
12911     The~module~'#2'~does~not~have~a~key~called~'#1'. \\
12912     Check~that~you~have~spelled~the~key~name~correctly.
12913   }
12914 \_kernel_msg_new:nnnn { kernel } { nested-choice-key }

```

```

12915 { Attempt~to~define~'#1'~as~a~nested~choice~key. }
12916 {
12917   The~key~'#1'~cannot~be~defined~as~a~choice~as~the~parent~key~'#2'~is~
12918   itself~a~choice.
12919 }
12920 \__kernel_msg_new:nnn { kernel } { value-forbidden }
12921 { The~key~'#1'~does~not~take~a~value. }
12922 {
12923   The~key~'#1'~should~be~given~without~a~value.\\
12924   The~value~'#2'~was~present:~the~key~will~be~ignored.
12925 }
12926 \__kernel_msg_new:nnn { kernel } { value-required }
12927 { The~key~'#1'~requires~a~value. }
12928 {
12929   The~key~'#1'~must~have~a~value.\\
12930   No~value~was~present:~the~key~will~be~ignored.
12931 }
12932 \__kernel_msg_new:nn { kernel } { show-key }
12933 {
12934   The~key~#1
12935   \tl_if_empty:nTF {#2}
12936   { is~undefined. }
12937   { has~the~properties: #2 . }
12938 }
12939 </initex | package>

```

## 21 **I3intarray** implementation

```

12940 <*initex | package>
12941 <@=intarray>

```

### 21.1 Allocating arrays

`\__intarray_entry:w`    We use these primitives quite a lot in this module.  
`\__intarray_count:w`

```

12942 \cs_new_eq:NN \__intarray_entry:w \tex_fontdimen:D
12943 \cs_new_eq:NN \__intarray_count:w \tex_hyphenchar:D

```

(End definition for `\__intarray_entry:w` and `\__intarray_count:w`.)

`\l__intarray_loop_int`    A loop index.

```

12944 \int_new:N \l__intarray_loop_int

```

(End definition for `\l__intarray_loop_int`.)

`\c__intarray_sp_dim`    Used to convert integers to dimensions fast.

```

12945 \dim_const:Nn \c__intarray_sp_dim { 1 sp }

```

(End definition for `\c__intarray_sp_dim`.)

`\g__intarray_font_int`    Used to assign one font per array.

```

12946 \int_new:N \g__intarray_font_int

```

(End definition for `\g_intarray_font_int`.)

```
12947 \__kernel_msg_new:nnn { kernel } { negative-array-size }
12948   { Size~of~array~may~not~be~negative:~#1 }
```

**\intarray\_new:Nn** Declare #1 to be a font (arbitrarily `cmr10` at a never-used size). Store the array's size as the `\hyphenchar` of that font and make sure enough `\fontdimen` are allocated, by setting the last one. Then clear any `\fontdimen` that `cmr10` starts with. It seems LuaTeX's `cmr10` has an extra `\fontdimen` parameter number 8 compared to other engines (for a math font we would replace 8 by 22 or some such). Every `intarray` must be global; it's enough to run this check in `\intarray_new:Nn`.

```
12949 \cs_new_protected:Npn \__intarray_new:N #1
12950   {
12951     \__kernel_chk_if_free_cs:N #1
12952     \int_gincr:N \g_intarray_font_int
12953     \tex_global:D \tex_font:D #1
12954     = cmr10-at- \g_intarray_font_int \c_intarray_sp_dim \scan_stop:
12955     \int_step_inline:nn { 8 }
12956     { \__kernel_intarray_gset:Nnn #1 {##1} \c_zero_int }
12957   }
12958 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN g #1 } { }
12959 \cs_new_protected:Npn \intarray_new:Nn #1#2
12960   {
12961     \__intarray_new:N #1
12962     \__intarray_count:w #1 = \int_eval:n {#2} \scan_stop:
12963     \int_compare:nNnT { \intarray_count:N #1 } < 0
12964     {
12965       \__kernel_msg_error:nnx { kernel } { negative-array-size }
12966       { \intarray_count:N #1 }
12967     }
12968     \int_compare:nNnT { \intarray_count:N #1 } > 0
12969     { \__kernel_intarray_gset:Nnn #1 { \intarray_count:N #1 } { 0 } }
12970   }
```

(End definition for `\intarray_new:Nn` and `\__intarray_new:N`. This function is documented on page [182](#).)

**\intarray\_count:N** Size of an array.

```
12971 \cs_new:Npn \intarray_count:N #1 { \int_value:w \__intarray_count:w #1 }
```

(End definition for `\intarray_count:N`. This function is documented on page [182](#).)

## 21.2 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  $\pm\c_max_dim$ .

```
12972 \cs_new:Npn \__intarray_signed_max_dim:n #1
12973   { \int_value:w \int_compare:nNnT {#1} < 0 { - } \c_max_dim }
```

(End definition for `\__intarray_signed_max_dim:n`.)

`\__intarray_bounds:NNnTF` and `\__intarray_bounds_error:NNn` share bounds checking.  
The T branch is used if #3 is within bounds of the array #2.

```

12974 \cs_new:Npn \__intarray_bounds:NNnTF #1#2#3#4#5
12975 {
12976     \if_int_compare:w 1 > #3 \exp_stop_f:
12977         \__intarray_bounds_error:NNn #1 #2 {#3}
12978         #5
12979     \else:
12980         \if_int_compare:w #3 > \intarray_count:N #2 \exp_stop_f:
12981             \__intarray_bounds_error:NNn #1 #2 {#3}
12982             #5
12983         \else:
12984             #4
12985         \fi:
12986     \fi:
12987 }
12988 \cs_new:Npn \__intarray_bounds_error:NNn #1#2#3
12989 {
12990     #1 { kernel } { out-of-bounds }
12991     { \token_to_str:N #2 } {#3} { \intarray_count:N #2 }
12992 }
```

(End definition for `\__intarray_bounds:NNnTF` and `\__intarray_bounds_error:NNn`.)

### `\intarray_gset:Nnn`

```
\__kernel_intarray_gset:Nnn
\__intarray_gset:Nnn
\__intarray_gset_overflow: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.

```

12993 \cs_new_protected:Npn \__kernel_intarray_gset:Nnn #1#2#3
12994     { \__intarray_entry:w #2 #1 #3 \c_intarray_sp_dim }
12995 \cs_new_protected:Npn \intarray_gset:Nnn #1#2#3
12996     {
12997         \exp_after:wN \__intarray_gset:Nww
12998         \exp_after:wN #1
12999         \int_value:w \int_eval:n {#2} \exp_after:wN ;
13000         \int_value:w \int_eval:n {#3} ;
13001     }
13002 \cs_new_protected:Npn \__intarray_gset:Nww #1#2 ; #3 ;
13003     {
13004         \__intarray_bounds:NNnTF \__kernel_msg_error:nxxxx #1 {#2}
13005         {
13006             \__intarray_gset_overflow_test:nw {#3}
13007             \__kernel_intarray_gset:Nnn #1 {#2} {#3}
13008         }
13009     }
13010 }
13011 \cs_if_exist:NTF \tex_ifabsnum:D
13012     {
13013         \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
13014         {
13015             \tex_ifabsnum:D #1 > \c_max_dim
13016                 \exp_after:wN \__intarray_gset_overflow>NNnn
13017             \fi:
13018         }
13019 }
```

```

13020  {
13021    \cs_new_protected:Npn \__intarray_gset_overflow_test:nw #1
13022    {
13023      \if_int_compare:w \int_abs:n {#1} > \c_max_dim
13024        \exp_after:wN \__intarray_gset_overflow>NNnn
13025      \fi:
13026    }
13027  }
13028 \cs_new_protected:Npn \__intarray_gset_overflow:NNnn #1#2#3#4
13029  {
13030    \__kernel_msg_error:nxxxx { kernel } { overflow }
13031    { \token_to_str:N #2 } {#3} {#4} { \__intarray_signed_max_dim:n {#4} }
13032    #1 #2 {#3} { \__intarray_signed_max_dim:n {#4} }
13033  }

```

(End definition for `\intarray_gset:Nnn` and others. This function is documented on page 182.)

**\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.

```

13034 \cs_new_protected:Npn \intarray_gzero:N #1
13035  {
13036    \int_zero:N \l__intarray_loop_int
13037    \prg_replicate:nn { \intarray_count:N #1 }
13038    {
13039      \int_incr:N \l__intarray_loop_int
13040      \__intarray_entry:w \l__intarray_loop_int #1 \c_zero_dim
13041    }
13042  }

```

(End definition for `\intarray_gzero:N`. This function is documented on page 182.)

**\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`.

```

13043 \cs_new:Npn \__kernel_intarray_item:Nn #1#2
13044  { \int_value:w \__intarray_entry:w #2 #1 }
13045 \cs_new:Npn \intarray_item:Nn #1#2
13046  {
13047    \exp_after:wN \__intarray_item:Nw
13048    \exp_after:wN #1
13049    \int_value:w \int_eval:n {#2} ;
13050  }
13051 \cs_new:Npn \__intarray_item:Nw #1#2 ;
13052  {
13053    \__intarray_bounds>NNnTF \__kernel_msg_expandable_error:nnfff #1 {#2}
13054    { \__kernel_intarray_item:Nn #1 {#2} }
13055    { 0 }
13056  }

```

(End definition for `\intarray_item:Nn`, `\__kernel_intarray_item:Nn`, and `\__intarray_item:Nn`. This function is documented on page 182.)

`\intarray_rand_item:N` Importantly, `\intarray_item:Nn` only evaluates its argument once.

```
13057 \cs_new:Npn \intarray_rand_item:N #1
13058   { \intarray_item:Nn #1 { \int_rand:n { \intarray_count:N #1 } } }
```

(End definition for `\intarray_rand_item:N`. This function is documented on page 182.)

### 21.3 Working with contents of integer arrays

At the time of writing these are candidates, but we need at least `\intarray_const_from_clist:Nn` in l3fp so before l3candidates.

`\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.

```
13059 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
13060 \cs_new_protected:Npn \intarray_const_from_clist:Nn #1#2
13061   {
13062     \__intarray_new:N #1
13063     \int_zero:N \l__intarray_loop_int
13064     \clist_map_inline:nn {#2}
13065       { \exp_args:Nf \__intarray_const_from_clist:nN { \int_eval:n {##1} } #1 }
13066     \__intarray_count:w #1 \l__intarray_loop_int
13067   }
13068 \cs_new_protected:Npn \__intarray_const_from_clist:nN #1#2
13069   {
13070     \int_incr:N \l__intarray_loop_int
13071     \__intarray_gset_overflow_test:nw {#1}
13072     \__kernel_intarray_gset:Nnn #2 \l__intarray_loop_int {#1}
13073   }
```

(End definition for `\intarray_const_from_clist:Nn` and `\__intarray_const_from_clist:nN`. This function is documented on page 247.)

`\intarray_to_clist:N` 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.

```
13074 \cs_new:Npn \intarray_to_clist:N #1 { \__intarray_to_clist:Nn #1 { , } }
13075 \cs_new:Npn \__intarray_to_clist:Nn #1#2
13076   {
13077     \int_compare:nNnF { \intarray_count:N #1 } = \c_zero_int
13078     {
13079       \exp_last_unbraced:Nf \use_none:n
13080         { \__intarray_to_clist:w 1 ; #1 {#2} \prg_break_point: }
13081     }
13082   }
13083 \cs_new:Npn \__intarray_to_clist:w #1 ; #2#3
13084   {
13085     \if_int_compare:w #1 > \__intarray_count:w #2
13086       \prg_break:n
13087     \fi:
```

```

13088     #3 \_kernel_intarray_item:Nn #2 {#1}
13089     \exp_after:wN \_intarray_to_clist:w
13090     \int_value:w \int_eval:w #1 + \c_one_int ; #2 {#3}
13091 }

```

(End definition for `\intarray_to_clist:N`, `\_intarray_to_clist:Nn`, and `\_intarray_to_clist:w`. This function is documented on page 247.)

`\intarray_show:N`

`\intarray_log:N`

Convert the list to a comma list (with spaces after each comma)

```

13092 \cs_new_protected:Npn \intarray_show:N { \_intarray_show>NN \msg_show:nxxxxx }
13093 \cs_generate_variant:Nn \intarray_show:N { c }
13094 \cs_new_protected:Npn \intarray_log:N { \_intarray_show>NN \msg_log:nxxxxx }
13095 \cs_generate_variant:Nn \intarray_log:N { c }
13096 \cs_new_protected:Npn \_intarray_show>NN #1#2
13097 {
13098     \_kernel_chk_defined:NT #2
13099     {
13100         #1 { LaTeX/kernel } { show-intarray }
13101         { \token_to_str:N #2 }
13102         { \intarray_count:N #2 }
13103         { >~ \_intarray_to_clist:Nn #2 { , ~ } }
13104         { }
13105     }
13106 }

```

(End definition for `\intarray_show:N` and `\intarray_log:N`. These functions are documented on page 247.)

## 21.4 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  $\langle min \rangle$ , to avoid overflow when  $\langle max \rangle$  are  $\pm\c_max_int$ ), otherwise `\_kernel_randint:nn`. Finally, if there are no random numbers do not define any of the auxiliaries.

```

13107 \cs_new_protected:Npn \intarray_gset_rand:Nn #1
13108     { \intarray_gset_rand:Nnn #1 { 1 } }
13109 \sys_if_rand_exist:TF
13110 {
13111     \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
13112     {
13113         \_intarray_gset_rand:Nff #1
13114         { \int_eval:n {#2} } { \int_eval:n {#3} }
13115     }
13116 \cs_new_protected:Npn \_intarray_gset_rand:Nnn #1#2#3
13117 {
13118     \int_compare:nNnTF {#2} > {#3}
13119     {
13120         \_kernel_msg_expandable_error:nnnn
13121         { kernel } { randint-backward-range } {#2} {#3}
13122         \_intarray_gset_rand:Nnn #1 {#3} {#2}

```

```

13123     }
13124     {
13125         \__intarray_gset_overflow_test:nw {#2}
13126         \__intarray_gset_rand_auxi:Nnnn #1 { } {#2} {#3}
13127     }
13128 }
13129 \cs_generate_variant:Nn \__intarray_gset_rand:Nnn { Nff }
13130 \cs_new_protected:Npn \__intarray_gset_rand_auxi:Nnnn #1#2#3#4
13131 {
13132     \__intarray_gset_overflow_test:nw {#4}
13133     \__intarray_gset_rand_auxii:Nnnn #1 { } {#4} {#3}
13134 }
13135 \cs_new_protected:Npn \__intarray_gset_rand_auxii:Nnnn #1#2#3#4
13136 {
13137     \exp_args:NNf \__intarray_gset_rand_auxiii:Nnnn #1
13138     { \int_eval:n { #3 - #4 + 1 } } {#4} {#3}
13139 }
13140 \cs_new_protected:Npn \__intarray_gset_rand_auxiii:Nnnn #1#2#3#4
13141 {
13142     \exp_args:NNf \__intarray_gset_all_same:Nn #1
13143     {
13144         \int_compare:nNnTF {#2} > \c_kernel_randint_max_int
13145         {
13146             \exp_stop_f:
13147             \int_eval:n { \__kernel_randint:nn {#3} {#4} }
13148         }
13149         {
13150             \exp_stop_f:
13151             \int_eval:n { \__kernel_randint:n {#2} - 1 + #3 }
13152         }
13153     }
13154 }
13155 \cs_new_protected:Npn \__intarray_gset_all_same:Nn #1#2
13156 {
13157     \int_zero:N \l__intarray_loop_int
13158     \prg_replicate:nn { \intarray_count:N #1 }
13159     {
13160         \int_incr:N \l__intarray_loop_int
13161         \__kernel_intarray_gset:Nnn #1 \l__intarray_loop_int {#2}
13162     }
13163 }
13164 }
13165 {
13166     \cs_new_protected:Npn \intarray_gset_rand:Nnn #1#2#3
13167     {
13168         \__kernel_msg_error:nnn { kernel } { fp-no-random }
13169         { \intarray_gset_rand:Nnn #1 {#2} {#3} }
13170     }
13171 }

```

(End definition for `\intarray_gset_rand:Nn` and others. These functions are documented on page 247.)

13172 ⟨/initex | package⟩

## 22 I3fp implementation

Nothing to see here: everything is in the subfiles!

## 23 I3fp-aux implementation

13173  $\langle * \text{initex} \mid \text{package} \rangle$

13174  $\langle @ @ = \text{fp} \rangle$

### 23.1 Access to primitives

$\_\_fp\_int\_eval:w$  Largely for performance reasons, we need to directly access primitives rather than use  $\inteval:n$ . This happens *a lot*, so we use private names. The same is true for  $\romannumeral$ , although it is used much less widely.

13175  $\csneweq:NN \_\_fp\_int\_eval:w \texnumexpr:D$

13176  $\csneweq:NN \_\_fp\_int\_eval\_end: \scanstop:$

13177  $\csneweq:NN \_\_fp\_int\_to\_roman:w \texromannumeral:D$

(End definition for  $\_\_fp\_int\_eval:w$ ,  $\_\_fp\_int\_eval\_end:$ , and  $\_\_fp\_int\_to\_roman:w$ .)

### 23.2 Internal representation

Internally, a floating point number  $\langle X \rangle$  is a token list containing

$\sfp \_\_fp\_chk:w \langle \text{case} \rangle \langle \text{sign} \rangle \langle \text{body} \rangle ;$

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,  $\sfp$  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  $\langle \text{case} \rangle$ , which is a single digit:

- 0 zeros: +0 and -0,
- 1 “normal” numbers (positive and negative),
- 2 infinities: +inf and -inf,
- 3 quiet and signalling nan.

The  $\langle \text{sign} \rangle$  is 0 (positive) or 2 (negative), except in the case of nan, which have  $\langle \text{sign} \rangle = 1$ . This ensures that changing the  $\langle \text{sign} \rangle$  digit to  $2 - \langle \text{sign} \rangle$  is exactly equivalent to changing the sign of the number.

Special floating point numbers have the form

$\sfp \_\_fp\_chk:w \langle \text{case} \rangle \langle \text{sign} \rangle \sfp \dots ;$

Table 1: 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.

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 <sign> {\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.

### 23.3 Using arguments and semicolons

<code>\fp_use_none_stop_f:n</code>	This function removes an argument (typically a digit) and replaces it by <code>\exp_stop_f:</code> , a marker which stops f-type expansion.
	<code>13178 \cs_new:Npn \fp_use_none_stop_f:n #1 { \exp_stop_f: }</code>
	<i>(End definition for <code>\fp_use_none_stop_f:n</code>.)</i>
<code>\fp_use_s:n</code> <code>\fp_use_s:nn</code>	Those functions place a semicolon after one or two arguments (typically digits).  <code>13179 \cs_new:Npn \fp_use_s:n #1 { #1; }</code> <code>13180 \cs_new:Npn \fp_use_s:nn #1#2 { #1#2; }</code>  <i>(End definition for <code>\fp_use_s:n</code> and <code>\fp_use_s:nn</code>.)</i>
<code>\fp_use_none_until_s:w</code> <code>\fp_use_i_until_s:nw</code> <code>\fp_use_ii_until_s:nnw</code>	Those functions select specific arguments among a set of arguments delimited by a semicolon.  <code>13181 \cs_new:Npn \fp_use_none_until_s:w #1; { }</code> <code>13182 \cs_new:Npn \fp_use_i_until_s:nw #1#2; { #1 }</code> <code>13183 \cs_new:Npn \fp_use_ii_until_s:nnw #1#2#3; { #2 }</code>  <i>(End definition for <code>\fp_use_none_until_s:w</code>, <code>\fp_use_i_until_s:nw</code>, and <code>\fp_use_ii_until_s:nnw</code>.)</i>
<code>\fp_reverse_args:Nww</code>	Many internal functions take arguments delimited by semicolons, and it is occasionally useful to swap two such arguments.  <code>13184 \cs_new:Npn \fp_reverse_args:Nww #1 #2; #3; { #1 #3; #2; }</code>

(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.

```
13185 \cs_new:Npn \_fp_rrot:www #1; #2; #3; { #2; #3; #1; }
```

(End definition for `\_fp_rrot:www`.)

`\_fp_use_i:ww` Many internal functions take arguments delimited by semicolons, and it is occasionally useful to remove one or two such arguments.

```
13186 \cs_new:Npn \_fp_use_i:ww #1; #2; { #1; }
```

```
13187 \cs_new:Npn \_fp_use_i:www #1; #2; #3; { #1; }
```

(End definition for `\_fp_use_i:ww` and `\_fp_use_i:www`.)

## 23.4 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.

```
13188 \cs_new_protected:Npn \_fp_misused:n #1
```

```
13189 { \_kernel_msg_error:nnx { kernel } { misused-fp } { \fp_to_tl:n {#1} } }
```

(End definition for `\_fp_misused:n`.)

`\s_fp` 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.

```
13190 \scan_new:N \s_fp
```

```
13191 \cs_new_protected:Npn \_fp_chk:w #1 ;
```

```
13192 { \_fp_misused:n { \s_fp \_fp_chk:w #1 ; } }
```

(End definition for `\s_fp` and `\_fp_chk:w`.)

`\s_fp_mark` Aliases of `\tex_relax:D`, used to terminate expressions.

`\s_fp_stop`

```
13193 \scan_new:N \s_fp_mark
```

```
13194 \scan_new:N \s_fp_stop
```

(End definition for `\s_fp_mark` and `\s_fp_stop`.)

`\s_fp_invalid` A couple of scan marks used to indicate where special floating point numbers come from.

`\s_fp_underflow`

```
13195 \scan_new:N \s_fp_invalid
```

`\s_fp_overflow`

```
13196 \scan_new:N \s_fp_underflow
```

`\s_fp_division`

```
13197 \scan_new:N \s_fp_overflow
```

`\s_fp_exact`

```
13198 \scan_new:N \s_fp_division
```

```
13199 \scan_new:N \s_fp_exact
```

(End definition for `\s_fp_invalid` and others.)

`\c_zero_fp`

```

13200 \tl_const:Nn \c_zero_fp      { \s_fp \fp_chk:w 0 0 \s_fp_exact ; }
13201 \tl_const:Nn \c_minus_zero_fp { \s_fp \fp_chk:w 0 2 \s_fp_exact ; }
13202 \tl_const:Nn \c_inf_fp       { \s_fp \fp_chk:w 2 0 \s_fp_exact ; }
13203 \tl_const:Nn \c_minus_inf_fp { \s_fp \fp_chk:w 2 2 \s_fp_exact ; }
13204 \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 191.)

`\c_fp_prec_int`

```

\c_fp_half_prec_int
\c_fp_block_int

```

The number of digits of floating points.

```

13205 \int_const:Nn \c_fp_prec_int { 16 }
13206 \int_const:Nn \c_fp_half_prec_int { 8 }
13207 \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.

```
13208 \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 `-minus_min_exponent` and `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 `min_exponent` with the opposite sign but that would waste one T<sub>E</sub>X count.

```

13209 \int_const:Nn \c_fp_minus_min_exponent_int { 10000 }
13210 \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.

```
13211 \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.

```

13212 \tl_const:Nx \c_fp_overflowing_fp
13213 {
13214   \s_fp \fp_chk:w 1 0
13215   { \int_eval:n { \c_fp_max_exponent_int + 1 } }
13216   {1000} {0000} {0000} {0000} ;
13217 }

```

(End definition for `\c_fp_overflowing_fp`.)

`\_fp_zero_fp:N`

`\_fp_inf_fp:N`

In case of overflow or underflow, we have to output a zero or infinity with a given sign.

```

13218 \cs_new:Npn \_fp_zero_fp:N #1
13219   { \s_fp \fp_chk:w 0 #1 \s_fp_underflow ; }
13220 \cs_new:Npn \_fp_inf_fp:N #1
13221   { \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.

```

13222 \cs_new:Npn \_\_fp_exponent:w \s_\_fp \_\_fp_chk:w #1
13223 {
13224   \if_meaning:w 1 #1
13225     \exp_after:wN \_\_fp_use_ii_until_s:nw
13226   \else:
13227     \exp_after:wN \_\_fp_use_i_until_s:nw
13228     \exp_after:wN 0
13229   \fi:
13230 }
```

(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.

```

13231 \cs_new:Npn \_\_fp_neg_sign:N #1
13232 { \_\_fp_int_eval:w 2 - #1 \_\_fp_int_eval_end: }
```

(End definition for \\_\\_fp\\_neg\\_sign:N.)

## 23.5 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.

```

13233 \cs_new:Npn \_\_fp_SANITIZE:Nw #1 #2;
13234 {
13235   \if_case:w
13236     \if_int_compare:w #2 > \c_fp_max_exponent_int 1 ~ \else:
13237       \if_int_compare:w #2 < - \c_fp_minus_min_exponent_int 2 ~ \else:
13238         \if_meaning:w 1 #1 3 ~ \fi: \fi: \fi: 0 ~
13239       \or: \exp_after:wN \_\_fp_overflow:w
13240       \or: \exp_after:wN \_\_fp_underflow:w
13241       \or: \exp_after:wN \_\_fp_SANITIZE_zero:w
13242       \fi:
13243       \s_\_fp \_\_fp_chk:w 1 #1 {#2}
13244   }
13245 \cs_new:Npn \_\_fp_SANITIZE:wN #1; #2 { \_\_fp_SANITIZE:Nw #2 #1; }
13246 \cs_new:Npn \_\_fp_SANITIZE_zero:w \s_\_fp \_\_fp_chk:w #1 #2 #3;
13247 { \c_zero_fp }
```

(End definition for \\_\\_fp\\_sanitize:Nw, \\_\\_fp\\_sanitize:wN, and \\_\\_fp\\_sanitize\\_zero:w.)

## 23.6 Expanding after a floating point number

\\_\\_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.

```

13248 \cs_new:Npn \__fp_exp_after_o:w \s__fp \__fp_chk:w #1
13249 {
13250     \if_meaning:w 1 #1
13251         \exp_after:wN \__fp_exp_after_normal:nNnw
13252     \else:
13253         \exp_after:wN \__fp_exp_after_special:nNnw
13254     \fi:
13255     { }
13256     #1
13257 }
13258 \cs_new:Npn \__fp_exp_after_f:nw #1 \s__fp \__fp_chk:w #2
13259 {
13260     \if_meaning:w 1 #2
13261         \exp_after:wN \__fp_exp_after_normal:nNnw
13262     \else:
13263         \exp_after:wN \__fp_exp_after_special:nNnw
13264     \fi:
13265     { \exp:w \exp_end_continue_f:w #1 }
13266     #2
13267 }
```

(End definition for `\__fp_exp_after_o:w` and `\__fp_exp_after_f:nw`.)

`\__fp_exp_after_special:nNnw` {(after)} {case} {sign} {scan mark} ;  
Special floating point numbers are easy to jump over since they contain few tokens.

```

13268 \cs_new:Npn \__fp_exp_after_special:nNnw #1#2#3#4;
13269 {
13270     \exp_after:wN \s__fp
13271     \exp_after:wN \__fp_chk:w
13272     \exp_after:wN #2
13273     \exp_after:wN #3
13274     \exp_after:wN #4
13275     \exp_after:wN ;
13276     #1
13277 }
```

(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.

```

13278 \cs_new:Npn \__fp_exp_after_normal:nNnw #1 1 #2 #3 #4#5#6#7;
13279 {
13280     \exp_after:wN \__fp_exp_after_normal:Nwwwww
13281     \exp_after:wN #2
13282     \int_value:w #3 \exp_after:wN ;
13283     \int_value:w 1 #4 \exp_after:wN ;
13284     \int_value:w 1 #5 \exp_after:wN ;
13285     \int_value:w 1 #6 \exp_after:wN ;
13286     \int_value:w 1 #7 \exp_after:wN ; #1
13287 }
13288 \cs_new:Npn \__fp_exp_after_normal:Nwwwww
```

```

13289      #1 #2; 1 #3 ; 1 #4 ; 1 #5 ; 1 #6 ;
13290  { \s_fp \_fp_chk:w 1 #1 {#2} {#3} {#4} {#5} {#6} ; }

```

(End definition for `\_fp_exp_after_normal:nNnw.`)

## 23.7 Other floating point types

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.

```

13291  \scan_new:N \s_fp_tuple
13292  \cs_new_protected:Npn \_fp_tuple_chk:w #1 ;
13293  { \_fp_misused:n { \s_fp_tuple \_fp_tuple_chk:w #1 ; } }
13294  \tl_const:Nn \c_fp_empty_tuple_fp
13295  { \s_fp_tuple \_fp_tuple_chk:w { } ; }

```

(End definition for `\s_fp_tuple`, `\_fp_tuple_chk:w`, and `\c_fp_empty_tuple_fp`.)

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.

```

13296 \cs_new:Npn \_fp_array_count:n #1
13297  { \_fp_tuple_count:w \s_fp_tuple \_fp_tuple_chk:w {#1} ; }
13298  \cs_new:Npn \_fp_tuple_count:w \s_fp_tuple \_fp_tuple_chk:w #1 ;
13299  {
13300  \int_value:w \_fp_int_eval:w 0
13301  \_fp_tuple_count_loop:Nw #1 { ? \prg_break: } ;
13302  \prg_break_point:
13303  \_fp_int_eval_end:
13304  }
13305 \cs_new:Npn \_fp_tuple_count_loop:Nw #1#2;
13306  { \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`.)

Used as `\_fp_if_type_fp:NTwFw <marker> {<true code>} \s_fp {<false code>} \q_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.

```
13307 \cs_new:Npn \_fp_if_type_fp:NTwFw #1 \s_fp #2 #3 \q_stop {#2}
```

(End definition for `\_fp_if_type_fp:NTwFw`.)

True if all items are floating point numbers. Used for `min`.

```

13308 \cs_new:Npn \_fp_array_if_all_fp:nTF #1
13309  {
13310  \_fp_array_if_all_fp_loop:w #1 { \s_fp \prg_break: } ;
13311  \prg_break_point: \use_i:nn
13312  }
13313 \cs_new:Npn \_fp_array_if_all_fp_loop:w #1#2 ;
13314  {
13315  \_fp_if_type_fp:NTwFw
13316  #1 \_fp_array_if_all_fp_loop:w

```

```

13317   \s_fp { \prg_break:n \use_iii:nnn }
13318   \q_stop
13319 }

(End definition for \fp_array_if_all_fp:nTF and \fp_array_if_all_fp_loop:w.)
```

\fp\_type\_from\_scan:N 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 `_?`.

```

13320 \cs_new:Npn \fp_type_from_scan:N #1
13321 {
13322   \fp_if_type_fp:NTwFw
13323   #1 { }
13324   \s_fp { \fp_type_from_scan_other:N #1 }
13325   \q_stop
13326 }
13327 \cs_new:Npx \fp_type_from_scan_other:N #1
13328 {
13329   \exp_not:N \exp_after:wN \exp_not:N \fp_type_from_scan:w
13330   \exp_not:N \token_to_str:N #1 \exp_not:N \q_mark
13331   \tl_to_str:n { s_fp _? } \exp_not:N \q_mark \exp_not:N \q_stop
13332 }
13333 \exp_last_unbraced:NNNNo
13334 \cs_new:Npn \fp_type_from_scan:w #1
13335 { \tl_to_str:n { s_fp } } #2 \q_mark #3 \q_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`.

```

13336 \cs_new:Npn \fp_change_func_type:NNN #1#2#3
13337 {
13338   \fp_if_type_fp:NTwFw
13339   #1 #2
13340   \s_fp
13341   {
13342     \exp_after:wN \fp_change_func_type_chk:NNN
13343     \cs:w
13344       __fp \fp_type_from_scan_other:N #1
13345       \exp_after:wN \fp_change_func_type_aux:w \token_to_str:N #2
13346     \cs_end:
13347       #2 #3
13348   }
13349   \q_stop
13350 }
13351 \exp_last_unbraced:NNNNo
13352 \cs_new:Npn \fp_change_func_type_aux:w #1 { \tl_to_str:n { __fp } } { }
13353 \cs_new:Npn \fp_change_func_type_chk:NNN #1#2#3
13354 {
13355   \if_meaning:w \scan_stop: #1
13356   \exp_after:wN #3 \exp_after:wN #2
13357   \else:
```

```

13358     \exp_after:wN #1
13359     \fi:
13360 }

```

(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_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.

```

13361 \cs_new:Npn \fp_exp_after_any_f:Nnw #1
13362   { \cs:w __fp_exp_after __fp_type_from_scan_other:N #1 _f:nw \cs_end: }
13363 \cs_new:Npn \fp_exp_after_any_f:nw #1#2
13364   {
13365     \fp_if_type_fp:NTwFw
13366     #2 \fp_exp_after_f:nw
13367     \s_fp { \fp_exp_after_any_f:Nnw #2 }
13368     \q_stop
13369     {#1} #2
13370   }
13371 \cs_new_eq:NN \fp_exp_after_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_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_stop

13372 \cs_new:Npn \fp_exp_after_tuple_o:w
13373   { \fp_exp_after_tuple_f:nw { \exp_after:wN \exp_stop_f: } }
13374 \cs_new:Npn \fp_exp_after_tuple_f:nw
13375   #1 \s_fp_tuple \fp_tuple_chk:w #2 ;
13376   {
13377     \exp_after:wN \s_fp_tuple
13378     \exp_after:wN \fp_tuple_chk:w
13379     \exp_after:wN {
13380       \exp:w \exp_end_continue_f:w
13381       \fp_exp_after_array_f:w #2 \s_fp_stop
13382     \exp_after:wN }
13383     \exp_after:wN ;
13384     \exp:w \exp_end_continue_f:w #1
13385   }
13386 \cs_new:Npn \fp_exp_after_array_f:w
13387   { \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`.)

## 23.8 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 `l3fp-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:NNNNw
\fp_int_value:w \fp_int_eval:w 4 9995 0000
  + 12345 * 6677
\exp_after:wN \pack:NNNNw
\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  $-50000$  (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  $499950000$  (the “middle shift”). That, in turn, expands `\exp_after:wN` which triggers the third computation. The third computation's value is  $500000000 + 12345 \times 8899$ , 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  $500000000/10^4 + 499950000 = 500000000$ . 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.

```
13388 \int_const:Nn \c_fp_leading_shift_int { - 5 0000 }
13389 \int_const:Nn \c_fp_middle_shift_int { 5 0000 * 9999 }
13390 \int_const:Nn \c_fp_trailing_shift_int { 5 0000 * 10000 }
13391 \cs_new:Npn \fp_pack:NNNNw #1 #2#3#4#5 #6; { + #1#2#3#4#5 ; {#6} }
```

(End definition for `\_fp_pack:NNNNNw` and others.)

```
\_fp_pack_big:NNNNNw
\c_fp_big_trailing_shift_int
\c_fp_big_middle_shift_int
\c_fp_big_leading_shift_int
```

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.

```
13392 \int_const:Nn \c_fp_big_leading_shift_int { - 15 2374 }
13393 \int_const:Nn \c_fp_big_middle_shift_int { 15 2374 * 9999 }
13394 \int_const:Nn \c_fp_big_trailing_shift_int { 15 2374 * 10000 }
13395 \cs_new:Npn \_fp_pack_big:NNNNNw #1#2 #3#4#5#6 #7;
13396 { + #1#2#3#4#5#6 ; {#7} }
```

(End definition for `\_fp_pack_big:NNNNNw` 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.

```
13397 \int_const:Nn \c_fp_Bigg_leading_shift_int { - 20 0000 }
13398 \int_const:Nn \c_fp_Bigg_middle_shift_int { 20 0000 * 9999 }
13399 \int_const:Nn \c_fp_Bigg_trailing_shift_int { 20 0000 * 10000 }
13400 \cs_new:Npn \_fp_pack_Bigg:NNNNNNw #1#2 #3#4#5#6 #7;
13401 { + #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:wNNNNNNNN tokens ;  $\geq 8$  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.

```
13402 \cs_new:Npn \_fp_pack_twice_four:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
13403 { #1 {#2#3#4#5} {#6#7#8#9} ; }
```

(End definition for `\_fp_pack_twice_four:wNNNNNNNN`.)

```
\_fp_pack_eight:wNNNNNNNN
```

`\_fp_pack_eight:wNNNNNNNN tokens ;  $\geq 8$  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.

```
13404 \cs_new:Npn \_fp_pack_eight:wNNNNNNNN #1; #2#3#4#5 #6#7#8#9
13405 { #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.

```
13406 \cs_new:Npn \_fp_basics_pack_low:NNNNw #1 #2#3#4#5 #6;
13407 { + #1 - 1 ; {#2#3#4#5} {#6} ; }
13408 \cs_new:Npn \_fp_basics_pack_high:NNNNw #1 #2#3#4#5 #6;
13409 { }
```

```

13410      \if_meaning:w 2 #1
13411          \_fp_basics_pack_high_carry:w
13412      \fi:
13413      ; {#2#3#4#5} {#6}
13414  }
13415 \cs_new:Npn \_fp_basics_pack_high_carry:w \fi: ; #1
13416  { \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:NNNNNNNw`

This is used in l3fp-basics for additions and divisions. Their syntax is confusing, hence the name.

```

13417 \cs_new:Npn \_fp_basics_pack_weird_low:NNNNw #1 #2#3#4 #5;
13418  {
13419      \if_meaning:w 2 #1
13420          + 1
13421      \fi:
13422      \_fp_int_eval_end:
13423      #2#3#4; {#5} ;
13424  }
13425 \cs_new:Npn \_fp_basics_pack_weird_high:NNNNNNNw
13426  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:NNNNNNNw.)

```

## 23.9 Decimate (dividing by a power of 10)

```
\_fp_decimate:nNnnn {\langle shift \rangle} {\langle f1 \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.  $\langle f1 \rangle$  is called as follows:

```
\langle f1 \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.

```

13427 \cs_new:Npn \_fp_decimate:nNnnn #1
13428  {
```

```

13429   \cs:w
13430     __fp_decimate_
13431     \if_int_compare:w __fp_int_eval:w #1 > \c__fp_prec_int
13432       tiny
13433     \else:
13434       __fp_int_to_roman:w __fp_int_eval:w #1
13435     \fi:
13436     :Nnnnn
13437   \cs_end:
13438 }

```

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.

```

13439 \cs_new:Npn __fp_decimate_:Nnnnn #1 #2#3#4#5
13440   { #1 0 {#2#3} {#4#5} ; }
13441 \cs_new:Npn __fp_decimate_tiny:Nnnnn #1 #2#3#4#5
13442   { #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>8</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.

```

13443 \cs_new:Npn __fp_tmp:w #1 #2 #3
13444   {
13445     \cs_new:cpn { __fp_decimate_ #1 :Nnnnn } ##1 ##2##3##4##5
13446     {
13447       \exp_after:wN ##1
13448       \int_value:w
13449       \exp_after:wN __fp_round_digit:Nw #2 ;
13450       __fp_decimate_pack:nnnnnnnnnw #3 ;
13451     }
13452   }
13453 __fp_tmp:w {i}  {\use_none:nnn      #50}{ 0{#2}#3{#4}#5}
13454 __fp_tmp:w {ii} {\use_none:nn      #5 }{ 00{#2}#3{#4}#5}
13455 __fp_tmp:w {iii} {\use_none:n      #5 }{ 000{#2}#3{#4}#5}
13456 __fp_tmp:w {iv}  {                  #5 }{ {0000}#2{#3}#4 #5}
13457 __fp_tmp:w {v}   {\use_none:nnn    #4#5}{ 0{0000}#2{#3}#4 #5}
13458 __fp_tmp:w {vi}  {\use_none:nn      #4#5}{ 00{0000}#2{#3}#4 #5}
13459 __fp_tmp:w {vii} {\use_none:n      #4#5}{ 000{0000}#2{#3}#4 #5}
13460 __fp_tmp:w {viii}{                  #4#5}{ {0000}0000{#2}#3 #4 #5}
13461 __fp_tmp:w {ix}  {\use_none:nnn   #3#4+#5}{ 0{0000}0000{#2}#3 #4 #5}

```

---

<sup>8</sup>No, the argument spec is not a mistake: the function calls an auxiliary to do half of the job.

```

13462 \__fp_tmp:w {x}   {\use_none:nn #3#4+#5}{ 00{0000}0000{#2}#3 #4 #5      }
13463 \__fp_tmp:w {xi}  {\use_none:n  #3#4+#5}{ 000{0000}0000{#2}#3 #4 #5      }
13464 \__fp_tmp:w {xii} {           #3#4+#5}{ {0000}0000{0000}#2 #3 #4 #5      }
13465 \__fp_tmp:w {xiii}{\use_none:nnn#2#3+#4#5}{ 0{0000}0000{0000}#2 #3 #4 #5      }
13466 \__fp_tmp:w {xiv} {\use_none:nn #2#3+#4#5}{ 00{0000}0000{0000}#2 #3 #4 #5      }
13467 \__fp_tmp:w {xv}  {\use_none:n #2#3+#4#5}{ 000{0000}0000{0000}#2 #3 #4 #5      }
13468 \__fp_tmp:w {xvi} {           #2#3+#4#5}{ {0000}0000{0000}0000 #2 #3 #4 #5}

```

(End definition for `\__fp_decimate_auxi:Nnnnn` and others.)

`\__fp_decimate_pack:nnnnnnnnnw`

The computation of the `\langle 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.

```

13469 \cs_new:Npn \__fp_decimate_pack:nnnnnnnnnw #1#2#3#4#5
13470   { \__fp_decimate_pack:nnnnnnnw { #1#2#3#4#5 } }
13471 \cs_new:Npn \__fp_decimate_pack:nnnnnnw #1 #2#3#4#5#6
13472   { {#1} {#2#3#4#5#6} }

```

(End definition for `\__fp_decimate_pack:nnnnnnnnnnw`.)

## 23.10 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.

```
13473 \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.

```
13474 \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.

```
13475 \cs_new:Npn \_fp_case_return_o:Nw #1#2 \fi: #3 \s_fp #4 ;  
13476 { \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.

```
13477 \cs_new:Npn \_fp_case_return_same_o:w #1 \fi: #2 \s_fp  
13478 { \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.

```
13479 \cs_new:Npn \_fp_case_return_o:Nww #1#2 \fi: #3 \s_fp #4 ; #5 ;  
13480 { \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.

```
13481 \cs_new:Npn \_fp_case_return_i_o:ww #1 \fi: #2 \s_fp #3 ; \s_fp #4 ;  
13482 { \fi: \_fp_exp_after_o:w \s_fp #3 ; }  
13483 \cs_new:Npn \_fp_case_return_ii_o:ww #1 \fi: #2 \s_fp #3 ;  
13484 { \fi: \_fp_exp_after_o:w }
```

(End definition for \\_fp\_case\_return\_i\_o:ww and \\_fp\_case\_return\_ii\_o:ww.)

## 23.11 Integer floating points

`\_fp_int_p:w` 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.

```

13485 \prg_new_conditional:Npnn \_fp_int:w \s_fp \_fp_chk:w #1 #2 #3 #4;
13486   { TF , T , F , p }
13487   {
13488     \if_case:w #1 \exp_stop_f:
13489       \prg_return_true:
13490     \or:
13491       \if_charcode:w 0
13492         \_fp_decimate:nNnnnn { \c_fp_prec_int - #3 }
13493           \_fp_use_i_until_s:nw #4
13494         \prg_return_true:
13495       \else:
13496         \prg_return_false:
13497       \fi:
13498     \else: \prg_return_false:
13499     \fi:
13500   }
```

(End definition for `\_fp_int:wTF`.)

## 23.12 Small integer floating points

`\_fp_small_int:wTF` Tests if the floating point argument is an integer or  $\pm\infty$ . If so, it is converted 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  $10^8$ .

```

13501 \cs_new:Npn \_fp_small_int:wTF \s_fp \_fp_chk:w #1#2
13502   {
13503     \if_case:w #1 \exp_stop_f:
13504       \_fp_case_return:nw { \_fp_small_int_true:wTF 0 ; }
13505     \or: \exp_after:wN \_fp_small_int_normal:NnwTF
13506     \or:
13507       \_fp_case_return:nw
13508       {
13509         \exp_after:wN \_fp_small_int_true:wTF \int_value:w
13510           \if_meaning:w 2 #2 - \fi: 1 0000 0000 ;
13511       }
13512     \else: \_fp_case_return:nw \use_i:nn
13513     \fi:
13514     #2
13515   }
13516 \cs_new:Npn \_fp_small_int_true:wTF #1; #2#3 { #2 {#1} }
13517 \cs_new:Npn \_fp_small_int_normal:NnwTF #1#2#3;
13518   {
13519     \_fp_decimate:nNnnnn { \c_fp_prec_int - #2 }
13520       \_fp_small_int_test:NnnwNw
13521       #3 #1
13522   }
```

```

13523 \cs_new:Npn \__fp_small_int_test:NnnwNw #1#2#3#4; #5
13524 {
13525     \if_meaning:w 0 #1
13526         \exp_after:wN \__fp_small_int_true:wTF
13527         \int_value:w \if_meaning:w 2 #5 - \fi:
13528             \if_int_compare:w #2 > 0 \exp_stop_f:
13529                 1 0000 0000
13530             \else:
13531                 #3
13532             \fi:
13533         \exp_after:wN ;
13534     \else:
13535         \exp_after:wN \use_i:nn
13536     \fi:
13537 }

```

(End definition for `\__fp_small_int:wTF` and others.)

### 23.13 x-like expansion expandably

```
\__fp_expand:n
\__fp_expand_loop:nwnN
```

This expandable function behaves in a way somewhat similar to `\use:x`, but much less robust. The argument is f-expanded, then the leading item (often a single character token) is moved to a storage area after `\s_fp_mark`, and f-expansion is applied again, repeating until the argument is empty. The result built one piece at a time is then inserted in the input stream. Note that spaces are ignored by this procedure, unless surrounded with braces. Multiple tokens which do not need expansion can be inserted within braces.

```

13538 \cs_new:Npn \__fp_expand:n #1
13539 {
13540     \__fp_expand_loop:nwnN { }
13541     #1 \prg_do_nothing:
13542     \s_fp_mark { } \__fp_expand_loop:nwnN
13543     \s_fp_mark { } \__fp_use_i_until_s:nw ;
13544 }
13545 \cs_new:Npn \__fp_expand_loop:nwnN #1#2 \s_fp_mark #3 #4
13546 {
13547     \exp_after:wN #4 \exp:w \exp_end_continue_f:w
13548     #2
13549     \s_fp_mark { #3 #1 } #4
13550 }

```

(End definition for `\__fp_expand:n` and `\__fp_expand_loop:nwnN`.)

### 23.14 Fast string comparison

```
\__fp_str_if_eq:nn
```

A private version of the low-level string comparison function. As the nature of the arguments is restricted and as speed is of the essence, this version does not seek to deal with # tokens. No l3sys or l3luatex just yet so we have to define in terms of primitives.

```

13551 \cs_new:Npn \__fp_str_if_eq:nn #1#2 { \tex_strcmp:D {#1} {#2} }
13552 \sys_if_engine_luatex:T
13553 {
13554     \cs_set:Npn \__fp_str_if_eq:nn #1#2
13555     {

```

```

13556          \tex_directlua:D
13557          {
13558              13kernel.strcmp
13559              (
13560                  " \tex_luaescapestring:D {#1}",
13561                  " \tex_luaescapestring:D {#2}"
13562              )
13563          }
13564      }
13565  }

```

(End definition for `\_fp_str_if_eq:nn`.)

### 23.15 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.

```

13566  \cs_new:Npn \_fp_func_to_name:N #1
13567  {
13568      \exp_last_unbraced:Nf
13569          \_fp_func_to_name_aux:w { \cs_to_str:N #1 } X
13570      }
13571 \cs_set_protected:Npn \_fp_tmp:w #1 #2
13572  { \cs_new:Npn \_fp_func_to_name_aux:w ##1 #1 ##2 #2 ##3 X {##2} }
13573 \exp_args:Nff \_fp_tmp:w { \tl_to_str:n { _fp_ } }
13574  { \tl_to_str:n { _o: } }

```

(End definition for `\_fp_func_to_name:N` and `\_fp_func_to_name_aux:w`.)

### 23.16 Messages

Using a floating point directly is an error.

```

13575 \_kernel_msg_new:nnnn { kernel } { misused-fp }
13576  { A-floating-point-with-value-'#1'-was-misused. }
13577  {
13578      To-obtain-the-value-of-a-floating-point-variable,-use-
13579      '\token_to_str:N \fp_to_decimal:N',-
13580      '\token_to_str:N \fp_to_tl:N',-or-other-
13581      conversion-functions.
13582  }
13583 </initex | package>

```

## 24 l3fp-traps Implementation

```

13584 <*initex | package>
13585 <@@=fp>

```

Exceptions should be accessed by an n-type argument, among

- `invalid_operation`
- `division_by_zero`
- `overflow`

- `underflow`
- `inexact` (actually never used).

## 24.1 Flags

`flag_fp_invalid_operation`

`flag_fp_division_by_zero`

`flag_fp_overflow`

`flag_fp_underflow`

```
13586 \flag_new:n { fp_invalid_operation }
13587 \flag_new:n { fp_division_by_zero }
13588 \flag_new:n { fp_overflow }
13589 \flag_new:n { fp_underflow }
```

(End definition for flag `fp_invalid_operation` and others. These variables are documented on page 193.)

## 24.2 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_tl_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`

```
13590 \cs_new_protected:Npn \fp_trap:nn #1#2
13591   {
13592     \cs_if_exist_use:cF { __fp_trap_#1_set_#2: }
13593     {
13594       \clist_if_in:nnTF
13595         { invalid_operation , division_by_zero , overflow , underflow }
13596         {#1}
13597     }
```

```

13598     \_\_kernel\_msg\_error:nxxx { kernel }
13599         { unknown-fpu-trap-type } {#1} {#2}
13600     }
13601     {
13602         \_\_kernel\_msg\_error:nnx
13603             { kernel } { unknown-fpu-exception } {#1}
13604     }
13605 }
13606 }
```

(End definition for `\fp_trap:nn`. This function is documented on page 193.)

```
\_\_fp_trap_invalid_operation_set_error:
\_\_fp_trap_invalid_operation_set_flag:
\_\_fp_trap_invalid_operation_set_none:
\_\_fp_trap_invalid_operation_set:N
```

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.

```

13607 \cs_new_protected:Npn \_\_fp_trap_invalid_operation_set_error:
13608     { \_\_fp_trap_invalid_operation_set:N \prg_do_nothing: }
13609 \cs_new_protected:Npn \_\_fp_trap_invalid_operation_set_flag:
13610     { \_\_fp_trap_invalid_operation_set:N \use_none:nnnnn }
13611 \cs_new_protected:Npn \_\_fp_trap_invalid_operation_set_none:
13612     { \_\_fp_trap_invalid_operation_set:N \use_none:nnnnnnn }
13613 \cs_new_protected:Npn \_\_fp_trap_invalid_operation_set:N #1
13614     {
13615         \exp_args:Nno \use:n
13616             { \cs_set:Npn \_\_fp_invalid_operation:nw ##1##2##3; }
13617             {
13618                 #1
13619                 \_\_fp_error:nfn { fp-invalid } {##2} { \fp_to_tl:n { ##3; } } { }
13620                 \flag_raise_if_clear:n { fp_invalid_operation }
13621                 ##1
13622             }
13623         \exp_args:Nno \use:n
13624             { \cs_set:Npn \_\_fp_invalid_operation_o:Nww ##1##2; ##3; }
13625             {
13626                 #1
13627                 \_\_fp_error:nffn { fp-invalid-ii }
13628                     { \fp_to_tl:n { ##2; } } { \fp_to_tl:n { ##3; } } {##1}
13629                     \flag_raise_if_clear:n { fp_invalid_operation }
13630                     \exp_after:wN \c_nan_fp
13631             }
13632         \exp_args:Nno \use:n
13633             { \cs_set:Npn \_\_fp_invalid_operation_tl_o:ff ##1##2 }
13634             {
13635                 #1
13636                 \_\_fp_error:nffn { fp-invalid } {##1} {##2} { }
13637                 \flag_raise_if_clear:n { fp_invalid_operation }
13638                 \exp_after:wN \c_nan_fp
13639             }
13640 }
```

(End definition for `\_\_fp_trap_invalid_operation_set_error:` and others.)

```
\_\_fp_trap_division_by_zero_set_error:
\_\_fp_trap_division_by_zero_set_flag:
\_\_fp_trap_division_by_zero_set_none:
\_\_fp_trap_division_by_zero_set:N
```

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.

```

13641 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_error:
13642   { \__fp_trap_division_by_zero_set:N \prg_do_nothing: }
13643 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_flag:
13644   { \__fp_trap_division_by_zero_set:N \use_none:nnnnn }
13645 \cs_new_protected:Npn \__fp_trap_division_by_zero_set_none:
13646   { \__fp_trap_division_by_zero_set:N \use_none:nnnnnnn }
13647 \cs_new_protected:Npn \__fp_trap_division_by_zero_set:N #1
13648   {
13649     \exp_args:Nno \use:n
13650     { \cs_set:Npn \__fp_division_by_zero_o:Nnw ##1##2##3; }
13651   {
13652     #1
13653     \__fp_error:nnfn { fp-zero-div } {##2} { \fp_to_tl:n { ##3; } } { }
13654     \flag_raise_if_clear:n { fp_division_by_zero }
13655     \exp_after:wN ##1
13656   }
13657 \exp_args:Nno \use:n
13658   { \cs_set:Npn \__fp_division_by_zero_o>NNww ##1##2##3; ##4; }
13659   {
13660     #1
13661     \__fp_error:nffn { fp-zero-div-ii }
13662     { \fp_to_tl:n { ##3; } } { \fp_to_tl:n { ##4; } } {##2}
13663     \flag_raise_if_clear:n { fp_division_by_zero }
13664     \exp_after:wN ##1
13665   }
13666 }
```

(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 T<sub>E</sub>X, 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.

```

13667 \cs_new_protected:Npn \__fp_trap_overflow_set_error:
13668   { \__fp_trap_overflow_set:N \prg_do_nothing: }
13669 \cs_new_protected:Npn \__fp_trap_overflow_set_flag:
13670   { \__fp_trap_overflow_set:N \use_none:nnnnn }
13671 \cs_new_protected:Npn \__fp_trap_overflow_set_none:
13672   { \__fp_trap_overflow_set:N \use_none:nnnnnnn }
13673 \cs_new_protected:Npn \__fp_trap_overflow_set:N #1
13674   { \__fp_trap_overflow_set:NnNn #1 { overflow } \__fp_inf_fp:N { inf } }
13675 \cs_new_protected:Npn \__fp_trap_underflow_set_error:
13676   { \__fp_trap_underflow_set:N \prg_do_nothing: }
13677 \cs_new_protected:Npn \__fp_trap_underflow_set_flag:
13678   { \__fp_trap_underflow_set:N \use_none:nnnnn }
13679 \cs_new_protected:Npn \__fp_trap_underflow_set_none:
```

```

13680 { \__fp_trap_underflow_set:N \use_none:nnnnnnn }
13681 \cs_new_protected:Npn \__fp_trap_underflow_set:N #1
13682 { \__fp_trap_overflow_set:NnNn #1 { underflow } \__fp_zero_fp:N { 0 } }
13683 \cs_new_protected:Npn \__fp_trap_overflow_set:NnNn #1#2#3#4
13684 {
13685     \exp_args:Nno \use:n
13686     { \cs_set:cpn { __fp_#2 :w } \s__fp \__fp_chk:w ##1##2##3; }
13687     {
13688         #1
13689         \__fp_error:nffn
13690         { fp-flow \if_meaning:w 1 ##1 -to \fi: }
13691         { \fp_to_tl:n { \s__fp \__fp_chk:w ##1##2##3; } }
13692         { \token_if_eq_meaning:NNF 0 ##2 { - } #4 }
13693         {#2}
13694         \flag_raise_if_clear:n { fp_#2 }
13695         #3 ##2
13696     }
13697 }

```

(End definition for `\__fp_trap_overflow_set_error:` and others.)

`\__fp_invalid_operation:nw` 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.

```

\__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

```

```

13698 \cs_new:Npn \__fp_invalid_operation:nw #1#2#3; { }
13699 \cs_new:Npn \__fp_invalid_operation_o:Nww #1#2; #3; { }
13700 \cs_new:Npn \__fp_invalid_operation_tl_o:ff #1 #2 { }
13701 \cs_new:Npn \__fp_division_by_zero_o:Nnw #1#2#3; { }
13702 \cs_new:Npn \__fp_division_by_zero_o>NNww #1#2#3; #4; { }
13703 \cs_new:Npn \__fp_overflow:w { }
13704 \cs_new:Npn \__fp_underflow:w { }
13705 \fp_trap:nn { invalid_operation } { error }
13706 \fp_trap:nn { division_by_zero } { flag }
13707 \fp_trap:nn { overflow } { flag }
13708 \fp_trap:nn { underflow } { flag }

```

(End definition for `\__fp_invalid_operation:nw` and others.)

`\__fp_invalid_operation_o:nw` Convenient short-hands for returning `\c_nan_fp` for a unary or binary operation, and expanding after.

```

\__fp_invalid_operation_o:fw

```

```

13709 \cs_new:Npn \__fp_invalid_operation_o:nw
13710 { \__fp_invalid_operation:nw { \exp_after:wN \c_nan_fp } }
13711 \cs_generate_variant:Nn \__fp_invalid_operation_o:nw { f }

```

(End definition for `\__fp_invalid_operation_o:nw`.)

## 24.3 Errors

```

\__fp_error:nnnn
\__fp_error:nnfn
\__fp_error:nffn
\__fp_error:nfff

```

```

13712 \cs_new:Npn \__fp_error:nnnn
13713 { \__kernel_msg_expandable_error:nnnnn { kernel } }
13714 \cs_generate_variant:Nn \__fp_error:nnnn { nnf, nff , nfff }

```

(End definition for `\__fp_error:nnnn`.)

## 24.4 Messages

Some messages.

```
13715 \_kernel_msg_new:nnn { kernel } { unknown-fpu-exception }
13716 {
13717     The~FPU~exception~'#1'~is~not~known:~
13718     that~trap~will~never~be~triggered.
13719 }
13720 {
13721     The~only~exceptions~to~which~traps~can~be~attached~are \\
13722     \iow_indent:n
13723 {
13724     * ~ invalid_operation \\
13725     * ~ division_by_zero \\
13726     * ~ overflow \\
13727     * ~ underflow
13728 }
13729 }
13730 \_kernel_msg_new:nnn { kernel } { unknown-fpu-trap-type }
13731 {
13732     The~FPU~trap~type~'#2'~is~not~known. }
13733 {
13734     The~trap~type~must~be~one~of \\
13735     \iow_indent:n
13736 {
13737     * ~ error \\
13738     * ~ flag \\
13739     * ~ none
13740 }
13741 \_kernel_msg_new:nnn { kernel } { fp-flow }
13742 {
13743     An ~ #3 ~ occurred. }
13744 \_kernel_msg_new:nnn { kernel } { fp-flow-to }
13745 {
13746     #1 ~ #3 ed ~ to ~ #2 . }
13747 \_kernel_msg_new:nnn { kernel } { fp-zero-div }
13748 {
13749     Division~by~zero~in~ #1 (#2) }
13750 \_kernel_msg_new:nnn { kernel } { fp-zero-div-ii }
13751 {
13752     Division~by~zero~in~ (#1) #3 (#2) }
13753 \_kernel_msg_new:nnn { kernel } { fp-invalid }
13754 {
13755     Invalid~operation~ #1 (#2) }
13756 \_kernel_msg_new:nnn { kernel } { fp-invalid-ii }
13757 {
13758     Invalid~operation~ (#1) #3 (#2) }
13759 \_kernel_msg_new:nnn { kernel } { fp-unknown-type }
13760 {
13761     Unknown~type~for~'#1' }
13762 
```

## 25 I3fp-round implementation

```
13756 {*initex | package}
13757 (@@=fp)

\__fp_parse_word_trunc:N
\__fp_parse_word_floor:N
\__fp_parse_word_ceil:N
13758 \cs_new:Npn \__fp_parse_word_trunc:N
13759 {
    \__fp_parse_function:NNN \__fp_round_o:Nw \__fp_round_to_zero:NNN }
```

```

13760 \cs_new:Npn \__fp_parse_word_floor:N
13761   { \__fp_parse_function>NNN \__fp_round_o:Nw \__fp_round_to_ninf>NNN }
13762 \cs_new:Npn \__fp_parse_word_ceil:N
13763   { \__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

```

13764 \cs_new:Npn \__fp_parse_word_round:N #1#2
13765 {
13766   \__fp_parse_function>NNN
13767   \__fp_round_o:Nw \__fp_round_to_nearest>NNN #1
13768   #2
13769 }
13770 \cs_new:Npn \__fp_parse_round:Nw #1 #2 \__fp_round_to_nearest>NNN #3#4
13771   { #2 #1 #3 }
13772
```

(End definition for \\_\_fp\_parse\_word\_round:N and \\_\_fp\_parse\_round:Nw.)

## 25.1 Rounding tools

\c\_fp\_five\_int This is used as the half-point for which numbers are rounded up/down.

```
13773 \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  $\langle sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle$  can expand to 0\exp\_stop\_f: or 1\exp\_stop\_f:.

- `\_fp_round_s:NNNw` (*sign*)  $\langle digit_1 \rangle \langle digit_2 \rangle \langle more\ digits \rangle$ ; can expand to `0\exp_stop_f:;` or `1\exp_stop_f:;`.
- `\_fp_round_neg:NNN` (*sign*)  $\langle digit_1 \rangle \langle digit_2 \rangle$  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
```

`\_fp_round:NNN` (*final sign*)  $\langle digit_1 \rangle \langle digit_2 \rangle$

If rounding the number  $\langle final\ sign \rangle \langle digit_1 \rangle \langle digit_2 \rangle$  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  $\langle final\ sign \rangle$  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  $\langle final\ sign \rangle$  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  $\langle digit_2 \rangle$  is larger than 5, then round up. If it is less than 5, round down. If it is exactly 5, then round such that  $\langle digit_1 \rangle$  plus the result is even. In other words, round up if  $\langle digit_1 \rangle$  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  $+\infty$ .

```
13774 \cs_new:Npn \_fp_round_return_one:
13775   { \exp_after:wN 1 \exp_after:wN \exp_stop_f: \exp:w }
13776 \cs_new:Npn \_fp_round_to_ninf:NNN #1 #2 #3
13777   {
13778     \if_meaning:w 2 #1
13779       \if_int_compare:w #3 > 0 \exp_stop_f:
13780         \_fp_round_return_one:
13781       \fi:
13782     \fi:
13783     0 \exp_stop_f:
13784   }
13785 \cs_new:Npn \_fp_round_to_zero:NNN #1 #2 #3 { 0 \exp_stop_f: }
13786 \cs_new:Npn \_fp_round_to_pinf:NNN #1 #2 #3
13787   {
13788     \if_meaning:w 0 #1
13789       \if_int_compare:w #3 > 0 \exp_stop_f:
13790         \_fp_round_return_one:
13791       \fi:
13792     \fi:
13793     0 \exp_stop_f:
13794   }
13795 \cs_new:Npn \_fp_round_to_nearest:NNN #1 #2 #3
13796   {
13797     \if_int_compare:w #3 > \c_fp_five_int
13798       \_fp_round_return_one:
13799     \else:
```

```

13800   \if_meaning:w 5 #3
13801     \if_int_odd:w #2 \exp_stop_f:
13802       \_fp_round_return_one:
13803     \fi:
13804   \fi:
13805   \fi:
13806   0 \exp_stop_f:
13807 }
13808 \cs_new:Npn \_fp_round_to_nearest_ninf:NNN #1 #2 #3
13809 {
13810   \if_int_compare:w #3 > \c_fp_five_int
13811     \_fp_round_return_one:
13812   \else:
13813     \if_meaning:w 5 #3
13814       \if_meaning:w 2 #1
13815         \_fp_round_return_one:
13816       \fi:
13817     \fi:
13818   \fi:
13819   0 \exp_stop_f:
13820 }
13821 \cs_new:Npn \_fp_round_to_nearest_zero:NNN #1 #2 #3
13822 {
13823   \if_int_compare:w #3 > \c_fp_five_int
13824     \_fp_round_return_one:
13825   \fi:
13826   0 \exp_stop_f:
13827 }
13828 \cs_new:Npn \_fp_round_to_nearest_pinf:NNN #1 #2 #3
13829 {
13830   \if_int_compare:w #3 > \c_fp_five_int
13831     \_fp_round_return_one:
13832   \else:
13833     \if_meaning:w 5 #3
13834       \if_meaning:w 0 #1
13835         \_fp_round_return_one:
13836       \fi:
13837     \fi:
13838   \fi:
13839   0 \exp_stop_f:
13840 }
13841 \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.

```

```

13842 \cs_new:Npn \_fp_round_s:NNNw #1 #2 #3 #4;
13843 {
13844   \exp_after:wN \_fp_round:NNN

```

```

13845   \exp_after:wN #1
13846   \exp_after:wN #2
13847   \int_value:w \__fp_int_eval:w
13848     \if_int_odd:w 0 \if_meaning:w 0 #3 1 \fi:
13849       \if_meaning:w 5 #3 1 \fi:
13850         \exp_stop_f:
13851         \if_int_compare:w \__fp_int_eval:w #4 > 0 \exp_stop_f:
13852           1 +
13853           \fi:
13854         \fi:
13855       #3
13856     ;
13857   }

```

(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.

```

13858 \cs_new:Npn \__fp_round_digit:Nw #1 #2;
13859   {
13860     \if_int_odd:w \if_meaning:w 0 #1 1 \else:
13861       \if_meaning:w 5 #1 1 \else:
13862         0 \fi: \fi: \exp_stop_f:
13863         \if_int_compare:w \__fp_int_eval:w #2 > 0 \exp_stop_f:
13864           \__fp_int_eval:w 1 +
13865           \fi:
13866         \fi:
13867       #1
13868   }

```

(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_neg:NNN <final sign> <digit1> <digit2>`

This expands to `0\exp_stop_f:` or `1\exp_stop_f:` after doing the following test. Starting from a number of the form `<final sign>0.(15 digits)<digit1>` with exactly 15 (non-all-zero) digits before `<digit1>`, subtract from it `<final sign>0.0...0<digit2>`, 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.

```

13869 \cs_new_eq:NN \__fp_round_to_ninf_neg:NNN \__fp_round_to_pinf:NNN
13870 \cs_new:Npn \__fp_round_to_zero_neg:NNN #1 #2 #3
13871   {
13872     \if_int_compare:w #3 > 0 \exp_stop_f:
13873       \__fp_round_return_one:
13874     \fi:
13875     0 \exp_stop_f:
13876   }
13877 \cs_new_eq:NN \__fp_round_to_pinf_neg:NNN \__fp_round_to_ninf:NNN

```

```

13878 \cs_new_eq:NN \__fp_round_to_nearest_neg:NNN \__fp_round_to_nearest:NNN
13879 \cs_new_eq:NN \__fp_round_to_nearest_ninf_neg:NNN
13880   \__fp_round_to_nearest_pinf:NNN
13881 \cs_new:Npn \__fp_round_to_nearest_zero_neg:NNN #1 #2 #3
13882   {
13883     \if_int_compare:w #3 < \c_fp_five_int \else:
13884       \__fp_round_return_one:
13885     \fi:
13886     0 \exp_stop_f:
13887   }
13888 \cs_new_eq:NN \__fp_round_to_nearest_pinf_neg:NNN
13889   \__fp_round_to_nearest_ninf:NNN
13890 \cs_new_eq:NN \__fp_round_neg:NNN \__fp_round_to_nearest_neg:NNN

```

(End definition for `\__fp_round_neg:NNN` and others.)

## 25.2 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.

```

13891 \cs_new:Npn \__fp_round_o:Nw #1
13892   {
13893     \__fp_parse_function_all_fp_o:fnw
13894     { \__fp_round_name_from_cs:N #1 }
13895     { \__fp_round_aux_o:Nw #1 }
13896   }
13897 \cs_new:Npn \__fp_round_aux_o:Nw #1#2 @
13898   {
13899     \if_case:w
13900       \__fp_int_eval:w \__fp_array_count:n {#2} \__fp_int_eval_end:
13901         \__fp_round_no_arg_o:Nw #1 \exp:w
13902       \or: \__fp_round:Nwn #1 #2 {0} \exp:w
13903       \or: \__fp_round:Nww #1 #2 \exp:w
13904       \else: \__fp_round:Nwww #1 #2 @ \exp:w
13905     \fi:
13906     \exp_after:wN \exp_end:
13907   }

```

(End definition for `\__fp_round_o:Nw` and `\__fp_round_aux_o:Nw`.)

`\__fp_round_no_arg_o:Nw`

```

13908 \cs_new:Npn \__fp_round_no_arg_o:Nw #1
13909   {
13910     \cs_if_eq:NNTF #1 \__fp_round_to_nearest:NNN
13911       { \__fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 } }
13912       {
13913         \__fp_error:nffn { fp-num-args }
13914           { \__fp_round_name_from_cs:N #1 () } { 1 } { 2 }
13915       }
13916     \exp_after:wN \c_nan_fp
13917   }

```

(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.

```
13918 \cs_new:Npn \_fp_round:Nwww #1#2 ; #3 ; \s_fp \_fp_chk:w #4#5#6 ; #7 @
13919   {
13920     \cs_if_eq:NNTF #1 \_fp_round_to_nearest:NNN
13921     {
13922       \tl_if_empty:nTF {#7}
13923       {
13924         \exp_args:Nc \_fp_round:Nww
13925         {
13926           \_fp_round_to_nearest
13927           \if_meaning:w 0 #4 _zero \else:
13928             \if_case:w #5 \exp_stop_f: _pinf \or: \else: _ninf \fi: \fi:
13929             :NNN
13930         }
13931         #2 ; #3 ;
13932       }
13933     {
13934       \_fp_error:nnnn { fp-num-args } { round () } { 1 } { 3 }
13935       \exp_after:wN \c_nan_fp
13936     }
13937   }
13938   {
13939     \_fp_error:nffn { fp-num-args }
13940     { \_fp_round_name_from_cs:N #1 () } { 1 } { 2 }
13941     \exp_after:wN \c_nan_fp
13942   }
13943 }
```

(End definition for `\_fp_round:Nwww.`)

`\_fp_round_name_from_cs:N`

```
13944 \cs_new:Npn \_fp_round_name_from_cs:N #1
13945   {
13946     \cs_if_eq:NNTF #1 \_fp_round_to_zero:NNN { trunc }
13947     {
13948       \cs_if_eq:NNTF #1 \_fp_round_to_ninf:NNN { floor }
13949       {
13950         \cs_if_eq:NNTF #1 \_fp_round_to_pinfn:NNN { ceil }
13951         { round }
13952       }
13953     }
13954 }
```

(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_pack:Nw
\_fp_round_normal:NNwNnn
\_fp_round_normal_end:wwNnn
\_fp_round_special:NwwNnn
\_fp_round_special_aux:Nw
```

```

13958 {
13959     \__fp_invalid_operation_tl_o:ff
13960     { \__fp_round_name_from_cs:N #1 }
13961     { \__fp_array_to_clist:n { #2; #3; } }
13962 }
13963 }
13964 \cs_new:Npn \__fp_round:Nwn #1 \s__fp \__fp_chk:w #2#3#4; #5
13965 {
13966     \if_meaning:w 1 #2
13967         \exp_after:wN \__fp_round_normal:NwNNnw
13968         \exp_after:wN #1
13969         \int_value:w #5
13970     \else:
13971         \exp_after:wN \__fp_exp_after_o:w
13972     \fi:
13973     \s__fp \__fp_chk:w #2#3#4;
13974 }
13975 \cs_new:Npn \__fp_round_normal:NwNNnw #1#2 \s__fp \__fp_chk:w 1#3#4#5;
13976 {
13977     \__fp_decimate:nNnnnn { \c__fp_prec_int - #4 - #2 }
13978     \__fp_round_normal:NnnwNNnn #5 #1 #3 {#4} {#2}
13979 }
13980 \cs_new:Npn \__fp_round_normal:NnnwNNnn #1#2#3#4; #5#6
13981 {
13982     \exp_after:wN \__fp_round_normal>NNwNnn
13983     \int_value:w \__fp_int_eval:w
13984     \if_int_compare:w #2 > 0 \exp_stop_f:
13985         1 \int_value:w #2
13986         \exp_after:wN \__fp_round_pack:Nw
13987         \int_value:w \__fp_int_eval:w 1#3 +
13988     \else:
13989         \if_int_compare:w #3 > 0 \exp_stop_f:
13990             1 \int_value:w #3 +
13991         \fi:
13992     \fi:
13993     \exp_after:wN #5
13994     \exp_after:wN #6
13995     \use_none:nnnnnnn #3
13996     #1
13997     \__fp_int_eval_end:
13998     0000 0000 0000 0000 ; #6
13999 }
14000 \cs_new:Npn \__fp_round_pack:Nw #1
14001 { \if_meaning:w 2 #1 + 1 \fi: \__fp_int_eval_end: }
14002 \cs_new:Npn \__fp_round_normal>NNwNnn #1 #2
14003 {
14004     \if_meaning:w 0 #2
14005         \exp_after:wN \__fp_round_special:NwwNnn
14006         \exp_after:wN #1
14007     \fi:
14008     \__fp_pack_twice_four:wNNNNNNNNN
14009     \__fp_pack_twice_four:wNNNNNNNNN
14010     \__fp_round_normal_end:wwNnn
14011     ; #2

```

```

14012      }
14013 \cs_new:Npn \__fp_round_normal_end:wwNnn #1;#2;#3#4#5
14014  {
14015      \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
14016      \__fp_SANITIZE:Nw #3 #4 ; #1 ;
14017  }
14018 \cs_new:Npn \__fp_round_special:NwwNnn #1#2;#3;#4#5#6
14019  {
14020      \if_meaning:w 0 #1
14021          \__fp_case_return:nw
14022          { \exp_after:wN \__fp_zero_fp:N \exp_after:wN #4 }
14023      \else:
14024          \exp_after:wN \__fp_round_special_aux:Nw
14025          \exp_after:wN #4
14026          \int_value:w \__fp_int_eval:w 1
14027          \if_meaning:w 1 #1 -#6 \else: +#5 \fi:
14028      \fi:
14029      ;
14030  }
14031 \cs_new:Npn \__fp_round_special_aux:Nw #1#2;
14032  {
14033      \exp_after:wN \__fp_exp_after_o:w \exp:w \exp_end_continue_f:w
14034      \__fp_SANITIZE:Nw #1#2; {1000}{0000}{0000}{0000};
14035  }

```

(End definition for `\__fp_round:Nww` and others.)

14036 ⟨/initex | package⟩

## 26 l3fp-parse implementation

14037 ⟨\*initex | package⟩  
 14038 ⟨@@=fp⟩

### 26.1 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_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
```

Floating point expressions are composed of numbers, given in various forms, infix operators, such as `+`, `**`, or `,` (which joins two numbers into a list), and prefix operators, such as the unary `-`, functions, or opening parentheses. Here is a list of precedences which control the order of evaluation (some distinctions are irrelevant for the order of evaluation, but serve as signals), from the tightest binding to the loosest binding.

16 Function calls.

13/14 Binary `**` and `^` (right to left).

12 Unary `+`, `-`, `!` (right to left).

10 Binary `*`, `/`, and juxtaposition (implicit `*`).

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.

```
14039 \int_const:Nn \c_fp_prec_func_int { 16 }
14040 \int_const:Nn \c_fp_prec_hatii_int { 14 }
14041 \int_const:Nn \c_fp_prec_hat_int { 13 }
14042 \int_const:Nn \c_fp_prec_not_int { 12 }
14043 \int_const:Nn \c_fp_prec_times_int { 10 }
14044 \int_const:Nn \c_fp_prec_plus_int { 9 }
14045 \int_const:Nn \c_fp_prec_comp_int { 7 }
14046 \int_const:Nn \c_fp_prec_and_int { 6 }
14047 \int_const:Nn \c_fp_prec_or_int { 5 }
14048 \int_const:Nn \c_fp_prec_quest_int { 4 }
14049 \int_const:Nn \c_fp_prec_colon_int { 3 }
14050 \int_const:Nn \c_fp_prec_comma_int { 2 }
14051 \int_const:Nn \c_fp_prec_tuple_int { 1 }
14052 \int_const:Nn \c_fp_prec_end_int { 0 }
```

(End definition for `\c_fp_prec_func_int` and others.)

### 26.1.1 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.

### 26.1.2 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  $\langle \text{precedence} \rangle$  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  $\langle \text{number} \rangle$ , performing no computation, and finds the following binary  $\langle \text{operator} \rangle$ . Then it expands to

```
 $\langle \text{number} \rangle$ 
\_fp_parse_infix_\langle \text{operator} \rangle:N \langle \text{precedence} \rangle
```

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_\langle \text{operator} \rangle: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  $\langle \text{precedence} \rangle$  (of the earlier operator) to the `infix` auxiliary for the following  $\langle \text{operator} \rangle$ , to know whether to perform the computation of the  $\langle \text{operator} \rangle$ . If it should not be performed, the `infix` auxiliary expands to

```
@ \use_none:n \_fp_parse_infix_\langle \text{operator} \rangle:N
```

and otherwise it calls `\_fp_parse_operand:Nw` with the precedence of the  $\langle \text{operator} \rangle$  to find its second operand  $\langle \text{number}_2 \rangle$  and the next  $\langle \text{operator}_2 \rangle$ , and expands to

```
@ \_fp_parse_apply_binary:NwNwN
\langle \text{operator} \rangle \langle \text{number}_2 \rangle
@ \_fp_parse_infix_\langle \text{operator}_2 \rangle:N
```

The `infix` function is responsible for comparing precedences, but cannot directly call the computation functions, because the first operand  $\langle \text{number} \rangle$  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 \langle \text{precedence} \rangle` 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 `⟨number⟩ @ \fp_parse_infix_⟨operator⟩:N`. In the second case, #3 is `\fp_parse_apply_binary:NwNwN`, whose role is to compute `⟨number⟩ ⟨operator⟩ ⟨number₂⟩` and to prepare for the next comparison of precedences: first we get

```
\fp_parse_apply_binary:NwNwN
  ⟨precedence⟩ ⟨number⟩ @
  ⟨operator⟩ ⟨number₂⟩
@ \fp_parse_infix_⟨operator₂⟩:N
```

then

```
\exp_after:wN \fp_parse_continue:NwN
\exp_after:wN ⟨precedence⟩
\exp:w \exp_end_continue_f:w
\fp_⟨operator⟩_o:ww ⟨number⟩ ⟨number₂⟩
\exp:w \exp_end_continue_f:w
\fp_parse_infix_⟨operator₂⟩:N ⟨precedence⟩
```

where `\fp_⟨operator⟩_o:ww` computes `⟨number⟩ ⟨operator⟩ ⟨number₂⟩` and expands after the result, thus triggers the comparison of the precedence of the `⟨operator₂⟩` and the `⟨precedence⟩`, continuing the loop.

We have introduced the most important functions here, and the next few paragraphs we describe various subtleties.

### 26.1.3 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 `\_\_fp_parse_one:Nw` earlier. This function is responsible for reading in the input stream the first possible `<number>` and the next infix `<operator>`. If what follows `\_\_fp_parse_one:Nw <precedence>` is a prefix operator, then we must find the operand of this prefix operator through a nested call to `\_\_fp_parse_operand:Nw` with the appropriate precedence, then apply the operator to the operand found to yield the result of `\_\_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 `\_\_fp_parse_operand:Nw` with the `<precedence>` 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.

### 26.1.4 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 `<significand>e<exponent>`, where the `<significand>` is any non-empty string composed of decimal digits and at most one decimal separator (a period), the exponent “`e<exponent>`” 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 `<significand>` 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 `<exponent>` 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_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 *`<significand>`* 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_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  $\varepsilon$ -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`.

## 26.2 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>` which follows, and expands to

`@ \_fp_parse_apply_binary:NwNwN + <operand> @ \_fp_parse_infix_<operation>: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

```
⟨operand1⟩ © \__fp_parse_apply_binary:NwNwN ⟨operation⟩ ⟨operand2⟩ ©\__fp_parse_infix_⟨operation2⟩:N ...
```

and otherwise expands to

```
⟨operand⟩ © \use_none:n \__fp_parse_infix_⟨operation⟩:N ...
```

(End definition for `\__fp_parse_one:Nw`.)

### 26.3 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.

```
14053 \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.

```
14054 \cs_new:Npn \__fp_parse_return_semicolon:w  
14055     #1 \fi: \__fp_parse_expand:w { \fi: ; #1 }
```

(End definition for `\__fp_parse_return_semicolon:w`.)

```
\__fp_parse_digits_vii: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.

```
14056 \cs_set_protected:Npn \__fp_tmp:w #1 #2 #3  
14057     {  
14058         \cs_new:cpn { __fp_parse_digits_ #1 :N } ##1  
14059             {
```

```

14060     \if_int_compare:w 9 < 1 \token_to_str:N ##1 \exp_stop_f:
14061         \token_to_str:N ##1 \exp_after:wN #2 \exp:w
14062     \else:
14063         \__fp_parse_return_semicolon:w #3 ##1
14064     \fi:
14065     \__fp_parse_expand:w
14066 }
14067 }
14068 \__fp_tmp:w {vii} \__fp_parse_digits_vi:N { 0000000 ; 7 }
14069 \__fp_tmp:w {vi} \__fp_parse_digits_v:N { 000000 ; 6 }
14070 \__fp_tmp:w {v} \__fp_parse_digits_iv:N { 00000 ; 5 }
14071 \__fp_tmp:w {iv} \__fp_parse_digits_iii:N { 00000 ; 4 }
14072 \__fp_tmp:w {iii} \__fp_parse_digits_ii:N { 0000 ; 3 }
14073 \__fp_tmp:w {ii} \__fp_parse_digits_i:N { 000 ; 2 }
14074 \__fp_tmp:w {i} \__fp_parse_digits_:N { 00 ; 1 }
14075 \cs_new:Npn \__fp_parse_digits_:N { ; ; 0 }

(End definition for \__fp_parse_digits_vii:N and others.)

```

## 26.4 Parsing one number

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>$\epsilon$</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.

```

14076 \cs_new:Npn \__fp_parse_one:Nw #1 #2
14077 {
14078     \if_catcode:w \scan_stop: \exp_not:N #2
14079         \exp_after:wN \if_meaning:w \exp_not:N #2 #2 \else:
14080             \exp_after:wN \reverse_if:N
14081         \fi:
14082         \if_meaning:w \scan_stop: #2
14083             \exp_after:wN \exp_after:wN
14084             \exp_after:wN \__fp_parse_one_fp>NN
14085         \else:
14086             \exp_after:wN \exp_after:wN
14087             \exp_after:wN \__fp_parse_one_register>NN
14088         \fi:
14089     \else:
14090         \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
14091             \exp_after:wN \exp_after:wN
14092             \exp_after:wN \__fp_parse_one_digit>NN
14093         \else:
14094             \exp_after:wN \exp_after:wN
14095             \exp_after:wN \__fp_parse_one_other>NN
14096         \fi:
14097     \fi:
14098     #1 #2
14099 }
```

(End definition for `\_fp_parse_one:Nw`.)

This function receives a  $\langle\text{precedence}\rangle$  and a control sequence equal to `\scan_stop:` in meaning. There are three cases.

- `\s_fp` starts a floating point number, and we call `\_fp_exp_after_f:nw`, which `f`-expands after the floating point.
- `\s_fp_mark` is a premature end, we call `\_fp_exp_after_mark_f:nw`, which triggers an `fp-early-end` error.
- For a control sequence not containing `\s_fp`, 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 `\s_fp_<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>$\varepsilon$</sub>  uses `\@unexpandable@protect` rather than `\protect` because `\protect` is often `\scan_stop:` hence “does not exist”.

```

14100 \cs_new:Npn \_fp_parse_one_fp:NN #1
14101   {
14102     \_fp_exp_after_any_f:nw
14103     {
14104       \exp_after:wN \_fp_parse_infix:NN
14105       \exp_after:wN #1 \exp:w \_fp_parse_expand:w
14106     }
14107   }
14108 \cs_new:Npn \_fp_exp_after_mark_f:nw #1
14109   {
14110     \int_case:nnF { \exp_after:wN \use_i:nnn \use_none:nnn #1 }
14111     {
14112       \c_fp_prec_comma_int { }
14113       \c_fp_prec_tuple_int { }
14114       \c_fp_prec_end_int
14115       {
14116         \exp_after:wN \c_fp_empty_tuple_fp
14117         \exp:w \exp_end_continue_f:w
14118       }
14119     }
14120   {
14121     \__kernel_msg_expandable_error:nn { kernel } { fp-early-end }
14122     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
14123   }
14124   #1
14125 }
14126 \cs_new:cpn { \_fp_exp_after_?_f:nw } #1#2
14127   {
14128     \__kernel_msg_expandable_error:nnn { kernel } { bad-variable }
14129     {#2}
14130     \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w #1
14131   }
14132 (*package)

```

```

14133 \cs_set_protected:Npn \_fp_tmp:w #1
14134 {
14135   \cs_if_exist:NT #1
14136   {
14137     \cs_gset:cpn { _fp_exp_after_?_f:nw } ##1##2
14138   }
14139   \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w ##1##2
14140   \str_if_eq:nnTF {##2} { \protect }
14141   {
14142     \cs_if_eq:NNTF ##2 #1 { \use_i:nn } { \use:n }
14143   {
14144     \_kernel_msg_expandable_error:nnn { kernel }
14145     { fp-robust-cmd }
14146   }
14147   }
14148   {
14149     \_kernel_msg_expandable_error:nnn { kernel }
14150     { bad-variable } {##2}
14151   }
14152 }
14153 }
14154 }
14155 \exp_args:Nc \_fp_tmp:w { @unexpandable@protect }
14156 </package>

```

(End definition for `\_fp_parse_one_fp>NN`, `\_fp_exp_after_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 `<value>e<exponent>` 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 { <decimal value> pt }`, and use an auxiliary of `\dim_to_fp:n`, which performs the multiplication by  $2^{-16}$ , correctly rounded.

```

14157 \cs_new:Npn \_fp_parse_one_register>NN #1#
14158 {
14159   \exp_after:wN \_fp_parse_infix_after_operand:NwN
14160   \exp_after:wN #1
14161   \exp:w \exp_end_continue_f:w
14162   \_fp_parse_one_register_special:N #2
14163   \exp_after:wN \_fp_parse_one_register_aux:Nw
14164   \exp_after:wN #2
14165   \int_value:w
14166   \exp_after:wN \_fp_parse_exponent:N
14167   \exp:w \_fp_parse_expand:
14168 }
14169 \cs_new:Npx \_fp_parse_one_register_aux:Nw #1
14170 {
14171   \exp_not:n

```

```

14172 {
14173   \exp_after:wN \use:nn
14174   \exp_after:wN \_fp_parse_one_register_auxii:wwwNw
14175 }
14176 \exp_not:N \exp_after:wN { \exp_not:N \tex_the:D #1 }
14177   ; \exp_not:N \_fp_parse_one_register_dim:ww
14178   \tl_to_str:n { pt } ; \exp_not:N \_fp_parse_one_register_mu:www
14179   . \tl_to_str:n { pt } ; \exp_not:N \_fp_parse_one_register_int:www
14180   \exp_not:N \q_stop
14181 }
14182 \exp_args:Nno \use:nn
14183   { \cs_new:Npn \_fp_parse_one_register_auxii:wwwNw #1 . #2 }
14184   { \tl_to_str:n { pt } #3 ; #4#5 \q_stop }
14185   { #4 #1.#2; }
14186 \exp_args:Nno \use:nn
14187   { \cs_new:Npn \_fp_parse_one_register_mu:www #1 }
14188   { \tl_to_str:n { mu } ; #2 ; }
14189   { \_fp_parse_one_register_dim:ww #1 ; }
14190 \cs_new:Npn \_fp_parse_one_register_int:www #1; #2.; #3;
14191   { \_fp_parse:n { #1 e #3 } }
14192 \cs_new:Npn \_fp_parse_one_register_dim:ww #1; #2;
14193   {
14194     \exp_after:wN \_fp_from_dim_test:ww
14195     \int_value:w #2 \exp_after:wN ,
14196     \int_value:w \dim_to_decimal_in_sp:n { #1 pt } ;
14197 }

```

(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.

```

14198 \cs_new:Npn \_fp_parse_one_register_special:N #1
14199   {
14200     \if_meaning:w \box_wd:N #1 \_fp_parse_one_register_wd:w \fi:
14201     \if_meaning:w \box_ht:N #1 \_fp_parse_one_register_wd:w \fi:
14202     \if_meaning:w \box_dp:N #1 \_fp_parse_one_register_wd:w \fi:
14203     \if_meaning:w \infty #1
14204       \_fp_parse_one_register_math>NNw \infty #1
14205     \fi:
14206     \if_meaning:w \pi #1
14207       \_fp_parse_one_register_math>NNw \pi #1
14208     \fi:
14209   }
14210 \cs_new:Npn \_fp_parse_one_register_math>NNw
14211   #1#2#3#4 \_fp_parse_expand:w
14212   {
14213     #3
14214     \str_if_eq:nnTF {#1} {#2}
14215     {
14216       \_kernel_msg_expandable_error:nnn
14217         { kernel } { fp-infty-pi } {#1}
14218       \c_nan_fp

```

```

14219      }
14220      { #4 \__fp_parse_expand:w }
14221  }
14222 \cs_new:Npn \__fp_parse_one_register_wd:w
14223     #1#2 \exp_after:wN #3#4 \__fp_parse_expand:w
14224  {
14225      #1
14226      \exp_after:wN \__fp_parse_one_register_wd:Nw
14227      #4 \__fp_parse_expand:w e
14228  }
14229 \cs_new:Npn \__fp_parse_one_register_wd:Nw #1#2 ;
14230  {
14231      \exp_after:wN \__fp_from_dim_test:ww
14232      \exp_after:wN 0 \exp_after:wN ,
14233      \int_value:w \dim_to_decimal_in_sp:n { #1 #2 } ;
14234  }

```

(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.

```

14235 \cs_new:Npn \__fp_parse_one_digit>NN #1
14236  {
14237      \exp_after:wN \__fp_parse_infix_after_operand:NwN
14238      \exp_after:wN #1
14239      \exp:w \exp_end_continue_f:w
14240      \exp_after:wN \__fp_sanitize:wN
14241      \int_value:w \__fp_int_eval:w 0 \__fp_parse_trim_zeros:N
14242  }

```

(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`.

```

14243 \cs_new:Npn \__fp_parse_one_other>NN #1 #2
14244  {
14245      \if_int_compare:w
14246          \__fp_int_eval:w
14247              ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
14248              = 3 \exp_stop_f:
14249          \exp_after:wN \__fp_parse_word:Nw
14250          \exp_after:wN #1
14251          \exp_after:wN #2
14252          \exp:w \exp_after:wN \__fp_parse_letters:N
14253          \exp:w
14254      \else:
14255          \exp_after:wN \__fp_parse_prefix:NNN
14256          \exp_after:wN #1
14257          \exp_after:wN #2

```

```

14258 \cs:w
14259   __fp_parse_prefix_ \token_to_str:N #2 :Nw
14260   \exp_after:wN
14261   \cs_end:
14262   \exp:w
14263   \fi:
14264   \__fp_parse_expand:w
14265 }

```

(End definition for `\__fp_parse_one_other>NN.`)

`\__fp_parse_word:Nw` 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 `\fp` word to have an arbitrary mixture of lower and upper case, so we test and use a differently-named control sequence.

```

14266 \cs_new:Npn \__fp_parse_word:Nw #1#2;
14267 {
14268   \cs_if_exist_use:cF { __fp_parse_word:#2:N }
14269   {
14270     \cs_if_exist_use:cF
14271       { __fp_parse_caseless_ \str_fold_case:n {#2} :N }
14272     {
14273       \__kernel_msg_expandable_error:nnn
14274         { kernel } { unknown-fp-word } {#2}
14275       \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
14276       \__fp_parse_infix:NN
14277     }
14278   }
14279   #1
14280 }
14281 \cs_new:Npn \__fp_parse_letters:N #1
14282 {
14283   \exp_end_continue_f:w
14284   \if_int_compare:w
14285     \if_catcode:w \scan_stop: \exp_not:N #1
14286     0
14287   \else:
14288     \__fp_int_eval:w
14289       ( '#1 \if_int_compare:w '#1 > 'Z - 32 \fi: ) / 26
14290   \fi:
14291   = 3 \exp_stop_f:
14292   \exp_after:wN #1
14293   \exp:w \exp_after:wN \__fp_parse_letters:N
14294   \exp:w
14295   \else:
14296     \__fp_parse_return_semicolon:w #1
14297   \fi:
14298   \__fp_parse_expand:w
14299 }

```

(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  $\langle\text{precedence}\rangle$ , #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`.

```
14300 \cs_new:Npn \_fp_parse_prefix:NNN #1#2#3
14301 {
14302     \if_meaning:w \scan_stop: #3
14303         \exp_after:wN \_fp_parse_prefix_unknown:NNN
14304             \exp_after:wN #2
14305         \fi:
14306         #3 #1
14307     }
14308 \cs_new:Npn \_fp_parse_prefix_unknown:NNN #1#2#3
14309 {
14310     \cs_if_exist:cTF { \_fp_parse_infix_ \token_to_str:N #1 :N }
14311     {
14312         \_kernel_msg_expandable_error:nnn
14313             { kernel } { fp-missing-number } {#1}
14314             \exp_after:wN \c_nan_fp \exp:w \exp_end_continue_f:w
14315                 \_fp_parse_infix:NN #3 #1
14316     }
14317     {
14318         \_kernel_msg_expandable_error:nnn
14319             { kernel } { fp-unknown-symbol } {#1}
14320             \_fp_parse_one:Nw #3
14321     }
14322 }
```

(End definition for `\_fp_parse_prefix:NNN` and `\_fp_parse_prefix_unknown:NNN`.)

#### 26.4.1 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`  
`\_fp_parse_trim_end:w`

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.

```
14323 \cs_new:Npn \_fp_parse_trim_zeros:N #1
14324 {
14325     \if:w 0 \exp_not:N #1
14326         \exp_after:wN \_fp_parse_trim_zeros:N
```

```

14327     \exp:w
14328 \else:
14329   \if:w . \exp_not:N #1
14330     \exp_after:wN \__fp_parse_strim_zeros:N
14331     \exp:w
14332   \else:
14333     \__fp_parse_trim_end:w #1
14334   \fi:
14335   \fi:
14336   \__fp_parse_expand:w
14337 }
14338 \cs_new:Npn \__fp_parse_trim_end:w #1 \fi: \fi: \__fp_parse_expand:w
14339 {
14340   \fi:
14341   \fi:
14342   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
14343     \exp_after:wN \__fp_parse_large:N
14344   \else:
14345     \exp_after:wN \__fp_parse_zero:
14346   \fi:
14347   #1
14348 }

```

(End definition for `\__fp_parse_strim_zeros:N` and `\__fp_parse_trim_end:w`.)

`\__fp_parse_strim_zeros:N`  
`\__fp_parse_strim_end:w`

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”.

```

14349 \cs_new:Npn \__fp_parse_strim_zeros:N #1
14350 {
14351   \if:w 0 \exp_not:N #1
14352     - 1
14353     \exp_after:wN \__fp_parse_strim_zeros:N \exp:w
14354   \else:
14355     \__fp_parse_strim_end:w #1
14356   \fi:
14357   \__fp_parse_expand:w
14358 }
14359 \cs_new:Npn \__fp_parse_strim_end:w #1 \fi: \__fp_parse_expand:w
14360 {
14361   \fi:
14362   \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
14363     \exp_after:wN \__fp_parse_small:N
14364   \else:
14365     \exp_after:wN \__fp_parse_zero:
14366   \fi:
14367   #1
14368 }

```

(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.

```

14369 \cs_new:Npn \__fp_parse_zero:
14370 {
14371     \exp_after:wN ; \exp_after:wN 1
14372     \int_value:w \__fp_parse_exponent:N
14373 }

```

(End definition for `\__fp_parse_zero:..`)

#### 26.4.2 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.

```

14374 \cs_new:Npn \__fp_parse_small:N #1
14375 {
14376     \exp_after:wN \__fp_parse_pack_leading:NNNNNww
14377     \int_value:w \__fp_int_eval:w 1 \token_to_str:N #1
14378     \exp_after:wN \__fp_parse_small_leading:wwNN
14379     \int_value:w 1
14380     \exp_after:wN \__fp_parse_digits_vii:N
14381     \exp:w \__fp_parse_expand:w
14382 }

```

(End definition for `\__fp_parse_small:N.`)

`\__fp_parse_small_leading:wwNN`

`\__fp_parse_small_leading:wwNN 1 <digits> ; <zeros> ; <number of zeros>`

We leave `<digits>` `<zeros>` 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.

```

14383 \cs_new:Npn \__fp_parse_small_leading:wwNN 1 #1 ; #2; #3 #4
14384 {
14385     #1 #2
14386     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
14387     \exp_after:wN 0
14388     \int_value:w \__fp_int_eval:w 1
14389     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
14390         \token_to_str:N #4
14391         \exp_after:wN \__fp_parse_small_trailing:wwNN
14392         \int_value:w 1
14393         \exp_after:wN \__fp_parse_digits_vi:N
14394         \exp:w
14395     \else:
14396         0000 0000 \__fp_parse_exponent:Nw #4
14397     \fi:

```

```

14398     \_fp_parse_expand:w
14399 }
```

(End definition for `\_fp_parse_small_leading:wwNN`.)

```

\_\_fp\_parse\_small\_trailing:wwNN 1 <digits> ; <zeros> ; <number of zeros>
<next token>
```

Leave digits 10 to 15 (arguments #1 and #2) in the input stream. If the *<next token>* 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.

```

14400 \cs_new:Npn \_\_fp_parse_small_trailing:wwNN 1 #1 ; #2; #3 #4
14401 {
14402     #1 #2
14403     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
14404         \token_to_str:N #4
14405         \exp_after:wN \_\_fp_parse_small_round:NN
14406         \exp_after:wN #4
14407         \exp:w
14408     \else:
14409         0 \_\_fp_parse_exponent:Nw #4
14410     \fi:
14411     \_\_fp_parse_expand:w
14412 }
```

(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.

```

14413 \cs_new:Npn \_\_fp_parse_pack_trailing:NNNNNNww #1 #2 #3#4#5#6 #7; #8 ;
14414 {
14415     \if_meaning:w 2 #2 + 1 \fi:
14416     ; #8 + #1 ; {#3#4#5#6} {#7};
14417 }
14418 \cs_new:Npn \_\_fp_parse_pack_leading:NNNNNNww #1 #2#3#4#5 #6; #7;
14419 {
14420     + #7
14421     \if_meaning:w 2 #1 \_\_fp_parse_pack_carry:w \fi:
14422     ; 0 {#2#3#4#5} {#6}
14423 }
14424 \cs_new:Npn \_\_fp_parse_pack_carry:w \fi: ; 0 #1
14425     { \fi: + 1 ; 0 {1000} }
```

(End definition for `\_fp_parse_pack_trailing:NNNNNNww`, `\_fp_parse_pack_leading:NNNNNNww`, and `\_fp_parse_pack_carry:w`.)

### 26.4.3 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.

```

14426 \cs_new:Npn \_\_fp_parse_large:N #1
14427 {
14428     \exp_after:wN \_\_fp_parse_large_leading:wwNN
14429     \int_value:w 1 \token_to_str:N #1
14430     \exp_after:wN \_\_fp_parse_digits_vii:N
14431     \exp:w \_\_fp_parse_expand:w
14432 }
```

(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 `<number of zeros>` (number of digits missing). Then prepare to pack the 8 first digits. If the `<next token>` 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 `<zeros>` to complete the 8 first digits, insert 8 more, and look for an exponent.

```

14433 \cs_new:Npn \_\_fp_parse_large_leading:wwNN 1 #1 ; #2; #3 #4
14434 {
14435     + \c_fp_half_prec_int - #3
14436     \exp_after:wN \_\_fp_parse_pack_leading:NNNNNNww
14437     \int_value:w \_\_fp_int_eval:w 1 #1
14438     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
14439         \exp_after:wN \_\_fp_parse_large_trailing:wwNN
14440         \int_value:w 1 \token_to_str:N #4
14441         \exp_after:wN \_\_fp_parse_digits_vii:N
14442         \exp:w
14443     \else:
14444         \if:w . \exp_not:N #4
14445             \exp_after:wN \_\_fp_parse_small_leading:wwNN
14446             \int_value:w 1
14447                 \cs:w
14448                     \_\_fp_parse_digits_
14449                     \_\_fp_int_to_roman:w #3
14450                     :N \exp_after:wN
14451                     \cs_end:
14452                     \exp:w
14453     \else:
14454         #2
14455         \exp_after:wN \_\_fp_parse_pack_trailing:NNNNNNNNww
14456         \exp_after:wN 0
```

```

14457           \int_value:w 1 0000 0000
14458           \__fp_parse_exponent:Nw #4
14459           \fi:
14460           \fi:
14461           \__fp_parse_expand:w
14462       }

```

(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 `<next token>` 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 `<digits>` 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 `<digits>`, 7 minus the `<number of zeros>`, 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.

```

14463 \cs_new:Npn \__fp_parse_large_trailing:wwNN 1 #1 ; #2; #3 #4
14464   {
14465     \if_int_compare:w 9 < 1 \token_to_str:N #4 \exp_stop_f:
14466     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
14467     \exp_after:wN \c__fp_half_prec_int
14468     \int_value:w \__fp_int_eval:w 1 #1 \token_to_str:N #4
14469     \exp_after:wN \__fp_parse_large_round:NN
14470     \exp_after:wN #4
14471     \exp:w
14472   \else:
14473     \exp_after:wN \__fp_parse_pack_trailing:NNNNNNww
14474     \int_value:w \__fp_int_eval:w 7 - #3 \exp_stop_f:
14475     \int_value:w \__fp_int_eval:w 1 #1
14476     \if:w . \exp_not:N #4
14477     \exp_after:wN \__fp_parse_small_trailing:wwNN
14478     \int_value:w 1
14479     \cs:w
14480       \__fp_parse_digits_
14481       \__fp_int_to_roman:w #3
14482       :N \exp_after:wN
14483       \cs_end:
14484       \exp:w
14485     \else:
14486       #2 0 \__fp_parse_exponent:Nw #4
14487     \fi:
14488   \fi:
14489   \__fp_parse_expand:w
14490 }

```

(End definition for `\__fp_parse_large_trailing:wwNN`.)

#### 26.4.4 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.

```

14491 \cs_new:Npn \_\_fp_parse_round_loop:N #1
14492 {
14493     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
14494         + 1
14495         \if:w 0 \token_to_str:N #1
14496             \exp_after:wN \_\_fp_parse_round_loop:N
14497             \exp:w
14498         \else:
14499             \exp_after:wN \_\_fp_parse_round_up:N
14500             \exp:w
14501         \fi:
14502     \else:
14503         \_\_fp_parse_return_semicolon:w 0 #1
14504     \fi:
14505         \_\_fp_parse_expand:w
14506     }
14507 \cs_new:Npn \_\_fp_parse_round_up:N #1
14508 {
14509     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
14510         + 1
14511         \exp_after:wN \_\_fp_parse_round_up:N
14512         \exp:w
14513     \else:
14514         \_\_fp_parse_return_semicolon:w 1 #1
14515     \fi:
14516         \_\_fp_parse_expand:w
14517     }

```

(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.

```

14518 \cs_new:Npn \_\_fp_parse_round_after:wN #1; #2
14519 {
14520     + #2 \exp_after:wN ;
14521     \int_value:w \_\_fp_int_eval:w #1 + \_\_fp_parse_exponent:N
14522 }

```

(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`.

```

14523 \cs_new:Npn \_fp_parse_small_round:NN #1#2
14524 {
14525   \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
14526     +
14527     \exp_after:wN \_fp_round_s:NNNw
14528     \exp_after:wN 0
14529     \exp_after:wN #1
14530     \exp_after:wN #2
14531     \int_value:w \_fp_int_eval:w
14532     \exp_after:wN \_fp_parse_round_after:wN
14533     \int_value:w \_fp_int_eval:w 0 * \_fp_int_eval:w 0
14534     \exp_after:wN \_fp_parse_round_loop:N
14535     \exp:w
14536   \else:
14537     \_fp_parse_exponent:Nw #2
14538   \fi:
14539   \_fp_parse_expand:w
14540 }

```

(End definition for `\_fp_parse_small_round:NN` and `\_fp_parse_round_after:wN`.)

`\_fp_parse_large_round:NN`

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.

```

14541 \cs_new:Npn \_fp_parse_large_round:NN #1#2
14542 {
14543   \if_int_compare:w 9 < 1 \token_to_str:N #2 \exp_stop_f:
14544     +
14545     \exp_after:wN \_fp_round_s:NNNw
14546     \exp_after:wN 0
14547     \exp_after:wN #1
14548     \exp_after:wN #2
14549     \int_value:w \_fp_int_eval:w
14550     \exp_after:wN \_fp_parse_large_round_aux:wNN
14551     \int_value:w \_fp_int_eval:w 1
14552     \exp_after:wN \_fp_parse_round_loop:N
14553   \else: %^A could be dot, or e, or other
14554     \exp_after:wN \_fp_parse_large_round_test:NN
14555     \exp_after:wN #1
14556     \exp_after:wN #2
14557   \fi:
14558 }

```

```

14559 \cs_new:Npn \__fp_parse_large_round_test:NN #1#2
14560 {
14561   \if:w . \exp_not:N #2
14562     \exp_after:wN \__fp_parse_small_round:NN
14563     \exp_after:wN #1
14564     \exp:w
14565   \else:
14566     \__fp_parse_exponent:Nw #2
14567   \fi:
14568   \__fp_parse_expand:w
14569 }
14570 \cs_new:Npn \__fp_parse_large_round_aux:wNN #1 ; #2 #3
14571 {
14572   + #2
14573   \exp_after:wN \__fp_parse_round_after:wN
14574   \int_value:w \__fp_int_eval:w #1
14575   \if:w . \exp_not:N #3
14576     + 0 * \__fp_int_eval:w 0
14577     \exp_after:wN \__fp_parse_round_loop:N
14578     \exp:w \exp_after:wN \__fp_parse_expand:w
14579   \else:
14580     \exp_after:wN ;
14581     \exp_after:wN 0
14582     \exp_after:wN #3
14583   \fi:
14584 }

```

(End definition for `\__fp_parse_large_round:NN`, `\__fp_parse_large_round_test:NN`, and `\__fp_parse_large_round_aux:wNN`.)

#### 26.4.5 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.

```
14585 \cs_new:Npn \_\_fp_parse_exponent:Nw #1 #2 \_\_fp_parse_expand:w
14586 {
14587     \exp_after:wN ;
14588     \int_value:w #2 \_\_fp_parse_exponent:N #1
14589 }
```

(End definition for \\_\\_fp\_parse\_exponent:Nw.)

\\_\\_fp\_parse\_exponent:N  
\\_\\_fp\_parse\_exponent\_aux: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, leave an exponent of 0. If there is an 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.

```
14590 \cs_new:Npn \_\_fp_parse_exponent:N #1
14591 {
14592     \if:w e \exp_not:N #1
14593         \exp_after:wN \_\_fp_parse_exponent_aux:N
14594         \exp:w
14595     \else:
14596         0 \_\_fp_parse_return_semicolon:w #1
14597     \fi:
14598     \_\_fp_parse_expand:w
14599 }
14600 \cs_new:Npn \_\_fp_parse_exponent_aux:N #1
14601 {
14602     \if_int_compare:w \if_catcode:w \scan_stop: \exp_not:N #1
14603         0 \else: '#1 \fi: > '9 \exp_stop_f:
14604         0 \exp_after:wN ; \exp_after:wN e
14605     \else:
14606         \exp_after:wN \_\_fp_parse_exponent_sign:N
14607     \fi:
14608     #1
14609 }
```

(End definition for \\_\\_fp\_parse\_exponent:N and \\_\\_fp\_parse\_exponent\_aux:N.)

\\_\\_fp\_parse\_exponent\_sign:N

Read signs one by one (if there is any).

```
14610 \cs_new:Npn \_\_fp_parse_exponent_sign:N #1
14611 {
14612     \if:w + \if:w - \exp_not:N #1 + \fi: \token_to_str:N #1
14613         \exp_after:wN \_\_fp_parse_exponent_sign:N
14614         \exp:w \exp_after:wN \_\_fp_parse_expand:w
14615     \else:
14616         \exp_after:wN \_\_fp_parse_exponent_body:N
14617         \exp_after:wN #1
14618     \fi:
14619 }
```

(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).

```

14620 \cs_new:Npn \_\_fp_parse_exponent_body:N #1
14621 {
14622     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
14623         \token_to_str:N #1
14624         \exp_after:wN \_\_fp_parse_exponent_digits:N
14625         \exp:w
14626     \else:
14627         \_\_fp_parse_exponent_keep:NTF #1
14628             { \_\_fp_parse_return_semicolon:w #1 }
14629             {
14630                 \exp_after:wN ;
14631                 \exp:w
14632             }
14633     \fi:
14634     \_\_fp_parse_expand:w
14635 }
```

(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.

```

14636 \cs_new:Npn \_\_fp_parse_exponent_digits:N #1
14637 {
14638     \if_int_compare:w 9 < 1 \token_to_str:N #1 \exp_stop_f:
14639         \token_to_str:N #1
14640         \exp_after:wN \_\_fp_parse_exponent_digits:N
14641         \exp:w
14642     \else:
14643         \_\_fp_parse_return_semicolon:w #1
14644     \fi:
14645     \_\_fp_parse_expand:w
14646 }
```

(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.

```

14647 \prg_new_conditional:Npnn \_\_fp_parse_exponent_keep:N #1 { TF }
14648 {
14649     \if_catcode:w \scan_stop: \exp_not:N #1
14650         \if_meaning:w \scan_stop: #1
14651             \if_int_compare:w
```

```

14652   \__fp_str_if_eq:nn { \s__fp } { \exp_not:N #1 }
14653   = 0 \exp_stop_f:
14654   0
14655   \__kernel_msg_expandable_error:nnn
14656   { kernel } { fp-after-e } { floating-point~ }
14657   \prg_return_true:
14658 \else:
14659   0
14660   \__kernel_msg_expandable_error:nnn
14661   { kernel } { bad-variable } {#1}
14662   \prg_return_false:
14663 \fi:
14664 \else:
14665   \if_int_compare:w
14666     \__fp_str_if_eq:nn { \int_value:w #1 } { \tex_the:D #1 }
14667     = 0 \exp_stop_f:
14668     \int_value:w #1
14669 \else:
14670   0
14671   \__kernel_msg_expandable_error:nnn
14672   { kernel } { fp-after-e } { dimension~#1 }
14673 \fi:
14674 \prg_return_false:
14675 \fi:
14676 \else:
14677   0
14678   \__kernel_msg_expandable_error:nnn
14679   { kernel } { fp-missing } { exponent }
14680   \prg_return_true:
14681 \fi:
14682 }

```

(End definition for `\__fp_parse_exponent_keep:NTF.`)

## 26.5 Constants, functions and prefix operators

### 26.5.1 Prefix operators

`\__fp_parse_prefix_+:Nw` A unary + does nothing; we should continue looking for a number.

```
14683 \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.

```

14684 \cs_new:Npn \__fp_parse_apply_function:NNNwN #1#2#3#4#5
14685   {
14686     #3 #2 #4 @
14687     \exp:w \exp_end_continue_f:w #5 #1
14688   }

```

(End definition for `\__fp_parse_apply_function:NNNwN.`)

```
\_fp_parse_apply_unary:NNNwN
\fp_parse_apply_unary_chk:NwNw
\fp_parse_apply_unary_chk:nNNNw
\fp_parse_apply_unary_type:NNN
\fp_parse_apply_unary_error:Nw
```

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.

```
14689 \cs_new:Npn \fp_parse_apply_unary:NNNwN #1#2#3#4@#5
14690 {
14691     \fp_parse_apply_unary_chk:NwNw #4 @ . \q_stop
14692     \fp_parse_apply_unary_type:NNN
14693     #3 #2 #4 @
14694     \exp:w \exp_end_continue_f:w #5 #1
14695 }
14696 \cs_new:Npn \fp_parse_apply_unary_chk:NwNw #1#2 ; #3#4 \q_stop
14697 {
14698     \if_meaning:w @ #3 \else:
14699         \token_if_eq_meaning:NNTF . #3
14700         { \fp_parse_apply_unary_chk:nNNNw { no } }
14701         { \fp_parse_apply_unary_chk:nNNNw { multi } }
14702     \fi:
14703 }
14704 \cs_new:Npn \fp_parse_apply_unary_chk:nNNNw #1#2#3#4#5#6 @
14705 {
14706     #2
14707     \fp_error:nffn { fp-#1-arg } { \fp_func_to_name:N #4 } { } { }
14708     \exp_after:wN #4 \exp_after:wN #5 \c_nan_fp @
14709 }
14710 \cs_new:Npn \fp_parse_apply_unary_type:NNN #1#2#3
14711 {
14712     \fp_change_func_type:NNN #3 #1 \fp_parse_apply_unary_error:Nw
14713     #2 #3
14714 }
14715 \cs_new:Npn \fp_parse_apply_unary_error:Nw #1#2#3 @
14716 { \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`.

```
14717 \cs_set_protected:Npn \fp_tmp:w #1#2#3#4
14718 {
14719     \cs_new:cpn { _fp_parse_prefix_ #1 :Nw } ##1
14720     {
14721         \exp_after:wN \fp_parse_apply_unary:NNNwN
14722         \exp_after:wN ##1
14723         \exp_after:wN #4
14724         \exp_after:wN #3
14725         \exp:w
```

```

14726     \if_int_compare:w #2 < ##1
14727         \__fp_parse_operand:Nw ##1
14728     \else:
14729         \__fp_parse_operand:Nw #2
14730     \fi:
14731     \__fp_parse_expand:w
14732 }
14733 }
14734 \__fp_tmp:w - \c_fp_prec_not_int \__fp_set_sign_o:w 2
14735 \__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.

```

14736 \cs_new:cpn { __fp_parse_prefix_.:Nw } #1
14737 {
14738     \exp_after:wN \__fp_parse_infix_after_operand:NwN
14739     \exp_after:wN #1
14740     \exp:w \exp_end_continue_f:w
14741     \exp_after:wN \__fp_SANITIZE:wN
14742     \int_value:w \__fp_int_eval:w 0 \__fp_parse_strim_zeros:N
14743 }

```

(End definition for `\__fp_parse_prefix_.:Nw`.)

`\__fp_parse_prefix_(:Nw`  
`\__fp_parse_lparen_after:NwN`

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 `lparen_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.

```

14744 \cs_new:cpn { __fp_parse_prefix_(:Nw } #1
14745 {
14746     \exp_after:wN \__fp_parse_lparen_after:NwN
14747     \exp_after:wN #1
14748     \exp:w
14749     \if_int_compare:w #1 = \c_fp_prec_func_int
14750         \__fp_parse_operand:Nw \c_fp_prec_comma_int
14751     \else:
14752         \__fp_parse_operand:Nw \c_fp_prec_tuple_int
14753     \fi:
14754     \__fp_parse_expand:w
14755 }
14756 \cs_new:Npx \__fp_parse_lparen_after:NwN #1#2 @ #3
14757 {
14758     \exp_not:N \token_if_eq_meaning:NNTF #3
14759     \exp_not:c { __fp_parse_infix_):N }
14760 }

```

```

14761     \exp_not:N \__fp_parse_after_array_f:w #2 \s__fp_stop
14762     \exp_not:N \exp_after:wN
14763     \exp_not:N \__fp_parse_infix:NN
14764     \exp_not:N \exp_after:wN #1
14765     \exp_not:N \exp:w
14766     \exp_not:N \__fp_parse_expand:w
14767 }
14768 {
14769     \exp_not:N \__kernel_msg_expandable_error:nnn
14770         { kernel } { fp-missing } { ) }
14771     \exp_not:N \tl_if_empty:nT [#2] \exp_not:N \c__fp_empty_tuple_fp
14772         #2 @
14773     \exp_not:N \use_none:n #3
14774 }
14775 }
```

(End definition for `\__fp_parse_prefix_(:Nw` and `\__fp_parse_lparen_after:NwN`.)

`\__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()`.

```

14776 \cs_new:cpxn { __fp_parse_prefix_):Nw } #1
14777 {
14778     \if_int_compare:w #1 = \c__fp_prec_comma_int
14779     \else:
14780         \if_int_compare:w #1 = \c__fp_prec_tuple_int
14781             \exp_after:wN \c__fp_empty_tuple_fp \exp:w
14782         \else:
14783             \__kernel_msg_expandable_error:nnn
14784                 { kernel } { fp-missing-number } { ) }
14785             \exp_after:wN \c_nan_fp \exp:w
14786         \fi:
14787         \exp_end_continue_f:w
14788     \fi:
14789     \__fp_parse_infix:NN #1 )
14790 }
```

(End definition for `\__fp_parse_prefix_):Nw`.)

### 26.5.2 Constants

`\__fp_parse_word_inf:N`  
`\__fp_parse_word_nan:N`  
`\__fp_parse_word_pi:N`  
`\__fp_parse_word_deg:N`  
`\__fp_parse_word_true:N`  
`\__fp_parse_word_false:N`

```

14791 \cs_set_protected:Npn \__fp_tmp:w #1 #
14792 {
14793     \cs_new:cpxn { __fp_parse_word_#1:N }
14794         { \exp_after:wN #2 \exp:w \exp_end_continue_f:w \__fp_parse_infix:NN }
14795 }
14796 \__fp_tmp:w { inf } \c_inf_fp
14797 \__fp_tmp:w { nan } \c_nan_fp
14798 \__fp_tmp:w { pi } \c_pi_fp
14799 \__fp_tmp:w { deg } \c_one_degree_fp
14800 \__fp_tmp:w { true } \c_one_fp
14801 \__fp_tmp:w { false } \c_zero_fp
```

(End definition for `\_fp_parse_word_inf:N` and others.)

`\_fp_parse_caseless_inf:N` Copies of `\_fp_parse_word_...:N` commands, to allow arbitrary case as mandated by the standard.

`\_fp_parse_caseless_nan:N`

```
14802 \cs_new_eq:NN \_fp_parse_caseless_inf:N \_fp_parse_word_inf:N
14803 \cs_new_eq:NN \_fp_parse_caseless_infinity:N \_fp_parse_word_inf:N
14804 \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.

```
14805 \cs_set_protected:Npn \_fp_tmp:w #1 #2
14806 {
14807     \cs_new:cpn { \_fp_parse_word_#1:N }
14808     {
14809         \_fp_exp_after:f:nw { \_fp_parse_infix:NN }
14810         \s_fp \_fp_chk:w 10 #2 ;
14811     }
14812 }
14813 \_fp_tmp:w {pt} { {1} {1000} {0000} {0000} {0000} }
14814 \_fp_tmp:w {in} { {2} {7227} {0000} {0000} {0000} }
14815 \_fp_tmp:w {pc} { {2} {1200} {0000} {0000} {0000} }
14816 \_fp_tmp:w {cm} { {2} {2845} {2755} {9055} {1181} }
14817 \_fp_tmp:w {mm} { {1} {2845} {2755} {9055} {1181} }
14818 \_fp_tmp:w {dd} { {1} {1070} {0085} {6496} {0630} }
14819 \_fp_tmp:w {cc} { {2} {1284} {0102} {7795} {2756} }
14820 \_fp_tmp:w {nd} { {1} {1066} {9783} {4645} {6693} }
14821 \_fp_tmp:w {nc} { {2} {1280} {3740} {1574} {8031} }
14822 \_fp_tmp:w {bp} { {1} {1003} {7500} {0000} {0000} }
14823 \_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`.

```
14824 \tl_map_inline:nn { {em} {ex} }
14825 {
14826     \cs_new:cpn { \_fp_parse_word_#1:N }
14827     {
14828         \exp_after:wN \_fp_from_dim_test:ww
14829         \exp_after:wN 0 \exp_after:wN ,
14830         \int_value:w \dim_to_decimal_in_sp:n { 1 #1 } \exp_after:wN ;
14831         \exp:w \exp_end_continue_f:w \_fp_parse_infix:NN
14832     }
14833 }
```

(End definition for `\_fp_parse_word_em:N` and `\_fp_parse_word_ex:N`.)

### 26.5.3 Functions

```

\__fp_parse_unary_function:NNN
\__fp_parse_function:NNN
14834 \cs_new:Npn \__fp_parse_unary_function:NNN #1#2#3
14835 {
14836     \exp_after:wN \__fp_parse_apply_unary:NNNwN
14837     \exp_after:wN #3
14838     \exp_after:wN #2
14839     \exp_after:wN #1
14840     \exp:w
14841     \__fp_parse_operand:Nw \c_fp_prec_func_int \__fp_parse_expand:w
14842 }
14843 \cs_new:Npn \__fp_parse_function:NNN #1#2#3
14844 {
14845     \exp_after:wN \__fp_parse_apply_function:NNNwN
14846     \exp_after:wN #3
14847     \exp_after:wN #2
14848     \exp_after:wN #1
14849     \exp:w
14850     \__fp_parse_operand:Nw \c_fp_prec_func_int \__fp_parse_expand:w
14851 }

```

(End definition for `\__fp_parse_unary_function:NNN` and `\__fp_parse_function:NNN`.)

## 26.6 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_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:`.

```

14852 \cs_new:Npn \__fp_parse:n #1
14853 {
14854     \exp:w
14855     \exp_after:wN \__fp_parse_after:ww
14856     \exp:w
14857     \__fp_parse_operand:Nw \c_fp_prec_end_int
14858     \__fp_parse_expand:w #1
14859     \s_fp_mark \__fp_parse_infix_end:N
14860     \s_fp_stop
14861     \exp_end:
14862 }
14863 \cs_new:Npn \__fp_parse_after:ww
14864     #1@ \__fp_parse_infix_end:N \s_fp_stop #2 { #2 #1 }
14865 \cs_new:Npn \__fp_parse_o:n #1
14866 {
14867     \exp:w
14868     \exp_after:wN \__fp_parse_after:ww
14869     \exp:w
14870     \__fp_parse_operand:Nw \c_fp_prec_end_int
14871     \__fp_parse_expand:w #1
14872     \s_fp_mark \__fp_parse_infix_end:N
14873     \s_fp_stop

```

```

14874     {
14875         \exp_end_continue_f:w
14876         \_\_fp\_exp\_after\_any\_f:nw { \exp\_after:wN \exp\_stop\_f: }
14877     }
14878 }
```

(End definition for `\_\_fp\_parse:n`, `\_\_fp\_parse:o:n`, and `\_\_fp\_parse\_after:ww`.)

`\_\_fp_parse_operand:Nw`  
`\_\_fp_parse_continue:NwN`

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`.

```

14879 \cs_new:Npn \_\_fp_parse_operand:Nw #1
14880 {
14881     \exp_end_continue_f:w
14882     \exp_after:wN \_\_fp_parse_continue:NwN
14883     \exp_after:wN #1
14884     \exp:w \exp_end_continue_f:w
14885     \exp_after:wN \_\_fp_parse_one:Nw
14886     \exp_after:wN #1
14887     \exp:w
14888 }
14889 \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 `<precedence> <operand1> @ <operation> <operand2> @ <infix command>`. Builds the appropriate call to the `<operation> #3`, dispatching on both types. If the resulting control sequence does not exist, the operation is not allowed.

This is redefined in l3fp-extras.

```

14890 \cs_new:Npn \_\_fp_parse_apply_binary:NwNwN #1 #2#3@ #4 #5#6@ #7
14891 {
14892     \exp_after:wN \_\_fp_parse_continue:NwN
14893     \exp_after:wN #1
14894     \exp:w \exp_end_continue_f:w
14895     \exp_after:wN \_\_fp_parse_apply_binary_chk:NN
14896     \cs:w
14897         __fp
14898         \_\_fp_type_from_scan:N #2
14899         _#4
14900         \_\_fp_type_from_scan:N #5
14901         _o:ww
14902         \cs_end:
14903         #4
14904         #2#3 #5#6
14905         \exp:w \exp_end_continue_f:w #7 #1
14906     }
14907 \cs_new:Npn \_\_fp_parse_apply_binary_chk:NN #1#2
14908 {
14909     \if_meaning:w \scan_stop: #1
14910     \_\_fp_parse_apply_binary_error:NNN #2
14911     \fi:
14912     #1
14913 }
14914 \cs_new:Npn \_\_fp_parse_apply_binary_error:NNN #1#2#3
14915 {
```

```

14916      #2
14917      \_\_fp\_invalid\_operation\_o:Nww #1
14918  }

(End definition for \_\_fp\_parse\_apply\_binary:NwNwN, \_\_fp\_parse\_apply\_binary\_chk:NN, and \_\_fp\_parse\_apply\_binary\_error:NNN.)
```

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.

```

14919 \cs_new:Npn \_\_fp\_binary\_type\_o:Nww #1 #2#3 ; #4
14920  {
14921      \exp_after:wN \_\_fp\_parse\_apply\_binary\_chk:NN
14922      \cs:w
14923          \_\_fp
14924          \_\_fp\_type\_from\_scan:N #2
14925          _ #1
14926          \_\_fp\_type\_from\_scan:N #4
14927          _o:ww
14928          \cs_end:
14929          #1
14930          #2 #3 ; #4
14931  }
14932 \cs_new:Npn \_\_fp\_binary\_rev\_type\_o:Nww #1 #2#3 ; #4#5 ;
14933  {
14934      \exp_after:wN \_\_fp\_parse\_apply\_binary\_chk:NN
14935      \cs:w
14936          \_\_fp
14937          \_\_fp\_type\_from\_scan:N #4
14938          _ #1
14939          \_\_fp\_type\_from\_scan:N #2
14940          _o:ww
14941          \cs_end:
14942          #1
14943          #4 #5 ; #2 #3 ;
14944 }
```

(End definition for \\_\\_fp\\_binary\\_type\\_o:Nww and \\_\\_fp\\_binary\\_rev\\_type\\_o:Nww.)

## 26.7 Infix operators

```

\_\_fp\_parse\_infix\_after\_operand:NwN

14945 \cs_new:Npn \_\_fp\_parse\_infix\_after\_operand:NwN #1 #2;
14946  {
14947      \_\_fp\_exp\_after_f:nw { \_\_fp\_parse\_infix:NN #1 }
14948      #2;
14949  }
14950 \cs_new:Npn \_\_fp\_parse\_infix:NN #1 #2
14951  {
14952      \if_catcode:w \scan_stop: \exp_not:N #2
14953      \if_int_compare:w
14954          \_\_fp\_str_if_eq:nn { \s_\_fp_mark } { \exp_not:N #2 }
14955          = 0 \exp_stop_f:
14956      \exp_after:wN \exp_after:wN
14957      \exp_after:wN \_\_fp\_parse\_infix\_mark:NNN
```

```

14958 \else:
14959   \exp_after:wN \exp_after:wN
14960   \exp_after:wN \__fp_parse_infix_mul:N
14961 \fi:
14962 \else:
14963   \if_int_compare:w
14964     \__fp_int_eval:w
14965     ( '#2 \if_int_compare:w '#2 > 'Z - 32 \fi: ) / 26
14966     = 3 \exp_stop_f:
14967   \exp_after:wN \exp_after:wN
14968   \exp_after:wN \__fp_parse_infix_mul:N
14969 \else:
14970   \exp_after:wN \__fp_parse_infix_check:NNN
14971   \cs:w
14972     __fp_parse_infix_ \token_to_str:N #2 :N
14973     \exp_after:wN \exp_after:wN \exp_after:wN
14974     \cs_end:
14975   \fi:
14976 \fi:
14977 #1
14978 #2
14979 }
14980 \cs_new:Npx \__fp_parse_infix_check:NNN #1#2#3
14981 {
14982   \exp_not:N \if_meaning:w \scan_stop: #1
14983   \exp_not:N \__kernel_msg_expandable_error:nnn
14984     { kernel } { fp-missing } { * }
14985   \exp_not:N \exp_after:wN
14986   \exp_not:c { __fp_parse_infix_*:N }
14987   \exp_not:N \exp_after:wN #2
14988   \exp_not:N \exp_after:wN #3
14989 \exp_not:N \else:
14990   \exp_not:N \exp_after:wN #1
14991   \exp_not:N \exp_after:wN #2
14992   \exp_not:N \exp:w
14993   \exp_not:N \exp_after:wN
14994   \exp_not:N \__fp_parse_expand:w
14995 \exp_not:N \fi:
14996 }

```

(End definition for `\__fp_parse_infix_after_operand:NwN.`)

### 26.7.1 Closing parentheses and commas

As an infix operator, `\s__fp_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.

```
14997 \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.

```
14998 \cs_new:Npn \__fp_parse_infix_end:N #1
14999   { @ \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`.

```
15000 \cs_set_protected:Npn \_\_fp\_tmp:w #1
15001 {
15002     \cs_new:Npn #1 ##1
15003     {
15004         \if_int_compare:w ##1 > \c\_\_fp\_prec\_end\_int
15005             \exp_after:wN @
15006             \exp_after:wN \use_none:n
15007             \exp_after:wN #1
15008         \else:
15009             \__kernel_msg_expandable_error:nnn { kernel } { fp-extra } { }
15010             \exp_after:wN \_\_fp_parse_infix:NN
15011             \exp_after:wN ##1
15012             \exp:w \exp_after:wN \_\_fp_parse_expand:w
15013         \fi:
15014     }
15015 }
15016 \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.

```
15017 \cs_set_protected:Npn \_\_fp\_tmp:w #1
15018 {
15019     \cs_new:Npn #1 ##1
15020     {
15021         \if_int_compare:w ##1 > \c\_\_fp\_prec\_comma\_int
15022             \exp_after:wN @
15023             \exp_after:wN \use_none:n
15024             \exp_after:wN #1
15025         \else:
15026             \if_int_compare:w ##1 < \c\_\_fp\_prec\_comma\_int
15027                 \exp_after:wN @
15028                 \exp_after:wN \_\_fp_parse_apply_comma:NwNwN
15029                 \exp_after:wN ,
15030                 \exp:w
15031             \else:
15032                 \exp_after:wN \_\_fp_parse_infix_comma:w
15033                 \exp:w
15034             \fi:
15035             \_\_fp_parse_operand:Nw \c\_\_fp\_prec\_comma\_int
15036             \exp_after:wN \_\_fp_parse_expand:w
15037         \fi:
```

```

15038     }
15039   }
15040 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_,:N }
15041 \cs_new:Npn \_fp_parse_infix_comma:w #1 @
15042   { #1 @ \use_none:n }
15043 \cs_new:Npn \_fp_parse_apply_comma:NwNwN #1 #2@ #3 #4@ #5
15044   {
15045     \exp_after:wN \_fp_parse_continue:NwN
15046     \exp_after:wN #1
15047     \exp:w \exp_end_continue_f:w
15048     \_fp_exp_after_tuple_f:nw { }
15049     \s_fp_tuple \_fp_tuple_chk:w { #2 #4 } ;
15050     #5 #1
15051   }

```

(End definition for `\_fp_parse_infix_,:N`, `\_fp_parse_infix_comma:w`, and `\_fp_parse_apply_comma:NwNwN`.)

### 26.7.2 Usual infix operators

`\_fp_parse_infix_+:N` `\_fp_parse_infix_-: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.

```

15052 \cs_set_protected:Npn \_fp_tmp:w #1#2#3#4
15053   {
15054     \cs_new:Npn #1 ##1
15055       {
15056         \if_int_compare:w ##1 < #3
15057           \exp_after:wN @
15058           \exp_after:wN \_fp_parse_apply_binary:NwNwN
15059           \exp_after:wN #2
15060           \exp:w
15061           \_fp_parse_operand:Nw #4
15062           \exp_after:wN \_fp_parse_expand:w
15063         \else:
15064           \exp_after:wN @
15065           \exp_after:wN \use_none:n
15066           \exp_after:wN #1
15067         \fi:
15068       }
15069     }
15070   \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_^-N } ^
15071     \c_fp_prec_hatii_int \c_fp_prec_hat_int
15072   \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_/:N } /
15073     \c_fp_prec_times_int \c_fp_prec_times_int
15074   \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_mul:N } *
15075     \c_fp_prec_times_int \c_fp_prec_times_int
15076   \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_-:N } -
15077     \c_fp_prec_plus_int \c_fp_prec_plus_int
15078   \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_+:N } +
15079     \c_fp_prec_plus_int \c_fp_prec_plus_int
15080   \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_and:N } &
15081     \c_fp_prec_and_int \c_fp_prec_and_int

```

```

15082 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_or:N } |
15083   \c__fp_prec_or_int   \c__fp_prec_or_int

```

(End definition for `\_fp_parse_infix_+:N` and others.)

### 26.7.3 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`.

```

15084 \cs_new:cpn { __fp_parse_infix_(:N) #1
15085   { \_fp_parse_infix_mul:N #1 ( }

```

(End definition for `\_fp_parse_infix_(:N)`.)

### 26.7.4 Multi-character cases

```

\_fp_parse_infix_*:N
15086 \cs_set_protected:Npn \_fp_tmp:w #1
15087   {
15088     \cs_new:cpn { __fp_parse_infix_*:N } ##1##2
15089     {
15090       \if:w * \exp_not:N ##2
15091         \exp_after:wN #1
15092         \exp_after:wN ##1
15093       \else:
15094         \exp_after:wN \_fp_parse_infix_mul:N
15095         \exp_after:wN ##1
15096         \exp_after:wN ##2
15097       \fi:
15098     }
15099   }
15100 \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
15101 \cs_set_protected:Npn \_fp_tmp:w #1#2#3
15102   {
15103     \cs_new:Npn #1 ##1##2
15104     {
15105       \if:w #2 \exp_not:N ##2
15106         \exp_after:wN #1
15107         \exp_after:wN ##1
15108         \exp:w \exp_after:wN \_fp_parse_expand:w
15109       \else:
15110         \exp_after:wN #3
15111         \exp_after:wN ##1
15112         \exp_after:wN ##2
15113       \fi:
15114     }
15115   }
15116 \exp_args:Nc \_fp_tmp:w { __fp_parse_infix_|:N } | \_fp_parse_infix_or:N
15117 \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`.)

### 26.7.5 Ternary operator

```
\__fp_parse_infix_?:N
\__fp_parse_infix_::N
15118 \cs_set_protected:Npn \__fp_tmp:w #1#2#3#4
15119 {
15120     \cs_new:Npn #1 ##1
15121 {
15122     \if_int_compare:w ##1 < \c_fp_prec_quest_int
15123         #4
15124         \exp_after:wN @
15125         \exp_after:wN #2
15126         \exp:w
15127         \__fp_parse_operand:Nw #3
15128         \exp_after:wN \__fp_parse_expand:w
15129     \else:
15130         \exp_after:wN @
15131         \exp_after:wN \use_none:n
15132         \exp_after:wN #1
15133     \fi:
15134 }
15135 }
15136 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_?:N }
15137     \__fp_ternary:NwwN \c_fp_prec_quest_int { }
15138 \exp_args:Nc \__fp_tmp:w { __fp_parse_infix_::N }
15139     \__fp_ternary_auxii:NwwN \c_fp_prec_colon_int
15140 {
15141     \__kernel_msg_expandable_error:nnnn
15142     { kernel } { fp-missing } { ? } { ~for~?: }
15143 }
```

(End definition for `\__fp_parse_infix_?:N` and `\__fp_parse_infix_::N`.)

### 26.7.6 Comparisons

```
\__fp_parse_infix_<:N
\__fp_parse_infix_=:N
\__fp_parse_infix_>:N
\__fp_parse_infix_!:N
\__fp_parse_excl_error:
\__fp_parse_compare:NNNNNNN
\__fp_parse_compare_auxi:NNNNNNN
\__fp_parse_compare_auxii:NNNNN
\__fp_parse_compare_end:NNNNw
\__fp_compare:wNNNNw
15144 \cs_new:cpn { __fp_parse_infix_<:N } #1
15145     { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 < }
15146 \cs_new:cpn { __fp_parse_infix_=:N } #1
15147     { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 = }
15148 \cs_new:cpn { __fp_parse_infix_>:N } #1
15149     { \__fp_parse_compare:NNNNNNN #1 1 0 0 0 0 > }
15150 \cs_new:cpn { __fp_parse_infix_!:N } #1
15151 {
15152     \exp_after:wN \__fp_parse_compare:NNNNNNN
15153     \exp_after:wN #1
15154     \exp_after:wN 0
15155     \exp_after:wN 1
15156     \exp_after:wN 1
15157     \exp_after:wN 1
15158     \exp_after:wN 1
15159 }
15160 \cs_new:Npn \__fp_parse_excl_error:
15161 {
15162     \__kernel_msg_expandable_error:nnnn
```

```

15163      { kernel } { fp-missing } { = } { ~after~!. }
15164    }
15165 \cs_new:Npn \__fp_parse_compare:NNNNNNN #1
15166  {
15167    \if_int_compare:w #1 < \c__fp_prec_comp_int
15168      \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
15169      \exp_after:wN \__fp_parse_excl_error:
15170    \else:
15171      \exp_after:wN @
15172      \exp_after:wN \use_none:n
15173      \exp_after:wN \__fp_parse_compare:NNNNNNN
15174    \fi:
15175  }
15176 \cs_new:Npn \__fp_parse_compare_auxi:NNNNNNN #1#2#3#4#5#6#7
15177  {
15178    \if_case:w
15179      \__fp_int_eval:w \exp_after:wN ` \token_to_str:N #7 - '<
15180      \__fp_int_eval_end:
15181      \__fp_parse_compare_auxii:NNNNN #2#2#4#5#6
15182      \or: \__fp_parse_compare_auxii:NNNNN #2#3#2#5#6
15183      \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#2#6
15184      \or: \__fp_parse_compare_auxii:NNNNN #2#3#4#5#2
15185      \else: #1 \__fp_parse_compare_end:NNNNw #3#4#5#6#7
15186    \fi:
15187  }
15188 \cs_new:Npn \__fp_parse_compare_auxii:NNNNN #1#2#3#4#5
15189  {
15190    \exp_after:wN \__fp_parse_compare_auxi:NNNNNNN
15191    \exp_after:wN \prg_do_nothing:
15192    \exp_after:wN #1
15193    \exp_after:wN #2
15194    \exp_after:wN #3
15195    \exp_after:wN #4
15196    \exp_after:wN #5
15197    \exp:w \exp_after:wN \__fp_parse_expand:w
15198  }
15199 \cs_new:Npn \__fp_parse_compare_end:NNNNw #1#2#3#4#5 \fi:
15200  {
15201    \fi:
15202    \exp_after:wN @
15203    \exp_after:wN \__fp_parse_apply_compare:NwNNNNNwN
15204    \exp_after:wN \c_one_fp
15205    \exp_after:wN #1
15206    \exp_after:wN #2
15207    \exp_after:wN #3
15208    \exp_after:wN #4
15209    \exp:w
15210    \__fp_parse_operand:Nw \c__fp_prec_comp_int \__fp_parse_expand:w #5
15211  }
15212 \cs_new:Npn \__fp_parse_apply_compare:NwNNNNNwN
15213  #1 #2@ #3 #4#5#6#7 #8@ #9
15214  {
15215    \if_int_odd:w
15216      \if_meaning:w \c_zero_fp #3

```

```

15217          0
15218      \else:
15219          \if_case:w \__fp_compare_back_any:ww #8 #2 \exp_stop_f:
15220              #5 \or: #6 \or: #7 \else: #4
15221          \fi:
15222      \fi:
15223      \exp_stop_f:
15224          \exp_after:wN \__fp_parse_apply_compare_aux>NNwN
15225          \exp_after:wN \c_one_fp
15226      \else:
15227          \exp_after:wN \__fp_parse_apply_compare_aux>NNwN
15228          \exp_after:wN \c_zero_fp
15229      \fi:
15230      #1 #8 #9
15231  }
15232 \cs_new:Npn \__fp_parse_apply_compare_aux>NNwN #1 #2 #3; #4
15233  {
15234      \if_meaning:w \__fp_parse_compare:NNNNNNN #4
15235          \exp_after:wN \__fp_parse_continue_compare:NNwNN
15236          \exp_after:wN #1
15237          \exp_after:wN #2
15238          \exp:w \exp_end_continue_f:w
15239          \__fp_exp_after_o:w #3;
15240          \exp:w \exp_end_continue_f:w
15241      \else:
15242          \exp_after:wN \__fp_parse_continue:NwN
15243          \exp_after:wN #2
15244          \exp:w \exp_end_continue_f:w
15245          \exp_after:wN #1
15246          \exp:w \exp_end_continue_f:w
15247      \fi:
15248      #4 #2
15249  }
15250 \cs_new:Npn \__fp_parse_continue_compare:NNwNN #1#2 #3@ #4#5
15251  { #4 #2 #3@ #1 }

```

(End definition for `\__fp_parse_infix_<:N` and others.)

## 26.8 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).

```

15252 \cs_new:Npn \__fp_parse_function_all_fp_o:fnw #1#2#3 @
15253  {
15254      \__fp_array_if_all_fp:nTF {#3}
15255      { #2 #3 @ }
15256      {
15257          \__fp_error:nffn { fp-bad-args }
15258          {#1}
15259          { \fp_to_tl:n { \s__fp_tuple \__fp_tuple_chk:w {#3} ; } }
15260          { }
15261          \exp_after:wN \c_nan_fp
15262      }
15263  }

```

(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.

```

15264 \cs_new:Npn \_fp_parse_function_one_two:nnw #1#2#3
15265 {
15266     \_fp_if_type_fp:NTwFw
15267         #3 { } \s__fp \_fp_parse_function_one_two_error_o:w \q_stop
15268         \_fp_parse_function_one_two_aux:nnw {#1} {#2} #3
15269     }
15270 \cs_new:Npn \_fp_parse_function_one_two_error_o:w #1#2#3#4 @
15271 {
15272     \_fp_error:nffn { fp-bad-args }
15273     {#2}
15274     { \fp_to_tl:n { \s__fp_tuple \_fp_tuple_chk:w {#4} ; } }
15275     { }
15276     \exp_after:wN \c_nan_fp
15277 }
15278 \cs_new:Npn \_fp_parse_function_one_two_aux:nnw #1#2 #3; #4
15279 {
15280     \_fp_if_type_fp:NTwFw
15281     #4 { }
15282     \s__fp
15283     {
15284         \if_meaning:w @ #4
15285             \exp_after:wN \use_iv:nnnn
15286             \fi:
15287             \_fp_parse_function_one_two_error_o:w
15288         }
15289         \q_stop
15290         \_fp_parse_function_one_two_auxii:nnw {#1} {#2} #3; #4
15291     }
15292 \cs_new:Npn \_fp_parse_function_one_two_auxii:nnw #1#2#3; #4; #5
15293 {
15294     \if_meaning:w @ #5 \else:
15295         \exp_after:wN \_fp_parse_function_one_two_error_o:w
15296         \fi:
15297         \use_i:nn {#1} { \use_none:n #2 } #3; #4; #5
15298 }
```

(End definition for `\_fp_parse_function_one_two:nnw` and others.)

`\_fp_tuple_map_o:nw`  
`\_fp_tuple_map_loop_o:nw`

Apply #1 to all items in the following tuple and expand once afterwards. The code #1 should itself expand once after its result.

```

15299 \cs_new:Npn \_fp_tuple_map_o:nw #1 \s__fp_tuple \_fp_tuple_chk:w #2 ;
15300 {
```

```

15301      \exp_after:wN \s_fp_tuple
15302      \exp_after:wN \_fp_tuple_chk:w
15303      \exp_after:wN {
15304          \exp:w \exp_end_continue_f:w
15305          \_fp_tuple_map_loop_o:nw {#1} #2
15306          { \s_fp \prg_break: } ;
15307          \prg_break_point:
15308          \exp_after:wN } \exp_after:wN ;
15309      }
15310 \cs_new:Npn \_fp_tuple_map_loop_o:nw #1#2#3 ;
15311  {
15312      \use_none:n #2
15313      #1 #2 #3 ;
15314      \exp:w \exp_end_continue_f:w
15315      \_fp_tuple_map_loop_o:nw {#1}
15316  }

```

(End definition for `\_fp_tuple_map_o:nw` and `\_fp_tuple_map_loop_o:nw`.)

`\_fp_tuple_mapthread_o:nww` Apply #1 to pairs of items in the two following tuples and expand once afterwards.

```

\_fp_tuple_mapthread_o:nww
15317 \cs_new:Npn \_fp_tuple_mapthread_o:nww #1
15318     \s_fp_tuple \_fp_tuple_chk:w #2 ;
15319     \s_fp_tuple \_fp_tuple_chk:w #3 ;
15320  {
15321      \exp_after:wN \s_fp_tuple
15322      \exp_after:wN \_fp_tuple_chk:w
15323      \exp_after:wN {
15324          \exp:w \exp_end_continue_f:w
15325          \_fp_tuple_mapthread_loop_o:nw {#1}
15326          #2 { \s_fp \prg_break: } ; @
15327          #3 { \s_fp \prg_break: } ;
15328          \prg_break_point:
15329          \exp_after:wN } \exp_after:wN ;
15330  }
15331 \cs_new:Npn \_fp_tuple_mapthread_loop_o:nw #1#2#3 ; #4 @ #5#6 ;
15332  {
15333      \use_none:n #2
15334      \use_none:n #5
15335      #1 #2 #3 ; #5 #6 ;
15336      \exp:w \exp_end_continue_f:w
15337      \_fp_tuple_mapthread_loop_o:nw {#1} #4 @
15338  }

```

(End definition for `\_fp_tuple_mapthread_o:nww` and `\_fp_tuple_mapthread_loop_o:nw`.)

## 26.9 Messages

```

15339 \_kernel_msg_new:nnn { kernel } { fp-deprecated }
15340     { '#1'~deprecated;~use~'#2' }
15341 \_kernel_msg_new:nnn { kernel } { unknown-fp-word }
15342     { Unknown~fp~word~#1. }
15343 \_kernel_msg_new:nnn { kernel } { fp-missing }
15344     { Missing~#1~inserted #2. }
15345 \_kernel_msg_new:nnn { kernel } { fp-extra }
15346     { Extra~#1~ignored. }

```

```

15347 \__kernel_msg_new:nnn { kernel } { fp-early-end }
15348   { Premature~end~in~fp~expression. }
15349 \__kernel_msg_new:nnn { kernel } { fp-after-e }
15350   { Cannot~use~#1 after~'e'. }
15351 \__kernel_msg_new:nnn { kernel } { fp-missing-number }
15352   { Missing~number~before~'#1'. }
15353 \__kernel_msg_new:nnn { kernel } { fp-unknown-symbol }
15354   { Unknown~symbol~#1~ignored. }
15355 \__kernel_msg_new:nnn { kernel } { fp-extra-comma }
15356   { Unexpected~comma~turned~to~nan~result. }
15357 \__kernel_msg_new:nnn { kernel } { fp-no-arg }
15358   { #1~got~no~argument;~used~nan. }
15359 \__kernel_msg_new:nnn { kernel } { fp-multi-arg }
15360   { #1~got~more~than~one~argument;~used~nan. }
15361 \__kernel_msg_new:nnn { kernel } { fp-num-args }
15362   { #1~expects~between~#2~and~#3~arguments. }
15363 \__kernel_msg_new:nnn { kernel } { fp-bad-args }
15364   { Arguments~in~#1#2~are~invalid. }
15365 \__kernel_msg_new:nnn { kernel } { fp-infty-pi }
15366   { Math~command~#1~is~not~an~fp }
15367 (*package)
15368 \cs_if_exist:cT { @unexpandable@protect }
15369 {
15370   \__kernel_msg_new:nnn { kernel } { fp-robust-cmd }
15371   { Robust~command~#1~invalid~in~fp~expression! }
15372 }
15373 
```

15374 </initex | package>

## 27 l3fp-assign implementation

15375 <\*initex | package>

15376 <@@=fp>

### 27.1 Assigning values

**\fp\_new:N** Floating point variables are initialized to be +0.

```

15377 \cs_new_protected:Npn \fp_new:N #1
15378   { \cs_new_eq:NN #1 \c_zero_fp }
15379 \cs_generate_variant:Nn \fp_new:N {c}
```

(End definition for \fp\_new:N. This function is documented on page 185.)

**\fp\_set:Nn** Simply use \\_\_fp\_parse:n within various f-expanding assignments.

```

\fp_set:cn
15380 \cs_new_protected:Npn \fp_set:Nn #1#2
15381   { \tl_set:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_gset:cn
15382 \cs_new_protected:Npn \fp_gset:Nn #1#2
15383   { \tl_gset:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
\fp_const:Nn
15384 \cs_new_protected:Npn \fp_const:Nn #1#2
15385   { \tl_const:Nx #1 { \exp_not:f { \__fp_parse:n {#2} } } }
15386 \cs_generate_variant:Nn \fp_set:Nn {c}
15387 \cs_generate_variant:Nn \fp_gset:Nn {c}
15388 \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 186.)

`\fp_set_eq:NN` Copying a floating point is the same as copying the underlying token list.

```

\fp_set_eq:cN 15389 \cs_new_eq:NN \fp_set_eq:NN \tl_set_eq:NN
\fp_set_eq:Nc 15390 \cs_new_eq:NN \fp_gset_eq:NN \tl_gset_eq:NN
\fp_set_eq:cc 15391 \cs_generate_variant:Nn \fp_set_eq:NN { c , Nc , cc }
\fp_gset_eq>NN 15392 \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 186.)

`\fp_gset_eq:cN` Setting a floating point to zero: copy `\c_zero_fp`.

```

\fp_zero:c 15393 \cs_new_protected:Npn \fp_zero:N #1 { \fp_set_eq:NN #1 \c_zero_fp }
\fp_gzero:N 15394 \cs_new_protected:Npn \fp_gzero:N #1 { \fp_gset_eq:NN #1 \c_zero_fp }
\fp_gzero:c 15395 \cs_generate_variant:Nn \fp_zero:N { c }
15396 \cs_generate_variant:Nn \fp_gzero:N { c }
```

(End definition for `\fp_zero:N` and `\fp_gzero:N`. These functions are documented on page 185.)

`\fp_zero_new:N` Set the floating point to zero, or define it if needed.

```

\fp_zero_new:c 15397 \cs_new_protected:Npn \fp_zero_new:N #1
15398 { \fp_if_exist:NTF #1 { \fp_zero:N #1 } { \fp_new:N #1 } }
\fp_gzero_new:N 15399 \cs_new_protected:Npn \fp_gzero_new:N #1
15400 { \fp_if_exist:NTF #1 { \fp_gzero:N #1 } { \fp_new:N #1 } }
15401 \cs_generate_variant:Nn \fp_zero_new:N { c }
15402 \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 186.)

## 27.2 Updating values

These match the equivalent functions in `\3int` and `\3skip`.

`\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.

```

\fp_sub:cN 15403 \cs_new_protected:Npn \fp_add:Nn { \_fp_add:NNNn \fp_set:Nn + }
\fp_gadd:Nn 15404 \cs_new_protected:Npn \fp_gadd:Nn { \_fp_add:NNNn \fp_gset:Nn + }
\fp_gsub:cN 15405 \cs_new_protected:Npn \fp_sub:Nn { \_fp_add:NNNn \fp_set:Nn - }
\fp_gsub:Nn 15406 \cs_new_protected:Npn \fp_gsub:Nn { \_fp_add:NNNn \fp_gset:Nn - }
15407 \cs_new_protected:Npn \_fp_add:NNNn #1#2#3#4
15408 { #1 #3 { #3 #2 \_fp_parse:n {#4} } }
15409 \cs_generate_variant:Nn \fp_add:Nn { c }
15410 \cs_generate_variant:Nn \fp_gadd:Nn { c }
15411 \cs_generate_variant:Nn \fp_sub:Nn { c }
15412 \cs_generate_variant:Nn \fp_gsub:Nn { c }
```

(End definition for `\fp_add:Nn` and others. These functions are documented on page 186.)

### 27.3 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.

```
15413 \cs_new_protected:Npn \fp_show:N { \_fp_show:NN \tl_show:n }
15414 \cs_generate_variant:Nn \fp_show:N { c }
15415 \cs_new_protected:Npn \fp_log:N { \_fp_show:NN \tl_log:n }
15416 \cs_generate_variant:Nn \fp_log:N { c }
15417 \cs_new_protected:Npn \_fp_show:NN #1#2
15418 {
15419     \_kernel_chk_defined:NT #2
15420     { \exp_args:Nx #1 { \token_to_str:N #2 = \fp_to_tl:N #2 } }
15421 }
```

(End definition for \fp\_show:N, \fp\_log:N, and \\_fp\_show:NN. These functions are documented on page 193.)

**\fp\_show:n** Use general tools.

```
15422 \cs_new_protected:Npn \fp_show:n
15423     { \msg_show_eval:Nn \fp_to_tl:n }
15424 \cs_new_protected:Npn \fp_log:n
15425     { \msg_log_eval:Nn \fp_to_tl:n }
```

(End definition for \fp\_show:n and \fp\_log:n. These functions are documented on page 193.)

### 27.4 Some useful constants and scratch variables

**\c\_one\_fp** Some constants.

```
15426 \fp_const:Nn \c_e_fp           { 2.718 2818 2845 9045 }
15427 \fp_const:Nn \c_one_fp          { 1 }
```

(End definition for \c\_one\_fp and \c\_e\_fp. These variables are documented on page 191.)

**\c\_pi\_fp** We simply round  $\pi$  to and  $\pi/180$  to 16 significant digits.

```
15428 \fp_const:Nn \c_pi_fp          { 3.141 5926 5358 9793 }
15429 \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 192.)

**\l\_tmpa\_fp** Scratch variables are simply initialized there.

```
15430 \fp_new:N \l_tmpa_fp
15431 \fp_new:N \l_tmpb_fp
15432 \fp_new:N \g_tmpa_fp
15433 \fp_new:N \g_tmpb_fp
```

(End definition for \l\_tmpa\_fp and others. These variables are documented on page 192.)

15434 ⟨/initex | package⟩

## 28 l3fp-logic Implementation

```
15435  {*initex | package}
15436  @@-fp
\__fp_parse_word_max:N Those functions may receive a variable number of arguments.
\__fp_parse_word_min:N
15437  \cs_new:Npn \__fp_parse_word_max:N
15438  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 2 }
15439  \cs_new:Npn \__fp_parse_word_min:N
15440  { \__fp_parse_function:NNN \__fp_minmax_o:Nw 0 }

(End definition for \__fp_parse_word_max:N and \__fp_parse_word_min:N.)
```

### 28.1 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.

### 28.2 Existence test

```
\fp_if_exist_p:N Copies of the cs functions defined in l3basics.
\fp_if_exist_p:c
15441 \prg_new_eq_conditional:NNn \fp_if_exist:N \cs_if_exist:N { TF , T , F , p }
15442 \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 188.)
```

### 28.3 Comparison

Within floating point expressions, comparison operators are treated as operations, so we evaluate #1, then compare with  $\pm 0$ . Tuples are `true`.

```
\__fp_compare_return:w
15443 \prg_new_conditional:Npnn \fp_compare:n #1 { p , T , F , TF }
15444  {
15445    \exp_after:wN \__fp_compare_return:w
15446    \exp:w \exp_end_continue_f:w \__fp_parse:n {#1}
15447  }
15448 \cs_new:Npn \__fp_compare_return:w #1#2#3;
15449  {
15450    \if_charcode:w 0
15451      \__fp_if_type_fp:NTwFw
15452        #1 { \use_i_delimit_by_q_stop:nw #3 \q_stop }
15453        \s__fp 1 \q_stop
15454        \prg_return_false:
15455    \else:
15456      \prg_return_true:
15457    \fi:
15458 }
```

(End definition for `\fp_compare:nTF` and `\_fp_compare_return:w`. This function is documented on page 189.)

`\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  $?$ .

```

15459 \prg_new_conditional:Npnn \fp_compare:nNn #1#2#3 { p , T , F , TF }
15460   {
15461     \if_int_compare:w
15462       \exp_after:wN \_fp_compare_aux:wn
15463         \exp:w \exp_end_continue_f:w \_fp_parse:n {#1} {#3}
15464         = \_fp_int_eval:w '#2 - ' = \_fp_int_eval_end:
15465         \prg_return_true:
15466       \else:
15467         \prg_return_false:
15468       \fi:
15469     }
15470   \cs_new:Npn \_fp_compare_aux:wn #1; #2
15471   {
15472     \exp_after:wN \_fp_compare_back_any:ww
15473       \exp:w \exp_end_continue_f:w \_fp_parse:n {#2} #1;
15474   }

```

(End definition for `\fp_compare:nNnTF` and `\_fp_compare_aux:wn`. This function is documented on page 188.)

`\_fp_compare_back_any:ww` 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.

```

15475 \cs_new:Npn \_fp_compare_back_any:ww #1#2; #3
15476   {
15477     \_fp_if_type_fp:NTwFw
15478       #1 { \_fp_if_type_fp:NTwFw #3 \use_i:nn \s_fp \use_ii:nn \q_stop }
15479         \s_fp \use_ii:nn \q_stop
15480     \_fp_compare_back:ww
15481   {
15482     \cs:w
15483       _fp
15484         \_fp_type_from_scan:N #1
15485         _compare_back
15486         \_fp_type_from_scan:N #3
15487         :ww
15488       \cs_end:
15489     }
15490   #1#2 ; #3
15491 }
15492 \cs_new:Npn \_fp_compare_back:ww
15493   \s_fp \_fp_chk:w #1 #2 #3;
15494   \s_fp \_fp_chk:w #4 #5 #6;

```

```

15495   {
15496     \int_value:w
15497       \if_meaning:w 3 #1 \exp_after:wN \__fp_compare_nan:w \fi:
15498       \if_meaning:w 3 #4 \exp_after:wN \__fp_compare_nan:w \fi:
15499       \if_meaning:w 2 #5 - \fi:
15500       \if_meaning:w #2 #5
15501         \if_meaning:w #1 #4
15502           \if_meaning:w 1 #1
15503             \__fp_compare_npos:nwnw #6; #3;
15504           \else:
15505             0
15506           \fi:
15507           \else:
15508             \if_int_compare:w #4 < #1 - \fi: 1
15509           \fi:
15510           \else:
15511             \if_int_compare:w #1#4 = 0 \exp_stop_f:
15512               0
15513             \else:
15514               1
15515             \fi:
15516           \fi:
15517           \exp_stop_f:
15518     }
15519 \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:w

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:).

```

15520 \cs_new:Npn \__fp_compare_back_tuple:ww #1; #2; { 2 }
15521 \cs_new:Npn \__fp_tuple_compare_back:ww #1; #2; { 2 }
15522 \cs_new:Npn \__fp_tuple_compare_back_tuple:ww
15523   \s__fp_tuple \__fp_tuple_chk:w #1;
15524   \s__fp_tuple \__fp_tuple_chk:w #2;
15525   {
15526     \int_compare:nNnTF { \__fp_array_count:n {#1} } =
15527       { \__fp_array_count:n {#2} }
15528     {
15529       \int_value:w 0
15530       \__fp_tuple_compare_back_loop:w
15531         #1 { \s__fp \prg_break: } ; @
15532         #2 { \s__fp \prg_break: } ;
15533         \prg_break_point:
15534       \exp_stop_f:
15535     }
15536   { 2 }
15537 }
15538 \cs_new:Npn \__fp_tuple_compare_back_loop:w #1#2 ; #3 0 #4#5 ;
15539 {
15540   \use_none:n #1
15541   \use_none:n #4

```

```

15542     \if_int_compare:w
15543         \__fp_compare_back_any:ww #1 #2 ; #4 #5 ; = 0 \exp_stop_f:
15544     \else:
15545         2 \exp_after:wN \prg_break:
15546     \fi:
15547     \__fp_tuple_compare_back_loop:w #3 @
15548 }

```

(End definition for `\__fp_compare_back_tuple:ww` and others.)

`\__fp_compare_npos:nwnw`  
`\_fp_compare significand:nnnnnnnn`

`\__fp_compare_npos:nwnw {\langle expo1\rangle} {\langle body1\rangle} ; {\langle expo2\rangle} {\langle body2\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.

```

15549 \cs_new:Npn \__fp_compare_npos:nwnw #1#2; #3#4;
15550 {
15551     \if_int_compare:w #1 = #3 \exp_stop_f:
15552         \__fp_compare_significand:nnnnnnnn #2 #4
15553     \else:
15554         \if_int_compare:w #1 < #3 - \fi: 1
15555     \fi:
15556 }
15557 \cs_new:Npn \__fp_compare_significand:nnnnnnnn #1#2#3#4#5#6#7#8
15558 {
15559     \if_int_compare:w #1#2 = #5#6 \exp_stop_f:
15560         \if_int_compare:w #3#4 = #7#8 \exp_stop_f:
15561             0
15562         \else:
15563             \if_int_compare:w #3#4 < #7#8 - \fi: 1
15564         \fi:
15565     \else:
15566         \if_int_compare:w #1#2 < #5#6 - \fi: 1
15567     \fi:
15568 }

```

(End definition for `\__fp_compare_npos:nwnw` and `\__fp_compare_significand:nnnnnnnn`.)

## 28.4 Floating point expression loops

`\fp_do_until:nn`  
`\fp_do_while:nn`  
`\fp_until_do:nn`  
`\fp_while_do:nn`

```

15569 \cs_new:Npn \fp_do_until:nn #1#2
15570 {
15571     #2
15572     \fp_compare:nF {#1}
15573         { \fp_do_until:nn {#1} {#2} }
15574     }
15575 \cs_new:Npn \fp_do_while:nn #1#2
15576 {
15577     #2
15578     \fp_compare:nT {#1}

```

```

15579      { \fp_do_while:nn {#1} {#2} }
15580    }
15581 \cs_new:Npn \fp_until_do:nn #1#2
15582  {
15583    \fp_compare:nF {#1}
15584    {
15585      #2
15586      \fp_until_do:nn {#1} {#2}
15587    }
15588  }
15589 \cs_new:Npn \fp_while_do:nn #1#2
15590  {
15591    \fp_compare:nT {#1}
15592    {
15593      #2
15594      \fp_while_do:nn {#1} {#2}
15595    }
15596  }

```

(End definition for `\fp_do_until:nn` and others. These functions are documented on page 190.)

`\fp_do_until:nNnn` As above but not using the nNn syntax.

```

\fp_do_while:nNnn
\fp_until_do:nNnn
\fp_while_do:nNnn
15597 \cs_new:Npn \fp_do_until:nNnn #1#2#3#4
15598  {
15599    #4
15600    \fp_compare:nNnF {#1} #2 {#3}
15601    { \fp_do_until:nNnn {#1} #2 {#3} {#4} }
15602  }
15603 \cs_new:Npn \fp_do_while:nNnn #1#2#3#4
15604  {
15605    #4
15606    \fp_compare:nNnT {#1} #2 {#3}
15607    { \fp_do_while:nNnn {#1} #2 {#3} {#4} }
15608  }
15609 \cs_new:Npn \fp_until_do:nNnn #1#2#3#4
15610  {
15611    \fp_compare:nNnF {#1} #2 {#3}
15612    {
15613      #4
15614      \fp_until_do:nNnn {#1} #2 {#3} {#4}
15615    }
15616  }
15617 \cs_new:Npn \fp_while_do:nNnn #1#2#3#4
15618  {
15619    \fp_compare:nNnT {#1} #2 {#3}
15620    {
15621      #4
15622      \fp_while_do:nNnn {#1} #2 {#3} {#4}
15623    }
15624  }

```

(End definition for `\fp_do_until:nNnn` and others. These functions are documented on page 189.)

`\fp_step_function:nnnN` The approach here is somewhat similar to `\int_step_function:nnnN`. There are two  
`\fp_step_function:nncn` subtleties: we use the internal parser `\_fp_parse:n` to avoid converting back and forth

```

\__fp_step:wwwN
\__fp_step_fp:wwwN
\__fp_step:NnnnnN
\__fp_step:NfnnnN

```

from the internal representation; and (due to rounding) even a non-zero step does not guarantee that the loop counter increases.

```

15625 \cs_new:Npn \fp_step_function:nnnN #1#2#3
15626 {
15627   \exp_after:wN \__fp_step:wwwN
15628     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#1}
15629     \exp:w \exp_end_continue_f:w \__fp_parse_o:n {#2}
15630     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
15631 }
15632 \cs_generate_variant:Nn \fp_step_function:nnnN { nnnc }
15633 %           \end{macrocode}
15634 % Only floating point numbers (not tuples) are allowed arguments.
15635 % Only \enquote{normal} floating points (not $\pm 0$,
15636 % $\pm\texttt{inf}$, $\texttt{nan}$) can be used as step; if positive,
15637 % call \cs{\__fp_step:NnnnnN} with argument $|>|$ otherwise $\mid<|$. This
15638 % function has one more argument than its integer counterpart, namely
15639 % the previous value, to catch the case where the loop has made no
15640 % progress. Conversion to decimal is done just before calling the
15641 % user's function.
15642 %           \begin{macrocode}
15643 \cs_new:Npn \__fp_step:wwwN #1#2; #3#4; #5#6; #7
15644 {
15645   \__fp_if_type_fp:NTwFw #1 { } \s__fp \prg_break: \q_stop
15646   \__fp_if_type_fp:NTwFw #3 { } \s__fp \prg_break: \q_stop
15647   \__fp_if_type_fp:NTwFw #5 { } \s__fp \prg_break: \q_stop
15648   \use_i:nnnn { \__fp_step_fp:wwwN #1#2; #3#4; #5#6; #7 }
15649   \prg_break_point:
15650   \use:n
15651   {
15652     \__fp_error:nfff { fp-step-tuple } { \fp_to_tl:n { #1#2 ; } }
15653     { \fp_to_tl:n { #3#4 ; } } { \fp_to_tl:n { #5#6 ; } }
15654   }
15655 }
15656 \cs_new:Npn \__fp_step_fp:wwwN #1 ; \s__fp \__fp_chk:w #2#3#4 ; #5; #6
15657 {
15658   \token_if_eq_meaning:NNTF #2 1
15659   {
15660     \token_if_eq_meaning:NNTF #3 0
15661     { \__fp_step:NnnnnN > }
15662     { \__fp_step:NnnnnN < }
15663   }
15664   {
15665     \token_if_eq_meaning:NNTF #2 0
15666     {
15667       \__kernel_msg_expandable_error:nnn { kernel }
15668       { zero-step } {#6}
15669     }
15670   {
15671     \__fp_error:nfn { fp-bad-step } { }
15672     { \fp_to_tl:n { \s__fp \__fp_chk:w #2#3#4 ; } } {#6}
15673   }
15674   \use_none:nnnn
15675 }
15676 { #1 ; } { \c_nan_fp } { \s__fp \__fp_chk:w #2#3#4 ; } { #5 ; } #6

```

```

15677      }
15678 \cs_new:Npn \__fp_step:NnnnnN #1#2#3#4#5#6
15679  {
15680    \fp_compare:nNnTF {#2} = {#3}
15681    {
15682      \__fp_error:nffn { fp-tiny-step }
15683      { \fp_to_tl:n {#3} } { \fp_to_tl:n {#4} } {#6}
15684    }
15685    {
15686      \fp_compare:nNnF {#2} #1 {#5}
15687      {
15688        \exp_args:Nf #6 { \__fp_to_decimal_dispatch:w #2 }
15689        \__fp_step:NfnnnN
15690        #1 { \__fp_parse:n {#2 + #4} } {#2} {#4} {#5} #6
15691      }
15692    }
15693  }
15694 \cs_generate_variant:Nn \__fp_step:NnnnnN { Nf }
```

(End definition for `\fp_step_function:nnnN` and others. This function is documented on page 191.)

`\fp_step_inline:nnn`  
`\fp_step_variable:nnnNn`

As for `\int_step_inline:nnnn`, create a global function and apply it, following up with a break point.

```

15695 \cs_new_protected:Npn \fp_step_inline:nnnn
15696  {
15697    \int_gincr:N \g_kernel_prg_map_int
15698    \exp_args:NNc \__fp_step>NNnnnn
15699    \cs_gset_protected:Npn
15700    { __fp_map_ \int_use:N \g_kernel_prg_map_int :w }
15701  }
15702 \cs_new_protected:Npn \fp_step_variable:nnnNn #1#2#3#4#5
15703  {
15704    \int_gincr:N \g_kernel_prg_map_int
15705    \exp_args:NNc \__fp_step>NNnnnn
15706    \cs_gset_protected:Npx
15707    { __fp_map_ \int_use:N \g_kernel_prg_map_int :w }
15708    {#1} {#2} {#3}
15709    {
15710      \tl_set:Nn \exp_not:N #4 {##1}
15711      \exp_not:n {#5}
15712    }
15713  }
15714 \cs_new_protected:Npn \__fp_step>NNnnnn #1#2#3#4#5#6
15715  {
15716    #1 #2 ##1 {#6}
15717    \fp_step_function:nnnN {#3} {#4} {#5} #2
15718    \prg_break_point:Nn \scan_stop: { \int_gdecr:N \g_kernel_prg_map_int }
15719  }
```

(End definition for `\fp_step_inline:nnnn`, `\fp_step_variable:nnnNn`, and `\__fp_step>NNnnnn`. These functions are documented on page 191.)

```

15720 \__kernel_msg_new:nnn { kernel } { fp-step-tuple }
15721  { Tuple~argument~in~fp_step_...~{#1}{#2}{#3}. }
15722 \__kernel_msg_new:nnn { kernel } { fp-bad-step }
```

```

15723   { Invalid~step~size~#2~in~step~function~#3. }
15724 \__kernel_msg_new:nnn { kernel } { fp-tiny-step }
15725   { Tiny~step~size~(#1+#2=#1)~in~step~function~#3. }

```

## 28.5 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`.

```

15726 \cs_new:Npn \__fp_minmax_o:Nw #1
15727   {
15728     \__fp_parse_function_all_fp_o:fnw
15729       { \token_if_eq_meaning:NNTF 0 #1 { min } { max } }
15730       { \__fp_minmax_aux_o:Nw #1 }
15731   }
15732 \cs_new:Npn \__fp_minmax_aux_o:Nw #1#2 @
15733   {
15734     \if_meaning:w 0 #1
15735       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN +
15736     \else:
15737       \exp_after:wN \__fp_minmax_loop:Nww \exp_after:wN -
15738     \fi:
15739   #2
15740   \s__fp \__fp_chk:w 2 #1 \s__fp_exact ;
15741   \s__fp \__fp_chk:w { 3 \__fp_minmax_break_o:w } ;
15742 }
```

(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.

```

15743 \cs_new:Npn \__fp_minmax_loop:Nww
15744   #1 \s__fp \__fp_chk:w #2#3; \s__fp \__fp_chk:w #4#5;
15745   {
15746     \if_meaning:w 3 #4
15747       \if_meaning:w 3 #2
15748         \__fp_minmax_auxi:ww
15749       \else:
15750         \__fp_minmax_auxii:ww
15751       \fi:
15752     \else:
15753       \if_int_compare:w
15754         \__fp_compare_back:ww
```

```

15755           \s_fp \fp_chk:w #4#5;
15756           \s_fp \fp_chk:w #2#3;
15757 = #1 1 \exp_stop_f:
15758     \fp_minmax_auxii:ww
15759   \else:
15760     \fp_minmax_auxi:ww
15761   \fi:
15762 \fi:
15763 \fp_minmax_loop:Nww #1
15764   \s_fp \fp_chk:w #2#3;
15765   \s_fp \fp_chk:w #4#5;
15766 }

```

(End definition for `\fp_minmax_loop:Nww`.)

`\fp_minmax_auxi:ww` Keep the first/second number, and remove the other.

```

15767 \cs_new:Npn \fp_minmax_auxi:ww #1 \fi: \fi: #2 \s_fp #3 ; \s_fp #4;
15768   { \fi: \fi: #2 \s_fp #3 ; }
15769 \cs_new:Npn \fp_minmax_auxii:ww #1 \fi: \fi: #2 \s_fp #3 ;
15770   { \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.

```

15771 \cs_new:Npn \fp_minmax_break_o:w #1 \fi: \fi: #2 \s_fp #3; #4;
15772   { \fi: \fp_exp_after_o:w \s_fp #3; }

```

(End definition for `\fp_minmax_break_o:w`.)

## 28.6 Boolean operations

`\fp_not_o:w` Return `true` or `false`, with two expansions, one to exit the conditional, and one to please `\fp_parse`. The first argument is provided by `\fp_parse` and is ignored.

```

15773 \cs_new:Npn \fp_not_o:w #1 \s_fp \fp_chk:w #2#3; @
15774   {
15775     \if_meaning:w 0 #2
15776       \exp_after:wN \exp_after:wN \exp_after:wN \c_one_fp
15777     \else:
15778       \exp_after:wN \exp_after:wN \exp_after:wN \c_zero_fp
15779     \fi:
15780   }
15781 \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` 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.

```

15782 \group_begin:
15783   \char_set_catcode_letter:N &

```

```

\fp_tuple_&_tuple_o:ww
\fp_and_return:wNw

```

```

15784   \char_set_catcode_letter:N |
15785   \cs_new:Npn \__fp_&_o:ww #1 \s__fp \__fp_chk:w #2#3;
15786   {
15787     \if_meaning:w 0 #2 #1
15788       \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
15789     \fi:
15790     \__fp_exp_after_o:w
15791   }
15792   \cs_new:Npn \__fp_&_tuple_o:ww #1 \s__fp \__fp_chk:w #2#3;
15793   {
15794     \if_meaning:w 0 #2 #1
15795       \__fp_and_return:wNw \s__fp \__fp_chk:w #2#3;
15796     \fi:
15797     \__fp_exp_after_tuple_o:w
15798   }
15799   \cs_new:Npn \__fp_tuple_&_o:ww #1; { \__fp_exp_after_o:w }
15800   \cs_new:Npn \__fp_tuple_&_tuple_o:ww #1; { \__fp_exp_after_tuple_o:w }
15801   \cs_new:Npn \__fp_|_o:ww { \__fp_&_o:ww \else: }
15802   \cs_new:Npn \__fp_|_tuple_o:ww { \__fp_&_tuple_o:ww \else: }
15803   \cs_new:Npn \__fp_tuple_|_o:ww #1; #2; { \__fp_exp_after_tuple_o:w #1; }
15804   \cs_new:Npn \__fp_tuple_|_tuple_o:ww #1; #2;
15805   { \__fp_exp_after_tuple_o:w #1; }
15806 \group_end:
15807 \cs_new:Npn \__fp_and_return:wNw #1; \fi: #2;
15808   { \fi: \__fp_exp_after_o:w #1; }

```

(End definition for `\__fp_&_o:ww` and others.)

## 28.7 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.

```

15809 \cs_new:Npn \__fp_ternary:NwwN #1 #2#3@ #4@ #5
15810   {
15811     \if_meaning:w \__fp_parse_infix_::N #5
15812     \if_charcode:w 0
15813       \__fp_if_type_fp:NTwFw
15814         #2 { \use_i:nn \use_i_delimit_by_q_stop:nw #3 \q_stop }
15815         \s__fp 1 \q_stop
15816       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxii:NwwN
15817     \else:
15818       \exp_after:wN \exp_after:wN \exp_after:wN \__fp_ternary_auxi:NwwN
15819     \fi:
15820     \exp_after:wN #1
15821     \exp:w \exp_end_continue_f:w
15822     \__fp_exp_after_array_f:w #4 \s__fp_stop
15823     \exp_after:wN @
15824     \exp:w
15825       \__fp_parse_operand:Nw \c__fp_prec_colon_int
15826       \__fp_parse_expand:w
15827   \else:
15828     \__kernel_msg_expandable_error:nnnn

```

```

15829      { kernel } { fp-missing } { : } { ~for~?: }
15830      \exp_after:wN \_fp_parse_continue:NwN
15831      \exp_after:wN #1
15832      \exp:w \exp_end_continue_f:w
15833      \_fp_exp_after_array_f:w #4 \s_fp_stop
15834      \exp_after:wN #5
15835      \exp_after:wN #1
15836      \fi:
15837      }
15838 \cs_new:Npn \_fp_ternary_auxi:NwwN #1#2@#3@#4
15839  {
15840      \exp_after:wN \_fp_parse_continue:NwN
15841      \exp_after:wN #1
15842      \exp:w \exp_end_continue_f:w
15843      \_fp_exp_after_array_f:w #2 \s_fp_stop
15844      #4 #1
15845      }
15846 \cs_new:Npn \_fp_ternary_auxii:NwwN #1#2@#3@#4
15847  {
15848      \exp_after:wN \_fp_parse_continue:NwN
15849      \exp_after:wN #1
15850      \exp:w \exp_end_continue_f:w
15851      \_fp_exp_after_array_f:w #3 \s_fp_stop
15852      #4 #1
15853  }

```

(End definition for `\_fp_ternary:NwwN`, `\_fp_ternary_auxi:NwwN`, and `\_fp_ternary_auxii:NwwN`.)

```
15854  ⟨/initex | package⟩
```

## 29 l3fp-basics Implementation

```
15855  ⟨*initex | package⟩
```

```
15856  ⟨@@=fp⟩
```

The l3fp-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.

```

\fp_parse_word_abs:N
\fp_parse_word_logb:N
\fp_parse_word_sign:N
\fp_parse_word_sqrt:N
15857 \cs_new:Npn \_fp_parse_word_abs:N
15858  { \_fp_parse_unary_function:NNN \_fp_set_sign_o:w 0 }
15859 \cs_new:Npn \_fp_parse_word_logb:N
15860  { \_fp_parse_unary_function:NNN \_fp_logb_o:w ? }
15861 \cs_new:Npn \_fp_parse_word_sign:N
15862  { \_fp_parse_unary_function:NNN \_fp_sign_o:w ? }
15863 \cs_new:Npn \_fp_parse_word_sqrt:N
15864  { \_fp_parse_unary_function:NNN \_fp_sqrt_o:w ? }

```

(End definition for `\_fp_parse_word_abs:N` and others.)

## 29.1 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 `l3fp-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.

### 29.1.1 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`.

```
15865 \cs_new:cpx { _fp\_o:ww } \s_fp
15866   {
15867     \exp_not:c { _fp\+_o:ww }
15868     \exp_not:n { \s_fp \_fp_neg_sign:N }
15869 }
```

(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 `\langle types \rangle` #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 `\langle sign_2 \rangle` (expansion of #1#5) as an argument. If the `\langle types \rangle` are distinct, the result is simply the floating point number with the highest `\langle type \rangle`. Since case 3 (used for two `nan`) also picks the first operand, we can also use it when `\langle type_1 \rangle` is greater than `\langle type_2 \rangle`. Also note that we don't need to worry about `\langle sign_2 \rangle` in that case since the second operand is discarded.

```
15870 \cs_new:cpxn { _fp\+_o:ww }
15871   \s_fp #1 \_fp_chk:w #2 #3 ; \s_fp \_fp_chk:w #4 #5
```

```

15872   {
15873     \if_case:w
15874       \if_meaning:w #2 #4
15875         #2
15876       \else:
15877         \if_int_compare:w #2 > #4 \exp_stop_f:
15878           3
15879         \else:
15880           4
15881         \fi:
15882       \fi:
15883     \exp_stop_f:
15884       \exp_after:wN \__fp_add_zeros_o:Nww \int_value:w
15885     \or: \exp_after:wN \__fp_add_normal_o:Nww \int_value:w
15886     \or: \exp_after:wN \__fp_add_inf_o:Nww \int_value:w
15887     \or: \__fp_case_return_i_o:ww
15888     \else: \exp_after:wN \__fp_add_return_ii_o:Nww \int_value:w
15889     \fi:
15890   #1 #5
15891   \s__fp \__fp_chk:w #2 #3 ;
15892   \s__fp \__fp_chk:w #4 #5
15893 }

```

(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.

```

15894 \cs_new:Npn \__fp_add_return_ii_o:Nww #1 #2 ; \s__fp \__fp_chk:w #3 #4
15895   { \__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  $-0$ .

```

15896 \cs_new:Npn \__fp_add_zeros_o:Nww #1 \s__fp \__fp_chk:w 0 #2
15897   {
15898     \if_int_compare:w #2 #1 = 20 \exp_stop_f:
15899       \exp_after:wN \__fp_add_return_ii_o:Nww
15900     \else:
15901       \__fp_case_return_i_o:ww
15902     \fi:
15903   #1
15904   \s__fp \__fp_chk:w 0 #2
15905 }

```

(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.

```

15906 \cs_new:Npn \__fp_add_inf_o:Nww
15907   #1 \s__fp \__fp_chk:w 2 #2 #3; \s__fp \__fp_chk:w 2 #4
15908   {
15909     \if_meaning:w #1 #2
15910       \__fp_case_return_i_o:ww

```

```

15911     \else:
15912         \_fp_case_use:nw
15913         {
15914             \exp_last_unbraced:Nf \_fp_invalid_operation_o:Nww
15915             { \token_if_eq_meaning:NNTF #1 #4 + - }
15916         }
15917     \fi:
15918     \s_fp \_fp_chk:w 2 #2 #3;
15919     \s_fp \_fp_chk:w 2 #4
15920 }

```

(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.

```

15921 \cs_new:Npn \_fp_add_normal_o:Nww #1 \s_fp \_fp_chk:w 1 #2
15922 {
15923     \if_meaning:w #1#2
15924         \exp_after:wN \_fp_add_npos_o:NnwNnw
15925     \else:
15926         \exp_after:wN \_fp_sub_npos_o:NnwNnw
15927     \fi:
15928     #2
15929 }

```

(End definition for `\_fp_add_normal_o:Nww`.)

### 29.1.2 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.

```

15930 \cs_new:Npn \_fp_add_npos_o:NnwNnw #1#2#3 ; \s_fp \_fp_chk:w 1 #4 #5
15931 {
15932     \exp_after:wN \_fp_sanitize:Nw
15933     \exp_after:wN #1
15934     \int_value:w \_fp_int_eval:w
15935     \if_int_compare:w #2 > #5 \exp_stop_f:
15936         #2
15937         \exp_after:wN \_fp_add_big_i_o:wNww \int_value:w -
15938     \else:
15939         #5
15940         \exp_after:wN \_fp_add_big_ii_o:wNww \int_value:w

```

```

15941     \fi:
15942     \_fp_int_eval:w #5 - #2 ; #1 #3;
15943 }

```

(End definition for `\_fp_add_npos_o:NnwNnw.`)

`\_fp_add_big_i_o:wNww`

`\_fp_add_big_ii_o:wNww` Used in l3fp-expo. Shift the significand of the small number, then add with `\_fp-add_significand_o:NnnwnnnnN`.

```

15944 \cs_new:Npn \_fp_add_big_i_o:wNww #1; #2 #3; #4;
15945 {
15946     \_fp_decimate:nNnnnn {#1}
15947     \_fp_add_significand_o:NnnwnnnnN
15948     #4
15949     #3
15950     #2
15951 }
15952 \cs_new:Npn \_fp_add_big_ii_o:wNww #1; #2 #3; #4;
15953 {
15954     \_fp_decimate:nNnnnn {#1}
15955     \_fp_add_significand_o:NnnwnnnnN
15956     #3
15957     #4
15958     #2
15959 }

```

(End definition for `\_fp_add_big_i_o:wNww` and `\_fp_add_big_ii_o:wNww.`)

`\_fp_add_significand_o:NnnwnnnnN` *rounding digit* { $\langle Y_1 \rangle$ } { $\langle Y_2 \rangle$ } { $\langle X_1 \rangle$ } { $\langle X_2 \rangle$ } { $\langle X_3 \rangle$ } { $\langle X_4 \rangle$ } *final sign*

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.

```

15960 \cs_new:Npn \_fp_add_significand_o:NnnwnnnnN #1 #2#3 #4; #5#6#7#8
15961 {
15962     \exp_after:wN \_fp_add_significand_test_o:N
15963     \int_value:w \_fp_int_eval:w 1#5#6 + #2
15964     \exp_after:wN \_fp_add_significand_pack:NNNNNNNN
15965     \int_value:w \_fp_int_eval:w 1#7#8 + #3 ; #1
15966 }
15967 \cs_new:Npn \_fp_add_significand_pack:NNNNNNNN #1 #2#3#4#5#6#7
15968 {
15969     \if_meaning:w 2 #1
15970     + 1
15971     \fi:
15972     ; #2 #3 #4 #5 #6 #7 ;
15973 }
15974 \cs_new:Npn \_fp_add_significand_test_o:N #1
15975 {
15976     \if_meaning:w 2 #1
15977     \exp_after:wN \_fp_add_significand_carry_o:wwwNN
15978     \else:

```

```

15979      \exp_after:wN \__fp_add_significand_no_carry_o:wwwNN
15980      \fi:
15981  }
(End definition for \__fp_add_significand_o:NnnnnnnnN, \__fp_add_significand_pack:NNNNNNN, and
\__fp_add_significand_test_o:N.)
```

\\_\_fp\_add\_significand\_no\_carry\_o:wwwNN  
 $\backslash\_\_fp\_add\_significand\_no\_carry\_o:wwwNN \langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding digit \rangle \langle sign \rangle$

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.

```

15982 \cs_new:Npn \__fp_add_significand_no_carry_o:wwwNN
15983      #1; #2; #3#4 ; #5#6
15984  {
15985      \exp_after:wN \__fp_basics_pack_high:NNNNNw
15986      \int_value:w \__fp_int_eval:w 1 #1
15987      \exp_after:wN \__fp_basics_pack_low:NNNNNw
15988      \int_value:w \__fp_int_eval:w 1 #2 #3#4
15989      + \__fp_round:NNN #6 #4 #5
15990      \exp_after:wN ;
15991  }
```

(End definition for \\_\_fp\_add\_significand\_no\_carry\_o:wwwNN.)

\\_\_fp\_add\_significand\_carry\_o:wwwNN  
 $\backslash\_\_fp\_add\_significand\_carry\_o:wwwNN \langle 8d \rangle ; \langle 6d \rangle ; \langle 2d \rangle ; \langle rounding digit \rangle \langle sign \rangle$

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.

```

15992 \cs_new:Npn \__fp_add_significand_carry_o:wwwNN
15993      #1; #2; #3#4; #5#6
15994  {
15995      + 1
15996      \exp_after:wN \__fp_basics_pack_weird_high:NNNNNNNNw
15997      \int_value:w \__fp_int_eval:w 1 1 #1
15998      \exp_after:wN \__fp_basics_pack_weird_low:NNNNw
15999      \int_value:w \__fp_int_eval:w 1 #2#3 +
16000      \exp_after:wN \__fp_round:NNN
16001      \exp_after:wN #6
16002      \exp_after:wN #3
16003      \int_value:w \__fp_round_digit:Nw #4 #5 ;
16004      \exp_after:wN ;
16005  }
```

(End definition for \\_\_fp\_add\_significand\_carry\_o:wwwNN.)

### 29.1.3 Absolute subtraction

\\_\_fp\_sub\_npos\_o:NnwNnw  
 $\backslash\_\_fp\_sub\_npos\_o:NnwNnw \langle sign_1 \rangle \langle exp_1 \rangle \langle body_1 \rangle ; \backslash s\_fp \backslash\_\_fp\_chk:w 1$   
 $\langle initial sign_2 \rangle \langle exp_2 \rangle \langle body_2 \rangle ;$

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:Nnw with the opposite of  $\langle sign_1 \rangle$ .

```
16006 \cs_new:Npn \__fp_sub_npos_o:NnwNnw #1#2#3; \s_fp \__fp_chk:w 1 #4#5#6;
```

```

16007    {
16008        \if_case:w \__fp_compare_npos:nwnw {#2} #3; {#5} #6; \exp_stop_f:
16009            \exp_after:wN \__fp_sub_eq_o:Nnwnw
16010        \or:
16011            \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
16012        \else:
16013            \exp_after:wN \__fp_sub_npos_ii_o:Nnwnw
16014        \fi:
16015        #1 {#2} #3; {#5} #6;
16016    }
16017 \cs_new:Npn \__fp_sub_eq_o:Nnwnw #1#2; #3; { \exp_after:wN \c_zero_fp }
16018 \cs_new:Npn \__fp_sub_npos_ii_o:Nnwnw #1 #2; #3;
16019    {
16020        \exp_after:wN \__fp_sub_npos_i_o:Nnwnw
16021        \int_value:w \__fp_neg_sign:N #1
16022        #3; #2;
16023    }

```

(End definition for `\__fp_sub_npos_o:NnwNnw`, `\__fp_sub_eq_o:Nnwnw`, and `\__fp_sub_npos_ii_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.

```

16024 \cs_new:Npn \__fp_sub_npos_i_o:Nnwnw #1 #2#3; #4#5;
16025    {
16026        \exp_after:wN \__fp_sanitize:Nw
16027        \exp_after:wN #1
16028        \int_value:w \__fp_int_eval:w
16029        #2
16030        \if_int_compare:w #2 = #4 \exp_stop_f:
16031            \exp_after:wN \__fp_sub_back_near_o:nnnnnnnnN
16032        \else:
16033            \exp_after:wN \__fp_decimate:nNnnnn \exp_after:wN
16034                { \int_value:w \__fp_int_eval:w #2 - #4 - 1 \exp_after:wN }
16035            \exp_after:wN \__fp_sub_back_far_o:NnnwnnnnnN
16036        \fi:
16037        #5
16038        #3
16039        #1
16040    }

```

(End definition for `\__fp_sub_npos_i_o:Nnwnw`.)

`\__fp_sub_back_near_o:nnnnnnnnN`  $\{Y_1\}$   $\{Y_2\}$   $\{Y_3\}$   $\{Y_4\}$   $\{X_1\}$   
`\__fp_sub_back_near_pack:NNNNNNw`  $\{X_2\}$   $\{X_3\}$   $\{X_4\}$  *(final sign)*

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.

```

16041 \cs_new:Npn \__fp_sub_back_near_o:nnnnnnnnnN #1#2#3#4 #5#6#7#8 #9
16042 {
16043   \exp_after:wN \__fp_sub_back_near_after:wNNNNw
16044   \int_value:w \__fp_int_eval:w 10#5#6 - #1#2 - 11
16045   \exp_after:wN \__fp_sub_back_near_pack:NNNNNNw
16046   \int_value:w \__fp_int_eval:w 11#7#8 - #3#4 \exp_after:wN ;
16047 }
16048 \cs_new:Npn \__fp_sub_back_near_pack:NNNNNNw #1#2#3#4#5#6#7 ;
16049 { + #1#2 ; {#3#4#5#6} {#7} ; }
16050 \cs_new:Npn \__fp_sub_back_near_after:wNNNNw 10 #1#2#3#4 #5 ;
16051 {
16052   \if_meaning:w 0 #
16053     \exp_after:wN \__fp_sub_back_shift:wnnnn
16054   \fi:
16055   ; {#1#2#3#4} {#5}
16056 }

```

(End definition for `\__fp_sub_back_near_o:nnnnnnnnnN`, `\__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`

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.

```

16057 \cs_new:Npn \__fp_sub_back_shift:wnnnn ; #1#2
16058 {
16059   \exp_after:wN \__fp_sub_back_shift_ii:ww
16060   \int_value:w #1 #2 0 ;
16061 }
16062 \cs_new:Npn \__fp_sub_back_shift_ii:ww #1 0 ; #2#3 ;
16063 {
16064   \if_meaning:w @ #1 @
16065   - 7
16066   - \exp_after:wN \use_i:nnn
16067   \exp_after:wN \__fp_sub_back_shift_iii:NNNNNNNNw
16068   \int_value:w #2#3 0 ~ 123456789;
16069 \else:
16070   - \__fp_sub_back_shift_iii:NNNNNNNNw #1 123456789;
16071 \fi:
16072 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
16073 \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
16074 \exp_after:wN \__fp_sub_back_shift_iv:nnnnw
16075 \exp_after:wN ;
16076 \int_value:w
16077 #1 ~ #2#3 0 ~ 0000 0000 0000 000 ;
16078 }
16079 \cs_new:Npn \__fp_sub_back_shift_iii:NNNNNNNNw #1#2#3#4#5#6#7#8#9; {#8}
16080 \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:NnnwnnnN
\_\_fp\_sub\_back\_far\_o:NnnwnnnN rounding {\langle Y'_1\rangle} {\langle Y'_2\rangle}
extra-digits ; {\langle X_1\rangle} {\langle X_2\rangle} {\langle X_3\rangle} {\langle X_4\rangle} final sign
```

If the difference is greater than  $10^{\langle \text{expo}_x \rangle}$ , call the `very_far` auxiliary. If the result is less than  $10^{\langle \text{expo}_x \rangle}$ , 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).

```
16081 \cs_new:Npn \_fp_sub_back_far_o:NnnwnnnN #1 #2#3 #4; #5#6#7#8
16082 {
16083   \if_case:w
16084     \if_int_compare:w 1 #2 = #5#6 \use_i:nnnn #7 \exp_stop_f:
16085       \if_int_compare:w #3 = \use_none:n #7#8 0 \exp_stop_f:
16086         0
16087       \else:
16088         \if_int_compare:w #3 > \use_none:n #7#8 0 - \fi: 1
16089       \fi:
16090     \else:
16091       \if_int_compare:w 1 #2 > #5#6 \use_i:nnnn #7 - \fi: 1
16092     \fi:
16093     \exp_stop_f:
16094       \exp_after:wN \_fp_sub_back_quite_far_o:wwNN
16095     \or: \exp_after:wN \_fp_sub_back_very_far_o:wwwNN
16096     \else: \exp_after:wN \_fp_sub_back_not_far_o:wwwNN
16097     \fi:
16098     #2 ~ #3 ; #5 #6 ~ #7 #8 ; #1
16099 }
```

(End definition for `\_fp_sub_back_far_o:NnnwnnnN`.)

`\_fp_sub_back_quite_far_o:wwNN`  
`\_fp_sub_back_quite_far_i:i>NN`

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).

```
16100 \cs_new:Npn \_fp_sub_back_quite_far_o:wwNN #1; #2; #3#4
16101 {
16102   \exp_after:wN \_fp_sub_back_quite_far_i:iNN
16103   \exp_after:wN #3
16104   \exp_after:wN #4
16105 }
16106 \cs_new:Npn \_fp_sub_back_quite_far_i:iNN #1#2
16107 {
16108   \if_case:w \_fp_round_neg:NNN #2 0 #1
16109     \exp_after:wN \use_i:nn
16110   \else:
16111     \exp_after:wN \use_i:nn
16112   \fi:
16113   { ; {1000} {0000} {0000} {0000} ; }
16114   { - 1 ; {9999} {9999} {9999} {9999} ; }
16115 }
```

(End definition for `\_fp_sub_back Quite_far_o:wwwNN` and `\_fp_sub_back Quite_far_i: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.

```
16116 \cs_new:Npn \_fp_sub_back_not_far_o:wwwNN #1 ~ #2; #3 ~ #4; #5#6
16117 {
16118     - 1
16119     \exp_after:wN \_fp_sub_back_near_after:wNNNNw
16120     \int_value:w \_fp_int_eval:w 1#30 - #1 - 11
16121     \exp_after:wN \_fp_sub_back_near_pack:NNNNNNw
16122     \int_value:w \_fp_int_eval:w 11 0000 0000 + #40 - #2
16123     - \exp_after:wN \_fp_round_neg:NNN
16124     \exp_after:wN #6
16125     \use_none:nnnnnnn #2 #5
16126     \exp_after:wN ;
16127 }
```

(End definition for `\_fp_sub_back_not_far_o:wwwNN`.)

`\_fp_sub_back_very_far_o:wwwNN`  
`\_fp_sub_back_very_far_i:nnNwwNN`

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.

```
16128 \cs_new:Npn \_fp_sub_back_very_far_o:wwwNN #1#2#3#4#5#6#7
16129 {
16130     \_fp_pack_eight:wNNNNNNNN
16131     \_fp_sub_back_very_far_i:nnNwwNN
16132     { 0 #1#2#3 #4#5#6#7 }
16133     ;
16134 }
16135 \cs_new:Npn \_fp_sub_back_very_far_i:nnNwwNN #1#2 ; #3 ; #4 ~ #5; #6#7
16136 {
16137     \exp_after:wN \_fp_basics_pack_high:NNNNNw
16138     \int_value:w \_fp_int_eval:w 1#4 - #1 - 1
16139     \exp_after:wN \_fp_basics_pack_low:NNNNNw
16140     \int_value:w \_fp_int_eval:w 2#5 - #2
16141     - \exp_after:wN \_fp_round_neg:NNN
16142     \exp_after:wN #7
16143     \int_value:w
16144         \if_int_odd:w \_fp_int_eval:w #5 - #2 \_fp_int_eval_end:
16145             1 \else: 2 \fi:
16146         \int_value:w \_fp_round_digit:Nw #3 #6 ;
16147     \exp_after:wN ;
```

```
16148 }
```

(End definition for `\_fp_sub_back_very_far_o:wwwwNN` and `\_fp_sub_back_very_far_i_o:nnNwwNN`.)

## 29.2 Multiplication

### 29.2.1 Signs, and special numbers

```
\_fp_*_o:ww
```

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`.

```
16149 \cs_new:cpn { \_fp_*_o:ww }
16150 {
16151     \_fp_mul_cases_o:NnNnww
16152     *
16153     { - 2 + }
16154     \_fp_mul_npos_o:Nww
16155     { }
16156 }
```

(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.

```
16157 \cs_new:Npn \_fp_mul_cases_o:NnNnww
16158     #1#2#3#4 \s_fp \_fp_chk:w #5#6#7; \s_fp \_fp_chk:w #8#9
16159 {
16160     \if_case:w \_fp_int_eval:w
16161         \if_int_compare:w #5 #8 = 11 ~
16162             1
16163         \else:
16164             \if_meaning:w 3 #8
16165                 3
16166             \else:
16167                 \if_meaning:w 3 #5
16168                     2
16169             \else:
16170                 \if_int_compare:w #5 #8 = 10 ~
16171                     9 #2 - 2
16172             \else:
16173                 (#5 #2 #8) / 2 * 2 + 7
16174         \fi:
```

```

16175           \fi:
16176           \fi:
16177           \fi:
16178           \if_meaning:w #6 #9 - 1 \fi:
16179           \_\_fp\_int\_eval\_end:
16180           \_\_fp\_case\_use:nw { #3 0 }
16181           \or: \_\_fp\_case\_use:nw { #3 2 }
16182           \or: \_\_fp\_case\_return\_i\_o:ww
16183           \or: \_\_fp\_case\_return\_ii\_o:ww
16184           \or: \_\_fp\_case\_return\_o:Nww \c_zero_fp
16185           \or: \_\_fp\_case\_return\_o:Nww \c_minus_zero_fp
16186           \or: \_\_fp\_case\_use:nw { \_\_fp\_invalid\_operation\_o:Nww #1 }
16187           \or: \_\_fp\_case\_use:nw { \_\_fp\_invalid\_operation\_o:Nww #1 }
16188           \or: \_\_fp\_case\_return\_o:Nww \c_inf_fp
16189           \or: \_\_fp\_case\_return\_o:Nww \c_minus_inf_fp
16190           #4
16191           \fi:
16192           \s\_fp \_\_fp\_chk:w #5 #6 #7;
16193           \s\_fp \_\_fp\_chk:w #8 #9
16194       }

```

(End definition for `\_\_fp\_mul\_cases\_o:nNnnww`.)

### 29.2.2 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.

```

16195   \cs_new:Npn \_\_fp\_mul\_npos\_o:Nww
16196     #1 \s\_fp \_\_fp\_chk:w #2 #3 #4 #5 ; \s\_fp \_\_fp\_chk:w #6 #7 #8 #9 ;
16197   {
16198     \exp_after:wN \_\_fp\_sanitize:Nw
16199     \exp_after:wN #1
16200     \int_value:w \_\_fp\_int\_eval:w
16201     #4 + #8
16202     \_\_fp\_mul\_significand\_o:nnnnNnnnn #5 #1 #9
16203   }

```

(End definition for `\_\_fp\_mul\_npos\_o:Nww`.)

```
\_\_fp\_mul\_significand\_o:nnnnNnnnn \_\_fp\_mul\_significand\_o:nnnnNnnnn {<X1>} {<X2>} {<X3>} {<X4>} <sign>
{<Y1>} {<Y2>} {<Y3>} {<Y4>}
```

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`.

```

16204 \cs_new:Npn \_fp_mul_significand_o:nnnnNnnnn #1#2#3#4 #5 #6#7#8#9
16205 {
16206     \exp_after:wN \_fp_mul_significand_test_f:NNN
16207     \exp_after:wN #5
16208     \int_value:w \_fp_int_eval:w 99990000 + #1*#6 +
16209     \exp_after:wN \_fp_mul_significand_keep:NNNNNw
16210     \int_value:w \_fp_int_eval:w 99990000 + #1*#7 + #2*#6 +
16211     \exp_after:wN \_fp_mul_significand_keep:NNNNNw
16212     \int_value:w \_fp_int_eval:w 99990000 + #1*#8 + #2*#7 + #3*#6 +
16213     \exp_after:wN \_fp_mul_significand_drop:NNNNNw
16214     \int_value:w \_fp_int_eval:w 99990000 + #1*#9 + #2*#8 +
16215     #3*#7 + #4*#6 +
16216     \exp_after:wN \_fp_mul_significand_drop:NNNNNw
16217     \int_value:w \_fp_int_eval:w 99990000 + #2*#9 + #3*#8 +
16218     #4*#7 +
16219     \exp_after:wN \_fp_mul_significand_drop:NNNNNw
16220     \int_value:w \_fp_int_eval:w 99990000 + #3*#9 + #4*#8 +
16221     \exp_after:wN \_fp_mul_significand_drop:NNNNNw
16222     \int_value:w \_fp_int_eval:w 100000000 + #4*#9 ;
16223 ; \exp_after:wN ;
16224 }
16225 \cs_new:Npn \_fp_mul_significand_drop:NNNNNw #1#2#3#4#5 #6;
16226 { #1#2#3#4#5 ; + #6 }
16227 \cs_new:Npn \_fp_mul_significand_keep:NNNNNw #1#2#3#4#5 #6;
16228 { #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`  $\langle sign \rangle$  1  $\langle digits\ 1-8 \rangle$  ;  $\langle digits\ 9-12 \rangle$  ;  
 $\langle digits\ 13-16 \rangle$  ; +  $\langle digits\ 17-20 \rangle$  +  $\langle digits\ 21-24 \rangle$  +  $\langle digits\ 25-28 \rangle$  +  $\langle digits\ 29-32 \rangle$  ; `\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.

```

16229 \cs_new:Npn \_fp_mul_significand_test_f:NNN #1 #2 #3
16230 {
16231     \if_meaning:w 0 #3
16232     \exp_after:wN \_fp_mul_significand_small_f:NNwwwN
16233     \else:
16234     \exp_after:wN \_fp_mul_significand_large_f:NwwNNNN
16235     \fi:
16236     #1 #3
16237 }
```

(End definition for `\_fp_mul_significand_test_f:NNN`.)

\\_fp\\_mul\\_significand\\_large\\_f:NwwNNNN 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.

```

16238 \cs_new:Npn \_fp_mul_significand_large_f:NwwNNNN #1 #2; #3; #4#5#6#7; +
16239 {
16240     \exp_after:wN \_fp_basics_pack_high:NNNNNw
16241     \int_value:w \_fp_int_eval:w 1#2
16242     \exp_after:wN \_fp_basics_pack_low:NNNNNw
16243     \int_value:w \_fp_int_eval:w 1#3#4#5#6#7
16244     + \exp_after:wN \_fp_round:NNN
16245     \exp_after:wN #1
16246     \exp_after:wN #7
16247     \int_value:w \_fp_round_digit:Nw
16248 }

```

(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.

```

16249 \cs_new:Npn \_fp_mul_significand_small_f:NNwwwN #1 #2#3; #4#5; #6; + #7
16250 {
16251     - 1
16252     \exp_after:wN \_fp_basics_pack_high:NNNNNw
16253     \int_value:w \_fp_int_eval:w 1#3#4
16254     \exp_after:wN \_fp_basics_pack_low:NNNNNw
16255     \int_value:w \_fp_int_eval:w 1#5#6#7
16256     + \exp_after:wN \_fp_round:NNN
16257     \exp_after:wN #1
16258     \exp_after:wN #7
16259     \int_value:w \_fp_round_digit:Nw
16260 }

```

(End definition for \\_fp\\_mul\\_significand\\_small\\_f:NNwwwN.)

## 29.3 Division

### 29.3.1 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.

```

16261 \cs_new:cpn { _fp_/_o:ww }
16262 {
16263     \_fp_mul_cases_o:NnNnww

```

```

16264      /
16265      { - }
16266      \__fp_div_npos_o:Nww
16267      {
16268          \or:
16269              \__fp_case_use:nw
16270                  { \__fp_division_by_zero_o>NNww \c_inf_fp / }
16271          \or:
16272              \__fp_case_use:nw
16273                  { \__fp_division_by_zero_o>NNww \c_minus_inf_fp / }
16274      }
16275  }

```

(End definition for  $\_\_fp\_/_o:ww$ .)

```

\__fp_div_npos_o:Nww \__fp_chk:w 1 <sign_A> {\langle exp A\rangle}
{\langle A_1\rangle} {\langle A_2\rangle} {\langle A_3\rangle} {\langle A_4\rangle} ; \__fp_chk:w 1 <sign_Z> {\langle exp Z\rangle}
{\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  $\langle final\ sign\rangle$ , 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  $\langle y\rangle$  obtained by adding 1 to the first 5 digits of  $Z$  (explanation given soon below), then the four  $\{\langle A_i\rangle\}$ , then the four  $\{\langle Z_i\rangle\}$ , a semi-colon, and the  $\langle final\ sign\rangle$ , used for rounding at the end.

```

16276  \cs_new:Npn \__fp_div_npos_o:Nww
16277      #1 \__fp_chk:w 1 #2 #3 #4 ; \__fp_chk:w 1 #5 #6 #7#8#9;
16278  {
16279      \exp_after:wN \__fp_sanitize:Nw
16280      \exp_after:wN #1
16281      \int_value:w \__fp_int_eval:w
16282          #3 - #6
16283          \exp_after:wN \__fp_div_significand_i_o:wnnw
16284              \int_value:w \__fp_int_eval:w #7 \use_i:nnnn #8 + 1 ;
16285              #4
16286              {#7}{#8}#9 ;
16287              #1
16288  }

```

(End definition for  $\_\_fp\_div\_npos_o:Nww$ .)

### 29.3.2 Work plan

In this subsection, we explain how to avoid overflowing T<sub>E</sub>X'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  $\varepsilon$ - $\text{\TeX}$ '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  $\text{\TeX}$  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  $\varepsilon$ - $\text{\TeX}$  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  $B$ :

$$\begin{aligned}
10^5 B &= A_1 A_2 0 + 10 \cdot 0 \cdot A_3 A_4 - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 \cdot Q_A \\
&< A_1 A_2 0 \cdot \left(1 - 10 \cdot \frac{Z_1 \cdot Z_2 Z_3 Z_4}{y}\right) + \frac{3}{2} \cdot 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4 + 10 \\
&\leq \frac{A_1 A_2 0 \cdot (y - 10 \cdot Z_1 \cdot Z_2 Z_3 Z_4)}{y} + \frac{3}{2} y + 10 \\
&\leq \frac{A_1 A_2 0 \cdot 1}{y} + \frac{3}{2} y + 10 \leq \frac{10^9 A}{y} + 1.6 \cdot y.
\end{aligned}$$

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  $(2^{31} - 1)/10^9 = 2.147 \dots$

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 T<sub>E</sub>X'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.

### 29.3.3 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  $\_\_fp\_div\_significand\_calc:wnnnnnnn$ . 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.
  16289 \cs_new:Npn \_\_fp\_div\_significand\_i\_o:wnnw #1 ; #2#3 #4 ;
  16290   {
  16291     \exp_after:wN \_\_fp\_div\_significand\_test_o:w
  16292     \int_value:w \_\_fp\_int\_eval:w
  16293     \exp_after:wN \_\_fp\_div\_significand\_calc:wnnnnnnn
  16294     \int_value:w \_\_fp\_int\_eval:w 999999 + #2 #3 0 / #1 ;
  16295       #2 #3 ;
  16296       #4
  16297       { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \int_value:w #1 }
  16298       { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \int_value:w #1 }
  16299       { \exp_after:wN \_\_fp\_div\_significand\_ii:wwn \int_value:w #1 }
  16300       { \exp_after:wN \_\_fp\_div\_significand\_iii:wwnnnn \int_value:w #1 }
  16301   }
```

(End definition for  $\_\_fp\_div\_significand\_i\_o:wnnw$ .)

```
\_\_fp\_div\_significand\_calc:wnnnnnnn
\_\_fp\_div\_significand\_calc_i:wnnnnnnn
\_\_fp\_div\_significand\_calc_ii:wnnnnnnn
\_\_fp\_div\_significand\_calc_iii:wnnnnnnn
```

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 `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 `\langle 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  $\#1, \#2, \dots$  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  $-8 \cdot 10^8 4$ , 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.

```

16302 \cs_new:Npn \__fp_div_significand_calc:wwnnnnnnn 1#1
16303 {
16304     \if_meaning:w 1 #1
16305         \exp_after:wN \__fp_div_significand_calc_i:wwnnnnnnn
16306     \else:
16307         \exp_after:wN \__fp_div_significand_calc_ii:wwnnnnnnn
16308     \fi:
16309 }
16310 \cs_new:Npn \__fp_div_significand_calc_i:wwnnnnnnn
16311     #1; #2;#3#4 #5#6#7#8 #9
16312 {
16313     1 1 #1
16314     #9 \exp_after:wN ;
16315     \int_value:w \__fp_int_eval:w \c_fp_Bigg_leading_shift_int
16316         + #2 - #1 * #5 - #5#60
16317         \exp_after:wN \__fp_pack_Bigg:NNNNNNw
16318         \int_value:w \__fp_int_eval:w \c_fp_Bigg_middle_shift_int
16319             + #3 - #1 * #6 - #70
16320             \exp_after:wN \__fp_pack_Bigg:NNNNNNw
16321             \int_value:w \__fp_int_eval:w \c_fp_Bigg_middle_shift_int
16322                 + #4 - #1 * #7 - #80
16323                 \exp_after:wN \__fp_pack_Bigg:NNNNNNw
16324                 \int_value:w \__fp_int_eval:w \c_fp_Bigg_trailing_shift_int

```

```

16325      - #1 * #8 ;
16326      {#5}{#6}{#7}{#8}
16327  }
16328 \cs_new:Npn \__fp_div_significand_calc_ii:wwnnnnnnn
16329  #1; #2;#3#4 #5#6#7#8 #9
16330  {
16331  1 0 #1
16332  #9 \exp_after:wN ;
16333  \int_value:w \__fp_int_eval:w \c_fp_Bigg_leading_shift_int
16334  + #2 - #1 * #5
16335  \exp_after:wN \__fp_pack_Bigg:NNNNNNw
16336  \int_value:w \__fp_int_eval:w \c_fp_Bigg_middle_shift_int
16337  + #3 - #1 * #6
16338  \exp_after:wN \__fp_pack_Bigg:NNNNNNw
16339  \int_value:w \__fp_int_eval:w \c_fp_Bigg_middle_shift_int
16340  + #4 - #1 * #7
16341  \exp_after:wN \__fp_pack_Bigg:NNNNNNw
16342  \int_value:w \__fp_int_eval:w \c_fp_Bigg_trailing_shift_int
16343  - #1 * #8 ;
16344  {#5}{#6}{#7}{#8}
16345  }

```

(End definition for `\__fp_div_significand_calc:wwnnnnnnn`, `\__fp_div_significand_calc_i:wwnnnnnnn`, and `\__fp_div_significand_calc_ii:wwnnnnnnn`)

`\__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  $B$ .

```

16346 \cs_new:Npn \__fp_div_significand_ii:wwn #1; #2;#3
16347  {
16348  \exp_after:wN \__fp_div_significand_pack:NNN
16349  \int_value:w \__fp_int_eval:w
16350  \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
16351  \int_value:w \__fp_int_eval:w 999999 + #2 #3 0 / #1 ; #2 #3 ;
16352  }

```

(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  $Q_E$ .

```

16353 \cs_new:Npn \__fp_div_significand_iii:wwnnnn #1; #2;#3#4#5 #6#7
16354  {
16355  0
16356  \exp_after:wN \__fp_div_significand_iv:wwnnnnnnn
16357  \int_value:w \__fp_int_eval:w ( 2 * #2 #3 ) / #6 #7 ; % <- P
16358  #2 ; {#3} {#4} {#5}
16359  {#6} {#7}

```

16360 }

(End definition for `\_fp_div_significand_iii:wwnnnnnn.`)

```
\_fp_div_significand_iv:wwnnnnnnn
\_\_fp\_div\_significand\_v:Nw
\_\_fp\_div\_significand\_vi:Nw
```

`\_fp_div_significand_iv:wwnnnnnnn`  $\langle P \rangle ; \langle E_1 \rangle ; \{ \langle E_2 \rangle \} \{ \langle E_3 \rangle \} \{ \langle E_4 \rangle \}$

$\{ \langle Z_1 \rangle \} \{ \langle Z_2 \rangle \} \{ \langle Z_3 \rangle \} \{ \langle Z_4 \rangle \} \langle sign \rangle$

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  $\langle rounding \rangle$  digit. This  $\langle rounding \rangle$  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:  $T = 0$ .

```
16361 \cs_new:Npn \_fp_div_significand_iv:wwnnnnnnn #1; #2;#3#4#5 #6#7#8#9
16362 {
16363     + 5 * #1
16364     \exp_after:wN \_fp_div_significand_vi:Nw
16365     \int_value:w \_fp_int_eval:w -20 + 2*#2#3 - #1*#6#7 +
16366         \exp_after:wN \_fp_div_significand_v>NN
16367         \int_value:w \_fp_int_eval:w 199980 + 2*#4 - #1*#8 +
16368         \exp_after:wN \_fp_div_significand_v>NN
16369         \int_value:w \_fp_int_eval:w 200000 + 2*#5 - #1*#9 ;
16370 }
16371 \cs_new:Npn \_fp_div_significand_v>NN #1#2 { #1#2 \_fp_int_eval_end: + }
16372 \cs_new:Npn \_fp_div_significand_vi:Nw #1#2;
16373 {
16374     \if_meaning:w 0 #1
16375         \if_int_compare:w \_fp_int_eval:w #2 > 0 + 1 \fi:
16376     \else:
16377         \if_meaning:w - #1 - \else: + \fi: 1
16378     \fi:
16379 ;
16380 }
```

(End definition for `\_fp_div_significand_iv:wwnnnnnnn`, `\_fp_div_significand_v>NN`, and `\_fp_div_significand_vi:Nw`.)

\\_fp\\_div\\_significand\\_pack:NNN At this stage, we are in the following situation: T<sub>E</sub>X is in the process of expanding several integer expressions, thus functions at the bottom expand before those above.

```
\_fp_div_significand_test_o:w 106 + QA \_fp_div_significand_
pack:NNN 106 + QB \_fp_div_significand_pack:NNN 106 + QC \_fp-
div_significand_pack:NNN 107 + 10 · QD + 5 · P + ε ; ⟨sign⟩
```

Here, ε = 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).

```
16381 \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.

```
16382 \cs_new:Npn \_fp_div_significand_test_o:w 10 #1
16383 {
16384     \if_meaning:w 0 #1
16385         \exp_after:wN \_fp_div_significand_small_o:wwwNNNNwN
16386     \else:
16387         \exp_after:wN \_fp_div_significand_large_o:wwwNNNNwN
16388     \fi:
16389     #1
16390 }
```

(End definition for \\_fp\_div\_significand\_test\_o:w.)

\\_fp\_div\_significand\_small\_o:wwwNNNNwN \\_fp\_div\_significand\_small\_o:wwwNNNNwN 0 ⟨4d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨5d⟩ ; ⟨final sign⟩

Standard use of the functions \\_fp\_basics\_pack\_low:NNNNNw and \\_fp\_basics\_pack\_high:NNNNNw. We finally get to use the ⟨final sign⟩ which has been sitting there for a while.

```
16391 \cs_new:Npn \_fp_div_significand_small_o:wwwNNNNwN
16392     0 #1; #2; #3; #4#5#6#7#8; #9
16393 {
16394     \exp_after:wN \_fp_basics_pack_high:NNNNNw
16395     \int_value:w \_fp_int_eval:w 1 #1#2
16396     \exp_after:wN \_fp_basics_pack_low:NNNNNw
16397     \int_value:w \_fp_int_eval:w 1 #3#4#5#6#7
16398     + \_fp_round:NNN #9 #7 #8
16399     \exp_after:wN ;
16400 }
```

(End definition for \\_fp\_div\_significand\_small\_o:wwwNNNNwN.)

```
\_\_fp\_div\_significand\_large\_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.

```
16401 \cs_new:Npn \_\_fp\_div\_significand\_large\_o:wwwNNNNwN
16402     #1; #2; #3; #4#5#6#7#8; #9
16403 {
16404     +
16405     \exp_after:wN \_\_fp\_basis_pack_weird_high:NNNNNNNNw
16406     \int_value:w \_\_fp\_int\_eval:w 1 #1 #2
16407     \exp_after:wN \_\_fp\_basis_pack_weird_low:NNNNw
16408     \int_value:w \_\_fp\_int\_eval:w 1 #3 #4 #5 #6 +
16409     \exp_after:wN \_\_fp\_round:NNN
16410     \exp_after:wN #9
16411     \exp_after:wN #6
16412     \int_value:w \_\_fp\_round\_digit:Nw #7 #8 ;
16413     \exp_after:wN ;
16414 }
```

(End definition for `\_\_fp\_div\_significand\_large\_o:wwwNNNNwN`.)

## 29.4 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.

```
16415 \cs_new:Npn \_\_fp_sqrt_o:w #1 \s_fp \_\_fp_chk:w #2#3#4; @
16416 {
16417     \if_meaning:w 0 #2 \_\_fp_case_return_same_o:w \fi:
16418     \if_meaning:w 2 #3
16419         \_\_fp_case_use:nw { \_\_fp_invalid_operation_o:nw { sqrt } }
16420     \fi:
16421     \if_meaning:w 1 #2 \else: \_\_fp_case_return_same_o:w \fi:
16422     \_\_fp_sqrt_npos_o:w
16423     \s_fp \_\_fp_chk:w #2 #3 #4;
16424 }
```

(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  $17782794 \simeq 10^{7.25}$ .

```
16425 \cs_new:Npn \_\_fp_sqrt_npos_o:w \s_fp \_\_fp_chk:w 1 0 #1#2#3#4#5;
16426 {
16427     \exp_after:wN \_\_fp_sanitize:Nw
16428     \exp_after:wN 0
16429     \int_value:w \_\_fp_int_eval:w
16430     \if_int_odd:w #1 \exp_stop_f:
16431         \exp_after:wN \_\_fp_sqrt_npos_auxi_o:wwnnN
```

```

16432      \fi:
16433      #1 / 2
16434      \_\_fp_sqrt_Newton_o:wwn 56234133; 0; {\#2\#3} {\#4\#5} 0
16435    }
16436 \cs_new:Npn \_\_fp_sqrt_npos_auxi_o:wwnnN #1 / 2 #2; 0; #3#4#5
16437  {
16438  ( #1 + 1 ) / 2
16439  \_\_fp_pack_eight:wNNNNNNNN
16440  \_\_fp_sqrt_npos_auxii_o:wNNNNNNNN
16441  ;
16442  0 #3 #4
16443  }
16444 \cs_new:Npn \_\_fp_sqrt_npos_auxii_o:wNNNNNNNN #1; #2#3#4#5#6#7#8#9
16445  { \_\_fp_sqrt_Newton_o:wwn 17782794; 0; {\#1} {\#2\#3#4#5#6#7#8#9} }

(End definition for \_\_fp_sqrt_npos_o:w, \_\_fp_sqrt_npos_auxi_o:wwnnN, and \_\_fp_sqrt_npos-
auxii_o:wNNNNNNNN.)

```

\\_\\_fp\_sqrt\_Newton\_o:wwn 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:wwn` 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  $[10^7, 10^8]$ .

```

16446 \cs_new:Npn \_fp_sqrt_Newton_o:wwn #1; #2; #3
16447 {
16448     \if_int_compare:w #1 = #2 \exp_stop_f:
16449         \exp_after:wN \_fp_sqrt_auxi_o:NNNNwnnN
16450         \int_value:w \_fp_int_eval:w 9999 9999 +
16451             \exp_after:wN \_fp_use_none_until_s:w
16452         \fi:
16453         \exp_after:wN \_fp_sqrt_Newton_o:wwn
16454         \int_value:w \_fp_int_eval:w (#1 + #3 * 1 0000 0000 / #1) / 2 ;
16455         #1; {#3}
16456     }

```

(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,  $y < 1$ .

```

16457 \cs_new:Npn \_fp_sqrt_auxi_o:NNNNwnnN 1 #1#2#3#4#5;
16458 {
16459     \_fp_sqrt_auxii_o:NnnnnnnnnN
16460         \_fp_sqrt_auxiii_o:wnnnnnnnn
16461         {#1#2#3#4} {#5} {2499} {9988} {7500}
16462 }

```

(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)

$$10^{4j}z = \left[ \left( \lfloor 10^{4j}(a - y^2) \rfloor - 257 \right) \cdot (0.5 \cdot 10^8) \middle/ \lfloor 10^8 y + 1 \rfloor \right].$$

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 TeX'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.

```

16463 \cs_new:Npn \__fp_sqrt_auxii_o:NnnnnnnN #1 #2#3#4#5#6 #7#8#9
16464 {
16465   \exp_after:wN #1
16466   \int_value:w \__fp_int_eval:w \c__fp_big_leading_shift_int
16467     + #7 - #2 * #2
16468   \exp_after:wN \__fp_pack_big:NNNNNNw
16469   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
16470     - 2 * #2 * #3
16471   \exp_after:wN \__fp_pack_big:NNNNNNw
16472   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
16473     + #8 - #3 * #3 - 2 * #2 * #4
16474   \exp_after:wN \__fp_pack_big:NNNNNNw
16475   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
16476     - 2 * #3 * #4 - 2 * #2 * #5
16477   \exp_after:wN \__fp_pack_big:NNNNNNw
16478   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
16479     + #9 000 0000 - #4 * #4 - 2 * #3 * #5 - 2 * #2 * #6
16480   \exp_after:wN \__fp_pack_big:NNNNNNw
16481   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
16482     - 2 * #4 * #5 - 2 * #3 * #6
16483   \exp_after:wN \__fp_pack_big:NNNNNNw
16484   \int_value:w \__fp_int_eval:w \c__fp_big_middle_shift_int
16485     - #5 * #5 - 2 * #4 * #6
16486   \exp_after:wN \__fp_pack_big:NNNNNNw
16487   \int_value:w \__fp_int_eval:w
16488     \c__fp_big_middle_shift_int
16489     - 2 * #5 * #6
16490   \exp_after:wN \__fp_pack_big:NNNNNNw
16491   \int_value:w \__fp_int_eval:w
16492     \c__fp_big_trailing_shift_int
16493     - #6 * #6 ;
16494 %
16495 - 257 ) * 5000 0000 / (#2#3 + 1) + 10 0000 0000 ;
16496 {#2}{#3}{#4}{#5}{#6} {#7}{#8}#9
16497 }

```

(End definition for `\__fp_sqrt_auxii_o:NnnnnnnN`.)

```

\fp_sqrt_auxiii_o:wnnnnnnn
\fp_sqrt_auxiv_o:NNNNNw
\fp_sqrt_auxv_o:NNNNNw
\fp_sqrt_auxvi_o:NNNNNw
\fp_sqrt_auxvii_o:NNNNNw

```

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  $\lfloor 10^{4j}(a - y^2) \rfloor$  in an integer expression. The closing parenthesis is provided by the caller `\fp_sqrt_auxiii_o:NnnnnnnnN`, 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:NnnnnnnnN`. 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

$$10^{4j}z \geq \frac{(20000 - 257)(0.5 \cdot 10^8)}{\lfloor 10^8y + 1 \rfloor} \geq (20000 - 257) \cdot 0.5 > 0.$$

```

16498 \cs_new:Npn \fp_sqrt_auxiii_o:wnnnnnnnn
16499   #1; #2#3#4#5#6#7#8#9
16500   {
16501     \if_int_compare:w #1 > 1 \exp_stop_f:
16502       \exp_after:wN \fp_sqrt_auxiv_o:NNNNNw
16503       \int_value:w \fp_int_eval:w (#1#2 %)
16504     \else:
16505       \if_int_compare:w #1#2 > 1 \exp_stop_f:
16506         \exp_after:wN \fp_sqrt_auxv_o:NNNNNw
16507         \int_value:w \fp_int_eval:w (#1#2#3 %)
16508       \else:
16509         \if_int_compare:w #1#2#3 > 1 \exp_stop_f:
16510           \exp_after:wN \fp_sqrt_auxvi_o:NNNNNw
16511           \int_value:w \fp_int_eval:w (#1#2#3#4 %)
16512         \else:
16513           \exp_after:wN \fp_sqrt_auxvii_o:NNNNNw
16514           \int_value:w \fp_int_eval:w (#1#2#3#4#5 %)
16515         \fi:
16516       \fi:
16517     \fi:
16518   }
16519 \cs_new:Npn \fp_sqrt_auxiv_o:NNNNNw 1#1#2#3#4#5#6;
16520   { \fp_sqrt_auxviii_o:nnnnnnnn {#1#2#3#4#5#6} {00000000} }
16521 \cs_new:Npn \fp_sqrt_auxv_o:NNNNNw 1#1#2#3#4#5#6;
16522   { \fp_sqrt_auxviii_o:nnnnnnnn {000#1#2#3#4#5} {#60000} }
16523 \cs_new:Npn \fp_sqrt_auxvi_o:NNNNNw 1#1#2#3#4#5#6;
16524   { \fp_sqrt_auxviii_o:nnnnnnnn {0000000#1} {#2#3#4#5#6} }
16525 \cs_new:Npn \fp_sqrt_auxvii_o:NNNNNw 1#1#2#3#4#5#6;
16526   {
16527     \if_int_compare:w #1#2 = 0 \exp_stop_f:
16528       \exp_after:wN \fp_sqrt_auxxx_o:Nnnnnnnn
16529     \fi:

```

```

16530     \__fp_sqrt_auxviii_o:nnnnnnnn {00000000} {000#1#2#3#4#5}
16531 }

```

(End definition for  $\_\_fp\_sqrt\_auxviii\_o:wnnnnnnn$  and others.)

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  $y'^2 = (y + z)^2$ .

```

16532 \cs_new:Npn \__fp_sqrt_auxviii_o:nnnnnnnn #1#2 #3#4#5#6#7
16533 {
16534     \exp_after:wN \__fp_sqrt_auxix_o:wnwnw
16535     \int_value:w \__fp_int_eval:w #3
16536     \exp_after:wN \__fp_basics_pack_low>NNNNW
16537     \int_value:w \__fp_int_eval:w #1 + 1#4#5
16538     \exp_after:wN \__fp_basics_pack_low>NNNNW
16539     \int_value:w \__fp_int_eval:w #2 + 1#6#7 ;
16540 }
16541 \cs_new:Npn \__fp_sqrt_auxix_o:wnwnw #1; #2#3; #4#5;
16542 {
16543     \__fp_sqrt_auxii_o:NnnnnnnnN
16544     \__fp_sqrt_auxiii_o:wnnnnnnnn {#1}{#2}{#3}{#4}{#5}
16545 }

```

(End definition for  $\_\_fp\_sqrt\_auxviii\_o:nnnnnnn$  and  $\_\_fp\_sqrt\_auxix\_o:wnwnw$ .)

$\_\_fp\_sqrt\_auxx\_o:Nnnnnnnn$  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 **auxxxi** 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  $a \leq 1 - 10^{-16}$ .

```

16546 \cs_new:Npn \__fp_sqrt_auxx_o:Nnnnnnnn #1#2#3 #4#5#6#7#8
16547 {
16548     \exp_after:wN \__fp_sqrt_auxxi_o:wnnnN
16549     \int_value:w \__fp_int_eval:w
16550     (#8 + 2499) / 5000 * 5000 ;
16551     {#4} {#5} {#6} {#7} ;
16552 }
16553 \cs_new:Npn \__fp_sqrt_auxxi_o:wnnnN #1; #2; #3#4#5
16554 {
16555     \__fp_sqrt_auxii_o:NnnnnnnnN

```

```

16556     \_fp_sqrt_auxxii_o:nnnnnnnnw
16557     #2 {#1}
16558     {#3} { #4 + 1 } #5
16559   }

```

(End definition for `\_fp_sqrt_auxx_o:Nnnnnnnn` and `\_fp_sqrt_auxxi_o:wwnnN`.)

`\_fp_sqrt_auxxii_o:nnnnnnnnw`  
`\_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.

```

16560 \cs_new:Npn \_fp_sqrt_auxxii_o:nnnnnnnnw 0; #1#2#3#4#5#6#7#8 #9;
16561   {
16562     \if_int_compare:w #1#2 > 0 \exp_stop_f:
16563     \if_int_compare:w #1#2 = 1 \exp_stop_f:
16564       \if_int_compare:w #3#4 = 0 \exp_stop_f:
16565         \if_int_compare:w #5#6 = 0 \exp_stop_f:
16566           \if_int_compare:w #7#8 = 0 \exp_stop_f:
16567             \_fp_sqrt_auxxiii_o:w
16568           \fi:
16569         \fi:
16570       \fi:
16571     \fi:
16572     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
16573     \int_value:w 9998
16574   \else:
16575     \exp_after:wN \_fp_sqrt_auxxiv_o:wnnnnnnnN
16576     \int_value:w 10000
16577   \fi:
16578   ;
16579 }
16580 \cs_new:Npn \_fp_sqrt_auxxiii_o:w \fi: \fi: \fi: \fi: \fi: #1 \fi: ;
16581   {
16582     \fi: \fi: \fi: \fi: \fi:
16583     \_fp_sqrt_auxxiv_o:wnnnnnnnN 9999 ;
16584   }

```

(End definition for `\_fp_sqrt_auxxii_o:nnnnnnnnw` 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.

```

16585 \cs_new:Npn \__fp_sqrt_auxxiv_o:wnnnnnnN #1; #2#3#4#5#6 #7#8#9
16586 {
16587     \exp_after:wN \__fp_basics_pack_high:NNNNNw
16588     \int_value:w \__fp_int_eval:w 1 0000 0000 + #2#3
16589     \exp_after:wN \__fp_basics_pack_low:NNNNNw
16590     \int_value:w \__fp_int_eval:w 1 0000 0000
16591     + #4#5
16592     \if_int_compare:w #6 > #1 \exp_stop_f: + 1 \fi:
16593     + \exp_after:wN \__fp_round:NNN
16594     \exp_after:wN 0
16595     \exp_after:wN 0
16596     \int_value:w
16597         \exp_after:wN \use_i:nn
16598         \exp_after:wN \__fp_round_digit:Nw
16599         \int_value:w \__fp_int_eval:w #6 + 19999 - #1 ;
16600     \exp_after:wN ;
16601 }

```

(End definition for `\__fp_sqrt_auxxiv_o:wnnnnnnN`.)

## 29.5 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
16602 \cs_new:Npn \__fp_logb_o:w ? \s__fp \__fp_chk:w #1#2; @
16603 {
16604     \if_case:w #1 \exp_stop_f:
16605         \__fp_case_use:nw
16606             { \__fp_division_by_zero_o:Nnw \c_minus_inf_fp { logb } }
16607     \or: \exp_after:wN \__fp_logb_aux_o:w
16608     \or: \__fp_case_return_o:Nw \c_inf_fp
16609     \else: \__fp_case_return_same_o:w
16610     \fi:
16611     \s__fp \__fp_chk:w #1 #2;
16612 }
16613 \cs_new:Npn \__fp_logb_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 #4 ;
16614 {
16615     \exp_after:wN \__fp_parse:n \exp_after:wN
16616         { \int_value:w \int_eval:w #3 - 1 \exp_after:wN }
16617 }

```

(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
16618 \cs_new:Npn \__fp_sign_o:w ? \s__fp \__fp_chk:w #1#2; @
16619 {
16620     \if_case:w #1 \exp_stop_f:
16621         \__fp_case_return_same_o:w
16622     \or: \exp_after:wN \__fp_sign_aux_o:w
16623     \or: \exp_after:wN \__fp_sign_aux_o:w
16624     \else: \__fp_case_return_same_o:w
16625     \fi:
16626     \s__fp \__fp_chk:w #1 #2;
16627 }
16628 \cs_new:Npn \__fp_sign_aux_o:w \s__fp \__fp_chk:w #1 #2 #3 ;
16629     { \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:w`

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`.

```

16630 \cs_new:Npn \_fp_set_sign_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
16631   {
16632     \exp_after:wN \_fp_exp_after_o:w
16633     \exp_after:wN \s_fp
16634     \exp_after:wN \_fp_chk:w
16635     \exp_after:wN #2
16636     \int_value:w
16637     \if_case:w #3 \exp_stop_f: #1 \or: 1 \or: 0 \fi: \exp_stop_f:
16638     #4;
16639   }

```

(End definition for `\_fp_set_sign_o:w`.)

## 29.6 Operations on tuples

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.

```

16640 \cs_new:Npn \_fp_tuple_set_sign_o:w #1
16641   {
16642     \if_meaning:w 2 #1
16643       \exp_after:wN \_fp_tuple_set_sign_aux_o:Nnw
16644     \fi:
16645     \_fp_invalid_operation_o:nw { abs }
16646   }
16647 \cs_new:Npn \_fp_tuple_set_sign_aux_o:Nnw #1#2#3 @
16648   { \_fp_tuple_map_o:nw \_fp_tuple_set_sign_aux_o:w #3 }
16649 \cs_new:Npn \_fp_tuple_set_sign_aux_o:w #1#2 ;
16650   {
16651     \_fp_change_func_type:NNN #1 \_fp_set_sign_o:w
16652     \_fp_parse_apply_unary_error:NNw
16653     2 #1 #2 ; @
16654   }

```

(End definition for `\_fp_tuple_set_sign_o:w`, `\_fp_tuple_set_sign_aux_o:Nnw`, and `\_fp_tuple_set_sign_aux_o:w`.)

`\_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.

```

16655 \cs_new:cpn { \_fp_*_tuple_o:ww } #1 ;
16656   { \_fp_tuple_map_o:nw { \_fp_binary_type_o:Nww * #1 ; } }
16657 \cs_new:cpn { \_fp_tuple_*_o:ww } #1 ; #2 ;
16658   { \_fp_tuple_map_o:nw { \_fp_binary_rev_type_o:Nww * #2 ; } #1 ; }
16659 \cs_new:cpn { \_fp_tuple/_o:ww } #1 ; #2 ;
16660   { \_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  $(\text{nan}, 4)$ .

```

16661 \cs_set_protected:Npn \_fp_tmp:w #1
16662 {
16663   \cs_new:cpn { \_fp_tuple_#1_tuple_o:ww }
16664   \s_fp_tuple \_fp_tuple_chk:w ##1 ;
16665   \s_fp_tuple \_fp_tuple_chk:w ##2 ;
16666 {
16667   \int_compare:nNnTF
16668     { \_fp_array_count:n {##1} } = { \_fp_array_count:n {##2} }
16669     { \_fp_tuple_mapthread_o:nww { \_fp_binary_type_o:Nww #1 } }
16670     { \_fp_invalid_operation_o:nww #1 }
16671     \s_fp_tuple \_fp_tuple_chk:w {##1} ;
16672     \s_fp_tuple \_fp_tuple_chk:w {##2} ;
16673 }
16674 }
16675 \_fp_tmp:w +
16676 \_fp_tmp:w -

```

(End definition for `\_fp_tuple_+tuple_o:ww` and `\_fp_tuple_-tuple_o:ww`.)

16677 ⟨/initex | package⟩

## 30 I3fp-extended implementation

16678 ⟨\*initex | package⟩  
16679 ⟨@=fp⟩

### 30.1 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

{⟨ $a_1$ ⟩} {⟨ $a_2$ ⟩} {⟨ $a_3$ ⟩} {⟨ $a_4$ ⟩} {⟨ $a_5$ ⟩} {⟨ $a_6$ ⟩} ;

where each ⟨ $a_i$ ⟩ is exactly 4 digits (ranging from 0000 to 9999), except ⟨ $a_1$ ⟩, 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:wn ⟨X1⟩ ; ⟨X2⟩ ;
\_\_fp\_fixed\_mul:wn ⟨X3⟩ ;
\_\_fp\_fixed\_add:wn ⟨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.

## 30.2 Helpers for numbers with extended precision

`\c\_fp\_one\_fixed_t1` The fixed-point number 1, used in `\3fp-expo`.

```
16680 \tl_const:Nn \c\_fp\_one\_fixed_t1
16681 { {10000} {0000} {0000} {0000} {0000} {0000} ; }
```

(End definition for `\c\_fp\_one\_fixed_t1`.)

`\_\_fp\_fixed\_continue:wn` This function simply calls the next function.

```
16682 \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 *⟨a⟩*, by changing  $a_1$  to  $10000 + a_1$ , then calls the *⟨continuation⟩*. This requires  $a_1 + 10000 < 2^{31}$ .

```
16683 \cs_new:Npn \_\_fp\_fixed\_add\_one:wN #1#2; #3
16684 {
16685     \exp_after:wN #3 \exp_after:wN
16686     { \int_value:w \_\_fp\_int\_eval:w \c\_fp\_myriad\_int + #1 } #2 ;
16687 }
```

(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  $[0, 5 \cdot 10^8 - 1]$ .

```
16688 \cs_new:Npn \_\_fp\_fixed\_div\_myriad:wn #1#2#3#4#5#6;
16689 {
16690     \exp_after:wN \_\_fp\_fixed\_mul\_after:wwn
16691     \int_value:w \_\_fp\_int\_eval:w \c\_fp\_leading\_shift\_int
16692         \exp_after:wN \_\_fp\_pack:NNNNw
16693         \int_value:w \_\_fp\_int\_eval:w \c\_fp\_trailing\_shift\_int
16694             + #1 ; {#2}{#3}{#4}{#5};
16695 }
```

(End definition for `\_fp_fixed_div_myriad:wn`.)

`\_fp_fixed_mul_after:wn`

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.

```
16696 \cs_new:Npn \_fp_fixed_mul_after:wn #1; #2; #3 { #3 {#1} #2; }
```

(End definition for `\_fp_fixed_mul_after:wn`.)

### 30.3 Multiplying a fixed point number by a short one

`\_fp_fixed_mul_short:wn`

```
\_fp_fixed_mul_short:wn
{⟨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  $\langle\text{continuation}\rangle \{c_1\} \dots \{c_6\}$ ; in the input stream, where each of the  $\langle c_i \rangle$  are blocks of 4 digits, except  $\langle c_1 \rangle$ , which is any TeX integer. Note that indices for  $\langle b \rangle$  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:wn` would).

```
16697 \cs_new:Npn \_fp_fixed_mul_short:wn #1#2#3#4#5#6; #7#8#9;
16698 {
16699   \exp_after:wN \_fp_fixed_mul_after:wn
16700   \int_value:w \_fp_int_eval:w \c_fp_leading_shift_int
16701     + #1*#7
16702   \exp_after:wN \_fp_pack:NNNNw
16703   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
16704     + #1*#8 + #2*#7
16705   \exp_after:wN \_fp_pack:NNNNNw
16706   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
16707     + #1*#9 + #2*#8 + #3*#7
16708   \exp_after:wN \_fp_pack:NNNNNw
16709   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
16710     + #2*#9 + #3*#8 + #4*#7
16711   \exp_after:wN \_fp_pack:NNNNNw
16712   \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
16713     + #3*#9 + #4*#8 + #5*#7
16714   \exp_after:wN \_fp_pack:NNNNNw
16715   \int_value:w \_fp_int_eval:w \c_fp_trailing_shift_int
16716     + #4*#9 + #5*#8 + #6*#7
16717     + ( #5*#9 + #6*#8 + #6*#9 / \c_fp_myriad_int )
16718   / \c_fp_myriad_int ; }
```

(End definition for `\_fp_fixed_mul_short:wn`.)

### 30.4 Dividing a fixed point number by a small integer

`\_fp_fixed_div_int:wn`

`\_fp_fixed_div_int:wnN`

`\_fp_fixed_div_int_auxi:wn`

`\_fp_fixed_div_int_auxi: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  $\langle a \rangle$  by the (small) integer  $0 < \langle n \rangle < 10^4$  and feeds the result to the  $\langle\text{continuation}\rangle$ . There is no bound on  $a_1$ .

The arguments of the `i` auxiliary are 1: one of the  $a_i$ , 2:  $n$ , 3: the `ii` or the `iii` auxiliary. It computes a (somewhat tight) lower bound  $Q_i$  for the ratio  $a_i/n$ .

The **ii** auxiliary receives  $Q_i$ ,  $n$ , and  $a_i$  as arguments. It adds  $Q_i$  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  $a_i - n \cdot Q_i$ , placing the result in front of the 4 digits of  $a_{i+1}$ . The resulting  $a'_{i+1} = 10^4(a_i - n \cdot Q_i) + a_{i+1}$  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.

```
16720 \cs_new:Npn \_\_fp\_fixed\_div\_int:wnN #1#2#3#4#5#6 ; #7 ; #8
16721 {
16722   \exp_after:wN \_\_fp\_fixed\_div\_int\_after:Nw
16723   \exp_after:wN #8
16724   \int_value:w \_\_fp\_int\_eval:w - 1
16725   \_\_fp\_fixed\_div\_int:wnN
16726   #1; {#7} \_\_fp\_fixed\_div\_int\_auxi:wnn
16727   #2; {#7} \_\_fp\_fixed\_div\_int\_auxi:wnn
16728   #3; {#7} \_\_fp\_fixed\_div\_int\_auxi:wnn
16729   #4; {#7} \_\_fp\_fixed\_div\_int\_auxi:wnn
16730   #5; {#7} \_\_fp\_fixed\_div\_int\_auxi:wnn
16731   #6; {#7} \_\_fp\_fixed\_div\_int\_auxi:wnn ;
16732 }
16733 \cs_new:Npn \_\_fp\_fixed\_div\_int:wnN #1; #2 #3
16734 {
16735   \exp_after:wN #3
16736   \int_value:w \_\_fp\_int\_eval:w #1 / #2 - 1 ;
16737   {#2}
16738   {#1}
16739 }
16740 \cs_new:Npn \_\_fp\_fixed\_div\_int\_auxi:wnn #1; #2 #3
16741 {
16742   + #1
16743   \exp_after:wN \_\_fp\_fixed\_div\_int\_pack:Nw
16744   \int_value:w \_\_fp\_int\_eval:w 9999
16745   \exp_after:wN \_\_fp\_fixed\_div\_int:wnN
16746   \int_value:w \_\_fp\_int\_eval:w #3 - #1*#2 \_\_fp\_int\_eval_end:
16747 }
16748 \cs_new:Npn \_\_fp\_fixed\_div\_int\_auxii:wnn #1; #2 #3 { + #1 + 2 ; }
16749 \cs_new:Npn \_\_fp\_fixed\_div\_int\_pack:Nw #1 #2; { + #1; {#2} }
16750 \cs_new:Npn \_\_fp\_fixed\_div\_int\_after:Nw #1 #2; { #1 {#2} }
```

(End definition for `\_fp_fixed_div_int:wwN` and others.)

### 30.5 Adding and subtracting fixed points

```
\_fp_fixed_add:wwn
\_fp_fixed_sub:wwn
\_fp_fixed_add:Nnnnnwnn
\_fp_fixed_add:nnNnnwn
\_fp_fixed_add_pack:NNNNNwn
\_fp_fixed_add_after:NNNNNwn
```

`\_fp_fixed_add:wwn`  $\langle a \rangle ; \langle b \rangle ; \langle continuation \rangle$

Computes  $a + b$  (resp.  $a - b$ ) and feeds the result to the  $\langle continuation \rangle$ . 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  $\langle continuation \rangle$  as arguments. After going down through the various level, we go back up, packing digits and bringing the  $\langle continuation \rangle$  (#8, then #7) from the end of the argument list to its start.

```
16751 \cs_new:Npn \_fp_fixed_add:wwn { \_fp_fixed_add:Nnnnnwnn + }
16752 \cs_new:Npn \_fp_fixed_sub:wwn { \_fp_fixed_add:Nnnnnwnn - }
16753 \cs_new:Npn \_fp_fixed_add:Nnnnnwnn #1 #2#3#4#5 #6; #7#8
16754 {
16755     \exp_after:wN \_fp_fixed_add_after:NNNNNwn
16756     \int_value:w \_fp_int_eval:w 9 9999 9998 + #2#3 #1 #7#8
16757         \exp_after:wN \_fp_fixed_add_pack:NNNNNwn
16758         \int_value:w \_fp_int_eval:w 1 9999 9998 + #4#5
16759             \_fp_fixed_add:nnNnnwnn #6 #1
16760     }
16761 \cs_new:Npn \_fp_fixed_add:nnNnnwnn #1#2 #3 #4#5 #6#7 ; #8
16762 {
16763     #3 #4#5
16764     \exp_after:wN \_fp_fixed_add_pack:NNNNNwn
16765     \int_value:w \_fp_int_eval:w 2 0000 0000 #3 #6#7 + #1#2 ; {#8} ;
16766 }
16767 \cs_new:Npn \_fp_fixed_add_pack:NNNNNwnn #1 #2#3#4#5 #6; #7
16768 { + #1 ; {#7} {#2#3#4#5} {#6} }
16769 \cs_new:Npn \_fp_fixed_add_after:NNNNNwnn 1 #1 #2#3#4#5 #6; #7
16770 { #7 {#1#2#3#4#5} {#6} }
```

(End definition for `\_fp_fixed_add:wwn` and others.)

### 30.6 Multiplying fixed points

```
\_fp_fixed_mul:wwn
\_fp_fixed_mul:nnnnnnnw
```

`\_fp_fixed_mul:wwn`  $\langle a \rangle ; \langle b \rangle ; \langle continuation \rangle$

Computes  $a \times b$  and feeds the result to  $\langle continuation \rangle$ . 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`.

```

16771 \cs_new:Npn \_\_fp_fixed_mul:wwn #1#2#3#4 #5; #6#7#8#9
16772 {
16773   \exp_after:wN \_\_fp_fixed_mul_after:wwn
16774   \int_value:w \_\_fp_int_eval:w \c__fp_leading_shift_int
16775   \exp_after:wN \_\_fp_pack:NNNNNw
16776   \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
16777   + #1*#6
16778   \exp_after:wN \_\_fp_pack:NNNNNw
16779   \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
16780   + #1*#7 + #2*#6
16781   \exp_after:wN \_\_fp_pack:NNNNNw
16782   \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
16783   + #1*#8 + #2*#7 + #3*#6
16784   \exp_after:wN \_\_fp_pack:NNNNNw
16785   \int_value:w \_\_fp_int_eval:w \c__fp_middle_shift_int
16786   + #1*#9 + #2*#8 + #3*#7 + #4*#6
16787   \exp_after:wN \_\_fp_pack:NNNNNw
16788   \int_value:w \_\_fp_int_eval:w \c__fp_trailing_shift_int
16789   + #2*#9 + #3*#8 + #4*#7
16790   + ( #3*#9 + #4*#8
16791   + \_\_fp_fixed_mul:nnnnnnnw #5 {#6}{#7} {#1}{#2}
16792   )
16793 \cs_new:Npn \_\_fp_fixed_mul:nnnnnnnw #1#2 #3#4 #5#6 #7#8 ;
16794 {
16795   #1*#4 + #2*#3 + #5*#8 + #6*#7 ) / \c__fp_myriad_int
16796   + #1*#3 + #5*#7 ; ;
16797 }

```

(End definition for `\_\_fp_fixed_mul:wwn` and `\_\_fp_fixed_mul:nnnnnnnw`.)

## 30.7 Combining product and sum of fixed points

```
\_\_fp\_fixed\_mul\_add:wwn
\_\_fp\_fixed\_mul\_sub\_back:wwn
\_\_fp\_fixed\_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  $\langle continuation \rangle$ . 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  $+ c_5 c_6 ; \{\langle continuation \rangle\} ;$ . The  $+ c_5 c_6$  piece, which is omitted for `\_\_fp\_fixed\_one\_minus\_mul:wwn`, is taken in the integer expression for the  $10^{-24}$  level.

```
16798 \cs_new:Npn \_\_fp\_fixed\_mul\_add:wwn #1; #2; #3#4#5#6#7#8;
16799 {
16800   \exp_after:wN \_\_fp\_fixed\_mul\_after:wwn
16801   \int_value:w \_\_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
16802     \exp_after:wN \_\_fp\_pack\_big:NNNNNNw
16803     \int_value:w \_\_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
16804       \_\_fp\_fixed\_mul\_add:Nwnnnnnn +
16805         + #5 #6 ; #2 ; #1 ; #2 ; +
16806         + #7 #8 ; ;
16807   }
16808 \cs_new:Npn \_\_fp\_fixed\_mul\_sub\_back:wwn #1; #2; #3#4#5#6#7#8;
16809 {
16810   \exp_after:wN \_\_fp\_fixed\_mul\_after:wwn
16811   \int_value:w \_\_fp\_int\_eval:w \c\_fp\_big\_leading\_shift\_int
16812     \exp_after:wN \_\_fp\_pack\_big:NNNNNNw
16813     \int_value:w \_\_fp\_int\_eval:w \c\_fp\_big\_middle\_shift\_int + #3 #4
16814       \_\_fp\_fixed\_mul\_add:Nwnnnnnn -
16815         + #5 #6 ; #2 ; #1 ; #2 ; -
16816         + #7 #8 ; ;
16817   }
16818 \cs_new:Npn \_\_fp\_fixed\_one\_minus\_mul:wwn #1; #2;
16819 {
```

```

16820      \exp_after:wN \_fp_fixed_mul_after:wwn
16821      \int_value:w \_fp_int_eval:w \c_fp_big_leading_shift_int
16822      \exp_after:wN \_fp_pack_big:NNNNNNw
16823      \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int +
16824      1 0000 0000
16825      \_fp_fixed_mul_add:Nwnnnwnnn -
16826      ; #2 ; #1 ; #2 ; -
16827      ; ;
16828  }

```

(End definition for `\_fp_fixed_mul_add:wwn`, `\_fp_fixed_mul_sub_back:wwn`, and `\_fp_fixed_minus_mul:wwn`)

```

\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.

```

16829 \cs_new:Npn \fp_fixed_mul_add:Nwnnnwnnn #1 #2; #3#4#5#6; #7#8#9
16830  {
16831  #1 #7*#3
16832  \exp_after:wN \_fp_pack_big:NNNNNNw
16833  \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int
16834  #1 #7*#4 #1 #8*#3
16835  \exp_after:wN \_fp_pack_big:NNNNNNw
16836  \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int
16837  #1 #7*#5 #1 #8*#4 #1 #9*#3 #2
16838  \exp_after:wN \_fp_pack_big:NNNNNNw
16839  \int_value:w \_fp_int_eval:w \c_fp_big_middle_shift_int
16840  #1 \_fp_fixed_mul_add:nnnnwnnnn {#7}{#8}{#9}
16841  }

```

(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  $\langle b \rangle$ .

```

16842 \cs_new:Npn \fp_fixed_mul_add:nnnnwnnnn #1#2#3#4#5; #6#7#8#9
16843  {

```

```

16844      ( #1##9 + #2##8 + #3##7 + #4##6 )
16845      \exp_after:wN \_fp_pack_big:NNNNNNw
16846      \int_value:w \_fp_int_eval:w \c_fp_big_trailing_shift_int
16847          \_fp_fixed_mul_add:nnnnwnnwN
16848              { #6 + #4##7 + #3##8 + #2##9 + #1 }
16849              { #7 + #4##8 + #3##9 + #2 }
16850              {#1} #5;
16851              {#6}
16852      }

```

(End definition for `\_fp_fixed_mul_add:nnnnwnnnn.`)

```

\_fp_fixed_mul_add:nnnnwnnwN {\partial1} {\partial2}
    {\a1} {\a5} {\a6} ; {\b1} {\b5} {\b6} ;
    \op + \c5 \c6 ;

```

Complete the  $\langle \partial_1 \rangle$  and  $\langle \partial_2 \rangle$  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.

```

16853 \cs_new:Npn \_fp_fixed_mul_add:nnnnwnnwN #1#2 #3#4#5; #6#7#8; #9
16854 {
16855     #9 (#4* #1 **7)
16856     #9 (#5##6+#4* #2 **7+#3##8) / \c_fp_myriad_int
16857 }

```

(End definition for `\_fp_fixed_mul_add:nnnnwnnwN.`)

### 30.8 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  $0.\langle digits \rangle \cdot 10^{\langle exponent \rangle}$ . This convention differs from floating points.

`\_fp_ep_to_fixed:wwn` 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.

```

16858 \cs_new:Npn \_fp_ep_to_fixed:wwn #1,#2
16859 {
16860     \exp_after:wN \_fp_ep_to_fixed_auxi:www
16861     \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \exp_after:wN ;
16862     \exp:w \exp_end_continue_f:w
16863     \prg_replicate:nn { 4 - \int_max:nn {#1} { -32 } } { 0 } ;
16864 }
16865 \cs_new:Npn \_fp_ep_to_fixed_auxi:www 1#1; #2; #3#4#5#6#7;
16866 {
16867     \_fp_pack_eight:wNNNNNNNNN
16868     \_fp_pack_twice_four:wNNNNNNNNN

```

```

16869      \_fp_pack_twice_four:wNNNNNNNNN
16870      \_fp_pack_twice_four:wNNNNNNNNN
16871      \_fp_ep_to_fixed_auxii:nnnnnnnwn ;
16872      #2 #1#3#4#5#6#7 0000 !
16873  }
16874 \cs_new:Npn \_fp_ep_to_fixed_auxii:nnnnnnnwn #1#2#3#4#5#6#7; #8! #9
16875  { #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:nnnnnnnwn.)

```

\\_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).

```

16876 \cs_new:Npn \_fp_ep_to_ep:wwN #1,#2#3#4#5#6#7; #8
16877  {
16878      \exp_after:wN #8
16879      \int_value:w \_fp_int_eval:w #1 + 4
16880      \exp_after:wN \use_i:nn
16881      \exp_after:wN \_fp_ep_to_ep_loop:N
16882      \int_value:w \_fp_int_eval:w 1 0000 0000 + #2 \_fp_int_eval_end:
16883      #3#4#5#6#7 ; ;
16884  }
16885 \cs_new:Npn \_fp_ep_to_ep_loop:N #1
16886  {
16887      \if_meaning:w 0 #1
16888      - 1
16889      \else:
16890      \_fp_ep_to_ep_end:www #1
16891      \fi:
16892      \_fp_ep_to_ep_loop:N
16893  }
16894 \cs_new:Npn \_fp_ep_to_ep_end:www
16895  #1 \fi: \_fp_ep_to_ep_loop:N #2; #3!
16896  {
16897      \fi:
16898      \if_meaning:w ; #1
16899      - 2 * \c_fp_max_exponent_int
16900      \_fp_ep_to_ep_zero:ww
16901      \fi:
16902      \_fp_pack_twice_four:wNNNNNNNNN
16903      \_fp_pack_twice_four:wNNNNNNNNN
16904      \_fp_pack_twice_four:wNNNNNNNNN
16905      \_fp_use_i:ww , ;
16906      #1 #2 0000 0000 0000 0000 0000 ;
16907  }

```

```

16908 \cs_new:Npn \__fp_ep_to_ep_zero:ww \fi: #1; #2; #3;
16909   { \fi: , {1000}{0000}{0000}{0000}{0000} ; }

```

(End definition for `\__fp_ep_to_ep:wwN` and others.)

In `l3fp-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].

```

16910 \cs_new:Npn \__fp_ep_compare:wwww #1,#2#3#4#5#6#7;
16911   { \__fp_ep_compare_aux:www {#1}{#2}{#3}{#4}{#5}; #6#7; }
16912 \cs_new:Npn \__fp_ep_compare_aux:www #1;#2;#3,#4#5#6#7#8#9;
16913   {
16914     \if_case:w
16915       \__fp_compare_npos:nwnw #1; {#3}{#4}{#5}{#6}{#7}; \exp_stop_f:
16916         \if_int_compare:w #2 = #8#9 \exp_stop_f:
16917           0
16918         \else:
16919           \if_int_compare:w #2 < #8#9 - \fi: 1
16920           \fi:
16921         \or: 1
16922         \else: -1
16923         \fi:
16924   }

```

(End definition for `\__fp_ep_compare:wwww` and `\__fp_ep_compare_aux:wwww`.)

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].

```

16925 \cs_new:Npn \__fp_ep_mul:wwwnn #1,#2; #3,#4;
16926   {
16927     \__fp_ep_to_ep:wwN #3,#4;
16928     \__fp_fixed_continue:wn
16929     {
16930       \__fp_ep_to_ep:wwN #1,#2;
16931       \__fp_ep_mul_raw:wwwnN
16932     }
16933     \__fp_fixed_continue:wn
16934   }
16935 \cs_new:Npn \__fp_ep_mul_raw:wwwnN #1,#2; #3,#4; #5
16936   {
16937     \__fp_fixed_mul:wnN #2; #4;
16938     { \exp_after:wN #5 \int_value:w \__fp_int_eval:w #1 + #3 , }
16939   }

```

(End definition for `\__fp_ep_mul:wwwnn` and `\__fp_ep_mul_raw:wwwnN`.)

### 30.9 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  $\langle n_1 \rangle \dots \langle n_6 \rangle$  and the 6 blocks  $\langle d_1 \rangle \dots \langle d_6 \rangle$ , 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.

```

16940 \cs_new:Npn \__fp_ep_div:wwwn #1,#2; #3,#4;
16941 {
16942     \__fp_ep_to_ep:wwN #1,#2;
16943     \__fp_fixed_continue:wn
16944 {
16945     \__fp_ep_to_ep:wwN #3,#4;
16946     \__fp_ep_div_esti:wwwn
16947 }
16948 }
```

(End definition for `\__fp_ep_div:wwwn`.)

`\__fp_ep_div_esti:wwwn` 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:wnNNNNn`, which computes  $10^{-9}a / (1 - \epsilon)$ , that is,  $1 / (10 \langle d \rangle)$  and we finally multiply this by the numerator #8.

```

16949 \cs_new:Npn \__fp_ep_div_esti:wwwn #1,#2#3; #4,
16950 {
16951     \exp_after:wN \__fp_ep_div_estii:wwnnwwn
16952     \int_value:w \__fp_int_eval:w 10 0000 0000 / ( #2 + 1 )
16953     \exp_after:wN ;
16954     \int_value:w \__fp_int_eval:w #4 - #1 + 1 ,
16955     {#2} #3;
16956 }
```

```

16957 \cs_new:Npn \__fp_ep_div_estii:wwnnwwn #1; #2,#3#4#5; #6; #7
16958 {
16959     \exp_after:wN \__fp_ep_div_estii:NNNNNwwnn
16960     \int_value:w \__fp_int_eval:w 10 0000 0000 - 1750
16961     + #1 000 + (10 0000 0000 / #3 - #1) * (1000 - #4 / 10) ;
16962     {#3}{#4}#5; #6; { #7 #2, }
16963 }
16964 \cs_new:Npn \__fp_ep_div_estiii:NNNNNwwnn 1#1#2#3#4#5#6; #7;
16965 {
16966     \__fp_fixed_mul_short:wwn #7; {#1}{#2#3#4#5}{#6};
16967     \__fp_ep_div_epsi:wnNNNNNn {#1#2#3#4}#5#6
16968     \__fp_fixed_mul:wwn
16969 }

```

(End definition for `\__fp_ep_div_esti:wwnn`, `\__fp_ep_div_estii:wwnnwwn`, and `\__fp_ep_div-estiii:NNNNNwwnn`.)

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 `epsi` 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`.

```

16970 \cs_new:Npn \__fp_ep_div_epsi:wnNNNNNn #1#2#3#4#5#6;
16971 {
16972     \exp_after:wN \__fp_ep_div_epsii:wwnNNNNNn
16973     \int_value:w \__fp_int_eval:w 1 9998 - #2
16974     \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
16975     \int_value:w \__fp_int_eval:w 1 9999 9998 - #3#4
16976     \exp_after:wN \__fp_ep_div_eps_pack:NNNNNw
16977     \int_value:w \__fp_int_eval:w 2 0000 0000 - #5#6 ; ;
16978 }
16979 \cs_new:Npn \__fp_ep_div_eps_pack:NNNNNw #1#2#3#4#5#6;
16980 { + #1 ; {#2#3#4#5} {#6} }
16981 \cs_new:Npn \__fp_ep_div_epsii:wwnNNNNNn 1#1; #2; #3#4#5#6#7#8
16982 {
16983     \__fp_fixed_mul:wwn {0000}{#1}#2; {0000}{#1}#2;
16984     \__fp_fixed_add_one:wn
16985     \__fp_fixed_mul:wwn {10000} {#1} #2 ;
16986     {
16987         \__fp_fixed_mul_short:wwn {0000}{#1}#2; {#3}{#4#5#6#7}{#8000};
16988         \__fp_fixed_div_myriad:wn
16989         \__fp_fixed_mul:wwn
16990     }
16991     \__fp_fixed_add:wwn {#3}{#4#5#6#7}{#8000}{0000}{0000}{0000};
16992 }

```

(End definition for `\__fp_ep_div_epsi:wnNNNNNn`, `\__fp_ep_div_eps_pack:NNNNNw`, and `\__fp_ep-div_epsii:wwnNNNNNn`.)

### 30.10 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:wwnnwn
```

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.

```
16993 \cs_new:Npn \_\_fp\_ep\_isqrt:wwn #1,#2;
16994 {
16995     \_\_fp\_ep\_to\_ep:wwN #1,#2;
16996     \_\_fp\_ep\_isqrt\_auxi:wwn
16997 }
16998 \cs_new:Npn \_\_fp\_ep\_isqrt\_auxi:wwn #1,
16999 {
17000     \exp_after:wN \_\_fp\_ep\_isqrt\_auxii:wwnnwn
17001     \int_value:w \_\_fp\_int\_eval:w
17002     \int_if_odd:nTF {#1}
17003     { (1 - #1) / 2 , 535 , { 0 } { } }
17004     { 1 - #1 / 2 , 168 , { } { 0 } }
17005 }
17006 \cs_new:Npn \_\_fp\_ep\_isqrt\_auxii:wwnnwn #1, #2, #3#4 #5#6; #7
17007 {
17008     \_\_fp\_ep\_isqrt\_esti:wwnnwn #2, 0, #5, {#3} {#4}
17009     {#5} #6 ; { #7 #1 , }
17010 }
```

(End definition for `\_\_fp\_ep\_isqrt:wwn`, `\_\_fp\_ep\_isqrt\_aux:wwn`, and `\_\_fp\_ep\_isqrt\_auxii:wwnnwn`.)

```
\_\_fp\_ep\_isqrt\_esti:wwnnwn
\_\_fp\_ep\_isqrt\_estii:wwnnwn
\_\_fp\_ep\_isqrt\_estiii:NNNNNWWWN
```

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  $\text{\_fp\_ep\_isqrt\_epsi:wn}$ , which computes  $1/\sqrt{y}$ , and we finally multiply the result by  $r$ .

```

17011 \cs_new:Npn \_fp_ep_isqrt_esti:wwnnwn #1, #2, #3, #4
17012 {
17013     \if_int_compare:w #1 = #2 \exp_stop_f:
17014         \exp_after:wN \_fp_ep_isqrt_estii:wwnnwn
17015     \fi:
17016     \exp_after:wN \_fp_ep_isqrt_esti:wwnnwn
17017     \int_value:w \_fp_int_eval:w
17018         (#1 + 1 0050 0000 #4 / (#1 * #3)) / 2 ,
17019     #1, #3, {#4}
17020 }
17021 \cs_new:Npn \_fp_ep_isqrt_estii:wwnnwn #1, #2, #3, #4#5
17022 {
17023     \exp_after:wN \_fp_ep_isqrt_estiii:NNNNNwwnn
17024     \int_value:w \_fp_int_eval:w 1000 0000 + #2 * #2 #5 * 5
17025     \exp_after:wN , \int_value:w \_fp_int_eval:w 10000 + #2 ;
17026 }
17027 \cs_new:Npn \_fp_ep_isqrt_estiii:NNNNNwwnn 1#1#2#3#4#5#6, 1#7#8; #9;
17028 {
17029     \_fp_fixed_mul_short:wwn #9; {#1} {#2#3#4#5} {#600} ;
17030     \_fp_ep_isqrt_epsi:wn
17031     \_fp_fixed_mul_short:wwn {#7} {#80} {0000} ;
17032 }
```

(End definition for  $\text{\_fp\_ep\_isqrt\_esti:wwnnwn}$ ,  $\text{\_fp\_ep\_isqrt\_estii:wwnnwn}$ , and  $\text{\_fp\_ep\_isqrt\_estiii:NNNNNwwnn}$ .)

$\text{\_fp\_ep\_isqrt\_epsi:wn}$   $\text{\_fp\_ep\_isqrt\_epsii:wwN}$  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  $\text{epsii}$  auxiliary receives  $z$  as  $\#1$  and  $y$  as  $\#2$ .

```

17033 \cs_new:Npn \_fp_ep_isqrt_epsi:wn #1;
17034 {
17035     \_fp_fixed_sub:wwn {15000}{0000}{0000}{0000}{0000}; #1;
17036     \_fp_ep_isqrt_epsii:wwN #1;
17037     \_fp_ep_isqrt_epsii:wwN #1;
17038     \_fp_ep_isqrt_epsii:wwN #1;
17039 }
17040 \cs_new:Npn \_fp_ep_isqrt_epsii:wwN #1; #2;
17041 {
17042     \_fp_fixed_mul:wwn #1; #1;
17043     \_fp_fixed_mul_sub_back:wwnn #2;
17044     {15000}{0000}{0000}{0000}{0000};
17045     \_fp_fixed_mul:wwn #1;
17046 }
```

(End definition for  $\text{\_fp\_ep\_isqrt\_epsi:wn}$  and  $\text{\_fp\_ep\_isqrt\_epsii:wwN}$ .)

### 30.11 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.

```
17047 \cs_new:Npn \__fp_ep_to_float_o:wwN #1,
17048   { + \__fp_int_eval:w #1 \__fp_fixed_to_float_o:wwN }
17049 \cs_new:Npn \__fp_inv_to_float_o:wwN #1,#2;
17050   {
17051     \__fp_ep_div:wwwn 1,{1000}{0000}{0000}{0000}{0000}; #1,#2;
17052     \__fp_ep_to_float_o:wwN
17053 }
```

(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.

```
17054 \cs_new:Npn \__fp_fixed_inv_to_float_o:wn
17055   { \__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`.

```
17056 \cs_new:Npn \__fp_fixed_to_float_rad_o:wn #1;
17057   {
17058     \__fp_fixed_mul:wwn #1; {5729}{5779}{5130}{8232}{0876}{7981};
17059     { \__fp_ep_to_float_o:wwN 2, }
17060 }
```

(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>9</sup>

```
17061 \cs_new:Npn \__fp_fixed_to_float_o:Nw #1#2;
17062   { \__fp_fixed_to_float_o:wn #2; #1 }
17063 \cs_new:Npn \__fp_fixed_to_float_o:wn #1#2#3#4#5#6; #7
17064   { % for the 8-digit-at-the-start thing
17065     + \__fp_int_eval:w \c__fp_block_int
17066     \exp_after:wn \exp_after:wn
17067     \exp_after:wn \__fp_fixed_to_loop:N
17068     \exp_after:wn \use_none:n
17069     \int_value:w \__fp_int_eval:w
17070     1 0000 0000 + #1 \exp_after:wn \__fp_use_none_stop_f:n
17071     \int_value:w 1#2 \exp_after:wn \__fp_use_none_stop_f:n
17072     \int_value:w 1#3#4 \exp_after:wn \__fp_use_none_stop_f:n
```

---

<sup>9</sup>Bruno: I must double check this assumption.

```

17073      \int_value:w 1#5#6
17074      \exp_after:wN ;
17075      \exp_after:wN ;
17076  }
17077 \cs_new:Npn \__fp_fixed_to_loop:N #1
17078  {
17079      \if_meaning:w 0 #1
17080      - 1
17081      \exp_after:wN \__fp_fixed_to_loop:N
17082  \else:
17083      \exp_after:wN \__fp_fixed_to_loop_end:w
17084      \exp_after:wN #1
17085  \fi:
17086 }
17087 \cs_new:Npn \__fp_fixed_to_loop_end:w #1 #2 ;
17088  {
17089      \if_meaning:w ; #1
17090      \exp_after:wN \__fp_fixed_to_float_zero:w
17091  \else:
17092      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
17093      \exp_after:wN \__fp_pack_twice_four:wNNNNNNNN
17094      \exp_after:wN \__fp_fixed_to_float_pack:ww
17095      \exp_after:wN ;
17096  \fi:
17097      #1 #2 0000 0000 0000 0000 ;
17098 }
17099 \cs_new:Npn \__fp_fixed_to_float_zero:w ; 0000 0000 0000 0000 ;
17100  {
17101      - 2 * \c_fp_max_exponent_int ;
17102      {0000} {0000} {0000} {0000} ;
17103 }
17104 \cs_new:Npn \__fp_fixed_to_float_pack:ww #1 ; #2#3 ; ;
17105  {
17106      \if_int_compare:w #2 > 4 \exp_stop_f:
17107      \exp_after:wN \__fp_fixed_to_float_round_up:wnnnnw
17108  \fi:
17109      ; #1 ;
17110 }
17111 \cs_new:Npn \__fp_fixed_to_float_round_up:wnnnnw ; #1#2#3#4 ;
17112  {
17113      \exp_after:wN \__fp_basics_pack_high>NNNNw
17114      \int_value:w \__fp_int_eval:w 1 #1#2
17115      \exp_after:wN \__fp_basics_pack_low>NNNNw
17116      \int_value:w \__fp_int_eval:w 1 #3#4 + 1 ;
17117 }

```

(End definition for `\__fp_fixed_to_float_o:wN` and `\__fp_fixed_to_float_o:Nw`.)

17118 ⟨/initex | package⟩

## 31 I3fp-expo implementation

```

17119  {*initex | package}
17120  ⟨@C=fp⟩

```

```

\_\_fp\_parse\_word\_exp:N Unary functions.
\_\_fp\_parse\_word\_ln:N
17121 \cs_new:Npn \_\_fp\_parse\_word\_exp:N
17122   { \_\_fp\_parse\_unary\_function:NNN \_\_fp\_exp_o:w ? }
17123 \cs_new:Npn \_\_fp\_parse\_word\_ln:N
17124   { \_\_fp\_parse\_unary\_function:NNN \_\_fp\_ln_o:w ? }

```

(End definition for `\_\_fp\_parse\_word\_exp:N` and `\_\_fp\_parse\_word\_ln:N`.)

## 31.1 Logarithm

### 31.1.1 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.

### 31.1.2 Some constants

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  $\ln(5)$ .

```

\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
17125 \tl_const:Nn \c_fp_ln_i_fixed_tl  { {0000}{0000}{0000}{0000}{0000}{0000};}
17126 \tl_const:Nn \c_fp_ln_ii_fixed_tl  { {6931}{4718}{0559}{9453}{0941}{7232};}
17127 \tl_const:Nn \c_fp_ln_iii_fixed_tl  { {10986}{1228}{8668}{1096}{9139}{5245};}
17128 \tl_const:Nn \c_fp_ln_iv_fixed_tl  { {13862}{9436}{1119}{8906}{1883}{4464};}
17129 \tl_const:Nn \c_fp_ln_vi_fixed_tl  { {17917}{5946}{9228}{0550}{0081}{2477};}
17130 \tl_const:Nn \c_fp_ln_vii_fixed_tl { {19459}{1014}{9055}{3133}{0510}{5353};}
17131 \tl_const:Nn \c_fp_ln_viii_fixed_tl { {20794}{4154}{1679}{8359}{2825}{1696};}
17132 \tl_const:Nn \c_fp_ln_ix_fixed_tl  { {21972}{2457}{7336}{2193}{8279}{0490};}
17133 \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.)

### 31.1.3 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 `nan` is itself. Positive normal numbers call `\_fp_ln_npos_o:w`.

```

17134 \cs_new:Npn \_fp_ln_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
17135   {
17136     \if_meaning:w 2 #3
17137       \_fp_case_use:nw { \_fp_invalid_operation_o:nw { ln } }
17138     \fi:
17139     \if_case:w #2 \exp_stop_f:
17140       \_fp_case_use:nw
17141         { \_fp_division_by_zero_o:Nnw \c_minus_inf_fp { ln } }
17142     \or:
17143     \else:
17144       \_fp_case_return_same_o:w
17145     \fi:
17146     \_fp_ln_npos_o:w \s_fp \_fp_chk:w #2#3#4;
17147   }

```

(End definition for `\_fp_ln_o:w`.)

### 31.1.4 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.

```

17148 \cs_new:Npn \_fp_ln_npos_o:w \s_fp \_fp_chk:w 10#1#2#3;
17149   { %^^A todo: ln(1) should be "exact zero", not "underflow"
17150     \exp_after:wN \_fp_sanitize:Nw
17151     \int_value:w % for the overall sign
17152       \if_int_compare:w #1 < 1 \exp_stop_f:
17153         2
17154       \else:
17155         0
17156       \fi:
17157     \exp_after:wN \exp_stop_f:
17158     \int_value:w \_fp_int_eval:w % for the exponent
17159       \_fp_ln_significand:NNNNnnnN #2#3
17160       \_fp_ln_exponent:wn {#1}
17161   }

```

(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

*(continuation)*  $\{ \langle Y_1 \rangle \} \{ \langle Y_2 \rangle \} \{ \langle Y_3 \rangle \} \{ \langle Y_4 \rangle \} \{ \langle Y_5 \rangle \} \{ \langle Y_6 \rangle \}$  ;

where  $Y = -\ln(X)$  as an extended fixed point.

```

17162 \cs_new:Npn \_fp_ln_significand:NNNNnnnN #1#2#3#4
17163   {
17164     \exp_after:wN \_fp_ln_x_ii:wnnnn
17165     \int_value:w
17166       \if_case:w #1 \exp_stop_f:

```

```

17167 \or:
17168   \if_int_compare:w #2 < 4 \exp_stop_f:
17169     \__fp_int_eval:w 10 - #2
17170   \else:
17171     6
17172   \fi:
17173   \or: 4
17174   \or: 3
17175   \or: 2
17176   \or: 2
17177   \or: 2
17178   \else: 1
17179   \fi:
17180 ; { #1 #2 #3 #4 }
17181 }

```

(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  $1 + x = 1 + ac \in [1.7, 2.4]$ .

```

17182 \cs_new:Npn \_fp_ln_x_ii:wnnnn #1; #2#3#4#5
17183 {
17184   \exp_after:wN \_fp_ln_div_after:Nw
17185   \cs:w c_fp_ln_ \_fp_int_to_roman:w #1 _fixed_tl \exp_after:wN \cs_end:
17186   \int_value:w
17187   \exp_after:wN \_fp_ln_x_iv:wnnnnnnn
17188   \int_value:w \_fp_int_eval:w
17189   \exp_after:wN \_fp_ln_x_iii_var:NNNNNw
17190   \int_value:w \_fp_int_eval:w 9999 9990 + #1##2#3 +
17191   \exp_after:wN \_fp_ln_x_iii:NNNNNNw
17192   \int_value:w \_fp_int_eval:w 10 0000 0000 + #1##4#5 ;
17193   {20000} {0000} {0000}
17194 } %^A todo: reoptimize (a generalization attempt failed).
17195 \cs_new:Npn \_fp_ln_x_iii:NNNNNNw #1#2 #3#4#5#6 #7;
17196   { #1#2; {#3#4#5#6} {#7} }
17197 \cs_new:Npn \_fp_ln_x_iii_var:NNNNNw #1 #2#3#4#5 #6;
17198   {
17199     #1#2#3#4#5 + 1 ;
17200     {#1#2#3#4#5} {#6}
17201 }

```

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:wnnnnnnnnn {1 or 2} {8d} ; {{4d}} {{4d}} {fixed-tl}
```

The number is  $x$ . Compute  $y$  by adding 1 to the five first digits.

```
17202 \cs_new:Npn \__fp_ln_x_iv:wnnnnnnnnn #1; #2#3#4#5 #6#7#8#9
17203 {
17204   \exp_after:wN \__fp_div_significand_pack:NNN
17205   \int_value:w \__fp_int_eval:w
17206   \__fp_ln_div_i:w #1 ;
17207   #6 #7 ; {#8} {#9}
17208   {#2} {#3} {#4} {#5}
17209   { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
17210   { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
17211   { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
17212   { \exp_after:wN \__fp_ln_div_ii:wwn \int_value:w #1 }
17213   { \exp_after:wN \__fp_ln_div_vi:wwn \int_value:w #1 }
```

```

17214     }
17215 \cs_new:Npn \__fp_ln_div_i:w #1;
17216 {
17217     \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
17218     \int_value:w \__fp_int_eval:w 999999 + 2 0000 0000 / #1 ; % Q1
17219 }
17220 \cs_new:Npn \__fp_ln_div_ii:wwn #1; #2;#3 % y; B1;B2 <- for k=1
17221 {
17222     \exp_after:wN \__fp_div_significand_pack:NNN
17223     \int_value:w \__fp_int_eval:w
17224         \exp_after:wN \__fp_div_significand_calc:wwnnnnnnn
17225         \int_value:w \__fp_int_eval:w 999999 + #2 #3 / #1 ; % Q2
17226         #2 #3 ;
17227 }
17228 \cs_new:Npn \__fp_ln_div_vi:wwn #1; #2;#3#4#5 #6#7#8#9 %y;F1;F2F3F4x1x2x3x4
17229 {
17230     \exp_after:wN \__fp_div_significand_pack:NNN
17231     \int_value:w \__fp_int_eval:w 1000000 + #2 #3 / #1 ; % Q6
17232 }

```

We now have essentially

```

\__fp_ln_div_after:Nw ⟨fixed tl⟩
\__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  $10^6 + Q_6$  ;
⟨exponent⟩ ; ⟨continuation⟩

```

where  $\langle \text{fixed } tl \rangle$  holds the logarithm of a number in  $[1, 10]$ , and  $\langle \text{exponent} \rangle$  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 ⟨fixed-tl⟩ ⟨1d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ;
⟨4d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨4d⟩ ; ⟨exponent⟩ ;

```

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.

```

17233 \cs_new:Npn \__fp_ln_div_after:Nw #1#2;
17234 {
17235     \if_meaning:w 0 #2
17236         \exp_after:wN \__fp_ln_t_small:Nw
17237     \else:
17238         \exp_after:wN \__fp_ln_t_large>NNw
17239         \exp_after:wN -
17240     \fi:
17241     #1
17242 }
17243 \cs_new:Npn \__fp_ln_t_small:Nw #1 #2; #3; #4; #5; #6; #7;
17244 {
17245     \exp_after:wN \__fp_ln_t_large>NNw

```

```

17246      \exp_after:wN + % <sign>
17247      \exp_after:wN #1
17248      \int_value:w \_fp_int_eval:w 9999 - #2 \exp_after:wN ;
17249      \int_value:w \_fp_int_eval:w 9999 - #3 \exp_after:wN ;
17250      \int_value:w \_fp_int_eval:w 9999 - #4 \exp_after:wN ;
17251      \int_value:w \_fp_int_eval:w 9999 - #5 \exp_after:wN ;
17252      \int_value:w \_fp_int_eval:w 9999 - #6 \exp_after:wN ;
17253      \int_value:w \_fp_int_eval:w 1 0000 - #7 ;
17254  }

\_fp_ln_t_large>NNw <sign> <fixed tl>
  <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.

```

17255 \cs_new:Npn \_fp_ln_t_large>NNw #1 #2 #3; #4; #5; #6; #7; #8;
17256  {
17257  \exp_after:wN \_fp_ln_square_t_after:w
17258  \int_value:w \_fp_int_eval:w 9999 0000 + #3*#3
17259  \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
17260  \int_value:w \_fp_int_eval:w 9999 0000 + 2*#3*#4
17261  \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
17262  \int_value:w \_fp_int_eval:w 9999 0000 + 2*#3*#5 + #4*#4
17263  \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
17264  \int_value:w \_fp_int_eval:w 9999 0000 + 2*#3*#6 + 2*#4*#5
17265  \exp_after:wN \_fp_ln_square_t_pack:NNNNNw
17266  \int_value:w \_fp_int_eval:w
17267  1 0000 0000 + 2*#3*#7 + 2*#4*#6 + #5*#5
17268  + (2*#3*#8 + 2*#4*#7 + 2*#5*#6) / 1 0000
17269  % ; ; ;
17270  \exp_after:wN \_fp_ln_twice_t_after:w
17271  \int_value:w \_fp_int_eval:w -1 + 2*#3
17272  \exp_after:wN \_fp_ln_twice_t_pack:Nw
17273  \int_value:w \_fp_int_eval:w 9999 + 2*#4
17274  \exp_after:wN \_fp_ln_twice_t_pack:Nw
17275  \int_value:w \_fp_int_eval:w 9999 + 2*#5
17276  \exp_after:wN \_fp_ln_twice_t_pack:Nw
17277  \int_value:w \_fp_int_eval:w 9999 + 2*#6
17278  \exp_after:wN \_fp_ln_twice_t_pack:Nw
17279  \int_value:w \_fp_int_eval:w 9999 + 2*#7
17280  \exp_after:wN \_fp_ln_twice_t_pack:Nw
17281  \int_value:w \_fp_int_eval:w 10000 + 2*#8 ; ;
17282  { \_fp_ln_c:NwNw #1 }
17283  #2
17284  }
17285 \cs_new:Npn \_fp_ln_twice_t_pack:Nw #1 #2; { + #1 ; {#2} }
17286 \cs_new:Npn \_fp_ln_twice_t_after:w #1; { ; ; {#1} }
17287 \cs_new:Npn \_fp_ln_square_t_pack:NNNNNw #1 #2#3#4#5 #6;
17288 { + #1#2#3#4#5 ; {#6} }
17289 \cs_new:Npn \_fp_ln_square_t_after:w 1 0 #1#2#3 #4;
17290 { \_fp_ln_Taylor:wwNw {0#1#2#3} {#4} }

```

(End definition for  $\_fp_ln_x_{ii}:wnnnn$ .)

\\_\\_fp\\_ln\\_Taylor:wwNw Denoting  $T = t^2$ , we get

```
\_\_fp\_ln\_Taylor:wwNw
{ $T_1$ } { $T_2$ } { $T_3$ } { $T_4$ } { $T_5$ } { $T_6$ } ; ;
{ $(2t)_1$ } { $(2t)_2$ } { $(2t)_3$ } { $(2t)_4$ } { $(2t)_5$ } { $(2t)_6$ } ;
{ \_\_fp\_ln\_c:NwNw <sign> }
<fixed t1> <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 ( $< 10^4$ ).

```
17291 \cs_new:Npn \_\_fp\_ln\_Taylor:wwNw
17292   { \_\_fp\_ln\_Taylor\_loop:www 21 ; {0000}{0000}{0000}{0000}{0000}{0000} ; }
17293 \cs_new:Npn \_\_fp\_ln\_Taylor\_loop:www #1; #2; #3;
17294   {
17295     \if_int_compare:w #1 = 1 \exp_stop_f:
17296       \_\_fp\_ln\_Taylor\_break:w
17297     \fi:
17298     \exp_after:wN \_\_fp\_fixed\_div\_int:wwN \c\_fp\_one\_fixed\_tl #1;
17299     \_\_fp\_fixed\_add:wwn #2;
17300     \_\_fp\_fixed\_mul:wwn #3;
17301   {
17302     \exp_after:wN \_\_fp\_ln\_Taylor\_loop:www
17303       \int_value:w \_\_fp\_int\_eval:w #1 - 2 ;
17304   }
17305   #3;
17306 }
17307 \cs_new:Npn \_\_fp\_ln\_Taylor\_break:w \fi: #1 \_\_fp\_fixed\_add:wwn #2#3; #4 ;
17308   {
17309     \fi:
17310     \exp_after:wN \_\_fp\_fixed\_mul:wwn
17311     \exp_after:wN { \int_value:w \_\_fp\_int\_eval:w 10000 + #2 } #3;
17312 }
```

(End definition for \\_\\_fp\\_ln\\_Taylor:wwNw.)

```
\_\_fp\_ln\_c:NwNw <sign>
{ $r_1$ } { $r_2$ } { $r_3$ } { $r_4$ } { $r_5$ } { $r_6$ } ;
<fixed t1> <exponent> ; <continuation>
```

We are now reduced to finding  $\ln(c)$  and  $\langle exponent \rangle \ln(10)$  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  $\ln(10/7) \simeq 0.35$ .

```

17313 \cs_new:Npn \__fp_ln_c:NwNw #1 #2; #3
17314 {
17315     \if_meaning:w + #1
17316         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_sub:wwn
17317     \else:
17318         \exp_after:wN \exp_after:wN \exp_after:wN \__fp_fixed_add:wwn
17319     \fi:
17320     #3 #2 ;
17321 }
```

(End definition for  $\_\_fp\_ln\_c:NwNw$ .)

```

\_\_fp_ln_exponent:wn
{⟨s1⟩} {⟨s2⟩} {⟨s3⟩} {⟨s4⟩} {⟨s5⟩} {⟨s6⟩} ;
{⟨exponent⟩}
```

Compute  $\langle exponent \rangle$  times  $\ln(10)$ . Apart from the cases where  $\langle exponent \rangle$  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.

```

17322 \cs_new:Npn \__fp_ln_exponent:wn #1; #2
17323 {
17324     \if_case:w #2 \exp_stop_f:
17325         0 \__fp_case_return:nw { \__fp_fixed_to_float_o:Nw 2 }
17326     \or:
17327         \exp_after:wN \__fp_ln_exponent_one:ww \int_value:w
17328     \else:
17329         \if_int_compare:w #2 > 0 \exp_stop_f:
17330             \exp_after:wN \__fp_ln_exponent_small:NNww
17331             \exp_after:wN 0
17332             \exp_after:wN \__fp_fixed_sub:wwn \int_value:w
17333         \else:
17334             \exp_after:wN \__fp_ln_exponent_small:NNww
17335             \exp_after:wN 2
17336             \exp_after:wN \__fp_fixed_add:wwn \int_value:w -
17337         \fi:
17338     \fi:
17339     #2; #1;
17340 }
```

Now we painfully write all the cases.<sup>10</sup> No overflow nor underflow can happen, except when computing  $\ln(1)$ .

```

17341 \cs_new:Npn \__fp_ln_exponent_one:ww 1; #1;
17342 {
17343     0
17344     \exp_after:wN \__fp_fixed_sub:wwn \c__fp_ln_x_fixed_tl #1;
17345     \__fp_fixed_to_float_o:wN 0
17346 }
```

---

<sup>10</sup>Bruno: do rounding.

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.

```

17347 \cs_new:Npn \__fp_ln_exponent_small:NNww #1#2#3; #4#5#6#7#8#9;
17348 {
17349     4
17350     \exp_after:wN \__fp_fixed_mul:wwn
17351         \c_fp_ln_x_fixed_tl
17352             {#3}{0000}{0000}{0000}{0000}{0000} ;
17353     #2
17354         {0000}{#4}{#5}{#6}{#7}{#8};
17355     \__fp_fixed_to_float_o:wN #1
17356 }
```

(End definition for `\__fp_ln_exponent:wn`.)

## 31.2 Exponential

### 31.2.1 Sign, exponent, and special numbers

```
\__fp_exp_o:w
17357 \cs_new:Npn \__fp_exp_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
17358 {
17359     \if_case:w #2 \exp_stop_f:
17360         \__fp_case_return_o:Nw \c_one_fp
17361     \or:
17362         \exp_after:wN \__fp_exp_normal_o:w
17363     \or:
17364         \if_meaning:w 0 #3
17365             \exp_after:wN \__fp_case_return_o:Nw
17366             \exp_after:wN \c_inf_fp
17367         \else:
17368             \exp_after:wN \__fp_case_return_o:Nw
17369             \exp_after:wN \c_zero_fp
17370         \fi:
17371     \or:
17372         \__fp_case_return_same_o:w
17373     \fi:
17374     \s_fp \__fp_chk:w #2#3#4;
17375 }
```

(End definition for `\__fp_exp_o:w`.)

```
\__fp_exp_normal_o:w
\__fp_exp_pos_o:NNnw
\__fp_exp_overflow>NN
17376 \cs_new:Npn \__fp_exp_normal_o:w \s_fp \__fp_chk:w 1#1
17377 {
17378     \if_meaning:w 0 #1
17379         \__fp_exp_pos_o:NNnw + \__fp_fixed_to_float_o:wN
17380     \else:
17381         \__fp_exp_pos_o:NNnw - \__fp_fixed_inv_to_float_o:wN
17382     \fi:
17383 }
17384 \cs_new:Npn \__fp_exp_pos_o:NNnw #1#2#3 \fi: #4#5;
17385 {
```

```

17386   \fi:
17387   \if_int_compare:w #4 > \c_fp_max_exp_exponent_int
17388     \token_if_eq_charcode:NNTF + #1
17389       { \__fp_exp_overflow>NN \__fp_overflow:w \c_inf_fp }
17390       { \__fp_exp_overflow>NN \__fp_underflow:w \c_zero_fp }
17391     \exp:w
17392   \else:
17393     \exp_after:wN \__fp_sanitize:Nw
17394     \exp_after:wN 0
17395     \int_value:w #1 \__fp_int_eval:w
17396       \if_int_compare:w #4 < 0 \exp_stop_f:
17397         \exp_after:wN \use_i:nn
17398       \else:
17399         \exp_after:wN \use_ii:nn
17400       \fi:
17401     {
17402       0
17403       \__fp_decimate:nNnnn { - #4 }
17404         \__fp_exp_Taylor:Nnnwn
17405     }
17406     {
17407       \__fp_decimate:nNnnnn { \c_fp_prec_int - #4 }
17408         \__fp_exp_pos_large:NnnNwn
17409     }
17410     #5
17411     {#4}
17412     #1 #2 0
17413     \exp:w
17414   \fi:
17415   \exp_after:wN \exp_end:
17416 }
17417 \cs_new:Npn \__fp_exp_overflow>NN #1#2
17418 {
17419   \exp_after:wN \exp_after:wN
17420   \exp_after:wN #1
17421   \exp_after:wN #2
17422 }

```

(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.

```

17423 \cs_new:Npn \__fp_exp_Taylor:Nnnwn #1#2#3 #4; #5 #6
17424 {
17425   #6
17426   \__fp_pack_twice_four:wNNNNNNNN
17427   \__fp_pack_twice_four:wNNNNNNNN
17428   \__fp_pack_twice_four:wNNNNNNNN
17429   \__fp_exp_Taylor_ii:ww
17430   ; #2#3#4 0000 0000 ;
17431 }
17432 \cs_new:Npn \__fp_exp_Taylor_ii:ww #1; #2;

```

```

17433 { \__fp_exp_Taylor_loop:www 10 ; #1 ; #1 ; \s_stop }
17434 \cs_new:Npn \__fp_exp_Taylor_loop:www #1; #2; #3;
17435 {
17436     \if_int_compare:w #1 = 1 \exp_stop_f:
17437         \exp_after:wN \__fp_exp_Taylor_break:Nww
17438     \fi:
17439     \__fp_fixed_div_int:wwN #3 ; #1 ;
17440     \__fp_fixed_add_one:wN
17441     \__fp_fixed_mul:wwn #2 ;
17442 {
17443     \exp_after:wN \__fp_exp_Taylor_loop:www
17444     \int_value:w \__fp_int_eval:w #1 - 1 ;
17445     #2 ;
17446 }
17447 }
17448 \cs_new:Npn \__fp_exp_Taylor_break:Nww #1 #2; #3 \s_stop
17449 { \__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.

```

17450 \intarray_const_from_clist:Nn \c__fp_exp_intarray
17451 {
17452     1 , 1 1105 1709 , 1 1807 5647 , 1 6248 1171 ,
17453     1 , 1 1221 4027 , 1 5816 0169 , 1 8339 2107 ,
17454     1 , 1 1349 8588 , 1 0757 6003 , 1 1039 8374 ,
17455     1 , 1 1491 8246 , 1 9764 1270 , 1 3178 2485 ,
17456     1 , 1 1648 7212 , 1 7070 0128 , 1 1468 4865 ,
17457     1 , 1 1822 1188 , 1 0039 0508 , 1 9748 7537 ,
17458     1 , 1 2013 7527 , 1 0747 0476 , 1 5216 2455 ,
17459     1 , 1 2225 5409 , 1 2849 2467 , 1 6045 7954 ,
17460     1 , 1 2459 6031 , 1 1115 6949 , 1 6638 0013 ,
17461     1 , 1 2718 2818 , 1 2845 9045 , 1 2353 6029 ,
17462     1 , 1 7389 0560 , 1 9893 0650 , 1 2272 3043 ,
17463     2 , 1 2008 5536 , 1 9231 8766 , 1 7740 9285 ,
17464     2 , 1 5459 8150 , 1 0331 4423 , 1 9078 1103 ,
17465     3 , 1 1484 1315 , 1 9102 5766 , 1 0342 1116 ,
17466     3 , 1 4034 2879 , 1 3492 7351 , 1 2260 8387 ,
17467     4 , 1 1096 6331 , 1 5842 8458 , 1 5992 6372 ,
17468     4 , 1 2980 9579 , 1 8704 1728 , 1 2747 4359 ,
17469     4 , 1 8103 0839 , 1 2757 5384 , 1 0077 1000 ,
17470     5 , 1 2202 6465 , 1 7948 0671 , 1 6516 9579 ,
17471     9 , 1 4851 6519 , 1 5409 7902 , 1 7796 9107 ,
17472     14 , 1 1068 6474 , 1 5815 2446 , 1 2146 9905 ,
17473     18 , 1 2353 8526 , 1 6837 0199 , 1 8540 7900 ,
17474     22 , 1 5184 7055 , 1 2858 7072 , 1 4640 8745 ,
```

```

17475      27 , 1 1142 0073 , 1 8981 5684 , 1 2836 6296 ,
17476      31 , 1 2515 4386 , 1 7091 9167 , 1 0062 6578 ,
17477      35 , 1 5540 6223 , 1 8439 3510 , 1 0525 7117 ,
17478      40 , 1 1220 4032 , 1 9431 7840 , 1 8020 0271 ,
17479      44 , 1 2688 1171 , 1 4181 6135 , 1 4484 1263 ,
17480      87 , 1 7225 9737 , 1 6812 5749 , 1 2581 7748 ,
17481      131 , 1 1942 4263 , 1 9524 1255 , 1 9365 8421 ,
17482      174 , 1 5221 4696 , 1 8976 4143 , 1 9505 8876 ,
17483      218 , 1 1403 5922 , 1 1785 2837 , 1 4107 3977 ,
17484      261 , 1 3773 0203 , 1 0092 9939 , 1 8234 0143 ,
17485      305 , 1 1014 2320 , 1 5473 5004 , 1 5094 5533 ,
17486      348 , 1 2726 3745 , 1 7211 2566 , 1 5673 6478 ,
17487      391 , 1 7328 8142 , 1 2230 7421 , 1 7051 8866 ,
17488      435 , 1 1970 0711 , 1 1401 7046 , 1 9938 8888 ,
17489      869 , 1 3881 1801 , 1 9428 4368 , 1 5764 8232 ,
17490      1303 , 1 7646 2009 , 1 8905 4704 , 1 8893 1073 ,
17491      1738 , 1 1506 3559 , 1 7005 0524 , 1 9009 7592 ,
17492      2172 , 1 2967 6283 , 1 8402 3667 , 1 0689 6630 ,
17493      2606 , 1 5846 4389 , 1 5650 2114 , 1 7278 5046 ,
17494      3041 , 1 1151 7900 , 1 5080 6878 , 1 2914 4154 ,
17495      3475 , 1 2269 1083 , 1 0850 6857 , 1 8724 4002 ,
17496      3909 , 1 4470 3047 , 1 3316 5442 , 1 6408 6591 ,
17497      4343 , 1 8806 8182 , 1 2566 2921 , 1 5872 6150 ,
17498      8686 , 1 7756 0047 , 1 2598 6861 , 1 0458 3204 ,
17499      13029 , 1 6830 5723 , 1 7791 4884 , 1 1932 7351 ,
17500      17372 , 1 6015 5609 , 1 3095 3052 , 1 3494 7574 ,
17501      21715 , 1 5297 7951 , 1 6443 0315 , 1 3251 3576 ,
17502      26058 , 1 4665 6719 , 1 0099 3379 , 1 5527 2929 ,
17503      30401 , 1 4108 9724 , 1 3326 3186 , 1 5271 5665 ,
17504      34744 , 1 3618 6973 , 1 3140 0875 , 1 3856 4102 ,
17505      39087 , 1 3186 9209 , 1 6113 3900 , 1 6705 9685 ,
17506      }

```

(End definition for `\c_fp_exp_intarray`.)

The first two arguments are irrelevant (a rounding digit, and a brace group with 8 zeros). 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.

```

17507 \cs_new:Npn \_fp_exp_pos_large:NnnNwn #1#2#3 #4#5; #6
17508  {
17509      \exp_after:wN \exp_after:wN \exp_after:wN \_fp_exp_large:NwN
17510      \exp_after:wN \exp_after:wN \exp_after:wN #6
17511      \exp_after:wN \c_fp_one_fixed_tl
17512      \int_value:w #3 #4 \exp_stop_f:
17513      #5 00000 ;
17514  }
17515 \cs_new:Npn \_fp_exp_large:NwN #1#2; #3
17516  {

```

```

17517   \if_case:w #3 ~
17518     \exp_after:wN \__fp_fixed_continue:wn
17519   \else:
17520     \exp_after:wN \__fp_exp_intarray:w
17521     \int_value:w \__fp_int_eval:w 36 * #1 + 4 * #3 \exp_after:wN ;
17522   \fi:
17523 #2;
17524 {
17525   \if_meaning:w 0 #1
17526     \exp_after:wN \__fp_exp_large_after:wwn
17527   \else:
17528     \exp_after:wN \__fp_exp_large:NwN
17529     \int_value:w \__fp_int_eval:w #1 - 1 \exp_after:wN \scan_stop:
17530   \fi:
17531 }
17532 }
17533 \cs_new:Npn \__fp_exp_intarray:w #1 ;
17534 {
17535 +
17536 \__kernel_intarray_item:Nn \c__fp_exp_intarray
17537   { \__fp_int_eval:w #1 - 3 \scan_stop: }
17538 \exp_after:wN \use_i:nnn
17539 \exp_after:wN \__fp_fixed_mul:wwn
17540 \int_value:w 0
17541 \exp_after:wN \__fp_exp_intarray_aux:w
17542 \int_value:w \__kernel_intarray_item:Nn
17543   \c__fp_exp_intarray { \__fp_int_eval:w #1 - 2 }
17544 \exp_after:wN \__fp_exp_intarray_aux:w
17545 \int_value:w \__kernel_intarray_item:Nn
17546   \c__fp_exp_intarray { \__fp_int_eval:w #1 - 1 }
17547 \exp_after:wN \__fp_exp_intarray_aux:w
17548 \int_value:w \__kernel_intarray_item:Nn \c__fp_exp_intarray {#1} ; ;
17549 }
17550 \cs_new:Npn \__fp_exp_intarray_aux:w 1 #1#2#3#4#5 ; { ; {#1#2#3#4} {#5} }
17551 \cs_new:Npn \__fp_exp_large_after:wwn #1; #2; #3
17552 {
17553   \__fp_exp_Taylor:Nnnwn ? { } { } 0 #2; {} #3
17554   \__fp_fixed_mul:wwn #1;
17555 }

```

(End definition for  $\_\_fp\_exp\_pos\_large:NnnNwn$  and others.)

### 31.3 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
$+0$	$+\infty$	$+\infty$	$+1$		$+0$		$+0$	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
$-\infty$	$+0$	$(-1)^b 0$	$+1$	$(-1)^b \infty$	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  $\{ b \}$ , is inserted between  $a$  and  $b$ ). Then do some tests to find the final sign of the result if it exists.

```

17556 \cs_new:cpn { __fp_ \iow_char:N \^ _o:ww }
17557   \s_fp \__fp_chk:w #1#2#3; \s_fp \__fp_chk:w #4#5#6;
17558   {
17559     \if_meaning:w 0 #4
17560       \__fp_case_return_o:Nw \c_one_fp
17561     \fi:
17562     \if_case:w #2 \exp_stop_f:
17563       \exp_after:wN \use_i:nn
17564     \or:
17565       \__fp_case_return_o:Nw \c_nan_fp
17566     \else:
17567       \exp_after:wN \__fp_pow_neg:www
17568       \exp:w \exp_end_continue_f:w \exp_after:wN \use:nn
17569     \fi:
17570   {
17571     \if_meaning:w 1 #1
17572       \exp_after:wN \__fp_pow_normal_o:ww
17573     \else:
17574       \exp_after:wN \__fp_pow_zero_or_inf:ww
17575     \fi:
17576     \s_fp \__fp_chk:w #1#2#3;
17577   }

```

```

17578     { \s_fp \fp_chk:w #4#5#6; \s_fp \fp_chk:w #1#2#3; }
17579     \s_fp \fp_chk:w #4#5#6;
17580 }
```

(End definition for  $\_\_fp\_{^o:ww}$ .)

$\_\_fp\_pow\_zero\_or\_inf:ww$

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  $b = -\infty$ .

```

17581 \cs_new:Npn \fp_pow_zero_or_inf:ww
17582     \s_fp \fp_chk:w #1#2; \s_fp \fp_chk:w #3#4
17583 {
17584     \if_meaning:w 1 #4
17585         \fp_case_return_same_o:w
17586     \fi:
17587     \if_meaning:w #1 #4
17588         \fp_case_return_o:Nw \c_zero_fp
17589     \fi:
17590     \if_meaning:w 2 #1
17591         \fp_case_return_o:Nw \c_inf_fp
17592     \fi:
17593     \if_meaning:w 2 #3
17594         \fp_case_return_o:Nw \c_inf_fp
17595     \else:
17596         \fp_case_use:nw
17597         {
17598             \fp_division_by_zero_o>NNww \c_inf_fp ^
17599                 \s_fp \fp_chk:w #1 #2 ;
17600         }
17601     \fi:
17602     \s_fp \fp_chk:w #3#4
17603 }
```

(End definition for  $\_\_fp\_pow\_zero\_or\_inf:ww$ .)

$\_\_fp\_pow\_normal_o:ww$

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  $b$ .

```

17604 \cs_new:Npn \fp_pow_normal_o:ww
17605     \s_fp \fp_chk:w 1 #1#2#3; \s_fp \fp_chk:w #4#5
17606 {
17607     \if_int_compare:w \fp_str_if_eq:nn { #2 #3 }
```

```

17608      { 1 {1000} {0000} {0000} {0000} } = 0 \exp_stop_f:
17609      \if_int_compare:w #4 #1 = 32 \exp_stop_f:
17610          \exp_after:wN \__fp_case_return_ii_o:ww
17611      \fi:
17612          \__fp_case_return_o:Nww \c_one_fp
17613      \fi:
17614      \if_case:w #4 \exp_stop_f:
17615      \or:
17616          \exp_after:wN \__fp_pow_npos_o:Nww
17617          \exp_after:wN #5
17618      \or:
17619          \if_meaning:w 2 #5 \exp_after:wN \reverse_if:N \fi:
17620          \if_int_compare:w #2 > 0 \exp_stop_f:
17621              \exp_after:wN \__fp_case_return_o:Nww
17622              \exp_after:wN \c_inf_fp
17623          \else:
17624              \exp_after:wN \__fp_case_return_o:Nww
17625              \exp_after:wN \c_zero_fp
17626          \fi:
17627          \or:
17628              \__fp_case_return_ii_o:ww
17629          \fi:
17630          \s_fp \__fp_chk:w 1 #1 {#2} #3 ;
17631          \s_fp \__fp_chk:w #4 #5
17632      }

```

(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.

```

17633 \cs_new:Npn \__fp_pow_npos_o:Nww #1 \s_fp \__fp_chk:w 1#2#3
17634 {
17635     \exp_after:wN \__fp_SANITIZE:Nw
17636     \exp_after:wN 0
17637     \int_value:w
17638         \if:w #1 \if_int_compare:w #3 > 0 \exp_stop_f: 0 \else: 2 \fi:
17639             \exp_after:wN \__fp_pow_npos_aux:NNnww
17640             \exp_after:wN +
17641             \exp_after:wN \__fp_fixed_to_float_o:wN
17642         \else:
17643             \exp_after:wN \__fp_pow_npos_aux:NNnww
17644             \exp_after:wN -
17645             \exp_after:wN \__fp_fixed_inv_to_float_o:wN
17646         \fi:
17647     {#3}
17648 }

```

(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  $-\ln(x)$ .

```
17649 \cs_new:Npn \__fp_pow_npos_aux:NNnww #1#2#3#4#5; \s_fp \__fp_chk:w 1#6#7#8;
17650 {
17651     #1
17652     \__fp_int_eval:w
17653         \__fp_ln_significand:NNNNnnnnN #4#5
17654         \__fp_pow_exponent:wnN {#3}
17655         \__fp_fixed_mul:wwn #8 {0000}{0000} ;
17656         \__fp_pow_B:wwN #7;
17657         #1 #2 0 % fixed_to_float_o:wn
17658     }
17659 \cs_new:Npn \__fp_pow_exponent:wnN #1; #2
17660 {
17661     \if_int_compare:w #2 > 0 \exp_stop_f:
17662         \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % n\ln(10) - (-\ln(x))
17663         \exp_after:wN +
17664     \else:
17665         \exp_after:wN \__fp_pow_exponent:Nwnnnnnw % -(|n|\ln(10) + (-\ln(x)))
17666         \exp_after:wN -
17667     \fi:
17668     #2; #1;
17669 }
17670 \cs_new:Npn \__fp_pow_exponent:Nwnnnnnw #1#2; #3#4#5#6#7#8;
17671 { %^^A todo: use that in ln.
17672     \exp_after:wN \__fp_fixed_mul_after:wwn
17673     \int_value:w \__fp_int_eval:w \c__fp_leading_shift_int
17674         \exp_after:wN \__fp_pack:NNNNNw
17675     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
17676         #1#2*23025 - #1 #3
17677         \exp_after:wN \__fp_pack:NNNNNw
17678     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
17679         #1 #2*8509 - #1 #4
17680         \exp_after:wN \__fp_pack:NNNNNw
17681     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
17682         #1 #2*2994 - #1 #5
17683         \exp_after:wN \__fp_pack:NNNNNw
17684     \int_value:w \__fp_int_eval:w \c__fp_middle_shift_int
17685         #1 #2*0456 - #1 #6
17686         \exp_after:wN \__fp_pack:NNNNNw
17687     \int_value:w \__fp_int_eval:w \c__fp_trailing_shift_int
17688         #1 #2*8401 - #1 #7
17689         #1 ( #2*7991 - #8 ) / 1 0000 ; ;
17690     }
17691 \cs_new:Npn \__fp_pow_B:wwN #1#2#3#4#5#6; #7;
17692 {
17693     \if_int_compare:w #7 < 0 \exp_stop_f:
17694         \exp_after:wN \__fp_pow_C_neg:w \int_value:w -
17695     \else:
17696         \if_int_compare:w #7 < 22 \exp_stop_f:
17697             \exp_after:wN \__fp_pow_C_pos:w \int_value:w
17698         \else:
17699             \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
17700         \fi:
```

```

17701     \fi:
17702     #7 \exp_after:wN ;
17703     \int_value:w \__fp_int_eval:w 10 0000 + #1 \__fp_int_eval_end:
17704     #2#3#4#5#6 0000 0000 0000 0000 0000 0000 ; %^A todo: how many 0?
17705   }
17706 \cs_new:Npn \__fp_pow_C_overflow:w #1; #2; #3
17707   {
17708     + 2 * \c_fp_max_exponent_int
17709     \exp_after:wN \__fp_fixed_continue:wn \c_fp_one_fixed_tl
17710   }
17711 \cs_new:Npn \__fp_pow_C_neg:w #1 ; 1
17712   {
17713     \exp_after:wN \exp_after:wN \exp_after:wN \__fp_pow_C_pack:w
17714     \prg_replicate:nn {#1} {0}
17715   }
17716 \cs_new:Npn \__fp_pow_C_pos:w #1; 1
17717   { \__fp_pow_C_pos_loop:wN #1; }
17718 \cs_new:Npn \__fp_pow_C_pos_loop:wN #1; #2
17719   {
17720     \if_meaning:w 0 #1
17721       \exp_after:wN \__fp_pow_C_pack:w
17722       \exp_after:wN #2
17723     \else:
17724       \if_meaning:w 0 #2
17725         \exp_after:wN \__fp_pow_C_pos_loop:wN \int_value:w
17726       \else:
17727         \exp_after:wN \__fp_pow_C_overflow:w \int_value:w
17728       \fi:
17729       \__fp_int_eval:w #1 - 1 \exp_after:wN ;
17730     \fi:
17731   }
17732 \cs_new:Npn \__fp_pow_C_pack:w
17733   {
17734     \exp_after:wN \__fp_exp_large:Nww
17735     \exp_after:wN 5
17736     \c_fp_one_fixed_tl
17737   }

```

(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.

```

17738 \cs_new:Npn \__fp_pow_neg:www \s_fp \__fp_chk:w #1#2; #3; #4;
17739   {
17740     \if_case:w \__fp_pow_neg_case:w #4 ;
17741       \exp_after:wN \__fp_pow_neg_aux:wNN
17742     \or:
17743       \if_int_compare:w \__fp_int_eval:w #1 / 2 = 1 \exp_stop_f:
17744         \__fp_invalid_operation_o:Nww ^ #3; #4;
17745         \exp:w \exp_end_continue_f:w

```

```

17746      \exp_after:wN \exp_after:wN
17747      \exp_after:wN \__fp_use_none_until_s:w
17748      \fi:
17749      \fi:
17750      \__fp_exp_after_o:w
17751      \s_fp \__fp_chk:w #1#2;
17752  }
17753 \cs_new:Npn \__fp_pow_neg_aux:wNN #1 \s_fp \__fp_chk:w #2#3
17754  {
17755      \exp_after:wN \__fp_exp_after_o:w
17756      \exp_after:wN \s_fp
17757      \exp_after:wN \__fp_chk:w
17758      \exp_after:wN #2
17759      \int_value:w \__fp_int_eval:w 2 - #3 \__fp_int_eval_end:
17760  }

```

(End definition for `\__fp_pow:www` and `\__fp_pow_neg_aux:wNN`.)

```

\__fp_pow_neg_case:w
\__fp_pow_neg_case_aux:nnnn
\__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.

```

17761 \cs_new:Npn \__fp_pow_neg_case:w \s_fp \__fp_chk:w #1#2#3;
17762  {
17763      \if_case:w #1 \exp_stop_f:
17764          -1
17765      \or: \__fp_pow_neg_case_aux:nnnnn #3
17766      \or: -1
17767      \else: 1
17768      \fi:
17769      \exp_stop_f:
17770  }
17771 \cs_new:Npn \__fp_pow_neg_case_aux:nnnnn #1#2#3#4#5
17772  {
17773      \if_int_compare:w #1 > \c_fp_prec_int
17774          -1
17775      \else:
17776          \__fp_decimate:nNnnnn { \c_fp_prec_int - #1 }
17777          \__fp_pow_neg_case_aux:Nnnw
17778          {#2} {#3} {#4} {#5}
17779      \fi:
17780  }
17781 \cs_new:Npn \__fp_pow_neg_case_aux:Nnnw #1#2#3#4 ;
17782  {
17783      \if_meaning:w 0 #1
17784          \if_int_odd:w #3 \exp_stop_f:
17785              0
17786          \else:
17787              -1
17788          \fi:
17789      \else:

```

```

17790      1
17791      \fi:
17792  }

(End definition for \_fp_pow_neg_case:w, \_fp_pow_neg_case_aux:nnnn, and \_fp_pow_neg-
case_aux:Nnnw.)

17793 </initex | package>

```

## 32 I3fp-trig Implementation

```

17794 <*initex | package>
17795 <@@=fp>

\_fp_parse_word_acos:N Unary functions.
\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

(End definition for \_fp_parse_word_acos:N and others.)

```

Those functions may receive a variable number of arguments.

```

17815 \cs_new:Npn \fp_parse_word_acot:N
17816  { \fp_parse_function>NNN \fp_acot_o:Nw \use_i:nn }
17817 \cs_new:Npn \fp_parse_word_acotd:N
17818  { \fp_parse_function>NNN \fp_acot_o:Nw \use_ii:nn }
17819 \cs_new:Npn \fp_parse_word_atan:N
17820  { \fp_parse_function>NNN \fp_atan_o:Nw \use_i:nn }
17821 \cs_new:Npn \fp_parse_word_atand:N
17822  { \fp_parse_function>NNN \fp_atan_o:Nw \use_ii:nn }

(End definition for \_fp_parse_word_acot:N and others.)

```

## 32.1 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.

### 32.1.1 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  $\sin(x) = \#3 \sin|x|$ .

```

17823 \cs_new:Npn \_fp_sin_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
17824   {
17825     \if_case:w #2 \exp_stop_f:
17826       \_fp_case_return_same_o:w
17827     \or: \_fp_case_use:nw
17828       {
17829         \_fp_trig:NNNNnw #1 \_fp_sin_series_o:NNwww
17830           \_fp_ep_to_float_o:wwN #3 0
17831       }
17832     \or: \_fp_case_use:nw
17833       { \_fp_invalid_operation_o:fw { #1 { sin } { sind } } }
17834     \else: \_fp_case_return_same_o:w
17835     \fi:
17836     \s_fp \_fp_chk:w #2 #3 #4;
17837   }

```

(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  $\cos(x) = +\sin(\pi/2 + |x|)$ .

```

17838 \cs_new:Npn \__fp_cos_o:w #1 \s__fp \__fp_chk:w #2#3; @
17839   {
17840     \if_case:w #2 \exp_stop_f:
17841       \__fp_case_return_o:Nw \c_one_fp
17842     \or: \__fp_case_use:nw
17843       {
17844         \__fp_trig>NNNNwn #1 \__fp_sin_series_o>NNwwww
17845           \__fp_ep_to_float_o:wwN 0 2
17846       }
17847     \or: \__fp_case_use:nw
17848       { \__fp_invalid_operation_o:fw { #1 { cos } { cosd } } }
17849   \else: \__fp_case_return_same_o:w
17850   \fi:
17851   \s__fp \__fp_chk:w #2 #3;
17852 }
```

(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  $\csc(x) = \#3(\sin|x|)^{-1}$ .

```

17853 \cs_new:Npn \__fp_csc_o:w #1 \s__fp \__fp_chk:w #2#3#4; @
17854   {
17855     \if_case:w #2 \exp_stop_f:
17856       \__fp_cot_zero_o:Nfw #3 { #1 { csc } { cscd } }
17857     \or: \__fp_case_use:nw
17858       {
17859         \__fp_trig>NNNNwn #1 \__fp_sin_series_o>NNwwww
17860           \__fp_inv_to_float_o:wwN #3 0
17861       }
17862     \or: \__fp_case_use:nw
17863       { \__fp_invalid_operation_o:fw { #1 { csc } { cscd } } }
17864   \else: \__fp_case_return_same_o:w
17865   \fi:
17866   \s__fp \__fp_chk:w #2 #3 #4;
17867 }
```

(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  $\sec(x) = +1/\sin(\pi/2 + |x|)$ .

```

17868 \cs_new:Npn \__fp_sec_o:w #1 \s__fp \__fp_chk:w #2#3; @
17869   {
17870     \if_case:w #2 \exp_stop_f:
17871       \__fp_case_return_o:Nw \c_one_fp
```

```

17872     \or:  \__fp_case_use:nw
17873     {
17874         \__fp_trig>NNNNNwn #1 \__fp_sin_series_o>NNwwww
17875             \__fp_ep_inv_to_float_o:wwN 0 2
17876         }
17877     \or:  \__fp_case_use:nw
17878         { \__fp_invalid_operation_o:fw { #1 { sec } { secd } } }
17879     \else: \__fp_case_return_same_o:w
17880     \fi:
17881     \s_fp \__fp_chk:w #2 #3;
17882 }

```

(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.

```

17883 \cs_new:Npn \__fp_tan_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
17884 {
17885     \if_case:w #2 \exp_stop_f:
17886         \__fp_case_return_same_o:w
17887     \or:  \__fp_case_use:nw
17888     {
17889         \__fp_trig>NNNNNwn #1
17890             \__fp_tan_series_o>NNwwww 0 #3 1
17891     }
17892     \or:  \__fp_case_use:nw
17893         { \__fp_invalid_operation_o:fw { #1 { tan } { tand } } }
17894     \else: \__fp_case_return_same_o:w
17895     \fi:
17896     \s_fp \__fp_chk:w #2 #3 #4;
17897 }

```

(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.

```

17898 \cs_new:Npn \__fp_cot_o:w #1 \s_fp \__fp_chk:w #2#3#4; @
17899 {
17900     \if_case:w #2 \exp_stop_f:
17901         \__fp_cot_zero_o:Nfw #3 { #1 { cot } { cotd } }
17902     \or:  \__fp_case_use:nw
17903     {
17904         \__fp_trig>NNNNNwn #1
17905             \__fp_tan_series_o>NNwwww 2 #3 3
17906     }
17907     \or:  \__fp_case_use:nw

```

```

17908          { \__fp_invalid_operation_o:fw { #1 { cot } { cotd } } }
17909      \else: \__fp_case_return_same_o:w
17910      \fi:
17911      \s__fp \__fp_chk:w #2 #3 #4;
17912  }
17913 \cs_new:Npn \__fp_cot_zero_o:Nfw #1#2#3 \fi:
17914  {
17915  \fi:
17916  \token_if_eq_meaning:NNTF 0 #1
17917    { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_inf_fp }
17918    { \exp_args:NNf \__fp_division_by_zero_o:Nnw \c_minus_inf_fp }
17919  {#2}
17920 }

```

(End definition for `\__fp_cot_o:w` and `\__fp_cot_zero_o:Nfw`.)

### 32.1.2 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.

```

17921 \cs_new:Npn \__fp_trig:NNNNNwn #1#2#3#4#5 \s__fp \__fp_chk:w 1#6#7#8;
17922  {
17923  \exp_after:wN #2
17924  \exp_after:wN #3
17925  \exp_after:wN #4
17926  \int_value:w \__fp_int_eval:w #5
17927  \exp_after:wN \exp_after:wN \exp_after:wN \exp_after:wN
17928  \if_int_compare:w #7 > #1 0 1 \exp_stop_f:
17929    #1 \__fp_trig_large:ww \__fp_trigd_large:ww
17930  \else:
17931    #1 \__fp_trig_small:ww \__fp_trigd_small:ww
17932  \fi:
17933  #7,#8{0000}{0000};
17934 }

```

(End definition for `\__fp_trig:NNNNNwn`.)

### 32.1.3 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.

```
17935 \cs_new:Npn \_fp_trig_small:ww #1,#2;
17936 { \_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.

```
17937 \cs_new:Npn \_fp_trigd_small:ww #1,#2;
17938 {
17939     \_fp_ep_mul_raw:wwwn
17940     -1,{1745}{3292}{5199}{4329}{5769}{2369}; #1,#2;
17941     \_fp_trig_small:ww
17942 }
```

(End definition for \\_fp\_trigd\_small:ww.)

### 32.1.4 Argument reduction in degrees

\\_fp\_trigd\_large:ww

\\_fp\_trigd\_large\_auxi:nmmnwNNNN

\\_fp\_trigd\_large\_auxii:wNw

\\_fp\_trigd\_large\_auxiii:www

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).

```
17943 \cs_new:Npn \_fp_trigd_large:ww #1, #2#3#4#5#6#7;
17944 {
17945     \exp_after:wN \_fp_pack_eight:wNNNNNNNN
17946     \exp_after:wN \_fp_pack_eight:wNNNNNNNN
17947     \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
17948     \exp_after:wN \_fp_pack_twice_four:wNNNNNNNN
17949     \exp_after:wN \_fp_trigd_large_auxi:nmmnwNNNN
17950     \exp_after:wN ;
17951     \exp:w \exp_end_continue_f:w
```

```

17952      \prg_replicate:nn { \int_max:nn { 22 - #1 } { 0 } } { 0 }
17953      #2#3#4#5#6#7 0000 0000 0000 !
17954  }
17955 \cs_new:Npn \__fp_trigd_large_auxi:nnnnNNNN #1#2#3#4#5; #6#7#8#9
17956  {
17957      \exp_after:wN \__fp_trigd_large_auxii:wNw
17958      \int_value:w \__fp_int_eval:w #1 + #2
17959      - (#1 + #2 - 4) / 9 * 9 \__fp_int_eval_end:
17960      #3;
17961      #4; #5{#6#7#8#9};
17962  }
17963 \cs_new:Npn \__fp_trigd_large_auxii:wNw #1; #2#3;
17964  {
17965      + (#1#2 - 4) / 9 * 2
17966      \exp_after:wN \__fp_trigd_large_auxiii:www
17967      \int_value:w \__fp_int_eval:w #1#2
17968      - (#1#2 - 4) / 9 * 9 \__fp_int_eval_end: #3 ;
17969  }
17970 \cs_new:Npn \__fp_trigd_large_auxiii:www #1; #2; #3!
17971  {
17972      \if_int_compare:w #1 < 4500 \exp_stop_f:
17973      \exp_after:wN \__fp_use_i_until_s:nw
17974      \exp_after:wN \__fp_fixed_continue:wn
17975      \else:
17976      + 1
17977      \fi:
17978      \__fp_fixed_sub:wwn {9000}{0000}{0000}{0000}{0000}{0000};
17979      {#1}#2{0000}{0000};
17980      { \__fp_trigd_small:ww 2, }
17981  }

```

(End definition for `\__fp_trigd_large:ww` and others.)

### 32.1.5 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.

```
17982 \intarray_const_from_clist:Nn \c_fp_trig_intarray
17983 {
17984     100000000, 100000000, 115915494, 130918953, 135768883, 176337251,
17985     143620344, 159645740, 145644874, 176673440, 158896797, 163422653,
17986     150901138, 102766253, 108595607, 128427267, 157958036, 189291184,
17987     161145786, 152877967, 141073169, 198392292, 139966937, 140907757,
17988     130777463, 196925307, 168871739, 128962173, 197661693, 136239024,
17989     117236290, 111832380, 111422269, 197557159, 140461890, 108690267,
17990     139561204, 189410936, 193784408, 155287230, 199946443, 140024867,
17991     123477394, 159610898, 132309678, 130749061, 166986462, 180469944,
17992     186521878, 181574786, 156696424, 110389958, 174139348, 160998386,
17993     180991999, 162442875, 158517117, 188584311, 117518767, 116054654,
17994     175369880, 109739460, 136475933, 137680593, 102494496, 163530532,
17995     171567755, 103220324, 177781639, 171660229, 146748119, 159816584,
17996     106060168, 103035998, 113391198, 174988327, 186654435, 127975507,
17997     100162406, 177564388, 184957131, 108801221, 199376147, 168137776,
17998     147378906, 133068046, 145797848, 117613124, 127314069, 196077502,
17999     145002977, 159857089, 105690279, 167851315, 125210016, 131774602,
18000     109248116, 106240561, 145620314, 164840892, 148459191, 143521157,
18001     154075562, 100871526, 160680221, 171591407, 157474582, 172259774,
18002     162853998, 175155329, 139081398, 117724093, 158254797, 107332871,
18003     190406999, 175907657, 170784934, 170393589, 182808717, 134256403,
18004     166895116, 162545705, 194332763, 112686500, 126122717, 197115321,
18005     112599504, 138667945, 103762556, 108363171, 116952597, 158128224,
18006     194162333, 143145106, 112353687, 185631136, 136692167, 114206974,
18007     169601292, 150578336, 105311960, 185945098, 139556718, 170995474,
18008     165104316, 123815517, 158083944, 129799709, 199505254, 138756612,
18009     194458833, 106846050, 178529151, 151410404, 189298850, 163881607,
18010     176196993, 107341038, 199957869, 118905980, 193737772, 106187543,
18011     122271893, 101366255, 126123878, 103875388, 181106814, 106765434,
18012     108282785, 126933426, 179955607, 107903860, 160352738, 199624512,
18013     159957492, 176297023, 159409558, 143011648, 129641185, 157771240,
18014     157544494, 157021789, 176979240, 194903272, 194770216, 164960356,
18015     153181535, 144003840, 168987471, 176915887, 163190966, 150696440,
18016     147769706, 187683656, 177810477, 197954503, 153395758, 130188183,
18017     186879377, 166124814, 195305996, 155802190, 183598751, 103512712,
18018     190432315, 180498719, 168687775, 194656634, 162210342, 104440855,
18019     149785037, 192738694, 129353661, 193778292, 187359378, 143470323,
18020     102371458, 137923557, 111863634, 119294601, 183182291, 196416500,
18021     187830793, 131353497, 179099745, 186492902, 167450609, 189368909,
18022     145883050, 133703053, 180547312, 132158094, 131976760, 132283131,
18023     141898097, 149822438, 133517435, 169898475, 101039500, 168388003,
18024     197867235, 199608024, 100273901, 108749548, 154787923, 156826113,
18025     199489032, 168997427, 108349611, 149208289, 103776784, 174303550,
18026     145684560, 183671479, 130845672, 133270354, 185392556, 120208683,
18027     193240995, 162211753, 131839402, 109707935, 170774965, 149880868,
```

18028	160663609, 168661967, 103747454, 121028312, 119251846, 122483499,
18029	111611495, 166556037, 196967613, 199312829, 196077608, 127799010,
18030	107830360, 102338272, 198790854, 102387615, 157445430, 192601191,
18031	100543379, 198389046, 154921248, 129516070, 172853005, 122721023,
18032	160175233, 113173179, 175931105, 103281551, 109373913, 163964530,
18033	157926071, 180083617, 195487672, 146459804, 173977292, 144810920,
18034	109371257, 186918332, 189588628, 139904358, 168666639, 175673445,
18035	114095036, 137327191, 174311388, 106638307, 125923027, 159734506,
18036	105482127, 178037065, 133778303, 121709877, 134966568, 149080032,
18037	169885067, 141791464, 168350828, 116168533, 114336160, 173099514,
18038	198531198, 119733758, 144420984, 116559541, 152250643, 139431286,
18039	144403838, 183561508, 179771645, 101706470, 167518774, 156059160,
18040	187168578, 157939226, 123475633, 117111329, 198655941, 159689071,
18041	198506887, 144230057, 151919770, 156900382, 118392562, 120338742,
18042	135362568, 108354156, 151729710, 188117217, 195936832, 156488518,
18043	174997487, 108553116, 159830610, 113921445, 144601614, 188452770,
18044	125114110, 170248521, 173974510, 138667364, 103872860, 109967489,
18045	131735618, 112071174, 104788993, 168886556, 192307848, 150230570,
18046	157144063, 163863202, 136852010, 174100574, 185922811, 115721968,
18047	100397824, 175953001, 166958522, 112303464, 118773650, 143546764,
18048	164565659, 171901123, 108476709, 193097085, 191283646, 166919177,
18049	169387914, 133315566, 150669813, 121641521, 100895711, 172862384,
18050	126070678, 145176011, 113450800, 169947684, 122356989, 162488051,
18051	157759809, 153397080, 185475059, 175362656, 149034394, 145420581,
18052	178864356, 183042000, 131509559, 147434392, 152544850, 167491429,
18053	108647514, 142303321, 133245695, 111634945, 167753939, 142403609,
18054	105438335, 152829243, 142203494, 184366151, 146632286, 102477666,
18055	166049531, 140657343, 157553014, 109082798, 180914786, 169343492,
18056	127376026, 134997829, 195701816, 119643212, 133140475, 176289748,
18057	140828911, 174097478, 126378991, 181699939, 148749771, 151989818,
18058	172666294, 160183053, 195832752, 109236350, 168538892, 128468247,
18059	125997252, 183007668, 156937583, 165972291, 198244297, 147406163,
18060	181831139, 158306744, 134851692, 185973832, 137392662, 140243450,
18061	119978099, 140402189, 161348342, 173613676, 144991382, 171541660,
18062	163424829, 136374185, 106122610, 186132119, 198633462, 184709941,
18063	183994274, 129559156, 128333990, 148038211, 175011612, 111667205,
18064	119125793, 103552929, 124113440, 131161341, 112495318, 138592695,
18065	184904438, 146807849, 109739828, 108855297, 104515305, 139914009,
18066	188698840, 188365483, 166522246, 168624087, 125401404, 100911787,
18067	142122045, 123075334, 173972538, 114940388, 141905868, 142311594,
18068	163227443, 139066125, 116239310, 162831953, 123883392, 113153455,
18069	163815117, 152035108, 174595582, 101123754, 135976815, 153401874,
18070	107394340, 136339780, 138817210, 104531691, 182951948, 179591767,
18071	139541778, 179243527, 161740724, 160593916, 102732282, 187946819,
18072	136491289, 149714953, 143255272, 135916592, 198072479, 198580612,
18073	169007332, 118844526, 179433504, 155801952, 149256630, 162048766,
18074	116134365, 133992028, 175452085, 155344144, 109905129, 182727454,
18075	165911813, 122232840, 151166615, 165070983, 175574337, 129548631,
18076	120411217, 116380915, 160616116, 157320000, 183306114, 160618128,
18077	103262586, 195951602, 146321661, 138576614, 180471993, 127077713,
18078	116441201, 159496011, 106328305, 120759583, 148503050, 179095584,
18079	198298218, 167402898, 138551383, 123957020, 180763975, 150429225,
18080	198476470, 171016426, 197438450, 143091658, 164528360, 132493360,
18081	143546572, 137557916, 113663241, 120457809, 196971566, 134022158,

18082	180545794, 131328278, 100552461, 132088901, 187421210, 192448910,
18083	141005215, 149680971, 113720754, 100571096, 134066431, 135745439,
18084	191597694, 135788920, 179342561, 177830222, 137011486, 142492523,
18085	192487287, 113132021, 176673607, 156645598, 127260957, 141566023,
18086	143787436, 129132109, 174858971, 150713073, 191040726, 143541417,
18087	197057222, 165479803, 181512759, 157912400, 125344680, 148220261,
18088	173422990, 101020483, 106246303, 137964746, 178190501, 181183037,
18089	151538028, 179523433, 141955021, 135689770, 191290561, 143178787,
18090	192086205, 174499925, 178975690, 118492103, 124206471, 138519113,
18091	188147564, 102097605, 154895793, 178514140, 141453051, 151583964,
18092	128232654, 106020603, 131189158, 165702720, 186250269, 191639375,
18093	115278873, 160608114, 155694842, 110322407, 177272742, 116513642,
18094	134366992, 171634030, 194053074, 180652685, 109301658, 192136921,
18095	141431293, 171341061, 157153714, 106203978, 147618426, 150297807,
18096	186062669, 169960809, 118422347, 163350477, 146719017, 145045144,
18097	161663828, 146208240, 186735951, 102371302, 190444377, 194085350,
18098	134454426, 133413062, 163074595, 113830310, 122931469, 134466832,
18099	185176632, 182415152, 110179422, 164439571, 181217170, 121756492,
18100	119644493, 196532222, 118765848, 182445119, 109401340, 150443213,
18101	198586286, 121083179, 139396084, 143898019, 114787389, 177233102,
18102	186310131, 148695521, 126205182, 178063494, 157118662, 177825659,
18103	188310053, 151552316, 165984394, 109022180, 163144545, 121212978,
18104	197344714, 188741258, 126822386, 102360271, 109981191, 152056882,
18105	134723983, 158013366, 106837863, 128867928, 161973236, 172536066,
18106	185216856, 132011948, 197807339, 158419190, 166595838, 167852941,
18107	124187182, 117279875, 106103946, 106481958, 157456200, 160892122,
18108	184163943, 173846549, 158993202, 184812364, 133466119, 170732430,
18109	195458590, 173361878, 162906318, 150165106, 126757685, 112163575,
18110	188696307, 145199922, 100107766, 176830946, 198149756, 122682434,
18111	179367131, 108412102, 119520899, 148191244, 140487511, 171059184,
18112	141399078, 189455775, 118462161, 190415309, 134543802, 180893862,
18113	180732375, 178615267, 179711433, 123241969, 185780563, 176301808,
18114	184386640, 160717536, 183213626, 129671224, 126094285, 140110963,
18115	121826276, 151201170, 122552929, 128965559, 146082049, 138409069,
18116	107606920, 103954646, 119164002, 115673360, 117909631, 187289199,
18117	186343410, 186903200, 157966371, 103128612, 135698881, 176403642,
18118	152540837, 109810814, 183519031, 121318624, 172281810, 150845123,
18119	169019064, 166322359, 138872454, 163073727, 128087898, 130041018,
18120	194859136, 173742589, 141812405, 167291912, 138003306, 134499821,
18121	196315803, 186381054, 124578934, 150084553, 128031351, 118843410,
18122	107373060, 159565443, 173624887, 171292628, 198074235, 139074061,
18123	178690578, 144431052, 174262641, 176783005, 182214864, 162289361,
18124	192966929, 192033046, 169332843, 181580535, 164864073, 118444059,
18125	195496893, 153773183, 167266131, 130108623, 158802128, 180432893,
18126	144562140, 147978945, 142337360, 158506327, 104399819, 132635916,
18127	168734194, 136567839, 101281912, 120281622, 195003330, 112236091,
18128	185875592, 101959081, 122415367, 194990954, 148881099, 175891989,
18129	108115811, 163538891, 163394029, 123722049, 184837522, 142362091,
18130	100834097, 156679171, 100841679, 157022331, 178971071, 102928884,
18131	189701309, 195339954, 124415335, 106062584, 139214524, 133864640,
18132	134324406, 157317477, 155340540, 144810061, 177612569, 108474646,
18133	114329765, 143900008, 138265211, 145210162, 136643111, 197987319,
18134	102751191, 144121361, 169620456, 193602633, 161023559, 162140467,
18135	102901215, 167964187, 135746835, 187317233, 110047459, 163339773,

18136	124770449, 118885134, 141536376, 100915375, 164267438, 145016622,
18137	113937193, 106748706, 128815954, 164819775, 119220771, 102367432,
18138	189062690, 170911791, 194127762, 112245117, 123546771, 115640433,
18139	135772061, 166615646, 174474627, 130562291, 133320309, 153340551,
18140	138417181, 194605321, 150142632, 180008795, 151813296, 175497284,
18141	167018836, 157425342, 150169942, 131069156, 134310662, 160434122,
18142	105213831, 158797111, 150754540, 163290657, 102484886, 148697402,
18143	187203725, 198692811, 149360627, 140384233, 128749423, 132178578,
18144	177507355, 171857043, 178737969, 134023369, 102911446, 196144864,
18145	197697194, 134527467, 144296030, 189437192, 154052665, 188907106,
18146	162062575, 150993037, 199766583, 167936112, 181374511, 104971506,
18147	115378374, 135795558, 167972129, 135876446, 130937572, 103221320,
18148	124605656, 161129971, 131027586, 191128460, 143251843, 143269155,
18149	129284585, 173495971, 150425653, 199302112, 118494723, 121323805,
18150	116549802, 190991967, 168151180, 122483192, 151273721, 199792134,
18151	133106764, 121874844, 126215985, 112167639, 167793529, 182985195,
18152	185453921, 106957880, 158685312, 132775454, 133229161, 198905318,
18153	190537253, 191582222, 192325972, 178133427, 181825606, 148823337,
18154	160719681, 101448145, 131983362, 137910767, 112550175, 128826351,
18155	183649210, 135725874, 110356573, 189469487, 154446940, 118175923,
18156	106093708, 128146501, 185742532, 149692127, 164624247, 183221076,
18157	154737505, 168198834, 156410354, 158027261, 125228550, 131543250,
18158	139591848, 191898263, 104987591, 115406321, 103542638, 190012837,
18159	142615518, 178773183, 175862355, 117537850, 169565995, 170028011,
18160	158412588, 170150030, 117025916, 174630208, 142412449, 112839238,
18161	105257725, 114737141, 123102301, 172563968, 130555358, 132628403,
18162	183638157, 168682846, 143304568, 105994018, 170010719, 152092970,
18163	117799058, 132164175, 179868116, 158654714, 177489647, 116547948,
18164	183121404, 131836079, 184431405, 157311793, 149677763, 173989893,
18165	102277656, 107058530, 140837477, 152640947, 143507039, 152145247,
18166	101683884, 107090870, 161471944, 137225650, 128231458, 172995869,
18167	173831689, 171268519, 139042297, 111072135, 107569780, 137262545,
18168	181410950, 138270388, 198736451, 162848201, 180468288, 120582913,
18169	153390138, 135649144, 130040157, 106509887, 192671541, 174507066,
18170	186888783, 143805558, 135011967, 145862340, 180595327, 124727843,
18171	182925939, 157715840, 136885940, 198993925, 152416883, 178793572,
18172	179679516, 154076673, 192703125, 164187609, 162190243, 104699348,
18173	159891990, 160012977, 174692145, 132970421, 167781726, 115178506,
18174	153008552, 155999794, 102099694, 155431545, 127458567, 104403686,
18175	168042864, 184045128, 181182309, 179349696, 127218364, 192935516,
18176	120298724, 169583299, 148193297, 183358034, 159023227, 105261254,
18177	121144370, 184359584, 194433836, 138388317, 175184116, 108817112,
18178	151279233, 137457721, 193398208, 119005406, 132929377, 175306906,
18179	160741530, 149976826, 147124407, 176881724, 186734216, 185881509,
18180	191334220, 175930947, 117385515, 193408089, 157124410, 163472089,
18181	131949128, 180783576, 131158294, 100549708, 191802336, 165960770,
18182	170927599, 101052702, 181508688, 197828549, 143403726, 142729262,
18183	110348701, 139928688, 153550062, 106151434, 130786653, 196085995,
18184	100587149, 139141652, 106530207, 100852656, 124074703, 166073660,
18185	153338052, 163766757, 120188394, 197277047, 122215363, 138511354,
18186	183463624, 161985542, 159938719, 133367482, 104220974, 149956672,
18187	170250544, 164232439, 157506869, 159133019, 137469191, 142980999,
18188	134242305, 150172665, 121209241, 145596259, 160554427, 159095199,
18189	168243130, 184279693, 171132070, 121049823, 123819574, 171759855,

```

18190    119501864, 163094029, 175943631, 194450091, 191506160, 149228764,
18191    132319212, 197034460, 193584259, 126727638, 168143633, 109856853,
18192    127860243, 132141052, 133076065, 188414958, 158718197, 107124299,
18193    159592267, 181172796, 144388537, 196763139, 127431422, 179531145,
18194    100064922, 112650013, 132686230, 121550837,
18195    }

```

(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.

```

18196 \cs_new:Npn \fp_trig_large:ww #1, #2#3#4#5#6;
18197 {
18198     \exp_after:wN \fp_trig_large_auxi:w
18199     \int_value:w \fp_int_eval:w (#1 - 4) / 8 \exp_after:wN ,
18200     \int_value:w #1 , ;
18201     {#2}{#3}{#4}{#5} ;
18202 }
18203 \cs_new:Npn \fp_trig_large_auxi:w #1, #2,
18204 {
18205     \exp_after:wN \exp_after:wN
18206     \exp_after:wN \fp_trig_large_auxii:w
18207     \cs:w
18208         use_none:n \prg_replicate:nn { #2 - #1 * 8 } { n }
18209         \exp_after:wN
18210     \cs_end:
18211     \int_value:w
18212     \kernel_intarray_item:Nn \c_fp_trig_intarray
18213         { \fp_int_eval:w #1 + 1 \scan_stop: }
18214     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
18215     \kernel_intarray_item:Nn \c_fp_trig_intarray
18216         { \fp_int_eval:w #1 + 2 \scan_stop: }
18217     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
18218     \kernel_intarray_item:Nn \c_fp_trig_intarray
18219         { \fp_int_eval:w #1 + 3 \scan_stop: }
18220     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
18221     \kernel_intarray_item:Nn \c_fp_trig_intarray
18222         { \fp_int_eval:w #1 + 4 \scan_stop: }
18223     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
18224     \kernel_intarray_item:Nn \c_fp_trig_intarray
18225         { \fp_int_eval:w #1 + 5 \scan_stop: }
18226     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
18227     \kernel_intarray_item:Nn \c_fp_trig_intarray
18228         { \fp_int_eval:w #1 + 6 \scan_stop: }
18229     \exp_after:wN \fp_trig_large_auxiii:w \int_value:w
18230     \kernel_intarray_item:Nn \c_fp_trig_intarray
18231         { \fp_int_eval:w #1 + 7 \scan_stop: }

```

```

18232     \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
18233     \_kernel_intarray_item:Nn \c_fp_trig_intarray
18234     { \_fp_int_eval:w #1 + 8 \scan_stop: }
18235     \exp_after:wN \_fp_trig_large_auxiii:w \int_value:w
18236     \_kernel_intarray_item:Nn \c_fp_trig_intarray
18237     { \_fp_int_eval:w #1 + 9 \scan_stop: }
18238     \exp_stop_f:
18239   }
18240 \cs_new:Npn \_fp_trig_large_auxii:w
18241   {
18242     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
18243     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
18244     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
18245     \_fp_pack_twice_four:wNNNNNNNN \_fp_pack_twice_four:wNNNNNNNN
18246     \_fp_trig_large_auxv:www ;
18247   }
18248 \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:wmnnnnnn`  
`\_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.

```

18249 \cs_new:Npn \_fp_trig_large_auxv:www #1; #2; #3;
18250   {
18251     \exp_after:wN \_fp_use_i_until_s:nw
18252     \exp_after:wN \_fp_trig_large_auxvii:w
18253     \int_value:w \_fp_int_eval:w \c_fp_leading_shift_int
18254     \prg_replicate:nn { 13 }
18255       { \_fp_trig_large_auxvi:wnnnnnnn }
18256       + \c_fp_trailing_shift_int - \c_fp_middle_shift_int
18257     \_fp_use_i_until_s:nw
18258     ; #3 #1 ; ;
18259   }
18260 \cs_new:Npn \_fp_trig_large_auxvi:wnnnnnnn #1; #2#3#4#5#6#7#8#9
18261   {
18262     \exp_after:wN \_fp_trig_large_pack:NNNNNw
18263     \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
18264     + #2*#9 + #3*#8 + #4*#7 + #5*#6
18265     #1; {#2}{#3}{#4}{#5} {#7}{#8}{#9}
18266   }
18267 \cs_new:Npn \_fp_trig_large_pack:NNNNNw #1#2#3#4#5#6;
18268   { + #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.

```
18269 \cs_new:Npn \_fp_trig_large_auxvii:w #1#2#3
18270 {
18271     \exp_after:wN \_fp_trig_large_auxviii:ww
18272     \int_value:w \_fp_int_eval:w (#1#2#3 - 62) / 125 ;
18273     #1#2#3
18274 }
18275 \cs_new:Npn \_fp_trig_large_auxviii:ww #1;
18276 {
18277     + #1
18278     \if_int_odd:w #1 \exp_stop_f:
18279         \exp_after:wN \_fp_trig_large_auxix:Nw
18280         \exp_after:wN -
18281     \else:
18282         \exp_after:wN \_fp_trig_large_auxix:Nw
18283         \exp_after:wN +
18284     \fi:
18285 }
18286 \cs_new:Npn \_fp_trig_large_auxix:Nw
18287 {
18288     \exp_after:wN \_fp_use_i_until_s:nw
18289     \exp_after:wN \_fp_trig_large_auxxi:w
18290     \int_value:w \_fp_int_eval:w \c_fp_leading_shift_int
18291     \prg_replicate:nn { 13 }
18292         { \_fp_trig_large_auxx:wNNNNN }
18293         + \c_fp_trailing_shift_int - \c_fp_middle_shift_int
18294     ;
18295 }
18296 \cs_new:Npn \_fp_trig_large_auxx:wNNNNN #1; #2 #3#4#5#6
18297 {
18298     \exp_after:wN \_fp_trig_large_pack:NNNNNw
18299     \int_value:w \_fp_int_eval:w \c_fp_middle_shift_int
18300     #2 8 * #3#4#5#6
18301     #1; #2
18302 }
18303 \cs_new:Npn \_fp_trig_large_auxxi:w #1;
18304 {
18305     \exp_after:wN \_fp_ep_mul_raw:wwwwN
18306     \int_value:w \_fp_int_eval:w 0 \_fp_ep_to_ep_loop:N #1 ; ;
18307     0,{7853}{9816}{3397}{4483}{0961}{5661};
18308     \_fp_trig_small:ww
```

18309 }

(End definition for `\_fp_trig_large_auxvii:w` and others.)

### 32.1.6 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.

```

18310 \cs_new:Npn \_fp_sin_series_o:NNwww #1#2#3. #4;
18311 {
18312   \_fp_fixed_mul:wwn #4; #4;
18313 {
18314   \exp_after:wN \_fp_sin_series_aux_o:NNnwww
18315   \exp_after:wN #1
18316   \int_value:w
18317   \if_int_odd:w \_fp_int_eval:w (#3 + 2) / 4 \_fp_int_eval_end:
18318     #2
18319   \else:
18320     \if_meaning:w #2 0 2 \else: 0 \fi:
18321   \fi:
18322 {#3}
18323 }
18324 }
18325 \cs_new:Npn \_fp_sin_series_aux_o:NNnwww #1#2#3 #4; #5,#6;
18326 {
18327   \if_int_odd:w \_fp_int_eval:w #3 / 2 \_fp_int_eval_end:
18328     \exp_after:wN \use_i:nn
18329   \else:
18330     \exp_after:wN \use_ii:nn
18331   \fi:

```

```

18332 { % 1/18!
18333     \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0001}{5619}{2070};
18334                         #4;{0000}{0000}{0000}{0477}{9477}{3324};
18335     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0011}{4707}{4559}{7730};
18336     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{2087}{6756}{9878}{6810};
18337     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0027}{5573}{1922}{3985}{8907};
18338     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{2480}{1587}{3015}{8730}{1587};
18339     \__fp_fixed_mul_sub_back:wwwn #4;{0013}{8888}{8888}{8888}{8888}{8889};
18340     \__fp_fixed_mul_sub_back:wwwn #4;{0416}{6666}{6666}{6666}{6666}{6667};
18341     \__fp_fixed_mul_sub_back:wwwn #4;{5000}{0000}{0000}{0000}{0000}{0000};
18342     \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
18343     { \__fp_continue:wn 0, }
18344 }
18345 { % 1/17!
18346     \__fp_fixed_mul_sub_back:wwwn    {0000}{0000}{0000}{0028}{1145}{7254};
18347                         #4;{0000}{0000}{0000}{17647}{1637}{3182};
18348     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0000}{0160}{5904}{3836}{8216};
18349     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0002}{5052}{1083}{8544}{1719};
18350     \__fp_fixed_mul_sub_back:wwwn #4;{0000}{0275}{5731}{9223}{9858}{9065};
18351     \__fp_fixed_mul_sub_back:wwwn #4;{0001}{9841}{2698}{4126}{9841}{2698};
18352     \__fp_fixed_mul_sub_back:wwwn #4;{0083}{3333}{3333}{3333}{3333}{3333};
18353     \__fp_fixed_mul_sub_back:wwwn #4;{1666}{6666}{6666}{6666}{6666}{6667};
18354     \__fp_fixed_mul_sub_back:wwwn#4;{10000}{0000}{0000}{0000}{0000}{0000};
18355     { \__fp_ep_mul:wwwn 0, } #5,#6;
18356 }
18357 {
18358     \exp_after:wN \__fp_sanitize:Nw
18359     \exp_after:wN #2
18360     \int_value:w \__fp_int_eval:w #1
18361 }
18362 #2
18363 }

```

(End definition for `\__fp_sin_series_o>NNwww` and `\__fp_sin_series_aux_o>NNnwww`.)

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:wwwn`, 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.

```

18364 \cs_new:Npn \__fp_tan_series_o:NNwww #1#2#3. #4;
18365   {
18366     \__fp_fixed_mul:wwn #4; #4;
18367   {
18368     \exp_after:wN \__fp_tan_series_aux_o:Nnwww
18369     \int_value:w
18370     \if_int_odd:w \__fp_int_eval:w #3 / 2 \__fp_int_eval_end:
18371       \exp_after:wN \reverse_if:N
18372     \fi:
18373     \if_meaning:w #1#2 2 \else: 0 \fi:
18374   {#3}
18375   }
18376   }
18377 \cs_new:Npn \__fp_tan_series_aux_o:Nnwww #1 #2 #3; #4,#5;
18378   {
18379     \__fp_fixed_mul_sub_back:wwnn      {0000}{0000}{1527}{3493}{0856}{7059};
18380                               #3; {0000}{0159}{6080}{0274}{5257}{6472};
18381     \__fp_fixed_mul_sub_back:wwnn #3; {0002}{4571}{2320}{0157}{2558}{8481};
18382     \__fp_fixed_mul_sub_back:wwnn #3; {0115}{5830}{7533}{5397}{3168}{2147};
18383     \__fp_fixed_mul_sub_back:wwnn #3; {1929}{8245}{6140}{3508}{7719}{2982};
18384     \__fp_fixed_mul_sub_back:wwnn #3; {10000}{0000}{0000}{0000}{0000}{0000};
18385   { \__fp_ep_mul:wwnn 0, } #4,#5;
18386   {
18387     \__fp_fixed_mul_sub_back:wwnn      {0000}{0007}{0258}{0681}{9408}{4706};
18388                               #3; {0000}{2343}{7175}{1399}{6151}{7670};
18389     \__fp_fixed_mul_sub_back:wwnn #3; {0019}{2638}{4588}{9232}{8861}{3691};
18390     \__fp_fixed_mul_sub_back:wwnn #3; {0536}{6357}{0691}{4344}{6852}{4252};
18391     \__fp_fixed_mul_sub_back:wwnn #3; {5263}{1578}{9473}{6842}{1052}{6315};
18392     \__fp_fixed_mul_sub_back:wwnn#3; {10000}{0000}{0000}{0000}{0000}{0000};
18393   {
18394     \reverse_if:N \if_int_odd:w
18395       \__fp_int_eval:w (#2 - 1) / 2 \__fp_int_eval_end:
18396     \exp_after:wN \__fp_reverse_args:Nww
18397     \fi:
18398     \__fp_ep_div:wwnn 0,
18399   }
18400   }
18401   {
18402     \exp_after:wN \__fp_sanitize:Nw
18403     \exp_after:wN #1
18404     \int_value:w \__fp_int_eval:w \__fp_ep_to_float_o:wwN
18405   }
18406   #1
18407 }

```

(End definition for `\__fp_tan_series_o:NNwww` and `\__fp_tan_series_aux_o:Nnwww`.)

## 32.2 Inverse trigonometric functions

All inverse trigonometric functions (arcsine, arccosine, arctangent, arccotangent, arcosecant, and arcsecant) are based on a function often denoted `atan2`. This function 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 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.

### 32.2.1 Arctangent and arccotangent

```
\_\_fp_atan_o:Nw
\_\_fp_acot_o:Nw
\_\_fp_atan_default:w
```

The parsing step manipulates `atan` and `acot` like `min` and `max`, reading in an array of operands, but also leaves `\use_i:nn` or `\use_ii:nn` depending on whether the result 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  $\_\_fp\_parse\_function\_one\_two:nnw$ .

```

18408 \cs_new:Npn \_\_fp_atan_o:Nw #1
18409 {
18410     \_\_fp_parse_function_one_two:nnw
18411     { #1 { atan } { atand } }
18412     { \_\_fp_atan_default:w \_\_fp_atanii_o:Nww #1 }
18413 }
18414 \cs_new:Npn \_\_fp_acot_o:Nw #1
18415 {
18416     \_\_fp_parse_function_one_two:nnw
18417     { #1 { acot } { acotd } }
18418     { \_\_fp_atan_default:w \_\_fp_acotii_o:Nww #1 }
18419 }
18420 \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.)
```

$\_\_fp\_atanii\_o:Nww$   $\_\_fp\_acotii\_o:Nww$  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.

```

18421 \cs_new:Npn \_\_fp_atanii_o:Nww
18422     #1 \s__fp \_\_fp_chk:w #2#3#4; \s__fp \_\_fp_chk:w #5 #6 @
18423 {
18424     \if_meaning:w 3 #2 \_\_fp_case_return_i_o:ww \fi:
18425     \if_meaning:w 3 #5 \_\_fp_case_return_ii_o:ww \fi:
18426     \if_case:w
18427         \if_meaning:w #2 #5
18428             \if_meaning:w 1 #2 10 \else: 0 \fi:
18429         \else:
18430             \if_int_compare:w #2 > #5 \exp_stop_f: 1 \else: 2 \fi:
18431         \fi:
18432         \exp_stop_f:
18433             \_\_fp_case_return:nw { \_\_fp_atan_inf_o:NNNw #1 #3 2 }
18434             \or: \_\_fp_case_return:nw { \_\_fp_atan_inf_o:NNNw #1 #3 4 }
18435             \or: \_\_fp_case_return:nw { \_\_fp_atan_inf_o:NNNw #1 #3 0 }
18436         \fi:
18437         \_\_fp_atan_normal_o:NNnwNnw #1
18438         \s__fp \_\_fp_chk:w #2#3#4;
18439         \s__fp \_\_fp_chk:w #5 #6
18440     }
18441 \cs_new:Npn \_\_fp_acotii_o:Nww #1#2; #3;
18442     { \_\_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,  $\_\_fp\_atan\_combine\_o:NwwwwN$ , 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.

```

18443 \cs_new:Npn \__fp_atan_inf_o:NNNw #1#2#3 \s__fp \__fp_chk:w #4#5#6;
18444 {
18445   \exp_after:wN \__fp_atan_combine_o:NwwwwN
18446   \exp_after:wN #2
18447   \int_value:w \__fp_int_eval:w
18448     \if_meaning:w 2 #5 7 - \fi: #3 \exp_after:wN ;
18449   \c__fp_one_fixed_tl
18450   {0000}{0000}{0000}{0000}{0000}{0000};
18451   0,{0000}{0000}{0000}{0000}{0000}; #1
18452 }

```

(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.

```

18453 \cs_new_protected:Npn \__fp_atan_normal_o:NNnwNnw
18454   #1 \s__fp \__fp_chk:w 1#2#3#4; \s__fp \__fp_chk:w 1#5#6#7;
18455 {
18456   \__fp_atan_test_o:NwwNwwN
18457   #2 #3, #4{0000}{0000};
18458   #5 #6, #7{0000}{0000}; #1
18459 }

```

(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:NwwwwN` 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.

```

18460 \cs_new:Npn \__fp_atan_test_o:NwwNwwN #1#2,#3; #4#5,#6;
18461 {
18462   \exp_after:wN \__fp_atan_combine_o:NwwwwN
18463   \exp_after:wN #1
18464   \int_value:w \__fp_int_eval:w
18465     \if_meaning:w 2 #4
18466       7 - \__fp_int_eval:w
18467     \fi:
18468     \if_int_compare:w
18469       \__fp_ep_compare:www #2,#3; #5,#6; > 0 \exp_stop_f:
18470       3 -

```

```

18471      \exp_after:wN \__fp_reverse_args:Nww
18472      \fi:
18473      \__fp_atan_div:wnwwnw #2,#3; #5,#6;
18474  }

```

(End definition for `\__fp_atan_test_o:NwwNwwN.`)

```

\__fp_atan_div:wnwwnw
\__fp_atan_near:wwwn
\__fp_atan_near_aux:www

```

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.

```

18475 \cs_new:Npn \__fp_atan_div:wnwwnw #1,#2#3; #4,#5#6;
18476  {
18477      \if_int_compare:w
18478          \__fp_int_eval:w 41421 * #5 < #2 000
18479          \if_case:w \__fp_int_eval:w #4 - #1 \__fp_int_eval_end:
18480              00 \or: 0 \fi:
18481          \exp_stop_f:
18482          \exp_after:wN \__fp_atan_near:wwwn
18483      \fi:
18484      0
18485      \__fp_ep_div:wwwn #1,{#2}#3; #4,{#5}#6;
18486      \__fp_atan_auxi:ww
18487  }
18488 \cs_new:Npn \__fp_atan_near:wwwn
18489  0 \__fp_ep_div:wwwn #1,#2; #3,
18490  {
18491      1
18492      \__fp_ep_to_fixed:wwwn #1 - #3, #2;
18493      \__fp_atan_near_aux:wwwn
18494  }
18495 \cs_new:Npn \__fp_atan_near_aux:wwwn #1; #2;
18496  {
18497      \__fp_fixed_add:wwwn #1; #2;
18498      { \__fp_fixed_sub:wwwn #2; #1; { \__fp_ep_div:wwwn 0, } 0, }
18499  }

```

(End definition for `\__fp_atan_div:wnwwnw`, `\__fp_atan_near:wwwn`, and `\__fp_atan_near_aux:wwwn`.)

```

\__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 of  $z$ .

```

18500 \cs_new:Npn \__fp_atan_auxi:ww #1,#2;
18501  { \__fp_ep_to_fixed:wwwn #1,#2; \__fp_atan_auxii:w #1,#2; }
18502 \cs_new:Npn \__fp_atan_auxii:w #1;
18503  {
18504      \__fp_fixed_mul:wwwn #1; #1;
18505  {
18506      \__fp_atan_Taylor_loop:www 39 ;
18507      {0000}{0000}{0000}{0000}{0000}{0000} ;

```

```

18508      }
18509      ! #1;
18510  }

(End definition for \_fp_atan_auxi:ww and \_fp_atan_auxii:w.)
```

`\_fp_atan_Taylor_loop:www`  
`\_fp_atan_Taylor_break: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.

```

18511 \cs_new:Npn \_fp_atan_Taylor_loop:www #1; #2; #3;
18512 {
18513     \if_int_compare:w #1 = -1 \exp_stop_f:
18514         \_fp_atan_Taylor_break:w
18515     \fi:
18516     \exp_after:wN \_fp_fixed_div_int:wwN \c_fp_one_fixed_tl #1;
18517     \_fp_rrot:www \_fp_fixed_mul_sub_back:wwn #2; #3;
18518 {
18519     \exp_after:wN \_fp_atan_Taylor_loop:www
18520     \int_value:w \_fp_int_eval:w #1 - 2 ;
18521 }
18522 #3;
18523 }
18524 \cs_new:Npn \_fp_atan_Taylor_break:w
18525     \fi: #1 \_fp_fixed_mul_sub_back:wwn #2; #3 !
18526 { \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`.

```

18527 \cs_new:Npn \_fp_atan_combine_o:NwwwwN #1 #2; #3; #4; #5; #6; #7
18528 {
18529     \exp_after:wN \_fp_sanitize:Nw
18530     \exp_after:wN #1
18531     \int_value:w \_fp_int_eval:w
18532     \if_meaning:w 0 #2
18533         \exp_after:wN \use_i:nn
18534     \else:
```

```

18535           \exp_after:wN \use_i:nn
18536   \fi:
18537   { #5 \fp_fixed_mul:wwn #3; #6; }
18538   {
18539     \fp_fixed_mul:wwn #3; #4;
18540   {
18541     \exp_after:wN \fp atan_combine_aux:ww
18542     \int_value:w \fp_int_eval:w #2 / 2 ; #2;
18543   }
18544 }
18545 { #7 \fp_fixed_to_float_o:wn \fp_fixed_to_float_rad_o:wn }
18546 #1
18547 }
18548 \cs_new:Npn \fp atan_combine_aux:ww #1; #2;
18549 {
18550   \fp_fixed_short:wn
18551   {7853}{9816}{3397}{4483}{0961}{5661};
18552   {#1}{0000}{0000};
18553 {
18554   \if_int_odd:w #2 \exp_stop_f:
18555     \exp_after:wN \fp_fixed_sub:wn
18556   \else:
18557     \exp_after:wN \fp_fixed_add:wn
18558   \fi:
18559 }
18560 }

```

(End definition for `\fp atan_combine_o:NwwwwN` and `\fp atan_combine_aux:ww`.)

### 32.2.2 Arcsine and arccosine

`\fp_asin_o:w` Again, the first argument provided by `\fp-parse` is `\use_i:nn` if we are to work in radians and `\use_i: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).

```

18561 \cs_new:Npn \fp_asin_o:w #1 \s_fp \fp_chk:w #2#3; @
18562 {
18563   \if_case:w #2 \exp_stop_f:
18564     \fp_case_return_same_o:w
18565   \or:
18566     \fp_case_use:nw
18567     { \fp_asin_normal_o:NfwNnnnnw #1 { #1 { asin } { asind } } }
18568   \or:
18569     \fp_case_use:nw
18570     { \fp_invalid_operation_o:fw { #1 { asin } { asind } } }
18571   \else:
18572     \fp_case_return_same_o:w
18573   \fi:
18574   \s_fp \fp_chk:w #2 #3;
18575 }

```

(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.

```
18576 \cs_new:Npn \_fp_acos_o:w #1 \s_fp \_fp_chk:w #2#3; @
18577 {
18578     \if_case:w #2 \exp_stop_f:
18579         \_fp_case_use:nw { \_fp_atan_inf_o:NNNw #1 0 4 }
18580     \or:
18581         \_fp_case_use:nw
18582             {
18583                 \_fp_asin_normal_o:NfwNnnnnw #1 { #1 { acos } { acosd } }
18584                     \_fp_reverse_args:Nww
18585             }
18586     \or:
18587         \_fp_case_use:nw
18588             { \_fp_invalid_operation_o:fw { #1 { acos } { acosd } } }
18589     \else:
18590         \_fp_case_return_same_o:w
18591     \fi:
18592     \s_fp \_fp_chk:w #2 #3;
18593 }
```

(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.

```
18594 \cs_new:Npn \_fp_asin_normal_o:NfwNnnnnw
18595     #1#2#3 \s_fp \_fp_chk:w 1#4#5#6#7#8#9;
18596 {
18597     \if_int_compare:w #5 < 1 \exp_stop_f:
18598         \exp_after:wN \_fp_use_none_until_s:w
18599     \fi:
18600     \if_int_compare:w \_fp_int_eval:w #5 + #6#7 + #8#9 = 1000 0001 ~
18601         \exp_after:wN \_fp_use_none_until_s:w
18602     \fi:
18603     \_fp_use_i:ww
18604     \_fp_invalid_operation_o:fw {#2}
18605         \s_fp \_fp_chk:w 1#4{#5}{#6}{#7}{#8}{#9};
18606     \_fp_asin_auxi_o:NnNww
18607         #1 {#3} #4 #5,{#6}{#7}{#8}{#9}{0000}{0000};
18608 }
```

(End definition for \\_fp\_asin\_normal\_o:NfwNnnnnw.)

\\_fp\_asin\_auxi\_o:NnNww  
\\_fp\_asin\_isqrt:wn

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`.

```

18609 \cs_new:Npn \_fp_asin_auxi_o:NnNww #1#2#3#4,#5;
18610 {
18611     \_fp_ep_to_fixed:wwn #4,#5;
18612     \_fp_asin_isqrt:wn
18613     \_fp_ep_mul:wwwn #4,#5;
18614     \_fp_ep_to_ep:wwN
18615     \_fp_fixed_continue:wn
18616     { #2 \_fp_atan_test_o:NwwNwwN #3 }
18617     0 1,{1000}{0000}{0000}{0000}{0000}; #1
18618 }
18619 \cs_new:Npn \_fp_asin_isqrt:wn #1;
18620 {
18621     \exp_after:wN \_fp_fixed_sub:wwn \c_fp_one_fixed_tl #1;
18622 {
18623     \_fp_fixed_add_one:wN #1;
18624     \_fp_fixed_continue:wn { \_fp_ep_mul:wwwn 0, } 0,
18625 }
18626     \_fp_ep_isqrt:wwn
18627 }
```

(End definition for `\_fp_asin_auxi_o:NnNww` and `\_fp_asin_isqrt:wn`.)

### 32.2.3 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.

```

18628 \cs_new:Npn \_fp_acsc_o:w #1 \s_fp \_fp_chk:w #2#3#4; @
18629 {
18630     \if_case:w \if_meaning:w 2 #2 #3 \fi: #2 \exp_stop_f:
18631         \_fp_case_use:nw
18632         { \_fp_invalid_operation_o:fw { #1 { acsc } { acscl } } }
18633     \or: \_fp_case_use:nw
18634         { \_fp_acsc_normal_o:NfwNnw #1 { #1 { acsc } { acscl } } }
18635     \or: \_fp_case_return_o:Nw \c_zero_fp
18636     \or: \_fp_case_return_same_o:w
18637     \else: \_fp_case_return_o:Nw \c_minus_zero_fp
18638     \fi:
18639     \s_fp \_fp_chk:w #2 #3 #4;
18640 }
```

(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`.

```

18641 \cs_new:Npn \_fp_asec_o:w #1 \s__fp \_fp_chk:w #2#3; @
18642   {
18643     \if_case:w #2 \exp_stop_f:
18644       \_fp_case_use:nw
18645         { \_fp_invalid_operation_o:fw { #1 { asec } { asecd } } }
18646     \or:
18647       \_fp_case_use:nw
18648         {
18649           \_fp_acsc_normal_o:NfwNnw #1 { #1 { asec } { asecd } }
18650             \_fp_reverse_args:Nww
18651         }
18652     \or: \_fp_case_use:nw { \_fp_atan_inf_o>NNNw #1 0 4 }
18653     \else: \_fp_case_return_same_o:w
18654     \fi:
18655     \s__fp \_fp_chk:w #2 #3;
18656   }

```

(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  $\text{asec}(x) = \text{acos}(1/x)$ .

```

18657 \cs_new:Npn \_fp_acsc_normal_o:NfwNnw #1#2#3 \s__fp \_fp_chk:w 1#4#5#6;
18658   {
18659     \int_compare:nNnTF {#5} < 1
18660     {
18661       \_fp_invalid_operation_o:fw {#2}
18662         \s__fp \_fp_chk:w 1#4{#5}#6;
18663     }
18664     {
18665       \_fp_ep_div:wwwn
18666         1,{1000}{0000}{0000}{0000}{0000}{0000};
18667         #5,#6{0000}{0000};
18668         { \_fp_asin_auxi_o:NnNww #1 {#3} #4 }
18669     }
18670   }

```

(End definition for `\_fp_acsc_normal_o:NfwNnw`.)

18671 ⟨/initex | package⟩

## 33 13fp-convert implementation

```

18672 (*initex | package)
18673 (@@=fp)

```

### 33.1 Dealing with tuples

The first argument is for instance `\_fp_to_tl_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 `,`.

```

\_fp_tuple_convert:Nw
\_fp_tuple_convert_loop:nNw
\_fp_tuple_convert_end:w

```

```

18674 \cs_new:Npn \__fp_tuple_convert:Nw #1 \s_fp_tuple \__fp_tuple_chk:w #2 ;
18675   {
18676     \int_case:nnF { \__fp_array_count:n {#2} }
18677     {
18678       { 0 } { () }
18679       { 1 } { \__fp_tuple_convert_end:w @ { #1 #2 , } }
18680     }
18681   {
18682     \__fp_tuple_convert_loop:nNw { } #1
18683       #2 { ? \__fp_tuple_convert_end:w } ;
18684       @ { \use_none:nn }
18685     }
18686   }
18687 \cs_new:Npn \__fp_tuple_convert_loop:nNw #1#2#3#4; #5 @ #6
18688   {
18689     \use_none:n #3
18690     \exp_args:Nf \__fp_tuple_convert_loop:nNw { #2 #3#4 ; } #2 #5
18691       @ { #6 , ~ #1 }
18692   }
18693 \cs_new:Npn \__fp_tuple_convert_end:w #1 @ #2
18694   { \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.)

```

### 33.2 Trimming trailing zeros

If #1 ends with a 0, the loop auxiliary takes that zero as an end-delimiter for its first argument, and the second argument is the same loop auxiliary. Once the last trailing 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.

```

18695 \cs_new:Npn \__fp_trim_zeros:w #1 ;
18696   {
18697     \__fp_trim_zeros_loop:w #1
18698       ; \__fp_trim_zeros_loop:w 0; \__fp_trim_zeros_dot:w .; \s_stop
18699   }
18700 \cs_new:Npn \__fp_trim_zeros_loop:w #1 0; #2 { #2 #1 ; #2 }
18701 \cs_new:Npn \__fp_trim_zeros_dot:w #1 .; { \__fp_trim_zeros_end:w #1 ; }
18702 \cs_new:Npn \__fp_trim_zeros_end:w #1 ; #2 \s_stop { #1 }

(End definition for \__fp_trim_zeros:w and others.)

```

### 33.3 Scientific notation

**\fp\_to\_scientific:N** The three public functions evaluate their argument, then pass it to \\_\_fp\_to\_scientific\_dispatch:w.

```

\fp_to_scientific:c
18703 \cs_new:Npn \fp_to_scientific:N #1
18704   { \exp_after:wN \__fp_to_scientific_dispatch:w #1 }
18705 \cs_generate_variant:Nn \fp_to_scientific:N { c }
18706 \cs_new:Npn \fp_to_scientific:n
18707   {
18708     \exp_after:wN \__fp_to_scientific_dispatch:w
18709     \exp:w \exp_end_continue_f:w \__fp_parse:n
18710   }

```

(End definition for `\fp_to_scientific:N` and `\fp_to_scientific:n`. These functions are documented on page 187.)

We allow tuples.

```

\__fp_to_scientific_dispatch:w
  \__fp_to_scientific_recover:w
\__fp_tuple_to_scientific:w
18711  \cs_new:Npn \__fp_to_scientific_dispatch:w #1
18712    {
18713      \__fp_change_func_type:NNN
18714        #1 \__fp_to_scientific:w \__fp_to_scientific_recover:w
18715        #1
18716    }
18717  \cs_new:Npn \__fp_to_scientific_recover:w #1 #2 ;
18718    {
18719      \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
18720      nan
18721    }
18722  \cs_new:Npn \__fp_tuple_to_scientific:w
18723    { \__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.

```

18724  \cs_new:Npn \__fp_to_scientific:w \s__fp \__fp_chk:w #1#2
18725    {
18726      \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
18727      \if_case:w #1 \exp_stop_f:
18728        \__fp_case_return:nw { 0.00000000000000e0 }
18729        \or: \exp_after:wN \__fp_to_scientific_normal:wnnnnn
18730        \or:
18731          \__fp_case_use:nw
18732            {
18733              \__fp_invalid_operation:nnw
18734                { \fp_to_scientific:N \c__fp_overflowing_fp }
18735                { fp_to_scientific }
18736            }
18737        \or:
18738          \__fp_case_use:nw
18739            {
18740              \__fp_invalid_operation:nnw
18741                { \fp_to_scientific:N \c_zero_fp }
18742                { fp_to_scientific }
18743            }
18744          \fi:
18745          \s__fp \__fp_chk:w #1 #2
18746        }
18747  \cs_new:Npn \__fp_to_scientific_normal:wnnnnn
18748    \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
```

```

18749   {
18750     \exp_after:wN \_fp_to_scientific_normal:wNw
18751     \exp_after:wN e
18752     \int_value:w \_fp_int_eval:w #2 - 1
18753     ; #3 #4 #5 #6 ;
18754   }
18755 \cs_new:Npn \_fp_to_scientific_normal:wNw #1 ; #2#3;
18756   { #2.#3 #1 }

(End definition for \_fp_to_scientific:w, \_fp_to_scientific_normal:wnnnnn, and \_fp_to-
scientific_normal:wNw.)

```

### 33.4 Decimal representation

\fp\_to\_decimal:N  
\fp\_to\_decimal:c  
\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.

```

18757 \cs_new:Npn \fp_to_decimal:N #1
18758   { \exp_after:wN \_fp_to_decimal_dispatch:w #1 }
18759 \cs_generate_variant:Nn \fp_to_decimal:N { c }
18760 \cs_new:Npn \fp_to_decimal:n
18761   {
18762     \exp_after:wN \_fp_to_decimal_dispatch:w
18763     \exp:w \exp_end_continue_f:w \_fp_parse:n
18764   }

(End definition for \fp_to_decimal:N and \fp_to_decimal:n. These functions are documented on page
187.)

```

We allow tuples.

```

18765 \cs_new:Npn \_fp_to_decimal_dispatch:w #1
18766   {
18767     \_fp_change_func_type:NNN
18768     #1 \_fp_to_decimal:w \_fp_to_decimal_recover:w
18769     #1
18770   }
18771 \cs_new:Npn \_fp_to_decimal_recover:w #1 #2 ;
18772   {
18773     \_fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
18774     nan
18775   }
18776 \cs_new:Npn \_fp_tuple_to_decimal:w
18777   { \_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.)

```

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.

```

18778 \cs_new:Npn \__fp_to_decimal:w \s__fp \__fp_chk:w #1#2
18779 {
18780     \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
18781     \if_case:w #1 \exp_stop_f:
18782         \__fp_case_return:nw { 0 }
18783     \or: \exp_after:wN \__fp_to_decimal_normal:wnnnnn
18784     \or:
18785         \__fp_case_use:nw
18786         {
18787             \__fp_invalid_operation:nnw
18788                 { \fp_to_decimal:N \c__fp_overflowing_fp }
18789                 { fp_to_decimal }
18790             }
18791     \or:
18792         \__fp_case_use:nw
18793         {
18794             \__fp_invalid_operation:nnw
18795                 { 0 }
18796                 { fp_to_decimal }
18797             }
18798     \fi:
18799     \s__fp \__fp_chk:w #1 #2
1880 }
18801 \cs_new:Npn \__fp_to_decimal_normal:wnnnnn
18802     \s__fp \__fp_chk:w 1 #1 #2 #3#4#5#6 ;
18803 {
18804     \int_compare:nNnTF {#2} > 0
18805     {
18806         \int_compare:nNnTF {#2} < \c__fp_prec_int
18807         {
18808             \__fp_decimate:nNnnnn { \c__fp_prec_int - #2 }
18809                 \__fp_to_decimal_large:Nnnw
18810             }
18811             {
18812                 \exp_after:wN \exp_after:wN
18813                     \exp_after:wN \__fp_to_decimal_huge:wnnnn
18814                     \prg_replicate:nn { #2 - \c__fp_prec_int } { 0 } ;
18815             }
18816             {#3} {#4} {#5} {#6}
18817         }
18818         {
18819             \exp_after:wN \__fp_trim_zeros:w
18820             \exp_after:wN 0
18821             \exp_after:wN .
18822             \exp:w \exp_end_continue_f:w \prg_replicate:nn { - #2 } { 0 }
18823             #3#4#5#6 ;
18824         }
18825     }
18826 \cs_new:Npn \__fp_to_decimal_large:Nnnw #1#2#3#4;
18827 {
18828     \exp_after:wN \__fp_trim_zeros:w \int_value:w
18829         \if_int_compare:w #2 > 0 \exp_stop_f:
18830             #2
18831         \fi:

```

```

18832      \exp_stop_f:
18833      #3.#4 ;
18834  }
18835 \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.)

```

### 33.5 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

```

18836 \cs_new:Npn \fp_to_tl:N #1 { \exp_after:wN \__fp_to_tl_dispatch:w #1 }
18837 \cs_generate_variant:Nn \fp_to_tl:N { c }
18838 \cs_new:Npn \fp_to_tl:n
18839  {
18840      \exp_after:wN \__fp_to_tl_dispatch:w
18841      \exp:w \exp_end_continue_f:w \__fp_parse:n
18842  }

```

(End definition for \fp\_to\_tl:N and \fp\_to\_tl:n. These functions are documented on page 187.)

\\_\_fp\_to\_tl\_dispatch:w We allow tuples.

```

18843 \cs_new:Npn \__fp_to_tl_dispatch:w #1
18844  { \__fp_change_func_type:NNN #1 \__fp_to_tl:w \__fp_to_tl_recover:w #1 }
18845 \cs_new:Npn \__fp_to_tl_recover:w #1 #2 ;
18846  {
18847      \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
18848      nan
18849  }
18850 \cs_new:Npn \__fp_tuple_to_tl:w
18851  { \__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\_dispatch:w, but without the “invalid operation” exception. First filter special cases. \\_\_fp\_to\_tl\_normal:nnnnn We express normal numbers in decimal notation if the exponent is in the range  $[-2, 16]$ , and otherwise use scientific notation.

```

18852 \cs_new:Npn \__fp_to_tl:w \s_fp \__fp_chk:w #1#2
18853  {
18854      \if_meaning:w 2 #2 \exp_after:wN - \exp:w \exp_end_continue_f:w \fi:
18855      \if_case:w #1 \exp_stop_f:
18856          \__fp_case_return:nw { 0 }
18857      \or: \exp_after:wN \__fp_to_tl_normal:nnnnn
18858      \or: \__fp_case_return:nw { inf }
18859      \else: \__fp_case_return:nw { nan }
18860      \fi:
18861  }
18862 \cs_new:Npn \__fp_to_tl_normal:nnnnn #1
18863  {
18864      \int_compare:nTF
18865      { -2 <= #1 <= \c_fp_prec_int }
18866      { \__fp_to_decimal_normal:wnnnnn }
18867      { \__fp_to_tl_scientific:wnnnnn }

```

```

18868      \s_fp \fp_chk:w 1 0 {#1}
18869    }
18870  \cs_new:Npn \fp_to_tl_scientific:wnnnnn
18871    \s_fp \fp_chk:w 1 #1 #2 #3#4#5#6 ;
18872  {
18873    \exp_after:wN \fp_to_tl_scientific:wNw
18874    \exp_after:wN e
18875    \int_value:w \fp_int_eval:w #2 - 1
18876    ; #3 #4 #5 #6 ;
18877  }
18878 \cs_new:Npn \fp_to_tl_scientific:wNw #1 ; #2#3;
18879  { \fp_trim_zeros:w #2.#3 ; #1 }

(End definition for \fp_to_tl:w and others.)

```

### 33.6 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.

### 33.7 Convert to dimension or integer

**\fp\_to\_dim:N** **\fp\_to\_dim:c** **\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, not tuples.

```

\fp_to_dim_dispatch:w
\fp_to_dim_recover:w
\fp_to_dim:w
18880 \cs_new:Npn \fp_to_dim:N #1
18881  { \exp_after:wN \fp_to_dim_dispatch:w #1 }
18882 \cs_generate_variant:Nn \fp_to_dim:N { c }
18883 \cs_new:Npn \fp_to_dim:n
18884  {
18885    \exp_after:wN \fp_to_dim_dispatch:w
18886    \exp:w \exp_end_continue_f:w \fp_parse:n
18887  }
18888 \cs_new:Npn \fp_to_dim_dispatch:w #1#2 ;
18889  {
18890    \fp_change_func_type:NNN #1 \fp_to_dim:w \fp_to_dim_recover:w
18891    #1 #2 ;
18892  }
18893 \cs_new:Npn \fp_to_dim_recover:w #1
18894  { \fp_invalid_operation:nnw { Opt } { fp_to_dim } }
18895 \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 187.)

```

**\fp\_to\_int:N** **\fp\_to\_int:c** **\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
18896 \cs_new:Npn \fp_to_int:N #1 { \exp_after:wN \fp_to_int_dispatch:w #1 }
18897 \cs_generate_variant:Nn \fp_to_int:N { c }
18898 \cs_new:Npn \fp_to_int:n
18899  {


```

```

18900      \exp_after:wN \_fp_to_int_dispatch:w
18901      \exp:w \exp_end_continue_f:w \_fp_parse:n
18902  }
18903 \cs_new:Npn \_fp_to_int_dispatch:w #1#2 ;
18904  {
18905      \_fp_change_func_type:NNN #1 \_fp_to_int:w \_fp_to_int_recover:w
18906      #1 #2 ;
18907  }
18908 \cs_new:Npn \_fp_to_int_recover:w #1
18909  { \_fp_invalid_operation:nnw { 0 } { fp_to_int } }
18910 \cs_new:Npn \_fp_to_int:w #1;
18911  {
18912      \exp_after:wN \_fp_to_decimal:w \exp:w \exp_end_continue_f:w
18913      \_fp_round:Nnw \_fp_round_to_nearest:NNN #1; { 0 }
18914  }

```

(End definition for `\fp_to_int:N` and others. These functions are documented on page 187.)

### 33.8 Convert from a dimension

```

\dim_to_fp:n
\_fp_from_dim_test:ww
    \_fp_from_dim:wNw
\_fp_from_dim:wNNnnnnnn
\_fp_from_dim:wnnnnnNw

```

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 `\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.

```

18915 \_kernel_patch_args:nNNpn { { (#1) } }
18916 \cs_new:Npn \dim_to_fp:n #1
18917  {
18918      \exp_after:wN \_fp_from_dim_test:ww
18919      \exp_after:wN 0
18920      \exp_after:wN ,
18921      \int_value:w \tex_glueexpr:D #1 ;
18922  }
18923 \cs_new:Npn \_fp_from_dim_test:ww #1, #2
18924  {
18925      \if_meaning:w 0 #2
18926          \_fp_case_return:nw { \exp_after:wN \c_zero_fp }
18927      \else:
18928          \exp_after:wN \_fp_from_dim:wNw
18929          \int_value:w \_fp_int_eval:w #1 - 4
18930          \if_meaning:w - #2
18931              \exp_after:wN , \exp_after:wN 2 \int_value:w
18932          \else:
18933              \exp_after:wN , \exp_after:wN 0 \int_value:w #2
18934          \fi:
18935      \fi:
18936  }
18937 \cs_new:Npn \_fp_from_dim:wNw #1,#2#3;
18938  {

```

```

18939      \_\_fp\_pack\_twice\_four:wNNNNNNNN \_\_fp\_from\_dim:wNNnnnnnn ;
18940      #3 000 0000 00 {10}987654321; #2 {#1}
18941      }
18942 \cs_new:Npn \_\_fp\_from\_dim:wNNnnnnnn #1; #2#3#4#5#6#7#8#9
18943  { \_\_fp\_from\_dim:wnnnnwNn #1 {#2#300} {0000} ; }
18944 \cs_new:Npn \_\_fp\_from\_dim:wnnnnwNn #1; #2#3#4#5#6; #7#8
18945  {
18946      \_\_fp\_mul\_npos\_o:Nww #7
18947      \s\_fp \_\_fp\_chk:w 1 #7 {#5} #1 ;
18948      \s\_fp \_\_fp\_chk:w 1 0 {#8} {1525} {8789} {0625} {0000} ;
18949      \prg\_do\_nothing:
18950  }

```

(End definition for `\dim_to_fp:n` and others. This function is documented on page 161.)

### 33.9 Use and eval

**\fp\_use:N** Those public functions are simple copies of the decimal conversions.

```

\fp_use:c 18951 \cs_new_eq:NN \fp_use:N \fp_to_decimal:N
\fp_eval:n 18952 \cs_generate_variant:Nn \fp_use:N { c }
18953 \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 188.)

**\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.

```

18954 \cs_new:Npn \fp_abs:n #1
18955  { \fp_to_decimal:n { abs \_\_fp_parse:n {#1} } }

```

(End definition for `\fp_abs:n`. This function is documented on page 202.)

**\fp\_max:nn** Similar to `\fp_abs:n`, for consistency with `\int_max:nn`, etc.

```

\fp_min:nn 18956 \cs_new:Npn \fp_max:nn #1#2
18957  { \fp_to_decimal:n { max ( \_\_fp_parse:n {#1} , \_\_fp_parse:n {#2} ) } }
18958 \cs_new:Npn \fp_min:nn #1#2
18959  { \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 202.)

### 33.10 Convert an array of floating points to a comma list

`\_\_fp_array_to_clist:n` 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.

```

18960 \cs_new:Npn \_fp_array_to_clist:n #1
18961   {
18962     \tl_if_empty:nF {#1}
18963     {
18964       \_fp_expand:n
18965       {
18966         { \use_ii:nn }
18967         \_fp_array_to_clist_loop:Nw #1 { ? \prg_break: } ;
18968         \prg_break_point:
18969       }
18970     }
18971   }
18972 \cs_new:Npx \_fp_array_to_clist_loop:Nw #1#2;
18973   {
18974     \exp_not:N \use_none:n #1
18975     \exp_not:N \exp_after:wN
18976     {
18977       \exp_not:N \exp_after:wN ,
18978       \exp_not:N \exp_after:wN \c_space_tl
18979       \exp_not:N \exp:w
18980       \exp_not:N \exp_end_continue_f:w
18981       \exp_not:N \_fp_to_tl_dispatch:w #1 #2 ;
18982     }
18983     \exp_not:N \_fp_array_to_clist_loop:Nw
18984   }

```

(End definition for `\_fp_array_to_clist:n` and `\_fp_array_to_clist_loop:Nw`.)

18985 ⟨/initex | package⟩

## 34 I3fp-random Implementation

```

18986 ⟨*initex | package⟩
18987 ⟨@=fp⟩

```

`\_fp_parse_word_rand:N` Those functions may receive a variable number of arguments. We won't use the argument `?`.

```

18988 \cs_new:Npn \_fp_parse_word_rand:N
18989   { \_fp_parse_function:NNN \_fp_rand_o:Nw ? }
18990 \cs_new:Npn \_fp_parse_word_randint:N
18991   { \_fp_parse_function:NNN \_fp_randint_o:Nw ? }

```

(End definition for `\_fp_parse_word_rand:N` and `\_fp_parse_word_randint:N`.)

### 34.1 Engine support

Most engines provide random numbers, but not all. We write the test twice simply in order to write the `false` branch first.

```

18992 \sys_if_rand_exist:F
18993   {
18994     \_kernel_msg_new:nnn { kernel } { fp-no-random }

```

```

18995 { Random~numbers~unavailable~for~#1 }
18996 \cs_new:Npn \_fp_rand_o:Nw ? #1 @
18997 {
18998     \_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
18999     { fp-rand }
19000     \exp_after:wN \c_nan_fp
19001 }
19002 \cs_new_eq:NN \_fp_randint_o:Nw \_fp_rand_o:Nw
19003 \cs_new:Npn \int_rand:nn #1#2
19004 {
19005     \_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
19006     { \int_rand:nn {#1} {#2} }
19007     \int_eval:n {#1}
19008 }
19009 \cs_new:Npn \int_rand:n #1
19010 {
19011     \_kernel_msg_expandable_error:nnn { kernel } { fp-no-random }
19012     { \int_rand:n {#1} }
19013     1
19014 }
19015 }
19016 \sys_if_rand_exist:T
19017 {

```

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 **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  $\varepsilon$ -**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  $\text{ediv}(R \text{random}(2^{14}) + \text{random}(R) + 2^{13}, 2^{14}) - 1 + \langle \text{min} \rangle$ . The shifts by  $2^{13}$  and  $-1$  convert  $\varepsilon$ -TeX division to truncated division. The bound on  $R$  ensures that the number obtained after the shift is less than  $\text{\c_max_int}$ . 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  $\langle \text{min} \rangle + R_2 X_1 2^{14} + (R_2 Y_1 + R_1 X_1) + \text{ediv}(R_2 Y_0 + R_1 Y_1 + R_0 X_1 + \text{ediv}(R_2 X_0 + R_0 Y_1 + \text{ediv}((2^{14} R_1 + R_0)(2^{14} Y_0 + X_0), 2^{28}), 2^{14}), 2^{14})$  then map a result of  $\langle \text{max} \rangle + 1$  to  $\langle \text{min} \rangle$ . Writing each  $\text{ediv}$  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  $\langle \text{exact} \rangle = \langle \text{min} \rangle + R \times 0.X_1 Y_1 Y_0 X_0$ . Given we map  $\langle \text{max} \rangle + 1$  to  $\langle \text{min} \rangle$ , 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  $\langle \text{max} \rangle + 1$  with  $\langle \text{max} \rangle = 2^{31} - 1$  (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  $\langle \text{second} \rangle = R_2 2^{14} + R_0 + R_2 2^{14}$  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.

```
19018     \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  $p = 0$ .

```
19019     \cs_new:Npn \_\_kernel_randint:n #1
19020     {
19021         (#1 * \tex_uniformdeviate:D 16384
19022             + \tex_uniformdeviate:D #1 + 8192 ) / 16384
19023     }
```

(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.

```

19024   \cs_new:Npn \__fp_rand_myriads:n #1
19025     { \__fp_rand_myriads_loop:w #1 \prg_break: X \prg_break_point: ; }
19026   \cs_new:Npn \__fp_rand_myriads_loop:w #1 X
19027     {
19028       #1
19029       \exp_after:wN \__fp_rand_myriads_get:w
19030       \int_value:w \__fp_int_eval:w 9999 +
19031         \__kernel_randint:n { 10000 }
19032         \__fp_rand_myriads_loop:w
19033     }
19034   \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.)

```

## 34.2 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.

```

19035   \cs_new:Npn \__fp_rand_o:Nw ? #1 @
19036     {
19037       \tl_if_empty:nTF {#1}
19038         {
19039           \exp_after:wN \__fp_rand_o:w
19040           \exp:w \exp_end_continue_f:w
19041           \__fp_rand_myriads:n { XXXX } { 0000 } { 0000 } ; 0
19042         }
19043         {
19044           \__kernel_msg_expandable_error:nnnnn
19045             { kernel } { fp-num-args } { rand() } { 0 } { 0 }
19046             \exp_after:wN \c_nan_fp
19047         }
19048     }
19049   \cs_new:Npn \__fp_rand_o:w ;
19050     {
19051       \exp_after:wN \__fp_sanitize:Nw
19052       \exp_after:wN 0
19053       \int_value:w \__fp_int_eval:w \c_zero_int
19054       \__fp_fixed_to_float_o:wN
19055     }

(End definition for \__fp_rand_o:Nw and \__fp_rand_o:w.)

```

## 34.3 Random integer

`\__fp_randint_o:Nw`  
`\__fp_randint_default:w`  
`\__fp_randint_badarg:w`  
`\__fp_randint_o:w`  
`\__fp_randint_auxi_o:ww`  
`\__fp_randint_auxii:wn`  
`\__fp_randint_auxiii_o:ww`  
`\__fp_randint_auxiv_o:ww`  
`\__fp_randint_auxv_o:w` 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.

```

19056      \cs_new:Npn \_\_fp_randint_o:Nw ?
19057      {
19058          \_\_fp_parse_function_one_two:nnw
19059          { randint }
19060          { \_\_fp_randint_default:w \_\_fp_randint_o:w }
19061      }
19062      \cs_new:Npn \_\_fp_randint_default:w #1 { \exp_after:wN #1 \c_one_fp }
19063      \cs_new:Npn \_\_fp_randint_badarg:w \s_fp \_\_fp_chk:w #1#2#3;
19064      {
19065          \_\_fp_int:wTF \s_fp \_\_fp_chk:w #1#2#3;
19066      {
19067          \if_meaning:w 1 #1
19068              \if_int_compare:w
19069                  \_\_fp_use_i_until_s:nw #3 ; > \c_fp_prec_int
19070                  1 \exp_stop_f:
19071          \fi:
19072      \fi:
19073  }
19074  { 1 \exp_stop_f: }
19075 }
19076 \cs_new:Npn \_\_fp_randint_o:w #1; #2; @
19077 {
19078     \if_case:w
19079         \_\_fp_randint_badarg:w #1;
19080         \_\_fp_randint_badarg:w #2;
19081         \if:w 1 \_\_fp_compare_back:ww #2; #1; 1 \exp_stop_f: \fi:
19082         0 \exp_stop_f:
19083         \_\_fp_randint_auxi_o:ww #1; #2;
19084     \or:
19085         \_\_fp_invalid_operation_tl_o:ff
19086             { randint } { \_\_fp_array_to_clist:n { #1; #2; } }
19087         \exp:w
19088     \fi:
19089     \exp_after:wN \exp_end:
19090 }
19091 \cs_new:Npn \_\_fp_randint_auxi_o:ww #1 ; #2 ; #3 \exp_end:
19092 {
19093     \fi:
19094     \_\_fp_randint_auxii:wn #2 ;
19095     { \_\_fp_randint_auxii:wn #1 ; \_\_fp_randint_auxiii_o:ww }
19096 }
19097 \cs_new:Npn \_\_fp_randint_auxii:wn \s_fp \_\_fp_chk:w #1#2#3#4 ;

```

```

19098 {
19099   \if_meaning:w 0 #1
19100     \exp_after:wN \use_i:nn
19101   \else:
19102     \exp_after:wN \use_ii:nn
19103   \fi:
19104   { \exp_after:wN \__fp_fixed_continue:wn \c_fp_one_fixed_tl }
19105   {
19106     \exp_after:wN \__fp_ep_to_fixed:wwn
19107     \int_value:w \__fp_int_eval:w
19108       #3 - \c_fp_prec_int , #4 {0000} {0000} ;
19109   {
19110     \if_meaning:w 0 #2
19111       \exp_after:wN \use_i:nnnn
19112         \exp_after:wN \__fp_fixed_add_one:wN
19113       \fi:
19114         \exp_after:wN \__fp_fixed_sub:wwn \c_fp_one_fixed_tl
19115       }
19116       \__fp_fixed_continue:wn
19117     }
19118   }
19119 \cs_new:Npn \__fp_randint_auxiii_o:ww #1 ; #2 ;
19120   {
19121     \__fp_fixed_add:wwn #2 ;
19122       {0000} {0000} {0001} {0000} {0000} ;
19123     \__fp_fixed_sub:wwn #1 ;
19124   {
19125     \exp_after:wN \use_i:nn
19126     \exp_after:wN \__fp_fixed_mul_add:wwn
19127     \exp:w \exp_end_continue_f:w \__fp_rand_myriads:n { XXXXXX } ;
19128   }
19129   #1 ;
19130   \__fp_randint_auxiv_o:ww
19131   #2 ;
19132   \__fp_randint_auxv_o:w #1 ; @
19133 }
19134 \cs_new:Npn \__fp_randint_auxiv_o:ww #1#2#3#4#5 ; #6#7#8#9
19135   {
19136     \if_int_compare:w
19137       \if_int_compare:w #1#2 > #6#7 \exp_stop_f: 1 \else:
19138         \if_int_compare:w #1#2 < #6#7 \exp_stop_f: - \fi: \fi:
19139         #3#4 > #8#9 \exp_stop_f:
19140           \__fp_use_i_until_s:nw
19141         \fi:
19142         \__fp_randint_auxv_o:w {#1}{#2}{#3}{#4}#5
19143   }
19144 \cs_new:Npn \__fp_randint_auxv_o:w #1#2#3#4#5 ; #6 @
19145   {
19146     \exp_after:wN \__fp_SANITIZE:Nw
19147     \int_value:w
19148     \if_int_compare:w #1 < 10000 \exp_stop_f:
19149       2
19150     \else:
19151       0

```

```

19152   \exp_after:wN \exp_after:wN
19153   \exp_after:wN \_fp_reverse_args:Nww
19154 \fi:
19155 \exp_after:wN \_fp_fixed_sub:wwn \c_fp_one_fixed_tl
19156 {\#1} {\#2} {\#3} {\#4} {0000} {0000} ;
19157 {
19158   \exp_after:wN \exp_stop_f:
19159   \int_value:w \_fp_int_eval:w \c_fp_prec_int
19160     \_fp_fixed_to_float_o:wN
19161 }
19162 0
19163 \exp:w \exp_after:wN \exp_end:
19164 }

```

(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.

```

19165 \cs_new:Npn \int_rand:nn #1#2
19166 {
19167   \int_eval:n
19168   {
19169     \exp_after:wN \_fp_randint:ww
19170     \int_value:w \int_eval:n {#1} \exp_after:wN ;
19171     \int_value:w \int_eval:n {#2} ;
19172   }
19173 }
19174 \cs_new:Npn \_fp_randint:ww #1; #2;
19175 {
19176   \if_int_compare:w #1 > #2 \exp_stop_f:
19177     \_kernel_msg_expandable_error:nnnn
19178     { kernel } { randint-backward-range } {#1} {#2}
19179     \_fp_randint:ww #2; #1;
19180   \else:
19181     \if_int_compare:w \_fp_int_eval:w #2
19182       \if_int_compare:w #1 > \c_zero_int
19183         - #1 < \_fp_int_eval:w
19184       \else:
19185         < \_fp_int_eval:w #1 +
19186       \fi:
19187       \c_kernel_randint_max_int
19188       \_fp_int_eval_end:
19189       \_kernel_randint:n
19190         { \_fp_int_eval:w #2 - #1 + 1 \_fp_int_eval_end: }
19191         - 1 + #1
19192   \else:
19193     \_kernel_randint:nn {#1} {#2}
19194   \fi:
19195 \fi:
19196 }

```

(End definition for `\int_rand:nn` and `\_fp_randint:ww`. This function is documented on page 89.)

$\text{\_\_kernel\_randint:nn}$   
 $\text{\_\_fp\_randint\_split\_o:Nw}$   
 $\text{\_\_fp\_randint\_split\_aux:w}$   
 $\text{\_\_fp\_randinat\_wide\_aux:w}$   
 $\text{\_\_fp\_randinat\_wide\_auxii:w}$

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  $\text{\_\_fp\_randint\_split\_o:Nw } n$ ; gives  $n_1$ ;  $n_2$ ; and expands the next token once. We do this for two random numbers and apply  $\text{\_\_fp\_randint\_split\_o:Nw}$  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  $\text{\_\_fp\_randint\_wide\_aux:w } \langle X_1 \rangle; \langle X_0 \rangle; \langle Y_1 \rangle; \langle Y_0 \rangle; \langle R_2 \rangle; \langle R_1 \rangle; \langle R_0 \rangle$ ; . and we apply the algorithm described earlier.

```

19197   \cs_new:Npn \_\_kernel_randint:nn #1#2
19198   {
19199     #1
19200     \exp_after:wN \_\_fp_randint_wide_aux:w
19201     \int_value:w
19202       \exp_after:wN \_\_fp_randint_split_o:Nw
19203         \tex_uniformdeviate:D 268435456 ;
19204     \int_value:w
19205       \exp_after:wN \_\_fp_randint_split_o:Nw
19206         \tex_uniformdeviate:D 268435456 ;
19207     \int_value:w
19208       \exp_after:wN \_\_fp_randint_split_o:Nw
19209         \int_value:w \_\_fp_int_eval:w 131072 +
19210           \exp_after:wN \_\_fp_randint_split_o:Nw
19211             \int_value:w
19212               \_\_kernel_int_add:nnn {#2} { -#1 } { -\c_max_int } ;
19213
19214 .
19215 \cs_new:Npn \_\_fp_randint_split_o:Nw #1#2 ;
19216 {
19217   \if_meaning:w 0 #1
19218     0 \exp_after:wN ; \int_value:w 0
19219   \else:
19220     \exp_after:wN \_\_fp_randint_split_aux:w
19221       \int_value:w \_\_fp_int_eval:w (#1#2 - 8192) / 16384 ;
19222         + #1#2
19223   \fi:
19224   \exp_after:wN ;
19225 }
19226 \cs_new:Npn \_\_fp_randint_split_aux:w #1 ;
19227 {
19228   #1 \exp_after:wN ;
19229   \int_value:w \_\_fp_int_eval:w - #1 * 16384
19230 }
19231 \cs_new:Npn \_\_fp_randint_wide_aux:w #1;#2; #3;#4; #5;#6;#7; .
19232 {
19233   \exp_after:wN \_\_fp_randint_wide_auxii:w
19234   \int_value:w \_\_fp_int_eval:w #5 * #3 + #6 * #1 +
19235     (#5 * #4 + #6 * #3 + #7 * #1 +
19236       (#5 * #2 + #7 * #3 +
19237         (16384 * #6 + #7) * (16384 * #4 + #2) / 268435456) / 16384
19238       ) / 16384 \exp_after:wN ;
19239   \int_value:w \_\_fp_int_eval:w (#5 + #6) * 16384 + #7 ;
19240     #1 ; #5 ;
19241 }
19242 \cs_new:Npn \_\_fp_randint_wide_auxii:w #1; #2; #3; #4;

```

```

19243 {
19244   \if_int_odd:w 0
19245     \if_int_compare:w #1 = #2 \else: \exp_stop_f: \fi:
19246     \if_int_compare:w #4 = \c_zero_int 1 \fi:
19247     \if_int_compare:w #3 = 16383 ~ 1 \fi:
19248     \exp_stop_f:
19249     \exp_after:wN \prg_break:
19250   \fi:
19251   \if_int_compare:w #4 < 8 \exp_stop_f:
19252     + #4 * #3 * 16384
19253   \else:
19254     + 8 * #3 * 16384 + (#4 - 8) * #3 * 16384
19255   \fi:
19256   + #1
19257   \prg_break_point:
19258 }

```

(End definition for `\_kernel_randint:nn` and others.)

**\int\_rand:n** Similar to `\int_rand:nn`, but needs fewer checks.

```

\__fp_randint:n
19259   \cs_new:Npn \int_rand:n #1
19260   {
19261     \int_eval:n
19262       { \exp_args:Nf \__fp_randint:n { \int_eval:n {#1} } }
19263   }
19264   \cs_new:Npn \__fp_randint:n #1
19265   {
19266     \if_int_compare:w #1 < 1 \exp_stop_f:
19267       \__kernel_msg_expandable_error:nnnn
19268         { kernel } { randint-backward-range } { 1 } { #1 }
19269       \__fp_randint:ww #1; 1;
19270   \else:
19271     \if_int_compare:w #1 > \c_kernel_randint_max_int
19272       \__kernel_randint:nn { 1 } { #1 }
19273     \else:
19274       \__kernel_randint:n {#1}
19275     \fi:
19276   \fi:
19277 }

```

(End definition for `\int_rand:n` and `\__fp_randint:n`. This function is documented on page 89.)

End the initial conditional that ensures these commands are only defined in engines that support random numbers.

```

19278   }
19279   </initex | package>

```

## 35 I3fparray implementation

```

19280   <*initex | package>
19281   <@=fp>

```

In analogy to `\l3intarray` it would make sense to have `<@@=fparray>`, but we need direct access to `\__fp_parse:n` from `\l3fp-parse`, and a few other (less crucial) internals of the `\l3fp` family.

### 35.1 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.

```
19282 \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`.

```
19283 \int_new:N \l__fp_array_loop_int
```

*(End definition for `\l__fp_array_loop_int`.)*

`\fpararray_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.

```
19284 \cs_new_protected:Npn \fpararray_new:Nn #1#2
19285 {
19286   \tl_new:N #1
19287   \prg_replicate:nn { 3 }
19288   {
19289     \int_gincr:N \g__fp_array_int
19290     \exp_args:NNo \tl_gput_right:Nn #1
19291     { g__fp_array_ \__fp_int_to_roman:w \g__fp_array_int _intarray }
19292   }
19293   \exp_last_unbraced:Nfo \__fp_array_new:nNNNN
19294   { \int_eval:n {#2} } #1 #1
19295 }
19296 \cs_new_protected:Npn \__fp_array_new:nNNNN #1#2#3#4#5
19297 {
19298   \int_compare:nNnTF {#1} < 0
19299   {
19300     \__kernel_msg_error:nnn { kernel } { negative-array-size } {#1}
19301     \cs_undefine:N #1
19302     \int_gsub:Nn \g__fp_array_int { 3 }
19303   }
19304   {
19305     \intarray_new:Nn #2 {#1}
19306     \intarray_new:Nn #3 {#1}
19307     \intarray_new:Nn #4 {#1}
19308   }
19309 }
```

*(End definition for `\fpararray_new:Nn` and `\__fp_array_new:nNNN`. This function is documented on page 244.)*

**\fparray\_count:N** Size of any of the intarrays, here we pick the third.

```

19310 \cs_new:Npn \fparray_count:N #1
19311 {
19312     \exp_after:wN \use_i:n
19313     \exp_after:wN \intarray_count:N #1
19314 }
```

(End definition for `\fparray_count:N`. This function is documented on page 244.)

## 35.2 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.

```

19315 \cs_new:Npn \__fp_array_bounds:NNnTF #1#2#3#4#5
19316 {
19317     \if_int_compare:w 1 > #3 \exp_stop_f:
19318         \__fp_array_bounds_error>NNn #1 #2 {#3}
19319         #5
19320     \else:
19321         \if_int_compare:w #3 > \fparray_count:N #2 \exp_stop_f:
19322             \__fp_array_bounds_error>NNn #1 #2 {#3}
19323             #5
19324         \else:
19325             #4
19326         \fi:
19327     \fi:
19328 }
19329 \cs_new:Npn \__fp_array_bounds_error>NNn #1#2#3
19330 {
19331     #1 { kernel } { out-of-bounds }
19332     { \token_to_str:N #2 } {#3} { \fparray_count:N #2 }
19333 }
```

(End definition for `\__fp_array_bounds:NNnTF` and `\__fp_array_bounds_error>NNn`.)

**\fparray\_gset:Nnn**

`\__fp_array_gset:NNNNww`

`\__fp_array_gset:w`

`\__fp_array_gset_recover:Nw`

`\__fp_array_gset_special:mN`

`\__fp_array_gset_normal:w`

```

19334 \cs_new_protected:Npn \fparray_gset:Nnn #1#2#3
19335 {
19336     \exp_after:wN \exp_after:wN
19337     \exp_after:wN \__fp_array_gset:NNNNww
19338     \exp_after:wN #1
19339     \exp_after:wN #1
19340     \int_value:w \int_eval:n {#2} \exp_after:wN ;
19341     \exp:w \exp_end_continue_f:w \__fp_parse:n {#3}
19342 }
19343 \cs_new_protected:Npn \__fp_array_gset:NNNNww #1#2#3#4#5 ; #6 ;
19344 {
19345     \__fp_array_bounds:NNnTF \__kernel_msg_error:nnxxx #4 {#5}
19346 {
19347     \exp_after:wN \__fp_change_func_type:NNN
19348         \__fp_use_i_until_s:nw #6 ;
19349     \__fp_array_gset:w
```

```

19350           \__fp_array_gset_recover:Nw
19351             #6 ; {#5} #1 #2 #3
19352         }
19353     {
19354   }
19355 \cs_new_protected:Npn \__fp_array_gset_recover:Nw #1#2 ;
19356   {
19357     \__fp_error:nffn { fp-unknown-type } { \tl_to_str:n { #2 ; } } { } { }
19358     \exp_after:wN #1 \c_nan_fp
19359   }
19360 \cs_new_protected:Npn \__fp_array_gset:w \s_fp \__fp_chk:w #1#2
19361   {
19362     \if_case:w #1 \exp_stop_f:
19363       \__fp_case_return:nw { \__fp_array_gset_special:nnNNN {#2} }
19364       \or: \exp_after:wN \__fp_array_gset_normal:w
19365       \or: \__fp_case_return:nw { \__fp_array_gset_special:nnNNN { #2 3 } }
19366       \or: \__fp_case_return:nw { \__fp_array_gset_special:nnNNN { 1 } }
19367       \fi:
19368       \s_fp \__fp_chk:w #1 #2
19369     }
19370 \cs_new_protected:Npn \__fp_array_gset_normal:w
19371   \s_fp \__fp_chk:w 1 #1 #2 #3#4#5 ; #6#7#8#9
19372   {
19373     \__kernel_intarray_gset:Nnn #7 {#6} {#2}
19374     \__kernel_intarray_gset:Nnn #8 {#6}
19375     { \if_meaning:w 2 #1 3 \else: 1 \fi: #3#4 }
19376     \__kernel_intarray_gset:Nnn #9 {#6} { 1 \use:nn #5 }
19377   }
19378 \cs_new_protected:Npn \__fp_array_gset_special:nnNNN #1#2#3#4#5
19379   {
19380     \__kernel_intarray_gset:Nnn #3 {#2} {#1}
19381     \__kernel_intarray_gset:Nnn #4 {#2} {0}
19382     \__kernel_intarray_gset:Nnn #5 {#2} {0}
19383   }

```

(End definition for `\fparray_gset:Nnn` and others. This function is documented on page 244.)

### `\fparray_gzero:N`

```

19384 \cs_new_protected:Npn \fparray_gzero:N #1
19385   {
19386     \int_zero:N \l__fp_array_loop_int
19387     \prg_replicate:nn { \fparray_count:N #1 }
19388     {
19389       \int_incr:N \l__fp_array_loop_int
19390       \exp_after:wN \__fp_array_gset_special:nnNNN
19391       \exp_after:wN 0
19392       \exp_after:wN \l__fp_array_loop_int
19393       #1
19394     }
19395   }

```

(End definition for `\fparray_gzero:N`. This function is documented on page 245.)

### `\fparray_item:Nn`

### `\fparray_item_to_tl:Nn`

`\__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`

```

19396 \cs_new:Npn \fparray_item:Nn #1#2
19397 {
19398     \exp_after:wN \__fp_array_item:NwN
19399     \exp_after:wN #1
19400     \int_value:w \int_eval:n {#2} ;
19401     \__fp_to_decimal:w
19402 }
19403 \cs_new:Npn \fparray_item_to_tl:Nn #1#2
19404 {
19405     \exp_after:wN \__fp_array_item:NwN
19406     \exp_after:wN #1
19407     \int_value:w \int_eval:n {#2} ;
19408     \__fp_to_tl:w
19409 }
19410 \cs_new:Npn \__fp_array_item:NwN #1#2 ; #3
19411 {
19412     \__fp_array_bounds>NNnTF \__kernel_msg_expandable_error:nnfff #1 {#2}
19413     { \exp_after:wN \__fp_array_item>NNNnN #1 {#2} #3 }
19414     { \exp_after:wN #3 \c_nan_fp }
19415 }
19416 \cs_new:Npn \__fp_array_item>NNNnN #1#2#3#4
19417 {
19418     \exp_after:wN \__fp_array_item:N
19419     \int_value:w \__kernel_intarray_item:Nn #2 {#4} \exp_after:wN ;
19420     \int_value:w \__kernel_intarray_item:Nn #3 {#4} \exp_after:wN ;
19421     \int_value:w \__kernel_intarray_item:Nn #1 {#4} ;
19422 }
19423 \cs_new:Npn \__fp_array_item:N #1
19424 {
19425     \if_meaning:w 0 #1 \exp_after:wN \__fp_array_item_special:w \fi:
19426     \__fp_array_item:w #1
19427 }
19428 \cs_new:Npn \__fp_array_item:w #1 #2#3#4#5 #6 ; 1 #7 ;
19429 {
19430     \exp_after:wN \__fp_array_item_normal:w
19431     \int_value:w \if_meaning:w #1 1 0 \else: 2 \fi: \exp_stop_f:
19432     #7 ; {#2#3#4#5} {#6} ;
19433 }
19434 \cs_new:Npn \__fp_array_item_special:w #1 ; #2 ; #3 ; #4
19435 {
19436     \exp_after:wN #4
19437     \exp:w \exp_end_continue_f:w
19438     \if_case:w #3 \exp_stop_f:
19439         \exp_after:wN \c_zero_fp
19440         \or: \exp_after:wN \c_nan_fp
19441         \or: \exp_after:wN \c_minus_zero_fp
19442         \or: \exp_after:wN \c_inf_fp
19443         \else: \exp_after:wN \c_minus_inf_fp
19444         \fi:
19445 }
19446 \cs_new:Npn \__fp_array_item_normal:w #1 #2#3#4#5 #6 ; #7 ; #8 ; #9
19447 { #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 245.)

19448 ⟨/initex | package⟩

## 36 I3sort implementation

19449 ⟨\*initex | package⟩  
19450 ⟨@@=sort⟩

### 36.1 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 T<sub>E</sub>X does not need to re-read tokens. For clist we don't gain anything since the result is converted from seq to clist anyways.

19451 \seq\_new:N \g\_\_sort\_internal\_seq  
19452 \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 \l\_\_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.

19453 \int\_new:N \l\_\_sort\_length\_int  
19454 \int\_new:N \l\_\_sort\_min\_int  
19455 \int\_new:N \l\_\_sort\_top\_int  
19456 \int\_new:N \l\_\_sort\_max\_int  
19457 \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.

19458 \int\_new:N \l\_\_sort\_block\_int

(End definition for \l\_\_sort\_block\_int.)

\l\_\_sort\_begin\_int  
\l\_\_sort\_end\_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.

19459 \int\_new:N \l\_\_sort\_begin\_int  
19460 \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.

```

19461 \int_new:N \l__sort_A_int
19462 \int_new:N \l__sort_B_int
19463 \int_new:N \l__sort_C_int

(End definition for \l__sort_A_int, \l__sort_B_int, and \l__sort_C_int.)

```

## 36.2 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.

```

19464 \cs_new_protected:Npn \_\_sort_shrink_range:
19465 {
19466     \int_set:Nn \l__sort_A_int
19467         { \l__sort_true_max_int - \l__sort_min_int + 1 }
19468     \int_set:Nn \l__sort_block_int { \c_max_register_int / 2 }
19469     \_\_sort_shrink_range_loop:
19470     \int_set:Nn \l__sort_max_int
19471         {
19472             \int_compare:nNnTF
19473                 { \l__sort_block_int * 3 / 2 } > \l__sort_A_int
19474                 {
19475                     \l__sort_min_int
19476                     + ( \l__sort_A_int - 1 ) / 2
19477                     + \l__sort_block_int / 4
19478                     - 1
19479                 }
19480                 { \l__sort_true_max_int - \l__sort_block_int / 2 }
19481             }
19482         }
19483 \cs_new_protected:Npn \_\_sort_shrink_range_loop:
19484 {
19485     \if_int_compare:w \l__sort_A_int < \l__sort_block_int
19486         \tex_divide:D \l__sort_block_int 2 \exp_stop_f:
19487         \exp_after:wN \_\_sort_shrink_range_loop:
19488     \fi:
19489 }

```

(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 ConTeXt 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::`. The L<sup>A</sup>T<sub>E</sub>X3 format mode is easiest: no `\toks` are ever allocated so available `\toks` range from 0 to `\c_max_register_int` and we precompute the result of `\__sort_shrink_range::`.

```

19490  {*package}
19491  \cs_new_protected:Npn \__sort_compute_range:
19492  {
19493      \int_set:Nn \l__sort_min_int { \tex_count:D 15 + 1 }
19494      \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
19495      \__sort_shrink_range:
19496      \if_meaning:w \loctoks \tex_undefined:D \else:
19497          \if_meaning:w \loctoks \scan_stop: \else:
19498              \__sort_redefine_compute_range:
19499              \__sort_compute_range:
19500          \fi:
19501      \fi:
19502  }
19503  \cs_new_protected:Npn \__sort_redefine_compute_range:
19504  {
19505      \cs_if_exist:cTF { ver@elocalalloc.sty }
19506      {
19507          \cs_gset_protected:Npn \__sort_compute_range:
19508          {
19509              \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
19510              \int_set_eq:NN \l__sort_true_max_int \e@alloc@top
19511              \__sort_shrink_range:
19512          }
19513      }
19514      {
19515          \cs_gset_protected:Npn \__sort_compute_range:
19516          {
19517              \int_set:Nn \l__sort_min_int { \tex_count:D 265 }
19518              \int_set:Nn \l__sort_true_max_int { \tex_count:D 275 }
19519              \__sort_shrink_range:
19520          }
19521      }
19522  }
19523  \cs_if_exist:NT \loctoks { \__sort_redefine_compute_range: }
19524  \tl_map_inline:nn { \lastallocatedtoks \c_syst_last_allocated_toks }
19525  {
19526      \cs_if_exist:NT #1
19527      {
19528          \cs_gset_protected:Npn \__sort_compute_range:
19529          {
19530              \int_set:Nn \l__sort_min_int { #1 + 1 }
19531              \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
19532              \__sort_shrink_range:
19533          }
19534      }
19535  }
19536  </package>
19537  {*initex}

```

```

19538 \int_const:Nn \c__sort_max_length_int
19539   { ( \c_max_register_int + 1 ) * 3 / 4 }
19540 \cs_new_protected:Npn \__sort_compute_range:
19541   {
19542     \int_set:Nn \l__sort_min_int { 0 }
19543     \int_set:Nn \l__sort_true_max_int { \c_max_register_int + 1 }
19544     \int_set:Nn \l__sort_max_int { \c__sort_max_length_int }
19545   }
19546 
```

(End definition for `\__sort_compute_range:`, `\__sort_redefine_compute_range:`, and `\c__sort_max_length_int`.)

### 36.3 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`.

```

19547 \cs_new_protected:Npn \__sort_main:NNNn #1#2#3#4
19548   {
19549     (package) \__sort_disable_toksdef:
19550       \__sort_compute_range:
19551         \int_set_eq:NN \l__sort_top_int \l__sort_min_int
19552           #1 #3
19553           {
19554             \if_int_compare:w \l__sort_top_int = \l__sort_max_int
19555               \__sort_too_long_error>NNw #2 #3
19556             \fii:
19557               \tex_toks:D \l__sort_top_int {##1}
19558               \int_incr:N \l__sort_top_int
19559             }
19560             \int_set:Nn \l__sort_length_int
19561               { \l__sort_top_int - \l__sort_min_int }
19562             \cs_set:Npn \__sort_compare:nn ##1 ##2 {#4}
19563             \int_set:Nn \l__sort_block_int { 1 }
19564             \__sort_level:
19565           }

```

(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.

```

19566 \cs_new_protected:Npn \tl_sort:Nn { \__sort_tl:NNn \tl_set_eq:NN }
19567 \cs_generate_variant:Nn \tl_sort:Nn { c }
19568 \cs_new_protected:Npn \tl_gsort:Nn { \__sort_tl:NNn \tl_gset_eq:NN }
19569 \cs_generate_variant:Nn \tl_gsort:Nn { c }

```

```

19570 \cs_new_protected:Npn \__sort_tl:NNn #1#2#3
19571 {
19572     \group_begin:
19573         \__sort_main:NNNn \tl_map_inline:Nn \tl_map_break:n #2 {#3}
19574         \tl_gset:Nx \g__sort_internal_tl
19575             { \__sort_tl_toks:w \l__sort_min_int ; }
19576     \group_end:
19577         #1 #2 \g__sort_internal_tl
19578         \tl_gclear:N \g__sort_internal_tl
19579         \prg_break_point:
19580     }
19581 \cs_new:Npn \__sort_tl_toks:w #1 ;
19582 {
19583     \if_int_compare:w #1 < \l__sort_top_int
19584         { \tex_the:D \tex_toks:D #1 }
19585         \exp_after:wN \__sort_tl_toks:w
19586             \int_value:w \int_eval:n { #1 + 1 } \exp_after:wN ;
19587     \fi:
19588 }

```

(End definition for `\tl_sort:Nn` and others. These functions are documented on page 44.)

<code>\seq_sort:Nn</code> <code>\seq_sort:cn</code> <code>\seq_gsort:Nn</code> <code>\seq_gsort:cn</code> <code>\clist_sort:Nn</code> <code>\clist_sort:cn</code> <code>\clist_gsort:Nn</code> <code>\clist_gsort:cn</code> <code>\__sort_seq:NNNNn</code>	Use the same general framework for seq and clist. Apply the general sorting code, then unpack <code>\toks</code> into <code>\g__sort_internal_seq</code> . Outside the group copy or convert (for clist) the data to the target variable. The <code>\seq_gclear:N</code> reduces memory usage. The <code>\prg_break_point:</code> is used by <code>\__sort_main:NNNn</code> when the list is too long.
<code>\__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_set_eq:NN</code> <code>\cs_generate_variant:Nn \seq_sort:Nn { c }</code> <code>\cs_new_protected:Npn \seq_gsort:Nn</code> <code>{ \__sort_seq:NNNNn \seq_map_inline:Nn \seq_map_break:n \seq_gset_eq:NN }</code> <code>\cs_generate_variant:Nn \seq_gsort:Nn { c }</code> <code>\cs_new_protected:Npn \clist_sort:Nn</code> <code>{</code> <code>    \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n</code> <code>    \clist_set_from_seq:NN</code> <code>}</code> <code>\cs_generate_variant:Nn \clist_sort:Nn { c }</code> <code>\cs_new_protected:Npn \clist_gsort:Nn</code> <code>{</code> <code>    \__sort_seq:NNNNn \clist_map_inline:Nn \clist_map_break:n</code> <code>    \clist_gset_from_seq:NN</code> <code>}</code> <code>\cs_generate_variant:Nn \clist_gsort:Nn { c }</code> <code>\cs_new_protected:Npn \__sort_seq:NNNNn #1#2#3#4#5</code> <code>{</code> <code>    \group_begin:</code> <code>        \__sort_main:NNNn #1 #2 #4 {#5}</code> <code>        \seq_gset_from_inline_x:Nnn \g__sort_internal_seq</code> <code>        {</code> <code>            \int_step_function:nnN</code> <code>                { \l__sort_min_int } { \l__sort_top_int - 1 }</code> <code>        }</code> <code>        { \tex_the:D \tex_toks:D ##1 }</code>	

```

19617     \group_end:
19618     #3 #4 \g__sort_internal_seq
19619     \seq_gclear:N \g__sort_internal_seq
19620     \prg_break_point:
19621 }

```

(End definition for `\seq_sort:Nn` and others. These functions are documented on page 72.)

## 36.4 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.

```

19622 \cs_new_protected:Npn \__sort_level:
19623 {
19624     \if_int_compare:w \l__sort_block_int < \l__sort_length_int
19625         \l__sort_end_int \l__sort_min_int
19626         \__sort_merge_blocks:
19627         \tex_advance:D \l__sort_block_int \l__sort_block_int
19628         \exp_after:wN \__sort_level:
19629     \fi:
19630 }

```

(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.

```

19631 \cs_new_protected:Npn \__sort_merge_blocks:
19632 {
19633     \l__sort_begin_int \l__sort_end_int
19634     \tex_advance:D \l__sort_end_int \l__sort_block_int
19635     \if_int_compare:w \l__sort_end_int < \l__sort_top_int
19636         \l__sort_A_int \l__sort_end_int
19637         \tex_advance:D \l__sort_end_int \l__sort_block_int
19638         \if_int_compare:w \l__sort_end_int > \l__sort_top_int
19639             \l__sort_end_int \l__sort_top_int
19640         \fi:
19641         \l__sort_B_int \l__sort_A_int
19642         \l__sort_C_int \l__sort_top_int
19643         \__sort_copy_block:
19644         \int_decr:N \l__sort_A_int
19645         \int_decr:N \l__sort_B_int
19646         \int_decr:N \l__sort_C_int

```

```

19647     \exp_after:wN \__sort_merge_blocks_aux:
19648     \exp_after:wN \__sort_merge_blocks:
19649     \fi:
19650   }

```

(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`.

```

19651 \cs_new_protected:Npn \__sort_copy_block:
19652   {
19653     \tex_toks:D \l__sort_C_int \tex_toks:D \l__sort_B_int
19654     \int_incr:N \l__sort_C_int
19655     \int_incr:N \l__sort_B_int
19656     \if_int_compare:w \l__sort_B_int = \l__sort_end_int
19657       \use_i:nn
19658     \fi:
19659     \__sort_copy_block:
19660   }

```

(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.

```

19661 \cs_new_protected:Npn \__sort_merge_blocks_aux:
19662   {
19663     \exp_after:wN \__sort_compare:nn \exp_after:wN
19664     { \tex_the:D \tex_toks:D \exp_after:wN \l__sort_A_int \exp_after:wN }
19665     \exp_after:wN { \tex_the:D \tex_toks:D \l__sort_C_int }
19666     \prg_do_nothing:
19667     \__sort_return_mark:w
19668     \__sort_return_mark:w
19669     \q_mark
19670     \__sort_return_none_error:
19671   }

```

(End definition for `\__sort_merge_blocks_aux:..`)

`\sort_return_same:` 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 `\q_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.

```

19672 \cs_new_protected:Npn \sort_return_same:
19673   #1 \__sort_return_mark:w #2 \q_mark

```

```

19674  {
19675      #1
19676      #2
19677      \__sort_return_two_error:
19678      \__sort_return_mark:w
19679      \q_mark
19680      \__sort_return_same:w
19681  }
19682 \cs_new_protected:Npn \sort_return_swapped:
19683     #1 \__sort_return_mark:w #2 \q_mark
19684  {
19685      #1
19686      #2
19687      \__sort_return_two_error:
19688      \__sort_return_mark:w
19689      \q_mark
19690      \__sort_return_swapped:w
19691  }
19692 \cs_new_protected:Npn \__sort_return_mark:w #1 \q_mark { }
19693 \cs_new_protected:Npn \__sort_return_none_error:
19694  {
19695      \__kernel_msg_error:nxxx { kernel } { return-none }
19696      { \tex_the:D \tex_toks:D \l__sort_A_int }
19697      { \tex_the:D \tex_toks:D \l__sort_C_int }
19698      \__sort_return_same:w \__sort_return_none_error:
19699  }
19700 \cs_new_protected:Npn \__sort_return_two_error:
19701  {
19702      \__kernel_msg_error:nxxx { kernel } { return-two }
19703      { \tex_the:D \tex_toks:D \l__sort_A_int }
19704      { \tex_the:D \tex_toks:D \l__sort_C_int }
19705  }

```

(End definition for `\sort_return_same:` and others. These functions are documented on page 206.)

`\__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.

```

19706 \cs_new_protected:Npn \__sort_return_same:w #1 \__sort_return_none_error:
19707  {
19708      \tex_toks:D \l__sort_B_int \tex_toks:D \l__sort_C_int
19709      \int_decr:N \l__sort_B_int
19710      \int_decr:N \l__sort_C_int
19711      \if_int_compare:w \l__sort_C_int < \l__sort_top_int
19712          \use_i:nn
19713      \fi:
19714      \__sort_merge_blocks_aux:
19715  }

```

(End definition for `\__sort_return_same:w`.)

\\_\\_sort\\_return\\_swapped:  
 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`:

```

19716 \cs_new_protected:Npn \_\_sort_return_swapped:w #1 \_\_sort_return_none_error:
19717 {
19718     \tex_toks:D \l_\_sort_B_int \tex_toks:D \l_\_sort_A_int
19719     \int_decr:N \l_\_sort_B_int
19720     \int_decr:N \l_\_sort_A_int
19721     \if_int_compare:w \l_\_sort_A_int < \l_\_sort_begin_int
19722         \_\_sort_merge_blocks_end: \use_i:nn
19723     \fi:
19724     \_\_sort_merge_blocks_aux:
19725 }
```

(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`.

```

19726 \cs_new_protected:Npn \_\_sort_merge_blocks_end:
19727 {
19728     \tex_toks:D \l_\_sort_B_int \tex_toks:D \l_\_sort_C_int
19729     \int_decr:N \l_\_sort_B_int
19730     \int_decr:N \l_\_sort_C_int
19731     \if_int_compare:w \l_\_sort_B_int < \l_\_sort_begin_int
19732         \use_i:nn
19733     \fi:
19734     \_\_sort_merge_blocks_end:
19735 }
```

(End definition for `\_\_sort_merge_blocks_end`.)

### 36.5 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_recursion_tail \q_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}
{\extra arguments}
}
\__sort_loop:wNn ... {\langle prepared tokens \rangle}
{\end-loop} {} \q_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:nnnnNn #1#2#3#4#5#6
{
#5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i:nnnnNn
{#6} { #2 {#1} } {#3} {#4}
}
\cs_new:Npn \__sort_ii:nnnnNn #1#2#3#4#5#6

```

```
{
#5 {#4} {#6} \__sort_ii:nnnnNn \__sort_i: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 `<conditional> {<item>}`, 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 `\q_mark {<code>}`, and expands to `<code> {<sorted list>}`. Sorting the two parts of the list around the pivot is done with

```
\__sort_quick_split:NnNn #2 ... \q_mark
{
\__sort_quick_split:NnNn #1 ... \q_mark {<code>}
{<pivot>}
}
```

Items which are larger than the `<pivot>` are sorted, then placed after code that sorts the smaller items, and after the (braced) `<pivot>`.

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:nnnnNn` of the last example, but aware of whether the list of `<conditional> {<item>}` 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 `<end-loop>` 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 `<end-loop>` 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 `\q_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 `\q_mark`, namely removing the trailing `\s_stop` and `\q_stop` and leaving `\exp_stop_f:` to stop f-expansion.

```

19736 \cs_new:Npn \tl_sort:nN #1#2
19737 {
19738   \exp_not:f
19739   {
19740     \tl_if_blank:nF {#1}
19741     {
19742       \__sort_quick_prepare:Nnnn #2 { } { }
19743       #1
19744       { \prg_break_point: \__sort_quick_prepare_end:NNNnw }
19745       \q_stop
19746     }
19747   }
19748 }
19749 \cs_new:Npn \__sort_quick_prepare:Nnnn #1#2#3#4
19750 {
19751   \prg_break: #4 \prg_break_point:
19752   \__sort_quick_prepare:Nnnn #1 { #2 #3 } { #1 {#4} }
19753 }
19754 \cs_new:Npn \__sort_quick_prepare_end:NNNnw #1#2#3#4#5 \q_stop
19755 {
19756   \__sort_quick_split:NnNn #4 \__sort_quick_end:nnTFNn { }
19757   \q_mark { \__sort_quick_cleanup:w \exp_stop_f: }
19758   \s_stop \q_stop
19759 }
19760 \cs_new:Npn \__sort_quick_cleanup:w #1 \s_stop \q_stop {#1}

```

(End definition for `\tl_sort:nN` and others. This function is documented on page 44.)

`\__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 `<function>` #6, and an item #7. The `<function>` is the user's `<conditional>` except at the end of the list where it is `\__sort_quick_end:nnTFNn`. The comparison is applied to the `<pivot>` and the `<item>`, and calls the `only_i` or `split_i` auxiliaries if the `<item>` 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 `<conditional> {<item>}`, 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.

```

19761 \cs_new:Npn \__sort_quick_split:NnNn #1#2#3#4
19762 {
19763   #3 {#2} {#4} \__sort_quick_only_ii:NnnnnNn
19764   \__sort_quick_only_i:NnnnnNn
19765   \__sort_quick_single_end:nnnw
19766   { #3 {#4} } { } { } {#2}
19767 }
19768 \cs_new:Npn \__sort_quick_only_i:NnnnnNn #1#2#3#4#5#6#7
19769 {

```

```

19770      #6 {#5} {#7} \_\_sort_quick_split_i:NnnnnNn
19771          \_\_sort_quick_only_i:NnnnnNn
19772          \_\_sort_quick_only_i_end:nnnw
19773          { #6 {#7} } { #3 #2 } { } {#5}
19774      }
19775 \cs_new:Npn \_\_sort_quick_only_ii:NnnnnNn #1#2#3#4#5#6#7
19776  {
19777      #6 {#5} {#7} \_\_sort_quick_only_ii:NnnnnNn
19778          \_\_sort_quick_split_i:NnnnnNn
19779          \_\_sort_quick_only_ii_end:nnnw
19780          { #6 {#7} } { } { #4 #2 } {#5}
19781      }
19782 \cs_new:Npn \_\_sort_quick_split_i:NnnnnNn #1#2#3#4#5#6#7
19783  {
19784      #6 {#5} {#7} \_\_sort_quick_split_i:NnnnnNn
19785          \_\_sort_quick_split_i:NnnnnNn
19786          \_\_sort_quick_split_end:nnnw
19787          { #6 {#7} } { #3 #2 } {#4} {#5}
19788      }
19789 \cs_new:Npn \_\_sort_quick_split_ii:NnnnnNn #1#2#3#4#5#6#7
19790  {
19791      #6 {#5} {#7} \_\_sort_quick_split_ii:NnnnnNn
19792          \_\_sort_quick_split_i:NnnnnNn
19793          \_\_sort_quick_split_end:nnnw
19794          { #6 {#7} } {#3} { #4 #2 } {#5}
19795      }

```

(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 `\q_mark`. To avoid a memory problem described earlier, all of the ending functions read #6 until `\q_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.

```

19796 \cs_new:Npn \_\_sort_quick_end:nnTFNn #1#2#3#4#5#6 {#5}
19797 \cs_new:Npn \_\_sort_quick_single_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
19798  { #5 {#3} #6 \q_stop }
19799 \cs_new:Npn \_\_sort_quick_only_i_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
19800  {
19801      \_\_sort_quick_split:NnNn #1
19802          \_\_sort_quick_end:nnTFNn { } \q_mark {#5}
19803          {#3}
19804          #6 \q_stop
19805      }

```

```

19806 \cs_new:Npn \__sort_quick_only_ii_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
19807 {
19808     \__sort_quick_split:NnNn #2
19809         \__sort_quick_end:nnTFNn { } \q_mark { #5 {#3} }
19810     #6 \q_stop
19811 }
19812 \cs_new:Npn \__sort_quick_split_end:nnnw #1#2#3#4 \q_mark #5#6 \q_stop
19813 {
19814     \__sort_quick_split:NnNn #2 \__sort_quick_end:nnTFNn { } \q_mark
19815 {
19816     \__sort_quick_split:NnNn #1
19817         \__sort_quick_end:nnTFNn { } \q_mark {#5}
19818     {#3}
19819 }
19820     #6 \q_stop
19821 }

```

(End definition for `\__sort_quick_end:nnTFNn` and others.)

## 36.6 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.

```

19822 \cs_new_protected:Npn \__sort_error:
19823 {
19824     \cs_set_eq:NN \__sort_merge_blocks_aux: \prg_do_nothing:
19825     \cs_set_eq:NN \__sort_merge_blocks: \prg_do_nothing:
19826     \cs_set_protected:Npn \__sort_level: { \group_end: \prg_break: }
19827 }

```

(End definition for `\__sort_error::`)

`\__sort_disable_toksdef:` `\__sort_disabled_toksdef:n` 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.

```

19828 (*package)
19829 \cs_new_protected:Npn \__sort_disable_toksdef:
19830 {
19831     \cs_set_eq:NN \toksdef \__sort_disabled_toksdef:n
19832 }
19833     \__kernel_msg_error:nnx { kernel } { toksdef }
19834     { \token_to_str:N #1 }
19835     \__sort_error:
19836     \tex_toksdef:D #1
19837 }
19838 \__kernel_msg_new:nnnn { kernel } { toksdef }
19839 {
19840     Allocation-of-\iow_char:N\ \toks-registers-impossible-while-sorting.
19841     The-comparison-code-used-for-sorting-a-list-has-attempted-to-
19842     define-#1-as-a-new-\iow_char:N\ \toks-register-using-
19843     \iow_char:N\ \newtoks-
19844     or-a-similar-command.-The-list-will-not-be-sorted.

```

```

19845      }
19846  </package>

```

(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.

```

19847  \cs_new_protected:Npn \_sort_too_long_error:NNw #1#2 \fi:
19848    {
19849      \fi:
19850      \_kernel_msg_error:nnxx { kernel } { too-large }
19851        { \token_to_str:N #2 }
19852        { \int_eval:n { \l__sort_true_max_int - \l__sort_min_int } }
19853        { \int_eval:n { \l__sort_top_int - \l__sort_min_int } }
19854      #1 \_sort_error:
19855    }
19856  \_kernel_msg_new:nnn { kernel } { too-large }
19857    { The-list~#1-is~too-long-to~be~sorted~by~TeX. }
19858    {
19859      TeX-has~#2~toks~registers~still-available:~
19860      this~only~allows~to~sort~with~up~to~#3~
19861      items.~The~list~will~not~be~sorted.
19862    }

```

(End definition for `\_sort_too_long_error:NNw.`)

```

19863  \_kernel_msg_new:nnn { kernel } { return-none }
19864    { The~comparison~code~did~not~return. }
19865    {
19866      When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~
19867      did~not~call~
19868      \iow_char:N\sort_return_same:~nor~
19869      \iow_char:N\sort_return_swapped:~.
19870      Exactly~one~of~these~should~be~called.
19871    }
19872  \_kernel_msg_new:nnm { kernel } { return-two }
19873    { The~comparison~code~returned~multiple~times. }
19874    {
19875      When~sorting~a~list,~the~code~to~compare~items~#1~and~#2~called~
19876      \iow_char:N\sort_return_same:~or~
19877      \iow_char:N\sort_return_swapped:~multiple~times.~
19878      Exactly~one~of~these~should~be~called.
19879    }
19880  </initex | package>

```

## 37 I3tl-analysis implementation

```

19881  <@@=tl>

```

### 37.1 Internal functions

`\s__tl` The format used to store token lists internally uses the scan mark `\s__tl` as a delimiter.

(End definition for `\s__tl.`)

## 37.2 Internal format

The task of the `\l3tl-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 `\l3regex` where we need to support arbitrary tokens, and it is used in conversion functions in `\l3str-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 `\l3str-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 `\l3str-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_{\_}t1 0 -1 \s_{\_}t1`.
- A begin-group character `{` becomes `\exp_after:wN { \if_false: } \fi: \s_{\_}t1 1 \langle char code \rangle \s_{\_}t1`.
- An end-group character `}` becomes `\if_false: { \fi: } \s_{\_}t1 2 \langle char code \rangle \s_{\_}t1`.
- A character with any other category code becomes `\exp_not:n { \langle character \rangle } \s_{\_}t1 \langle hex catcode \rangle \langle char code \rangle \s_{\_}t1`.

19882  $\langle *initex | package \rangle$

## 37.3 Variables and helper functions

`\s_{\_}t1` The scan mark `\s_{\_}t1` 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 `\int_value:w '#1 \s_{\_}t1` with `\int_value:w '#1 \exp_stop_f: \exp_not:N \q_mark` to extract a character code followed by the delimiter in an x-expansion.

19883 `\scan_new:N \s_{\_}t1`

(End definition for `\s_{\_}t1`.)

\l\_\_tl\_analysis\_token  
\l\_\_tl\_analysis\_char\_token

The tokens in the token list are probed with the TeX primitive \futurelet. We use \l\_\_tl\_analysis\_token in that construction. In some cases, we convert the following token to a string before probing it: then the token variable used is \l\_\_tl\_analysis\_char\_token.

```
19884 \cs_new_eq:NN \l__tl_analysis_token ?
19885 \cs_new_eq:NN \l__tl_analysis_char_token ?
```

(End definition for \l\_\_tl\_analysis\_token and \l\_\_tl\_analysis\_char\_token.)

\l\_\_tl\_analysis\_normal\_int

The number of normal (N-type argument) tokens since the last special token.

```
19886 \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.

```
19887 \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.

```
19888 \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.

```
19889 \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
```

```
19890 \tl_new:N \g__tl_analysis_result_tl
```

(End definition for \g\_\_tl\_analysis\_result\_tl.)

\\_tl\_analysis\_extract\_charcode:  
\\_tl\_analysis\_extract\_charcode\_aux:w

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⟩’.

```
19891 \cs_new:Npn \_tl_analysis_extract_charcode:
19892 {
19893   \exp_after:wN \_tl_analysis_extract_charcode_aux:w
19894   \token_to_meaning:N \l__tl_analysis_token
19895 }
19896 \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.

```
19897 \cs_new:Npn \_\_tl_analysis_cs_space_count:NN #1 #2
19898 {
19899   \exp_after:wN #1
19900   \int_value:w \int_eval:w 0
19901   \exp_after:wN \_\_tl_analysis_cs_space_count:w
19902     \token_to_str:N #2
19903     \fi: \_\_tl_analysis_cs_space_count_end:w ; ~ !
19904 }
19905 \cs_new:Npn \_\_tl_analysis_cs_space_count:w #1 ~
19906 {
19907   \if_false: #1 #1 \fi:
19908   + 1
19909   \_\_tl_analysis_cs_space_count:w
19910 }
19911 \cs_new:Npn \_\_tl_analysis_cs_space_count_end:w ; #1 \fi: #2 !
19912 { \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.)

### 37.4 Plan of attack

Our goal is to produce a token list of the form roughly

```
token 1 \s@__ <catcode 1> <char code 1> \s@__
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.

```

19913 \cs_new_protected:Npn \_tl_analysis:n #1
19914   {
19915     \group_begin:
19916       \group_align_safe_begin:
19917         \_tl_analysis_a:n {#1}
19918         \_tl_analysis_b:n {#1}
19919       \group_align_safe_end:
19920     \group_end:
19921   }

```

(End definition for `\_tl_analysis:n`.)

### 37.5 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.

```

19922 \group_begin:
19923   \char_set_catcode_active:N \^^@ 
19924   \cs_new_protected:Npn \_tl_analysis_disable:n #1
19925   {
19926     \tex_lccode:D 0 = #1 \exp_stop_f:
19927     \tex_lowercase:D { \tex_let:D \^{\tex_lccode:D} } \tex_undefined:D
19928   }
19929 \bool_lazy_or:nnT
19930   { \sys_if_engine_ptex_p: }
19931   { \sys_if_engine_uptex_p: }
19932   {
19933     \cs_gset_protected:Npn \_tl_analysis_disable:n #1
19934     {
19935       \if_int_compare:w 256 > #1 \exp_stop_f:
19936         \tex_lccode:D 0 = #1 \exp_stop_f:
19937         \tex_lowercase:D { \tex_let:D \^{\tex_lccode:D} } \tex_undefined:D
19938       \fi:
19939     }
19940   }
19941 \group_end:

```

(End definition for `\__tl_analysis_disable:n`.)

### 37.6 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 `\__tl_analysis_nesting_int`. This reaches  $-1$  when we read the closing brace.

```
19942 \cs_new_protected:Npn \__tl_analysis_a:n #1
19943   {
19944     \__tl_analysis_disable:n { 32 }
19945     \int_set:Nn \tex_escapechar:D { 92 }
19946     \int_zero:N \l__tl_analysis_normal_int
19947     \int_zero:N \l__tl_analysis_index_int
19948     \int_zero:N \l__tl_analysis_nesting_int
19949     \if_false: { \fi: \__tl_analysis_a_loop:w #1 }
19950     \int_decr:N \l__tl_analysis_index_int
19951 }
```

(End definition for `\_tl_analysis_a:n`.)

`\_tl_analysis_a_loop:w` Read one character and check its type.

```
19952 \cs_new_protected:Npn \_tl_analysis_a_loop:w
19953   { \tex_futurelet:D \l__tl_analysis_token \_tl_analysis_a_type:w }
```

(End definition for `\_tl_analysis_a_loop:w`.)

`\_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).

```
19954 \cs_new_protected:Npn \_tl_analysis_a_type:w
19955   {
19956     \l__tl_analysis_type_int =
19957       \if_meaning:w \l__tl_analysis_token \c_space_token
19958         0
19959       \else:
19960         \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_begin_token
19961           1
19962         \else:
19963           \if_catcode:w \exp_not:N \l__tl_analysis_token \c_group_end_token
19964             - 1
19965           \else:
19966             2
19967           \fi:
19968         \fi:
19969       \fi:
19970       \exp_stop_f:
19971         \if_case:w \l__tl_analysis_type_int
19972           \exp_after:wN \_tl_analysis_a_space:w
19973           \or: \exp_after:wN \_tl_analysis_a_bgroup:w
19974           \or: \exp_after:wN \_tl_analysis_a_safe:N
19975           \else: \exp_after:wN \_tl_analysis_a_egroup:w
19976           \fi:
19977     }
```

(End definition for `\_tl_analysis_a_type:w`.)

`\_tl_analysis_a_space: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.

```

19978 \cs_new_protected:Npn \_\_tl_analysis_a_space:w
19979 {
19980     \tex_afterassignment:D \_\_tl_analysis_a_space_test:w
19981     \exp_after:wN \cs_set_eq:NN
19982     \exp_after:wN \l_\_tl_analysis_char_token
19983     \token_to_str:N
19984 }
19985 \cs_new_protected:Npn \_\_tl_analysis_a_space_test:w
19986 {
19987     \if_meaning:w \l_\_tl_analysis_char_token \c_space_token
19988         \tex_toks:D \l_\_tl_analysis_index_int { \exp_not:n { ~ } }
19989         \_\_tl_analysis_a_store:
19990     \else:
19991         \int_incr:N \l_\_tl_analysis_normal_int
19992     \fi:
19993     \_\_tl_analysis_a_loop:w
19994 }

```

(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_auxii: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.

```

19995 \group_begin:
19996     \char_set_catcode_group_begin:N \^@ % {
19997     \cs_new_protected:Npn \_\_tl_analysis_a_bgroup:w
19998         { \_\_tl_analysis_a_group:nw { \exp_after:wN \^@ \if_false: } \fi: } }
19999     \char_set_catcode_group_end:N \^@ %
20000     \cs_new_protected:Npn \_\_tl_analysis_a_egroup:w
20001         { \_\_tl_analysis_a_group:nw { \if_false: { \fi: \^@ } } % }
20002 \group_end:
20003 \cs_new_protected:Npn \_\_tl_analysis_a_group:nw #1
20004 {
20005     \tex_lccode:D 0 = \_\_tl_analysis_extract_charcode: \scan_stop:
20006     \tex_lowercase:D { \tex_toks:D \l_\_tl_analysis_index_int {#1} }
20007     \if_int_compare:w \tex_lccode:D 0 = \tex_escapechar:D
20008         \int_set:Nn \tex_escapechar:D { 139 - \tex_escapechar:D }

```

```

20009     \fi:
2010      \__tl_analysis_disable:n { \tex_lccode:D 0 }
2011      \tex_futurelet:D \l__tl_analysis_token \__tl_analysis_a_group_aux:w
2012    }
2013 \cs_new_protected:Npn \__tl_analysis_a_group_aux:w
2014  {
2015    \if_meaning:w \l__tl_analysis_token \tex_UNDEFINED:D
2016      \exp_after:wN \__tl_analysis_a_safe:N
2017    \else:
2018      \exp_after:wN \__tl_analysis_a_group_auxii:w
2019    \fi:
2020  }
2021 \cs_new_protected:Npn \__tl_analysis_a_group_auxii:w
2022  {
2023    \tex_afterassignment:D \__tl_analysis_a_group_test:w
2024    \exp_after:wN \cs_set_eq:NN
2025    \exp_after:wN \l__tl_analysis_char_token
2026    \token_to_str:N
2027  }
2028 \cs_new_protected:Npn \__tl_analysis_a_group_test:w
2029  {
2030    \if_charcode:w \l__tl_analysis_token \l__tl_analysis_char_token
2031      \__tl_analysis_a_store:
2032    \else:
2033      \int_incr:N \l__tl_analysis_normal_int
2034    \fi:
2035    \__tl_analysis_a_loop:w
2036  }

```

(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).

```

20037 \cs_new_protected:Npn \__tl_analysis_a_store:
20038 {
20039     \tex_advance:D \l__tl_analysis_nesting_int \l__tl_analysis_type_int
20040     \if_int_compare:w \tex_lccode:D 0 = '\ \exp_stop_f:
20041         \tex_advance:D \l__tl_analysis_type_int \l__tl_analysis_type_int
20042     \fi:
20043     \tex_skip:D \l__tl_analysis_index_int
20044         = \l__tl_analysis_normal_int sp
20045             plus \l__tl_analysis_type_int sp \scan_stop:
20046     \int_incr:N \l__tl_analysis_index_int
20047     \int_zero:N \l__tl_analysis_normal_int
20048     \if_int_compare:w \l__tl_analysis_nesting_int = -1 \exp_stop_f:
20049         \cs_set_eq:NN \__tl_analysis_a_loop:w \scan_stop:
20050     \fi:
20051 }

```

(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.

```

20052 \cs_new_protected:Npn \__tl_analysis_a_safe:N #1
20053 {
20054     \if_charcode:w
20055         \scan_stop:
20056         \exp_after:wN \use_none:n \token_to_str:N #1 \prg_do_nothing:
20057         \scan_stop:
20058         \exp_after:wN \use_i:nn
20059     \else:
20060         \exp_after:wN \use_ii:nn
20061     \fi:
20062     {
20063         \__tl_analysis_disable:n { '#1' }
20064         \int_incr:N \l__tl_analysis_normal_int
20065     }
20066     { \__tl_analysis_cs_space_count:NN \__tl_analysis_a_cs:ww #1 }
20067     \__tl_analysis_a_loop:w
20068 }
20069 \cs_new_protected:Npn \__tl_analysis_a_cs:ww #1; #2;
20070 {

```

```

20071     \if_int_compare:w #1 > 0 \exp_stop_f:
20072         \tex_skip:D \l__tl_analysis_index_int
20073             = \int_eval:n { \l__tl_analysis_normal_int + 1 } sp \exp_stop_f:
20074             \tex_advance:D \l__tl_analysis_index_int #1 \exp_stop_f:
20075         \else:
20076             \tex_advance:D
20077         \fi:
20078     \l__tl_analysis_normal_int #2 \exp_stop_f:
20079 }

```

(End definition for `\__tl_analysis_a_safe:N` and `\__tl_analysis_a_cs:ww`.)

### 37.7 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.

```

20080 \cs_new_protected:Npn \__tl_analysis_b:n #1
20081 {
20082     \tl_gset:Nx \g__tl_analysis_result_tl
20083     {
20084         \__tl_analysis_b_loop:w 0; #1
20085         \prg_break_point:
20086     }
20087 }
20088 \cs_new:Npn \__tl_analysis_b_loop:w #1;
20089 {
20090     \exp_after:wN \__tl_analysis_b_normals:ww
20091     \int_value:w \tex_skip:D #1 ; #1 ;
20092 }

```

(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).

```

20093 \cs_new:Npn \__tl_analysis_b_normals:ww #1;
20094 {
20095     \if_int_compare:w #1 = 0 \exp_stop_f:
20096         \__tl_analysis_b_special:w
20097     \fi:
20098     \__tl_analysis_b_normal:wwN #1;
20099 }
20100 \cs_new:Npn \__tl_analysis_b_normal:wwN #1; #2; #3
20101 {
20102     \exp_not:n { \exp_not:n { #3 } } \s__tl

```

```

20103     \if_charcode:w
20104         \scan_stop:
20105         \exp_after:wN \use_none:n \token_to_str:N #3 \prg_do_nothing:
20106         \scan_stop:
20107         \exp_after:wN \__tl_analysis_b_char:Nww
20108     \else:
20109         \exp_after:wN \__tl_analysis_b_cs:Nww
20110     \fi:
20111     #3 #1; #2;
20112 }

```

(End definition for `\__tl_analysis_b_normals:ww` and `\__tl_analysis_b_normal:ww`.)

`\__tl_analysis_b_char:Nww` 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.

```

20113 \cs_new:Npx \__tl_analysis_b_char:Nww #1
20114 {
20115     \exp_not:N \if_meaning:w #1 \exp_not:N \tex_undefined:D
20116         \token_to_str:N D \exp_not:N \else:
20117         \exp_not:N \if_catcode:w #1 \c_catcode_other_token
20118             \token_to_str:N C \exp_not:N \else:
20119             \exp_not:N \if_catcode:w #1 \c_catcode_letter_token
20120                 \token_to_str:N B \exp_not:N \else:
20121                 \exp_not:N \if_catcode:w #1 \c_math_toggle_token      3
20122                     \exp_not:N \else:
20123                     \exp_not:N \if_catcode:w #1 \c_alignment_token      4
20124                         \exp_not:N \else:
20125                         \exp_not:N \if_catcode:w #1 \c_math_superscript_token 7
20126                             \exp_not:N \else:
20127                             \exp_not:N \if_catcode:w #1 \c_math_subscript_token   8
20128                                 \exp_not:N \else:
20129                                 \exp_not:N \if_catcode:w #1 \c_space_token
20130                                     \token_to_str:N A \exp_not:N \else:
20131                                         6
20132                                         \exp_not:n { \fi: \fi: \fi: \fi: \fi: \fi: }
20133                                         \exp_not:N \int_value:w '#1 \s__tl
20134                                         \exp_not:N \exp_after:wN \exp_not:N \__tl_analysis_b_normals:ww
20135                                         \exp_not:N \int_value:w \exp_not:N \int_eval:w - 1 +
20136 }

```

(End definition for `\__tl_analysis_b_char: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.

```

20137 \cs_new:Npn \__tl_analysis_b_cs:Nww #1
20138 {
20139     0 -1 \s__tl
20140     \__tl_analysis_cs_space_count>NN \__tl_analysis_b_cs_test:ww #1
20141 }
20142 \cs_new:Npn \__tl_analysis_b_cs_test:ww #1 ; #2 ; #3 ; #4 ;
20143 {
20144     \exp_after:wN \__tl_analysis_b_normals:ww

```

```

20145   \int_value:w \int_eval:w
20146   \if_int_compare:w #1 = 0 \exp_stop_f:
20147     #3
20148   \else:
20149     \tex_skip:D \int_eval:n { #4 + #1 } \exp_stop_f:
20150   \fi:
20151   - #2
20152   \exp_after:wN ;
20153   \int_value:w \int_eval:n { #4 + #1 } ;
20154 }

```

(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:w
```

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.

```

20155 \group_begin:
20156   \char_set_catcode_other:N A
20157   \cs_new:Npn \_tl_analysis_b_special:w
20158     \fi: \_tl_analysis_b_normal:wwN 0 ; #1 ;
20159   {
20160     \fi:
20161     \if_int_compare:w #1 = \l__tl_analysis_index_int
20162       \exp_after:wN \prg_break:
20163     \fi:
20164     \tex_the:D \tex_toks:D #1 \s__tl
20165     \if_case:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
20166       \token_to_str:N A
20167       \or: 1
20168       \or: 1
20169       \else: 2
20170       \fi:
20171       \if_int_odd:w \tex_gluestretch:D \tex_skip:D #1 \exp_stop_f:
20172         \exp_after:wN \_tl_analysis_b_special_char:wN \int_value:w
20173       \else:
20174         \exp_after:wN \_tl_analysis_b_special_space:w \int_value:w
20175       \fi:
20176       \int_eval:n { 1 + #1 } \exp_after:wN ;
20177       \token_to_str:N
20178   }
20179 \group_end:
20180 \cs_new:Npn \_tl_analysis_b_special_char:wN #1 ; #2
20181   {
20182     \int_value:w '#2 \s__tl
20183     \_tl_analysis_b_loop:w #1 ;
20184   }
20185 \cs_new:Npn \_tl_analysis_b_special_space:w #1 ; ~
20186   {
20187     32 \s__tl
20188     \_tl_analysis_b_loop:w #1 ;
20189   }

```

(End definition for `\_tl_analysis_b_special:w`, `\_tl_analysis_b_special_char:wN`, and `\_tl_analysis_b_special_space:w`.)

### 37.8 Mapping through the analysis

```
\tl_analysis_map_inline:nn  
\tl_analysis_map_inline:Nn  
  \_tl_analysis_map_inline_aux:Nn  
    \_tl_analysis_map_inline_aux:nnn
```

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 looping macro, which has a name specific to that nesting depth. That looping 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 performs the user's code #2, and loops by calling itself. When the loop ends, remember to decrease the nesting depth.

```
20190 \cs_new_protected:Npn \tl_analysis_map_inline:nn #1  
20191 {  
20192   \_tl_analysis:n {#1}  
20193   \int_gincr:N \g__kernel_prg_map_int  
20194   \exp_args:Nc \_tl_analysis_map_inline_aux:Nn  
20195   { \_tl_analysis_map_inline_ \int_use:N \g__kernel_prg_map_int :wNw }  
20196 }  
20197 \cs_new_protected:Npn \tl_analysis_map_inline:Nn #1  
20198 { \exp_args:No \tl_analysis_map_inline:nn #1 }  
20199 \cs_new_protected:Npn \_tl_analysis_map_inline_aux:Nn #1#2  
20200 {  
20201   \cs_gset_protected:Npn #1 ##1 \s__tl ##2 ##3 \s__tl  
20202   {  
20203     \use_none:n ##2  
20204     \_tl_analysis_map_inline_aux:nnn {##1} {##3} {##2}  
20205   }  
20206   \cs_gset_protected:Npn \_tl_analysis_map_inline_aux:nnn ##1##2##3  
20207   {  
20208     #2  
20209     #1  
20210   }  
20211   \exp_after:wN #1  
20212   \g__tl_analysis_result_tl  
20213   \s__tl { ? \tl_map_break: } \s__tl  
20214   \prg_break_point:Nn \tl_map_break:  
20215   { \int_gdecr:N \g__kernel_prg_map_int }  
20216 }
```

(End definition for `\tl_analysis_map_inline:nn` and others. These functions are documented on page 207.)

### 37.9 Showing the results

```
\tl_analysis_show:N  
\tl_analysis_show:n
```

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.

```
20217 \cs_new_protected:Npn \tl_analysis_show:N #1  
20218 {  
2019 \tl_if_exist:NTF #1  
20220 {  
20221   \exp_args:No \_tl_analysis:n {#1}
```

```

20222     \msg_show:nxxxx { LaTeX / kernel } { show-tl-analysis }
20223         { \token_to_str:N #1 } { \_tl_analysis_show: } { } { }
20224     }
20225     { \tl_show:N #1 }
20226   }
20227 \cs_new_protected:Npn \tl_analysis_show:n #
20228   {
20229     \_tl_analysis:n {#1}
2030     \msg_show:nxxxx { LaTeX / kernel } { show-tl-analysis }
2031     { } { \_tl_analysis_show: } { } { }
2032   }

```

(End definition for `\tl_analysis_show:N` and `\tl_analysis_show:n`. These functions are documented on page 207.)

`\_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.

```

20233 \cs_new:Npn \_tl_analysis_show:
20234   {
20235     \exp_after:wN \_tl_analysis_show_loop:wNw \g__tl_analysis_result_tl
20236     \s__tl { ? \prg_break: } \s__tl
20237     \prg_break_point:
20238   }
20239 \cs_new:Npn \_tl_analysis_show_loop:wNw #1 \s__tl #2 #3 \s__tl
20240   {
20241     \use_none:n #2
20242     \iow_newline: > \use:nn { ~ } { ~ }
20243     \if_int_compare:w "#2 = 0 \exp_stop_f:
20244       \exp_after:wN \_tl_analysis_show_cs:n
20245     \else:
20246       \if_int_compare:w "#2 = 13 \exp_stop_f:
20247         \exp_after:wN \exp_after:wN
20248         \exp_after:wN \_tl_analysis_show_active:n
20249       \else:
20250         \exp_after:wN \exp_after:wN
20251         \exp_after:wN \_tl_analysis_show_normal:n
20252       \fi:
20253     \fi:
20254     {#1}
20255     \_tl_analysis_show_loop:wNw
20256   }

```

(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.

```

20257 \cs_new:Npn \_tl_analysis_show_normal:n #1
20258   {
20259     \exp_after:wN \token_to_str:N #1 ~
20260     ( \exp_after:wN \token_to_meaning:N #1 )
20261   }

```

(End definition for `\_tl_analysis_show_normal:n`.)

`\_tl_analysis_show_value:N` This expands to the value of #1 if it has any.

```
20262 \cs_new:Npn \_tl_analysis_show_value:N #1
20263 {
20264     \token_if_expandable:NF #1
20265     {
20266         \token_if_chardef:NTF      #1 \prg_break: { }
20267         \token_if_mathchardef:NTF  #1 \prg_break: { }
20268         \token_if_dim_register:NTF #1 \prg_break: { }
20269         \token_if_int_register:NTF #1 \prg_break: { }
20270         \token_if_skip_register:NTF #1 \prg_break: { }
20271         \token_if_toks_register:NTF #1 \prg_break: { }
20272         \use_none:nnn
20273         \prg_break_point:
20274         \use:n { \exp_after:wN = \tex_the:D #1 }
20275     }
20276 }
```

(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`.

```
20277 \cs_new:Npn \_tl_analysis_show_cs:n #1
20278 { \exp_args:No \_tl_analysis_show_long:nn {#1} { control-sequence= } }
20279 \cs_new:Npn \_tl_analysis_show_active:n #1
20280 { \exp_args:No \_tl_analysis_show_long:nn {#1} { active-character= } }
20281 \cs_new:Npn \_tl_analysis_show_long:nn #1
20282 {
20283     \_tl_analysis_show_long_aux:oofn
20284     { \token_to_str:N #1 }
20285     { \token_to_meaning:N #1 }
20286     { \_tl_analysis_show_value:N #1 }
20287 }
20288 \cs_new:Npn \_tl_analysis_show_long_aux:nnnn #1#2#3#4
20289 {
20290     \int_compare:nNnTF
20291     { \str_count:n { #1 ~ ( #4 #2 #3 ) } }
20292     > { \l_iow_line_count_int - 3 }
20293     {
20294         \str_range:nnn { #1 ~ ( #4 #2 #3 ) } { 1 }
20295         {
20296             \l_iow_line_count_int - 3
20297             - \str_count:N \c__tl_analysis_show_etc_str
20298         }
20299         \c__tl_analysis_show_etc_str
20300     }
20301     { #1 ~ ( #4 #2 #3 ) }
20302 }
```

`\cs_generate_variant:Nn` `\_tl_analysis_show_long_aux:nnnn` { oof }

(End definition for `\_tl_analysis_show_cs:n` and others.)

## 37.10 Messages

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.

```
20304 \tl_const:Nx \c__tl_analysis_show_etc_str % (
20305   { \token_to_str:N \ETC.) }

(End definition for \c__tl_analysis_show_etc_str.)

20306 \__kernel_msg_new:nnn { kernel } { show-tl-analysis }
20307   {
20308     The~token~list~ \tl_if_empty:nF {#1} { #1 ~ }
20309     \tl_if_empty:nTF {#2}
20310       { is~empty }
20311       { contains-the~tokens: #2 }
20312   }
```

## 37.11 Deprecated functions

\tl\_show\_analysis:N Simple renames.

```
20313 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 }
20314   { \tl_analysis_show:N }
20315 \cs_new_protected:Npn \tl_show_analysis:N #1
20316   { \tl_analysis_show:N #1 }
20317 \__kernel_patch_deprecation:nnNNpn { 2019-12-31 }
20318   { \tl_analysis_show:n }
20319 \cs_new_protected:Npn \tl_show_analysis:n #1
20320   { \tl_analysis_show:n {#1} }
```

(End definition for \tl\_show\_analysis:N and \tl\_show\_analysis:n.)

20321 ⟨/initex | package⟩

## 38 I3regex implementation

```
20322 ⟨*initex | package⟩
20323 ⟨@@=regex⟩
```

### 38.1 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  $\text{min\_pos} - 1 \leq \langle position \rangle \leq \text{max\_pos}$ . The lowest and highest positions correspond to imaginary begin and end markers (with inaccessible category code and character code).
- *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  $\text{min\_state} \leq \langle state \rangle < \text{max\_state}$ .
- *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  $\backslash\!l\_regex\_step\_int$  is a unique id for all the steps of the matching algorithm.

We use `I3intarray` to manipulate arrays of integers (stored into some dimension registers in scaled points). 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  $\langle state \rangle$  of the NFA. When matching,

- `\g__regex_state_active_intarray` holds the last  $\langle step \rangle$  in which each  $\langle state \rangle$  was active.
- `\g__regex_thread_state_intarray` maps each  $\langle thread \rangle$  (with  $\text{min\_active} \leq \langle thread \rangle < \text{max\_active}$ ) to the  $\langle state \rangle$  in which the  $\langle thread \rangle$  currently is. The  $\langle threads \rangle$  are ordered starting from the best to the least preferred.
- `\toks(thread)` holds the submatch information for the  $\langle thread \rangle$ , as the contents of a property list.
- `\g__regex_charcode_intarray` and `\g__regex_catcode_intarray` hold the character codes and category codes of tokens at each  $\langle position \rangle$  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.
- `\toks(position)` holds  $\langle tokens \rangle$  which o- and x-expand to the  $\langle position \rangle$ -th token in the query.
- `\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.

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.

## 38.2 Helpers

`\__regex_int_eval:w` Access the primitive: performance is key here, so we do not use the slower route *via* `\int_eval:n`.

```

20324 \cs_new_eq:NN \__regex_int_eval:w \tex_numexpr:D
20325 %     \end{macrocode}
20326 % \end{macro}
20327 %
20328 % \begin{macro}{\__regex_standard_escapechar:}
20329 %   Make the \tn{escapechar} into the standard backslash.
20330 %   \begin{macrocode}
20331 \cs_new_protected:Npn \__regex_standard_escapechar:
20332 { \int_set:Nn \tex_escapechar:D { '\\" } }

(End definition for \__regex_int_eval:w.)
```

`\__regex_toks_use:w` Unpack a `\toks` given its number.

```

20333 \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.

```

20334 \cs_new_protected:Npn \__regex_toks_clear:N #1
20335 { \__regex_toks_set:Nn #1 { } }
20336 \cs_new_eq:NN \__regex_toks_set:Nn \tex_toks:D
20337 \cs_new_protected:Npn \__regex_toks_set:No #1
20338 { \__regex_toks_set:Nn #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`.

```

20339 \cs_new_protected:Npn \__regex_toks_memcpy:NNn #1#2#3
20340 {
20341     \prg_replicate:nn {#3}
20342     {
20343         \tex_toks:D #1 = \tex_toks:D #2
20344         \int_incr:N #1
20345         \int_incr:N #2
20346     }
20347 }
```

(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`.

```
20348 \cs_new_protected:Npn \_\_regex_toks_put_left:Nx #1#2
20349  {
20350      \cs_set:Npx \_\_regex_tmp:w { #2 }
20351      \tex_toks:D #1 \exp_after:wN \exp_after:wN \exp_after:wN
20352          { \exp_after:wN \_\_regex_tmp:w \tex_the:D \tex_toks:D #1 }
20353  }
20354 \cs_new_protected:Npn \_\_regex_toks_put_right:Nx #1#2
20355  {
20356      \cs_set:Npx \_\_regex_tmp:w {#2}
20357      \tex_toks:D #1 \exp_after:wN
20358          { \tex_the:D \tex_toks:D \exp_after:wN #1 \_\_regex_tmp:w }
20359  }
20360 \cs_new_protected:Npn \_\_regex_toks_put_right:Nn #1#2
20361      { \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.

```
20362 \cs_new:Npn \_\_regex_curr_cs_to_str:
20363  {
20364      \exp_after:wN \exp_after:wN \exp_after:wN \cs_to_str:N
20365      \tex_the:D \tex_toks:D \l_\_\_regex_curr_pos_int
20366  }
```

(End definition for `\_\_regex_curr_cs_to_str:..`)

### 38.2.1 Constants and variables

```
\_\_regex_tmp:w
```

Temporary function used for various short-term purposes.

```
20367 \cs_new:Npn \_\_regex_tmp:w { }
```

(End definition for `\_\_regex_tmp:w`.)

```
\l_\_\_regex_internal_a_tl
\l_\_\_regex_internal_b_tl
\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
```

Temporary variables used for various purposes.

```
20368 \tl_new:N \l_\_\_regex_internal_a_tl
20369 \tl_new:N \l_\_\_regex_internal_b_tl
20370 \int_new:N \l_\_\_regex_internal_a_int
20371 \int_new:N \l_\_\_regex_internal_b_int
20372 \int_new:N \l_\_\_regex_internal_c_int
20373 \bool_new:N \l_\_\_regex_internal_bool
20374 \seq_new:N \l_\_\_regex_internal_seq
20375 \tl_new:N \g_\_\_regex_internal_tl
```

(End definition for `\l_\_\_regex_internal_a_tl` and others.)

```
\l_\_\_regex_build_tl
```

This temporary variable is specifically for use with the `tl_build` machinery.

```
20376 \tl_new:N \l_\_\_regex_build_tl
```

(End definition for `\l_regex_build_tl`.)

`\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`.

```
20377 \tl_const:Nn \c_regex_no_match_regex
20378 {
20379   \__regex_branch:n
20380   { \__regex_class:NnnN \c_true_bool { } { 1 } { 0 } \c_true_bool }
20381 }
```

(End definition for `\c_regex_no_match_regex`.)

`\g_regexCharCode_intarray` `\g_regexCatcode_intarray` `\g_regexBalance_intarray` The first thing we do when matching is to go once through the query token list and store the information for each token into `\g_regexCharCode_intarray`, `\g_regexCatcode_intarray` and `\toks` registers. We also store the balance of begin-group/end-group characters into `\g_regexBalance_intarray`.

```
20382 \intarray_new:Nn \g_regexCharCode_intarray { 65536 }
20383 \intarray_new:Nn \g_regexCatcode_intarray { 65536 }
20384 \intarray_new:Nn \g_regexBalance_intarray { 65536 }
```

(End definition for `\g_regexCharCode_intarray`, `\g_regexCatcode_intarray`, and `\g_regexBalance_intarray`.)

`\l_regexBalance_int` During this phase, `\l_regexBalance_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.

```
20385 \int_new:N \l_regexBalance_int
```

(End definition for `\l_regexBalance_int`.)

`\l_regexCsName_tl` This variable is used in `\__regex_item_cs:n` to store the csname of the currently-tested token when the regex contains a sub-regex for testing csnames.

```
20386 \tl_new:N \l_regexCsName_tl
```

(End definition for `\l_regexCsName_tl`.)

### 38.2.2 Testing characters

`\c_regexAsciiMinInt` `\c_regexAsciiMaxControlInt` `\c_regexAsciiMaxInt`   
 20387 `\int_const:Nn \c_regexAsciiMinInt { 0 }`  
 20388 `\int_const:Nn \c_regexAsciiMaxControlInt { 31 }`  
 20389 `\int_const:Nn \c_regexAsciiMaxInt { 127 }`  
  
(End definition for `\c_regexAsciiMinInt`, `\c_regexAsciiMaxControlInt`, and `\c_regexAsciiMaxInt`.)

`\c_regexAsciiLowerInt`   
 20390 `\int_const:Nn \c_regexAsciiLowerInt { 'a - 'A }`  
  
(End definition for `\c_regexAsciiLowerInt`.)

`\__regexBreakPoint:TF` `\__regexBreakTrue: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

```

⟨test1⟩ ... ⟨testn

```

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.

```

20391 \cs_new_protected:Npn \__regex_break_true:w
20392     #1 \__regex_break_point:TF #2 #3 {#2}
20393 \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.

```

20394 \cs_new_protected:Npn \__regex_item_reverse:n #1
20395     {
20396         #1
20397         \__regex_break_point:TF { } \__regex_break_true:w
20398     }

```

(End definition for `\__regex_item_reverse:n`.)

`\__regex_item_caseful_equal:n` Simple comparisons triggering `\__regex_break_true:w` when true.

```

20399 \cs_new_protected:Npn \__regex_item_caseful_equal:n #1
20400     {
20401         \if_int_compare:w #1 = \l__regex_curr_char_int
20402             \exp_after:wN \__regex_break_true:w
20403         \fi:
20404     }
20405 \cs_new_protected:Npn \__regex_item_caseful_range:nn #1 #2
20406     {
20407         \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
20408             \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
20409                 \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
20410             \fi:
20411         \fi:
20412     }

```

(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 `current_char` and on the `case_changed_char`. Before doing the second set of tests, we make sure that `case_changed_char` has been computed.

```

20413 \cs_new_protected:Npn \__regex_item_caseless_equal:n #1
20414     {
20415         \if_int_compare:w #1 = \l__regex_curr_char_int
20416             \exp_after:wN \__regex_break_true:w
20417         \fi:
20418         \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
20419             \__regex_compute_case_changed_char:
20420         \fi:
20421         \if_int_compare:w #1 = \l__regex_case_changed_char_int
20422             \exp_after:wN \__regex_break_true:w

```

```

20423     \fi:
20424   }
20425 \cs_new_protected:Npn \__regex_item_caseless_range:nn #1 #2
20426   {
20427     \reverse_if:N \if_int_compare:w #1 > \l__regex_curr_char_int
20428       \reverse_if:N \if_int_compare:w #2 < \l__regex_curr_char_int
20429         \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
20430       \fi:
20431     \fi:
20432     \if_int_compare:w \l__regex_case_changed_char_int = \c_max_int
20433       \__regex_compute_case_changed_char:
20434     \fi:
20435     \reverse_if:N \if_int_compare:w #1 > \l__regex_case_changed_char_int
20436       \reverse_if:N \if_int_compare:w #2 < \l__regex_case_changed_char_int
20437         \exp_after:wN \exp_after:wN \exp_after:wN \__regex_break_true:w
20438       \fi:
20439     \fi:
20440   }

```

(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 (or rather, when it is set to the marker value `\c_max_int`). 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.

```

20441 \cs_new_protected:Npn \__regex_compute_case_changed_char:
20442   {
20443     \int_set_eq:NN \l__regex_case_changed_char_int \l__regex_curr_char_int
20444     \if_int_compare:w \l__regex_curr_char_int > 'Z \exp_stop_f:
20445       \if_int_compare:w \l__regex_curr_char_int > 'z \exp_stop_f: \else:
20446         \if_int_compare:w \l__regex_curr_char_int < 'a \exp_stop_f: \else:
20447           \int_sub:Nn \l__regex_case_changed_char_int
20448             { \c__regex_ascii_lower_int }
20449         \fi:
20450       \fi:
20451     \else:
20452       \if_int_compare:w \l__regex_curr_char_int < 'A \exp_stop_f: \else:
20453         \int_add:Nn \l__regex_case_changed_char_int
20454           { \c__regex_ascii_lower_int }
20455       \fi:
20456     \fi:
20457   }

```

(End definition for `\__regex_compute_case_changed_char:.`)

`\__regex_item_equal:n` 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.

```

20458 \cs_new_eq:NN \__regex_item_equal:n ?
20459 \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.

```

20460 \cs_new_protected:Npn \_\_regex_item_catcode:
20461   {
20462     "
20463     \if_case:w \l_\_regex_curr_catcode_int
20464       1      \or: 4      \or: 10      \or: 40
20465       \or: 100     \or:        \or: 1000     \or: 4000
20466       \or: 10000    \or:        \or: 100000   \or: 400000
20467       \or: 1000000 \or: 4000000 \else: 1*0
20468     \fi:
20469   }
20470 \cs_new_protected:Npn \_\_regex_item_catcode:nT #1
20471   {
20472     \if_int_odd:w \int_eval:n { #1 / \_\_regex_item_catcode: } \exp_stop_f:
20473       \exp_after:wN \use:n
20474     \else:
20475       \exp_after:wN \use_none:n
20476     \fi:
20477   }
20478 \cs_new_protected:Npn \_\_regex_item_catcode_reverse:nT #1#2
20479   { \_\_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.

```

20480 \cs_new_protected:Npn \_\_regex_item_exact:nn #1#2
20481   {
20482     \if_int_compare:w #1 = \l_\_regex_curr_catcode_int
20483       \if_int_compare:w #2 = \l_\_regex_curr_char_int
20484         \exp_after:wN \exp_after:wN \exp_after:wN \_\_regex_break_true:w
20485       \fi:
20486     \fi:
20487   }
20488 \cs_new_protected:Npn \_\_regex_item_exact_cs:n #1
20489   {
20490     \int_compare:nNnTF \l_\_regex_curr_catcode_int = 0
20491       {
20492         \tl_set:Nx \l_\_regex_internal_a_tl
20493           { \scan_stop: \_\_regex_curr_cs_to_str: \scan_stop: }
20494         \tl_if_in:noTF { \scan_stop: #1 \scan_stop: }
20495           \l_\_regex_internal_a_tl
20496           { \_\_regex_break_true:w } { }
20497       }
20498     { }
20499   }
```

(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. The three `\exp_after:wN` expand the contents of the `\toks<current position>` (of the form `\exp_not:n {<control sequence>}`) to `{control sequence}`. We store the cs name before building states for the cs, as those states may overlap with toks registers storing the user's input.

```

20500 \cs_new_protected:Npn __regex_item_cs:n #1
20501   {
20502     \int_compare:nNnT \l__regex_curr_catcode_int = 0
20503     {
20504       \group_begin:
20505         \tl_set:Nx \l__regex_cs_name_tl { \__regex_curr_cs_to_str: }
20506         \__regex_single_match:
20507         \__regex_disable_submatches:
20508         \__regex_build_for_cs:n {#1}
20509         \bool_set_eq:NN \l__regex_saved_success_bool
20510           \g__regex_success_bool
20511         \exp_args:NV \__regex_match(cs:n) \l__regex_cs_name_tl
20512         \if_meaning:w \c_true_bool \g__regex_success_bool
20513           \group_insert_after:N \__regex_break_true:w
20514         \fi:
20515         \bool_gset_eq:NN \g__regex_success_bool
20516           \l__regex_saved_success_bool
20517       \group_end:
20518     }
20519   }

```

(End definition for `\__regex_item_cs:n`.)

### 38.2.3 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]`, `\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.

```

20520 \cs_new_protected:Npn __regex_prop_d:
20521   { \__regex_item_caseful_range:nn { '0 } { '9 } }
20522 \cs_new_protected:Npn __regex_prop_h:
20523   {
20524     \__regex_item_caseful_equal:n { '\ }
20525     \__regex_item_caseful_equal:n { '\^\^I }
20526   }
20527 \cs_new_protected:Npn __regex_prop_s:
20528   {
20529     \__regex_item_caseful_equal:n { '\ }
20530     \__regex_item_caseful_equal:n { '\^\^I }
20531     \__regex_item_caseful_equal:n { '\^\^J }
20532     \__regex_item_caseful_equal:n { '\^\^L }
20533     \__regex_item_caseful_equal:n { '\^\^M }
20534   }
20535 \cs_new_protected:Npn __regex_prop_v:
20536   { \__regex_item_caseful_range:nn { '\^\^J } { '\^\^M } } % lf, vtab, ff, cr
20537 \cs_new_protected:Npn __regex_prop_w:
20538   {

```

```

20539     \__regex_item_caseful_range:nn { 'a' } { 'z' }
20540     \__regex_item_caseful_range:nn { 'A' } { 'Z' }
20541     \__regex_item_caseful_range:nn { '0' } { '9' }
20542     \__regex_item_caseful_equal:n { '_' }
20543   }
20544 \cs_new_protected:Npn \__regex_prop_N:
20545   {
20546     \__regex_item_reverse:n
20547     { \__regex_item_caseful_equal:n { '\^J' } }
20548   }

```

(End definition for `\__regex_prop_d:` and others.)

```

\__regex_posix_alnum: POSIX properties. No surprise.
\__regex_posix_alpha: 20549 \cs_new_protected:Npn \__regex_posix_alnum:
\__regex_posix_ascii: 20550   { \__regex_posix_alpha: \__regex_posix_digit: }
\__regex_posix_blank: 20551 \cs_new_protected:Npn \__regex_posix_alpha:
\__regex_posix_cntrl: 20552   { \__regex_posix_lower: \__regex_posix_upper: }
\__regex_posix_digit: 20553 \cs_new_protected:Npn \__regex_posix_ascii:
\__regex_posix_graph: 20554   {
\__regex_posix_lower: 20555     \__regex_item_caseful_range:nn
\__regex_posix_print: 20556     \c__regex_ascii_min_int
\__regex_posix_punct: 20557     \c__regex_ascii_max_int
\__regex_posix_space: 20558   }
20559 \cs_new_eq:NN \__regex_posix_blank: \__regex_prop_h:
\__regex_posix_upper: 20560 \cs_new_protected:Npn \__regex_posix_cntrl:
\__regex_posix_word: 20561   {
\__regex_posix_xdigit: 20562     \__regex_item_caseful_range:nn
20563     \c__regex_ascii_min_int
20564     \c__regex_ascii_max_control_int
20565     \__regex_item_caseful_equal:n \c__regex_ascii_max_int
20566   }
20567 \cs_new_eq:NN \__regex_posix_digit: \__regex_prop_d:
20568 \cs_new_protected:Npn \__regex_posix_graph:
20569   { \__regex_item_caseful_range:nn { '!' } { '\~{}' } }
20570 \cs_new_protected:Npn \__regex_posix_lower:
20571   { \__regex_item_caseful_range:nn { 'a' } { 'z' } }
20572 \cs_new_protected:Npn \__regex_posix_print:
20573   { \__regex_item_caseful_range:nn { '\`{}' } { '\^{}' } }
20574 \cs_new_protected:Npn \__regex_posix_punct:
20575   {
20576     \__regex_item_caseful_range:nn { '!' } { '/' }
20577     \__regex_item_caseful_range:nn { ':' } { '0' }
20578     \__regex_item_caseful_range:nn { '[' } { '[' }
20579     \__regex_item_caseful_range:nn { '\{ ' } { '\~{}' }
20580   }
20581 \cs_new_protected:Npn \__regex_posix_space:
20582   {
20583     \__regex_item_caseful_equal:n { '\~{}' }
20584     \__regex_item_caseful_range:nn { '\^I' } { '\^M' }
20585   }
20586 \cs_new_protected:Npn \__regex_posix_upper:
20587   { \__regex_item_caseful_range:nn { 'A' } { 'Z' } }
20588 \cs_new_eq:NN \__regex_posix_word: \__regex_prop_w:

```

```

20589 \cs_new_protected:Npn \__regex_posix_xdigit:
20590 {
20591     \__regex_posix_digit:
20592     \__regex_item_caseful_range:nn { 'A } { 'F }
20593     \__regex_item_caseful_range:nn { 'a } { 'f }
20594 }

```

(End definition for `\__regex_posix_alnum`: and others.)

### 38.2.4 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_tl`, 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).

```

20595 \__kernel_patch:nnNnpn
20596 {
20597     \__regex_trace_push:nnN { regex } { 1 } \__regex_escape_use:nnnn
20598     \group_begin:
20599         \tl_set:Nx \l__regex_internal_a_tl
20600         { \__regex_trace_pop:nnN { regex } { 1 } \__regex_escape_use:nnnn }
20601         \use_none:nnn
20602     }
20603     { }
20604 \cs_new_protected:Npn \__regex_escape_use:nnnn #1#2#3#4
20605 {
20606     \group_begin:
20607         \tl_clear:N \l__regex_internal_a_tl
20608         \cs_set:Npn \__regex_escape_unescaped:N ##1 { #1 }
20609         \cs_set:Npn \__regex_escape_escaped:N ##1 { #2 }
20610         \cs_set:Npn \__regex_escape_raw:N ##1 { #3 }
20611         \__regex_standard_escapechar:
20612         \tl_gset:Nx \g__regex_internal_tl
20613         { \__kernel_str_to_other_fast:n {#4} }
20614         \tl_put_right:Nx \l__regex_internal_a_tl
20615         {
20616             \exp_after:wn \__regex_escape_loop:N \g__regex_internal_tl

```

```

20617         { break } \prg_break_point:
20618     }
20619     \exp_after:wN
20620     \group_end:
20621     \l__regex_internal_a_tl
20622 }
```

(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”.

```

20623 \cs_new:Npn \_regex_escape_loop:N #1
20624 {
20625     \cs_if_exist_use:cF { \_regex_escape_\token_to_str:N #1:w }
20626     { \_regex_escape_unescaped:N #1 }
20627     \_regex_escape_loop:N
20628 }
20629 \cs_new:cpn { \_regex_escape_\c_backslash_str :w }
20630     \_regex_escape_loop:N #1
20631 {
20632     \cs_if_exist_use:cF { \_regex_escape_/\token_to_str:N #1:w }
20633     { \_regex_escape_escaped:N #1 }
20634     \_regex_escape_loop:N
20635 }
```

(End definition for `\_regex_escape_loop:N` and `\_regex_escape_\:w`.)

`\_regex_escape_unescaped:N` `\_regex_escape_escaped:N` Those functions are never called before being given a new meaning, so their definitions here don’t matter.

```

20636 \cs_new_eq:NN \_regex_escape_unescaped:N ?
20637 \cs_new_eq:NN \_regex_escape_escaped:N ?
20638 \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_break:w` `\_regex_escape_/break: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.

```

20639 \cs_new_eq:NN \_regex_escape_break:w \prg_break:
20640 \cs_new:cpn { \_regex_escape_/break:w }
20641 {
20642     \_kernel_msg_expandable_error:nn { kernel } { trailing-backslash }
20643     \prg_break:
20644 }
20645 \cs_new:cpn { \_regex_escape_~:w } { }
20646 \cs_new:cpn { \_regex_escape_/_a:w }
20647     { \exp_not:N \_regex_escape_raw:N \iow_char:N \^G }
20648 \cs_new:cpn { \_regex_escape_/_t:w }
20649     { \exp_not:N \_regex_escape_raw:N \iow_char:N \^I }
20650 \cs_new:cpn { \_regex_escape_/_n:w }
20651     { \exp_not:N \_regex_escape_raw:N \iow_char:N \^J }
20652 \cs_new:cpn { \_regex_escape_/_f:w }
20653     { \exp_not:N \_regex_escape_raw:N \iow_char:N \^L }
```

```

20654 \cs_new:cpx { __regex_escape_/r:w }
20655   { \exp_not:N __regex_escape_raw:N \iow_char:N \^M }
20656 \cs_new:cpx { __regex_escape_/e:w }
20657   { \exp_not:N __regex_escape_raw:N \iow_char:N \^[] }

```

(End definition for `\__regex_escape_break: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.

```

20658 \cs_new:cpn { __regex_escape/_x:w } __regex_escape_loop:N
20659   {
20660     \exp_after:wN __regex_escape_x_end:w
20661     \int_value:w "0 __regex_escape_x_test:N
20662   }
20663 \cs_new:Npn __regex_escape_x_end:w #1 ;
20664   {
20665     \int_compare:nNnTF {#1} > \c_max_char_int
20666     {
20667       __kernel_msg_expandable_error:nnff { kernel } { x-overflow }
20668       {#1} { \int_to_Hex:n {#1} }
20669     }
20670   {
20671     \exp_last_unbraced:Nf __regex_escape_raw:N
20672     { \char_generate:nn {#1} { 12 } }
20673   }
20674 }

```

(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`.

```

20675 \cs_new:Npn __regex_escape_x_test:N #1
20676   {
20677     \str_if_eq:nnTF {#1} { break } { ; }
20678   {
20679     \if_charcode:w \c_space_token #1
20680       \exp_after:wN __regex_escape_x_test:N
20681     \else:
20682       \exp_after:wN __regex_escape_x_testii:N
20683       \exp_after:wN #1
20684     \fi:
20685   }
20686   }
20687 \cs_new:Npn __regex_escape_x_testii:N #1
20688   {
20689     \if_charcode:w \c_left_brace_str #1
20690       \exp_after:wN __regex_escape_x_loop:N
20691     \else:
20692       __regex_hexadecimal_use:NTF #1
20693       { \exp_after:wN __regex_escape_x:N }

```

```

20694     { ; \exp_after:wN \__regex_escape_loop:N \exp_after:wN #1 }
20695     \fi:
20696   }

```

(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.

```

20697 \cs_new:Npn \__regex_escape_x:N #1
20698   {
20699     \str_if_eq:nnTF {#1} { break } { ; }
20700     {
20701       \__regex_hexadecimal_use:NTF #1
20702         { ; \__regex_escape_loop:N }
20703         { ; \__regex_escape_loop:N #1 }
20704     }
20705   }

```

(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.

```

20706 \cs_new:Npn \__regex_escape_x_loop:N #1
20707   {
20708     \str_if_eq:nnTF {#1} { break }
20709     { ; \__regex_escape_x_loop_error:n { } {#1} }
20710     {
20711       \__regex_hexadecimal_use:NTF #1
20712         { \__regex_escape_x_loop:N }
20713         {
20714           \token_if_eq_charcode:NNTF \c_space_token #1
20715             { \__regex_escape_x_loop:N }
20716             {
20717               ;
20718               \exp_after:wN
20719                 \token_if_eq_charcode:NNTF \c_right_brace_str #1
20720                   { \__regex_escape_loop:N }
20721                   { \__regex_escape_x_loop_error:n {#1} }
20722               }
20723             }
20724           }
20725         }
20726 \cs_new:Npn \__regex_escape_x_loop_error:n #1
20727   {
20728     \__kernel_msg_expandable_error:nnn { kernel } { x-missing-rbrace } {#1}
20729     \__regex_escape_loop:N #1
20730   }

```

(End definition for `\__regex_escape_x_loop:N` and `\__regex_escape_x_loop_error:n`.)

`\__regex_hexadecimal_use:NTF` T<sub>E</sub>X detects uppercase hexadecimal digits for us but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

20731 \prg_new_conditional:Npnn \__regex_hexadecimal_use:N #1 { TF }
20732   {
20733     \if_int_compare:w 1 < "1 \token_to_str:N #1 \exp_stop_f:
20734       #1 \prg_return_true:

```

```

20735     \else:
20736         \if_case:w
20737             \int_eval:n { \exp_after:wN ` \token_to_str:N #1 - `a }
20738                 A
20739                 \or: B
20740                 \or: C
20741                 \or: D
20742                 \or: E
20743                 \or: F
20744             \else:
20745                 \prg_return_false:
20746                 \exp_after:wN \use_none:n
20747             \fi:
20748             \prg_return_true:
20749         \fi:
20750     }

```

(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).

```

20751 \prg_new_conditional:Npnn \_regex_char_if_special:N #1 { TF }
20752 {
20753     \if_int_compare:w '#1 > 'Z \exp_stop_f:
20754         \if_int_compare:w '#1 > 'z \exp_stop_f:
20755             \if_int_compare:w '#1 < \c_regex_ascii_max_int
20756                 \prg_return_true: \else: \prg_return_false: \fi:
20757             \else:
20758                 \if_int_compare:w '#1 < 'a \exp_stop_f:
20759                     \prg_return_true: \else: \prg_return_false: \fi:
20760             \fi:
20761         \else:
20762             \if_int_compare:w '#1 > '9 \exp_stop_f:
20763                 \if_int_compare:w '#1 < 'A \exp_stop_f:
20764                     \prg_return_true: \else: \prg_return_false: \fi:
20765                 \else:
20766                     \if_int_compare:w '#1 < '0 \exp_stop_f:
20767                         \if_int_compare:w '#1 < '\ \exp_stop_f:
20768                             \prg_return_false: \else: \prg_return_true: \fi:
20769                         \else: \prg_return_false: \fi:
20770                     \fi:

```

```

20771      \fi:
20772  }
20773 \prg_new_conditional:Npnn \__regex_char_if_alphanumeric:N #1 { TF }
20774  {
20775      \if_int_compare:w '#1 > 'Z \exp_stop_f:
20776          \if_int_compare:w '#1 > 'z \exp_stop_f:
20777              \prg_return_false:
20778          \else:
20779              \if_int_compare:w '#1 < 'a \exp_stop_f:
20780                  \prg_return_false: \else: \prg_return_true: \fi:
20781      \fi:
20782  \else:
20783      \if_int_compare:w '#1 > '9 \exp_stop_f:
20784          \if_int_compare:w '#1 < 'A \exp_stop_f:
20785              \prg_return_false: \else: \prg_return_true: \fi:
20786          \else:
20787              \if_int_compare:w '#1 < '0 \exp_stop_f:
20788                  \prg_return_false: \else: \prg_return_true: \fi:
20789          \fi:
20790      \fi:
20791  }

```

(End definition for `\__regex_char_if_alphanumeric:NTF` and `\__regex_char_if_special:NTF`.)

### 38.3 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:NnnnN`  $\langle\text{boolean}\rangle$   $\{\langle\text{tests}\rangle\}$   $\{\langle\text{min}\rangle\}$   $\{\langle\text{more}\rangle\}$   $\langle\text{lazyness}\rangle$
- `\__regex_group:nnnN`  $\{\langle\text{branches}\rangle\}$   $\{\langle\text{min}\rangle\}$   $\{\langle\text{more}\rangle\}$   $\langle\text{lazyness}\rangle$ , also `\__regex_group_no_capture:nnnN` and `\__regex_group_resetting:nnnN` with the same syntax.
- `\__regex_branch:n`  $\{\langle\text{contents}\rangle\}$
- `\__regex_command_K`:
- `\__regex_assertion:Nn`  $\langle\text{boolean}\rangle$   $\{\langle\text{assertion test}\rangle\}$ , where the  $\langle\text{assertion test}\rangle$  is `\__regex_b_test:` or  $\{\langle\text{__regex_anchor:N}\rangle\langle\text{integer}\rangle\}$

Tests can be the following:

- `\__regex_item_caseful_equal:n`  $\{\langle\text{char code}\rangle\}$
- `\__regex_item_caseless_equal:n`  $\{\langle\text{char code}\rangle\}$
- `\__regex_item_caseful_range:nn`  $\{\langle\text{min}\rangle\}$   $\{\langle\text{max}\rangle\}$
- `\__regex_item_caseless_range:nn`  $\{\langle\text{min}\rangle\}$   $\{\langle\text{max}\rangle\}$
- `\__regex_item_catcode:nT`  $\{\langle\text{catcode bitmap}\rangle\}$   $\{\langle\text{tests}\rangle\}$

- `\_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*}

### 38.3.1 Variables used when compiling

We make sure to open the same number of groups as we close.

20792 `\int_new:N \l_regex_group_level_int`

*(End definition for `\l_regex_group_level_int`.)*

While compiling, ten modes are recognized, labelled  $-63, -23, -6, -2, 0, 2, 3, 6, 23, 63$ . See section 38.3.3. We only define some of these as constants.

20793 `\int_new:N \l_regex_mode_int`  
 20794 `\int_const:Nn \c_regex_cs_in_class_mode_int` { -6 }  
 20795 `\int_const:Nn \c_regex_cs_mode_int` { -2 }  
 20796 `\int_const:Nn \c_regex_outer_mode_int` { 0 }  
 20797 `\int_const:Nn \c_regex_catcode_mode_int` { 2 }  
 20798 `\int_const:Nn \c_regex_class_mode_int` { 3 }  
 20799 `\int_const:Nn \c_regex_catcode_in_class_mode_int` { 6 }

*(End definition for `\l_regex_mode_int` and others.)*

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]`.

20800 `\int_new:N \l_regex_catcodes_int`  
 20801 `\int_new:N \l_regex_default_catcodes_int`  
 20802 `\bool_new:N \l_regex_catcodes_bool`

*(End definition for `\l_regex_catcodes_int`, `\l_regex_default_catcodes_int`, and `\l_regex_catcodes_bool`.)*

Constants:  $4^c$  for each category, and the sum of all powers of 4.

20803 `\int_const:Nn \c_regex_catcode_C_int` { "1" }  
 20804 `\int_const:Nn \c_regex_catcode_B_int` { "4" }  
 20805 `\int_const:Nn \c_regex_catcode_E_int` { "10" }  
 20806 `\int_const:Nn \c_regex_catcode_M_int` { "40" }  
 20807 `\int_const:Nn \c_regex_catcode_T_int` { "100" }  
 20808 `\int_const:Nn \c_regex_catcode_P_int` { "1000" }  
 20809 `\int_const:Nn \c_regex_catcode_U_int` { "4000" }  
 20810 `\int_const:Nn \c_regex_catcode_D_int` { "10000" }  
 20811 `\int_const:Nn \c_regex_catcode_S_int` { "100000" }

```

20812 \int_const:Nn \c__regex_catcode_L_int { "400000 }
20813 \int_const:Nn \c__regex_catcode_O_int { "1000000 }
20814 \int_const:Nn \c__regex_catcode_A_int { "4000000 }
20815 \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.

```

20816 \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.

```

20817 \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.

```

20818 \int_new:N \l__regex_show_lines_int

(End definition for \l__regex_show_lines_int.)

```

### 38.3.2 Generic helpers used when compiling

\\_\\_regex\\_two\\_if\\_eq:NNNNTF 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.

```

20819 \prg_new_conditional:Npnn \_\_regex_two_if_eq:NNNN #1#2#3#4 { TF }
20820 {
20821   \if_meaning:w #1 #3
20822     \if:w #2 #4
20823       \prg_return_true:
20824     \else:
20825       \prg_return_false:
20826     \fi:
20827   \else:
20828     \prg_return_false:
20829   \fi:
20830 }

(End definition for \_\_regex_two_if_eq:NNNNTF.)

```

\\_\\_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.

```

20831 \cs_new_protected:Npn \_\_regex_get_digits:NTFw #1#2#3#4#5
20832 {
20833   \_\_regex_if_raw_digit:NNTF #4 #5
20834   { #1 = #5 \_\_regex_get_digits_loop:nw {#2} }
20835   { #3 #4 #5 }
20836 }


```

```

20837 \cs_new:Npn \__regex_get_digits_loop:nw #1#2#3
20838 {
20839     \__regex_if_raw_digit:NNTF #2 #3
20840     { #3 \__regex_get_digits_loop:nw {#1} }
20841     { \scan_stop: #1 #2 #3 }
20842 }
```

(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.

```

20843 \prg_new_conditional:Npnn \__regex_if_raw_digit:NN #1#2 { TF }
20844 {
20845     \if_meaning:w \__regex_compile_raw:N #1
20846     \if_int_compare:w 1 < 1 #2 \exp_stop_f:
20847         \prg_return_true:
20848     \else:
20849         \prg_return_false:
20850     \fi:
20851     \else:
20852         \prg_return_false:
20853     \fi:
20854 }
```

(End definition for `\__regex_if_raw_digit:NNTF`.)

### 38.3.3 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 `[\c{...}]` control sequence in a class,
- 2 `\c{...}` control sequence,
- 0 ... outer,
- 2 `\c...` catcode test,
- 6 `[\c...]` catcode test in a class,
- 63 `[\c{[...]}]` class inside mode -6,
- 23 `\c{[...]}` class inside mode -2,
- 3 `[...]` class inside mode 0,
- 23 `\c[...]` class inside mode 2,
- 63 `[\c[...]]` class inside mode 6.

This list is exhaustive, because `\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 `\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.

```

20855 \cs_new:Npn \_\_regex_if_in_class:TF
20856 {
20857     \if_int_odd:w \l__regex_mode_int
20858         \exp_after:wN \use_i:nn
20859     \else:
20860         \exp_after:wN \use_ii:nn
20861     \fi:
20862 }
```

*(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).

```

20863 \cs_new:Npn \_\_regex_if_in_cs:TF
20864 {
20865     \if_int_odd:w \l__regex_mode_int
20866         \exp_after:wN \use_ii:nn
20867     \else:
20868         \if_int_compare:w \l__regex_mode_int < \c__regex_outer_mode_int
20869             \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
20870         \else:
20871             \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
20872         \fi:
20873     \fi:
20874 }
```

*(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.

```

20875 \cs_new:Npn \_\_regex_if_in_class_or_catcode:TF
20876 {
20877     \if_int_odd:w \l__regex_mode_int
```

```

20878     \exp_after:wN \use_i:nn
20879 \else:
20880     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
20881         \exp_after:wN \exp_after:wN \exp_after:wN \use_i:nn
20882     \else:
20883         \exp_after:wN \exp_after:wN \exp_after:wN \use_ii:nn
20884     \fi:
20885 \fi:
20886 }

```

(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.

```

20887 \cs_new:Npn \_regex_if_within_catcode:TF
20888 {
20889     \if_int_compare:w \l__regex_mode_int > \c__regex_outer_mode_int
20890         \exp_after:wN \use_i:nn
20891     \else:
20892         \exp_after:wN \use_ii:nn
20893     \fi:
20894 }

```

(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.

```

20895 \cs_new_protected:Npn \_regex_chk_c_allowed:T
20896 {
20897     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
20898         \exp_after:wN \use:n
20899     \else:
20900         \if_int_compare:w \l__regex_mode_int = \c__regex_class_mode_int
20901             \exp_after:wN \exp_after:wN \exp_after:wN \use:n
20902         \else:
20903             \_kernel_msg_error:nn { kernel } { c-bad-mode }
20904             \exp_after:wN \exp_after:wN \exp_after:wN \use_none:n
20905         \fi:
20906     \fi:
20907 }

```

(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.

```

20908 \cs_new_protected:Npn \_regex_mode_quit_c:
20909 {
20910     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
20911         \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
20912     \else:
20913         \if_int_compare:w \l__regex_mode_int =
20914             \c__regex_catcode_in_class_mode_int
20915             \int_set_eq:NN \l__regex_mode_int \c__regex_class_mode_int
20916         \fi:
20917     \fi:
20918 }

```

(End definition for `\_\_regex\_mode\_quit\_c::`)

### 38.3.4 Framework

`\_\_regex_compile:w`  
`\_\_regex_compile_end:`

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`.

```
20919 \cs_new_protected:Npn \_\_regex_compile:w
20920 {
20921     \group_begin:
20922         \tl_build_begin:N \l_\_regex_build_tl
20923         \int_zero:N \l_\_regex_group_level_int
20924         \int_set_eq:NN \l_\_regex_default_catcodes_int
20925             \c_\_regex_all_catcodes_int
20926             \int_set_eq:NN \l_\_regex_catcodes_int \l_\_regex_default_catcodes_int
20927             \cs_set:Npn \_\_regex_item_equal:n { \_\_regex_item_caseful_equal:n }
20928             \cs_set:Npn \_\_regex_item_range:nn { \_\_regex_item_caseful_range:nn }
20929             \tl_build_put_right:Nn \l_\_regex_build_tl
20930                 { \_\_regex_branch:n { \if_false: } \fi: }
20931         }
20932     \cs_new_protected:Npn \_\_regex_compile_end:
20933     {
20934         \_\_regex_if_in_class:TF
20935         {
20936             \_\_kernel_msg_error:nn { kernel } { missing-rbrack }
20937             \use:c { \_\_regex_compile_l: }
20938             \prg_do_nothing: \prg_do_nothing:
20939         }
20940         {
20941             \if_int_compare:w \l_\_regex_group_level_int > 0 \exp_stop_f:
20942                 \_\_kernel_msg_error:nnx { kernel } { missing-rparen }
20943                 { \int_use:N \l_\_regex_group_level_int }
20944                 \prg_replicate:nn
20945                     { \l_\_regex_group_level_int }
20946                 {
20947                     \tl_build_put_right:Nn \l_\_regex_build_tl
20948                     {
20949                         \if_false: { \fi: }
20950                         \if_false: { \fi: } { 1 } { 0 } \c_true_bool
20951                     }
20952                     \tl_build_end:N \l_\_regex_build_tl
20953                     \exp_args:NNNo
20954                     \group_end:
20955                     \tl_build_put_right:Nn \l_\_regex_build_tl
20956                         { \l_\_regex_build_tl }
20957                 }
20958             \fi:
20959             \tl_build_put_right:Nn \l_\_regex_build_tl { \if_false: { \fi: } }
20960             \tl_build_end:N \l_\_regex_build_tl
20961             \exp_args:NNNx
20962         \group_end:
```

```

20963     \tl_set:Nn \l__regex_internal_regex { \l__regex_build_tl }
20964 }

```

(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 alphanumeric, 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.

```

20965 \cs_new_protected:Npn \_regex_compile:n #1
20966 {
20967     \_regex_compile:w
20968     \_regex_standard_escapechar:
20969     \int_set_eq:NN \l__regex_mode_int \c__regex_outer_mode_int
20970     \_regex_escape_use:nnnn
20971     {
20972         \_regex_char_if_special:NTF ##1
20973         \_regex_compile_special:N \_regex_compile_raw:N ##1
20974     }
20975     {
20976         \_regex_char_if_alphanumeric:NTF ##1
20977         \_regex_compile_escaped:N \_regex_compile_raw:N ##1
20978     }
20979     { \_regex_compile_raw:N ##1 }
20980     { #1 }
20981 \prg_do_nothing: \prg_do_nothing:
20982 \prg_do_nothing: \prg_do_nothing:
20983 \int_compare:nNnT \l__regex_mode_int = \c__regex_catcode_mode_int
20984     { \_kernel_msg_error:nn { kernel } { c-trailing } }
20985 \int_compare:nNnT \l__regex_mode_int < \c__regex_outer_mode_int
20986     {
20987         \_kernel_msg_error:nn { kernel } { c-missing-rbrace }
20988         \_regex_compile_end_cs:
20989         \prg_do_nothing: \prg_do_nothing:
20990         \prg_do_nothing: \prg_do_nothing:
20991     }
20992     \_regex_compile_end:
20993 }

```

(End definition for `\_regex_compile:n`.)

`\_regex_compile_escaped: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.

```

20994 \cs_new_protected:Npn \_regex_compile_special:N #1
20995 {
20996     \cs_if_exist_use:cF { __regex_compile_#1: }
20997     { \_regex_compile_raw:N #1 }
20998 }
20999 \cs_new_protected:Npn \_regex_compile_escaped:N #1

```

```

21000  {
21001    \cs_if_exist_use:cF { __regex_compile_/#1: }
21002    { __regex_compile_raw:N #1 }
21003  }

(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.

```

21004 \cs_new_protected:Npn __regex_compile_one:n #1
21005  {
21006    __regex_mode_quit_c:
21007    __regex_if_in_class:TF {
21008      {
21009        \tl_build_put_right:Nn \l__regex_build_tl
21010        { __regex_class:NnnnN \c_true_bool { \if_false: } \fi: }
21011      }
21012      \tl_build_put_right:Nx \l__regex_build_tl
21013      {
21014        \if_int_compare:w \l__regex_catcodes_int <
21015          \c__regex_all_catcodes_int
21016        __regex_item_catcode:nT { \int_use:N \l__regex_catcodes_int }
21017        { \exp_not:N \exp_not:n {#1} }
21018      \else:
21019        \exp_not:N \exp_not:n {#1}
21020        \fi:
21021      }
21022      \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
21023      __regex_if_in_class:TF { } { __regex_compile_quantifier:w }
21024    }
```

(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.

```

21025 \cs_new_protected:Npn __regex_compile_abort_tokens:n #1
21026  {
21027    \use:x
21028    {
21029      \exp_args:No \tl_map_function:nN { \tl_to_str:n {#1} }
21030      __regex_compile_raw:N
21031    }
21032  }
21033 \cs_generate_variant:Nn __regex_compile_abort_tokens:n { x }
```

(End definition for \_\_regex\_compile\_abort\_tokens:n.)

### 38.3.5 Quantifiers

\\_\_regex\_compile\_quantifier:w This looks ahead and finds any quantifier (special character equal to either of ?+\*{}).

```

21034 \cs_new_protected:Npn __regex_compile_quantifier:w #1#2
21035  {
```

```

21036     \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
21037     {
21038         \cs_if_exist_use:cF { __regex_compile_quantifier_#2:w }
21039             { \__regex_compile_quantifier_none: #1 #2 }
21040     }
21041     { \__regex_compile_quantifier_none: #1 #2 }
21042 }
```

(End definition for `\__regex_compile_quantifier:w`.)

`\__regex_compile_quantifier_none:` 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).

```

21043 \cs_new_protected:Npn \__regex_compile_quantifier_none:
21044     {
21045         \tl_build_put_right:Nn \l__regex_build_tl
21046             { \if_false: { \fi: } { 1 } { 0 } \c_false_bool }
21047     }
21048 \cs_new_protected:Npn \__regex_compile_quantifier_abort:xNN #1#2#3
21049     {
21050         \__regex_compile_quantifier_none:
21051         \kernel_msg_warning:nxxx { kernel } { invalid-quantifier } {#1} {#3}
21052         \__regex_compile_abort_tokens:x {#1}
21053         #2 #3
21054 }
```

(End definition for `\__regex_compile_quantifier_none:` and `\__regex_compile_quantifier_abort:xNN`.)

`\__regex_compile_quantifier_lazyness:nnNN` 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.

```

21055 \cs_new_protected:Npn \__regex_compile_quantifier_lazyness:nnNN #1#2#3#4
21056     {
21057         \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ?
21058             {
21059                 \tl_build_put_right:Nn \l__regex_build_tl
21060                     { \if_false: { \fi: } { #1 } { #2 } \c_true_bool }
21061             }
21062             {
21063                 \tl_build_put_right:Nn \l__regex_build_tl
21064                     { \if_false: { \fi: } { #1 } { #2 } \c_false_bool }
21065                     #3 #4
21066             }
21067 }
```

(End definition for `\__regex_compile_quantifier_lazyness:nnNN`.)

`\__regex_compile_quantifier_?:w` 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.

```

21068 \cs_new_protected:cpn { __regex_compile_quantifier_?:w }
21069     { \__regex_compile_quantifier_lazyness:nnNN { 0 } { 1 } }
21070 \cs_new_protected:cpn { __regex_compile_quantifier_*:w }
21071     { \__regex_compile_quantifier_lazyness:nnNN { 0 } { -1 } }
```

```

21072 \cs_new_protected:cpn { __regex_compile_quantifier_+:w }
21073   { \__regex_compile_quantifier_lazyness:nnNN { 1 } { -1 } }

(End definition for \__regex_compile_quantifier_?:w, \__regex_compile_quantifier_*:w, and \__-
\__regex_compile_quantifier_+: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  $\{a\}\{b - a\}$ .

```

21074 \cs_new_protected:cpn { __regex_compile_quantifier_ \c_left_brace_str :w }
21075   {
21076     \__regex_get_digits:NTFw \l__regex_internal_a_int
21077     { \__regex_compile_quantifier_braced_auxi:w }
21078     { \__regex_compile_quantifier_abort:xNN { \c_left_brace_str } }
21079   }
21080 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxi:w #1#2
21081   {
21082     \str_case_e:nnF { #1 #2 }
21083     {
21084       { \__regex_compile_special:N \c_right_brace_str }
21085       {
21086         \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
21087           { \int_use:N \l__regex_internal_a_int } { 0 }
21088       }
21089       { \__regex_compile_special:N , }
21090       {
21091         \__regex_get_digits:NTFw \l__regex_internal_b_int
21092           { \__regex_compile_quantifier_braced_auxiii:w }
21093           { \__regex_compile_quantifier_braced_auxii:w }
21094       }
21095     }
21096   {
21097     \__regex_compile_quantifier_abort:xNN
21098       { \c_left_brace_str \int_use:N \l__regex_internal_a_int }
21099       #1 #2
21100     }
21101   }
21102 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxii:w #1#2
21103   {
21104     \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N \c_right_brace_str
21105     {
21106       \exp_args:No \__regex_compile_quantifier_lazyness:nnNN
21107         { \int_use:N \l__regex_internal_a_int } { -1 }
21108     }
21109   {
21110     \__regex_compile_quantifier_abort:xNN
21111       { \c_left_brace_str \int_use:N \l__regex_internal_a_int , }
21112       #1 #2
21113     }
21114   }
21115 \cs_new_protected:Npn \__regex_compile_quantifier_braced_auxiii:w #1#2

```

```

21116  {
21117    \_\_regex\_two\_if\_eq:NNNNTF #1 #2 \_\_regex\_compile\_special:N \c_right\_brace\_str
21118    {
21119      \if\_int\_compare:w \l\_regex\_internal\_a\_int >
21120        \l\_regex\_internal\_b\_int
21121        \_\_kernel\_msg\_error:nnxx { kernel } { backwards-quantifier }
21122          { \int\_use:N \l\_regex\_internal\_a\_int }
21123          { \int\_use:N \l\_regex\_internal\_b\_int }
21124          \int\_zero:N \l\_regex\_internal\_b\_int
21125    \else:
21126      \int\_sub:Nn \l\_regex\_internal\_b\_int \l\_regex\_internal\_a\_int
21127    \fi:
21128    \exp\_args:Noo \_\_regex\_compile\_quantifier\_lazyness:nnNN
21129      { \int\_use:N \l\_regex\_internal\_a\_int }
21130      { \int\_use:N \l\_regex\_internal\_b\_int }
21131  }
21132  {
21133    \_\_regex\_compile\_quantifier\_abort:xNN
21134    {
21135      \c_left\_brace\_str
21136      \int\_use:N \l\_regex\_internal\_a\_int ,
21137      \int\_use:N \l\_regex\_internal\_b\_int
21138    }
21139    #1 #2
21140  }
21141 }

```

(End definition for `\_\_regex\_compile\_quantifier\_{:w}` and others.)

### 38.3.6 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.

```

21142 \cs_new_protected:Npn \_\_regex\_compile\_raw\_error:N #
21143  {
21144    \_\_kernel\_msg\_error:nnx { kernel } { bad-escape } {#1}
21145    \_\_regex\_compile\_raw:N #1
21146  }

```

(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.

```

21147 \cs_new_protected:Npn \_\_regex\_compile\_raw:N #1#2#3
21148  {
21149    \_\_regex\_if\_in\_class:TF
21150    {
21151      \_\_regex\_two\_if\_eq:NNNNTF #2 #3 \_\_regex\_compile\_special:N -
21152        { \_\_regex\_compile\_range:Nw #1 }
21153    {
21154      \_\_regex\_compile\_one:n
21155        { \_\_regex\_item\_equal:n { \int\_value:w '#1 } }
21156      #2 #3

```

```

21157         }
21158     }
21159     {
21160         \_\_regex\_compile\_one:n
21161             { \_\_regex\_item\_equal:n { \int\_value:w '#1 } }
21162             #2 #3
21163     }
21164 }
```

(End definition for `\_\_regex\_compile\_raw:N.`)

`\_\_regex\_compile\_range:Nw`  
`\_\_regex\_if\_end\_range:NNTF`

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.

```

21165 \prg_new_protected_conditional:Npnn \_\_regex_if_end_range:NN #1#2 { TF }
21166   {
21167       \if_meaning:w \_\_regex_compile_raw:N #1
21168           \prg_return_true:
21169       \else:
21170           \if_meaning:w \_\_regex_compile_special:N #1
21171               \if_charcode:w ] #2
21172                   \prg_return_false:
21173               \else:
21174                   \prg_return_true:
21175               \fi:
21176           \else:
21177               \prg_return_false:
21178           \fi:
21179       \fi:
21180   }
21181 \cs_new_protected:Npn \_\_regex_compile_range:Nw #1#2#3
21182   {
21183       \_\_regex_if_end_range:NNTF #2 #3
21184       {
21185           \if_int_compare:w '#1 > '#3 \exp_stop_f:
21186               \_\_kernel_msg_error:nnxx { kernel } { range-backwards } {#1} {#3}
21187           \else:
21188               \tl_build_put_right:Nx \l_\_regex_build_tl
21189               {
21190                   \if_int_compare:w '#1 = '#3 \exp_stop_f:
21191                       \_\_regex_item_equal:n
21192                   \else:
21193                       \_\_regex_item_range:nn { \int_value:w '#1 }
21194                   \fi:
21195                   { \int_value:w '#3 }
21196               }
21197           \fi:
21198       }
21199       {
21200           \_\_kernel_msg_warning:nnxx { kernel } { range-missing-end }
21201               {#1} { \c_backslash_str #3 }
21202           \tl_build_put_right:Nx \l_\_regex_build_tl
21203               {
21204                   \_\_regex_item_equal:n { \int_value:w '#1 \exp_stop_f: }
```

```

21205         \_\_regex\_item\_equal:n { \int\_value:w ‘- \exp\_stop\_f: }
21206     }
21207 #2#3
21208 }
21209 }
```

(End definition for `\_\_regex_compile_range:Nw` and `\_\_regex_if_end_range:NNTF`.)

### 38.3.7 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).

```

21210 \cs_new_protected:cpx { \_\_regex_compile_..: }
21211 {
21212   \exp_not:N \_\_regex_if_in_class:TF
21213   { \_\_regex_compile_raw:N . }
21214   { \_\_regex_compile_one:n \exp_not:c { \_\_regex_prop_..: } }
21215 }
21216 \cs_new_protected:cpn { \_\_regex_prop_..: }
21217 {
21218   \if_int_compare:w \l_\_regex_curr_char_int > - 2 \exp_stop_f:
21219   \exp_after:wN \_\_regex_break_true:w
21220   \fi:
21221 }
```

(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.

```

21222 \cs_set_protected:Npn \_\_regex_tmp:w #1#2
21223 {
21224   \cs_new_protected:cpx { \_\_regex_compile_/#1: }
21225   { \_\_regex_compile_one:n \exp_not:c { \_\_regex_prop_#1: } }
21226   \cs_new_protected:cpx { \_\_regex_compile_/#2: }
21227   {
21228     \_\_regex_compile_one:n
21229     { \_\_regex_item_reverse:n \exp_not:c { \_\_regex_prop_#1: } }
21230   }
21231 }
21232 \_\_regex_tmp:w d D
21233 \_\_regex_tmp:w h H
21234 \_\_regex_tmp:w s S
21235 \_\_regex_tmp:w v V
21236 \_\_regex_tmp:w w W
21237 \cs_new_protected:cpn { \_\_regex_compile_/_N: }
21238 { \_\_regex_compile_one:n \_\_regex_prop_N: }
```

(End definition for `\_\_regex_compile/_d:` and others.)

### 38.3.8 Anchoring and simple assertions

\\_\\_regex\\_compile\\_anchor:NF

\\_\\_regex\\_compile\\_ $\wedge$ :

\\_\\_regex\\_compile\\_A:

\\_\\_regex\\_compile\\_G:

\\_\\_regex\\_compile\\_ $\$$ :

\\_\\_regex\\_compile\\_Z:

\\_\\_regex\\_compile\\_z:

In modes where assertions are allowed, anchor to the start of the query, the start of the match, or the end of the query, depending on the integer #1. In other modes, #2 treats the character as raw, with an error for escaped letters (\$ is valid in a class, but \A is definitely a mistake on the user's part).

```

21239 \cs_new_protected:Npn \_\_regex_compile_anchor:NF #1#2
21240   {
21241     \_\_regex_if_in_class_or_catcode:TF {#2}
21242     {
21243       \tl_build_put_right:Nn \l_\_regex_build_tl
21244         { \_\_regex_assertion:Nn \c_true_bool { \_\_regex_anchor:N #1 } }
21245     }
21246   }
21247 \cs_set_protected:Npn \_\_regex_tmp:w #1#2
21248   {
21249     \cs_new_protected:cpn { \_\_regex_compile_/#1: }
21250       { \_\_regex_compile_anchor:NF #2 { \_\_regex_compile_raw_error:N #1 } }
21251   }
21252 \_\_regex_tmp:w A \l_\_regex_min_pos_int
21253 \_\_regex_tmp:w G \l_\_regex_start_pos_int
21254 \_\_regex_tmp:w Z \l_\_regex_max_pos_int
21255 \_\_regex_tmp:w z \l_\_regex_max_pos_int
21256 \cs_set_protected:Npn \_\_regex_tmp:w #1#2
21257   {
21258     \cs_new_protected:cpn { \_\_regex_compile_#/1: }
21259       { \_\_regex_compile_anchor:NF #2 { \_\_regex_compile_raw:N #1 } }
21260   }
21261 \exp_args:Nx \_\_regex_tmp:w { \iow_char:N \^ } \l_\_regex_min_pos_int
21262 \exp_args:Nx \_\_regex_tmp:w { \iow_char:N \$ } \l_\_regex_max_pos_int

```

(End definition for \\_\\_regex\_compile\_anchor:NF and others.)

\\_\\_regex\_compile\_/\_b:

\\_\\_regex\_compile\_/\_B:

Contrarily to  $\wedge$  and  $\$$ , which could be implemented without really knowing what precedes in the token list, this requires more information, namely, the knowledge of the last character code.

```

21263 \cs_new_protected:cpn { \_\_regex_compile_/_b: }
21264   {
21265     \_\_regex_if_in_class_or_catcode:TF
21266       { \_\_regex_compile_raw_error:N b }
21267       {
21268         \tl_build_put_right:Nn \l_\_regex_build_tl
21269           { \_\_regex_assertion:Nn \c_true_bool { \_\_regex_b_test: } }
21270       }
21271   }
21272 \cs_new_protected:cpn { \_\_regex_compile_/_B: }
21273   {
21274     \_\_regex_if_in_class_or_catcode:TF
21275       { \_\_regex_compile_raw_error:N B }
21276       {
21277         \tl_build_put_right:Nn \l_\_regex_build_tl
21278           { \_\_regex_assertion:Nn \c_false_bool { \_\_regex_b_test: } }
21279       }
21280   }

```

(End definition for `\_\_regex_compile_/_b:` and `\_\_regex_compile_/_B::`)

### 38.3.9 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.

```
21281 \cs_new_protected:cpn { \_\_regex_compile_[:]
21282   {
21283     \_\_regex_if_in_class:TF
21284     {
21285       \if_int_compare:w \l_\_regex_mode_int >
21286         \c_\_regex_catcode_in_class_mode_int
21287         \tl_build_put_right:Nn \l_\_regex_build_tl { \if_false: { \fi: } }
21288       \fi:
21289       \tex_advance:D \l_\_regex_mode_int - 15 \exp_stop_f:
21290       \tex_divide:D \l_\_regex_mode_int 13 \exp_stop_f:
21291       \if_int_odd:w \l_\_regex_mode_int \else:
21292         \exp_after:wN \_\_regex_compile_quantifier:w
21293         \fi:
21294       }
21295     { \_\_regex_compile_raw:N }
21296   }
```

(End definition for `\_\_regex_compile_[:..]`)

- `\_\_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.

```
21297 \cs_new_protected:cpn { \_\_regex_compile_[::]
21298   {
21299     \_\_regex_if_in_class:TF
21300     { \_\_regex_compile_class_posix_test:w }
21301     {
21302       \_\_regex_if_within_catcode:TF
21303       {
21304         \exp_after:wN \_\_regex_compile_class_catcode:w
21305         \int_use:N \l_\_regex_catcodes_int ;
21306       }
21307     { \_\_regex_compile_class_normal:w }
21308   }
21309 }
```

(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.

```
21310 \cs_new_protected:Npn \_\_regex_compile_class_normal:w
21311   {
21312     \_\_regex_compile_class:TFNN
```

```

21313 { \__regex_class:NnnN \c_true_bool }
21314 { \__regex_class:NnnN \c_false_bool }
21315 }
```

(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.

```

21316 \cs_new_protected:Npn \__regex_compile_class_catcode:w #1;
21317 {
21318     \if_int_compare:w \l__regex_mode_int = \c__regex_catcode_mode_int
21319         \tl_build_put_right:Nn \l__regex_build_tl
21320             { \__regex_class:NnnN \c_true_bool { \if_false: } \fi: }
21321         \fi:
21322         \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
21323         \__regex_compile_class:TFNN
21324             { \__regex_item_catcode:nT {#1} }
21325             { \__regex_item_catcode_reverse:nT {#1} }
21326 }
```

(End definition for `\__regex_compile_class_catcode:w.`)

`\__regex_compile_class:TFNN`

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.

```

21327 \cs_new_protected:Npn \__regex_compile_class:TFNN #1#2#3#4
21328 {
21329     \l__regex_mode_int = \int_value:w \l__regex_mode_int 3 \exp_stop_f:
21330     \__regex_two_if_eq:NNNTF #3 #4 \__regex_compile_special:N ^
21331     {
21332         \tl_build_put_right:Nn \l__regex_build_tl { #2 { \if_false: } \fi: }
21333         \__regex_compile_class:NN
21334     }
21335     {
21336         \tl_build_put_right:Nn \l__regex_build_tl { #1 { \if_false: } \fi: }
21337         \__regex_compile_class:NN #3 #4
21338     }
21339 }
```

`\cs_new_protected:Npn \__regex_compile_class>NN #1#2`

```

21340 {
21341     \token_if_eq_charcode:NNTF #2 ]
21342         { \__regex_compile_raw:N #2 }
21343         { #1 #2 }
21344 }
```

(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.

```

21346 \cs_new_protected:Npn \__regex_compile_class_posix_test:w #1#2
21347 {
21348     \token_if_eq_meaning:NNT \__regex_compile_special:N #1
21349     {
21350         \str_case:nn { #2 }
21351         {
21352             : { \__regex_compile_class_posix:NNNNw }
21353             = {
21354                 \__kernel_msg_warning:nnx { kernel }
21355                 { posix-unsupported } { = }
21356             }
21357             . {
21358                 \__kernel_msg_warning:nnx { kernel }
21359                 { posix-unsupported } { . }
21360             }
21361         }
21362     }
21363     \__regex_compile_raw:N [ #1 #2
21364 }
21365 \cs_new_protected:Npn \__regex_compile_class_posix:NNNNw #1#2#3#4#5#6
21366 {
21367     \__regex_two_if_eq>NNNNTF #5 #6 \__regex_compile_special:N ^
21368     {
21369         \bool_set_false:N \l__regex_internal_bool
21370         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
21371         \__regex_compile_class_posix_loop:w
21372     }
21373     {
21374         \bool_set_true:N \l__regex_internal_bool
21375         \tl_set:Nx \l__regex_internal_a_tl { \if_false: } \fi:
21376         \__regex_compile_class_posix_loop:w #5 #6
21377     }
21378 }
21379 \cs_new:Npn \__regex_compile_class_posix_loop:w #1#2
21380 {
21381     \token_if_eq_meaning:NNTF \__regex_compile_raw:N #1
21382     { #2 \__regex_compile_class_posix_loop:w }
21383     { \if_false: { \fi: } \__regex_compile_class_posix_end:w #1 #2 }
21384 }
21385 \cs_new_protected:Npn \__regex_compile_class_posix_end:w #1#2#3#4
21386 {
21387     \__regex_two_if_eq>NNNNTF #1 #2 \__regex_compile_special:N :
21388     { \__regex_two_if_eq>NNNNTF #3 #4 \__regex_compile_special:N ] }
21389     { \use_ii:nn }
21390     {
21391         \cs_if_exist:cTF { __regex_posix_ \l__regex_internal_a_tl : }
21392         {
21393             \__regex_compile_one:n
21394             {
21395                 \bool_if:NF \l__regex_internal_bool \__regex_item_reverse:n
21396                 \exp_not:c { __regex_posix_ \l__regex_internal_a_tl : }
21397             }
21398         }
21399     }

```

```

21400     \__kernel_msg_warning:nnx { kernel } { posix-unknown }
21401         { \l__regex_internal_a_tl }
21402     \__regex_compile_abort_tokens:x
21403         {
21404             [: \bool_if:NF \l__regex_internal_bool { ^ }
21405                 \l__regex_internal_a_tl :]
21406         }
21407     }
21408 }
21409 {
21410     \__kernel_msg_error:nnxx { kernel } { posix-missing-close }
21411         { [: \l__regex_internal_a_tl ] { #2 #4 }
21412             \__regex_compile_abort_tokens:x { [: \l__regex_internal_a_tl ]
21413                 #1 #2 #3 #4
21414             }
21415         }

```

(End definition for `\__regex_compile_class_posix_test:w` and others.)

### 38.3.10 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.

```

21416 \cs_new_protected:Npn \__regex_compile_group_begin:N #1
21417 {
21418     \tl_build_put_right:Nn \l__regex_build_t1 { #1 { \if_false: } \fi: }
21419     \__regex_mode_quit_c:
21420     \group_begin:
21421         \tl_build_begin:N \l__regex_build_t1
21422             \int_set_eq:NN \l__regex_default_catcodes_int \l__regex_catcodes_int
21423             \int_incr:N \l__regex_group_level_int
21424             \tl_build_put_right:Nn \l__regex_build_t1
21425                 { \__regex_branch:n { \if_false: } \fi: }
21426     }
21427 \cs_new_protected:Npn \__regex_compile_group_end:
21428 {
21429     \if_int_compare:w \l__regex_group_level_int > 0 \exp_stop_f:
21430         \tl_build_put_right:Nn \l__regex_build_t1 { \if_false: { \fi: } }
21431         \tl_build_end:N \l__regex_build_t1
21432         \exp_args:NNNx
21433     \group_end:
21434         \tl_build_put_right:Nn \l__regex_build_t1 { \l__regex_build_t1 }
21435         \int_set_eq:NN \l__regex_catcodes_int \l__regex_default_catcodes_int
21436         \exp_after:wN \__regex_compile_quantifier:w
21437 \else:
21438     \__kernel_msg_warning:nn { kernel } { extra-rparen }

```

```

21439      \exp_after:wN \__regex_compile_raw:N \exp_after:wN )
21440      \fi:
21441  }

```

(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.

```

21442 \cs_new_protected:cpn { __regex_compile_(: }
21443 {
21444   \__regex_if_in_class:TF { \__regex_compile_raw:N ( }
21445   {
21446     \if_int_compare:w \l__regex_mode_int =
21447       \c__regex_catcode_in_class_mode_int
21448     \__kernel_msg_error:nn { kernel } { c-lparen-in-class }
21449     \exp_after:wN \__regex_compile_raw:N \exp_after:wN (
21450     \else:
21451       \exp_after:wN \__regex_compile_lparen:w
21452     \fi:
21453   }
21454 }
21455 \cs_new_protected:Npn \__regex_compile_lparen:w #1#2#3#4
21456 {
21457   \__regex_two_if_eq:NNNNTF #1 #2 \__regex_compile_special:N ?
21458   {
21459     \cs_if_exist_use:cF
21460     { __regex_compile_special_group_\token_to_str:N #4 :w }
21461     {
21462       \__kernel_msg_warning:nnx { kernel } { special-group-unknown }
21463       { (? #4 }
21464       \__regex_compile_group_begin:N \__regex_group:nnnN
21465         \__regex_compile_raw:N ? #3 #4
21466     }
21467   }
21468   {
21469     \__regex_compile_group_begin:N \__regex_group:nnnN
21470       #1 #2 #3 #4
21471   }
21472 }

```

(End definition for `\__regex_compile_(:.)`)

`\__regex_compile_|:`: In a class, the pipe is not special. Otherwise, end the current branch and open another one.

```

21473 \cs_new_protected:cpn { __regex_compile_|: }
21474 {
21475   \__regex_if_in_class:TF { \__regex_compile_raw:N | }
21476   {
21477     \tl_build_put_right:Nn \l__regex_build_tl
21478       { \if_false: { \fi: } \__regex_branch:n { \if_false: } \fi: }
21479   }
21480 }

```

(End definition for `\__regex_compile_|:.`)

\\_\\_regex\\_compile\\_): Within a class, parentheses are not special. Outside, close a group.

```
21481 \cs_new_protected:cpn { __regex_compile_): }
21482   {
21483     __regex_if_in_class:TF { __regex_compile_raw:N }
21484     { __regex_compile_group_end: }
21485   }
```

(End definition for \\_\\_regex\\_compile\\_):)

\\_\\_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.

```
21486 \cs_new_protected:cpn { __regex_compile_special_group_::w }
21487   { __regex_compile_group_begin:N __regex_group_no_capture:nnnN }
21488 \cs_new_protected:cpn { __regex_compile_special_group_|:w }
21489   { __regex_compile_group_begin:N __regex_group_resetting:nnnN }
```

(End definition for \\_\\_regex\\_compile\\_special\\_group\\_::w and \\_\\_regex\\_compile\\_special\\_group\\_|:w.)

\\_\\_regex\\_compile\\_special\\_group\\_i: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.

```
21490 \cs_new_protected:Npn __regex_compile_special_group_i:w #1#2
21491   {
21492     __regex_two_if_eq>NNNNTF #1 #2 __regex_compile_special:N )
21493     {
21494       \cs_set:Npn __regex_item_equal:n
21495         { __regex_item_caseless_equal:n }
21496       \cs_set:Npn __regex_item_range:nn
21497         { __regex_item_caseless_range:nn }
21498     }
21499     {
21500       __kernel_msg_warning:nnx { kernel } { unknown-option } { (?i #2 }
21501       __regex_compile_raw:N (
21502       __regex_compile_raw:N ?
21503       __regex_compile_raw:N i
21504       #1 #2
21505     }
21506   }
21507 \cs_new_protected:cpn { __regex_compile_special_group_-:w } #1#2#3#4
21508   {
21509     __regex_two_if_eq>NNNNTF #1 #2 __regex_compile_raw:N i
21510     { __regex_two_if_eq>NNNNTF #3 #4 __regex_compile_special:N ) }
21511     { \use_ii:nn }
21512     {
21513       \cs_set:Npn __regex_item_equal:n
21514         { __regex_item_caseful_equal:n }
21515       \cs_set:Npn __regex_item_range:nn
21516         { __regex_item_caseful_range:nn }
21517     }
21518   {
21519     __kernel_msg_warning:nnx { kernel } { unknown-option } { (?-#2#4 }
21520     __regex_compile_raw:N (
21521     __regex_compile_raw:N ?
21522     __regex_compile_raw:N -
21523     #1 #2 #3 #4
```

```

21524     }
21525 }
```

(End definition for `\_regex_compile_special_group_i:w` and `\_regex_compile_special_group_-:w`.)

### 38.3.11 Catcodes and csnames

`\_regex_compile/_c:` 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.

```

21526 \cs_new_protected:cpn { \_regex_compile/_c: }
21527   { \_regex_chk_c_allowed:T { \_regex_compile_c_test:NN } }
21528 \cs_new_protected:Npn \_regex_compile_c_test:NN #1#2
21529   {
21530     \token_if_eq_meaning:NNTF #1 \_regex_compile_raw:N
21531     {
21532       \int_if_exist:cTF { c__regex_catcode_#2_int }
21533       {
21534         \int_set_eq:Nc \l__regex_catcodes_int
21535           { c__regex_catcode_#2_int }
21536         \l__regex_mode_int
21537           = \if_case:w \l__regex_mode_int
21538             \c__regex_catcode_mode_int
21539           \else:
21540             \c__regex_catcode_in_class_mode_int
21541           \fi:
21542             \token_if_eq_charcode:NNT C #2 { \_regex_compile_c_C:NN }
21543           }
21544     }
21545     { \cs_if_exist_use:cF { \_regex_compile_c_#2:w } }
21546     {
21547       \_kernel_msg_error:nnx { kernel } { c-missing-category } {#2}
21548       #1 #2
21549     }
21550 }
```

(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.

```

21551 \cs_new_protected:Npn \_regex_compile_c_C:NN #1#2
21552   {
21553     \token_if_eq_meaning:NNTF #1 \_regex_compile_special:N
21554     {
21555       \token_if_eq_charcode:NNTF #2 .
21556       { \use_none:n }
21557       { \token_if_eq_charcode>NNF #2 ( ) % }
21558     }
21559     { \use:n }
21560     { \_kernel_msg_error:nnn { kernel } { c-C-invalid } {#2} }
21561     #1 #2
21562 }
```

(End definition for `\_regex_compile_c_C:NN`.)

\\_regex\_compile\_c\_[:w  
\\_\\_regex\_compile\_c\_lbrack\_loop:NN  
\\_\\_regex\_compile\_c\_lbrack\_add:N  
\\_\\_regex\_compile\_c\_lbrack\_end:

When encountering \c[, the task is to collect uppercase letters representing character categories. First check for ^ which negates the list of category codes.

```

21563 \cs_new_protected:cpn { __regex_compile_c_[:w] } #1#2
21564 {
21565     \l__regex_mode_int
21566         = \if_case:w \l__regex_mode_int
21567             \c__regex_catcode_mode_int
21568             \else:
21569                 \c__regex_catcode_in_class_mode_int
21570             \fi:
21571     \int_zero:N \l__regex_catcodes_int
21572     \_\_regex_two_if_eq:NNNNTF #1 #2 \_\_regex_compile_special:N ^
21573     {
21574         \bool_set_false:N \l__regex_catcodes_bool
21575         \_\_regex_compile_c_lbrack_loop:NN
21576     }
21577     {
21578         \bool_set_true:N \l__regex_catcodes_bool
21579         \_\_regex_compile_c_lbrack_loop:NN
21580         #1 #2
21581     }
21582 }
21583 \cs_new_protected:Npn \_\_regex_compile_c_lbrack_loop:NN #1#2
21584 {
21585     \token_if_eq_meaning:NNTF #1 \_\_regex_compile_raw:N
21586     {
21587         \int_if_exist:cTF { c__regex_catcode_#2_int }
21588         {
21589             \exp_args:Nc \_\_regex_compile_c_lbrack_add:N
21590                 { c__regex_catcode_#2_int }
21591             \_\_regex_compile_c_lbrack_loop:NN
21592         }
21593     }
21594     {
21595         \token_if_eq_charcode:NNTF #2 ]
21596             { \_\_regex_compile_c_lbrack_end: }
21597     }
21598     {
21599         \_\_kernel_msg_error:nnx { kernel } { c-missing-rbrack } {#2}
21600         \_\_regex_compile_c_lbrack_end:
21601         #1 #2
21602     }
21603 }
21604 \cs_new_protected:Npn \_\_regex_compile_c_lbrack_add:N #1
21605 {
21606     \if_int_odd:w \int_eval:n { \l__regex_catcodes_int / #1 } \exp_stop_f:
21607     \else:
21608         \int_add:Nn \l__regex_catcodes_int {#1}
21609     \fi:
21610 }
21611 \cs_new_protected:Npn \_\_regex_compile_c_lbrack_end:
21612 {
21613     \if_meaning:w \c_false_bool \l__regex_catcodes_bool
21614         \int_set:Nn \l__regex_catcodes_int
```

```

21615     { \c__regex_all_catcodes_int - \l__regex_catcodes_int }
21616   \fi:
21617 }

```

(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{...}`.

```

21618 \cs_new_protected:cpn { __regex_compile_c_ \c_left_brace_str :w }
21619   {
21620     \_regex_compile:w
21621     \_regex_disable_submatches:
21622     \l__regex_mode_int
21623       = \if_case:w \l__regex_mode_int
21624         \c__regex_cs_mode_int
21625       \else:
21626         \c__regex_cs_in_class_mode_int
21627     \fi:
21628   }

```

(End definition for `\_regex_compile_c_{::}`)

`\_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::`.

```

21629 \flag_new:n { __regex_cs }
21630 \cs_new_protected:cpn { __regex_compile_ \c_right_brace_str : }
21631   {
21632     \_regex_if_in_cs:TF
21633     { \_regex_compile_end_cs: }
21634     { \exp_after:wN \_regex_compile_raw:N \c_right_brace_str }
21635   }
21636 \cs_new_protected:Npn \_regex_compile_end_cs:
21637   {
21638     \_regex_compile_end:
21639     \flag_clear:n { __regex_cs }
21640     \tl_set:Nx \l__regex_internal_a_tl
21641     {
21642       \exp_after:wN \_regex_compile_cs_aux:Nn \l__regex_internal_regex
21643       \q_nil \q_nil \q_recursion_stop
21644     }
21645     \exp_args:Nx \_regex_compile_one:n
21646     {
21647       \flag_if_raised:nTF { __regex_cs }
21648         { \_regex_item_cs:n { \exp_not:o \l__regex_internal_regex } }
21649         {
21650           \_regex_item_exact_cs:n
21651             { \tl_tail:N \l__regex_internal_a_tl }
21652         }

```

```

21653     }
21654   }
21655 \cs_new:Npn \__regex_compile_cs_aux:Nn #1#2
21656   {
21657     \cs_if_eq:NNTF #1 \__regex_branch:n
21658     {
21659       \scan_stop:
21660       \__regex_compile_cs_aux>NNnnnN #2
21661       \q_nil \q_nil \q_nil \q_nil \q_nil \q_recursion_stop
21662       \__regex_compile_cs_aux:Nn
21663     }
21664   {
21665     \quark_if_nil:NF #1 { \flag_raise_if_clear:n { __regex_cs } }
21666     \use_none_delimit_by_q_recursion_stop:w
21667   }
21668 }
21669 \cs_new:Npn \__regex_compile_cs_aux:NNnnnN #1#2#3#4#5#6
21670   {
21671     \bool_lazy_all:nTF
21672     {
21673       { \cs_if_eq_p:NN #1 \__regex_class:NnnnN }
21674       {#2}
21675       { \tl_if_head_eq_meaning_p:nN {#3} \__regex_item_caseful_equal:n }
21676       { \int_compare_p:nNn { \tl_count:n {#3} } = { 2 } }
21677       { \int_compare_p:nNn {#5} = { 0 } }
21678     }
21679   {
21680     \prg_replicate:nn {#4}
21681       { \char_generate:nn { \use_i:nn #3 } {12} }
21682     \__regex_compile_cs_aux:NNnnnN
21683   }
21684   {
21685     \quark_if_nil:NF #1
21686     {
21687       \flag_raise_if_clear:n { __regex_cs }
21688       \use_i_delimit_by_q_recursion_stop:nw
21689     }
21690     \use_none_delimit_by_q_recursion_stop:w
21691   }
21692 }

```

(End definition for `\__regex_compile_`: and others.)

### 38.3.12 Raw token lists with \u

`\__regex_compile/_u:` The `\u` escape is invalid in classes and directly following a catcode test. Otherwise, it must be followed by a left brace. We then 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.

```

21693 \cs_new_protected:cpn { __regex_compile/_u: } #1#2
21694   {

```

```

21695     \__regex_if_in_class_or_catcode:TF
21696     { \__regex_compile_raw_error:N u #1 #2 }
21697     {
21698         \__regex_two_if_eq:NNNTF #1 #2 \__regex_compile_special:N \c_left_brace_str
21699         {
21700             \tl_set:Nx \l__regex_internal_a_t1 { \if_false: } \fi:
21701             \__regex_compile_u_loop:NN
21702         }
21703         {
21704             \__kernel_msg_error:nn { kernel } { u-missing-lbrace }
21705             \__regex_compile_raw:N u #1 #2
21706         }
21707     }
21708 }
21709 \cs_new:Npn \__regex_compile_u_loop:NN #1#2
21710 {
21711     \token_if_eq_meaning:NNTF #1 \__regex_compile_raw:N
21712     { #2 \__regex_compile_u_loop:NN }
21713     {
21714         \token_if_eq_meaning:NNTF #1 \__regex_compile_special:N
21715         {
21716             \exp_after:wN \token_if_eqCharCode:NNTF \c_right_brace_str #2
21717             { \if_false: { \fi: } \__regex_compile_u_end: }
21718             { #2 \__regex_compile_u_loop:NN }
21719         }
21720     }
21721     {
21722         \if_false: { \fi: }
21723         \__kernel_msg_error:nnx { kernel } { u-missing-rbrace } { #2 }
21724         \__regex_compile_u_end:
21725         #1 #2
21726     }
21727 }

```

(End definition for `\__regex_compile_u:` and `\__regex_compile_u_loop:NN`.)

`\__regex_compile_u_end:` Once we have extracted the variable's name, we store the contents of that variable in `\l__regex_internal_a_t1`. 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.

```

21728 \cs_new_protected:Npn \__regex_compile_u_end:
21729 {
21730     \tl_set:Nv \l__regex_internal_a_t1 { \l__regex_internal_a_t1 }
21731     \if_int_compare:w \l__regex_mode_int = \c__regex_outer_mode_int
21732         \__regex_compile_u_not_cs:
21733     \else:
21734         \__regex_compile_u_in_cs:
21735     \fi:
21736 }

```

(End definition for `\__regex_compile_u_end:.`)

`\__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.

```

21737 \cs_new_protected:Npn \__regex_compile_u_in_cs:
21738 {
21739     \tl_gset:Nx \g__regex_internal_tl
21740     {
21741         \exp_args:No \__kernel_str_to_other_fast:n
21742         { \l__regex_internal_a_tl }
21743     }
21744     \tl_build_put_right:Nx \l__regex_build_tl
21745     {
21746         \tl_map_function:NN \g__regex_internal_tl
21747         \__regex_compile_u_in_cs_aux:n
21748     }
21749 }
21750 \cs_new:Npn \__regex_compile_u_in_cs_aux:n #1
21751 {
21752     \__regex_class:NnnnN \c_true_bool
21753     { \__regex_item_caseful_equal:n { \int_value:w '#1 } }
21754     { 1 } { 0 } \c_false_bool
21755 }

```

(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.

```

21756 \cs_new_protected:Npn \__regex_compile_u_not_cs:
21757 {
21758     \tl_analysis_map_inline:Nn \l__regex_internal_a_tl
21759     {
21760         \tl_build_put_right:Nx \l__regex_build_tl
21761         {
21762             \__regex_class:NnnnN \c_true_bool
21763             {
21764                 \if_int_compare:w ##3 = 0 \exp_stop_f:
21765                     \__regex_item_exact_cs:n
21766                     { \exp_after:wN \cs_to_str:N ##1 }
21767                 \else:
21768                     \__regex_item_exact:nn { \int_value:w ##3 } { ##2 }
21769                     \fi:
21770                 }
21771             { 1 } { 0 } \c_false_bool
21772         }
21773     }
21774 }

```

(End definition for `\__regex_compile_u_not_cs..`)

### 38.3.13 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.

```

21775 \cs_new_protected:cpn { __regex_compile_/_K: }
21776 {

```

```

21777     \int_compare:nNnTF \l__regex_mode_int = \c__regex_outer_mode_int
21778     { \tl_build_put_right:Nn \l__regex_build_tl { \__regex_command_K: } }
21779     { \__regex_compile_raw_error:N K }
2180 }

```

(End definition for `\__regex_compile_/_K:.`)

### 38.3.14 Showing regexes

`\__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.

```

21781 \cs_new_protected:Npn \__regex_show:N #1
21782 {
21783     \group_begin:
21784     \tl_build_begin:N \l__regex_build_tl
21785     \cs_set_protected:Npn \__regex_branch:n
21786     {
21787         \seq_pop_right:NN \l__regex_show_prefix_seq
21788         \l__regex_internal_a_tl
21789         \__regex_show_one:n { +-branch }
21790         \seq_put_right:No \l__regex_show_prefix_seq
21791         \l__regex_internal_a_tl
21792         \use:n
21793     }
21794     \cs_set_protected:Npn \__regex_group:nnnN
21795     { \__regex_show_group_aux:nnnnN { } }
21796     \cs_set_protected:Npn \__regex_group_no_capture:nnnN
21797     { \__regex_show_group_aux:nnnnN { ~(no-capture) } }
21798     \cs_set_protected:Npn \__regex_group_resetting:nnnN
21799     { \__regex_show_group_aux:nnnnN { ~(resetting) } }
21800     \cs_set_eq:NN \__regex_class:NnnnN \__regex_show_class:NnnnN
21801     \cs_set_protected:Npn \__regex_command_K:
21802     { \__regex_show_one:n { reset~match~start~(\iow_char:N\\K) } }
21803     \cs_set_protected:Npn \__regex_assertion:Nn ##1##2
21804     {
21805         \__regex_show_one:n
21806         { \bool_if:NF ##1 { negative~ } assertion:~##2 }
21807     }
21808     \cs_set:Npn \__regex_b_test: { word~boundary }
21809     \cs_set_eq:NN \__regex_anchor:N \__regex_show_anchor_to_str:N
21810     \cs_set_protected:Npn \__regex_item_caseful_equal:n ##1
21811     { \__regex_show_one:n { char~code~\int_eval:n{##1} } }
21812     \cs_set_protected:Npn \__regex_item_caseful_range:nn ##1##2
21813     {
21814         \__regex_show_one:n
21815         { range~[\int_eval:n{##1}, \int_eval:n{##2}] }
21816     }
21817     \cs_set_protected:Npn \__regex_item_caseless_equal:n ##1
21818     { \__regex_show_one:n { char~code~\int_eval:n{##1}~(caseless) } }
21819     \cs_set_protected:Npn \__regex_item_caseless_range:nn ##1##2
21820     {
21821         \__regex_show_one:n
21822         { Range~[\int_eval:n{##1}, \int_eval:n{##2}]~(caseless) }

```

```

21823     }
21824     \cs_set_protected:Npn \__regex_item_catcode:nT
21825         { \__regex_show_item_catcode:NnT \c_true_bool }
21826     \cs_set_protected:Npn \__regex_item_catcode_reverse:nT
21827         { \__regex_show_item_catcode:NnT \c_false_bool }
21828     \cs_set_protected:Npn \__regex_item_reverse:n
21829         { \__regex_show_scope:nn { Reversed~match } }
21830     \cs_set_protected:Npn \__regex_item_exact:nn ##1##2
21831         { \__regex_show_one:n { char~##2,~catcode~##1 } }
21832     \cs_set_eq:NN \__regex_item_exact_cs:n \__regex_show_item_exact_cs:n
21833     \cs_set_protected:Npn \__regex_item_cs:n
21834         { \__regex_show_scope:nn { control~sequence } }
21835     \cs_set:cpn { __regex_prop_.. } { \__regex_show_one:n { any~token } }
21836     \seq_clear:N \l__regex_show_prefix_seq
21837     \__regex_show_push:n { ~ }
21838     \cs_if_exist_use:N #1
21839     \tl_build_end:N \l__regex_build_tl
21840     \exp_args:NNNo
21841     \group_end:
21842     \tl_set:Nn \l__regex_internal_a_tl { \l__regex_build_tl }
21843 }

```

(End definition for `\__regex_show: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.

```

21844 \cs_new_protected:Npn \__regex_show_one:n #1
21845 {
21846     \int_incr:N \l__regex_show_lines_int
21847     \tl_build_put_right:Nx \l__regex_build_tl
21848         {
21849             \exp_not:N \iow_newline:
21850             \seq_map_function:NN \l__regex_show_prefix_seq \use:n
21851                 #1
21852         }
21853     }

```

(End definition for `\__regex_show_one:n`.)

`\__regex_show_push:n` `\__regex_show_pop:` 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.

```

21854 \cs_new_protected:Npn \__regex_show_push:n #1
21855     { \seq_put_right:Nx \l__regex_show_prefix_seq { #1 ~ } }
21856 \cs_new_protected:Npn \__regex_show_pop:
21857     { \seq_pop_right:NN \l__regex_show_prefix_seq \l__regex_internal_a_tl }
21858 \cs_new_protected:Npn \__regex_show_scope:nn #1#2
21859     {
21860         \__regex_show_one:n {#1}
21861         \__regex_show_push:n { ~ }
21862         #2
21863         \__regex_show_pop:
21864     }

```

(End definition for `\__regex_show_push:n`, `\__regex_show_pop:`, and `\__regex_show_scope:nn`.)

```
\_regex_show_group_aux:nnnnN
```

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.

```
21865 \cs_new_protected:Npn \_regex_show_group_aux:nnnnN #1#2#3#4#5
21866 {
21867     \_regex_show_one:n { ,-group~begin #1 }
21868     \_regex_show_push:n { | }
21869     \use_ii:nn #2
21870     \_regex_show_pop:
21871     \_regex_show_one:n
21872         { '-group~end \_regex_msg_repeated:nnN {#3} {#4} #5 }
21873 }
```

*(End definition for `\_regex_show_group_aux:nnnnN`.)*

```
\_regex_show_class:NnnnN
```

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.

```
21874 \cs_set:Npn \_regex_show_class:NnnnN #1#2#3#4#5
21875 {
21876     \group_begin:
21877         \tl_build_begin:N \l__regex_build_tl
21878         \int_zero:N \l__regex_show_lines_int
21879         \_regex_show_push:n {~}
21880         #2
21881         \int_compare:nTF { \l__regex_show_lines_int = 0 }
21882             {
21883                 \group_end:
21884                 \_regex_show_one:n { \bool_if:NTF #1 { Fail } { Pass } }
21885             }
21886             {
21887                 \bool_if:nTF
21888                     { #1 && \int_compare_p:n { \l__regex_show_lines_int = 1 } }
21889                     {
21890                         \group_end:
21891                         #2
21892                         \tl_build_put_right:Nn \l__regex_build_tl
21893                             { \_regex_msg_repeated:nnN {#3} {#4} #5 }
21894                     }
21895                     {
21896                         \tl_build_end:N \l__regex_build_tl
21897                         \exp_args:NNNo
21898                         \group_end:
21899                         \tl_set:Nn \l__regex_internal_a_tl \l__regex_build_tl
21900                         \_regex_show_one:n
21901                             {
21902                                 \bool_if:NTF #1 { Match } { Don't~match }
21903                                 \_regex_msg_repeated:nnN {#3} {#4} #5
21904                             }
21905                         \tl_build_put_right:Nx \l__regex_build_tl
```

```

21906         { \exp_not:o \l__regex_internal_a_tl }
21907     }
21908 }
21909 }
```

(End definition for `\_regex_show_class:NnnnN`.)

`\_regex_show_anchor_to_str:N` The argument is an integer telling us where the anchor is. We convert that to the relevant info.

```

21910 \cs_new:Npn \_regex_show_anchor_to_str:N #1
21911 {
21912     anchor~at~
21913     \str_case:nnF { #1 }
21914     {
21915         { \l__regex_min_pos_int } { start~(\iow_char:N\A) }
21916         { \l__regex_start_pos_int } { start-of~match~(\iow_char:N\G) }
21917         { \l__regex_max_pos_int } { end~(\iow_char:N\Z) }
21918     }
21919     { <error:~'#1'~not~recognized> }
21920 }
```

(End definition for `\_regex_show_anchor_to_str:N`.)

`\_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.

```

21921 \cs_new_protected:Npn \_regex_show_item_catcode:NnT #1#2
21922 {
21923     \seq_set_split:Nnn \l__regex_internal_seq { } { CBEMTPUDSLOA }
21924     \seq_set_filter:NNn \l__regex_internal_seq \l__regex_internal_seq
21925     { \int_if_odd_p:n { #2 / \int_use:c { c__regex_catcode_##1_int } } }
21926     \_regex_show_scope:nn
21927     {
21928         categories~
21929         \seq_map_function:NN \l__regex_internal_seq \use:n
21930         ,
21931         \bool_if:NF #1 { negative~ } class
21932     }
21933 }
```

(End definition for `\_regex_show_item_catcode:NnT`.)

`\_regex_show_item_exact_cs:n`

```

21934 \cs_new_protected:Npn \_regex_show_item_exact_cs:n #1
21935 {
21936     \seq_set_split:Nnn \l__regex_internal_seq { \scan_stop: } {#1}
21937     \seq_set_map:NNn \l__regex_internal_seq
21938     \l__regex_internal_seq { \iow_char:N\##1 }
21939     \_regex_show_one:n
21940     { control-sequence~ \seq_use:Nn \l__regex_internal_seq { ~or~ } }
21941 }
```

(End definition for `\_regex_show_item_exact_cs:n`.)

## 38.4 Building

### 38.4.1 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.

```
21942 \int_new:N \l_regex_min_state_int  
21943 \int_set:Nn \l_regex_min_state_int { 1 }  
21944 \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.

```
21945 \int_new:N \l_regex_left_state_int  
21946 \int_new:N \l_regex_right_state_int  
21947 \seq_new:N \l_regex_left_state_seq  
21948 \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.

```
21949 \int_new:N \l_regex_capturing_group_int
```

(End definition for `\l_regex_capturing_group_int`.)

### 38.4.2 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`: inserted at the start of the regular expression to make it unanchored.
- `\__regex_action_success`: marks the exit state of the NFA.
- `\__regex_action_cost:n {\langle shift \rangle}` is a transition from the current `\langle state \rangle` to `\langle state \rangle + \langle shift \rangle`, 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 {\langle shift \rangle}`, and `\__regex_action_free_group:n {\langle shift \rangle}` are free transitions, which immediately perform the actions for the state `\langle state \rangle + \langle shift \rangle` 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:n {\langle key \rangle}` where the `\langle key \rangle` is a group number followed by `<` or `>` for the beginning or end of group. This causes the current position in the query to be stored as the `\langle key \rangle` submatch boundary.

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.

```

21950 \cs_new_protected:Npn \_regex_build:n #1
21951 {
21952     \_regex_compile:n {#1}
21953     \_regex_build:N \l__regex_internal_regex
21954 }
21955 \_kernel_patch:nnNpn
21956 { \_regex_trace_push:nnN { regex } { 1 } \_regex_build:N }
21957 {
21958     \_regex_trace_states:n { 2 }
21959     \_regex_trace_pop:nnN { regex } { 1 } \_regex_build:N
21960 }
21961 \cs_new_protected:Npn \_regex_build:N #1
21962 {
21963     \_regex_standard_escapechar:
21964     \int_zero:N \l__regex_capturing_group_int
21965     \int_set_eq:NN \l__regex_max_state_int \l__regex_min_state_int
21966     \_regex_build_new_state:
21967     \_regex_build_new_state:
21968     \_regex_toks_put_right:Nn \l__regex_left_state_int
21969         { \_regex_action_start_wildcard: }
21970     \_regex_group:nnnN {#1} { 1 } { 0 } \c_false_bool
21971     \_regex_toks_put_right:Nn \l__regex_right_state_int
21972         { \_regex_action_success: }
21973 }
```

(End definition for `\_regex_build:n` and `\_regex_build: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`;
- `\g__regex_thread_state_intarray` from `\l__regex_min_active_int` to `\l__regex_max_active_int`.

In fact, some data is stored in `\toks` registers (local) in the same ranges so these ranges mustn't overlap. This is done by setting `\l_regex_min_active_int` to `\l_regex_max_state_int` after building the NFA. Here, in this nested call to the matching code, we need the new versions of these ranges 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_active_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.

```

21974  \__kernel_patch:nnNpn
21975  { \__regex_trace_push:nnN { regex } { 1 } \__regex_build_for_cs:n }
21976  {
21977      \__regex_trace_states:n { 2 }
21978      \__regex_trace_pop:nnN { regex } { 1 } \__regex_build_for_cs:n
21979  }
21980 \cs_new_protected:Npn \__regex_build_for_cs:n #1
21981  {
21982      \int_set_eq:NN \l_regex_min_state_int \l_regex_max_active_int
21983      \int_set_eq:NN \l_regex_max_state_int \l_regex_min_state_int
21984      \__regex_build_new_state:
21985      \__regex_build_new_state:
21986      \__regex_push_lr_states:
21987      #1
21988      \__regex_pop_lr_states:
21989      \__regex_toks_put_right:Nn \l_regex_right_state_int
21990      {
21991          \if_int_compare:w \l_regex_curr_pos_int = \l_regex_max_pos_int
21992              \exp_after:wN \__regex_action_success:
21993          \fi:
21994      }
21995  }

```

(End definition for `\__regex_build_for_cs:n`.)

### 38.4.3 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.

```

21996 \cs_new_protected:Npn \__regex_push_lr_states:
21997  {
21998      \seq_push:No \l_regex_left_state_seq
21999      { \int_use:N \l_regex_left_state_int }
22000      \seq_push:No \l_regex_right_state_seq
22001      { \int_use:N \l_regex_right_state_int }
22002  }
22003 \cs_new_protected:Npn \__regex_pop_lr_states:
22004  {
22005      \seq_pop>NN \l_regex_left_state_seq \l_regex_internal_a_tl
22006      \int_set:Nn \l_regex_left_state_int \l_regex_internal_a_tl
22007      \seq_pop>NN \l_regex_right_state_seq \l_regex_internal_a_tl
22008      \int_set:Nn \l_regex_right_state_int \l_regex_internal_a_tl
22009  }

```

(End definition for `\_regex_push_lr_states`: and `\_regex_pop_lr_states`::.)

`\_regex_build_transition_left:NNN`  
`\_regex build transition right: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.

```
22010 \cs_new_protected:Npn \_regex_build_transition_left:NNN #1#2#3
22011   { \_regex_toks_put_left:Nx #2 { #1 { \int_eval:n { #3 - #2 } } } }
22012 \cs_new_protected:Npn \_regex_build_transition_right:nNn #1#2#3
22013   { \_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.

```
22014 \_kernel_patch:nnNNpn
22015   {
22016     \_regex_trace:nnx { regex } { 2 }
22017     {
22018       regex~new~state~
22019       L=\int_use:N \l__regex_left_state_int ~ -> ~
22020       R=\int_use:N \l__regex_right_state_int ~ -> ~
22021       M=\int_use:N \l__regex_max_state_int ~ -> ~
22022       \int_eval:n { \l__regex_max_state_int + 1 }
22023     }
22024   }
22025   { }
22026 \cs_new_protected:Npn \_regex_build_new_state:
22027   {
22028     \_regex_toks_clear:N \l__regex_max_state_int
22029     \int_set_eq:NN \l__regex_left_state_int \l__regex_right_state_int
22030     \int_set_eq:NN \l__regex_right_state_int \l__regex_max_state_int
22031     \int_incr:N \l__regex_max_state_int
22032   }
```

(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.

```
22033 \cs_new_protected:Npn \_regex_build_transitions_laziness:NNNNN #1#2#3#4#5
22034   {
22035     \_regex_build_new_state:
22036     \_regex_toks_put_right:Nx \l__regex_left_state_int
22037     {
22038       \if_meaning:w \c_true_bool #1
22039         #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
22040         #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
22041       \else:
22042         #4 { \int_eval:n { #5 - \l__regex_left_state_int } }
22043         #2 { \int_eval:n { #3 - \l__regex_left_state_int } }
22044       \fi:
22045     }
22046   }
```

(End definition for `\_regex_build_transitions_lazyness>NNNNN`.)

### 38.4.4 Building classes

`\_regex_class:NnnnN`  
`\_regex_tests_action_cost:n`

The arguments are:  $\langle\text{boolean}\rangle \{\langle\text{tests}\rangle\} \{\langle\text{min}\rangle\} \{\langle\text{more}\rangle\} \langle\text{lazyness}\rangle$ . 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  $\langle\text{more}\rangle$  is 0 for fixed repetitions, -1 for unbounded repetitions, and  $\langle\text{max}\rangle - \langle\text{min}\rangle$  for a range of repetitions.

```
22047 \cs_new_protected:Npn \_regex_class:NnnnN #1#2#3#4#5
22048 {
22049     \cs_set:Npx \_regex_tests_action_cost:n ##1
22050     {
22051         \exp_not:n { \exp_not:n {#2} }
22052         \bool_if:NTF #1
22053             { \_regex_break_point:TF { \_regex_action_cost:n {##1} } { } }
22054             { \_regex_break_point:TF { } { \_regex_action_cost:n {##1} } }
22055     }
22056     \if_case:w - #4 \exp_stop_f:
22057         \_regex_class_repeat:n {#3}
22058     \or: \_regex_class_repeat:nN {#3}      #5
22059     \else: \_regex_class_repeat:nnN {#3} {#4} #5
22060     \fi:
22061 }
22062 \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.

```
22063 \cs_new_protected:Npn \_regex_class_repeat:n #1
22064 {
22065     \prg_replicate:nn {#1}
22066     {
22067         \_regex_build_new_state:
22068         \_regex_build_transition_right:nNn \_regex_tests_action_cost:n
22069             \l__regex_left_state_int \l__regex_right_state_int
22070     }
22071 }
```

(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  $\#2$ .

```
22072 \cs_new_protected:Npn \_regex_class_repeat:nN #1#2
22073 {
22074     \if_int_compare:w #1 = 0 \exp_stop_f:
22075         \_regex_build_transitions_lazyness>NNNNN #2
```

```

22076     \_\_regex\_action\_free:n      \l\_regex\_right\_state\_int
22077     \_\_regex\_tests\_action\_cost:n \l\_regex\_left\_state\_int
22078 \_else:
22079     \_\_regex\_class\_repeat:n {#1}
22080     \int\_set\_eq:NN \l\_regex\_internal\_a\_int \l\_regex\_left\_state\_int
22081     \_\_regex\_build\_transitions\_lazyness:NNNNN #2
22082     \_\_regex\_action\_free:n \l\_regex\_right\_state\_int
22083     \_\_regex\_action\_free:n \l\_regex\_internal\_a\_int
22084 \_fi:
22085 }

```

(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`.

```

22086 \cs\_new\_protected:Npn \_\_regex\_class\_repeat:nnN #1#2#3
22087 {
22088     \_\_regex\_class\_repeat:n {#1}
22089     \int\_set:Nn \l\_regex\_internal\_a\_int
22090         { \l\_regex\_max\_state\_int + #2 - 1 }
22091     \prg\_replicate:nn { #2 }
22092     {
22093         \_\_regex\_build\_transitions\_lazyness:NNNNN #3
22094         \_\_regex\_action\_free:n      \l\_regex\_internal\_a\_int
22095         \_\_regex\_tests\_action\_cost:n \l\_regex\_right\_state\_int
22096     }
22097 }

```

(End definition for `\_\_regex\_class\_repeat:nnN`.)

### 38.4.5 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.

```

22098 \_\_kernel\_patch:nnNNpn
22099   { \_\_regex\_trace\_push:nnN { regex } { 1 } \_\_regex\_group\_aux:nnnnN }
22100   { \_\_regex\_trace\_pop:nnN { regex } { 1 } \_\_regex\_group\_aux:nnnnN }
22101 \cs\_new\_protected:Npn \_\_regex\_group\_aux:nnnnN #1#2#3#4#5
22102   {
22103     \if\_int\_compare:w #3 = 0 \exp\_stop\_f:
22104       \_\_regex\_build\_new\_state:
22105   \assert\assert\_int:n { \l\_regex\_max\_state\_int = \l\_regex\_right\_state\_int + 1 }

```

```

22106     \_\_regex_build_transition_right:nNn \_\_regex_action_free_group:n
22107     \l\_regex_left_state_int \l\_regex_right_state_int
22108 \fi:
22109 \_\_regex_build_new_state:
22110 \_\_regex_push_lr_states:
22111 #2
22112 \_\_regex_pop_lr_states:
22113 \if_case:w - #4 \exp_stop_f:
22114     \_\_regex_group_repeat:nn {#1} {#3}
22115 \or: \_\_regex_group_repeat:nnN {#1} {#3}      #5
22116 \else: \_\_regex_group_repeat:nnnN {#1} {#3} {#4} {#5}
22117 \fi:
22118 }

```

(End definition for `\_\_regex_group_aux:nnnnN`.)

`\_\_regex_group:nnnN` Hand to `\_\_regex_group_aux:nnnnnN` the label of that group (expanded), and the group itself, with some extra commands to perform.

```

22119 \cs_new_protected:Npn \_\_regex_group:nnnN #1
22120 {
22121     \exp_args:No \_\_regex_group_aux:nnnnN
22122     { \int_use:N \l\_regex_capturing_group_int }
22123     {
22124         \int_incr:N \l\_regex_capturing_group_int
22125         #1
22126     }
22127 }
22128 \cs_new_protected:Npn \_\_regex_group_no_capture:nnnN
22129     { \_\_regex_group_aux:nnnnN { -1 } }

```

(End definition for `\_\_regex_group:nnnN` and `\_\_regex_group_no_capture:nnnN`.)

`\_\_regex_group_resetting: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>}`.

```

22130 \cs_new_protected:Npn \_\_regex_group_resetting:nnnN #1
22131 {
22132     \_\_regex_group_aux:nnnnN { -1 }
22133     {
22134         \exp_args:Noo \_\_regex_group_resetting_loop:nnNn
22135         { \int_use:N \l\_regex_capturing_group_int }
22136         { \int_use:N \l\_regex_capturing_group_int }
22137         #1
22138         { ?? \prg_break:n } { }
22139         \prg_break_point:
22140     }
22141 }
22142 \cs_new_protected:Npn \_\_regex_group_resetting_loop:nnNn #1#2#3#4
22143 {
22144     \use_none:nn #3 { \int_set:Nn \l\_regex_capturing_group_int {#1} }
22145     \int_set:Nn \l\_regex_capturing_group_int {#2}
22146     #3 {#4}
22147     \exp_args:Nf \_\_regex_group_resetting_loop:nnNn

```

```

22148 { \int_max:nn {#1} { \l__regex_capturing_group_int } }
22149 {#2}
22150 }

```

(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.

```

22151 \_kernel_patch:nnNNpn
22152 { \_regex_trace_push:nnN { regex } { 1 } \_regex_branch:n }
22153 { \_regex_trace_pop:nnN { regex } { 1 } \_regex_branch:n }
22154 \cs_new_protected:Npn \_regex_branch:n #1
22155 {
22156     \_regex_build_new_state:
22157     \seq_get:NN \l__regex_left_state_seq \l__regex_internal_a_tl
22158     \int_set:Nn \l__regex_left_state_int \l__regex_internal_a_tl
22159     \_regex_build_transition_right:nNn \_regex_action_free:n
22160         \l__regex_left_state_int \l__regex_right_state_int
22161     #1
22162     \seq_get>NN \l__regex_right_state_seq \l__regex_internal_a_tl
22163     \_regex_build_transition_right:nNn \_regex_action_free:n
22164         \l__regex_right_state_int \l__regex_internal_a_tl
22165 }

```

(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).

```

22166 \cs_new_protected:Npn \_regex_group_repeat:nn #1#2
22167 {
22168     \if_int_compare:w #2 = 0 \exp_stop_f:
22169         \int_set:Nn \l__regex_max_state_int
22170             { \l__regex_left_state_int - 1 }
22171         \_regex_build_new_state:
22172     \else:
22173         \_regex_group_repeat_aux:n {#2}
22174         \_regex_group_submatches:nNN {#1}
22175             \l__regex_internal_a_int \l__regex_right_state_int
22176             \_regex_build_new_state:
22177     \fi:
22178 }

```

(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`.

```

22179 \cs_new_protected:Npn \_regex_group_submatches:nNN #1#2#3
22180 {

```

```

22181      \if_int_compare:w #1 > - 1 \exp_stop_f:
22182          \_regex_toks_put_left:Nx #2 { \_regex_action_submatch:n { #1 < } }
22183          \_regex_toks_put_left:Nx #3 { \_regex_action_submatch:n { #1 > } }
22184      \fi:
22185  }

```

(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.

```

22186 \cs_new_protected:Npn \_regex_group_repeat_aux:n #1
22187  {
22188      \_regex_build_transition_right:nNn \_regex_action_free:n
22189          \l__regex_right_state_int \l__regex_max_state_int
22190          \int_set_eq:NN \l__regex_internal_a_int \l__regex_left_state_int
22191          \int_set_eq:NN \l__regex_internal_b_int \l__regex_max_state_int
22192          \if_int_compare:w \int_eval:n {#1} > 1 \exp_stop_f:
22193              \int_set:Nn \l__regex_internal_c_int
22194              {
22195                  ( #1 - 1 )
22196                  * ( \l__regex_internal_b_int - \l__regex_internal_a_int )
22197              }
22198              \int_add:Nn \l__regex_right_state_int { \l__regex_internal_c_int }
22199              \int_add:Nn \l__regex_max_state_int { \l__regex_internal_c_int }
22200              \_regex_toks_memcpy>NNn
22201                  \l__regex_internal_b_int
22202                  \l__regex_internal_a_int
22203                  \l__regex_internal_c_int
22204      \fi:
22205  }

```

(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`.

```

22206 \cs_new_protected:Npn \_regex_group_repeat:nnN #1#2#3
22207  {
22208      \if_int_compare:w #2 = 0 \exp_stop_f:
22209          \_regex_group_submatches:nNN {#1}
22210          \l__regex_left_state_int \l__regex_right_state_int

```

```

22211 \int_set:Nn \l__regex_internal_a_int
22212   { \l__regex_left_state_int - 1 }
22213 \__regex_build_transition_right:nNn \__regex_action_free:n
22214   \l__regex_right_state_int \l__regex_internal_a_int
22215 \__regex_build_new_state:
22216 \if_meaning:w \c_true_bool #3
22217   \__regex_build_transition_left:NNN \__regex_action_free:n
22218     \l__regex_internal_a_int \l__regex_right_state_int
22219 \else:
22220   \__regex_build_transition_right:nNn \__regex_action_free:n
22221     \l__regex_internal_a_int \l__regex_right_state_int
22222 \fi:
22223 \else:
22224   \__regex_group_repeat_aux:n {#2}
22225   \__regex_group_submatches:nNN {#1}
22226   \l__regex_internal_a_int \l__regex_right_state_int
22227 \if_meaning:w \c_true_bool #3
22228   \__regex_build_transition_right:nNn \__regex_action_free_group:n
22229     \l__regex_right_state_int \l__regex_internal_a_int
22230 \else:
22231   \__regex_build_transition_left:NNN \__regex_action_free_group:n
22232     \l__regex_right_state_int \l__regex_internal_a_int
22233 \fi:
22234   \__regex_build_new_state:
22235 \fi:
22236 }

```

(End definition for `\__regex_group_repeat:nnN`.)

`\__regex_group_repeat:nnN`

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.

```

22237 \cs_new_protected:Npn \__regex_group_repeat:nnnN #1#2#3#4
22238 {
22239   \__regex_group_submatches:nNN {#1}
22240   \l__regex_left_state_int \l__regex_right_state_int
22241   \__regex_group_repeat_aux:n { #2 + #3 }
22242 \if_meaning:w \c_true_bool #4
22243   \int_set_eq:NN \l__regex_left_state_int \l__regex_max_state_int
22244   \prg_replicate:nn { #3 }
22245   {
22246     \int_sub:Nn \l__regex_left_state_int
22247       { \l__regex_internal_b_int - \l__regex_internal_a_int }
22248   \__regex_build_transition_left:NNN \__regex_action_free:n
22249     \l__regex_left_state_int \l__regex_max_state_int

```

```

22250     }
22251 \else:
22252     \prg_replicate:nn { #3 - 1 }
22253     {
22254         \int_sub:Nn \l__regex_right_state_int
22255             { \l__regex_internal_b_int - \l__regex_internal_a_int }
22256         \__regex_build_transition_right:nNn \__regex_action_free:n
22257             \l__regex_right_state_int \l__regex_max_state_int
22258     }
22259     \if_int_compare:w #2 = 0 \exp_stop_f:
22260         \int_set:Nn \l__regex_right_state_int
22261             { \l__regex_left_state_int - 1 }
22262     \else:
22263         \int_sub:Nn \l__regex_right_state_int
22264             { \l__regex_internal_b_int - \l__regex_internal_a_int }
22265     \fi:
22266     \__regex_build_transition_right:nNn \__regex_action_free:n
22267         \l__regex_right_state_int \l__regex_max_state_int
22268     \fi:
22269     \__regex_build_new_state:
22270 }

```

(End definition for `\__regex_group_repeat:nnnN`.)

### 38.4.6 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.  
`\__regex_b_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. Anchors at the start or end of match use `\__regex_anchor:N`, with a position controlled by the integer `#1`.

```

22271 \cs_new_protected:Npn \__regex_assertion:Nn #1#2
22272     {
22273         \__regex_build_new_state:
22274         \__regex_toks_put_right:Nx \l__regex_left_state_int
22275         {
22276             \exp_not:n {#2}
22277             \__regex_break_point:TF
22278                 \bool_if:NF #1 { { } }
22279                 {
22280                     \__regex_action_free:n
22281                         {
22282                             \int_eval:n
22283                                 { \l__regex_right_state_int - \l__regex_left_state_int }
22284                         }
22285                     }
22286                     \bool_if:NT #1 { { } }
22287                 }
22288             }
22289 \cs_new_protected:Npn \__regex_anchor:N #1
22290     {
22291         \if_int_compare:w #1 = \l__regex_curr_pos_int
22292             \exp_after:wN \__regex_break_true:w

```

```

22293     \fi:
22294   }
22295 \cs_new_protected:Npn \__regex_b_test:
22296   {
22297     \group_begin:
22298       \int_set_eq:NN \l__regex_curr_char_int \l__regex_last_char_int
22299       \__regex_prop_w:
22300       \__regex_break_point:TF
22301         { \group_end: \__regex_item_reverse:n \__regex_prop_w: }
22302         { \group_end: \__regex_prop_w: }
22303   }

```

(End definition for `\__regex_assertion:Nn`, `\__regex_b_test:`, and `\__regex_anchor:N`.)

`\__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.

```

22304 \cs_new_protected:Npn \__regex_command_K:
22305   {
22306     \__regex_build_new_state:
22307     \__regex_toks_put_right:Nx \l__regex_left_state_int
22308     {
22309       \__regex_action_submatch:n { 0< }
22310       \bool_set_true:N \l__regex_fresh_thread_bool
22311       \__regex_action_free:n
22312         {
22313           \int_eval:n
22314             { \l__regex_right_state_int - \l__regex_left_state_int }
22315         }
22316       \bool_set_false:N \l__regex_fresh_thread_bool
22317     }
22318   }

```

(End definition for `\__regex_command_K`.)

### 38.5 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_state_intarray`: 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.

### 38.5.1 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 don’t start from 0 because the `\toks` registers with low numbers are used to hold the states of the NFA. We match without backtracking, keeping all threads in lockstep at the `current_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.

```
22319 \int_new:N \l\_regex_min_pos_int
22320 \int_new:N \l\_regex_max_pos_int
22321 \int_new:N \l\_regex_curr_pos_int
22322 \int_new:N \l\_regex_start_pos_int
22323 \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_last_char_int`  
`\l\_regex_case_changed_char_int`

The character and category codes of the token at the current position; the character code of the token at the previous position; and the character code of the result of changing the case of the current token (`A-Z↔a-z`). This last integer is only computed when necessary, and is otherwise `\c_max_int`. The `current_char` variable is also used in various other phases to hold a character code.

```
22324 \int_new:N \l\_regex_curr_char_int
22325 \int_new:N \l\_regex_curr_catcode_int
22326 \int_new:N \l\_regex_last_char_int
22327 \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.

```
22328 \int_new:N \l\_regex_curr_state_int
```

(End definition for `\l\_regex_curr_state_int`.)

`\l\_regex_curr_submatches_prop`  
`\l\_regex_success_submatches_prop`

The submatches for the thread which is currently active are stored in the `current_-submatches` property list variable. This property list is stored by `\_\_regex_action_-cost:n` into the `\toks` register for the target state of the transition, to be retrieved when matching at the next position. When a thread succeeds, this property list is copied to `\l\_regex_success_submatches_prop`: only the last successful thread remains there.

```
22329 \prop_new:N \l__regex_curr_submatches_prop  
22330 \prop_new:N \l__regex_success_submatches_prop
```

(End definition for `\l__regex_curr_submatches_prop` and `\l__regex_success_submatches_prop`.)

`\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 `<state>` 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).

```
22331 \int_new:N \l__regex_step_int
```

(End definition for `\l__regex_step_int`.)

`\l__regex_min_active_int`  
`\l__regex_max_active_int`

All the currently active threads are kept in order of precedence in `\g__regex_thread_state_intarray`, and the corresponding submatches in the `\toks`. For our purposes, those serve as an array, indexed from `min_active` (inclusive) to `max_active` (excluded). At the start of every step, the whole array is unpacked, so that the space can immediately be reused, and `max_active` is reset to `min_active`, effectively clearing the array.

```
22332 \int_new:N \l__regex_min_active_int
```

```
22333 \int_new:N \l__regex_max_active_int
```

(End definition for `\l__regex_min_active_int` and `\l__regex_max_active_int`.)

`\g__regex_state_active_intarray`  
`\g__regex_thread_state_intarray`

`\g__regex_state_active_intarray` stores the last `<step>` in which each `<state>` was active. `\g__regex_thread_state_intarray` stores threads to be considered in the next step, more precisely the states in which these threads are.

```
22334 \intarray_new:Nn \g__regex_state_active_intarray { 65536 }
```

```
22335 \intarray_new:Nn \g__regex_thread_state_intarray { 65536 }
```

(End definition for `\g__regex_state_active_intarray` and `\g__regex_thread_state_intarray`.)

`\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`.

```
22336 \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.

```

22337 \bool_new:N \l__regex_fresh_thread_bool
22338 \bool_new:N \l__regex_empty_success_bool
22339 \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.

```

22340 \bool_new:N \g__regex_success_bool
22341 \bool_new:N \l__regex_saved_success_bool
22342 \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.)
```

### 38.5.2 Matching: framework

\\_\_regex\_match:n  
\\_\_regex\_match\_cs:n  
\\_\_regex\_match\_init:

First store the query into \toks registers and arrays (see \\_\_regex\_query\_set:nnn). Then 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.

```

22343 \__kernel_patch:nnNNpn
22344 {
22345     \__regex_trace_push:nnN { regex } { 1 } \__regex_match:n
22346     \__regex_trace:nnx { regex } { 1 } { analyzing~query~token~list }
22347 }
22348 { \__regex_trace_pop:nnN { regex } { 1 } \__regex_match:n }
22349 \cs_new_protected:Npn \__regex_match:n #1
22350 {
22351     \int_zero:N \l__regex_balance_int
22352     \int_set:Nn \l__regex_curr_pos_int { 2 * \l__regex_max_state_int }
22353     \__regex_query_set:nnn { } { -1 } { -2 }
22354     \int_set_eq:NN \l__regex_min_pos_int \l__regex_curr_pos_int
22355     \tl_analysis_map_inline:nn {#1}
22356         { \__regex_query_set:nnn {##1} {##3} {##2} }
22357     \int_set_eq:NN \l__regex_max_pos_int \l__regex_curr_pos_int
22358     \__regex_query_set:nnn { } { -1 } { -2 }
22359     \__regex_match_init:
22360     \__regex_match_once:
22361 }
22362 \__kernel_patch:nnNNpn
22363 {
22364     \__regex_trace_push:nnN { regex } { 1 } \__regex_match_cs:n
```

```

22365     \_\_regex_trace:nnx { regex } { 1 } { analyzing~query~token~list }
22366     }
22367     { \_\_regex_trace_pop:nnN { regex } { 1 } \_\_regex_match_cs:n }
22368 \cs_new_protected:Npn \_\_regex_match_cs:n #1
22369     {
22370         \int_zero:N \l\_regex_balance_int
22371         \int_set:Nn \l\_regex_curr_pos_int
22372             {
22373                 \int_max:nn { 2 * \l\_regex_max_state_int - \l\_regex_min_state_int }
22374                 { \l\_regex_max_pos_int }
22375                 + 1
22376             }
22377         \_\_regex_query_set:nnn { } { -1 } { -2 }
22378         \int_set_eq:NN \l\_regex_min_pos_int \l\_regex_curr_pos_int
22379         \str_map_inline:nn {#1}
22380             {
22381                 \_\_regex_query_set:nnn { \exp_not:n {##1} }
22382                     { \tl_if_blank:nTF {##1} { 10 } { 12 } }
22383                     { '##1 }
22384             }
22385         \int_set_eq:NN \l\_regex_max_pos_int \l\_regex_curr_pos_int
22386         \_\_regex_query_set:nnn { } { -1 } { -2 }
22387         \_\_regex_match_init:
22388             \_\_regex_match_once:
22389         }
22390     \_\_kernel_patch:nnNNpn
22391         { \_\_regex_trace:nnx { regex } { 1 } { initializing } }
22392         { }
22393 \cs_new_protected:Npn \_\_regex_match_init:
22394     {
22395         \bool_gset_false:N \g\_regex_success_bool
22396         \int_step_inline:nnn
22397             \l\_regex_min_state_int { \l\_regex_max_state_int - 1 }
22398             {
22399                 \_\_kernel_intarray_gset:Nnn
22400                     \g\_regex_state_active_intarray {##1} { 1 }
22401             }
22402         \int_set_eq:NN \l\_regex_min_active_int \l\_regex_max_state_int
22403         \int_zero:N \l\_regex_step_int
22404         \int_set_eq:NN \l\_regex_success_pos_int \l\_regex_min_pos_int
22405         \int_set:Nn \l\_regex_min_submatch_int
22406             { 2 * \l\_regex_max_state_int }
22407         \int_set_eq:NN \l\_regex_submatch_int \l\_regex_min_submatch_int
22408         \bool_set_false:N \l\_regex_empty_success_bool
22409     }

```

(End definition for \\_\\_regex\_match:n, \\_\\_regex\_match\_cs:n, and \\_\\_regex\_match\_init::)

\\_\\_regex\_match\_once: This function finds one match, then does some action defined by the `every_match` token list, which may recursively call \\_\\_regex\_match\_once:. 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, and get that token, to set `last_char` properly for word boundaries. Then call `\_\_regex_match_loop:`, which runs through the query until the end or until a successful match breaks early.

```

22410 \cs_new_protected:Npn \_\_regex_match_once:
22411 {
22412     \if_meaning:w \c_true_bool \l_\_regex_empty_success_bool
22413         \cs_set:Npn \_\_regex_if_two_empty_matches:F
22414             {
22415                 \int_compare:nNnF
22416                     \l_\_regex_start_pos_int = \l_\_regex_curr_pos_int
22417             }
22418     \else:
22419         \cs_set_eq:NN \_\_regex_if_two_empty_matches:F \use:n
22420     \fi:
22421     \int_set_eq:NN \l_\_regex_start_pos_int \l_\_regex_success_pos_int
22422     \bool_set_false:N \l_\_regex_match_success_bool
22423     \prop_clear:N \l_\_regex_curr_submatches_prop
22424     \int_set_eq:NN \l_\_regex_max_active_int \l_\_regex_min_active_int
22425     \_\_regex_store_state:n { \l_\_regex_min_state_int }
22426     \int_set:Nn \l_\_regex_curr_pos_int
22427         { \l_\_regex_start_pos_int - 1 }
22428     \_\_regex_query_get:
22429     \_\_regex_match_loop:
22430     \l_\_regex_every_match_tl
22431 }
```

(End definition for `\_\_regex_match_once:..`)

`\_\_regex_single_match:` `\_\_regex_multi_match:n`: 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.

```

22432 \cs_new_protected:Npn \_\_regex_single_match:
22433 {
22434     \tl_set:Nn \l_\_regex_every_match_tl
22435     {
22436         \bool_gset_eq:NN
22437             \g_\_regex_success_bool
22438             \l_\_regex_match_success_bool
22439     }
22440 }
22441 \cs_new_protected:Npn \_\_regex_multi_match:n #1
22442 {
22443     \tl_set:Nn \l_\_regex_every_match_tl
22444     {
22445         \if_meaning:w \c_true_bool \l_\_regex_match_success_bool
22446             \bool_gset_true:N \g_\_regex_success_bool
22447             #1
22448             \exp_after:wN \_\_regex_match_once:
22449         \fi:
22450     }
22451 }
```

(End definition for `\_\_regex_single_match:` and `\_\_regex_multi_match:n`.)

`\_\_regex\_match\_loop:` 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_active`). This results in a sequence of `\_\_regex\_use\_state\_and\_submatches:nn` {`(state)`} {`(prop)`}, 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 what `\_\_regex\_match\_once:` matches. We explain the `fresh_thread` business when describing `\_\_regex\_action\_wildcard::`.

```

22452 \cs_new_protected:Npn \_\_regex_match_loop:
22453   {
22454     \int_add:Nn \l__regex_step_int { 2 }
22455     \int_incr:N \l__regex_curr_pos_int
22456     \int_set_eq:NN \l__regex_last_char_int \l__regex_curr_char_int
22457     \int_set_eq:NN \l__regex_case_changed_char_int \c_max_int
22458     \_\_regex_query_get:
22459     \use:x
22460       {
22461         \int_set_eq:NN \l__regex_max_active_int \l__regex_min_active_int
22462         \int_step_function:nnN
22463           { \l__regex_min_active_int }
22464           { \l__regex_max_active_int - 1 }
22465           \_\_regex_match_one_active:n
22466         }
22467       \prg_break_point:
22468       \bool_set_false:N \l__regex_fresh_thread_bool
22469       \if_int_compare:w \l__regex_max_active_int > \l__regex_min_active_int
22470         \if_int_compare:w \l__regex_curr_pos_int < \l__regex_max_pos_int
22471           \exp_after:wN \exp_after:wN \exp_after:wN \_\_regex_match_loop:
22472         \fi:
22473       \fi:
22474     }
22475 \cs_new:Npn \_\_regex_match_one_active:n #1
22476   {
22477     \_\_regex_use_state_and_submatches:nn
22478       { \_\_kernel_intarray_item:Nn \g__regex_thread_state_intarray {#1} }
22479       { \_\_regex_toks_use:w #1 }
22480   }

```

*(End definition for `\_\_regex_match_loop:` and `\_\_regex_match_one_active:n.`)*

`\_\_regex_query_set:nnn` The arguments are: tokens that `o` and `x` expand to one token of the query, the catcode, and the character code. Store those, and the current brace balance (used later to check for overall brace balance) in a `\toks` register and some arrays, then update the `balance`.

```

22481 \cs_new_protected:Npn \_\_regex_query_set:nnn #1#2#3
22482   {
22483     \_\_kernel_intarray_gset:Nnn \g__regex_charcode_intarray
22484       { \l__regex_curr_pos_int } {#3}
22485     \_\_kernel_intarray_gset:Nnn \g__regex_catcode_intarray
22486       { \l__regex_curr_pos_int } {#2}
22487     \_\_kernel_intarray_gset:Nnn \g__regex_balance_intarray
22488       { \l__regex_curr_pos_int } { \l__regex_balance_int }
22489     \_\_regex_toks_set:Nn \l__regex_curr_pos_int {#1}
22490     \int_incr:N \l__regex_curr_pos_int

```

```

22491     \if_case:w #2 \exp_stop_f:
22492     \or: \int_incr:N \l__regex_balance_int
22493     \or: \int_decr:N \l__regex_balance_int
22494     \fi:
22495 }

```

(End definition for `\_regex_query_set:nnn.`)

`\_regex_query_get:` Extract the current character and category codes at the current position from the appropriate arrays.

```

22496 \cs_new_protected:Npn \_regex_query_get:
22497 {
22498     \l__regex_curr_char_int
22499         = \_kernel_intarray_item:Nn \g__regex_charcode_intarray
22500             { \l__regex_curr_pos_int } \scan_stop:
22501     \l__regex_curr_catcode_int
22502         = \_kernel_intarray_item:Nn \g__regex_catcode_intarray
22503             { \l__regex_curr_pos_int } \scan_stop:
22504 }

```

(End definition for `\_regex_query_get:..`)

### 38.5.3 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.

```

22505 \_kernel_patch:nnNnpn
22506 {
22507     \_regex_trace:nnx { regex } { 2 }
22508         { state~\int_use:N \l__regex_curr_state_int }
22509     }
22510     { }
22511 \cs_new_protected:Npn \_regex_use_state:
22512 {
22513     \_kernel_intarray_gset:Nnn \g__regex_state_active_intarray
22514         { \l__regex_curr_state_int } { \l__regex_step_int }
22515     \_regex_toks_use:w \l__regex_curr_state_int
22516     \_kernel_intarray_gset:Nnn \g__regex_state_active_intarray
22517         { \l__regex_curr_state_int }
22518         { \int_eval:n { \l__regex_step_int + 1 } }
22519 }

```

(End definition for `\_regex_use_state:..`)

`\_regex_use_state_and_submatches:nn` 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 `current_state` and `current_submatches` and use the state if it has not yet been encountered at this step.

```

22520 \cs_new_protected:Npn \_regex_use_state_and_submatches:nn #1 #2
22521 {
22522     \int_set:Nn \l__regex_curr_state_int {#1}
22523     \if_int_compare:w
22524         \_kernel_intarray_item:Nn \g__regex_state_active_intarray

```

```

22525         { \l__regex_curr_state_int }
22526             < \l__regex_step_int
22527     \tl_set:Nn \l__regex_curr_submatches_prop {\#2}
22528     \exp_after:wN \__regex_use_state:
22529     \fi:
22530     \scan_stop:
22531 }
```

(End definition for `\__regex_use_state_and_submatches:nn`.)

### 38.5.4 Actions when matching

`\__regex_action_start_wildcard:`

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_loop`: too.

```

22532 \cs_new_protected:Npn \__regex_action_start_wildcard:
22533 {
22534     \bool_set_true:N \l__regex_fresh_thread_bool
22535     \__regex_action_free:n {1}
22536     \bool_set_false:N \l__regex_fresh_thread_bool
22537     \__regex_action_cost:n {0}
22538 }
```

(End definition for `\__regex_action_start_wildcard:..`)

`\__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.

```

22539 \cs_new_protected:Npn \__regex_action_free:n
22540     { \__regex_action_free_aux:nn { > \l__regex_step_int \else: } }
22541 \cs_new_protected:Npn \__regex_action_free_group:n
22542     { \__regex_action_free_aux:nn { < \l__regex_step_int } }
22543 \cs_new_protected:Npn \__regex_action_free_aux:nn #1#2
22544 {
22545     \use:x
22546     {
22547         \int_add:Nn \l__regex_curr_state_int {\#2}
22548         \exp_not:n
22549         {
22550             \if_int_compare:w
22551                 \__kernel_intarray_item:Nn \g__regex_state_active_intarray
22552                 { \l__regex_curr_state_int }
22553                 #1
22554                 \exp_after:wN \__regex_use_state:
22555                 \fi:
22556             }
22557         \int_set:Nn \l__regex_curr_state_int
```

```

22558     { \int_use:N \l__regex_curr_state_int }
22559     \tl_set:Nn \exp_not:N \l__regex_curr_submatches_prop
22560     { \exp_not:o \l__regex_curr_submatches_prop }
22561   }
22562 }
```

(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.

```

22563 \cs_new_protected:Npn \_regex_action_cost:n #1
22564   {
22565     \exp_args:Nx \_regex_store_state:n
22566     { \int_eval:n { \l__regex_curr_state_int + #1 } }
22567   }
```

(End definition for `\_regex_action_cost:n`.)

`\_regex_store_state:n`, `\_regex_store_submatches:`: Put the given state in `\g__regex_thread_state_intarray`, and increment the length of the array. Also store the current submatch in the appropriate `\toks`.

```

22568 \cs_new_protected:Npn \_regex_store_state:n #1
22569   {
22570     \_regex_store_submatches:
22571     \kernel_intarray_gset:Nnn \g__regex_thread_state_intarray
22572     { \l__regex_max_active_int } {#1}
22573     \int_incr:N \l__regex_max_active_int
22574   }
22575 \cs_new_protected:Npn \_regex_store_submatches:
22576   {
22577     \_regex_toks_set:No \l__regex_max_active_int
22578     { \l__regex_curr_submatches_prop }
22579   }
```

(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.

```

22580 \cs_new_protected:Npn \_regex_disable_submatches:
22581   {
22582     \cs_set_protected:Npn \_regex_store_submatches: { }
22583     \cs_set_protected:Npn \_regex_action_submatch:n ##1 { }
22584   }
```

(End definition for `\_regex_disable_submatches:..`)

`\_regex_action_submatch:n`: Update the current submatches with the information from the current position. Maybe a bottleneck.

```

22585 \cs_new_protected:Npn \_regex_action_submatch:n #1
22586   {
22587     \prop_put:Nno \l__regex_curr_submatches_prop {#1}
22588     { \int_use:N \l__regex_curr_pos_int }
22589   }
```

(End definition for `\_\_regex\_action\_submatch:n`.)

`\_\_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.

```
22590 \cs_new_protected:Npn \_\_regex\_action\_success:
22591 {
22592     \_\_regex_if_two_empty_matches:F
22593     {
22594         \bool_set_true:N \l__regex_match_success_bool
22595         \bool_set_eq:NN \l__regex_empty_success_bool
22596             \l__regex_fresh_thread_bool
22597         \int_set_eq:NN \l__regex_success_pos_int \l__regex_curr_pos_int
22598         \prop_set_eq:NN \l__regex_success_submatches_prop
22599             \l__regex_curr_submatches_prop
22600         \prg_break:
22601     }
22602 }
```

(End definition for `\_\_regex\_action\_success:`.)

## 38.6 Replacement

### 38.6.1 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 `}``.

```
22603 \int_new:N \l__regex_replacement_csnames_int
```

(End definition for `\l__regex_replacement_csnames_int`.)

`\l__regex_replacement_category_tl` This sequence of letters is used to correctly restore categories in nested constructions such as `\cL(abc\cD(__)d)`.

```
22604 \tl_new:N \l__regex_replacement_category_tl
22605 \seq_new:N \l__regex_replacement_category_seq
```

(End definition for `\l__regex_replacement_category_tl` and `\l__regex_replacement_category_seq`.)

`\l__regex_balance_tl` This token list holds the replacement text for `\_\_regex_replacement_balance_one_match:n` while it is being built incrementally.

```
22606 \tl_new:N \l__regex_balance_tl
```

(End definition for `\l__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).

```
22607 \cs_new:Npn \__regex_replacement_balance_one_match:n #1
22608   { - \__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.

```
22609 \cs_new:Npn \__regex_replacement_do_one_match:n #1
22610   {
22611     \__regex_query_range:nn
22612       { \__kernel_intarray_item:Nn \g__regex_submatch_prev_intarray {#1} }
22613       { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
22614   }
```

(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 ##.

```
22615 \cs_new:Npn \__regex_replacement_exp_not:N #1 { \exp_not:n {#1} }
```

(End definition for `\__regex_replacement_exp_not:N`.)

### 38.6.2 Query and brace balance

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 {<min>} {<max>}` unpacks registers from the position `<min>` to the position `<max>` – 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.

```
22616 \cs_new:Npn \__regex_query_range:nn #1#2
22617   {
22618     \exp_after:wN \__regex_query_range_loop:ww
22619     \int_value:w \__regex_int_eval:w #1 \exp_after:wN ;
22620     \int_value:w \__regex_int_eval:w #2 ;
22621     \prg_break_point:
22622   }
22623 \cs_new:Npn \__regex_query_range_loop:ww #1 ; #2 ;
22624   {
22625     \if_int_compare:w #1 < #2 \exp_stop_f:
```

```

22626     \else:
22627         \exp_after:wN \prg_break:
22628     \fi:
22629     \__regex_toks_use:w #1 \exp_stop_f:
22630     \exp_after:wN \__regex_query_range_loop:ww
22631         \int_value:w \__regex_int_eval:w #1 + 1 ; #2 ;
22632     }

```

(End definition for `\__regex_query_range:nn` and `\__regex_query_range:ww`.)

`\__regex_query_submatch:n` Find the start and end positions for a given submatch (of a given match).

```

22633 \cs_new:Npn \__regex_query_submatch:n #1
22634 {
22635     \__regex_query_range:nn
22636     { \__kernel_intarray_item:Nn \g__regex_submatch_begin_intarray {#1} }
22637     { \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray {#1} }
22638 }

```

(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 `<max pos>` and `<min pos>`. These two positions are found in the corresponding “submatch” arrays.

```

22639 \cs_new_protected:Npn \__regex_submatch_balance:n #1
22640 {
22641     \int_eval:n
22642     {
22643         \int_compare:nNnTF
22644         {
22645             \__kernel_intarray_item:Nn
22646                 \g__regex_submatch_end_intarray {#1}
22647         }
22648         = 0
22649     { 0 }
22650     {
22651         \__kernel_intarray_item:Nn \g__regex_balance_intarray
22652         {
22653             \__kernel_intarray_item:Nn
22654                 \g__regex_submatch_end_intarray {#1}
22655         }
22656     }
22657     -
22658     \int_compare:nNnTF
22659     {
22660         \__kernel_intarray_item:Nn
22661             \g__regex_submatch_begin_intarray {#1}
22662     }
22663     = 0
22664     { 0 }
22665     {
22666         \__kernel_intarray_item:Nn \g__regex_balance_intarray
22667         {

```

```

22668         \__kernel_intarray_item:Nn
22669             \g__regex_submatch_begin_intarray {#1}
22670     }
22671 }
22672 }
22673 }

(End definition for \__regex_submatch_balance:n)

```

### 38.6.3 Framework

\\_\_regex\_replacement:n  
\\_\_regex\_replacement\_aux: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 \l\_\_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.

```

22674 \__kernel_patch:nnNpn
22675   { \__regex_trace_push:nnN { regex } { 1 } \__regex_replacement:n }
22676   { \__regex_trace_pop:nnN { regex } { 1 } \__regex_replacement:n }
22677 \cs_new_protected:Npn \__regex_replacement:n #1
22678   {
22679     \group_begin:
22680       \tl_build_begin:N \l__regex_build_tl
22681       \int_zero:N \l__regex_balance_int
22682       \tl_clear:N \l__regex_balance_tl
22683       \__regex_escape_use:nnnn
22684       {
22685         \if_charcode:w \c_right_brace_str ##1
22686           \__regex_replacement_rbrace:N
22687         \else:
22688           \__regex_replacement_normal:n
22689         \fi:
22690           ##1
22691       }
22692       { \__regex_replacementEscaped:N ##1 }
22693       { \__regex_replacement_normal:n ##1 }
22694       {##1}
22695     \prg_do_nothing: \prg_do_nothing:
22696     \if_int_compare:w \l__regex_replacement_csnames_int > 0 \exp_stop_f:
22697       \__kernel_msg_error:nnx { kernel } { replacement-missing-rbrace }
22698       { \int_use:N \l__regex_replacement_csnames_int }
22699       \tl_build_put_right:Nx \l__regex_build_tl
22700       { \prg_replicate:nn \l__regex_replacement_csnames_int \cs_end: }
22701   \fi:
22702   \seq_if_empty:NF \l__regex_replacement_category_seq
22703   {
22704     \__kernel_msg_error:nnx { kernel } { replacement-missing-rparen }
22705     { \seq_count:N \l__regex_replacement_category_seq }
22706     \seq_clear:N \l__regex_replacement_category_seq
22707   }
22708   \cs_gset:Npx \__regex_replacement_balance_one_match:n ##1
22709   {

```

```

22710      + \int_use:N \l__regex_balance_int
22711      \l__regex_balance_tl
22712      - \__regex_submatch_balance:n {##1}
22713      }
22714      \tl_build_end:N \l__regex_build_tl
22715      \exp_args:NNo
22716      \group_end:
22717      \__regex_replacement_aux:n \l__regex_build_tl
22718      }
22719      \cs_new_protected:Npn \__regex_replacement_aux:n #1
22720      {
22721      \cs_set:Npn \__regex_replacement_do_one_match:n ##1
22722      {
22723      \__regex_query_range:nn
22724      {
22725      \__kernel_intarray_item:Nn
22726      \g__regex_submatch_prev_intarray {##1}
22727      }
22728      {
22729      \__kernel_intarray_item:Nn
22730      \g__regex_submatch_begin_intarray {##1}
22731      }
22732      #1
22733      }
22734      }

```

(End definition for `\__regex_replacement:n` and `\__regex_replacement_aux:n`.)

`\__regex_replacement_normal: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: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`.

```

22735      \cs_new_protected:Npn \__regex_replacement_normal:n #1
22736      {
22737      \tl_if_empty:NTF \l__regex_replacement_category_tl
22738      { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
22739      { % (
22740      \token_if_eq_charcode:NNTF #1 )
22741      {
22742      \seq_pop:NN \l__regex_replacement_category_seq
22743      \l__regex_replacement_category_tl
22744      }
22745      {
22746      \use:c
22747      {
22748      \__regex_replacement_c_
22749      \l__regex_replacement_category_tl :w
22750      }
22751      \__regex_replacement_normal:n {#1}
22752      }
22753      }
22754      }

```

(End definition for `\_\_regex\_replacement\_normal: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. We use `\token_to_str:N` to give spaces the right category code.

```
22755 \cs_new_protected:Npn \_\_regex_replacement_escaped:N #1
22756 {
22757     \cs_if_exist_use:cF { \_\_regex_replacement_#1:w }
22758     {
22759         \if_int_compare:w 1 < 1#1 \exp_stop_f:
22760             \_\_regex_replacement_put_submatch:n {#1}
22761         \else:
22762             \exp_args:No \_\_regex_replacement_normal:n
22763                 { \token_to_str:N #1 }
22764         \fi:
22765     }
22766 }
```

(End definition for `\_\_regex\_replacement\_escaped:N`.)

### 38.6.4 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 `\l_\_\_regex_balance_t1` there is an `x`-type expansion which is needed to get `##1` in correctly.

```
22767 \cs_new_protected:Npn \_\_regex_replacement_put_submatch:n #1
22768 {
22769     \if_int_compare:w #1 < \l_\_\_regex_capturing_group_int
22770         \tl_build_put_right:Nn \l_\_\_regex_build_t1
22771             { \_\_regex_query_submatch:n { \int_eval:n { #1 + ##1 } } }
22772         \if_int_compare:w \l_\_\_regex_replacement_csnames_int = 0 \exp_stop_f:
22773             \tl_put_right:Nn \l_\_\_regex_balance_t1
22774             {
22775                 + \_\_regex_submatch_balance:n
22776                     { \exp_not:N \int_eval:n { #1 + ##1 } }
22777             }
22778         \fi:
22779     \fi:
2280 }
```

(End definition for `\_\_regex_replacement_put_submatch:n`.)

`\_\_regex_replacement_g:w` 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.

```
22781 \cs_new_protected:Npn \_\_regex_replacement_g:w #1#2
22782 {
22783     \_\_regex_two_if_eq>NNNNTF
22784         #1 #2 \_\_regex_replacement_normal:n \c_left_brace_str
22785             { \l_\_\_regex_internal_a_int = \_\_regex_replacement_g_digits:NN }
```

```

22786      { \__regex_replacement_error:NNN g #1 #2 }
22787    }
22788 \cs_new:Npn \__regex_replacement_g_digits:NN #1#2
22789  {
22790    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
22791    {
22792      \if_int_compare:w 1 < 1#2 \exp_stop_f:
22793        #2
22794        \exp_after:wN \use_i:n
22795        \exp_after:wN \__regex_replacement_g_digits:NN
22796      \else:
22797        \exp_stop_f:
22798        \exp_after:wN \__regex_replacement_error:NNN
22799        \exp_after:wN g
22800      \fi:
22801    }
22802  {
22803    \exp_stop_f:
22804    \if_meaning:w \__regex_replacement_rbrace:N #1
22805      \exp_args:No \__regex_replacement_put_submatch:n
22806        { \int_use:N \l__regex_internal_a_int }
22807        \exp_after:wN \use_none:nn
22808      \else:
22809        \exp_after:wN \__regex_replacement_error:NNN
22810        \exp_after:wN g
22811      \fi:
22812    }
22813  #1 #2
22814 }

```

(End definition for `\__regex_replacement_g:w` and `\__regex_replacement_g_digits>NN`.)

### 38.6.5 Csnames in replacement

`\__regex_replacement_c:w` `\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 `\u`. Otherwise test whether the category is known; if it is not, complain.

```

22815 \cs_new_protected:Npn \__regex_replacement_c:w #1#2
22816  {
22817    \token_if_eq_meaning:NNTF #1 \__regex_replacement_normal:n
22818    {
22819      \exp_after:wN \token_if_eq_charcode:NNTF \c_left_brace_str #2
22820      { \__regex_replacement_cu_aux:Nw \__regex_replacement_exp_not:N }
22821      {
22822        \cs_if_exist:cTF { __regex_replacement_c:#2:w }
22823          { \__regex_replacement_cat:NNN #2 }
22824          { \__regex_replacement_error:NNN c #1#2 }
22825      }
22826    }
22827    { \__regex_replacement_error:NNN c #1#2 }
22828  }

```

(End definition for `\__regex_replacement_c:w`.)

\\_\\_regex\\_replacement\\_cu\\_aux:Nw Start a control sequence with \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 \c or \u. We use \tl\\_to\\_str:V rather than \tl\\_to\\_str:N to deal with integers and other registers.

```

22829 \cs_new_protected:Npn \_\_regex_replacement_cu_aux:Nw #1
22830 {
22831   \if_case:w \l_\_regex_replacement_csnames_int
22832     \tl_build_put_right:Nn \l_\_regex_build_tl
22833       { \exp_not:n { \exp_after:wN #1 \cs:w } }
22834   \else:
22835     \tl_build_put_right:Nn \l_\_regex_build_tl
22836       { \exp_not:n { \exp_after:wN \tl_to_str:V \cs:w } }
22837   \fi:
22838   \int_incr:N \l_\_regex_replacement_csnames_int
22839 }
```

(End definition for \\_\\_regex\\_replacement\\_cu\\_aux:Nw.)

\\_\\_regex\\_replacement\\_u:w Check that \u is followed by a left brace. If so, start a control sequence with \cs:w, which is then unpacked either with \exp\\_not:V or \tl\\_to\\_str:V depending on the current context.

```

22840 \cs_new_protected:Npn \_\_regex_replacement_u:w #1#2
22841 {
22842   \_\_regex_two_if_eq:NNNNTF
22843     #1 #2 \_\_regex_replacement_normal:n \c_left_brace_str
22844     { \_\_regex_replacement_cu_aux:Nw \exp_not:V }
22845     { \_\_regex_replacement_error:NNN u #1#2 }
22846 }
```

(End definition for \\_\\_regex\\_replacement\\_u:w.)

\\_\\_regex\\_replacement\\_rbrace:N Within a \c{\dots} or \u{\dots} construction, end the control sequence, and decrease the brace count. Otherwise, this is a raw right brace.

```

22847 \cs_new_protected:Npn \_\_regex_replacement_rbrace:N #1
22848 {
22849   \if_int_compare:w \l_\_regex_replacement_csnames_int > 0 \exp_stop_f:
22850     \tl_build_put_right:Nn \l_\_regex_build_tl { \cs_end: }
22851     \int_decr:N \l_\_regex_replacement_csnames_int
22852   \else:
22853     \_\_regex_replacement_normal:n {#1}
22854   \fi:
22855 }
```

(End definition for \\_\\_regex\\_replacement\\_rbrace:N.)

### 38.6.6 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{\dots} or \u{\dots}, and detect the case of a parenthesis. In that case, store the current category in a sequence and switch to a new one.

```

22856 \cs_new_protected:Npn \_\_regex_replacement_cat:NNN #1#2#3
22857 {
```

```

22858 \token_if_eq_meaning:NNTF \prg_do_nothing: #3
22859 { \__kernel_msg_error:nn { kernel } { replacement-catcode-end } }
22860 {
22861     \int_compare:nNnTF { \l__regex_replacement_csnames_int } > 0
22862     {
22863         \__kernel_msg_error:nnnn
22864             { kernel } { replacement-catcode-in-cs } {#1} {#3}
22865             #2 #3
22866     }
22867 {
22868     \__regex_two_if_eq:NNNNTF #2 #3 \__regex_replacement_normal:n (
22869     {
22870         \seq_push:NV \l__regex_replacement_category_seq
22871             \l__regex_replacement_category_tl
22872                 \tl_set:Nn \l__regex_replacement_category_tl {#1}
22873             }
22874 {
22875     \token_if_eq_meaning:NNT #2 \__regex_replacement_escaped:N
22876     {
22877         \__regex_char_if_alphanumeric:NTF #3
22878         {
22879             \__kernel_msg_error:nnnn
22880                 { kernel } { replacement-catcode-escaped }
22881                 {#1} {#3}
22882             }
22883             { }
22884         }
22885         \use:c { __regex_replacement_c_#1:w } #2 #3
22886     }
22887 }
22888 }
22889 }

```

(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.

```
22890 \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).

```

22891 \cs_new_protected:Npn \__regex_replacement_char:nNN #1#2#3
22892 {
22893     \tex_lccode:D 0 = '#3 \scan_stop:
22894     \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {#1} }
22895 }

```

(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.

```
22896  \char_set_catcode_active:N `^`@  
22897  \cs_new_protected:Npn \_\_regex\_replacement_c_A:w  
22898  { \_\_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.

```
22899  \char_set_catcode_group_begin:N `^`@  
22900  \cs_new_protected:Npn \_\_regex\_replacement_c_B:w  
22901  {  
22902  \if_int_compare:w \l_\_regex_replacement_csnames_int = 0 \exp_stop_f:  
22903  \int_incr:N \l_\_regex_balance_int  
22904  \fi:  
22905  \_\_regex\_replacement\_char:nNN  
22906  { \exp_not:n { \exp_after:wN `^`@ \if_false: } \fi: } }  
22907  }
```

(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.

```
22908  \cs_new_protected:Npn \_\_regex\_replacement_c_C:w #1#2  
22909  {  
22910  \tl_build_put_right:Nn \l_\_regex_build_t1  
22911  { \exp_not:N \exp_not:N \exp_not:c {#2} }  
22912  }
```

(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.

```
22913  \char_set_catcode_math_subscript:N `^`@  
22914  \cs_new_protected:Npn \_\_regex\_replacement_c_D:w  
22915  { \_\_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.

```
22916  \char_set_catcode_group_end:N `^`@  
22917  \cs_new_protected:Npn \_\_regex\_replacement_c_E:w  
22918  {  
22919  \if_int_compare:w \l_\_regex_replacement_csnames_int = 0 \exp_stop_f:  
22920  \int_decr:N \l_\_regex_balance_int  
22921  \fi:  
22922  \_\_regex\_replacement\_char:nNN  
22923  { \exp_not:n { \if_false: { \fi: `^`@ } }  
22924  }
```

(End definition for `\_regex_replacement_c_E:w`.)

`\_regex_replacement_c_L:w` Simply `\lowercase` a letter null byte to produce an arbitrary letter.

```
22925 \char_set_catcode_letter:N \^@  
22926 \cs_new_protected:Npn \_regex_replacement_c_L:w  
22927 { \_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.

```
22928 \char_set_catcode_math_toggle:N \^@  
22929 \cs_new_protected:Npn \_regex_replacement_c_M:w  
22930 { \_regex_replacement_char:nNN { ^@ } }
```

(End definition for `\_regex_replacement_c_M:w`.)

`\_regex_replacement_c_O:w` Lowercase an other null byte.

```
22931 \char_set_catcode_other:N \^@  
22932 \cs_new_protected:Npn \_regex_replacement_c_O:w  
22933 { \_regex_replacement_char:nNN { ^@ } }
```

(End definition for `\_regex_replacement_c_O: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.

```
22934 \char_set_catcode_parameter:N \^@  
22935 \cs_new_protected:Npn \_regex_replacement_c_P:w  
22936 {  
22937 \_regex_replacement_char:nNN  
22938 { \exp_not:n { \exp_not:n { ^@^@^@^@^@ } } }  
22939 }
```

(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.

```
22940 \cs_new_protected:Npn \_regex_replacement_c_S:w #1#2  
22941 {  
22942 \if_int_compare:w '#2 = 0 \exp_stop_f:  
22943 \_kernel_msg_error:nn { kernel } { replacement-null-space }  
22944 \fi:  
22945 \tex_lccode:D '\ = '#2 \scan_stop:  
22946 \tex_lowercase:D { \tl_build_put_right:Nn \l__regex_build_tl {~} }  
22947 }
```

(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.

```
22948 \char_set_catcode_alignment:N \^@  
22949 \cs_new_protected:Npn \_regex_replacement_c_T:w  
22950 { \_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 lowercases the math superscript `^{\circ}`.

```
22951   \char_set_catcode_math_superscript:N \^{\circ}
22952   \cs_new_protected:Npn \_\_regex_replacement_c_U:w
22953     { \_\_regex_replacement_char:nNN { ^{\circ} } }
```

(End definition for `\_\_regex_replacement_c_U:w`.)

Restore the catcode of the null byte.

```
22954 \group_end:
```

### 38.6.7 An error

`\_\_regex replacement_error:NNN` Simple error reporting by calling one of the messages `replacement-c`, `replacement-g`, or `replacement-u`.

```
22955 \cs_new_protected:Npn \_\_regex_replacement_error:NNN #1#2#3
22956   {
22957     \_\_kernel_msg_error:nnx { kernel } { replacement-#1 } {#3}
22958     #2 #3
22959   }
```

(End definition for `\_\_regex_replacement_error:NNN`.)

## 38.7 User functions

`\regex_new:N` Before being assigned a sensible value, a regex variable matches nothing.

```
22960 \cs_new_protected:Npn \regex_new:N #1
22961   { \cs_new_eq:NN #1 \c_regex_no_match_regex }
```

(End definition for `\regex_new:N`. This function is documented on page 215.)

`\l_tmpa_regex` The usual scratch space.

```
22962 \regex_new:N \l_tmpa_regex
22963 \regex_new:N \l_tmpb_regex
22964 \regex_new:N \g_tmpa_regex
22965 \regex_new:N \g_tmpb_regex
```

(End definition for `\l_tmpa_regex` and others. These variables are documented on page 217.)

`\regex_set:Nn` Compile, then store the result in the user variable with the appropriate assignment function.

```
22966 \cs_new_protected:Npn \regex_set:Nn #1#2
22967   {
22968     \_\_regex_compile:n {#2}
22969     \tl_set_eq:NN #1 \l__regex_internal_regex
22970   }
22971 \cs_new_protected:Npn \regex_gset:Nn #1#2
22972   {
22973     \_\_regex_compile:n {#2}
22974     \tl_gset_eq:NN #1 \l__regex_internal_regex
22975   }
22976 \cs_new_protected:Npn \regex_const:Nn #1#2
22977   {
```

```

22978     \_\_regex\_compile:n {#2}
22979     \tl\_const:Nx #1 { \exp\_not:o \l\_\_regex\_internal\_regex }
22980   }

```

(End definition for `\regex_set:Nn`, `\regex_gset:Nn`, and `\regex_const:Nn`. These functions are documented on page 215.)

**\regex\_show:N** User functions: the `n` variant requires compilation first. Then show the variable with `\regex_show:n` some appropriate text. The auxiliary is defined in a different section.

```

22981 \cs_new_protected:Npn \regex_show:n #1
22982   {
22983     \_\_regex_compile:n {#1}
22984     \_\_regex_show:N \l\_\_regex_internal_regex
22985     \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
22986     { \tl_to_str:n {#1} } { }
22987     { \l\_\_regex_internal_a_tl } { }
22988   }
22989 \cs_new_protected:Npn \regex_show:N #1
22990   {
22991     \_\_kernel_chk_defined:NT #1
22992     {
22993       \_\_regex_show:N #1
22994       \msg_show:nnxxxx { LaTeX / kernel } { show-regex }
22995       { } { \token_to_str:N #1 }
22996       { \l\_\_regex_internal_a_tl } { }
22997     }
22998   }

```

(End definition for `\regex_show:N` and `\regex_show:n`. These functions are documented on page 215.)

**\regex\_match:nnTF** Those conditionals are based on a common auxiliary defined later. Its first argument `\regex_match:NnTF` 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`.

```

22999 \prg_new_protected_conditional:Npnn \regex_match:nn #1#2 { T , F , TF }
23000   {
23001     \_\_regex_if_match:nn { \_\_regex_build:n {#1} } {#2}
23002     \_\_regex_return:
23003   }
23004 \prg_new_protected_conditional:Npnn \regex_match:Nn #1#2 { T , F , TF }
23005   {
23006     \_\_regex_if_match:nn { \_\_regex_build:N #1 } {#2}
23007     \_\_regex_return:
23008   }

```

(End definition for `\regex_match:nnTF` and `\regex_match:NnTF`. These functions are documented on page 215.)

**\regex\_count:nnN** Again, use an auxiliary whose first argument builds the NFA.

**\regex\_count:NnN**

```

23009 \cs_new_protected:Npn \regex_count:nnN #1
23010   { \_\_regex_count:nnN { \_\_regex_build:n {#1} } }
23011 \cs_new_protected:Npn \regex_count:NnN #1
23012   { \_\_regex_count:nnN { \_\_regex_build:N #1 } }

```

(End definition for `\regex_count:nnN` and `\regex_count:NnN`. These functions are documented on page 216.)

```

\regex_extract_once:nnN We define here 40 user functions, following a common pattern in terms of :nnN auxiliaries,
\regex_extract_once:nnNTF defined in the coming subsections. The auxiliary is handed \__regex_build:n or \__-
\regex_extract_once:NnN regex_build:N with the appropriate regex argument, then all other necessary arguments
\regex_extract_once:NnNTF (replacement text, token list, etc. The conditionals call \__regex_return: to return
\regex_extract_all:nnN either true or false once matching has been performed.
\regex_extract_all:nnNTF
\regex_extract_all:NnN
\regex_extract_all:NnNTF
\regex_replace_once:nnN
\regex_replace_once:nnNTF
\regex_replace_once:NnN
\regex_replace_once:NnNTF
\regex_replace_all:nnN
\regex_replace_all:nnNTF
\regex_replace_all:NnN
\regex_replace_all:NnNTF
\regex_split:nnN
\regex_split:nnNTF
\regex_split:NnN
\regex_split:NnNTF

```

```

23013 \cs_set_protected:Npn \__regex_tmp:w #1#2#3
23014 {
23015   \cs_new_protected:Npn #2 ##1 { #1 { \__regex_build:n {##1} } }
23016   \cs_new_protected:Npn #3 ##1 { #1 { \__regex_build:N ##1 } }
23017   \prg_new_protected_conditional:Npnn #2 ##1##2##3 { T , F , TF }
23018     { #1 { \__regex_build:n {##1} } ##2 } ##3 \__regex_return: }
23019   \prg_new_protected_conditional:Npnn #3 ##1##2##3 { T , F , TF }
23020     { #1 { \__regex_build:N ##1 } ##2 } ##3 \__regex_return: }
23021 }
23022 \__regex_tmp:w \__regex_extract_once:nnN
23023   \regex_extract_once:nnN \regex_extract_once:NnN
23024 \__regex_tmp:w \__regex_extract_all:nnN
23025   \regex_extract_all:nnN \regex_extract_all:NnN
23026 \__regex_tmp:w \__regex_replace_once:nnN
23027   \regex_replace_once:nnN \regex_replace_once:NnN
23028 \__regex_tmp:w \__regex_replace_all:nnN
23029   \regex_replace_all:nnN \regex_replace_all:NnN
23030 \__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 216.)

### 38.7.1 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.

```
23031 \int_new:N \l__regex_match_count_int
```

(End definition for \l\_\_regex\_match\_count\_int.)

--regex\_begin Those flags are raised to indicate extra begin-group or end-group tokens when extracting submatches.  
--regex\_end

```
23032 \flag_new:n { --regex_begin }
23033 \flag_new:n { --regex_end }
```

(End definition for --regex\_begin and --regex\_end.)

\l\_\_regex\_min\_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.

```
23034 \int_new:N \l__regex_min_submatch_int
23035 \int_new:N \l__regex_submatch_int
23036 \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 Hold the place where the match attempt begun and the end-points of each submatch.

```
23037 \intarray_new:Nn \g_regex_submatch_prev_intarray { 65536 }
23038 \intarray_new:Nn \g_regex_submatch_begin_intarray { 65536 }
23039 \intarray_new:Nn \g_regex_submatch_end_intarray { 65536 }
```

*(End definition for \g\_regex\_submatch\_prev\_intarray, \g\_regex\_submatch\_begin\_intarray, and \g\_regex\_submatch\_end\_intarray.)*

\\_\_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.

```
23040 \cs_new_protected:Npn \__regex_return:
23041 {
23042     \if_meaning:w \c_true_bool \g_regex_success_bool
23043         \prg_return_true:
23044     \else:
23045         \prg_return_false:
23046     \fi:
23047 }
```

*(End definition for \\_\_regex\_return:.)*

### 38.7.2 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.

```
23048 \cs_new_protected:Npn \__regex_if_match:nn #1#2
23049 {
23050     \group_begin:
23051         \__regex_disable_submatches:
23052         \__regex_single_match:
23053             #1
23054             \__regex_match:n {#2}
23055     \group_end:
23056 }
```

*(End definition for \\_\_regex\_if\_match: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.

```
23057 \cs_new_protected:Npn \__regex_count:nnN #1#2#3
23058 {
23059     \group_begin:
23060         \__regex_disable_submatches:
23061         \int_zero:N \l_regex_match_count_int
23062         \__regex_multi_match:n { \int_incr:N \l_regex_match_count_int }
23063             #1
23064             \__regex_match:n {#2}
23065             \exp_args:NNNo
23066     \group_end:
23067     \int_set:Nn #3 { \int_use:N \l_regex_match_count_int }
23068 }
```

(End definition for `\_regex_count:nnN`.)

### 38.7.3 Extracting submatches

`\_regex_extract_once:nnN`  
`\_regex_extract_all: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.

```
23069 \cs_new_protected:Npn \_regex_extract_once:nnN #1#2#3
23070 {
23071     \group_begin:
23072         \_regex_single_match:
23073             #1
23074             \_regex_match:n {#2}
23075             \_regex_extract:
23076                 \_regex_group_end_extract_seq:N #3
23077             }
23078 \cs_new_protected:Npn \_regex_extract_all:nnN #1#2#3
23079 {
23080     \group_begin:
23081         \_regex_multi_match:n { \_regex_extract: }
23082             #1
23083             \_regex_match:n {#2}
23084             \_regex_group_end_extract_seq:N #3
23085 }
```

(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.

```
23086 \cs_new_protected:Npn \_regex_split:nnN #1#2#3
23087 {
23088     \group_begin:
23089         \_regex_multi_match:n
23090             {
23091                 \if_int_compare:w
23092                     \l_regex_start_pos_int < \l_regex_success_pos_int
23093                     \_regex_extract:
23094                         \_kernel_intarray_gset:Nnn \g_regex_submatch_prev_intarray
23095                             { \l_regex_zeroth_submatch_int } { 0 }
23096                         \_kernel_intarray_gset:Nnn \g_regex_submatch_end_intarray
23097                             { \l_regex_zeroth_submatch_int }
23098                             {
23099                                 \_kernel_intarray_item:Nn \g_regex_submatch_begin_intarray
23100                                     { \l_regex_zeroth_submatch_int }
23101                             }
23102                         \_kernel_intarray_gset:Nnn \g_regex_submatch_begin_intarray
23103                             { \l_regex_zeroth_submatch_int }
23104                             { \l_regex_start_pos_int }
```

```

23105          \fi:
23106      }
23107  #1
23108  \_regex_match:n {#2}
23109  <assert>\assert_int:n { \l__regex_curr_pos_int = \l__regex_max_pos_int }
23110  \_kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
23111  { \l__regex_submatch_int } { 0 }
23112  \_kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
23113  { \l__regex_submatch_int }
23114  { \l__regex_max_pos_int }
23115  \_kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
23116  { \l__regex_submatch_int }
23117  { \l__regex_start_pos_int }
23118  \int_incr:N \l__regex_submatch_int
23119  \if_meaning:w \c_true_bool \l__regex_empty_success_bool
23120  \if_int_compare:w \l__regex_start_pos_int = \l__regex_max_pos_int
23121  \int_decr:N \l__regex_submatch_int
23122  \fi:
23123  \fi:
23124  \_regex_group_end_extract_seq:N #3
23125 }

```

(End definition for `\_regex_split:nnN`.)

`\_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 `\l__regex_internal_a_t1`. We detect unbalanced results using the two flags `--regex-begin` and `--regex-end`, raised whenever we see too many begin-group or end-group tokens in a submatch.

```

23126 \cs_new_protected:Npn \_regex_group_end_extract_seq:N #1
23127 {
23128     \flag_clear:n { --regex_begin }
23129     \flag_clear:n { --regex_end }
23130     \seq_set_from_function:NnN \l__regex_internal_seq
23131     {
23132         \int_step_function:nnN { \l__regex_min_submatch_int }
23133         { \l__regex_submatch_int - 1 }
23134     }
23135     \_regex_extract_seq_aux:n
23136     \int_compare:nNnf
23137     {
23138         \flag_height:n { --regex_begin } +
23139         \flag_height:n { --regex_end }
23140     }
23141     = 0
23142     {
23143         \_kernel_msg_error:nnxxx { kernel } { result-unbalanced }
23144         { splitting-or-extracting-submatches }
23145         { \flag_height:n { --regex_end } }
23146         { \flag_height:n { --regex_begin } }
23147     }
23148     \seq_set_map:NNn \l__regex_internal_seq \l__regex_internal_seq {##1}
23149     \exp_args:NNNo
23150     \group_end:

```

```

23151      \tl_set:Nn #1 { \l__regex_internal_seq }
23152  }

```

(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.

```

23153  \cs_new:Npn \_regex_extract_seq_aux:n #1
23154  {
23155      \exp_after:wN \_regex_extract_seq_aux:ww
23156      \int_value:w \_regex_submatch_balance:n {#1} ; #1;
23157  }
23158  \cs_new:Npn \_regex_extract_seq_aux:ww #1; #2;
23159  {
23160      \if_int_compare:w #1 < 0 \exp_stop_f:
23161          \flag_raise:n { __regex_end }
23162          \prg_replicate:nn {-#1} { \exp_not:n { \if_false: } \fi: } }
23163      \fi:
23164      \_regex_query_submatch:n {#2}
23165      \if_int_compare:w #1 > 0 \exp_stop_f:
23166          \flag_raise:n { __regex_begin }
23167          \prg_replicate:nn {#1} { \exp_not:n { \if_false: { \fi: } } }
23168      \fi:
23169  }

```

(End definition for `\_regex_extract_seq_aux:n` and `\_regex_extract_seq_aux:ww.`)

```
\_regex_extract:
\_regex_extract_b:wn
\_regex_extract_e:wn
```

Our task here is to extract from the property list `\l__regex_success_submatches_prop` the list of end-points of submatches, and store them in appropriate array entries, from `\l__regex_zeroth_submatch_int` upwards. We begin by emptying those entries. Then for each `<key>–<value>` pair in the property list update the appropriate entry. This is somewhat a hack: the `<key>` is a non-negative integer followed by `<` or `>`, which we use in a comparison to `-1`. At the end, store the information about the position at which the match attempt started, in `\g__regex_submatch_prev_intarray`.

```

23170  \cs_new_protected:Npn \_regex_extract:
23171  {
23172      \if_meaning:w \c_true_bool \g__regex_success_bool
23173          \int_set_eq:NN \l__regex_zeroth_submatch_int \l__regex_submatch_int
23174          \prg_replicate:nn \l__regex_capturing_group_int
23175          {
23176              \__kernel_intarray_gset:Nnn \g__regex_submatch_begin_intarray
23177                  { \l__regex_submatch_int } { 0 }
23178              \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray
23179                  { \l__regex_submatch_int } { 0 }
23180              \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
23181                  { \l__regex_submatch_int } { 0 }
23182              \int_incr:N \l__regex_submatch_int
23183          }
23184          \prop_map_inline:Nn \l__regex_success_submatches_prop
23185          {
23186              \if_int_compare:w ##1 - 1 \exp_stop_f:
23187                  \exp_after:wN \_regex_extract_e:wn \int_value:w
23188              \else:

```

```

23189           \exp_after:wN \__regex_extract_b:wn \int_value:w
23190           \fi:
23191           \__regex_int_eval:w \l__regex_zeroth_submatch_int + ##1 {##2}
23192       }
23193       \__kernel_intarray_gset:Nnn \g__regex_submatch_prev_intarray
23194           { \l__regex_zeroth_submatch_int } { \l__regex_start_pos_int }
23195   \fi:
23196 }
23197 \cs_new_protected:Npn \__regex_extract_b:wn #1 < #2
23198 {
23199     \__kernel_intarray_gset:Nnn
23200         \g__regex_submatch_begin_intarray {#1} {#2}
23201     }
23202 \cs_new_protected:Npn \__regex_extract_e:wn #1 > #2
23203     { \__kernel_intarray_gset:Nnn \g__regex_submatch_end_intarray {#1} {#2} }

(End definition for \__regex_extract:, \__regex_extract_b:wn, and \__regex_extract_e:wn.)

```

### 38.7.4 Replacement

\\_\_regex\_replace\_once: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.

```

23204 \cs_new_protected:Npn \__regex_replace_once:nnN #1#2#3
23205 {
23206     \group_begin:
23207         \__regex_single_match:
23208             #1
23209             \__regex_replacement:n {#2}
23210             \exp_args:No \__regex_match:n { #3 }
23211             \if_meaning:w \c_false_bool \g__regex_success_bool
23212                 \group_end:
23213             \else:
23214                 \__regex_extract:
23215                     \int_set:Nn \l__regex_balance_int
23216                         {
23217                             \__regex_replacement_balance_one_match:n
23218                                 { \l__regex_zeroth_submatch_int }
23219                         }
23220                     \tl_set:Nx \l__regex_internal_a_tl
23221                         {
23222                             \__regex_replacement_do_one_match:n
23223                                 { \l__regex_zeroth_submatch_int }
23224                             \__regex_query_range:nn
23225                                 {
23226                                     \__kernel_intarray_item:Nn \g__regex_submatch_end_intarray
23227                                         { \l__regex_zeroth_submatch_int }
23228                                 }
23229                                 { \l__regex_max_pos_int }

```

```

23230         }
23231     \__regex_group_end_replace:N #3
23232 \fi:
23233 }

```

(End definition for `\__regex_replace_once: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.

```

23234 \cs_new_protected:Npn \__regex_replace_all:nnN #1#2#
23235 {
23236     \group_begin:
23237     \__regex_multi_match:n { \__regex_extract: }
23238     #1
23239     \__regex_replacement:n {#2}
23240     \exp_args:No \__regex_match:n {#3}
23241     \int_set:Nn \l__regex_balance_int
23242     {
23243         0
23244         \int_step_function:nnN
23245             { \l__regex_min_submatch_int }
23246             \l__regex_capturing_group_int
23247             { \l__regex_submatch_int - 1 }
23248             \__regex_replacement_balance_one_match:n
23249         }
23250     \tl_set:Nx \l__regex_internal_a_tl
23251     {
23252         \int_step_function:nnN
23253             { \l__regex_min_submatch_int }
23254             \l__regex_capturing_group_int
23255             { \l__regex_submatch_int - 1 }
23256             \__regex_replacement_do_one_match:n
23257             \__regex_query_range:nn
23258                 \l__regex_start_pos_int \l__regex_max_pos_int
23259         }
23260     \__regex_group_end_replace:N #3
23261 }

```

(End definition for `\__regex_replace_all:nnN`.)

`\__regex_group_end_replace:N` If the brace balance is not 0, raise an error. Then set the user's variable `#1` to the x-expansion of `\l__regex_internal_a_tl`, adding the appropriate braces to produce a balanced result. And end the group.

```

23262 \cs_new_protected:Npn \__regex_group_end_replace:N #1
23263 {
23264     \if_int_compare:w \l__regex_balance_int = 0 \exp_stop_f:
23265     \else:
23266         \__kernel_msg_error:nnxxx { kernel } { result-unbalanced }

```

```

23267     { replacing }
23268     { \int_max:nn { - \l_regex_balance_int } { 0 } }
23269     { \int_max:nn { \l_regex_balance_int } { 0 } }
23270 \fi:
23271 \use:x
23272 {
23273   \group_end:
23274   \tl_set:Nn \exp_not:N #1
23275   {
23276     \if_int_compare:w \l_regex_balance_int < 0 \exp_stop_f:
23277       \prg_replicate:nn { - \l_regex_balance_int }
23278         { { \if_false: } \fi: }
23279     \fi:
23280     \l_regex_internal_a_tl
23281     \if_int_compare:w \l_regex_balance_int > 0 \exp_stop_f:
23282       \prg_replicate:nn { \l_regex_balance_int }
23283         { \if_false: { \fi: } }
23284     \fi:
23285   }
23286 }
23287 }
```

(End definition for `\_regex_group_end_replace:N`.)

### 38.7.5 Storing and showing compiled patterns

## 38.8 Messages

Messages for the preparsing phase.

```

23288 \use:x
23289 {
23290   \__kernel_msg_new:nnn { kernel } { trailing-backslash }
23291     { Trailing~escape~char~'\iow_char:N\\'~in~regex~or~replacement. }
23292   \__kernel_msg_new:nnn { kernel } { x-missing-rbrace }
23293   {
23294     Missing-brace~'\iow_char:N\\'~in~regex~
23295     '...'\iow_char:N\\x\iow_char:N{...##1'.
23296   }
23297   \__kernel_msg_new:nnn { kernel } { x-overflow }
23298   {
23299     Character~code~##1~too~large~in~
23300     '\iow_char:N\\x\iow_char:N{##2\iow_char:N}\\~regex.
23301   }
23302 }
```

Invalid quantifier.

```

23303 \__kernel_msg_new:nnnn { kernel } { invalid-quantifier }
23304   { Braced~quantifier~'#1'~may~not~be~followed~by~'#2'. }
23305   {
23306     The~character~'#2'~is~invalid~in~the~braced~quantifier~'#1'.~
23307     The~only~valid~quantifiers~are~'*',~'?','~'+',~'{<int>}','~
23308     '{<min>,'~and~'{<min>,<max>}','~optionally~followed~by~'?'.
23309 }
```

Messages for missing or extra closing brackets and parentheses, with some fancy singular/plural handling for the case of parentheses.

```

23310 \__kernel_msg_new:nnnn { kernel } { missing-rbrack }
23311   { Missing-right-bracket-inserted-in-regular-expression. }
23312   {
23313     LaTeX-was-given-a-regular-expression-where-a-character-class-
23314     was-started-with-'[,~-but-the-matching-']'~is~missing.
23315   }
23316 \__kernel_msg_new:nnnn { kernel } { missing-rparen }
23317   {
23318     Missing-right-
23319     \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } ~
23320     inserted-in-regular-expression.
23321   }
23322   {
23323     LaTeX-was-given-a-regular-expression-with-\int_eval:n {#1} ~
23324     more-left-parentheses-than-right-parentheses.
23325   }
23326 \__kernel_msg_new:nnnn { kernel } { extra-rparen }
23327   { Extra-right-parenthesis-ignored-in-regular-expression. }
23328   {
23329     LaTeX-came-across-a-closing-parenthesis-when-no-submatch-group-
23330     was-open.-The-parenthesis-will-be-ignored.
23331   }

```

Some escaped alphanumerics are not allowed everywhere.

```

23332 \__kernel_msg_new:nnnn { kernel } { bad-escape }
23333   {
23334     Invalid-escape-'\iow_char:N\#1'~
23335     \__regex_if_in_cs:TF { within-a-control-sequence. }
23336     {
23337       \__regex_if_in_class:TF
23338         { in-a-character-class. }
23339         { following-a-category-test. }
23340     }
23341   }
23342   {
23343     The-escape-sequence-'\iow_char:N\#1'~may-not-appear-
23344     \__regex_if_in_cs:TF
23345     {
23346       within-a-control-sequence-test-introduced-by-
23347       '\iow_char:N\c\iow_char:N\{'.
23348     }
23349     {
23350       \__regex_if_in_class:TF
23351         { within-a-character-class~ }
23352         { following-a-category-test-such-as-'\iow_char:N\cL'~ }
23353         because-it-does-not-match-exactly-one-character.
23354     }
23355   }

```

Range errors.

```

23356 \__kernel_msg_new:nnnn { kernel } { range-missing-end }
23357   { Invalid-end-point-for-range-'#1-#2'~in-character-class. }
23358   {

```

```

23359     The~end-point~'#2'~of~the~range~'#1-#2'~may~not~serve~as~an~
23360     end-point~for~a~range:~alphanumeric~characters~should~not~be~
23361     escaped,~and~non-alphanumeric~characters~should~be~escaped.
23362 }
23363 \__kernel_msg_new:nnnn { kernel } { range-backwards }
23364 { Range~'[#1-#2]'~out~of~order~in~character~class. }
23365 {
23366     In~ranges~of~characters~'[x-y]'~appearing~in~character~classes,~
23367     the~first~character~code~must~not~be~larger~than~the~second.~
23368     Here,~'#1'~has~character~code~\int_eval:n {'#1},~while~
23369     '#2'~has~character~code~\int_eval:n {'#2}.
23370 }

Errors related to \c and \u.

23371 \__kernel_msg_new:nnnn { kernel } { c-bad-mode }
23372 { Invalid~nested~'\iow_char:N\\c'~escape~in~regular~expression. }
23373 {
23374     The~'\iow_char:N\\c'~escape~cannot~be~used~within~
23375     a~control~sequence~test~'\iow_char:N\\c{...}'~
23376     nor~another~category~test.~
23377     To~combine~several~category~tests,~use~'\iow_char:N\\c[...]'.
23378 }
23379 \__kernel_msg_new:nnnn { kernel } { c-C-invalid }
23380 { '\iow_char:N\\cC'~should~be~followed~by~'.~or~'('~or~'#,~not~'#1'. }
23381 {
23382     The~'\iow_char:N\\cC'~construction~restricts~the~next~item~to~be~a~
23383     control~sequence~or~the~next~group~to~be~made~of~control~sequences.~
23384     It~only~makes~sense~to~follow~it~by~'.~or~by~a~group.
23385 }
23386 \__kernel_msg_new:nnnn { kernel } { c-lparen-in-class }
23387 { Catcode~test~cannot~apply~to~group~in~character~class }
23388 {
23389     Construction~such~as~'\iow_char:N\\cL(abc)'~are~not~allowed~inside~a~
23390     class~' [...] '~because~classes~do~not~match~multiple~characters~at~once.
23391 }
23392 \__kernel_msg_new:nnnn { kernel } { c-missing-rbrace }
23393 { Missing~right~brace~inserted~for~'\iow_char:N\\c'~escape. }
23394 {
23395     LaTeX~was~given~a~regular~expression~where~a~
23396     '\iow_char:N\\c\iow_char:N\\{...}'~construction~was~not~ended~
23397     with~a~closing~brace~'\iow_char:N\\}'.
23398 }
23399 \__kernel_msg_new:nnnn { kernel } { c-missing-rbrack }
23400 { Missing~right~bracket~inserted~for~'\iow_char:N\\c'~escape. }
23401 {
23402     A~construction~'\iow_char:N\\c[...]'~appears~in~a~
23403     regular~expression,~but~the~closing~']'~is~not~present.
23404 }
23405 \__kernel_msg_new:nnnn { kernel } { c-missing-category }
23406 { Invalid~character~'#1'~following~'\iow_char:N\\c'~escape. }
23407 {
23408     In~regular~expressions,~the~'\iow_char:N\\c'~escape~sequence~
23409     may~only~be~followed~by~a~left~brace,~a~left~bracket,~or~a~
23410     capital~letter~representing~a~character~category,~namely~
23411     one~of~'ABCDELMOPTSTU'.

```

```

23412     }
23413 \__kernel_msg_new:nnnn { kernel } { c-trailing }
23414   { Trailing~category~code~escape~'\iow_char:N\\c'... }
23415   {
23416     A~regular~expression~ends~with~'\iow_char:N\\c'~followed~
23417     by~a~letter.~It~will~be~ignored.
23418   }
23419 \__kernel_msg_new:nnnn { kernel } { u-missing-lbrace }
23420   { Missing~left~brace~following~'\iow_char:N\\u'~escape. }
23421   {
23422     The~'\iow_char:N\\u'~escape~sequence~must~be~followed~by~
23423     a~brace~group~with~the~name~of~the~variable~to~use.
23424   }
23425 \__kernel_msg_new:nnnn { kernel } { u-missing-rbrace }
23426   { Missing~right~brace~inserted~for~'\iow_char:N\\u'~escape. }
23427   {
23428     LaTeX~
23429     \str_if_eq:eeTF { } {#2}
23430       { reached~the~end~of~the~string~}
23431       { encountered~an~escaped~alphanumeric~character~'\iow_char:N\\#2'~}
23432     when~parsing~the~argument~of~an~
23433     '\iow_char:N\\u\iow_char:N\\...\\'~escape.
23434   }

```

Errors when encountering the POSIX syntax [:...:].

```

23435 \__kernel_msg_new:nnnn { kernel } { posix-unsupported }
23436   { POSIX~collating~element~'[#1 ~ #1]'~not~supported. }
23437   {
23438     The~' [.foo.]'~and~'[=bar=]'~syntaxes~have~a~special~meaning~
23439     in~POSIX~regular~expressions.~This~is~not~supported~by~LaTeX.~
23440     Maybe~you~forgot~to~escape~a~left~bracket~in~a~character~class?
23441   }
23442 \__kernel_msg_new:nnnn { kernel } { posix-unknown }
23443   { POSIX~class~'[:#1:]'~unknown. }
23444   {
23445     '[:#1:]'~is~not~among~the~known~POSIX~classes~
23446     '[:alnum:]',~'[[:alpha:]]',~'[[:ascii:]]',~'[[:blank:]]',~'
23447     '[[:cntrl:]]',~'[[:digit:]]',~'[[:graph:]]',~'[[:lower:]]',~'
23448     '[[:print:]]',~'[[:punct:]]',~'[[:space:]]',~'[[:upper:]]',~'
23449     '[[:word:]]',~and~'[[:xdigit:]]'.
23450   }
23451 \__kernel_msg_new:nnnn { kernel } { posix-missing-close }
23452   { Missing~closing~':]'~for~POSIX~class. }
23453   { The~POSIX~syntax~'#1'~must~be~followed~by~':]',~not~'#2'. }

```

In various cases, the result of a `\B{regex}` operation can leave us with an unbalanced token list, which we must re-balance by adding begin-group or end-group character tokens.

```

23454 \__kernel_msg_new:nnnn { kernel } { result-unbalanced }
23455   { Missing~brace~inserted~when~#1. }
23456   {
23457     LaTeX~was~asked~to~do~some~regular~expression~operation,~
23458     and~the~resulting~token~list~would~not~have~the~same~number~
23459     of~begin~group~and~end~group~tokens.~Braces~were~inserted:~
23460     #2-left,~#3-right.

```

```
23461 }
```

Error message for unknown options.

```
23462 \__kernel_msg_new:nnnn { kernel } { unknown-option }
23463   { Unknown~option~'#1'~for~regular~expressions. }
23464   {
23465     The~only~available~option~is~'case-insensitive',~toggled~by~
23466     '(?i)'~and~'(?-i)'.
23467   }
23468 \__kernel_msg_new:nnnn { kernel } { special-group-unknown }
23469   { Unknown~special~group~'#1'...~in~a~regular~expression. }
23470   {
23471     The~only~valid~constructions~starting~with~'(~'~are~
23472     '(:~...~)',~'(?|~...~)',~'(?i)',~and~'(?-i)'.
23473 }
```

Errors in the replacement text.

```
23474 \__kernel_msg_new:nnnn { kernel } { replacement-c }
23475   { Misused~'\iow_char:N\\c'~command~in~a~replacement~text. }
23476   {
23477     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
23478     can~be~followed~by~one~of~the~letters~'ABCDELMPSTU'~
23479     or~a~brace~group,~not~by~'#1'.
23480   }
23481 \__kernel_msg_new:nnnn { kernel } { replacement-u }
23482   { Misused~'\iow_char:N\\u'~command~in~a~replacement~text. }
23483   {
23484     In~a~replacement~text,~the~'\iow_char:N\\u'~escape~sequence~
23485     must~be~followed~by~a~brace~group~holding~the~name~of~the~
23486     variable~to~use.
23487   }
23488 \__kernel_msg_new:nnnn { kernel } { replacement-g }
23489   {
23490     Missing~brace~for~the~'\iow_char:N\\g'~construction~
23491     in~a~replacement~text.
23492   }
23493   {
23494     In~the~replacement~text~for~a~regular~expression~search,~
23495     submatches~are~represented~either~as~'\iow_char:N \\g{dd..d}',~
23496     or~'\\d',~where~'d'~are~single~digits.~Here,~a~brace~is~missing.
23497   }
23498 \__kernel_msg_new:nnnn { kernel } { replacement-catcode-end }
23499   {
23500     Missing~character~for~the~'\iow_char:N\\c<category><character>'~
23501     construction~in~a~replacement~text.
23502   }
23503   {
23504     In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
23505     can~be~followed~by~one~of~the~letters~'ABCDELMPSTU'~representing~
23506     the~character~category.~Then,~a~character~must~follow.~LaTeX~
23507     reached~the~end~of~the~replacement~when~looking~for~that.
23508   }
23509 \__kernel_msg_new:nnnn { kernel } { replacement-catcode-escaped }
23510   {
23511     Escaped~letter~or~digit~after~category~code~in~replacement~text.
```

```

23512     }
23513     {
23514         In~a~replacement~text,~the~'\iow_char:N\\c'~escape~sequence~
23515         can~be~followed~by~one~of~the~letters~'ABCDELMOPSTU'~representing~
23516         the~character~category.~Then,~a~character~must~follow,~not~
23517         '\iow_char:N\\#2'.
23518     }
23519 \__kernel_msg_new:nnnn { kernel } { replacement-catcode-in-cs }
23520     {
23521         Category~code~'\iow_char:N\\c#1#3'~ignored~inside~
23522         '\iow_char:N\\c\\{...\\}'~in~a~replacement~text.
23523     }
23524     {
23525         In~a~replacement~text,~the~category~codes~of~the~argument~of~
23526         '\iow_char:N\\c\\{...\\}'~are~ignored~when~building~the~control-
23527         sequence~name.
23528     }
23529 \__kernel_msg_new:nnnn { kernel } { replacement-null-space }
23530     { TeX~cannot~build~a~space~token~with~character~code~0. }
23531     {
23532         You~asked~for~a~character~token~with~category~space,~
23533         and~character~code~0,~for~instance~through~
23534         '\iow_char:N\\cS\\iow_char:N\\x00'.~
23535         This~specific~case~is~impossible~and~will~be~replaced~
23536         by~a~normal~space.
23537     }
23538 \__kernel_msg_new:nnnn { kernel } { replacement-missing-rbrace }
23539     { Missing~right~brace~inserted~in~replacement~text. }
23540     {
23541         There~\int_compare:nTF { #1 = 1 } { was } { were } ~#1~
23542             missing~right~\int_compare:nTF { #1 = 1 } { brace } { braces } .
23543     }
23544 \__kernel_msg_new:nnnn { kernel } { replacement-missing-rparen }
23545     { Missing~right~parenthesis~inserted~in~replacement~text. }
23546     {
23547         There~\int_compare:nTF { #1 = 1 } { was } { were } ~#1~
23548             missing~right~
23549             \int_compare:nTF { #1 = 1 } { parenthesis } { parentheses } .
23550     }

```

Used when showing a regex.

```

23551 \__kernel_msg_new:nnn { kernel } { show-regex }
23552     {
23553         >~Compiled~regex~
23554         \tl_if_empty:nTF {#1} { variable~#2 } { {#1} } :
23555         #3
23556     }

```

\\_\_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 laziness.

```

23557 \cs_new:Npn \__regex_msg_repeated:nnN #1#2#3
23558     {
23559         \str_if_eq:eeF { #1 #2 } { 1 0 }

```

```

23560    {
23561        , ~ repeated ~
23562        \int_case:nnF {#2}
23563        {
23564            { -1 } { #1~or~more~times,~\bool_if:NTF #3 { lazy } { greedy } }
23565            { 0 } { #1~times }
23566        }
23567        {
23568            between~#1~and~\int_eval:n {#1+#2}~times,~
23569            \bool_if:NTF #3 { lazy } { greedy }
23570        }
23571    }
23572 }
```

(End definition for `\_regex_msg_repeated:nnN`.)

### 38.9 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` 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.

```

23573 \_kernel_if_debug:TF
23574 {
23575     \cs_new_protected:Npn \_regex_trace_push:nnN #1#2#3
23576     { \_regex_trace:nnx {#1} {#2} { entering~\token_to_str:N #3 } }
23577     \cs_new_protected:Npn \_regex_trace_pop:nnN #1#2#3
23578     { \_regex_trace:nnx {#1} {#2} { leaving~\token_to_str:N #3 } }
23579     \cs_new_protected:Npn \_regex_trace:nnx #1#2#3
23580     {
23581         \int_compare:nNnF
23582         { \int_use:c { g__regex_trace_#1_int } } < {#2}
23583         { \iow_term:x { Trace:~#3 } }
23584     }
23585 }
23586 }
```

(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.

```
23587 \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).

```

23588 \_kernel_if_debug:TF
23589 {
23590     \cs_new_protected:Npn \_regex_trace_states:n #1
23591     {
23592         \int_step_inline:nnn
23593             \l__regex_min_state_int
23594             { \l__regex_max_state_int - 1 }
```

```

23595      {
23596          \__regex_trace:nnx { regex } {#1}
23597              { \iow_char:N \\toks ##1 = { \__regex_toks_use:w ##1 } }
23598      }
23599  }
23600 }
23601 { }

(End definition for \__regex_trace_states:n)

23602 </initex | package>

```

## 39 I3box implementation

```

23603 <*initex | package>
23604 <@=box>

```

### 39.1 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.

```

23605 \cs_new_eq:NN \__box_dim_eval:w \tex_dimexpr:D
23606 \__kernel_patch_args:nNpn
23607 {
23608     {
23609         \__kernel_chk_expr:nNnN {#1}
23610         \__box_dim_eval:w { } \__box_dim_eval:n
23611     }
23612 }
23613 \cs_new:Npn \__box_dim_eval:n #1
23614     { \__box_dim_eval:w #1 \scan_stop: }

(End definition for \__box_dim_eval:w and \__box_dim_eval:n.)

```

### 39.2 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.

```

\box_new:c
23615 <*package>
23616 \cs_new_protected:Npn \box_new:N #1
23617 {
23618     \__kernel_chk_if_free_cs:N #1
23619     \cs:w newbox \cs_end: #1
23620 }
23621 </package>
23622 \cs_generate_variant:Nn \box_new:N { c }

```

Clear a `<box>` register.

```

\box_clear:N
23623 \cs_new_protected:Npn \box_clear:N #1
23624     { \box_set_eq:NN #1 \c_empty_box }
\box_clear:c
23625 \cs_new_protected:Npn \box_gclear:N #1
23626     { \box_gset_eq:NN #1 \c_empty_box }
\box_gclear:c

```

```

23627 \cs_generate_variant:Nn \box_clear:N { c }
23628 \cs_generate_variant:Nn \box_gclear:N { c }

```

Clear or new.

```

\box_clear_new:N
23629 \cs_new_protected:Npn \box_clear_new:N #1
23630   { \box_if_exist:NTF #1 { \box_clear:N #1 } { \box_new:N #1 } }
\box_clear_new:c
23631 \cs_new_protected:Npn \box_gclear_new:N #1
23632   { \box_if_exist:NTF #1 { \box_gclear:N #1 } { \box_new:N #1 } }
\box_gclear_new:N
23633 \cs_generate_variant:Nn \box_clear_new:N { c }
\box_gclear_new:c
23634 \cs_generate_variant:Nn \box_gclear_new:N { c }

```

Assigning the contents of a box to be another box.

```

\box_set_eq:NN
23635 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\box_set_eq:cN
23636 \cs_new_protected:Npn \box_set_eq:NN #1#2
23637   { \tex_setbox:D #1 \tex_copy:D #2 }
\box_set_eq:Nc
23638 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\box_set_eq:cc
23639 \cs_new_protected:Npn \box_gset_eq:NN #1#2
23640   { \tex_global:D \tex_setbox:D #1 \tex_copy:D #2 }
\box_gset_eq:NN
23641 \cs_generate_variant:Nn \box_set_eq:NN { c , Nc , cc }
\box_gset_eq:cN
23642 \cs_generate_variant:Nn \box_gset_eq:NN { c , Nc , cc }
\box_gset_eq:Nc
\box_gset_eq:cc

```

Assigning the contents of a box to be another box, then drops the original box.

```

\box_set_eq_drop:NN
23643 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\box_set_eq_drop:cN
23644 \cs_new_protected:Npn \box_set_eq_drop:NN #1#2
23645   { \tex_setbox:D #1 \tex_box:D #2 }
\box_set_eq_drop:Nc
23646 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
\box_set_eq_drop:cc
23647 \cs_new_protected:Npn \box_gset_eq_drop:NN #1#2
23648   { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
\box_gset_eq_drop:NN
23649 \cs_generate_variant:Nn \box_set_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:cN
23650 \cs_generate_variant:Nn \box_gset_eq_drop:NN { c , Nc , cc }
\box_gset_eq_drop:Nc
\box_gset_eq_drop:cc

```

Copies of the `cs` functions defined in `l3basics`.

```

\box_if_exist_p:N
23651 \prg_new_eq_conditional:NNn \box_if_exist:N \cs_if_exist:N
23652   { TF , T , F , p }
\box_if_exist_p:c
23653 \prg_new_eq_conditional:NNn \box_if_exist:c \cs_if_exist:c
23654   { TF , T , F , p }
\box_if_exist:cTF

```

### 39.3 Measuring and setting box dimensions

Accessing the height, depth, and width of a `(box)` register.

```

\box_ht:N
23655 \cs_new_eq:NN \box_ht:N \tex_ht:D
\box_ht:c
23656 \cs_new_eq:NN \box_dp:N \tex_dp:D
\box_dp:N
23657 \cs_new_eq:NN \box_wd:N \tex_wd:D
23658 \cs_generate_variant:Nn \box_ht:N { c }
\box_dp:c
23659 \cs_generate_variant:Nn \box_dp:N { c }
\box_wd:N
23660 \cs_generate_variant:Nn \box_wd:N { c }
\box_wd:c

```

Setting the size whilst respecting local scope requires copying; the same issue does not come up when working globally. When debugging, the dimension expression `#2` is surrounded by parentheses to catch early termination.

```

\box_set_ht:Nn
23661 \cs_new_protected:Npn \box_set_dp:Nn #1#2
23662   {
\box_set_dp:Nn
23663     \tex_setbox:D #1 = \tex_copy:D #1
\box_set_dp:cn
\box_set_wd:Nn
\box_set_wd:cn

```

```

23664      \box_dp:N #1 \__box_dim_eval:n {#2}
23665      }
23666 \cs_generate_variant:Nn \box_set_dp:Nn { c }
23667 \cs_new_protected:Npn \box_gset_dp:Nn #1#2
23668  { \box_dp:N #1 \__box_dim_eval:n {#2} }
23669 \cs_generate_variant:Nn \box_gset_dp:Nn { c }
23670 \cs_new_protected:Npn \box_set_ht:Nn #1#2
23671  {
23672      \tex_setbox:D #1 = \tex_copy:D #1
23673      \box_ht:N #1 \__box_dim_eval:n {#2}
23674  }
23675 \cs_generate_variant:Nn \box_set_ht:Nn { c }
23676 \cs_new_protected:Npn \box_gset_ht:Nn #1#2
23677  { \box_ht:N #1 \__box_dim_eval:n {#2} }
23678 \cs_generate_variant:Nn \box_gset_ht:Nn { c }
23679 \cs_new_protected:Npn \box_set_wd:Nn #1#2
23680  {
23681      \tex_setbox:D #1 = \tex_copy:D #1
23682      \box_wd:N #1 \__box_dim_eval:n {#2}
23683  }
23684 \cs_generate_variant:Nn \box_set_wd:Nn { c }
23685 \cs_new_protected:Npn \box_gset_wd:Nn #1#2
23686  { \box_wd:N #1 \__box_dim_eval:n {#2} }
23687 \cs_generate_variant:Nn \box_gset_wd:Nn { c }

```

## 39.4 Using boxes

Using a  $\langle box \rangle$ . These are just T<sub>E</sub>X primitives with meaningful names.

```

23688 \cs_new_eq:NN \box_use_drop:N \tex_box:D
\box_use_drop:N
23689 \cs_new_eq:NN \box_use:N \tex_copy:D
\box_use_drop:c
23690 \cs_generate_variant:Nn \box_use_drop:N { c }
\box_use:N
23691 \cs_generate_variant:Nn \box_use:N { c }
\box_use:c

```

Move box material in different directions. When debugging, the dimension expression #1 is surrounded by parentheses to catch early termination.

```

23692 \cs_new_protected:Npn \box_move_left:nn #1#2
23693  { \tex_moveleft:D \__box_dim_eval:n {#1} #2 }
23694 \cs_new_protected:Npn \box_move_right:nn #1#2
23695  { \tex_moveright:D \__box_dim_eval:n {#1} #2 }
23696 \cs_new_protected:Npn \box_move_up:nn #1#2
23697  { \tex_raise:D \__box_dim_eval:n {#1} #2 }
23698 \cs_new_protected:Npn \box_move_down:nn #1#2
23699  { \tex_lower:D \__box_dim_eval:n {#1} #2 }

```

## 39.5 Box conditionals

The primitives for testing if a  $\langle box \rangle$  is empty/void or which type of box it is.

```

23700 \cs_new_eq:NN \if_hbox:N      \tex_ifhbox:D
\if_hbox:N
23701 \cs_new_eq:NN \if_vbox:N      \tex_ifvbox:D
\if_vbox:N
23702 \cs_new_eq:NN \if_box_empty:N \tex_ifvoid:D
\if_box_empty:N

```

```
23703 \prg_new_conditional:Npnn \box_if_horizontal:N #1 { p , T , F , TF }
```

```

\box_if_horizontal_p:N
\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

```

```

23704   { \if_hbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
23705   \prg_new_conditional:Npnn \box_if_vertical:N #1 { p , T , F , TF }
23706   { \if_vbox:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
23707   \prg_generate_conditional_variant:Nnn \box_if_horizontal:N
23708   { c } { p , T , F , TF }
23709 \prg_generate_conditional_variant:Nnn \box_if_vertical:N
23710   { c } { p , T , F , TF }

```

Testing if a  $\langle box \rangle$  is empty/void.

```

\box_if_empty_p:N 23711 \prg_new_conditional:Npnn \box_if_empty:N #1 { p , T , F , TF }
\box_if_empty_p:c 23712 { \if_box_empty:N #1 \prg_return_true: \else: \prg_return_false: \fi: }
\box_if_empty:NTF 23713 \prg_generate_conditional_variant:Nnn \box_if_empty:N
\box_if_empty:cTF 23714 { c } { p , T , F , TF }

(End definition for \box_new:N and others. These functions are documented on page 221.)

```

## 39.6 The last box inserted

`\box_set_to_last:N` Set a box to the previous box.  
`\box_set_to_last:c` 23715  $\_\_kernel\_patch:nnNNpn$  {  $\_\_kernel\_chk\_var\_local:N$  #1 } { }  
`\box_gset_to_last:N` 23716  $\cs_{new\_protected}:Npn$   $\box_{set\_to\_last}:N$  #1  
`\box_gset_to_last:c` 23717 {  $\tex_{setbox}:D$  #1  $\tex_{lastbox}:D$  }  
23718  $\_\_kernel\_patch:nnNNpn$  {  $\_\_kernel\_chk\_var\_global:N$  #1 } { }  
23719  $\cs_{new\_protected}:Npn$   $\box_{gset\_to\_last}:N$  #1  
23720 {  $\tex_{global}:D$   $\tex_{setbox}:D$  #1  $\tex_{lastbox}:D$  }  
23721  $\cs_{generate\_variant}:Nn$   $\box_{set\_to\_last}:N$  { c }  
23722  $\cs_{generate\_variant}:Nn$   $\box_{gset\_to\_last}:N$  { c }

(End definition for `\box_set_to_last:N` and `\box_gset_to_last:N`. These functions are documented on page 223.)

## 39.7 Constant boxes

`\c_empty_box` A box we never use.  
23723  $\box_{new}:N$  `\c_empty_box`

(End definition for `\c_empty_box`. This variable is documented on page 223.)

## 39.8 Scratch boxes

`\l_tmpa_box` Scratch boxes.  
`\l_tmpb_box` 23724  $\box_{new}:N$  `\l_tmpa_box`  
`\g_tmpa_box` 23725  $\box_{new}:N$  `\l_tmpb_box`  
`\g_tmpb_box` 23726  $\box_{new}:N$  `\g_tmpa_box`  
23727  $\box_{new}:N$  `\g_tmpb_box`

(End definition for `\l_tmpa_box` and others. These variables are documented on page 223.)

## 39.9 Viewing box contents

$\text{\TeX}$ 's  $\text{\showbox}$  is not really that helpful in many cases, and it is also inconsistent with other  $\text{L}\text{\TeX}3$   $\text{show}$  functions as it does not actually shows material in the terminal. So we provide a richer set of functionality.

**$\text{\box\_show:N}$**  Essentially a wrapper around the internal function, but evaluating the breadth and depth arguments now outside the group.

```

\box_show:Nn
\box_show:cnn
\box_show:cnn
23728 \cs_new_protected:Npn \box_show:N #1
23729   { \box_show:Nnn #1 \c_max_int \c_max_int }
23730 \cs_generate_variant:Nn \box_show:N { c }
23731 \cs_new_protected:Npn \box_show:Nnn #1#2#3
23732   { \__box_show:NNff 1 #1 { \int_eval:n {#2} } { \int_eval:n {#3} } }
23733 \cs_generate_variant:Nn \box_show:Nnn { c }

```

(End definition for  $\text{\box\_show:N}$  and  $\text{\box\_show:Nnn}$ . These functions are documented on page 224.)

**$\text{\box_log:N}$**  Getting  $\text{\TeX}$  to write to the log without interruption the run is done by altering the interaction mode. For that, the  $\varepsilon$ - $\text{\TeX}$  extensions are needed.

```

\box_log:Nn
\box_log:cnn
\__box_log:nNnn
23734 \cs_new_protected:Npn \box_log:N #1
23735   { \box_log:Nnn #1 \c_max_int \c_max_int }
23736 \cs_generate_variant:Nn \box_log:N { c }
23737 \cs_new_protected:Npn \box_log:Nnn
23738   { \exp_args:No \__box_log:nNnn { \tex_the:D \tex_interactionmode:D } }
23739 \cs_new_protected:Npn \__box_log:nNnn #1#2#3#4
23740   {
23741     \int_set:Nn \tex_interactionmode:D { 0 }
23742     \__box_show:NNff 0 #2 { \int_eval:n {#3} } { \int_eval:n {#4} }
23743     \int_set:Nn \tex_interactionmode:D {#1}
23744   }
23745 \cs_generate_variant:Nn \box_log:Nnn { c }

```

(End definition for  $\text{\box\_log:N}$ ,  $\text{\box\_log:Nnn}$ , and  $\text{\__box\_log:nNnn}$ . These functions are documented on page 224.)

**$\text{\__box_show:NNnn}$**   **$\text{\__box_show:NNff}$**  The internal auxiliary to actually do the output uses a group to deal with breadth and depth values. The  $\text{\use:n}$  here gives better output appearance. Setting  $\text{\tracingonline}$  and  $\text{\errorcontextlines}$  is used to control what appears in the terminal.

```

\__box_show:NNnn #1#2#3#4
{
  \box_if_exist:NTF #2
  {
    \group_begin:
    \int_set:Nn \tex_showboxbreadth:D {#3}
    \int_set:Nn \tex_showboxdepth:D {#4}
    \int_set:Nn \tex_tracingonline:D {#1}
    \int_set:Nn \tex_errorcontextlines:D { -1 }
    \tex_showbox:D \use:n {#2}
    \group_end:
  }
  {
    \__kernel_msg_error:nny { kernel } { variable-not-defined }
    { \token_to_str:N #2 }
  }

```

```

23762     }
23763 \cs_generate_variant:Nn \__box_show:NNnn { NNff }
```

(End definition for `\__box_show:NNnn`.)

## 39.10 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.

```

23764 \cs_new_protected:Npn \hbox:n #1
23765   { \tex_hbox:D \scan_stop: { \color_group_begin: #1 \color_group_end: } }
```

(End definition for `\hbox:n`. This function is documented on page 224.)

```

\hbox_set:Nn
\hbox_set:cn
\hbox_gset:Nn
\hbox_gset:cn
23766 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23767 \cs_new_protected:Npn \hbox_set:Nn #1#2
23768   {
23769     \tex_setbox:D #1 \tex_hbox:D
23770       { \color_group_begin: #2 \color_group_end: }
23771   }
23772 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23773 \cs_new_protected:Npn \hbox_gset:Nn #1#2
23774   {
23775     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
23776       { \color_group_begin: #2 \color_group_end: }
23777   }
23778 \cs_generate_variant:Nn \hbox_set:Nn { c }
23779 \cs_generate_variant:Nn \hbox_gset:Nn { c }
```

(End definition for `\hbox_set:Nn` and `\hbox_gset:Nn`. These functions are documented on page 225.)

**\hbox\_set\_to\_wd:Nnn**  
**\hbox\_set\_to\_wd:cnn**  
Storing material in a horizontal box with a specified width. Again, put the dimension expression in parentheses when debugging.

```

\hbox_gset_to_wd:Nnn
\hbox_gset_to_wd:cnn
23780 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23781 \cs_new_protected:Npn \hbox_set_to_wd:Nnn #1#2#3
23782   {
23783     \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
23784       { \color_group_begin: #3 \color_group_end: }
23785   }
23786 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23787 \cs_new_protected:Npn \hbox_gset_to_wd:Nnn #1#2#3
23788   {
23789     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
23790       { \color_group_begin: #3 \color_group_end: }
23791   }
23792 \cs_generate_variant:Nn \hbox_set_to_wd:Nnn { c }
23793 \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 225.)

```

\hbox_set:Nw Storing material in a horizontal box. This type is useful in environment definitions.
\hbox_set:cw
23794 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\hbox_gset:Nw
23795 \cs_new_protected:Npn \hbox_set:Nw #1
\hbox_gset:cw
23796 {
23797     \tex_setbox:D #1 \tex_hbox:D
23798         \c_group_begin_token
23799             \color_group_begin:
23800         }
23801 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23802 \cs_new_protected:Npn \hbox_gset:Nw #1
23803 {
23804     \tex_global:D \tex_setbox:D #1 \tex_hbox:D
23805         \c_group_begin_token
23806             \color_group_begin:
23807         }
23808 \cs_generate_variant:Nn \hbox_set:Nw { c }
23809 \cs_generate_variant:Nn \hbox_gset:Nw { c }
23810 \cs_new_protected:Npn \hbox_set_end:
23811 {
23812     \color_group_end:
23813     \c_group_end_token
23814 }
23815 \cs_new_eq:NN \hbox_gset_end: \hbox_set_end:

```

(End definition for `\hbox_set:Nw` and others. These functions are documented on page [225](#).)

`\hbox_set_to_wd:Nnw` Combining the above ideas.

```

\hbox_set_to_wd:cw
23816 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
\hbox_gset_to_wd:Nnw
23817 \cs_new_protected:Npn \hbox_set_to_wd:Nnw #1#2
\hbox_gset_to_wd:cw
23818 {
23819     \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
23820         \c_group_begin_token
23821             \color_group_begin:
23822         }
23823 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23824 \cs_new_protected:Npn \hbox_gset_to_wd:Nnw #1#2
23825 {
23826     \tex_global:D \tex_setbox:D #1 \tex_hbox:D to \__box_dim_eval:n {#2}
23827         \c_group_begin_token
23828             \color_group_begin:
23829         }
23830 \cs_generate_variant:Nn \hbox_set_to_wd:Nnw { c }
23831 \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 [225](#).)

`\hbox_to_wd:nn` Put a horizontal box directly into the input stream.

```

\hbox_to_zero:n
23832 \cs_new_protected:Npn \hbox_to_wd:nn #1#2
23833 {
23834     \tex_hbox:D to \__box_dim_eval:n {#1}
23835         { \color_group_begin: #2 \color_group_end: }
23836     }
23837 \cs_new_protected:Npn \hbox_to_zero:n #1

```

```

23838     {
23839         \tex_hbox:D to \c_zero_dim
23840             { \color_group_begin: #1 \color_group_end: }
23841     }

```

(End definition for `\hbox_to_wd:nn` and `\hbox_to_zero:n`. These functions are documented on page 224.)

**\hbox\_overlap\_left: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.

```

23842 \cs_new_protected:Npn \hbox_overlap_left:n #1
23843     { \hbox_to_zero:n { \tex_hss:D #1 } }
23844 \cs_new_protected:Npn \hbox_overlap_right:n #1
23845     { \hbox_to_zero:n { #1 \tex_hss:D } }

```

(End definition for `\hbox_overlap_left:n` and `\hbox_overlap_right:n`. These functions are documented on page 225.)

**\hbox\_unpack:N** Unpacking a box and if requested also clear it.

```

\hbox_unpack:c 23846 \cs_new_eq:NN \hbox_unpack:N \tex_unhcopy:D
\hbox_unpack_drop:N 23847 \cs_new_eq:NN \hbox_unpack_drop:N \tex_unhbox:D
\hbox_unpack_drop:c 23848 \cs_generate_variant:Nn \hbox_unpack:N { c }
23849 \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 225.)

### 39.11 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.

```

23850 \cs_new_protected:Npn \vbox:n #1
23851     { \tex_vbox:D { \color_group_begin: #1 \par \color_group_end: } }
23852 \cs_new_protected:Npn \vbox_top:n #1
23853     { \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 226.)

**\vbox\_to\_ht:nn** Put a vertical box directly into the input stream.

```

\vbox_to_zero:n 23854 \cs_new_protected:Npn \vbox_to_ht:nn #1#2
\vbox_to_ht:nn 23855     {
23856         \tex_vbox:D to \__box_dim_eval:n {#1}
23857             { \color_group_begin: #2 \par \color_group_end: }
23858     }
23859 \cs_new_protected:Npn \vbox_to_zero:n #1
23860     {
23861         \tex_vbox:D to \c_zero_dim
23862             { \color_group_begin: #1 \par \color_group_end: }
23863     }

```

(End definition for `\vbox_to_ht:nn` and others. These functions are documented on page 226.)

**\vbox\_set:Nn** Storing material in a vertical box with a natural height.

```
23864 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23865 \cs_new_protected:Npn \vbox_set:Nn #1#2
23866 {
23867     \tex_setbox:D #1 \tex_vbox:D
23868     { \color_group_begin: #2 \par \color_group_end: }
23869 }
23870 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23871 \cs_new_protected:Npn \vbox_gset:Nn #1#2
23872 {
23873     \tex_global:D \tex_setbox:D #1 \tex_vbox:D
23874     { \color_group_begin: #2 \par \color_group_end: }
23875 }
23876 \cs_generate_variant:Nn \vbox_set:Nn { c }
23877 \cs_generate_variant:Nn \vbox_gset:Nn { c }
```

(End definition for `\vbox_set:Nn` and `\vbox_gset:Nn`. These functions are documented on page 226.)

**\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.

```
23878 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23879 \cs_new_protected:Npn \vbox_set_top:Nn #1#2
23880 {
23881     \tex_setbox:D #1 \tex_vtop:D
23882     { \color_group_begin: #2 \par \color_group_end: }
23883 }
23884 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23885 \cs_new_protected:Npn \vbox_gset_top:Nn #1#2
23886 {
23887     \tex_global:D \tex_setbox:D #1 \tex_vtop:D
23888     { \color_group_begin: #2 \par \color_group_end: }
23889 }
23890 \cs_generate_variant:Nn \vbox_set_top:Nn { c }
23891 \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 226.)

**\vbox\_set\_to\_ht:Nnn** Storing material in a vertical box with a specified height.

```
23892 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23893 \cs_new_protected:Npn \vbox_set_to_ht:Nnn #1#2#3
23894 {
23895     \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
23896     { \color_group_begin: #3 \par \color_group_end: }
23897 }
23898 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23899 \cs_new_protected:Npn \vbox_gset_to_ht:Nnn #1#2#3
23900 {
23901     \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
23902     { \color_group_begin: #3 \par \color_group_end: }
23903 }
23904 \cs_generate_variant:Nn \vbox_set_to_ht:Nnn { c }
23905 \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 226.)

`\vbox_set:Nw` Storing material in a vertical box. This type is useful in environment definitions.

```
23906 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23907 \cs_new_protected:Npn \vbox_set:Nw #1
23908 {
23909     \tex_setbox:D #1 \tex_vbox:D
23910     \c_group_begin_token
23911     \color_group_begin:
23912 }
23913 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23914 \cs_new_protected:Npn \vbox_gset:Nw #1
23915 {
23916     \tex_global:D \tex_setbox:D #1 \tex_vbox:D
23917     \c_group_begin_token
23918     \color_group_begin:
23919 }
23920 \cs_generate_variant:Nn \vbox_set:Nw { c }
23921 \cs_generate_variant:Nn \vbox_gset:Nw { c }
23922 \cs_new_protected:Npn \vbox_set_end:
23923 {
23924     \par
23925     \color_group_end:
23926     \c_group_end_token
23927 }
23928 \cs_new_eq:NN \vbox_gset_end: \vbox_set_end:
```

(End definition for `\vbox_set:Nw` and others. These functions are documented on page 226.)

`\vbox_set_to_ht:Nnw` A combination of the above ideas.

```
23929 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23930 \cs_new_protected:Npn \vbox_set_to_ht:Nnw #1#2
23931 {
23932     \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
23933     \c_group_begin_token
23934     \color_group_begin:
23935 }
23936 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23937 \cs_new_protected:Npn \vbox_gset_to_ht:Nnw #1#2
23938 {
23939     \tex_global:D \tex_setbox:D #1 \tex_vbox:D to \__box_dim_eval:n {#2}
23940     \c_group_begin_token
23941     \color_group_begin:
23942 }
23943 \cs_generate_variant:Nn \vbox_set_to_ht:Nnw { c }
23944 \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 227.)

`\vbox_unpack:N` Unpacking a box and if requested also clear it.

```
23945 \cs_new_eq:NN \vbox_unpack:N \tex_unvcopy:D
23946 \cs_new_eq:NN \vbox_unpack_drop:N \tex_unvbox:D
23947 \cs_generate_variant:Nn \vbox_unpack:N { c }
23948 \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 227.)

`\vbox_set_split_to_ht:NNn` Splitting a vertical box in two.

```
23949 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
23950 \cs_new_protected:Npn \vbox_set_split_to_ht:NNn #1#2#3
23951 { \tex_setbox:D #1 \tex_vsplit:D #2 to \__box_dim_eval:n {#3} }
23952 \cs_generate_variant:Nn \vbox_set_split_to_ht:NNn { c , Nc , cc }
23953 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
23954 \cs_new_protected:Npn \vbox_gset_split_to_ht:NNn #1#2#3
23955 {
23956     \tex_global:D \tex_setbox:D #1
23957         \tex_vsplit:D #2 to \__box_dim_eval:n {#3}
23958 }
23959 \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 227.)

### 39.12 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.

```
23960 \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.

```
23961 \fp_new:N \l_box_cos_fp
23962 \fp_new:N \l_box_sin_fp
```

(End definition for `\l_box_cos_fp` and `\l_box_sin_fp`.)

`\l_box_top_dim`

`\l_box_bottom_dim`

`\l_box_left_dim`

`\l_box_right_dim`

These are the positions of the four edges of a box before manipulation.

```
23963 \dim_new:N \l_box_top_dim
23964 \dim_new:N \l_box_bottom_dim
23965 \dim_new:N \l_box_left_dim
23966 \dim_new:N \l_box_right_dim
```

(End definition for `\l_box_top_dim` and others.)

`\l_box_top_new_dim`

`\l_box_bottom_new_dim`

`\l_box_left_new_dim`

`\l_box_right_new_dim`

These are the positions of the four edges of a box after manipulation.

```
23967 \dim_new:N \l_box_top_new_dim
23968 \dim_new:N \l_box_bottom_new_dim
23969 \dim_new:N \l_box_left_new_dim
23970 \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.

```
23971 \box_new:N \l_box_internal_box
```

(End definition for `\l_box_internal_box`.)

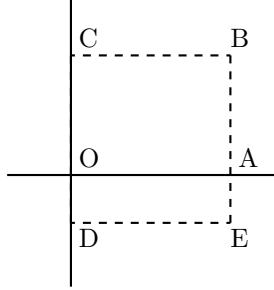


Figure 1: Co-ordinates of a box prior to rotation.

```

\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
\box_grotate:cn
\_box_rotate:NnN
  \_box_rotate:N
\_box_rotate_xdir:nnN
  \_box_rotate_ydir:nnN
\_box_rotate_quadrant_one:
\_box_rotate_quadrant_two:
  \_box_rotate_quadrant_three:
\_box_rotate_quadrant_four:

```

23972 \cs\_new\_protected:Npn \box\_rotate:Nn #1#2
23973 { \\_box\_rotate:NnN #1 {#2} \hbox\_set:Nn }
23974 \cs\_generate\_variant:Nn \box\_rotate:Nn { c }
23975 \cs\_new\_protected:Npn \box\_grotate:Nn #1#2
23976 { \\_box\_rotate:NnN #1 {#2} \hbox\_gset:Nn }
23977 \cs\_generate\_variant:Nn \box\_grotate:Nn { c }
23978 \cs\_new\_protected:Npn \\_box\_rotate:NnN #1#2#3
23979 {
23980 #3 #1
23981 {
23982 \fp\_set:Nn \l\_\_box\_angle\_fp {#2}
23983 \fp\_set:Nn \l\_\_box\_sin\_fp { sind ( \l\_\_box\_angle\_fp ) }
23984 \fp\_set:Nn \l\_\_box\_cos\_fp { cosd ( \l\_\_box\_angle\_fp ) }
23985 \\_box\_rotate:N #1
23986 }
23987 }

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.

```

23988 \cs_new_protected:Npn \_box_rotate:N #1
23989 {
23990     \dim_set:Nn \l__box_top_dim   { \box_ht:N #1 }
23991     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
23992     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
23993     \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.)

```

23994 \fp_compare:nNnTF \l_box_sin_fp > \c_zero_fp
23995 {
23996   \fp_compare:nNnTF \l_box_cos_fp > \c_zero_fp
23997   { \__box_rotate_quadrant_one: }
23998   { \__box_rotate_quadrant_two: }
23999 }
24000 {
24001   \fp_compare:nNnTF \l_box_cos_fp < \c_zero_fp
24002   { \__box_rotate_quadrant_three: }
24003   { \__box_rotate_quadrant_four: }
24004 }
```

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.

```

24005 \hbox_set:Nn \l_box_internal_box { \box_use:N #1 }
24006 \hbox_set:Nn \l_box_internal_box
24007 {
24008   \tex_kern:D -\l_box_left_new_dim
24009   \hbox:n
24010   {
24011     \driver_box_use_rotate:Nn
24012       \l_box_internal_box
24013       \l_box_angle_fp
24014   }
24015 }
```

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.

```

24016 \box_set_ht:Nn \l_box_internal_box { \l_box_top_new_dim }
24017 \box_set_dp:Nn \l_box_internal_box { -\l_box_bottom_new_dim }
24018 \box_set_wd:Nn \l_box_internal_box
24019 { \l_box_right_new_dim - \l_box_left_new_dim }
24020 \box_use_drop:N \l_box_internal_box
24021 }
```

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.

```

24022 \cs_new_protected:Npn \__box_rotate_xdir:nnN #1#2#3
24023 {
24024   \dim_set:Nn #3
24025   {
24026     \fp_to_dim:n
24027     {
24028       \l_box_cos_fp * \dim_to_fp:n {#1}
24029       - \l_box_sin_fp * \dim_to_fp:n {#2}
```

```

24030         }
24031     }
24032   }
24033 \cs_new_protected:Npn \__box_rotate_ydir:nnN #1#2#3
24034   {
24035     \dim_set:Nn #3
24036     {
24037       \fp_to_dim:n
24038       {
24039         \l__box_sin_fp * \dim_to_fp:n {#1}
24040         + \l__box_cos_fp * \dim_to_fp:n {#2}
24041       }
24042     }
24043   }

```

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.

```

24044 \cs_new_protected:Npn \__box_rotate_quadrant_one:
24045   {
24046     \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
24047     \l__box_top_new_dim
24048     \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
24049     \l__box_bottom_new_dim
24050     \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
24051     \l__box_left_new_dim
24052     \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
24053     \l__box_right_new_dim
24054   }
24055 \cs_new_protected:Npn \__box_rotate_quadrant_two:
24056   {
24057     \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
24058     \l__box_top_new_dim
24059     \__box_rotate_ydir:nnN \l__box_left_dim \l__box_top_dim
24060     \l__box_bottom_new_dim
24061     \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
24062     \l__box_left_new_dim
24063     \__box_rotate_xdir:nnN \l__box_left_dim \l__box_bottom_dim
24064     \l__box_right_new_dim
24065   }
24066 \cs_new_protected:Npn \__box_rotate_quadrant_three:
24067   {
24068     \__box_rotate_ydir:nnN \l__box_left_dim \l__box_bottom_dim
24069     \l__box_top_new_dim
24070     \__box_rotate_ydir:nnN \l__box_right_dim \l__box_top_dim
24071     \l__box_bottom_new_dim
24072     \__box_rotate_xdir:nnN \l__box_right_dim \l__box_bottom_dim
24073     \l__box_left_new_dim
24074     \__box_rotate_xdir:nnN \l__box_left_dim \l__box_top_dim
24075     \l__box_right_new_dim
24076   }
24077 \cs_new_protected:Npn \__box_rotate_quadrant_four:

```

```

24078  {
24079    \__box_rotate_ydir:nnN \l__box_left_dim  \l__box_top_dim
24080    \l__box_top_new_dim
24081    \__box_rotate_ydir:nnN \l__box_right_dim \l__box_bottom_dim
24082    \l__box_bottom_new_dim
24083    \__box_rotate_xdir:nnN \l__box_left_dim  \l__box_bottom_dim
24084    \l__box_left_new_dim
24085    \__box_rotate_xdir:nnN \l__box_right_dim \l__box_top_dim
24086    \l__box_right_new_dim
24087  }

```

(End definition for `\box_rotate:Nn` and others. These functions are documented on page 231.)

`\l__box_scale_x_fp` Scaling is potentially-different in the two axes.

```

24088 \fp_new:N \l__box_scale_x_fp
24089 \fp_new:N \l__box_scale_y_fp

```

(End definition for `\l__box_scale_x_fp` and `\l__box_scale_y_fp`.)

`\box_resize_to_wd_and_ht_plus_dp:Nnn` Resizing a box starts by working out the various dimensions of the existing box.

```

24090 \cs_new_protected:Npn \box_resize_to_wd_and_ht_plus_dp:cnn #1#2#3
24091  {
24092    \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
24093    \hbox_set:Nn
24094  }
24095 \cs_generate_variant:Nn \box_resize_to_wd_and_ht_plus_dp:Nnn { c }
24096 \cs_new_protected:Npn \box_gresize_to_wd_and_ht_plus_dp:Nnn #1#2#3
24097  {
24098    \__box_resize_to_wd_and_ht_plus_dp:NnnN #1 {#2} {#3}
24099    \hbox_gset:Nn
24100  }
24101 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht_plus_dp:Nnn { c }
24102 \cs_new_protected:Npn \__box_resize_to_wd_and_ht_plus_dp:NnnN #1#2#3#4
24103  {
24104    #4 #1
24105  {
24106    \__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.

```

24107   \fp_set:Nn \l__box_scale_x_fp
24108   { \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.

```

24109   \fp_set:Nn \l__box_scale_y_fp
24110   {
24111     \dim_to_fp:n {#3}
24112     / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
24113   }

```

Hand off to the auxiliary which does the rest of the work.

```

24114   \__box_resize:N #1
24115   }
24116   }
24117 \cs_new_protected:Npn \__box_resize_set_corners:N #1
24118  {

```

```

24119      \dim_set:Nn \l__box_top_dim    { \box_ht:N #1 }
24120      \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
24121      \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
24122      \dim_zero:N \l__box_left_dim
24123  }

```

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.

```

24124 \cs_new_protected:Npn \__box_resize:N #1
24125  {
24126      \__box_resize>NNN \l__box_right_new_dim
24127          \l__box_scale_x_fp \l__box_right_dim
24128      \__box_resize>NNN \l__box_bottom_new_dim
24129          \l__box_scale_y_fp \l__box_bottom_dim
24130      \__box_resize>NNN \l__box_top_new_dim
24131          \l__box_scale_y_fp \l__box_top_dim
24132      \__box_resize_common:N #1
24133  }
24134 \cs_new_protected:Npn \__box_resize>NNN #1#2#3
24135  {
24136      \dim_set:Nn #1
24137          { \fp_to_dim:n { \fp_abs:n { #2 } * \dim_to_fp:n { #3 } } }
24138  }

```

(End definition for `\box_resize_to_wd_and_ht_plus_dp:Nnn` and others. These functions are documented on page 230.)

`\box_resize_to_ht:Nn`  
`\box_resize_to_ht:cn`  
`\box_gresize_to_ht:Nn`  
`\box_gresize_to_ht:cn`

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).

```

24139 \cs_new_protected:Npn \box_resize_to_ht:Nn #1#2
24140  { \__box_resize_to_ht:NnN #1 {#2} \hbox_set:Nn }
24141 \cs_generate_variant:Nn \box_resize_to_ht:Nn { c }
24142 \cs_new_protected:Npn \box_gresize_to_ht:Nn #1#2
24143  { \__box_resize_to_ht:NnN #1 {#2} \hbox_gset:Nn }
24144 \cs_generate_variant:Nn \box_gresize_to_ht:Nn { c }
24145 \cs_new_protected:Npn \__box_resize_to_ht:NnN #1#2#3
24146  {
24147      #3 #1
24148  {
24149      \__box_resize_set_corners:N #1
24150      \fp_set:Nn \l__box_scale_y_fp
24151          {
24152              \dim_to_fp:n {#2}
24153                  / \dim_to_fp:n { \l__box_top_dim }
24154          }
24155      \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
24156      \__box_resize:N #1
24157  }
24158  }
24159 \cs_new_protected:Npn \box_resize_to_ht_plus_dp:Nn #1#2

```

```

24160 { \__box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_set:Nn }
24161 \cs_generate_variant:Nn \box_resize_to_ht_plus_dp:Nn { c }
24162 \cs_new_protected:Npn \box_gresize_to_ht_plus_dp:Nn #1#2
24163 { \__box_resize_to_ht_plus_dp:NnN #1 {#2} \hbox_gset:Nn }
24164 \cs_generate_variant:Nn \box_gresize_to_ht_plus_dp:Nn { c }
24165 \cs_new_protected:Npn \__box_resize_to_ht_plus_dp:NnN #1#2#3
24166 {
24167 \hbox_set:Nn #1
24168 {
24169 \__box_resize_set_corners:N #1
24170 \fp_set:Nn \l__box_scale_y_fp
24171 {
24172 \dim_to_fp:n {#2}
24173 / \dim_to_fp:n { \l__box_top_dim - \l__box_bottom_dim }
24174 }
24175 \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp
24176 \__box_resize:N #1
24177 }
24178 }
24179 \cs_new_protected:Npn \box_resize_to_wd:Nn #1#2
24180 { \__box_resize_to_wd:NnN #1 {#2} \hbox_set:Nn }
24181 \cs_generate_variant:Nn \box_resize_to_wd:Nn { c }
24182 \cs_new_protected:Npn \box_gresize_to_wd:Nn #1#2
24183 { \__box_resize_to_wd:NnN #1 {#2} \hbox_gset:Nn }
24184 \cs_generate_variant:Nn \box_gresize_to_wd:Nn { c }
24185 \cs_new_protected:Npn \__box_resize_to_wd:NnN #1#2#3
24186 {
24187 #3 #1
24188 {
24189 \__box_resize_set_corners:N #1
24190 \fp_set:Nn \l__box_scale_x_fp
24191 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
24192 \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp
24193 \__box_resize:N #1
24194 }
24195 }
24196 \cs_new_protected:Npn \box_resize_to_wd_and_ht:Nnn #1#2#3
24197 { \__box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_set:Nn }
24198 \cs_generate_variant:Nn \box_resize_to_wd_and_ht:Nnn { c }
24199 \cs_new_protected:Npn \box_gresize_to_wd_and_ht:Nnn #1#2#3
24200 { \__box_resize_to_wd_and_ht:NnnN #1 {#2} {#3} \hbox_gset:Nn }
24201 \cs_generate_variant:Nn \box_gresize_to_wd_and_ht:Nnn { c }
24202 \cs_new_protected:Npn \__box_resize_to_wd_and_ht:NnnN #1#2#3#4
24203 {
24204 #4 #1
24205 {
24206 \__box_resize_set_corners:N #1
24207 \fp_set:Nn \l__box_scale_x_fp
24208 { \dim_to_fp:n {#2} / \dim_to_fp:n { \l__box_right_dim } }
24209 \fp_set:Nn \l__box_scale_y_fp
24210 {
24211 \dim_to_fp:n {#3}
24212 / \dim_to_fp:n { \l__box_top_dim }
24213 }

```

```

24214         \__box_resize:N #1
24215     }
24216 }

```

(End definition for `\box_resize_to_ht:Nn` and others. These functions are documented on page 229.)

```

\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.

```

24217 \cs_new_protected:Npn \box_scale:Nnn #1#2#3
24218   { \__box_scale:NnnN #1 {#2} {#3} \hbox_set:Nn }
24219 \cs_generate_variant:Nn \box_scale:Nnn { c }
24220 \cs_new_protected:Npn \box_gscale:Nnn #1#2#3
24221   { \__box_scale:NnnN #1 {#2} {#3} \hbox_gset:Nn }
24222 \cs_generate_variant:Nn \box_gscale:Nnn { c }
24223 \cs_new_protected:Npn \__box_scale:NnnN #1#2#3#4
24224   {
24225     #4 #1
24226   {
24227     \fp_set:Nn \l__box_scale_x_fp {#2}
24228     \fp_set:Nn \l__box_scale_y_fp {#3}
24229     \__box_scale:N #1
24230   }
24231 }
24232 \cs_new_protected:Npn \__box_scale:N #1
24233   {
24234     \dim_set:Nn \l__box_top_dim { \box_ht:N #1 }
24235     \dim_set:Nn \l__box_bottom_dim { -\box_dp:N #1 }
24236     \dim_set:Nn \l__box_right_dim { \box_wd:N #1 }
24237     \dim_zero:N \l__box_left_dim
24238     \dim_set:Nn \l__box_top_new_dim
24239       { \fp_abs:n { \l__box_scale_y_fp } \l__box_top_dim }
24240     \dim_set:Nn \l__box_bottom_new_dim
24241       { \fp_abs:n { \l__box_scale_y_fp } \l__box_bottom_dim }
24242     \dim_set:Nn \l__box_right_new_dim
24243       { \fp_abs:n { \l__box_scale_x_fp } \l__box_right_dim }
24244     \__box_resize_common:N #1
24245   }

```

(End definition for `\box_scale:Nnn` and others. These functions are documented on page 231.)

```

\box_autosize_to_wd_and_ht:Nnn
\box_autosize_to_wd_and_ht:cnn
\box_gautosize_to_wd_and_ht:Nnn
\box_gautosize_to_wd_and_ht:cnn
\box_autosize_to_wd_and_ht_plus_dp:Nnn
\box_autosize_to_wd_and_ht_plus_dp:cnn
\box_gautosize_to_wd_and_ht_plus_dp:Nnn
\box_gautosize_to_wd_and_ht_plus_dp:cnn
\__box_autosize:NnnnN

```

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.

```

24246 \cs_new_protected:Npn \box_autosize_to_wd_and_ht:Nnn #1#2#3
24247   { \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_set:Nn }
24248 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
24249 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht:Nnn #1#2#3
24250   { \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 } \hbox_gset:Nn }
24251 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht:Nnn { c }
24252 \cs_new_protected:Npn \box_autosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
24253   {
24254     \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }

```

```

24255     \hbox_set:Nn
24256   }
24257 \cs_generate_variant:Nn \box_autosize_to_wd_and_ht_plus_dp:Nnn { c }
24258 \cs_new_protected:Npn \box_gautosize_to_wd_and_ht_plus_dp:Nnn #1#2#3
24259   {
24260     \__box_autosize:NnnnN #1 {#2} {#3} { \box_ht:N #1 + \box_dp:N #1 }
24261     \hbox_gset:Nn
24262   }
24263 \cs_generate_variant:Nn \box_gautosize_to_wd_and_ht_plus_dp:Nnn { c }
24264 \cs_new_protected:Npn \__box_autosize:NnnnN #1#2#3#4#5
24265   {
24266     #5 #1
24267   {
24268     \fp_set:Nn \l__box_scale_x_fp { ( #2 ) / \box_wd:N #1 }
24269     \fp_set:Nn \l__box_scale_y_fp { ( #3 ) / ( #4 ) }
24270     \fp_compare:nNnTF \l__box_scale_x_fp > \l__box_scale_y_fp
24271       { \fp_set_eq:NN \l__box_scale_x_fp \l__box_scale_y_fp }
24272       { \fp_set_eq:NN \l__box_scale_y_fp \l__box_scale_x_fp }
24273     \__box_scale:N #1
24274   }
24275 }

```

(End definition for `\box_autosize_to_wd_and_ht:Nnn` and others. These functions are documented on page [229](#).)

`\__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.

```

24276 \cs_new_protected:Npn \__box_resize_common:N #1
24277   {
24278     \hbox_set:Nn \l__box_internal_box
24279   {
24280     \driver_box_use_scale:Nnn
24281     #1
24282     \l__box_scale_x_fp
24283     \l__box_scale_y_fp
24284   }

```

The new height and depth can be applied directly.

```

24285 \fp_compare:nNnTF \l__box_scale_y_fp > \c_zero_fp
24286   {
24287     \box_set_ht:Nn \l__box_internal_box { \l__box_top_new_dim }
24288     \box_set_dp:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
24289   }
24290   {
24291     \box_set_dp:Nn \l__box_internal_box { \l__box_top_new_dim }
24292     \box_set_ht:Nn \l__box_internal_box { -\l__box_bottom_new_dim }
24293   }

```

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.

```

24294 \fp_compare:nNnTF \l__box_scale_x_fp < \c_zero_fp
24295   {
24296     \hbox_to_wd:nn { \l__box_right_new_dim }

```

```

24297   {
24298     \tex_kern:D \l_box_right_new_dim
24299     \box_use_drop:N \l_box_internal_box
24300     \tex_hss:D
24301   }
24302 }
24303 {
24304   \box_set_wd:Nn \l_box_internal_box { \l_box_right_new_dim }
24305   \hbox:n
24306   {
24307     \tex_kern:D \c_zero_dim
24308     \box_use_drop:N \l_box_internal_box
24309     \tex_hss:D
24310   }
24311 }
24312 }
```

(End definition for `\_box_resize_common:N`.)

### 39.13 Deprecated functions

```

\box_set_eq_clear:NN
\box_set_eq_clear:cN
\box_set_eq_clear:Nc
\box_set_eq_clear:cc
\box_gset_eq_clear:NN
\box_gset_eq_clear:cN
\box_gset_eq_clear:Nc
\box_gset_eq_clear:cc
24313 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \box_set_eq_drop:N }
24314 \cs_new_protected:Npn \box_set_eq_clear:NN #1#2
24315 { \tex_setbox:D #1 \tex_box:D #2 }
24316 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \box_gset_eq_drop:N }
24317 \cs_new_protected:Npn \box_gset_eq_clear:NN #1#2
24318 { \tex_global:D \tex_setbox:D #1 \tex_box:D #2 }
24319 \cs_generate_variant:Nn \box_set_eq_clear:NN { c , Nc , cc }
24320 \cs_generate_variant:Nn \box_gset_eq_clear:NN { c , Nc , cc }
```

(End definition for `\box_set_eq_clear:NN` and `\box_gset_eq_clear:NN`. These functions are documented on page ??.)

```

\hbox_unpack_clear:N
\hbox_unpack_clear:c
24321 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \hbox_unpack_drop:N }
24322 \cs_new_protected:Npn \hbox_unpack_clear:N
24323 { \hbox_unpack_drop:N }
24324 \cs_generate_variant:Nn \hbox_unpack_clear:N { c }
```

(End definition for `\hbox_unpack_clear:N`. This function is documented on page ??.)

```

\vbox_unpack_clear:N
\vbox_unpack_clear:c
24325 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \vbox_unpack_drop:N }
24326 \cs_new_protected:Npn \vbox_unpack_clear:N
24327 { \vbox_unpack_drop:N }
24328 \cs_generate_variant:Nn \vbox_unpack_clear:N { c }
```

(End definition for `\vbox_unpack_clear:N`. This function is documented on page ??.)

24329 ⟨/initex | package⟩

## 40 I3coffins Implementation

```
24330  {*initex | package}
24331  <@=coffin>
```

### 40.1 Coffins: data structures and general variables

\l\_coffin\_internal\_box Scratch variables.

```
24332  \box_new:N \l_coffin_internal_box
24333  \dim_new:N \l_coffin_internal_dim
24334  \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.

```
24335  \prop_const_from_keyval:Nn \c_coffin_corners_prop
24336  {
24337  t1 = { Opt } { Opt } ,
24338  tr = { Opt } { Opt } ,
24339  bl = { Opt } { Opt } ,
24340  br = { Opt } { Opt } ,
24341 }
```

(End definition for \c\_coffin\_corners\_prop.)

\c\_coffin\_poles\_prop Pole positions are given for horizontal, vertical and reference-point based values.

```
24342  \prop_const_from_keyval:Nn \c_coffin_poles_prop
24343  {
24344  l = { Opt } { Opt } { Opt } { 1000pt } ,
24345  hc = { Opt } { Opt } { Opt } { 1000pt } ,
24346  r = { Opt } { Opt } { Opt } { 1000pt } ,
24347  b = { Opt } { Opt } { 1000pt } { Opt } ,
24348  vc = { Opt } { Opt } { 1000pt } { Opt } ,
24349  t = { Opt } { Opt } { 1000pt } { Opt } ,
24350  B = { Opt } { Opt } { 1000pt } { Opt } ,
24351  H = { Opt } { Opt } { 1000pt } { Opt } ,
24352  T = { Opt } { Opt } { 1000pt } { Opt } ,
24353 }
```

(End definition for \c\_coffin\_poles\_prop.)

\l\_coffin\_slope\_x\_fp Used for calculations of intersections.

```
24354  \fp_new:N \l_coffin_slope_x_fp
24355  \fp_new:N \l_coffin_slope_y_fp
```

(End definition for \l\_coffin\_slope\_x\_fp and \l\_coffin\_slope\_y\_fp.)

\l\_coffin\_error\_bool For propagating errors so that parts of the code can work around them.

```
24356  \bool_new:N \l_coffin_error_bool
```

(End definition for \l\_coffin\_error\_bool.)

\l\_coffin\_offset\_x\_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.

```
24357 \dim_new:N \l_coffin_offset_x_dim  
24358 \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\_tl    Needed for finding the intersection of two poles.

```
24359 \tl_new:N \l_coffin_pole_a_tl  
24360 \tl_new:N \l_coffin_pole_b_tl
```

(End definition for \l\_coffin\_pole\_a\_tl and \l\_coffin\_pole\_b\_tl.)

\l\_coffin\_x\_dim    For calculating intersections and so forth.

```
24361 \dim_new:N \l_coffin_x_dim  
24362 \dim_new:N \l_coffin_y_dim  
24363 \dim_new:N \l_coffin_x_prime_dim  
24364 \dim_new:N \l_coffin_y_prime_dim
```

(End definition for \l\_coffin\_x\_dim and others.)

## 40.2 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.

```
24365 \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.

```
24366 \prg_new_conditional:Npnn \coffin_if_exist:N #1 { p , T , F , TF }  
24367 {  
24368   \cs_if_exist:NTF #1  
24369   {  
24370     \cs_if_exist:cTF { coffin ~ \_\coffin_to_value:N #1 ~ poles }  
24371     { \prg_return_true: }  
24372     { \prg_return_false: }  
24373   }  
24374   { \prg_return_false: }  
24375 }  
24376 \prg_generate_conditional_variant:Nnn \coffin_if_exist:N  
24377 { c } { p , T , F , TF }
```

(End definition for \coffin\_if\_exist:NTF. This function is documented on page 232.)

\\_\_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.

```

24378 \cs_new_protected:Npn \__coffin_if_exist:NT #1#2
24379  {
24380      \coffin_if_exist:NTF #1
24381      { #2 }
24382      {
24383          \__kernel_msg_error:nnx { kernel } { unknown-coffin }
24384          { \token_to_str:N #1 }
24385      }
24386  }

```

(End definition for \\_\_coffin\_if\_exist:NT.)

\coffin\_clear:N Clearing coffins means emptying the box and resetting all of the structures.

```

\coffin_clear:c
\coffin_gclear:N
\coffin_gclear:c
24387 \cs_new_protected:Npn \coffin_clear:N #1
24388  {
24389      \__coffin_if_exist:NT #1
24390      {
24391          \box_clear:N #1
24392          \__coffin_reset_structure:N #1
24393      }
24394  }
24395 \cs_generate_variant:Nn \coffin_clear:N { c }
24396 \cs_new_protected:Npn \coffin_gclear:N #1
24397  {
24398      \__coffin_if_exist:NTF #1
24399      {
24400          \box_gclear:N #1
24401          \__coffin_greset_structure:N #1
24402      }
24403  }
24404 \cs_generate_variant:Nn \coffin_gclear:N { c }

```

(End definition for \coffin\_clear:N and \coffin\_gclear:N. These functions are documented on page 232.)

\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.

```

\coffin_new:c
24405 \cs_new_protected:Npn \coffin_new:N #1
24406  {
24407      \box_new:N #1
24408      \debug_suspend:
24409      \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ corners }
24410      \prop_gclear_new:c { coffin ~ \__coffin_to_value:N #1 ~ poles }
24411      \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
24412          \c__coffin_corners_prop
24413      \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
24414          \c__coffin_poles_prop
24415      \debug_resume:
24416  }
24417 \cs_generate_variant:Nn \coffin_new:N { c }

```

(End definition for `\coffin_new:N`. This function is documented on page 232.)

`\hcoffin_set:Nn` Horizontal coffins are relatively easy: set the appropriate box, reset the structures then update the handle positions.

```
24418 \cs_new_protected:Npn \hcoffin_set:Nn #1#2
24419 {
24420     \__coffin_if_exist:NT #1
24421     {
24422         \hbox_set:Nn #1
24423         {
24424             \color_ensure_current:
24425             #2
24426         }
24427         \__coffin_update:N #1
24428     }
24429 }
24430 \cs_generate_variant:Nn \hcoffin_set:Nn { c }
24431 \cs_new_protected:Npn \hcoffin_gset:Nn #1#2
24432 {
24433     \__coffin_if_exist:NT #1
24434     {
24435         \hbox_gset:Nn #1
24436         {
24437             \color_ensure_current:
24438             #2
24439         }
24440         \__coffin_gupdate:N #1
24441     }
24442 }
24443 \cs_generate_variant:Nn \hcoffin_gset:Nn { c }
```

(End definition for `\hcoffin_set:Nn` and `\hcoffin_gset:Nn`. These functions are documented on page 233.)

`\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 whatsit to the start of the vertical box and mess up the location of the T pole (see *TEX by Topic* for discussion of the `\vtop` primitive, used to do the measuring).

```
\vcoffin_set:cnn
\vcoffin_set:cnn
\hcoffin_gset:Nnn
\hcoffin_gset:cnn
\__coffin_set_vertical:NnnNN
```

```
24444 \cs_new_protected:Npn \vcoffin_set:Nnn #1#2#3
24445 {
24446     \__coffin_set_vertical:NnnNN #1 {#2} {#3}
24447     \vbox_set:Nn \__coffin_update:N
24448 }
24449 \cs_generate_variant:Nn \vcoffin_set:Nnn { c }
24450 \cs_new_protected:Npn \vcoffin_gset:Nnn #1#2#3
24451 {
24452     \__coffin_set_vertical:NnnNN #1 {#2} {#3}
24453     \vbox_gset:Nn \__coffin_gupdate:N
24454 }
24455 \cs_generate_variant:Nn \vcoffin_gset:Nnn { c }
24456 \cs_new_protected:Npn \__coffin_set_vertical:NnnNN #1#2#3#4#5
24457 {
24458     \__coffin_if_exist:NT #1
```

```

24459 {
24460   #4 #1
24461   {
24462     \dim_set:Nn \tex_hsize:D {#2}
24463   (*package)
24464   \dim_set_eq:NN \linewidth \tex_hsize:D
24465   \dim_set_eq:NN \columnwidth \tex_hsize:D
24466   
```

`</package>`

`24467 #3`

`24468 }`

`24469 #5 #1`

`\vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }`

`\__coffin_set_pole:Nnx #1 { T }`

`24472 {`

`24473 { Opt }`

`24474 {`

`24475 \dim_eval:n`

`{ \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }`

`24477 }`

`24478 { 1000pt }`

`24479 { Opt }`

`24480 }`

`\box_clear:N \l__coffin_internal_box`

`24482 }`

`24483 }`

(End definition for `\vcoffin_set:Nnn`, `\vcoffin_gset:Nnn`, and `\__coffin_set_vertical:NnnNN`. These functions are documented on page 233.)

**`\hcoffin_set:Nw`** These are the “begin”/“end” versions of the above: watch the grouping!

```

\hcoffin_set:cw 24484 \cs_new_protected:Npn \hcoffin_set:Nw #1
\hcoffin_gset:Nw 24485 {
\hcoffin_gset:cw 24486   \__coffin_if_exist:NT #1
\hcoffin_set_end: 24487   {
\hcoffin_gset_end: 24488     \hbox_set:Nw #1 \color_ensure_current:
24489     \cs_set_protected:Npn \hcoffin_set_end:
24490     {
24491       \hbox_set_end:
24492       \__coffin_update:N #1
24493     }
24494   }
24495 }
24496 \cs_generate_variant:Nn \hcoffin_set:Nw { c }
24497 \cs_new_protected:Npn \hcoffin_gset:Nw #1
24498 {
24499   \__coffin_if_exist:NT #1
24500   {
24501     \hbox_gset:Nw #1 \color_ensure_current:
24502     \cs_set_protected:Npn \hcoffin_gset_end:
24503     {
24504       \hbox_gset_end:
24505       \__coffin_gupdate:N #1
24506     }
24507 }
```

```

24508     }
24509     \cs_generate_variant:Nn \hcoffin_gset:Nw { c }
24510     \cs_new_protected:Npn \hcoffin_set_end: { }
24511     \cs_new_protected:Npn \hcoffin_gset_end: { }

(End definition for \hcoffin_set:Nw and others. These functions are documented on page 233.)
```

**\vcoffin\_set:Nnw** The same for vertical coffins.

```

24512     \cs_new_protected:Npn \vcoffin_set:Nnw #1#2
24513     {
24514         \__coffin_set_vertical:NnNNNNw #1 {#2} \vbox_set:Nw
24515         \vcoffin_set_end:
24516         \vbox_set_end: \__coffin_update:N
24517     }
24518     \cs_generate_variant:Nn \vcoffin_set:Nnw { c }
24519     \cs_new_protected:Npn \vcoffin_gset:Nnw #1#2
24520     {
24521         \__coffin_set_vertical:NnNNNNw #1 {#2} \vbox_gset:Nw
24522         \vcoffin_gset_end:
24523         \vbox_gset_end: \__coffin_gupdate:N
24524     }
24525     \cs_generate_variant:Nn \vcoffin_gset:Nnw { c }
24526     \cs_new_protected:Npn \__coffin_set_vertical:NnNNNNw #1#2#3#4#5#6
24527     {
24528         \__coffin_if_exist:NT #1
24529         {
24530             #3 #1
24531             \dim_set:Nn \tex_hsize:D {#2}
24532         (*package)
24533             \dim_set_eq:NN \linewidth \tex_hsize:D
24534             \dim_set_eq:NN \columnwidth \tex_hsize:D
24535     (/package)
24536         \cs_set_protected:Npn #4
24537         {
24538             #5
24539             #6 #1
24540             \vbox_set_top:Nn \l__coffin_internal_box { \vbox_unpack:N #1 }
24541             \__coffin_set_pole:Nnx #1 { T }
24542             {
24543                 { Opt }
24544                 {
24545                     \dim_eval:n
24546                         { \box_ht:N #1 - \box_ht:N \l__coffin_internal_box }
24547                 }
24548                 { 1000pt }
24549                 { Opt }
24550                 }
24551                 \box_clear:N \l__coffin_internal_box
24552             }
24553         }
24554     }
24555     \cs_new_protected:Npn \vcoffin_set_end: { }
24556     \cs_new_protected:Npn \vcoffin_gset_end: { }

(End definition for \vcoffin_set:Nnw and others. These functions are documented on page 233.)
```

```

\coffin_set_eq:NN Setting two coffins equal is just a wrapper around other functions.
\coffin_set_eq:Nc
\coffin_gset_eq:cN
\coffin_gset_eq:cc
24557 \cs_new_protected:Npn \coffin_set_eq:NN #1#2
24558 {
24559     \__coffin_if_exist:NT #1
24560     {
24561         \box_set_eq:NN #1 #2
24562         \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
24563             { coffin ~ \__coffin_to_value:N #2 ~ corners }
24564             \prop_set_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
24565                 { coffin ~ \__coffin_to_value:N #2 ~ poles }
24566     }
24567 }
24568 \cs_generate_variant:Nn \coffin_set_eq:NN { c , Nc , cc }
24569 \cs_new_protected:Npn \coffin_gset_eq:NN #1#2
24570 {
24571     \__coffin_if_exist:NT #1
24572     {
24573         \box_gset_eq:NN #1 #2
24574         \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ corners }
24575             { coffin ~ \__coffin_to_value:N #2 ~ corners }
24576             \prop_gset_eq:cc { coffin ~ \__coffin_to_value:N #1 ~ poles }
24577                 { coffin ~ \__coffin_to_value:N #2 ~ poles }
24578     }
24579 }
24580 \cs_generate_variant:Nn \coffin_gset_eq:NN { c , Nc , cc }

```

(End definition for `\coffin_set_eq:NN` and `\coffin_gset_eq:cN`. These functions are documented on page 232.)

`\c_empty_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.

```

24581 \coffin_new:N \c_empty_coffin
24582 \coffin_new:N \l__coffin_aligned_coffin
24583 \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 236.)

`\l_tmpa_coffin` The usual scratch space.

```

24584 \coffin_new:N \l_tmpa_coffin
24585 \coffin_new:N \l_tmpb_coffin
24586 \coffin_new:N \g_tmpa_coffin
24587 \coffin_new:N \g_tmpb_coffin

```

(End definition for `\l_tmpa_coffin` and others. These variables are documented on page 236.)

### 40.3 Measuring coffins

`\coffin_dp:N` Coffins are just boxes when it comes to measurement. However, semantically a separate set of functions are required.

```

24588 \cs_new_eq:NN \coffin_dp:N \box_dp:N
24589 \cs_new_eq:NN \coffin_dp:c \box_dp:c
24590 \cs_new_eq:NN \coffin_ht:N \box_ht:N
\coffin_wd:c

```

```

24591 \cs_new_eq:NN \coffin_ht:c \box_ht:c
24592 \cs_new_eq:NN \coffin_wd:N \box_wd:N
24593 \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 235.)

## 40.4 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.

```

24594 \cs_new_protected:Npn \__coffin_get_pole:NnN #1#2#3
24595 {
24596     \prop_get:cnNF
24597         { coffin ~ \__coffin_to_value:N #1 ~ poles } {#2} #3
24598     {
24599         \__kernel_msg_error:nxxx { kernel } { unknown-coffin-pole }
24600             { \exp_not:n {#2} } { \token_to_str:N #1 }
24601             \tl_set:Nn #3 { { Opt } { Opt } { Opt } { Opt } }
24602     }
24603 }

```

(End definition for `\__coffin_get_pole:NnN`.)

`\__coffin_reset_structure:N`

Resetting the structure is a simple copy job.

```

24604 \cs_new_protected:Npn \__coffin_reset_structure:N #1
24605 {
24606     \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
24607         \c__coffin_corners_prop
24608     \prop_set_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
24609         \c__coffin_poles_prop
24610 }
24611 \cs_new_protected:Npn \__coffin_greset_structure:N #1
24612 {
24613     \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ corners }
24614         \c__coffin_corners_prop
24615     \prop_gset_eq:cN { coffin ~ \__coffin_to_value:N #1 ~ poles }
24616         \c__coffin_poles_prop
24617 }

```

(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_horizontal_pole:NnnN`

`\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.

```

24618 \cs_new_protected:Npn \coffin_set_horizontal_pole:Nnn #1#2#3
24619     { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_put:cnx }
24620 \cs_generate_variant:Nn \coffin_set_horizontal_pole:Nnn { c }
24621 \cs_new_protected:Npn \coffin_gset_horizontal_pole:Nnn #1#2#3
24622     { \__coffin_set_horizontal_pole:NnnN #1 {#2} {#3} \prop_gput:cnx }
24623 \cs_generate_variant:Nn \coffin_gset_horizontal_pole:Nnn { c }
24624 \cs_new_protected:Npn \__coffin_set_horizontal_pole:NnnN #1#2#3#4
24625     {
24626         \__coffin_if_exist:NT #1

```

```

24627 {
24628   #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
24629   {#2}
24630   {
24631     { Opt } { \dim_eval:n {#3} }
24632     { 1000pt } { Opt }
24633   }
24634 }
24635 }
24636 \cs_new_protected:Npn \coffin_set_vertical_pole:Nnn #1#2#3
24637   { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_put:cnn }
24638 \cs_generate_variant:Nn \coffin_set_vertical_pole:Nnn { c }
24639 \cs_new_protected:Npn \coffin_gset_vertical_pole:Nnn #1#2#3
24640   { \__coffin_set_vertical_pole:NnnN #1 {#2} {#3} \prop_gput:cnn }
24641 \cs_generate_variant:Nn \coffin_gset_vertical_pole:Nnn { c }
24642 \cs_new_protected:Npn \__coffin_set_vertical_pole:NnnN #1#2#3#4
24643 {
24644   \__coffin_if_exist:NT #1
24645   {
24646     #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
24647     {#2}
24648     {
24649       { \dim_eval:n {#3} } { Opt }
24650       { Opt } { 1000pt }
24651     }
24652   }
24653 }
24654 \cs_new_protected:Npn \__coffin_set_pole:Nnn #1#2#3
24655 {
24656   \prop_put:cnn { coffin ~ \__coffin_to_value:N #1 ~ poles }
24657   {#2} {#3}
24658 }
24659 \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 233.)
```

\\_\_coffin\_update:N Simple shortcuts.

```

24660 \cs_new_protected:Npn \__coffin_update:N #1
24661 {
24662   \__coffin_reset_structure:N #1
24663   \__coffin_update_corners:N #1
24664   \__coffin_update_poles:N #1
24665 }
24666 \cs_new_protected:Npn \__coffin_gupdate:N #1
24667 {
24668   \__coffin_greset_structure:N #1
24669   \__coffin_gupdate_corners:N #1
24670   \__coffin_gupdate_poles:N #1
24671 }
```

(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

```

24672 \cs_new_protected:Npn \__coffin_update_corners:N #1
24673   { \__coffin_update_corners:NN #1 \prop_put:Nnx }
24674 \cs_new_protected:Npn \__coffin_gupdate_corners:N #1
24675   { \__coffin_update_corners:NN #1 \prop_gput:Nnx }
24676 \cs_new_protected:Npn \__coffin_update_corners:NN #1#2
24677   {
24678     \exp_args:Nc \__coffin_update_corners:NNN
24679       { coffin ~ \__coffin_to_value:N #1 ~ corners }
24680       #1 #2
24681   }
24682 \cs_new_protected:Npn \__coffin_update_corners:NNN #1#2#3
24683   {
24684     #3 #1
24685       { tl }
24686       { { Opt } { \dim_eval:n { \box_ht:N #2 } } }
24687     #3 #1
24688       { tr }
24689       {
24690         { \dim_eval:n { \box_wd:N #2 } }
24691         { \dim_eval:n { \box_ht:N #2 } }
24692       }
24693     #3 #1
24694       { bl }
24695       { { Opt } { \dim_eval:n { -\box_dp:N #2 } } }
24696     #3 #1
24697       { br }
24698       {
24699         { \dim_eval:n { \box_wd:N #2 } }
24700         { \dim_eval:n { -\box_dp:N #2 } }
24701       }
24702   }

```

(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.

```

24703 \cs_new_protected:Npn \__coffin_update_poles:N #1
24704   { \__coffin_update_poles:NN #1 \prop_put:Nnx }
24705 \cs_new_protected:Npn \__coffin_gupdate_poles:N #1
24706   { \__coffin_update_poles:NN #1 \prop_gput:Nnx }
24707 \cs_new_protected:Npn \__coffin_update_poles:NN #1#2
24708   {
24709     \exp_args:Nc \__coffin_update_poles:NNN
24710       { coffin ~ \__coffin_to_value:N #1 ~ poles }
24711       #1 #2
24712   }
24713 \cs_new_protected:Npn \__coffin_update_poles:NNN #1#2#3
24714   {
24715     #3 #1 { hc }
24716     {
24717       { \dim_eval:n { 0.5 \box_wd:N #2 } }
24718       { Opt } { Opt } { 1000pt }

```

```

24719 }
24720 #3 #1 { r }
24721 {
24722   { \dim_eval:n { \box_wd:N #2 } }
24723   { Opt } { Opt } { 1000pt }
24724 }
24725 #3 #1 { vc }
24726 {
24727   { Opt }
24728   { \dim_eval:n { ( \box_ht:N #2 - \box_dp:N #2 ) / 2 } }
24729   { 1000pt }
24730   { Opt }
24731 }
24732 #3 #1 { t }
24733 {
24734   { Opt }
24735   { \dim_eval:n { \box_ht:N #2 } }
24736   { 1000pt }
24737   { Opt }
24738 }
24739 #3 #1 { b }
24740 {
24741   { Opt }
24742   { \dim_eval:n { -\box_dp:N #2 } }
24743   { 1000pt }
24744   { Opt }
24745 }
24746 }

```

(End definition for `\__coffin_update_poles:N` and others.)

## 40.5 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.

```

24747 \cs_new_protected:Npn \__coffin_calculate_intersection:Nnn #1#2#3
24748 {
24749   \__coffin_get_pole:NnN #1 {#2} \l__coffin_pole_a_tl
24750   \__coffin_get_pole:NnN #1 {#3} \l__coffin_pole_b_tl
24751   \bool_set_false:N \l__coffin_error_bool
24752   \exp_last_two_unbraced:Noo
24753     \__coffin_calculate_intersection:nnnnnnnn
24754     \l__coffin_pole_a_tl \l__coffin_pole_b_tl
24755   \bool_if:NT \l__coffin_error_bool
24756   {
24757     \__kernel_msg_error:nn { kernel } { no-pole-intersection }
24758     \dim_zero:N \l__coffin_x_dim
24759     \dim_zero:N \l__coffin_y_dim
24760   }
24761 }

```

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.

```

24762 \cs_new_protected:Npn \__coffin_calculate_intersection:nnnnnnn
24763   #1#2#3#4#5#6#7#8
24764   {
24765     \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.

```

24766   {
24767     \dim_set:Nn \l__coffin_x_dim {#1}
24768     \dim_compare:nNnTF {#7} = \c_zero_dim
24769       { \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,

$$y = \frac{d'}{c'} (x - a') + b'$$

with the  $x$ -component already known to be #1. This calculation is done as a generalised auxiliary.

```

24770   {
24771     \dim_compare:nNnTF {#8} = \c_zero_dim
24772       { \dim_set:Nn \l__coffin_y_dim {#6} }
24773       {
24774         \__coffin_calculate_intersection_aux:nnnnN
24775           {#1} {#5} {#6} {#7} {#8} \l__coffin_y_dim
24776       }
24777     }
24778   }

```

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.

```

24779   {
24780     \dim_compare:nNnTF {#4} = \c_zero_dim
24781     {
24782       \dim_set:Nn \l__coffin_y_dim {#2}
24783       \dim_compare:nNnTF {#8} = { \c_zero_dim }
24784         { \bool_set_true:N \l__coffin_error_bool }
24785         {
24786           \dim_compare:nNnTF {#7} = \c_zero_dim
24787             { \dim_set:Nn \l__coffin_x_dim {#5} }

```

The formula for the case where the second pole is neither horizontal nor vertical is

$$x = \frac{c'}{d'} (y - b') + a'$$

which is again handled by the same auxiliary.

```

24788   {
24789     \__coffin_calculate_intersection_aux:nnnnN
24790       {#2} {#6} {#5} {#8} {#7} \l__coffin_x_dim
24791     }
24792   }
24793 }
```

The first pole is neither horizontal nor vertical. This still leaves the second pole, which may be a special case. For those possibilities, the calculations are the same as above with the first and second poles interchanged.

```

24794 {
24795   \dim_compare:nNnTF {#7} = \c_zero_dim
24796   {
24797     \dim_set:Nn \l__coffin_x_dim {#5}
24798     \l__coffin_calculate_intersection_aux:nnnnN
24799       {#5} {#1} {#2} {#3} {#4} \l__coffin_y_dim
24800   }
24801   {
24802     \dim_compare:nNnTF {#8} = \c_zero_dim
24803     {
24804       \dim_set:Nn \l__coffin_y_dim {#6}
24805       \l__coffin_calculate_intersection_aux:nnnnN
24806         {#6} {#2} {#1} {#4} {#3} \l__coffin_x_dim
24807     }

```

If none of the special cases apply then there is still a need to check that there is a unique intersection between the two pole. This is the case if they have different slopes.

```

24808 {
24809   \fp_set:Nn \l__coffin_slope_x_fp
24810     { \dim_to_fp:n {#4} / \dim_to_fp:n {#3} }
24811   \fp_set:Nn \l__coffin_slope_y_fp
24812     { \dim_to_fp:n {#8} / \dim_to_fp:n {#7} }
24813   \fp_compare:nNnTF
24814     \l__coffin_slope_x_fp = \l__coffin_slope_y_fp
24815     { \bool_set_true:N \l__coffin_error_bool }

```

All of the tests pass, so there is the full complexity of the calculation:

$$x = \frac{a(d/c) - a'(d'/c') - b + b'}{(d/c) - (d'/c')}$$

and noting that the two ratios are already worked out from the test just performed. There is quite a bit of shuffling from dimensions to floating points in order to do the work. The  $y$ -values is then worked out using the standard auxiliary starting from the  $x$ -position.

```

24816 {
24817   \dim_set:Nn \l__coffin_x_dim
24818   {
24819     \fp_to_dim:n
24820     {
24821       (
24822         \dim_to_fp:n {#1} *
24823           \l__coffin_slope_x_fp
24824         - ( \dim_to_fp:n {#5} *
24825           \l__coffin_slope_y_fp )
24826         - \dim_to_fp:n {#2}
24827         + \dim_to_fp:n {#6}
24828       )
24829     /
24830     (
24831       \l__coffin_slope_x_fp -

```

```

24832                               \l__coffin_slope_y_fp
24833                           )
24834                       }
24835                   }
24836               \l__coffin_calculate_intersection_aux:nnnnnN
24837               { \l__coffin_x_dim }
24838               {#5} {#6} {#8} {#7} \l__coffin_y_dim
24839           }
24840       }
24841   }
24842 }
24843 }
24844 }

```

The formula for finding the intersection point is in most cases the same. The formula here is

$$\#6 = \#4 \cdot \left( \frac{\#1 - \#2}{\#5} \right) \#3$$

Thus #4 and #5 should be the directions of the pole while #2 and #3 are co-ordinates.

```

24845 \cs_new_protected:Npn \l__coffin_calculate_intersection_aux:nnnnnN
24846     #1#2#3#4#5#6
24847     {
24848         \dim_set:Nn #6
24849         {
24850             \fp_to_dim:n
24851             {
24852                 \dim_to_fp:n {#4} *
24853                 ( \dim_to_fp:n {#1} - \dim_to_fp:n {#2} ) /
24854                 \dim_to_fp:n {#5}
24855                 + \dim_to_fp:n {#3}
24856             }
24857         }
24858     }

```

(End definition for `\l__coffin_calculate_intersection:Nnn`, `\l__coffin_calculate_intersection:nnnnnnnN`, and `\l__coffin_calculate_intersection_aux:nnnnnN`.)

## 40.6 Aligning and typesetting of coffins

`\coffin_join:NnnNnnn`  
`\coffin_join:cnnNnnnn`  
`\coffin_join:Nnncnnnn`  
`\coffin_join:cnnccnnnn`  
`\coffin_gjoin:NnnNnnnN`  
`\coffin_gjoin:cnnNnnnn`  
`\coffin_gjoin:Nnncnnnn`  
`\coffin_gjoin:cnnccnnnn`  
`\l__coffin_join:NnnNnnnnN`

```

24859 \cs_new_protected:Npn \coffin_join:NnnNnnnn #1#2#3#4#5#6#7#8
24860     {
24861         \l__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
24862         \coffin_set_eq:NN
24863     }
24864 \cs_generate_variant:Nn \coffin_join:NnnNnnnn { c , Nnn , cnnc }
24865 \cs_new_protected:Npn \coffin_gjoin:NnnNnnnn #1#2#3#4#5#6#7#8
24866     {
24867         \l__coffin_join:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
24868         \coffin_gset_eq:NN
24869     }

```

```

24870 \cs_generate_variant:Nn \coffin_gjoin:NnnNnnnn { c , Nnnc , cnnc }
24871 \cs_new_protected:Npn \__coffin_join:NnnNnnnnN #1#2#3#4#5#6#7#8#9
24872 {
24873     \__coffin_align:NnnNnnnnN
24874     #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.

```

24875 \hbox_set:Nn \l__coffin_aligned_coffin
24876 {
24877     \dim_compare:nNnT { \l__coffin_offset_x_dim } < \c_zero_dim
24878         { \tex_kern:D -\l__coffin_offset_x_dim }
24879     \hbox_unpack:N \l__coffin_aligned_coffin
24880     \dim_set:Nn \l__coffin_internal_dim
24881         { \l__coffin_offset_x_dim - \box_wd:N #1 + \box_wd:N #4 }
24882     \dim_compare:nNnT \l__coffin_internal_dim < \c_zero_dim
24883         { \tex_kern:D -\l__coffin_internal_dim }
24884 }

```

The coffin structure is reset, and the corners are cleared: only those from the two parent coffins are needed.

```

24885 \__coffin_reset_structure:N \l__coffin_aligned_coffin
24886 \prop_clear:c
24887 {
24888     \coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
24889     \c_space_tl corners
24890 }
24891 \__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.

```

24892 \dim_compare:nNnTF \l__coffin_offset_x_dim < \c_zero_dim
24893 {
24894     \__coffin_offset_poles:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
24895     \__coffin_offset_poles:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
24896     \__coffin_offset_corners:Nnn #1 { -\l__coffin_offset_x_dim } { Opt }
24897     \__coffin_offset_corners:Nnn #4 { Opt } { \l__coffin_offset_y_dim }
24898 }
24899 {
24900     \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
24901     \__coffin_offset_poles:Nnn #4
24902         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
24903     \__coffin_offset_corners:Nnn #1 { Opt } { Opt }
24904     \__coffin_offset_corners:Nnn #4
24905         { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
24906 }
24907 \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
24908 #9 #1 \l__coffin_aligned_coffin
24909 }

```

(End definition for `\coffin_join:NnnNnnnn`, `\coffin_gjoin:NnnNnnnn`, and `\__coffin_join:NnnNnnnnN`. These functions are documented on page [234](#).)

\coffin\_attach:NnnNnnnn  
\coffin\_attach:cnnNnnnn  
\coffin\_attach:NnnCnnnn  
\coffin\_attach:cnnCnnnn  
\coffin\_gattach:NnnNnnnn  
\coffin\_gattach:cnnNnnnn  
\coffin\_gattach:NnnCnnnn  
\coffin\_gattach:cnnCnnnn  
\\_coffin\_attach:NnnNnnnnN  
  \\_coffin\_attach\_mark:NnnNnnnn

A more simple version of the above, as it simply uses the size of the first coffin for the new one. This means that the work here is rather simplified compared to the above code. The function used when marking a position is hear also as it is similar but without the structure updates.

```

24910 \cs_new_protected:Npn \coffin_attach:NnnNnnnn #1#2#3#4#5#6#7#8
24911 {
24912   \__coffin_attach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
24913   \coffin_set_eq:NN
24914 }
24915 \cs_generate_variant:Nn \coffin_attach:NnnNnnnn { c , Nnnc , cnnc }
24916 \cs_new_protected:Npn \coffin_gattach:NnnNnnnn #1#2#3#4#5#6#7#8
24917 {
24918   \__coffin_gattach:NnnNnnnnN #1 {#2} {#3} #4 {#5} {#6} {#7} {#8}
24919   \coffin_gset_eq:NN
24920 }
24921 \cs_generate_variant:Nn \coffin_gattach:NnnNnnnn { c , Nnnc , cnnc }
24922 \cs_new_protected:Npn \__coffin_attach:NnnNnnnnN #1#2#3#4#5#6#7#8#9
24923 {
24924   \__coffin_align:NnnNnnnnN
24925   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
24926   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
24927   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
24928   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
24929   \__coffin_reset_structure:N \l__coffin_aligned_coffin
24930   \prop_set_eq:cc
24931   {
24932     coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
24933     \c_space_tl corners
24934   }
24935   { coffin ~ \__coffin_to_value:N #1 ~ corners }
24936   \__coffin_update_poles:N \l__coffin_aligned_coffin
24937   \__coffin_offset_poles:Nnn #1 { Opt } { Opt }
24938   \__coffin_offset_poles:Nnn #4
24939   { \l__coffin_offset_x_dim } { \l__coffin_offset_y_dim }
24940   \__coffin_update_vertical_poles:NNN #1 #4 \l__coffin_aligned_coffin
24941   \coffin_set_eq:NN #1 \l__coffin_aligned_coffin
24942 }
24943 \cs_new_protected:Npn \__coffin_attach_mark:NnnNnnnn #1#2#3#4#5#6#7#8
24944 {
24945   \__coffin_align:NnnNnnnnN
24946   #1 {#2} {#3} #4 {#5} {#6} {#7} {#8} \l__coffin_aligned_coffin
24947   \box_set_ht:Nn \l__coffin_aligned_coffin { \box_ht:N #1 }
24948   \box_set_dp:Nn \l__coffin_aligned_coffin { \box_dp:N #1 }
24949   \box_set_wd:Nn \l__coffin_aligned_coffin { \box_wd:N #1 }
24950   \box_set_eq:NN #1 \l__coffin_aligned_coffin
24951 }
```

(End definition for \coffin\_attach:NnnNnnnn and others. These functions are documented on page 234.)

\\_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.

```

24952 \cs_new_protected:Npn \__coffin_align:NnnNnnnnN #1#2#3#4#5#6#7#8#9
24953   {
24954     \__coffin_calculate_intersection:Nnn #4 {#5} {#6}
24955     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
24956     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
24957     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
24958     \dim_set:Nn \l__coffin_offset_x_dim
24959       { \l__coffin_x_dim - \l__coffin_x_prime_dim + #7 }
24960     \dim_set:Nn \l__coffin_offset_y_dim
24961       { \l__coffin_y_dim - \l__coffin_y_prime_dim + #8 }
24962     \hbox_set:Nn \l__coffin_aligned_internal_coffin
24963   {
24964     \box_use:N #1
24965     \tex_kern:D -\box_wd:N #1
24966     \tex_kern:D \l__coffin_offset_x_dim
24967     \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #4 }
24968   }
24969   \coffin_set_eq:NN #9 \l__coffin_aligned_internal_coffin
24970 }
```

(End definition for `\__coffin_align:NnnNnnnnN`.)

`\__coffin_offset_poles:Nnn` `\__coffin_offset_pole:Nnnnnnn` 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 to 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.

```

24971 \cs_new_protected:Npn \__coffin_offset_poles:Nnn #1#2#3
24972   {
24973     \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ poles }
24974       { \__coffin_offset_pole:Nnnnnnn #1 {##1} ##2 {##2} {##3} }
24975   }
24976 \cs_new_protected:Npn \__coffin_offset_pole:Nnnnnnn #1#2#3#4#5#6#7#8
24977   {
24978     \dim_set:Nn \l__coffin_x_dim { #3 + #7 }
24979     \dim_set:Nn \l__coffin_y_dim { #4 + #8 }
24980     \tl_if_in:nTF {##2} { - }
24981       { \tl_set:Nn \l__coffin_internal_tl { {##2} } }
24982       { \tl_set:Nn \l__coffin_internal_tl { { #1 - #2 } } }
24983     \exp_last_unbraced:NNo \__coffin_set_pole:Nnx \l__coffin_aligned_coffin
24984       { \l__coffin_internal_tl }
24985       {
24986         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
24987         {##5} {##6}
24988       }
24989 }
```

(End definition for `\__coffin_offset_poles:Nnn` and `\__coffin_offset_pole:Nnnnnnn`.)

`\__coffin_offset_corners:Nnn` 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.

```

24990 \cs_new_protected:Npn \__coffin_offset_corners:Nnn #1#2#3
24991 {
24992     \prop_map_inline:cn { coffin ~ \__coffin_to_value:N #1 ~ corners }
24993         { \__coffin_offset_corner:Nnnnn #1 {##1} ##2 {#2} {#3} }
24994     }
24995 \cs_new_protected:Npn \__coffin_offset_corner:Nnnnn #1#2#3#4#5#6
24996 {
24997     \prop_put:cnx
24998     {
24999         coffin ~ \__coffin_to_value:N \l__coffin_aligned_coffin
25000             \c_space_tl corners
25001     }
25002     { #1 - #2 }
25003     {
25004         { \dim_eval:n { #3 + #5 } }
25005         { \dim_eval:n { #4 + #6 } }
25006     }
25007 }
```

(End definition for `\__coffin_offset_corners:Nnn` and `\__coffin_offset_corner:Nnnnn`.)

`\__coffin_update_vertical_poles:NNN` 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.

```

25008 \cs_new_protected:Npn \__coffin_update_vertical_poles:NNN #1#2#3
25009 {
25010     \__coffin_get_pole:NnN #3 { #1 -T } \l__coffin_pole_a_t1
25011     \__coffin_get_pole:NnN #3 { #2 -T } \l__coffin_pole_b_t1
25012     \exp_last_two_unbraced:Noo \__coffin_update_T:nnnnnnnnN
25013         \l__coffin_pole_a_t1 \l__coffin_pole_b_t1 #3
25014     \__coffin_get_pole:NnN #3 { #1 -B } \l__coffin_pole_a_t1
25015     \__coffin_get_pole:NnN #3 { #2 -B } \l__coffin_pole_b_t1
25016     \exp_last_two_unbraced:Noo \__coffin_update_B:nnnnnnnnN
25017         \l__coffin_pole_a_t1 \l__coffin_pole_b_t1 #3
25018 }
25019 \cs_new_protected:Npn \__coffin_update_T:nnnnnnnnN #1#2#3#4#5#6#7#8#9
25020 {
25021     \dim_compare:nNnTF {#2} < {#6}
25022     {
25023         \__coffin_set_pole:Nnx #9 { T }
25024             { { Opt } {#6} { 1000pt } { Opt } }
25025     }
25026     {
25027         \__coffin_set_pole:Nnx #9 { T }
25028             { { Opt } {#2} { 1000pt } { Opt } }
25029     }
25030 }
25031 \cs_new_protected:Npn \__coffin_update_B:nnnnnnnnN #1#2#3#4#5#6#7#8#9
25032 {
25033     \dim_compare:nNnTF {#2} < {#6}
25034     {
25035         \__coffin_set_pole:Nnx #9 { B }
```

```

25036      { { Opt } {#2} { 1000pt } { Opt } }
25037  }
25038  {
25039    \__coffin_set_pole:Nnx #9 { B }
25040    { { Opt } {#6} { 1000pt } { Opt } }
25041  }
25042 }

(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.

```

25043 \coffin_new:N \c\_coffin_empty_coffin
25044 \tex_setbox:D \c\_coffin_empty_coffin = \tex_hbox:D { }

(End definition for \c\_coffin_empty_coffin.)

```

**\coffin\_typeset:Nnnnn** 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.

```

25045 \cs_new_protected:Npn \coffin_typeset:Nnnnn #1#2#3#4#5
25046  {
25047    \mode_leave_vertical:
25048    \__coffin_align:NnnNnnnN \c\_coffin_empty_coffin { H } { 1 }
25049    #1 {#2} {#3} {#4} {#5} \l__coffin_aligned_coffin
25050    \box_use_drop:N \l__coffin_aligned_coffin
25051  }
25052 \cs_generate_variant:Nn \coffin_typeset:Nnnnn { c }

(End definition for \coffin_typeset:Nnnnn. This function is documented on page 234.)

```

## 40.7 Coffin diagnostics

\l\\_coffin\\_display\\_coffin Used for printing coffins with data structures attached.

```

25053 \coffin_new:N \l\_coffin_display_coffin
25054 \coffin_new:N \l\_coffin_display_coord_coffin
25055 \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.

```

25056 \prop_new:N \l\_coffin_display_handles_prop
25057 \prop_put:Nnn \l\_coffin_display_handles_prop { tl }
25058  { { b } { r } { -1 } { 1 } }
25059 \prop_put:Nnn \l\_coffin_display_handles_prop { thc }
25060  { { b } { hc } { 0 } { 1 } }
25061 \prop_put:Nnn \l\_coffin_display_handles_prop { tr }
25062  { { b } { 1 } { 1 } { 1 } }
25063 \prop_put:Nnn \l\_coffin_display_handles_prop { vcl }
25064  { { vc } { r } { -1 } { 0 } }
25065 \prop_put:Nnn \l\_coffin_display_handles_prop { vchc }
25066  { { vc } { hc } { 0 } { 0 } }
25067 \prop_put:Nnn \l\_coffin_display_handles_prop { vcr }

```

```

25068 { { vc } { 1 } { 1 } { 0 } }
25069 \prop_put:Nnn \l__coffin_display_handles_prop { bl }
25070 { { t } { r } { -1 } { -1 } }
25071 \prop_put:Nnn \l__coffin_display_handles_prop { bhc }
25072 { { t } { hc } { 0 } { -1 } }
25073 \prop_put:Nnn \l__coffin_display_handles_prop { br }
25074 { { t } { 1 } { 1 } { -1 } }
25075 \prop_put:Nnn \l__coffin_display_handles_prop { Tl }
25076 { { t } { r } { -1 } { -1 } }
25077 \prop_put:Nnn \l__coffin_display_handles_prop { Thc }
25078 { { t } { hc } { 0 } { -1 } }
25079 \prop_put:Nnn \l__coffin_display_handles_prop { Tr }
25080 { { t } { 1 } { 1 } { -1 } }
25081 \prop_put:Nnn \l__coffin_display_handles_prop { Hl }
25082 { { vc } { r } { -1 } { 1 } }
25083 \prop_put:Nnn \l__coffin_display_handles_prop { Hhc }
25084 { { vc } { hc } { 0 } { 1 } }
25085 \prop_put:Nnn \l__coffin_display_handles_prop { Hr }
25086 { { vc } { 1 } { 1 } { 1 } }
25087 \prop_put:Nnn \l__coffin_display_handles_prop { Bl }
25088 { { b } { r } { -1 } { -1 } }
25089 \prop_put:Nnn \l__coffin_display_handles_prop { Bhc }
25090 { { b } { hc } { 0 } { -1 } }
25091 \prop_put:Nnn \l__coffin_display_handles_prop { Br }
25092 { { 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.

```

25093 \dim_new:N \l__coffin_display_offset_dim
25094 \dim_set:Nn \l__coffin_display_offset_dim { 2pt }

```

(End definition for `\l__coffin_display_offset_dim`.)

`\l__coffin_display_x_dim` As the intersections of poles have to be calculated to find which ones to print, there is  
`\l__coffin_display_y_dim` a need to avoid repetition. This is done by saving the intersection into two dedicated values.

```

25095 \dim_new:N \l__coffin_display_x_dim
25096 \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.

```
25097 \prop_new:N \l__coffin_display_poles_prop
```

(End definition for `\l__coffin_display_poles_prop`.)

`\l__coffin_display_font_tl` Stores the settings used to print coffin data: this keeps things flexible.

```

25098 \tl_new:N \l__coffin_display_font_tl
25099 {*initex}
25100 \tl_set:Nn \l__coffin_display_font_tl { } % TODO
25101 
```

```

25102 {*package}
25103 \tl_set:Nn \l__coffin_display_font_tl { \sffamily \tiny }
25104 
```

(End definition for `\l_coffin_display_font_t1`.)

`\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.

```
25105 \cs_new_protected:Npn \coffin_mark_handle:Nnnn #1#2#3#4
25106  {
25107      \hcoffin_set:Nn \l_coffin_display_pole_coffin
25108      {
25109         /*initex}
25110         \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
25111     
```

`\*package`

`\color {#4}`

`\rule { 1pt } { 1pt }`

`\/package)`

`}`

`\_coffin_attach_mark:NnnNnnnn #1 {#2} {#3}`

`\l_coffin_display_pole_coffin { hc } { vc } { Opt } { Opt }`

`\hcoffin_set:Nn \l_coffin_display_coord_coffin`

`{`

`\*initex}`

`% TODO`

`\/initex}`

`\*package`

`\color {#4}`

`\/package)`

`\l_coffin_display_font_t1`

`( \tl_to_str:n { #2 , #3 } )`

`}`

`\prop_get:NnN \l_coffin_display_handles_prop`

`{ #2 #3 } \l_coffin_internal_tl`

`\quark_if_no_value:NTF \l_coffin_internal_tl`

`{`

`\prop_get:NnN \l_coffin_display_handles_prop`

`{ #3 #2 } \l_coffin_internal_tl`

`\quark_if_no_value:NTF \l_coffin_internal_tl`

`{`

`\_coffin_attach_mark:NnnNnnnn #1 {#2} {#3}`

`\l_coffin_display_coord_coffin { 1 } { vc }`

`{ 1pt } { Opt }`

`}`

`{`

`\exp_last_unbraced:No \_coffin_mark_handle_aux:nnnnNnn`

`\l_coffin_internal_tl #1 {#2} {#3}`

`}`

`}`

`{`

`\exp_last_unbraced:No \_coffin_mark_handle_aux:nnnnNnn`

`\l_coffin_internal_tl #1 {#2} {#3}`

`}`

`}`

`\cs_new_protected:Npn \_coffin_mark_handle_aux:nnnnNnn #1#2#3#4#5#6#7`

```

25153  {
25154    \__coffin_attach_mark:NnnNnnnn #5 {#6} {#7}
25155    \l__coffin_display_coord_coffin {#1} {#2}
25156    { #3 \l__coffin_display_offset_dim }
25157    { #4 \l__coffin_display_offset_dim }
25158  }
25159 \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 235.)

```

**\coffin\_display\_handles:Nn** 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.

```

25160 \cs_new_protected:Npn \coffin_display_handles:Nn #1#2
25161  {
25162    \hcoffin_set:Nn \l__coffin_display_pole_coffin
25163    {
25164      /*initex*/
25165      \hbox:n { \tex_vrule:D width 1pt height 1pt \scan_stop: } % TODO
25166    
```

/\*package\*/

25168     \color {#2}

25169     \rule { 1pt } { 1pt }

25170

/\*package\*/

25171     }

25172     \prop\_set\_eq:Nc \l\_\_coffin\_display\_poles\_prop

25173         { coffin ~ \\_\_coffin\_to\_value:N #1 ~ poles }

25174         \\_\_coffin\_get\_pole:NnN #1 { H } \l\_\_coffin\_pole\_a\_tl

25175         \\_\_coffin\_get\_pole:NnN #1 { T } \l\_\_coffin\_pole\_b\_tl

25176         \tl\_if\_eq:NNT \l\_\_coffin\_pole\_a\_tl \l\_\_coffin\_pole\_b\_tl

25177             { \prop\_remove:Nn \l\_\_coffin\_display\_poles\_prop { T } }

25178         \\_\_coffin\_get\_pole:NnN #1 { B } \l\_\_coffin\_pole\_b\_tl

25179         \tl\_if\_eq:NNT \l\_\_coffin\_pole\_a\_tl \l\_\_coffin\_pole\_b\_tl

25180             { \prop\_remove:Nn \l\_\_coffin\_display\_poles\_prop { B } }

25181         \coffin\_set\_eq:NN \l\_\_coffin\_display\_coffin #1

25182         \prop\_map\_inline:Nn \l\_\_coffin\_display\_poles\_prop

25183             {

25184                 \prop\_remove:Nn \l\_\_coffin\_display\_poles\_prop {##1}

25185                 \\_\_coffin\_display\_handles\_aux:nnnnnn {##1} ##2 {##2}

25186             }

25187         \box\_use\_drop:N \l\_\_coffin\_display\_coffin

25188 }

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.

```

25189 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnnnn #1#2#3#4#5#6
25190  {
25191    \prop_map_inline:Nn \l__coffin_display_poles_prop
25192    {
25193      \bool_set_false:N \l__coffin_error_bool
25194      \__coffin_calculate_intersection:nnnnnnnn {#2} {#3} {#4} {#5} ##2

```

```

25195     \bool_if:NF \l__coffin_error_bool
25196     {
25197         \dim_set:Nn \l__coffin_display_x_dim { \l__coffin_x_dim }
25198         \dim_set:Nn \l__coffin_display_y_dim { \l__coffin_y_dim }
25199         \__coffin_display_attach:Nnnnn
25200             \l__coffin_display_pole_coffin { hc } { vc }
25201             { Opt } { Opt }
25202         \hcoffin_set:Nn \l__coffin_display_coord_coffin
25203             {
25204             (*initex)
25205                 % TODO
25206             (/initex)
25207             (*package)
25208                 \color {#6}
25209             (/package)
25210                 \l__coffin_display_font_tl
25211                     ( \tl_to_str:n { #1 , ##1 } )
25212             }
25213         \prop_get:NnN \l__coffin_display_handles_prop
25214             { #1 ##1 } \l__coffin_internal_tl
25215         \quark_if_no_value:NTF \l__coffin_internal_tl
25216             {
25217                 \prop_get:NnN \l__coffin_display_handles_prop
25218                     { ##1 #1 } \l__coffin_internal_tl
25219                 \quark_if_no_value:NTF \l__coffin_internal_tl
25220                     {
25221                         \__coffin_display_attach:Nnnnn
25222                             \l__coffin_display_coord_coffin { 1 } { vc }
25223                             { 1pt } { Opt }
25224                     }
25225             {
25226                 \exp_last_unbraced:No
25227                     \__coffin_display_handles_aux:nnnn
25228                         \l__coffin_internal_tl
25229                     }
25230             }
25231             {
25232                 \exp_last_unbraced:No \__coffin_display_handles_aux:nnnn
25233                     \l__coffin_internal_tl
25234                 }
25235             }
25236         }
25237     }
25238 \cs_new_protected:Npn \__coffin_display_handles_aux:nnnn #1#2#3#4
25239     {
25240         \__coffin_display_attach:Nnnnn
25241             \l__coffin_display_coord_coffin {#1} {#2}
25242             { #3 \l__coffin_display_offset_dim }
25243             { #4 \l__coffin_display_offset_dim }
25244     }
25245 \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.

```

25246 \cs_new_protected:Npn \__coffin_display_attach:Nnnnn #1#2#3#4#5
25247 {
25248     \__coffin_calculate_intersection:Nnn #1 {#2} {#3}
25249     \dim_set:Nn \l__coffin_x_prime_dim { \l__coffin_x_dim }
25250     \dim_set:Nn \l__coffin_y_prime_dim { \l__coffin_y_dim }
25251     \dim_set:Nn \l__coffin_offset_x_dim
25252         { \l__coffin_display_x_dim - \l__coffin_x_prime_dim + #4 }
25253     \dim_set:Nn \l__coffin_offset_y_dim
25254         { \l__coffin_display_y_dim - \l__coffin_y_prime_dim + #5 }
25255     \hbox_set:Nn \l__coffin_aligned_coffin
25256     {
25257         \box_use:N \l__coffin_display_coffin
25258         \tex_kern:D -\box_wd:N \l__coffin_display_coffin
25259         \tex_kern:D \l__coffin_offset_x_dim
25260         \box_move_up:nn { \l__coffin_offset_y_dim } { \box_use:N #1 }
25261     }
25262     \box_set_ht:Nn \l__coffin_aligned_coffin
25263         { \box_ht:N \l__coffin_display_coffin }
25264     \box_set_dp:Nn \l__coffin_aligned_coffin
25265         { \box_dp:N \l__coffin_display_coffin }
25266     \box_set_wd:Nn \l__coffin_aligned_coffin
25267         { \box_wd:N \l__coffin_display_coffin }
25268     \box_set_eq:NN \l__coffin_display_coffin \l__coffin_aligned_coffin
25269 }
```

(End definition for `\coffin_display_handles:Nn` and others. This function is documented on page 235.)

**\coffin\_show\_structure:N** 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.

```

25270 \cs_new_protected:Npn \coffin_show_structure:N
25271     { \__coffin_show_structure>NN \msg_show:nnxxxx }
25272 \cs_generate_variant:Nn \coffin_show_structure:N { c }
25273 \cs_new_protected:Npn \coffin_log_structure:N
25274     { \__coffin_show_structure>NN \msg_log:nnxxxx }
25275 \cs_generate_variant:Nn \coffin_log_structure:N { c }
25276 \cs_new_protected:Npn \__coffin_show_structure>NN #1#2
25277     {
25278         \__coffin_if_exist:NT #2
25279         {
25280             #1 { LaTeX / kernel } { show-coffin }
25281             { \token_to_str:N #2 }
25282             {
25283                 \iow_newline: >~ ht ~== \dim_eval:n { \coffin_ht:N #2 }
25284                 \iow_newline: >~ dp ~== \dim_eval:n { \coffin_dp:N #2 }
25285                 \iow_newline: >~ wd ~== \dim_eval:n { \coffin_wd:N #2 }
25286             }
25287             {
25288                 \prop_map_function:cN
25289                     { coffin ~ \__coffin_to_value:N #2 ~ poles }
25290                     \msg_show_item_unbraced:nn
25291             }
25292         }
```

```

25293     }
25294 }

```

(End definition for `\coffin_show_structure:N`, `\coffin_log_structure:N`, and `\_coffin_show_-structure:NN`. These functions are documented on page 235.)

## 40.8 Messages

```

25295 \_kernel_msg_new:nnnn { kernel } { no-pole-intersection }
25296   { No~intersection~between~coffin~poles. }
25297   {
25298     LaTeX~was~asked~to~find~the~intersection~between~two~poles,~
25299     but~they~do~not~have~a~unique~meeting~point:~
25300     the~value~(Opt,~Opt)~will~be~used.
25301   }
25302 \_kernel_msg_new:nnnn { kernel } { unknown-coffin }
25303   { Unknown~coffin-'#1'. }
25304   { The~coffin-'#1'~was~never~defined. }
25305 \_kernel_msg_new:nnnn { kernel } { unknown-coffin-pole }
25306   { Pole-'#1'~unknown~for~coffin-'#2'. }
25307   {
25308     LaTeX~was~asked~to~find~a~typesetting~pole~for~a~coffin,~
25309     but~either~the~coffin~does~not~exist~or~the~pole~name~is~wrong.
25310   }
25311 \_kernel_msg_new:nnn { kernel } { show-coffin }
25312   {
25313     Size~of~coffin~#1 : #2 \\
25314     Poles~of~coffin~#1 : #3 .
25315   }
25316 </initex | package>

```

## 41 I3color-base Implementation

```

25317 <*initex | package>
25318 <@=color>

```

`\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` is space-separated as this allows it to be used directly in specials in many common cases. This internal representation is close to that used by the `dvips` program.

(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.

```
25319 \cs_new_eq:NN \color_group_begin: \group_begin:
25320 \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 237.)

`\color_ensure_current:` A driver-independent wrapper for setting the foreground color to the current color “now”.

```
25321 \cs_new_protected:Npn \color_ensure_current:
25322 {
25323   {*package}
25324     \driver_color_pickup:N \l_color_current_tl
25325   /package)
25326   \__color_select:V \l_color_current_tl
25327 }
```

(End definition for `\color_ensure_current:`. This function is documented on page 237.)

`\__color_select:n` `\__color_select:V` `\__color_select:w` 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 experimental material.

```
25328 \cs_new_protected:Npn \__color_select:n #1
25329   { \__color_select:w #1 \q_stop }
25330 \cs_generate_variant:Nn \__color_select:n { V }
25331 \cs_new_protected:Npn \__color_select:w #1 ~ #2 \q_stop
25332   { \use:c { __color_select_ #1 :w } #2 \q_stop }
25333 \cs_new_protected:Npn \__color_select_cmyk:w #1 ~ #2 ~ #3 ~ #4 \q_stop
25334   { \driver_color_cmyk:nnnn {#1} {#2} {#3} {#4} }
25335 \cs_new_protected:Npn \__color_select_gray:w #1 \q_stop
25336   { \driver_color_gray:n {#1} }
25337 \cs_new_protected:Npn \__color_select_rgb:w #1 ~ #2 ~ #3 \q_stop
25338   { \driver_color_rgb:nnn {#1} {#2} {#3} }
25339 \cs_new_protected:Npn \__color_select_spot:w #1 ~ #2 \q_stop
25340   { \driver_color_spot:nn {#1} {#2} }
```

(End definition for `\__color_select:n` and others.)

`\l_color_current_tl` As the setting data is used only for specials, and those are always space-separated, it makes most sense to hold the internal information in that form.

```
25341 \tl_new:N \l_color_current_tl
25342 \tl_set:Nn \l_color_current_tl { gray~0 }
```

(End definition for `\l_color_current_tl`.)

```
25343 </initex | package>
```

## 42 I3luateX implementation

25344  $\langle *initex \mid package \rangle$

### 42.1 Breaking out to Lua

25345  $\langle *tex \rangle$

25346  $\langle @@=lua \rangle$

$\_\_lua\_escape:n$  Copies of primitives.

25347  $\backslash cs\_new\_eq:NN \_\_lua\_escape:n \backslash tex\_luaescapestring:D$

25348  $\backslash cs\_new\_eq:NN \_\_lua\_now:n \backslash tex\_directlua:D$

25349  $\backslash cs\_new\_eq:NN \_\_lua\_shipout:n \backslash tex\_latelua:D$

(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.

25350  $\backslash cs\_undefine:N \backslash lua\_escape:e$

25351  $\backslash cs\_undefine:N \backslash lua\_now:e$

$\backslash lua\_now:n$  Wrappers around the primitives. As with engines other than LuaTeX these have to be

$\backslash lua\_now:e$  macros, we give them the same status in all cases. When LuaTeX is not in use, simply

$\backslash lua\_shipout\_e:n$  give an error message/

25352  $\backslash cs\_new:Npn \backslash lua\_now:e #1 \{ \_\_lua\_now:n \{ \#1 \} \}$

25353  $\backslash cs\_new:Npn \backslash lua\_now:n #1 \{ \backslash lua\_now:e \{ \backslash exp\_not:n \{ \#1 \} \} \}$

25354  $\backslash cs\_new\_protected:Npn \backslash lua\_shipout\_e:n #1 \{ \_\_lua\_shipout:n \{ \#1 \} \}$

25355  $\backslash cs\_new\_protected:Npn \backslash lua\_shipout:n #1$

25356  $\{ \backslash lua\_shipout\_e:n \{ \backslash exp\_not:n \{ \#1 \} \} \}$

25357  $\backslash cs\_new:Npn \backslash lua\_escape:e #1 \{ \_\_lua\_escape:n \{ \#1 \} \}$

25358  $\backslash cs\_new:Npn \backslash lua\_escape:n #1 \{ \backslash lua\_escape:e \{ \backslash exp\_not:n \{ \#1 \} \} \}$

25359  $\backslash sys\_if\_engine\_luatex:F$

25360  $\{$

25361  $\backslash clist\_map\_inline:nn$

25362  $\{$

25363  $\backslash lua\_escape:n , \backslash lua\_escape:e ,$

25364  $\backslash lua\_now:n , \backslash lua\_now:e$

25365  $\}$

25366  $\{$

25367  $\backslash cs\_set:Npn \#1 \#\#\#1$

25368  $\{$

25369  $\backslash \_\_kernel\_msg\_expandable\_error:nnn$

25370  $\{ kernel \} \{ luatex-required \} \{ \#1 \}$

25371  $\}$

25372  $\}$

25373  $\backslash clist\_map\_inline:nn$

25374  $\{ \backslash lua\_shipout\_e:n , \backslash lua\_shipout:n \}$

25375  $\{$

25376  $\backslash cs\_set\_protected:Npn \#1 \#\#\#1$

25377  $\{$

25378  $\backslash \_\_kernel\_msg\_error:nnn$

25379  $\{ kernel \} \{ luatex-required \} \{ \#1 \}$

25380  $\}$

25381  $\}$

25382  $\}$

(End definition for  $\backslash lua\_now:n$  and others. These functions are documented on page 238.)

## 42.2 Messages

```
25383 \__kernel_msg_new:nnnn { kernel } { luatex-required }
25384   { LuATeX-engine-not-in-use!~Ignoring~#1. }
25385   {
25386     The-feature-you-are-using-is-only-available-
25387     with-the-LuATeX-engine.-LaTeX3-ignored~'#1'.
25388 }
```

## 42.3 Deprecated functions

\lua\_now\_x:n For removal after 2019-12-31.  
\lua\_escape\_x:n  
\lua\_shipout\_x:n

```
25389 \__kernel_patch_deprecation:nnNpn { 2019-12-31 } { \lua_now:e }
25390 \cs_new:Npn \lua_now_x:n #1 { \__lua_now:n {#1} }
25391 \__kernel_patch_deprecation:nnNpn { 2019-12-31 } { \lua_escape:e }
25392 \cs_new:Npn \lua_escape_x:n #1 { \__lua_escape:n {#1} }
25393 \__kernel_patch_deprecation:nnNpn { 2019-12-31 } { \lua_shipout_e:n }
25394 \cs_new_protected:Npn \lua_shipout_x:n #1 { \__lua_shipout:n {#1} }

(End definition for \lua_now_x:n, \lua_escape_x:n, and \lua_shipout_x:n.)
```

25395 ⟨/tex⟩

## 42.4 Lua functions for internal use

```
25396 (*lua)
```

Most of the emulation of pdftEX here is based heavily on Heiko Oberdiek's pdfex-cmds package.

**13kernel** Create a table for the kernel's own use.

```
25397 13kernel = 13kernel or { }
```

(End definition for 13kernel. This function is documented on page 239.)

Local copies of global tables.

```
25398 local io      = io
25399 local kpse    = kpse
25400 local lfs     = lfs
25401 local math    = math
25402 local md5    = md5
25403 local os      = os
25404 local string  = string
25405 local tex     = tex
25406 local unicode = unicode
```

Local copies of standard functions.

```
25407 local abs      = math.abs
25408 local byte     = string.byte
25409 local floor    = math.floor
25410 local format   = string.format
25411 local gsub     = string.gsub
25412 if not resolvers then
25413   local kpse_find = kpse.find_file
25414 end
25415 local lfs_attr  = lfs.attributes
25416 local md5_sum   = md5.sum
25417 local open      = io.open
```

```

25418 local os_clock = os.clock
25419 local os_date = os.date
25420 local setcatcode = tex.setcatcode
25421 local sprint = tex.sprint
25422 local write = tex.write
25423 local utf8_char = unicode.utf8.char

    Deal with ConTEXt: doesn't use kpse library.

25424 if resolvers then
25425   kpse_find = resolvers.findfile
25426 end

```

`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.

```

25427 local function escapehex(str)
25428   write((gsub(str, ".",
25429     function (ch) return format("%02X", byte(ch)) end)))
25430 end

```

(End definition for `escapehex`.)

`l3kernel.charcat` Creating arbitrary chars needs a category code table. As set up here, one may have been assigned earlier (see `l3bootstrap`) or a hard-coded one is used. The latter is intended for format mode and should be adjusted to match an eventual allocator.

```

25431 local charcat_table = l3kernel.charcat_table or 1
25432 local function charcat(charcode, catcode)
25433   setcatcode(charcat_table, charcode, catcode)
25434   sprint(charcat_table, utf8_char(charcode))
25435 end
25436 l3kernel.charcat = charcat

```

(End definition for `l3kernel.charcat`. This function is documented on page [239](#).)

`l3kernel.elapsedtime` Simple timing set up: give the result from the system clock in scaled seconds.

```

13kernel.resettimer
25437 local base_time = 0
25438 local function elapsedtime()
25439   local val = (os_clock() - base_time) * 65536 + 0.5
25440   if val > 2147483647 then
25441     val = 2147483647
25442   end
25443   write(format("%d", floor(val)))
25444 end
25445 l3kernel.elapsedtime = elapsedtime
25446 local function resettimer()
25447   base_time = 0
25448 end
25449 l3kernel.resettimer = resettimer

```

(End definition for `l3kernel.elapsedtime` and `l3kernel.resettimer`. These functions are documented on page [239](#).)

**13kernel.filemdfivesum**

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.

```

25450 local function filemdfivesum(name)
25451   local file = kpse_find(name, "tex", true)
25452   if file then
25453     local f = open(file, "rb")
25454     if f then
25455       local data = f:read("*a")
25456       escapehex(md5_sum(data))
25457       f:close()
25458     end
25459   end
25460 end
25461 13kernel.filemdfivesum = filemdfivesum

```

(End definition for `13kernel.filemdfivesum`. This function is documented on page [239](#).)

**13kernel.filemoddate**

See procedure `makepdftime` in `utils.c` of `pdftEX`.

```

25462 local function filemoddate(name)
25463   local file = kpse_find(name, "tex", true)
25464   if file then
25465     local date = lfs_attr(file, "modification")
25466     if date then
25467       local d = os_date("*t", date)
25468       if d.sec >= 60 then
25469         d.sec = 59
25470       end
25471       local u = os_date("!*t", date)
25472       local off = 60 * (d.hour - u.hour) + d.min - u.min
25473       if d.year ~= u.year then
25474         if d.year > u.year then
25475           off = off + 1440
25476         else
25477           off = off - 1440
25478         end
25479       elseif d.yday ~= u.yday then
25480         if d.yday > u.yday then
25481           off = off + 1440
25482         else
25483           off = off - 1440
25484         end
25485       end
25486       local timezone
25487       if off == 0 then
25488         timezone = "Z"
25489       else
25490         local hours = floor(off / 60)
25491         local mins = abs(off - hours * 60)
25492         timezone = format("%+03d", hours)
25493         .. "," .. format("%02d", mins) .. ","
25494       end
25495       write("D:"

```

```

25496     .. format("%04d", d.year)
25497     .. format("%02d", d.month)
25498     .. format("%02d", d.day)
25499     .. format("%02d", d.hour)
25500     .. format("%02d", d.min)
25501     .. format("%02d", d.sec)
25502     .. timezone)
25503   end
25504 end
25505 end
25506 l3kernel.filemoddate = filemoddate

```

(End definition for `l3kernel.filemoddate`. This function is documented on page 239.)

**l3kernel.filesize** A simple disk lookup.

```

25507 local function filesize(name)
25508   local file = kpse_find(name, "tex", true)
25509   if file then
25510     local size = lfs_attr(file, "size")
25511     if size then
25512       write(size)
25513     end
25514   end
25515 end
25516 l3kernel.filesize = filesize

```

(End definition for `l3kernel.filesize`. This function is documented on page 239.)

**l3kernel.strcmp** String comparison which gives the same results as pdfTEX's \pdfstrcmp, although the ordering should likely not be relied upon!

```

25517 local function strcmp(A, B)
25518   if A == B then
25519     write("0")
25520   elseif A < B then
25521     write("-1")
25522   else
25523     write("1")
25524   end
25525 end
25526 l3kernel.strcmp = strcmp

```

(End definition for `l3kernel.strcmp`. This function is documented on page 239.)

## 42.5 Generic Lua and font support

```

25527 {*initex}
25528 (@@=alloc)

```

A small amount of generic code is used by almost all LuaTEX material so needs to be loaded by the format.

```

25529 attribute_count_name = "g__alloc_attribute_int"
25530 bytecode_count_name = "g__alloc_bytecode_int"
25531 chunkname_count_name = "g__alloc_chunkname_int"
25532 whatsit_count_name = "g__alloc_whatsit_int"
25533 require("ltluatex")

```

With the above available the font loader code used by plain T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> when used with LuaT<sub>E</sub>X can be loaded here. This is thus being treated more-or-less as part of the engine itself.

```

25534 require("luaotfload-main")
25535 local _void = luaotfload.main()
25536 </initex>
25537 </lua>
25538 </initex | package>
```

## 43 I3unicode implementation

```

25539 <*initex | package>
25540 <@=char>
```

Case changing both for strings and “text” requires data from the Unicode Consortium. Some of this is build in to the format (as `\lccode` and `\uccode` values) but this covers only the simple one-to-one situations and does not fully handle for example case folding.

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. For performance reasons, some of the code here is very low-level: the material is read during loading `expl3` in package mode.

```

25541 \ior_new:N \g__char_data_ior
25542 \bool_lazy_or:nnTF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: }
25543 {
25544     \group_begin:
```

Set up a private copy of the char-generation primitive.

```
25545     \cs_set_eq:NN \__char_generate:w \tex_Uchar:D
```

A fast local implementation for generating characters.

```

25546     \cs_set:Npx \__char_generate:n #1
25547     {
25548         \sys_if_engine_luatex:TF
25549         {
25550             \exp_not:N \lua_now:e
25551             {
25552                 13kernel.charcat
25553                 (
25554                     \exp_not:N \tex_number:D #1 ,
25555                     \exp_not:N \tex_the:D \tex_catcode:D #1
25556                 )
25557             }
25558         }
25559         {
25560             \exp_not:N \tex_Ucharcat:D
25561             #1 ~
25562             \tex_catcode:D #1 ~
25563         }
25564     }
```

Parse the main Unicode data file for title case exceptions (the one-to-one lower and upper case mappings it contains are all be covered by the `\TeX` data). There are no comments in the main data file so this can be done using a standard mapping and no checks.

```

25565   \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
25566   \cs_set_protected:Npn \__char_data_auxi:w
25567     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 ; #8 ; #9 ;
25568     { \__char_data_auxii:w #1 ; }
25569   \cs_set_protected:Npn \__char_data_auxii:w
25570     #1 ; #2 ; #3 ; #4 ; #5 ; #6 ; #7 \q_stop
25571   {
25572     \cs_set_nopar:Npn \l__char_tmpa_tl {#7}
25573     \reverse_if:N \if_meaning:w \l__char_tmpa_tl \c_empty_tl
25574       \cs_set_nopar:Npn \l__char_tmpb_tl {#5}
25575       \reverse_if:N \if_meaning:w \l__char_tmpa_tl \l__char_tmpb_tl
25576         \tl_const:cx
25577           { c__char_mixed_case_ \__char_generate:w "#1 _tl }
25578           { \__char_generate:n { "#7 } }
25579     \fi:
25580   \fi:
25581   }
25582 \ior_map_variable>NNn \g__char_data_ior \l__char_tmpa_tl
25583   {
25584     \if_meaning:w \l__char_tmpa_tl \c_space_tl
25585       \exp_after:wN \ior_map_break:
25586     \fi:
25587     \exp_after:wN \__char_data_auxi:w \l__char_tmpa_tl \q_stop
25588   }
25589 \ior_close:N \g__char_data_ior

```

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.

```

25590   \ior_open:Nn \g__char_data_ior { CaseFolding.txt }
25591   \cs_set_protected:Npn \__char_data_auxi:w #1 ;~ #2 ;~ #3 ; #4 \q_stop
25592   {
25593     \if:w \tl_head:n { #2 ? } C
25594       \reverse_if:N \if_int_compare:w
25595         \char_value_lccode:n {"#1} = "#3 ~
25596         \tl_const:cx
25597           { c__char_fold_case_ \__char_generate:w "#1 _tl }
25598           { \__char_generate:n { "#3 } }
25599     \fi:
25600   \else:
25601     \if:w \tl_head:n { #2 ? } F
25602       \__char_data_auxii:w #1 ~ #3 ~ \q_stop
25603     \fi:
25604   \fi:
25605   }
25606   \cs_set_protected:Npn \__char_data_auxii:w #1 ~ #2 ~ #3 ~ #4 \q_stop
25607   {
25608     \tl_const:cx { c__char_fold_case_ \__char_generate:w "#1 _tl }
25609   }

```

```

25610          \__char_generate:n { "#2 }
25611          \__char_generate:n { "#3 }
25612          \tl_if_blank:nF {#4}
25613              { \__char_generate:n { \int_value:w "#4 } }
25614      }
25615  }
25616 \ior_str_map_inline:Nn \g__char_data_ior
25617  {
25618      \reverse_if:F \if:w \c_hash_str \tl_head:w #1 \c_hash_str \q_stop
25619          \__char_data_auxi:w #1 \q_stop
25620      \fi:
25621  }
25622 \ior_close:N \g__char_data_ior

```

For upper and lower casing special situations, there is a bit more to do as we also have title casing to consider, plus we need to stop part-way through the file.

```

25623 \ior_open:Nn \g__char_data_ior { SpecialCasing.txt }
25624 \cs_set_protected:Npn \__char_data_auxi:w
25625     #1 ;~ #2 ;~ #3 ;~ #4 ; #5 \q_stop
25626  {
25627      \use:n { \__char_data_auxii:w #1 ~ lower ~ #2 ~ } ~ \q_stop
25628      \use:n { \__char_data_auxii:w #1 ~ upper ~ #4 ~ } ~ \q_stop
25629      \str_if_eq:nnF {#3} {#4}
25630          { \use:n { \__char_data_auxii:w #1 ~ mixed ~ #3 ~ } ~ \q_stop }
25631  }
25632 \cs_set_protected:Npn \__char_data_auxii:w
25633     #1 ~ #2 ~ #3 ~ #4 ~ #5 \q_stop
25634  {
25635      \tl_if_empty:nF {#4}
25636          {
25637              \tl_const:cx { c__char_ #2 _case_ \__char_generate:w "#1 _tl }
25638                  {
25639                      \__char_generate:n { "#3 }
25640                      \__char_generate:n { "#4 }
25641                      \tl_if_blank:nF {#5}
25642                          { \__char_generate:n { "#5 } }
25643                  }
25644          }
25645  }
25646 \ior_str_map_inline:Nn \g__char_data_ior
25647  {
25648      \str_if_eq:eeTF
25649          { \tl_head:w #1 \c_hash_str \q_stop }
25650          { \c_hash_str }
25651          {
25652              \str_if_eq:eeT
25653                  {#1}
25654                  { \c_hash_str \c_space_tl Conditional-Mappings }
25655                  { \ior_map_break: }
25656          }
25657          { \__char_data_auxi:w #1 \q_stop }
25658  }
25659 \ior_close:N \g__char_data_ior
25660 \group_end:
25661 }

```

For the 8-bit engines, the above is skipped but there is still some set up required. As case changing can only be applied to bytes, and they have to be in the ASCII range, we define a series of data stores to represent them, and the data are used such that only these are ever case-changed. We do open and close one file to force allocation of a read: this keeps all engines in line.

```

25662  {
25663  \group_begin:
25664  \cs_set_protected:Npn \__char_tmp:NN #1#2
25665  {
25666  \quark_if_recursion_tail_stop:N #2
25667  \tl_const:cn { c__char_upper_case_ #2 _tl } {#1}
25668  \tl_const:cn { c__char_lower_case_ #1 _tl } {#2}
25669  \tl_const:cn { c__char_fold_case_ #1 _tl } {#2}
25670  \__char_tmp:NN
25671  }
25672  \__char_tmp:NN
25673  AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz
25674  ? \q_recursion_tail \q_recursion_stop
25675  \ior_open:Nn \g__char_data_ior { UnicodeData.txt }
25676  \ior_close:N \g__char_data_ior
25677  \group_end:
25678  }
25679  /initex | package>

```

## 44 I3candidates Implementation

```
25680  {*initex | package>
```

### 44.1 Additions to I3basics

**\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.

```

25681 \cs_new_protected:Npn \mode_leave_vertical:
25682  {
25683  \if_mode_vertical:
25684  \exp_after:wN \tex_indent:D
25685  \fi:
25686  }

```

(End definition for \mode\_leave\_vertical:. This function is documented on page 242.)

### 44.2 Additions to I3box

```
25687  (@@=box)
```

#### 44.2.1 Viewing part of a box

**\box\_clip:N** A wrapper around the driver-dependent code.  
**\box\_clip:c** 25688 \cs\_new\_protected:Npn \box\_clip:N #1  
**\box\_gclip:N**  
**\box\_gclip:c**

```

25689 { \hbox_set:Nn #1 { \driver_box_use_clip:N #1 } }
25690 \cs_generate_variant:Nn \box_clip:N { c }
25691 \cs_new_protected:Npn \box_gclip:N #1
25692 { \hbox_gset:Nn #1 { \driver_box_use_clip:N #1 } }
25693 \cs_generate_variant:Nn \box_gclip:N { c }

```

(End definition for `\box_clip:N` and `\box_gclip:N`. These functions are documented on page 243.)

`\box_set_trim:Nnnn`  
`\box_set_trim:cnnn`

`\box_gset_trim:Nnnn`  
`\box_gset_trim:cnnn`  
`\__box_set_trim:Nnnnn`

```

25694 \cs_new_protected:Npn \box_set_trim:Nnnnn #1#2#3#4#5
25695 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
25696 \cs_generate_variant:Nn \box_set_trim:Nnnnn { c }
25697 \cs_new_protected:Npn \box_gset_trim:Nnnnn #1#2#3#4#5
25698 { \__box_set_trim:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
25699 \cs_generate_variant:Nn \box_gset_trim:Nnnnn { c }
25700 \cs_new_protected:Npn \__box_set_trim:NnnnnN #1#2#3#4#5#6
25701 {
25702     \hbox_set:Nn \l__box_internal_box
25703     {
25704         \tex_kern:D - \__box_dim_eval:n {#2}
25705         \box_use:N #1
25706         \tex_kern:D - \__box_dim_eval:n {#4}
25707     }

```

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`.

```

25708 \dim_compare:nNnTF { \box_dp:N #1 } > {#3}
25709 {
25710     \hbox_set:Nn \l__box_internal_box
25711     {
25712         \box_move_down:nn \c_zero_dim
25713         { \box_use_drop:N \l__box_internal_box }
25714     }
25715     \box_set_dp:Nn \l__box_internal_box { \box_dp:N #1 - (#3) }
25716 }
25717 {
25718     \hbox_set:Nn \l__box_internal_box
25719     {
25720         \box_move_down:nn { (#3) - \box_dp:N #1 }
25721         { \box_use_drop:N \l__box_internal_box }
25722     }
25723     \box_set_dp:Nn \l__box_internal_box \c_zero_dim
25724 }

```

Same thing, this time from the top of the box.

```

25725 \dim_compare:nNnTF { \box_ht:N \l__box_internal_box } > {#5}
25726 {
25727     \hbox_set:Nn \l__box_internal_box
25728     {
25729         \box_move_up:nn \c_zero_dim

```

```

25730           { \box_use_drop:N \l_box_internal_box }
25731       }
25732   \box_set_ht:Nn \l_box_internal_box
25733       { \box_ht:N \l_box_internal_box - (#5) }
25734   }
25735   {
25736       \hbox_set:Nn \l_box_internal_box
25737       {
25738           \box_move_up:nn { (#5) - \box_ht:N \l_box_internal_box }
25739               { \box_use_drop:N \l_box_internal_box }
25740           }
25741       \box_set_ht:Nn \l_box_internal_box \c_zero_dim
25742   }
25743   #6 #1 \l_box_internal_box
25744 }

(End definition for \box_set_trim:Nnnnn, \box_gset_trim:Nnnnn, and \__box_set_trim:NnnnnN. These
functions are documented on page 243.)
```

**\box\_set\_viewport:Nnnnn** 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.

```

\box_gset_viewport:Nnnnn
\box_gset_viewport:cnnnn
\__box_viewport:NnnnnN
25745 \cs_new_protected:Npn \box_set_viewport:Nnnnn #1#2#3#4#5
25746   { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_set_eq:NN }
25747   \cs_generate_variant:Nn \box_set_viewport:Nnnnn { c }
25748   \cs_new_protected:Npn \box_gset_viewport:Nnnnn #1#2#3#4#5
25749   { \__box_set_viewport:NnnnnN #1 {#2} {#3} {#4} {#5} \box_gset_eq:NN }
25750   \cs_generate_variant:Nn \box_gset_viewport:Nnnnn { c }
25751   \cs_new_protected:Npn \__box_set_viewport:NnnnnN #1#2#3#4#5#6
25752   {
25753       \hbox_set:Nn \l_box_internal_box
25754       {
25755           \tex_kern:D - \__box_dim_eval:n {#2}
25756           \box_use:N #1
25757           \tex_kern:D \__box_dim_eval:n { #4 - \box_wd:N #1 }
25758       }
25759       \dim_compare:nNnTF {#3} < \c_zero_dim
25760       {
25761           \hbox_set:Nn \l_box_internal_box
25762           {
25763               \box_move_down:nn \c_zero_dim
25764                   { \box_use_drop:N \l_box_internal_box }
25765           }
25766           \box_set_dp:Nn \l_box_internal_box { - \__box_dim_eval:n {#3} }
25767       }
25768       {
25769           \hbox_set:Nn \l_box_internal_box
25770               { \box_move_down:nn {#3} { \box_use_drop:N \l_box_internal_box } }
25771               \box_set_dp:Nn \l_box_internal_box \c_zero_dim
25772       }
25773       \dim_compare:nNnTF {#5} > \c_zero_dim
25774       {
25775           \hbox_set:Nn \l_box_internal_box
25776           {
25777               \box_move_up:nn \c_zero_dim
```

```

25778     { \box_use_drop:N \l__box_internal_box }
25779   }
25780 \box_set_ht:Nn \l__box_internal_box
25781   {
25782     (#5)
25783     \dim_compare:nNnT {#3} > \c_zero_dim
25784     { - (#3) }
25785   }
25786 }
25787 {
25788   \hbox_set:Nn \l__box_internal_box
25789   {
25790     \box_move_up:nn { - \__box_dim_eval:n {#5} }
25791     { \box_use_drop:N \l__box_internal_box }
25792   }
25793   \box_set_ht:Nn \l__box_internal_box \c_zero_dim
25794 }
25795 #6 #1 \l__box_internal_box
25796 }

```

(End definition for `\box_set_viewport:Nnnnn`, `\box_gset_viewport:Nnnnn`, and `\__box_viewport:NnnnnN`.  
These functions are documented on page 243.)

### 44.3 Additions to l3coffins

25797 `<@=coffin>`

#### 44.3.1 Rotating coffins

`\l_coffin_sin_fp` Used for rotations to get the sine and cosine values.

`\l_coffin_cos_fp` 25798 `\fp_new:N \l_coffin_sin_fp`  
25799 `\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.

25800 `\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` 25801 `\prop_new:N \l_coffin_corners_prop`  
25802 `\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.

25803 `\dim_new:N \l_coffin_bounding_shift_dim`

(End definition for `\l_coffin_bounding_shift_dim`.)

```

\l__coffin_left_corner_dim
\l__coffin_right_corner_dim
\l__coffin_bottom_corner_dim
\l__coffin_top_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.

```

25804 \dim_new:N \l__coffin_left_corner_dim
25805 \dim_new:N \l__coffin_right_corner_dim
25806 \dim_new:N \l__coffin_bottom_corner_dim
25807 \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:NnnN

```

Rotating a coffin requires several steps which can be conveniently run together. The sine and cosine of the angle in degrees are computed. This is then used to set `\l__coffin_sin_fp` and `\l__coffin_cos_fp`, which are carried through unchanged for the rest of the procedure.

```

25808 \cs_new_protected:Npn \coffin_rotate:Nn #1#2
25809   { \__coffin_rotate:NnnN #1 {##2} \box_rotate:Nn \prop_set_eq:cN }
25810 \cs_new_protected:Npn \coffin_grotate:Nn #1#2
25811   { \__coffin_rotate:NnnN #1 {##2} \box_grotate:Nn \prop_gset_eq:cN }
25812 \cs_new_protected:Npn \__coffin_rotate:NnnN #1#2#3#4
25813   {
25814     \fp_set:Nn \l__coffin_sin_fp { sind ( #2 ) }
25815     \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.

```

25816 \prop_set_eq:Nc \l__coffin_corners_prop
25817   { coffin ~ \__coffin_to_value:N #1 ~ corners }
25818 \prop_set_eq:Nc \l__coffin_poles_prop
25819   { 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.

```

25820 \prop_map_inline:Nn \l__coffin_corners_prop
25821   { \__coffin_rotate_corner:Nnnn #1 {##1} ##2 }
25822 \prop_map_inline:Nn \l__coffin_poles_prop
25823   { \__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.

```

25824 \__coffin_set_bounding:N #1
25825 \prop_map_inline:Nn \l__coffin_bounding_prop
25826   { \__coffin_rotate_bounding:nmm {##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.

```

25827 \__coffin_find_corner_maxima:N #1
25828 \__coffin_find_bounding_shift:
25829   #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.

```

25830 \hbox_set:Nn \l__coffin_internal_box

```

```

25831   {
25832     \tex_kern:D
25833     \dim_eval:n
25834       { \l__coffin_bounding_shift_dim - \l__coffin_left_corner_dim }
25835     \exp_stop_f:
25836     \box_move_down:nn { \l__coffin_bottom_corner_dim }
25837     { \box_use:N #1 }
25838   }

```

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.

```

25839   \box_set_ht:Nn \l__coffin_internal_box
25840     { \l__coffin_top_corner_dim - \l__coffin_bottom_corner_dim }
25841   \box_set_dp:Nn \l__coffin_internal_box { 0 pt }
25842   \box_set_wd:Nn \l__coffin_internal_box
25843     { \l__coffin_right_corner_dim - \l__coffin_left_corner_dim }
25844   \hbox_set:Nn #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.

```

25845   \prop_map_inline:Nn \l__coffin_corners_prop
25846     { \__coffin_shift_corner:Nnnn #1 {##1} ##2 }
25847   \prop_map_inline:Nn \l__coffin_poles_prop
25848     { \__coffin_shift_pole:Nnnnnn #1 {##1} ##2 }

```

Update the coffin data.

```

25849   #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
25850     \l__coffin_corners_prop
25851   #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
25852     \l__coffin_poles_prop
25853   }
25854 \cs_generate_variant:Nn \coffin_rotate:Nn { c }

```

(End definition for `\coffin_rotate:Nn`, `\coffin_grotate:Nn`, and `\__coffin_rotate:NnNN`. These functions are documented on page 243.)

`\__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.

```

25855 \cs_new_protected:Npn \__coffin_set_bounding:N #1
25856   {
25857     \prop_put:Nnx \l__coffin_bounding_prop { tl }
25858       { { 0 pt } { \dim_eval:n { \box_ht:N #1 } } }
25859     \prop_put:Nnx \l__coffin_bounding_prop { tr }
25860       {
25861         { \dim_eval:n { \box_wd:N #1 } }
25862         { \dim_eval:n { \box_ht:N #1 } }
25863       }
25864     \dim_set:Nn \l__coffin_internal_dim { -\box_dp:N #1 }
25865     \prop_put:Nnx \l__coffin_bounding_prop { bl }
25866       { { 0 pt } { \dim_use:N \l__coffin_internal_dim } }
25867     \prop_put:Nnx \l__coffin_bounding_prop { br }
25868       {

```

```

25869     { \dim_eval:n { \box_wd:N #1 } }
25870     { \dim_use:N \l__coffin_internal_dim }
25871   }
25872 }

```

(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.

```

25873 \cs_new_protected:Npn \_coffin_rotate_bounding:nnn #1#2#3
25874   {
25875     \_coffin_rotate_vector:nnNN {#2} {#3} \l__coffin_x_dim \l__coffin_y_dim
25876     \prop_put:Nnx \l__coffin_bounding_prop {#1}
25877       { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
25878   }
25879 \cs_new_protected:Npn \_coffin_rotate_corner:Nnnn #1#2#3#4
25880   {
25881     \_coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
25882     \prop_put:Nnx \l__coffin_corners_prop {#2}
25883       { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
25884 }

```

(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.

```

25885 \cs_new_protected:Npn \_coffin_rotate_pole:Nnnnnn #1#2#3#4#5#6
25886   {
25887     \_coffin_rotate_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
25888     \_coffin_rotate_vector:nnNN {#5} {#6}
25889       \l__coffin_x_prime_dim \l__coffin_y_prime_dim
25890     \prop_put:Nnx \l__coffin_poles_prop {#2}
25891       {
25892         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
25893         { \dim_use:N \l__coffin_x_prime_dim }
25894         { \dim_use:N \l__coffin_y_prime_dim }
25895       }
25896   }

```

(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.

```

25897 \cs_new_protected:Npn \_coffin_rotate_vector:nnNN #1#2#3#4
25898   {
25899     \dim_set:Nn #3
25900       {
25901         \fp_to_dim:n
25902           {
25903             \dim_to_fp:n {#1} * \l__coffin_cos_fp
25904             - \dim_to_fp:n {#2} * \l__coffin_sin_fp

```

```

25905         }
25906     }
25907     \dim_set:Nn #4
25908     {
25909         \fp_to_dim:n
25910         {
25911             \dim_to_fp:n {#1} * \l__coffin_sin_fp
25912             + \dim_to_fp:n {#2} * \l__coffin_cos_fp
25913         }
25914     }
25915 }
```

(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.

```

25916 \cs_new_protected:Npn \_coffin_find_corner_maxima:N #1
25917     {
25918         \dim_set:Nn \l__coffin_top_corner_dim { -\c_max_dim }
25919         \dim_set:Nn \l__coffin_right_corner_dim { -\c_max_dim }
25920         \dim_set:Nn \l__coffin_bottom_corner_dim { \c_max_dim }
25921         \dim_set:Nn \l__coffin_left_corner_dim { \c_max_dim }
25922         \prop_map_inline:Nn \l__coffin_corners_prop
25923             { \_coffin_find_corner_maxima_aux:nn ##2 }
25924     }
25925 \cs_new_protected:Npn \_coffin_find_corner_maxima_aux:nn #1#2
25926     {
25927         \dim_set:Nn \l__coffin_left_corner_dim
25928             { \dim_min:nn { \l__coffin_left_corner_dim } {#1} }
25929         \dim_set:Nn \l__coffin_right_corner_dim
25930             { \dim_max:nn { \l__coffin_right_corner_dim } {#1} }
25931         \dim_set:Nn \l__coffin_bottom_corner_dim
25932             { \dim_min:nn { \l__coffin_bottom_corner_dim } {#2} }
25933         \dim_set:Nn \l__coffin_top_corner_dim
25934             { \dim_max:nn { \l__coffin_top_corner_dim } {#2} }
25935     }
```

(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.

```

25936 \cs_new_protected:Npn \_coffin_find_bounding_shift:
25937     {
25938         \dim_set:Nn \l__coffin_bounding_shift_dim { \c_max_dim }
25939         \prop_map_inline:Nn \l__coffin_bounding_prop
25940             { \_coffin_find_bounding_shift_aux:nn ##2 }
25941     }
25942 \cs_new_protected:Npn \_coffin_find_bounding_shift_aux:nn #1#2
25943     {
25944         \dim_set:Nn \l__coffin_bounding_shift_dim
25945             { \dim_min:nn { \l__coffin_bounding_shift_dim } {#1} }
25946     }
```

(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.

```
25947 \cs_new_protected:Npn \__coffin_shift_corner:Nnnn #1#2#3#4
25948 {
25949   \prop_put:Nnx \l__coffin_corners_prop {#2}
25950   {
25951     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
25952     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
25953   }
25954 }
25955 \cs_new_protected:Npn \__coffin_shift_pole:Nnnnnn #1#2#3#4#5#6
25956 {
25957   \prop_put:Nnx \l__coffin_poles_prop {#2}
25958   {
25959     { \dim_eval:n { #3 - \l__coffin_left_corner_dim } }
25960     { \dim_eval:n { #4 - \l__coffin_bottom_corner_dim } }
25961     {#5} {#6}
25962   }
25963 }
```

(End definition for `\__coffin_shift_corner:Nnnn` and `\__coffin_shift_pole:Nnnnnn`.)

#### 44.3.2 Resizing coffins

`\l__coffin_scale_x_fp`  
`\l__coffin_scale_y_fp`

```
25964 \fp_new:N \l__coffin_scale_x_fp
25965 \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`

```
25966 \dim_new:N \l__coffin_scaled_total_height_dim
25967 \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`

```
25968 \cs_new_protected:Npn \coffin_resize:Nnn #1#2#3
25969 {
25970   \__coffin_resize:NnnNN #1 {#2} {#3}
25971   \box_resize_to_wd_and_ht_plus_dp:Nnn
25972   \prop_set_eq:cN
25973 }
25974 \cs_generate_variant:Nn \coffin_resize:Nnn { c }
25975 \cs_new_protected:Npn \coffin_gresize:Nnn #1#2#3
25976 {
25977   \__coffin_resize:NnnNN #1 {#2} {#3}
25978   \box_gresize_to_wd_and_ht_plus_dp:Nnn
```

```

25979     \prop_gset_eq:cN
25980   }
25981 \cs_generate_variant:Nn \coffin_gresize:Nnn { c }
25982 \cs_new_protected:Npn \__coffin_resize:NnnNN #1#2#3#4#5
25983   {
25984     \fp_set:Nn \l__coffin_scale_x_fp
25985       { \dim_to_fp:n {#2} / \dim_to_fp:n { \coffin_wd:N #1 } }
25986     \fp_set:Nn \l__coffin_scale_y_fp
25987       {
25988         \dim_to_fp:n {#3}
25989         / \dim_to_fp:n { \coffin_ht:N #1 + \coffin_dp:N #1 }
25990       }
25991     #4 #1 {#2} {#3}
25992     \__coffin_resize_common:NnnN #1 {#2} {#3} #5
25993   }

(End definition for \coffin_resize:Nnn, \coffin_gresize:Nnn, and \__coffin_resize:NnnNN. These functions are documented on page 243.)
```

`\__coffin_resize_common:NnnN` The poles and corners of the coffin are scaled to the appropriate places before actually resizing the underlying box.

```

25994 \cs_new_protected:Npn \__coffin_resize_common:NnnN #1#2#3#4
25995   {
25996     \prop_set_eq:Nc \l__coffin_corners_prop
25997       { coffin ~ \__coffin_to_value:N #1 ~ corners }
25998     \prop_set_eq:Nc \l__coffin_poles_prop
25999       { coffin ~ \__coffin_to_value:N #1 ~ poles }
26000     \prop_map_inline:Nn \l__coffin_corners_prop
26001       { \__coffin_scale_corner:Nnnn #1 {##1} ##2 }
26002     \prop_map_inline:Nn \l__coffin_poles_prop
26003       { \__coffin_scale_pole:Nnnnnn #1 {##1} ##2 }
```

Negative  $x$ -scaling values place the poles in the wrong location: this is corrected here.

```

26004   \fp_compare:nNnT \l__coffin_scale_x_fp < \c_zero_fp
26005     {
26006       \prop_map_inline:Nn \l__coffin_corners_prop
26007         { \__coffin_x_shift_corner:Nnnn #1 {##1} ##2 }
26008       \prop_map_inline:Nn \l__coffin_poles_prop
26009         { \__coffin_x_shift_pole:Nnnnnn #1 {##1} ##2 }
26010     }
26011   #4 { coffin ~ \__coffin_to_value:N #1 ~ corners }
26012     \l__coffin_corners_prop
26013   #4 { coffin ~ \__coffin_to_value:N #1 ~ poles }
26014     \l__coffin_poles_prop
26015 }
```

(End definition for \\_\_coffin\_resize\_common:NnnN.)

`\coffin_scale:Nnn` For scaling, the opposite calculation is done to find the new dimensions for the coffin.  
`\coffin_scale:cnn` 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.  
`\coffin_gscale:Nnn`

```

26016 \cs_new_protected:Npn \coffin_scale:Nnn #1#2#3
26017   { \__coffin_scale:NnnNN #1 {#2} {#3} \box_scale:Nnn \prop_set_eq:cN }
26018 \cs_generate_variant:Nn \coffin_scale:Nnn { c }
```

```

26019 \cs_new_protected:Npn \coffin_gscale:Nnn #1#2#3
26020   { \__coffin_scale:NnnNN #1 {#2} {#3} \box_gscale:Nnn \prop_gset_eq:cN }
26021 \cs_generate_variant:Nn \coffin_gscale:Nnn { c }
26022 \cs_new_protected:Npn \__coffin_scale:NnnNN #1#2#3#4#5
26023   {
26024     \fp_set:Nn \l__coffin_scale_x_fp {#2}
26025     \fp_set:Nn \l__coffin_scale_y_fp {#3}
26026     #4 #1 { \l__coffin_scale_x_fp } { \l__coffin_scale_y_fp }
26027     \dim_set:Nn \l__coffin_internal_dim
26028       { \coffin_ht:N #1 + \coffin_dp:N #1 }
26029     \dim_set:Nn \l__coffin_scaled_total_height_dim
26030       { \fp_abs:n { \l__coffin_scale_y_fp } \l__coffin_internal_dim }
26031     \dim_set:Nn \l__coffin_scaled_width_dim
26032       { -\fp_abs:n { \l__coffin_scale_x_fp } \coffin_wd:N #1 }
26033     \__coffin_resize_common:NnnN #1
26034       { \l__coffin_scaled_width_dim } { \l__coffin_scaled_total_height_dim }
26035     #5
26036   }

```

(End definition for `\coffin_scale:Nnn`, `\coffin_gscale:Nnn`, and `\coffin_scale:NnnNN`. These functions are documented on page 244.)

#### `\__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.

```

26037 \cs_new_protected:Npn \__coffin_scale_vector:nnNN #1#2#3#4
26038   {
26039     \dim_set:Nn #3
26040       { \fp_to_dim:n { \dim_to_fp:n {#1} * \l__coffin_scale_x_fp } }
26041     \dim_set:Nn #4
26042       { \fp_to_dim:n { \dim_to_fp:n {#2} * \l__coffin_scale_y_fp } }
26043   }

```

(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.

```

26044 \cs_new_protected:Npn \__coffin_scale_corner:Nnnn #1#2#3#4
26045   {
26046     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
26047     \prop_put:Nnx \l__coffin_corners_prop {#2}
26048       { { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim } }
26049   }
26050 \cs_new_protected:Npn \__coffin_scale_pole:Nnnnnn #1#2#3#4#5#6
26051   {
26052     \__coffin_scale_vector:nnNN {#3} {#4} \l__coffin_x_dim \l__coffin_y_dim
26053     \prop_put:Nnx \l__coffin_poles_prop {#2}
26054       {
26055         { \dim_use:N \l__coffin_x_dim } { \dim_use:N \l__coffin_y_dim }
26056         {#5} {#6}
26057       }
26058   }

```

(End definition for `\__coffin_scale_corner:Nnnn` and `\__coffin_scale_pole:Nnnnnn`.)

`\_coffin_x_shift_corner:Nnnn` These functions correct for the  $x$  displacement that takes place with a negative horizontal scaling.  
`\_coffin_x_shift_pole:Nnnnnn`

```

26059 \cs_new_protected:Npn \_coffin_x_shift_corner:Nnnn #1#2#3#4
26060 {
26061   \prop_put:Nnx \l__coffin_corners_prop {#2}
26062   {
26063     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
26064   }
26065 }
26066 \cs_new_protected:Npn \_coffin_x_shift_pole:Nnnnnn #1#2#3#4#5#6
26067 {
26068   \prop_put:Nnx \l__coffin_poles_prop {#2}
26069   {
26070     { \dim_eval:n { #3 + \box_wd:N #1 } } {#4}
26071     {#5} {#6}
26072   }
26073 }
```

(End definition for `\_coffin_x_shift_corner:Nnnn` and `\_coffin_x_shift_pole:Nnnnnn`.)

#### 44.4 Additions to l3fp-convert

26074 ⟨@C=fp⟩

`\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.

```

26075 \cs_new:Npn \fp_sign:n #1
26076   { \fp_to_decimal:n { sign \_fp_parse:n {#1} } }
```

(End definition for `\fp_sign:n`. This function is documented on page 244.)

#### 44.5 Additions to l3file

26077 ⟨@C=file⟩

`\file_get_mdfive_hash:nN`  
`\file_get_size:nN`  
`\file_get_timestamp:nN`  
`\_file_get_details:nnN` These are all wrappers around the pdfTeX primitives doing the same jobs: as we want consistent file paths to be found, they are all set up using `\file_get_full_name:nN` and so are non-expandable `get` functions. Much of the code is repetitive but we need to branch for LuaTeX (emulation in Lua), for the slightly different syntax needed for `\tex_mdfivesum:D` and for the fact that primitive coverage varies in other engines.

```

26078 \cs_new_protected:Npn \file_get_mdfive_hash:nN #1#2
26079   { \_file_get_details:nnN {#1} { mdfive } {#2} }
26080 \cs_new_protected:Npn \file_get_size:nN #1#2
26081   { \_file_get_details:nnN {#1} { size } {#2} }
26082 \cs_new_protected:Npn \file_get_timestamp:nN #1#2
26083   { \_file_get_details:nnN {#1} { moddate } {#2} }
26084 \cs_new_protected:Npn \_file_get_details:nnN #1#2#3
26085   {
26086     \file_get_full_name:nN {#1} \l__file_full_name_str
26087     \str_set:Nx #3
26088   {
26089     \use:c { tex_file #2 :D } \exp_after:wN
26090       { \l__file_full_name_str }
26091   }
```

```

26092    }
26093 \sys_if_engine_luatex:TF
26094 {
26095   \cs_set_protected:Npn \__file_get_details:nN #1#2#3
26096   {
26097     \file_get_full_name:nN {#1} \l__file_full_name_str
26098     \str_set:Nx #3
26099     {
26100       \lua_now:e
26101       {
26102         \l3kernel.file#2
26103         ( " \lua_escape:e { \l__file_full_name_str } " )
26104       }
26105     }
26106   }
26107 }
26108 {
26109   \cs_set_protected:Npn \file_get_mdfive_hash:nN #1#2
26110   {
26111     \file_get_full_name:nN {#1} \l__file_full_name_str
26112     \tl_set:Nx #2
26113     {
26114       \tex_mdfivesum:D file \exp_after:wN
26115       { \l__file_full_name_str }
26116     }
26117   }
26118 \cs_if_exist:NF \tex_filesize:D
26119 {
26120   \cs_set_protected:Npn \__file_get_details:nN #1#2#3
26121   {
26122     \tl_clear:N #3
26123     \__kernel_msg_error:nnx
26124     { kernel } { primitive-not-available }
26125     { \exp_not:c { (pdf)file #2 } }
26126   }
26127 }
26128 }
26129 \__kernel_msg_new:nnnn { kernel } { primitive-not-available }
26130 { Primitive~\token_to_str:N #1 not-available }
26131 {
26132   The~version~of~XeTeX~in~use~does~not~provide~functionality~equivalent~to~
26133   the~\token_to_str:N #1 primitive.
26134 }

```

(End definition for `\file_get_mdfive_hash:nN` and others. These functions are documented on page [245](#).)

`\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.

```

26135 \cs_new_protected:Npn \file_if_exist_input:n #1
26136 {
26137   \file_get_full_name:nN {#1} \l__file_full_name_str
26138   \str_if_empty:NF \l__file_full_name_str
26139   { \__file_input:V \l__file_full_name_str }

```

```

26140      }
26141  \cs_new_protected:Npn \file_if_exist_input:nF #1#2
26142  {
26143      \file_get_full_name:nN {#1} \l_file_full_name_str
26144      \str_if_empty:NTF \l_file_full_name_str
26145          {#2}
26146          { \__file_input:V \l_file_full_name_str }
26147      }

```

(End definition for `\file_if_exist_input:n` and `\file_if_exist_input:nF`. These functions are documented on page 245.)

**\file\_input\_stop:** A simple rename.

```
26148 \cs_new_protected:Npn \file_input_stop: { \tex_endinput:D }
```

(End definition for `\file_input_stop:`. This function is documented on page 246.)

## 44.6 Additions to l3flag

```
26149 ⟨@@=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.

```

26150 \__kernel_patch:nnNNpn { \__flag_chk_exist:n {#1} } { }
26151 \cs_new:Npn \flag_raise_if_clear:n #1
26152 {
26153     \if_cs_exist:w flag~#1~0 \cs_end:
26154     \else:
26155         \cs:w flag~#1 \cs_end: 0 ;
26156     \fi:
26157 }

```

(End definition for `\flag_raise_if_clear:n`. This function is documented on page 246.)

## 44.7 Additions to l3msg

```
26158 ⟨@@=int⟩
```

**\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.

```

26159 \__kernel_patch_args:nNNpn
26160  { { \__kernel_chk_expr:nNnN {#1} \__int_eval:w { } \int_sign:n } }
26161 \cs_new:Npn \int_sign:n #1
26162 {
26163     \int_value:w \exp_after:wN \__int_sign:Nw
26164     \int_value:w \__int_eval:w #1 \__int_eval_end: ;
26165     \exp_stop_f:
26166 }
26167 \cs_new:Npn \__int_sign:Nw #1#2 ;
26168 {
26169     \if_meaning:w 0 #1
26170     0

```

```

26171     \else:
26172         \if_meaning:w - #1 - \fi: 1
26173     \fi:
26174 }

```

(End definition for `\int_sign:n` and `\_int_sign:Nw`. This function is documented on page 246.)

## 44.8 Additions to l3msg

26175 ⟨@C=msg⟩

Pass to an auxiliary the message to display and the module name

```

26176 \cs_new:Npn \msg_expandable_error:nnnnnn #1#2#3#4#5#6
26177 {
26178     \exp_args:Nf \__msg_expandable_error_module:nn
26179     {
26180         \exp_args:Nf \tl_to_str:n
26181             { \use:c { \c__msg_text_prefix_tl #1 / #2 } {#3} {#4} {#5} {#6} }
26182     }
26183     {#1}
26184 }
26185 \cs_new:Npn \msg_expandable_error:nnnnn #1#2#3#4#5
26186     { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} {#5} { } }
26187 \cs_new:Npn \msg_expandable_error:nnnn #1#2#3#4
26188     { \msg_expandable_error:nnnnnn {#1} {#2} {#3} {#4} { } { } }
26189 \cs_new:Npn \msg_expandable_error:nnn #1#2#3
26190     { \msg_expandable_error:nnnnnn {#1} {#2} {#3} { } { } { } }
26191 \cs_new:Npn \msg_expandable_error:nn #1#2
26192     { \msg_expandable_error:nnnnnn {#1} {#2} { } { } { } { } }
26193 \cs_generate_variant:Nn \msg_expandable_error:nnnnnn { nnffff }
26194 \cs_generate_variant:Nn \msg_expandable_error:nnnnn { nnfff }
26195 \cs_generate_variant:Nn \msg_expandable_error:nnnn { nnff }
26196 \cs_generate_variant:Nn \msg_expandable_error:nnn { nnf }
26197 \cs_new:Npn \__msg_expandable_error_module:nn #1#2
26198 {
26199     \exp_after:wN \exp_after:wN
26200         \exp_after:wN \use_none_delimit_by_q_stop:w
26201         \use:n { \::error ! ~ #2 : ~ #1 } \q_stop
26202 }

```

(End definition for `\msg_expandable_error:nnnnnn` and others. These functions are documented on page 248.)

`\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:nnN`. This is only important for expressions involving the `\currentgrouplevel` or `\currentgroupstype`. On the other hand we want the expression to be converted to a string with the usual escape character, hence within the wrapping code.

```

26203 \cs_new_protected:Npn \msg_show_eval:Nn #1#2
26204     { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_show:n }
26205 \cs_new_protected:Npn \msg_log_eval:Nn #1#2
26206     { \exp_args:Nf \__msg_show_eval:nnN { #1 {#2} } {#2} \tl_log:n }
26207 \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 248.)

`\msg_show_item:n` 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.

```
26208 \cs_new:Npx \msg_show_item:n #1
26209   { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { #1 } }
26210 \cs_new:Npx \msg_show_item_unbraced:n #1
26211   { \iow_newline: > ~ \c_space_tl \exp_not:N \tl_to_str:n { #1 } }
26212 \cs_new:Npx \msg_show_item:nn #1#2
26213   {
26214     \iow_newline: > \use:nn { ~ } { ~ }
26215     \exp_not:N \tl_to_str:n { #1 }
26216     \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
26217     \exp_not:N \tl_to_str:n { #2 }
26218   }
26219 \cs_new:Npx \msg_show_item_unbraced:nn #1#2
26220   {
26221     \iow_newline: > \use:nn { ~ } { ~ }
26222     \exp_not:N \tl_to_str:n { #1 }
26223     \use:nn { ~ } { ~ } => \use:nn { ~ } { ~ }
26224     \exp_not:N \tl_to_str:n { #2 }
26225   }
```

(End definition for `\msg_show_item:n` and others. These functions are documented on page 248.)

## 44.9 Additions to l3prg

`\bool_const:Nn` A merger between `\tl_const:Nn` and `\bool_set:Nn`.

```
26226 \__kernel_patch:nnNNpn { \__kernel_chk_var_scope:NN c #1 } { }
26227 \cs_new_protected:Npn \bool_const:Nn #1#2
26228   {
26229     \__kernel_chk_if_free_cs:N #1
26230     \tex_global:D \tex_chardef:D #1 = \bool_if_p:n { #2 }
26231   }
26232 \cs_generate_variant:Nn \bool_const:Nn { c }
```

(End definition for `\bool_const:Nn`. This function is documented on page 248.)

`\bool_set_inverse:N` Set to `false` or `true` locally or globally.

```
26233 \cs_new_protected:Npn \bool_set_inverse:N #1
26234   { \bool_if:NTF #1 { \bool_set_false:N } { \bool_set_true:N } #1 }
26235 \cs_generate_variant:Nn \bool_set_inverse:N { c }
26236 \cs_new_protected:Npn \bool_gset_inverse:N #1
26237   { \bool_if:NTF #1 { \bool_gset_false:N } { \bool_gset_true:N } #1 }
26238 \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 249.)

## 44.10 Additions to `I3prop`

26239 ⟨@@=prop⟩

`\prop_count:N` 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.

```
26240 \cs_new:Npn \prop_count:N #1
26241   {
26242     \int_eval:n
26243     {
26244       0
26245       \prop_map_function:NN #1 \__prop_count:nn
26246     }
26247   }
26248 \cs_new:Npn \__prop_count:nn #1#2 { + 1 }
26249 \cs_generate_variant:Nn \prop_count:N { c }
```

(End definition for `\prop_count:N` and `\__prop_count:nn`. This function is documented on page 249.)

`\prop_map_tokens:Nn`  
`\prop_map_tokens:cn`  
`\__prop_map_tokens:nwwn`

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 argument delimited by `\__prop_pair:wn` is `\prg_break:` instead of being empty.

```
26250 \cs_new:Npn \prop_map_tokens:Nn #1#2
26251   {
26252     \exp_last_unbraced:Nno
26253     \use_i:nn { \__prop_map_tokens:nwwn { #2 } } #1
26254     \prg_break: \__prop_pair:wn \s__prop { } \prg_break_point:
26255     \prg_break_point:Nn \prop_map_break: { }
26256   }
26257 \cs_new:Npn \__prop_map_tokens:nwwn #1#2 \__prop_pair:wn #3 \s__prop #4
26258   {
26259     #2
26260     \use:n { #1 } { #3 } { #4 }
26261     \__prop_map_tokens:nwwn { #1 }
26262   }
26263 \cs_generate_variant:Nn \prop_map_tokens:Nn { c }
```

(End definition for `\prop_map_tokens:Nn` and `\__prop_map_tokens:nwwn`. This function is documented on page 249.)

`\prop_rand_key_value:N`  
`\prop_rand_key_value:c`  
`\__prop_rand_item:w`

Contrarily to `clist`, `seq` and `tl`, 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`.

```
26264 \cs_new:Npn \prop_rand_key_value:N #1
26265   {
26266     \prop_if_empty:NF #1
26267     {
26268       \exp_after:wN \__prop_rand_item:w
26269       \int_value:w \int_rand:nn { 1 } { \prop_count:N #1 }
26270       #1 \q_stop
26271     }
26272 }
```

```

26272     }
26273 \cs_generate_variant:Nn \prop_rand_key_value:N { c }
26274 \cs_new:Npn \__prop_rand_item:w #1 \s__prop \__prop_pair:wn #2 \s__prop #3
26275 {
26276     \int_compare:nNnF {#1} > 1
26277         { \use_i_delimit_by_q_stop:nw { \exp_not:n { {#2} {#3} } } }
26278     \exp_after:wN \__prop_rand_item:w
26279     \int_value:w \int_eval:n { #1 - 1 } \s__prop
26280 }

```

(End definition for `\prop_rand_key_value:N` and `\__prop_rand_item:w`. This function is documented on page [249](#).)

## 44.11 Additions to `\__seq`

```
26281 ⟨@=seq⟩
```

`\seq_mapthread_function:NNN`  
`\seq_mapthread_function:NcN`  
`\seq_mapthread_function:cNN`  
`\seq_mapthread_function:ccN`  
`\__seq_mapthread_function:wNN`  
`\__seq_mapthread_function:wNw`  
`\__seq_mapthread_function:Nnnwnn`

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 be 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.

```

26282 \cs_new:Npn \seq_mapthread_function:NNN #1#2#3
26283   { \exp_after:wN \__seq_mapthread_function:wNN #2 \q_stop #1 #3 }
26284 \cs_new:Npn \__seq_mapthread_function:wNN \s__seq #1 \q_stop #2#3
26285   {
26286     \exp_after:wN \__seq_mapthread_function:wNw #2 \q_stop #3
26287     #1 { ? \prg_break: } { }
26288     \prg_break_point:
26289   }
26290 \cs_new:Npn \__seq_mapthread_function:wNw \s__seq #1 \q_stop #2
26291   {
26292     \__seq_mapthread_function:Nnnwnn #2
26293     #1 { ? \prg_break: } { }
26294     \q_stop
26295   }
26296 \cs_new:Npn \__seq_mapthread_function:Nnnwnn #1#2#3#4 \q_stop #5#6
26297   {
26298     \use_none:n #2
26299     \use_none:n #5
26300     #1 {#3} {#6}
26301     \__seq_mapthread_function:Nnnwnn #1 #4 \q_stop
26302   }
26303 \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 [250](#).)

`\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.

```

26304 \cs_new_protected:Npn \seq_set_filter:NNn
26305   { \__seq_set_filter:NNNn \tl_set:Nx }
26306 \cs_new_protected:Npn \seq_gset_filter:NNn
26307   { \__seq_set_filter:NNNn \tl_gset:Nx }
26308 \cs_new_protected:Npn \__seq_set_filter:NNNn #1#2#3#4
26309   {
26310     \__seq_push_item_def:n { \bool_if:nT {##4} { \__seq_wrap_item:n {##1} } }
26311     #1 #2 { #3 }
26312     \__seq_pop_item_def:
26313   }

```

(End definition for `\seq_set_filter:NNn`, `\seq_gset_filter:NNn`, and `\__seq_set_filter:NNNn`. These functions are documented on page 250.)

`\seq_set_map:NNn`  
`\seq_gset_map: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:NNNn
26314 \cs_new_protected:Npn \seq_set_map:NNn
26315   { \__seq_set_map:NNNn \tl_set:Nx }
26316 \cs_new_protected:Npn \seq_gset_map:NNn
26317   { \__seq_set_map:NNNn \tl_gset:Nx }
26318 \cs_new_protected:Npn \__seq_set_map:NNNn #1#2#3#4
26319   {
26320     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {##4} }
26321     #1 #2 { #3 }
26322     \__seq_pop_item_def:
26323   }

```

(End definition for `\seq_set_map:NNn`, `\seq_gset_map:NNn`, and `\__seq_set_map:NNNn`. These functions are documented on page 250.)

`\seq_set_from_inline_x:Nnn` Set `\__seq_item:n` then map it using the loop code.

```

\__seq_set_from_inline_x:Nnnn
26324 \cs_new_protected:Npn \seq_set_from_inline_x:Nnn
26325   { \__seq_set_from_inline_x:NNnn \tl_set:Nx }
26326 \cs_new_protected:Npn \seq_gset_from_inline_x:Nnn
26327   { \__seq_set_from_inline_x:NNnn \tl_gset:Nx }
26328 \cs_new_protected:Npn \__seq_set_from_inline_x:NNnn #1#2#3#4
26329   {
26330     \__seq_push_item_def:n { \exp_not:N \__seq_item:n {##4} }
26331     #1 #2 { \s_seq #3 \__seq_item:n }
26332     \__seq_pop_item_def:
26333   }

```

(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 251.)

`\seq_set_from_function:NnN` Reuse `\seq_set_from_inline_x:Nnn`.

```

\__seq_gset_from_function:NnN
26334 \cs_new_protected:Npn \seq_set_from_function:NnN #1#2#3
26335   { \seq_set_from_inline_x:Nnn #1 {##2} { ##3 {##1} } }
26336 \cs_new_protected:Npn \seq_gset_from_function:NnN #1#2#3
26337   { \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 250.)

\seq\_const\_from\_clist:Nn Almost identical to \seq\_set\_from\_clist:Nn.

```

\seq_const_from_clist:cN
26338 \cs_new_protected:Npn \seq_const_from_clist:Nn #1#2
26339 {
2640   \tl_const:Nx #1
2641   { \s_seq \clist_map_function:nN {#2} \__seq_wrap_item:n }
2642 }
2643 \cs_generate_variant:Nn \seq_const_from_clist:Nn { c }

(End definition for \seq_const_from_clist:Nn. This function is documented on page 250.)
```

\seq\_shuffle:N  
\seq\_gshuffle:N  
\\_\\_seq\_shuffle:NN  
\\_\\_seq\_shuffle\_item:n  
\g\\_seq\_internal\_seq  
\l\\_seq\_internal\_a\_int  
\l\\_seq\_internal\_b\_int

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.

```

26344 \cs_if_exist:NTF \tex_uniformdeviate:D
26345 {
26346   \int_new:N \l_seq_internal_a_int
26347   \int_new:N \l_seq_internal_b_int
26348   \seq_new:N \g_seq_internal_seq
26349   \cs_new_protected:Npn \seq_shuffle:N { \_\_seq_shuffle:NN \seq_set_eq:NN }
26350   \cs_new_protected:Npn \seq_gshuffle:N { \_\_seq_shuffle:NN \seq_gset_eq:NN }
26351   \cs_new_protected:Npn \_\_seq_shuffle:NN #1#2
26352   {
26353     \int_compare:nNnTF { \seq_count:N #2 } > \c_max_register_int
26354     {
26355       \__kernel_msg_error:nnx { kernel } { shuffle-too-large }
26356       { \token_to_str:N #2 }
26357     }
26358   {
26359     \group_begin:
26360     \cs_set_eq:NN \__seq_item:n \_\_seq_shuffle_item:n
26361     \int_zero:N \l_seq_internal_a_int
26362     #2
26363     \seq_gset_from_inline_x:Nnn \g_seq_internal_seq
26364     { \int_step_function:nN { \l_seq_internal_a_int } }
26365     { \tex_the:D \tex_toks:D ##1 }
26366     \group_end:
26367     #1 #2 \g_seq_internal_seq
26368     \seq_gclear:N \g_seq_internal_seq
26369   }
26370 }
26371 \cs_new_protected:Npn \_\_seq_shuffle_item:n
26372 {
26373   \int_incr:N \l_seq_internal_a_int
26374   \int_set:Nn \l_seq_internal_b_int
26375   { 1 + \tex_uniformdeviate:D \l_seq_internal_a_int }
26376   \tex_toks:D \l_seq_internal_a_int
26377   = \tex_toks:D \l_seq_internal_b_int
26378   \tex_toks:D \l_seq_internal_b_int
26379 }
26380 \__kernel_msg_new:nnnn { kernel } { shuffle-too-large }
26381 { The~sequence~#1~is~too~long~to~be~shuffled~by~TeX. }
26382 {
```

```

26383     TeX~has~ \int_eval:n { \c_max_register_int + 1 } ~
26384     toks~registers:~this~only~allows~to~shuffle~up~to~
26385     \int_use:N \c_max_register_int \ items.~
26386     The~list~will~not~be~shuffled.
26387   }
26388 }
26389 {
26390   \cs_new_protected:Npn \seq_shuffle:N #1
26391   {
26392     \_kernel_msg_error:nnn { kernel } { fp-no-random }
26393     { \seq_shuffle:N #1 }
26394   }
26395   \cs_new_eq:NN \seq_gshuffle:N \seq_shuffle:N
26396 }

```

(End definition for `\seq_shuffle:N` and others. These functions are documented on page [251](#).)

`\seq_indexed_map_function:NN` Similar to `\seq_map_function:NN` but we keep track of the item index as a ;-delimited argument of `\_seq_indexed_map:Nw`.

```

\seq_indexed_map_inline:Nn
\seq_indexed_map:nNn
\seq_indexed_map:Nw
26397 \cs_new:Npn \seq_indexed_map_function:NN #1#2
26398 {
26399   \_seq_indexed_map:NN #1#2
26400   \prg_break_point:Nn \seq_map_break: { }
26401 }
26402 \cs_new_protected:Npn \seq_indexed_map_inline:Nn #1#2
26403 {
26404   \int_gincr:N \g_kernel_prg_map_int
26405   \cs_gset_protected:cpn
26406   { \_seq_map_ \int_use:N \g_kernel_prg_map_int :w } ##1##2 {#2}
26407   \exp_args:NNC \_seq_indexed_map:NN #1
26408   { \_seq_map_ \int_use:N \g_kernel_prg_map_int :w }
26409   \prg_break_point:Nn \seq_map_break:
26410   { \int_gdecr:N \g_kernel_prg_map_int }
26411 }
26412 \cs_new:Npn \_seq_indexed_map:NN #1#2
26413 {
26414   \exp_after:wN \_seq_indexed_map:Nw
26415   \exp_after:wN #2
26416   \int_value:w 1
26417   \exp_after:wN \use_i:nn
26418   \exp_after:wN ;
26419   #1
26420   \prg_break: \_seq_item:n { } \prg_break_point:
26421 }
26422 \cs_new:Npn \_seq_indexed_map:Nw #1#2 ; #3 \_seq_item:n #4
26423 {
26424   #3
26425   #1 {#2} {#4}
26426   \exp_after:wN \_seq_indexed_map:Nw
26427   \exp_after:wN #1
26428   \int_value:w \int_eval:w 1 + #2 ;
26429 }

```

(End definition for `\seq_indexed_map_function:NN` and others. These functions are documented on page [251](#).)

## 44.12 Additions to l3skip

26430 ⟨@C=dim⟩

**\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.

```
26431 \__kernel_patch_args:nNNpn
26432 { { \__kernel_chk_expr:nNnN {#1} \__dim_eval:w { } \dim_sign:n } }
26433 \cs_new:Npn \dim_sign:n #1
26434 {
26435     \int_value:w \exp_after:wN \__dim_sign:Nw
26436     \dim_use:N \__dim_eval:w #1 \__dim_eval_end: ;
26437     \exp_stop_f:
26438 }
26439 \cs_new:Npn \__dim_sign:Nw #1#2 ;
26440 {
26441     \if_dim:w #1#2 > \c_zero_dim
26442         1
26443     \else:
26444         \if_meaning:w - #1
26445             -1
26446         \else:
26447             0
26448         \fi:
26449     \fi:
26450 }
```

(End definition for `\dim_sign:n` and `\__dim_sign:Nw`. This function is documented on page 251.)

26451 ⟨@C=skip⟩

**\skip\_split\_finite\_else\_action:nnNN** This macro is useful when performing error checking in certain circumstances. If the `<skip>` register holds finite glue it sets #3 and #4 to the stretch and shrink component, resp. If it holds infinite glue set #3 and #4 to zero and issue the special action #2 which is probably an error message. Assignments are local.

```
26452 \cs_new:Npn \skip_split_finite_else_action:nnNN #1#2#3#4
26453 {
26454     \skip_if_finite:nTF {#1}
26455     {
26456         #3 = \tex_gluestretch:D #1 \scan_stop:
26457         #4 = \tex_glueshrink:D #1 \scan_stop:
26458     }
26459     {
26460         #3 = \c_zero_skip
26461         #4 = \c_zero_skip
26462         #2
26463     }
26464 }
```

(End definition for `\skip_split_finite_else_action:nnNN`. This function is documented on page 251.)

## 44.13 Additions to I3sys

26465 ⟨@@=sys⟩

26466 \c\_sys\_engine\_version\_str Various different engines, various different ways to extract the data!

```
26466 \str_const:Nx \c_sys_engine_version_str
26467 {
26468   \str_case:on \c_sys_engine_str
26469   {
26470     { pdftex }
26471     {
26472       \fp_eval:n { round(\int_use:N \tex_pdftexversion:D / 100 , 2) }
26473       .
26474       \tex_pdftexrevision:D
26475     }
26476     { ptex }
26477     {
26478       \cs_if_exist:NT \tex_ptexversion:D
26479       {
26480         p
26481         \int_use:N \tex_ptexversion:D
26482         \int_use:N \tex_ptexminorversion:D
26483         \tex_ptexrevision:D
26484         -
26485         \int_use:N \tex_epTeXversion:D
26486       }
26487     }
26488     { luatex }
26489     {
26490       \fp_eval:n { round(\int_use:N \tex_luatexversion:D / 100, 2) }
26491       .
26492       \tex_luatexrevision:D
26493     }
26494     { uptex }
26495     {
26496       \cs_if_exist:NT \tex_ptexversion:D
26497       {
26498         p
26499         \int_use:N \tex_ptexversion:D
26500         \int_use:N \tex_ptexminorversion:D
26501         \tex_ptexrevision:D
26502         -
26503         u
26504         \int_use:N \tex_uptexversion:D
26505         \tex_uptexrevision:D
26506         -
26507         \int_use:N \tex_epTeXversion:D
26508       }
26509     }
26510     { xetex }
26511     {
26512       \int_use:N \tex_XeTeXversion:D
26513       \tex_XeTeXrevision:D
26514     }
26515 }
```

```
26516 }
```

(End definition for `\c_sys_engine_version_str`. This variable is documented on page 252.)

- \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).

```
26517 \sys_if_rand_exist:TF
26518   { \cs_new:Npn \sys_rand_seed: { \tex_the:D \tex_randomseed:D } }
26519   {
26520     \cs_new:Npn \sys_rand_seed:
26521     {
26522       \int_value:w
26523       \__kernel_msg_expandable_error:n { kernel } { fp-no-random }
26524       { \sys_rand_seed: }
26525       \c_zero_int
26526     }
26527   }
```

(End definition for `\sys_rand_seed:`. This function is documented on page 252.)

- \sys\_gset\_rand\_seed:n** The primitive always assigns the seed globally.

```
26528 \sys_if_rand_exist:TF
26529   {
26530     \cs_new_protected:Npn \sys_gset_rand_seed:n #1
26531     { \tex_setrandomseed:D \int_eval:n {#1} \exp_stop_f: }
26532   }
26533   {
26534     \cs_new_protected:Npn \sys_gset_rand_seed:n #1
26535     {
26536       \__kernel_msg_error:n { kernel } { fp-no-random }
26537       { \sys_gset_rand_seed:n {#1} }
26538     }
26539   }
```

(End definition for `\sys_gset_rand_seed:n`. This function is documented on page 252.)

- \c\_sys\_shell\_escape\_int** Expose the engine's shell escape status to the user.

```
26540 \int_const:Nn \c_sys_shell_escape_int
26541   {
26542     \sys_if_engine_luatex:TF
26543     {
26544       \tex_directlua:D
26545       { \tex.sprint(status.shell_escape~or~os.execute()) }
26546     }
26547     {
26548       \tex_shellescape:D
26549     }
26550   }
```

(End definition for `\c_sys_shell_escape_int`. This variable is documented on page 253.)

```
\l__sys_internal_tl
```

```
26551 \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.

```
26552 \tl_const:Nx \c_sys_marker_tl { : \token_to_str:N : }
```

(End definition for `\c_sys_marker_tl`.)

`\sys_shell_get:nnTF` Setting using a shell is at this level just a slightly specialised file operation, with an additional check for quotes, as these are not supported.

```
26553 \cs_new_protected:Npn \sys_shell_get:nnN #1#2#3
26554 {
26555   \sys_shell_get:nnNF {#1} {#2} #3
26556   { \tl_clear:N #3 }
26557 }
26558 \prg_new_protected_conditional:Npnn \sys_shell_get:nnN #1#2#3 { T , F , TF }
26559 {
26560   \sys_if_shell:TF
26561   { \exp_args:No \__sys_get:nnN { \tl_to_str:n {#1} } {#2} #3 }
26562   { \prg_return_false: }
26563 }
26564 \cs_new_protected:Npn \__sys_get:nnN #1#2#3
26565 {
26566   \tl_if_in:nnTF {#1} { " }
26567   {
26568     \__kernel_msg_error:nnx
26569     { kernel } { quote-in-shell } {#1}
26570     \prg_return_false:
26571   }
26572   {
26573     \if_false: { \fi:
26574     \group_begin:
26575       \int_set_eq:NN \tex_tracingnesting:D \c_zero_int
26576       \exp_args:No \tex_everyeof:D { \c_sys_marker_tl }
26577       #2 \scan_stop:
26578       \exp_after:wN \__sys_get_do:Nw
26579       \exp_after:wN #3
26580       \exp_after:wN \prg_do_nothing:
26581         \tex_input:D | "#1" \scan_stop:
26582       \if_false: } \fi:
26583       \prg_return_true:
26584     }
26585   }
26586 \exp_args:Nno \use:nn
26587   { \cs_new_protected:Npn \__sys_get_do:Nw #1#2 }
26588   { \c_sys_marker_tl }
26589   {
26590     \group_end:
26591     \tl_set:No #1 {#2}
26592   }
26593 \__kernel_msg_new:nnnn { kernel } { quote-in-shell }
26594   { Quotes~in~shell~command-'#1'. }
26595   { Shell~commands~cannot~contain~quotes~("). }
```

(End definition for `\sys_shell_get:nnTF` and others. These functions are documented on page 252.)

**\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.

```

26596 \sys_if_engine_luatex:TF
26597   {
26598     \str_const:Nx \c_sys_platform_str
26599       { \lua_now:n { tex.print(os.type) } }
26600   }
26601   {
26602     \file_if_exist:nTF { nul: }
26603       {
26604         \file_if_exist:nF { /dev/null }
26605           { \str_const:Nn \c_sys_platform_str { windows } }
26606       }
26607       {
26608         \file_if_exist:nT { /dev/null }
26609           { \str_const:Nn \c_sys_platform_str { unix } }
26610       }
26611   }
26612 \cs_if_exist:NF \c_sys_platform_str
26613   { \str_const:Nn \c_sys_platform_str { unknown } }

```

(End definition for `\c_sys_platform_str`. This variable is documented on page 253.)

**\sys\_if\_platform\_unix:p:** We can now set up the tests.

```

26614 \clist_map_inline:nn { unix , windows }
26615   {
26616     \__sys_const:nn { sys_if_platform_ #1 }
26617       { \str_if_eq_p:Vn \c_sys_platform_str { #1 } }
26618   }

```

(End definition for `\sys_if_platform_unix:TF` and `\sys_if_platform_windows:TF`. These functions are documented on page 253.)

**\sys\_if\_shell:p:** Performs a check for whether shell escape is enabled. The first set of functions returns true if either of restricted or unrestricted shell escape is enabled, while the other two sets of functions return true in only one of these two cases.

```

26619 \__sys_const:nn { sys_if_shell }
26620   { \int_compare_p:nNn \c_sys_shell_escape_int > 0 }
26621 \__sys_const:nn { sys_if_shell_unrestricted }
26622   { \int_compare_p:nNn \c_sys_shell_escape_int = 1 }
26623 \__sys_const:nn { sys_if_shell_restricted }
26624   { \int_compare_p:nNn \c_sys_shell_escape_int = 2 }

```

(End definition for `\sys_if_shell:TF`, `\sys_if_shell_unrestricted:TF`, and `\sys_if_shell_restricted:TF`. These functions are documented on page 253.)

**\c\_\_sys\_shell\_stream\_int** This is not needed for LuaTeX: shell escape there isn't done using a TeX interface.

```

26625 \sys_if_engine_luatex:F
26626   { \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.

```
26627 \sys_if_engine_luatex:TF
26628 {
26629     \cs_new_protected:Npn \sys_shell_now:n #1
26630     {
26631         \lua_now:e
26632         { os.execute(" \lua_escape:e { \tl_to_str:n {#1} } ") }
26633     }
26634 }
26635 {
26636     \cs_new_protected:Npn \sys_shell_now:n #1
26637     { \iow_now:Nn \c__sys_shell_stream_int {#1} }
26638 }
26639 \cs_generate_variant:Nn \sys_shell_now:n { x }
```

(End definition for `\sys_shell_now:n`. This function is documented on page 253.)

**\sys\_shell\_shipout:n** Execute commands through shell escape at shipout.

```
26640 \sys_if_engine_luatex:TF
26641 {
26642     \cs_new_protected:Npn \sys_shell_shipout:n #1
26643     {
26644         \lua_shipout_e:n
26645         { os.execute(" \lua_escape:e { \tl_to_str:n {#1} } ") }
26646     }
26647 }
26648 {
26649     \cs_new_protected:Npn \sys_shell_shipout:n #1
26650     { \iow_shipout:Nn \c__sys_shell_stream_int {#1} }
26651 }
26652 \cs_generate_variant:Nn \sys_shell_shipout:n { x }
```

(End definition for `\sys_shell_shipout:n`. This function is documented on page 253.)

#### 44.14 Additions to l3tl

26653 ⟨@C=tl⟩

**\tl\_if\_single\_token\_p:n** 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.

```
26654 \prg_new_conditional:Npnn \tl_if_single_token:n #1 { p , T , F , TF }
26655 {
26656     \tl_if_head_is_N_type:nTF {#1}
26657     { \__tl_if_empty_if:o { \use_none:n #1 } }
26658     {
26659         \tl_if_empty:nTF {#1}
26660         { \if_false: }
26661         { \__tl_if_empty_if:o { \exp:w \exp_end_continue_f:w #1 } }
26662     }
26663     \prg_return_true:
26664     \else:
```

```

26665     \prg_return_false:
26666     \fi:
26667 }
```

(End definition for `\tl_if_single_token:nTF`. This function is documented on page 254.)

### `\tl_reverse_tokens:n`

```

\__tl_reverse_group:nn
26668 \cs_new:Npn \tl_reverse_tokens:n #1
26669 {
26670     \__kernel_exp_not:w \exp_after:wN
26671     {
26672         \exp:w
26673         \__tl_act:NNNnn
26674         \__tl_reverse_normal:nN
26675         \__tl_reverse_group:nn
26676         \__tl_reverse_space:n
26677         { }
26678         {#1}
26679     }
26680 }
26681 \cs_new:Npn \__tl_reverse_group:nn #1
26682 {
26683     \__tl_act_group_recurse:Nnn
26684     \__tl_act_reverse_output:n
26685     { \tl_reverse_tokens:n }
26686 }
```

In many applications of `\__tl_act:NNNnn`, we need to recursively apply some transformation within brace groups, then output. In this code, #1 is the output function, #2 is the transformation, which should expand in two steps, and #3 is the group.

```

26687 \cs_new:Npn \__tl_act_group_recurse:Nnn #1#2#3
26688 {
26689     \exp_args:Nf #1
26690     { \exp_after:wN \exp_after:wN \exp_after:wN { #2 {#3} } }
26691 }
```

(End definition for `\tl_reverse_tokens:n`, `\__tl_reverse_group:nn`, and `\__tl_act_group_recurse:Nnn`. This function is documented on page 254.)

### `\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!

```

26692 \cs_new:Npn \tl_count_tokens:n #1
26693 {
26694     \int_eval:n
26695     {
26696         \__tl_act:NNNnn
26697         \__tl_act_count_normal:nN
26698         \__tl_act_count_group:nn
26699         \__tl_act_count_space:n
26700         { }
26701         {#1}
26702     }
```

```

26703     }
26704 \cs_new:Npn \__tl_act_count_normal:nN #1 #2 { 1 + }
26705 \cs_new:Npn \__tl_act_count_space:n #1 { 1 + }
26706 \cs_new:Npn \__tl_act_count_group:nn #1 #2
26707   { 2 + \tl_count_tokens:n {#2} + }

```

(End definition for `\tl_count_tokens:n` and others. This function is documented on page 254.)

#### 44.14.1 Deprecated functions

```

\tl_set_from_file:Nnn
\tl_set_from_file:cnn
\tl_gset_from_file:Nnn
\tl_gset_from_file:cnn
\tl_set_from_file_x:Nnn
\tl_set_from_file_x:cnn
\tl_gset_from_file_x:Nnn
\tl_gset_from_file_x:cnn
26708 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \file_get:nnN }
26709 \cs_new_protected:Npn \tl_set_from_file:Nnn #1#2#3
26710   { \file_get:nnN {#2} {#3} #1 }
26711 \cs_generate_variant:Nn \tl_set_from_file:Nnn { c }
26712 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \file_get:nnN }
26713 \cs_new_protected:Npn \tl_gset_from_file:Nnn #1#2#3
26714   {
26715     \group_begin:
26716       \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
26717       \tl_gset_eq:NN #1 \l__tl_internal_a_tl
26718     \group_end:
26719   }
26720 \cs_generate_variant:Nn \tl_gset_from_file:Nnn { c }
26721 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \file_get:nnN }
26722 \cs_new_protected:Npn \tl_set_from_file_x:Nnn #1#2#3
26723   {
26724     \group_begin:
26725       \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
26726       #2 \scan_stop:
26727       \tl_set:Nx \l__tl_internal_a_tl { \l__tl_internal_a_tl }
26728       \exp_args:NNNo \group_end:
26729       \tl_set:Nn #1 \l__tl_internal_a_tl
26730   }
26731 \cs_generate_variant:Nn \tl_set_from_file_x:Nnn { c }
26732 \__kernel_patch_deprecation:nnNNpn { 2020-12-31 } { \file_get:nnN }
26733 \cs_new_protected:Npn \tl_gset_from_file_x:Nnn #1#2#3
26734   {
26735     \group_begin:
26736       \file_get:nnN {#3} {#2} \l__tl_internal_a_tl
26737       #2 \scan_stop:
26738       \tl_gset:Nx #1 { \l__tl_internal_a_tl }
26739     \group_end:
26740   }
26741 \cs_generate_variant:Nn \tl_gset_from_file_x:Nnn { c }

```

(End definition for `\tl_set_from_file:Nnn` and others.)

#### 44.14.2 Unicode case changing

The mechanisms needed for case changing are somewhat involved, particularly to allow for all of the special cases. These functions also require the appropriate data extracted from the Unicode documentation (either manually or automatically).

First, some code which “belongs” in `l3tokens` but has to come here.

26742 ⟨@C=char⟩

\char\_lower\_case:N Expandable character generation is done using a two-part approach. First, see if the current character has a special mapping for the current transformation. If it does, insert that. Otherwise, use the T<sub>E</sub>X data to look up the one-to-one mapping, and generate the appropriate character with the appropriate category code. Mixed case needs an extra step as it may be special-cased or might be a special upper case outcome. The internal when using non-Unicode engines has to be set up to only do anything with ASCII characters.

\char\_upper\_case:N

\char\_mixed\_case:N

\char\_fold\_case:N

\\_\_char\_change\_case:nNN

\\_\_char\_change\_case:nN

```
26743 \cs_new:Npn \char_lower_case:N #1
26744   { \__char_change_case:nNN { lower } \char_value_lccode:n #1 }
26745 \cs_new:Npn \char_upper_case:N #1
26746   { \__char_change_case:nNN { upper } \char_value_uccode:n #1 }
26747 \cs_new:Npn \char_mixed_case:N #1
26748   {
26749     \tl_if_exist:cTF { c__char_mixed_case_ \token_to_str:N #1 _tl }
26750       { \tl_use:c { c__char_mixed_case_ \token_to_str:N #1 _tl } }
26751       { \char_upper_case:N #1 }
26752   }
26753 \cs_new:Npn \char_fold_case:N #1
26754   { \__char_change_case:nNN { fold } \char_value_lccode:n #1 }
26755 \cs_new:Npn \__char_change_case:nNN #1#2#3
26756   {
26757     \tl_if_exist:cTF { c__char_ #1 _case_ \token_to_str:N #3 _tl }
26758       { \tl_use:c { c__char_ #1 _case_ \token_to_str:N #3 _tl } }
26759       { \exp_args:Nf \__char_change_case:nN { #2 { '#3 } } #3 }
26760   }
26761 \cs_new:Npn \__char_change_case:nN #1#2
26762   {
26763     \int_compare:nNnTF {#1} = 0
26764       {#2}
26765       { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
26766   }
26767 \bool_lazy_or:nnF { \sys_if_engine_luatex_p: } { \sys_if_engine_xetex_p: }
26768   {
26769     \cs_set_eq:NN \__char_change_case:nN \use_i:nn
26770   }
```

(End definition for \char\_lower\_case:N and others. These functions are documented on page 259.)

\char\_codepoint\_to\_bytes:n

\\_\_char\_codepoint\_to\_bytes\_auxi:n

\\_\_char\_codepoint\_to\_bytes\_auxii:Nnn

\\_\_char\_codepoint\_to\_bytes\_auxiii:n

\\_\_char\_codepoint\_to\_bytes\_outputi:nw

\\_\_char\_codepoint\_to\_bytes\_outputii:nw

\\_\_char\_codepoint\_to\_bytes\_outputiii:nw

\\_\_char\_codepoint\_to\_bytes\_outputiv:nw

\\_\_char\_codepoint\_to\_bytes\_outputv:nw

\\_\_char\_codepoint\_to\_bytes\_output:nnn

\\_\_char\_codepoint\_to\_bytes\_output:fnn

\\_\_char\_codepoint\_to\_bytes\_end:

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.

```
26771 \cs_new:Npn \char_codepoint_to_bytes:n #1
26772   {
26773     \exp_args:Nf \__char_codepoint_to_bytes_auxi:n
26774       { \int_eval:n {#1} }
26775   }
26776 \cs_new:Npn \__char_codepoint_to_bytes_auxi:n #1
26777   {
26778     \if_int_compare:w #1 > "80 \exp_stop_f:
26779       \if_int_compare:w #1 < "800 \exp_stop_f:
2680         \__char_codepoint_to_bytes_outputi:nw
2681           { \__char_codepoint_to_bytes_auxii:Nnn C {#1} { 64 } }
2682         \__char_codepoint_to_bytes_outputii:nw
2683           { \__char_codepoint_to_bytes_auxiii:n {#1} }
```

```

26784 \else:
26785   \if_int_compare:w #1 < "10000 \exp_stop_f:
26786     \__char_codepoint_to_bytes_outputi:nw
26787       { \__char_codepoint_to_bytes_auxii:Nnn E {#1} { 64 * 64 } }
26788     \__char_codepoint_to_bytes_outputii:nw
26789       {
26790         \__char_codepoint_to_bytes_auxiii:n
26791           { \int_div_truncate:nn {#1} { 64 } }
26792         }
26793       \__char_codepoint_to_bytes_outputiii:nw
26794         { \__char_codepoint_to_bytes_auxiii:n {#1} }
26795     \else:
26796       \__char_codepoint_to_bytes_outputi:nw
26797         {
26798           \__char_codepoint_to_bytes_auxii:Nnn F
26799             {#1} { 64 * 64 * 64 }
26800         }
26801       \__char_codepoint_to_bytes_outputii:nw
26802         {
26803           \__char_codepoint_to_bytes_auxiii:n
26804             { \int_div_truncate:nn {#1} { 64 * 64 } }
26805         }
26806       \__char_codepoint_to_bytes_outputiii:nw
26807         {
26808           \__char_codepoint_to_bytes_auxiii:n
26809             { \int_div_truncate:nn {#1} { 64 } }
26810         }
26811       \__char_codepoint_to_bytes_outputiv:nw
26812         { \__char_codepoint_to_bytes_auxiii:n {#1} }
26813     \fi:
26814   \fi:
26815 \else:
26816   \__char_codepoint_to_bytes_outputi:nw {#1}
26817   \fi:
26818   \__char_codepoint_to_bytes_end: { } { } { } { }
26819 }
26820 \cs_new:Npn \__char_codepoint_to_bytes_auxii:Nnn #1#2#3
26821   { "#10 + \int_div_truncate:nn {#2} {#3} }
26822 \cs_new:Npn \__char_codepoint_to_bytes_auxiii:n #1
26823   { \int_mod:nn {#1} { 64 } + 128 }
26824 \cs_new:Npn \__char_codepoint_to_bytes_outputi:nw
26825   #1 #2 \__char_codepoint_to_bytes_end: #3
26826   { \__char_codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { } {#2} }
26827 \cs_new:Npn \__char_codepoint_to_bytes_outputii:nw
26828   #1 #2 \__char_codepoint_to_bytes_end: #3#4
26829   { \__char_codepoint_to_bytes_output:fnn { \int_eval:n {#1} } { {#3} } {#2} }
26830 \cs_new:Npn \__char_codepoint_to_bytes_outputiii:nw
26831   #1 #2 \__char_codepoint_to_bytes_end: #3#4#5
26832   {
26833     \__char_codepoint_to_bytes_output:fnn
26834       { \int_eval:n {#1} } { {#3} {#4} } {#2}
26835   }
26836 \cs_new:Npn \__char_codepoint_to_bytes_outputiv:nw
26837   #1 #2 \__char_codepoint_to_bytes_end: #3#4#5#6

```

```

26838  {
26839      \__char_codepoint_to_bytes_output:fnn
26840          { \int_eval:n {#1} } { {#3} {#4} {#5} } {#2}
26841  }
26842 \cs_new:Npn \__char_codepoint_to_bytes_output:nnn #1#2#3
26843  {
26844      #3
26845      \__char_codepoint_to_bytes_end: #2 {#1}
26846  }
26847 \cs_generate_variant:Nn \__char_codepoint_to_bytes_output:nnn { f }
26848 \cs_new:Npn \__char_codepoint_to_bytes_end: { }

(End definition for \char_codepoint_to_bytes:n and others. This function is documented on page 259.)
```

26849 ⟨@=tl⟩

\tl\_if\_head\_eq\_catcode:oNTF Extra variants.

```
26850 \cs_generate_variant:Nn \tl_if_head_eq_catcode:nNTF { o }
```

(End definition for \tl\_if\_head\_eq\_catcode:nNTF. This function is documented on page 46.)

\tl\_lower\_case:n  
\tl\_upper\_case:n  
\tl\_mixed\_case:n  
\tl\_lower\_case:nn  
\tl\_upper\_case:nn  
\tl\_mixed\_case:nn

The user level functions here are all wrappers around the internal functions for case changing.

```

26851 \cs_new:Npn \tl_lower_case:n { \__tl_change_case:nnn { lower } { } }
26852 \cs_new:Npn \tl_upper_case:n { \__tl_change_case:nnn { upper } { } }
26853 \cs_new:Npn \tl_mixed_case:n { \__tl_change_case:nnn { mixed } { } }
26854 \cs_new:Npn \tl_lower_case:nn { \__tl_change_case:nnn { lower } }
26855 \cs_new:Npn \tl_upper_case:nn { \__tl_change_case:nnn { upper } }
26856 \cs_new:Npn \tl_mixed_case:nn { \__tl_change_case:nnn { mixed } }
```

(End definition for \tl\_lower\_case:n and others. These functions are documented on page 254.)

\\_\_tl\_change\_case:nnn  
\\_\_tl\_change\_case\_aux:nnn  
\\_\_tl\_change\_case\_loop:wnn  
\\_\_tl\_change\_case\_output:nnn  
\\_\_tl\_change\_case\_output:Vnn  
\\_\_tl\_change\_case\_output:own  
\\_\_tl\_change\_case\_output:vnn  
\\_\_tl\_change\_case\_output:fwn  
\\_\_tl\_change\_case\_end:wn  
\\_\_tl\_change\_case\_group:nnnn  
\\_\_tl\_change\_case\_group\_lower:nnnn  
\\_\_tl\_change\_case\_group\_upper:nnnn  
\\_\_tl\_change\_case\_group\_mixed:nnnn  
\\_\_tl\_change\_case\_space:wnn  
\\_\_tl\_change\_case\_N\_type:Nwnn  
\\_\_tl\_change\_case\_N\_type:NNNnnn  
\\_\_tl\_change\_case\_math:NNNnnn  
\\_\_tl\_change\_case\_math\_loop:WNnnn  
\\_\_tl\_change\_case\_math:WnnNnn  
\\_\_tl\_change\_case\_math\_group:wnNnnn  
\\_\_tl\_change\_case\_math\_space:WNnnn  
\\_\_tl\_change\_case\_N\_type:Nnnn  
\\_\_tl\_change\_case\_char\_lower:Nnn  
\\_\_tl\_change\_case\_char\_upper:Nnn  
\\_\_tl\_change\_case\_char\_mixed:Nnn  
\\_\_tl\_change\_case\_char:nN  
\\_\_tl\_change\_case\_char\_UTFviii:nNN  
\\_\_tl\_change\_case\_char\_UTFviii:NNNN  
\\_\_tl\_change\_case\_char\_UTFviii:nNNNN  
\\_\_tl\_change\_case\_char\_UTFviii:nn  
\\_\_tl\_change\_case\_cs\_letterlike:Nn

The mechanism for the core conversion of case is based on the idea that we can use a loop to grab the entire token list plus a quark: the latter is used as an end marker and to avoid any brace stripping. Depending on the nature of the first item in the grabbed argument, it can either be processed as a single token, treated as a group or treated as a space. These different cases all work by re-reading #1 in the appropriate way, hence the repetition of #1 \q\_recursion\_stop.

```

26857 \cs_new:Npn \__tl_change_case:nnn #1#2#3
26858  {
26859      \__kernel_exp_not:w \exp_after:wN
26860          {
26861              \exp:w
26862              \__tl_change_case_aux:nnn {#1} {#2} {#3}
26863          }
26864      }
26865 \cs_new:Npn \__tl_change_case_aux:nnn #1#2#3
26866  {
26867      \group_align_safe_begin:
26868      \__tl_change_case_loop:wnn
26869          #3 \q_recursion_tail \q_recursion_stop {#1} {#2}
26870          \__tl_change_case_result:n { }
26871      }
26872 \cs_new:Npn \__tl_change_case_loop:wnn #1 \q_recursion_stop
```

```

26873   {
26874     \tl_if_head_is_N_type:nTF {#1}
26875       { \__tl_change_case_N_type:Nnnn }
26876       {
26877         \tl_if_head_is_group:nTF {#1}
26878           { \__tl_change_case_group:nwnn }
26879           { \__tl_change_case_space:wnn }
26880       }
26881     #1 \q_recursion_stop
26882   }

```

Earlier versions of the code where only x-type expandable rather than f-type: this causes issues with nesting and so the slight performance hit is taken for a better outcome in usability terms. Setting up for f-type expandability has two requirements: a marker token after the main loop (see above) and a mechanism to “load” and finalise the result. That is handled in the code below, which includes the necessary material to end the \exp:w expansion.

```

26883 \cs_new:Npn \__tl_change_case_output:nwn #1#2 \__tl_change_case_result:n #3
26884   { #2 \__tl_change_case_result:n { #3 #1 } }
26885 \cs_generate_variant:Nn \__tl_change_case_output:nwn { V , o , v , f }
26886 \cs_new:Npn \__tl_change_case_end:wn #1 \__tl_change_case_result:n #2
26887   {
26888     \group_align_safe_end:
26889     \exp_end:
26890     #2
26891   }

```

Handling for the cases where the current argument is a brace group or a space is relatively easy. For the brace case, the routine works recursively, using the expandability of the mechanism to ensure that the result is finalised before storage. For the space case it is simply a question of removing the space in the input and storing it in the output. In both cases, and indeed for the N-type grabber, after removing the current item from the input \\_\_tl\_change\_case\_loop:wnn is inserted in front of the remaining tokens.

```

26892 \cs_new:Npn \__tl_change_case_group:nwnn #1#2 \q_recursion_stop #3#4
26893   {
26894     \use:c { __tl_change_case_group_ #3 : nnnn } {#1} {#2} {#3} {#4}
26895   }
26896 \cs_new:Npn \__tl_change_case_group_lower:nnnn #1#2#3#4
26897   {
26898     \__tl_change_case_output:own
26899     {
26900       \exp_after:wN
26901       {
26902         \exp_w
26903         \__tl_change_case_aux:nnn {#3} {#4} {#1}
26904       }
26905     }
26906     \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
26907   }
26908 \cs_new_eq:NN \__tl_change_case_group_upper:nnnn
26909   \__tl_change_case_group_lower:nnnn

```

For the “mixed” case, a group is taken as forcing a switch to lower casing. That means we need a separate auxiliary. (Tracking whether we have found a first character inside a

group and transferring the information out looks pretty horrible.)

```

26910 \cs_new:Npn \__tl_change_case_group_mixed:nnnn #1#2#3#4
26911   {
26912     \__tl_change_case_output:own
26913       {
26914         \exp_after:wN
26915           {
26916             \exp:w
26917               \__tl_change_case_aux:nnn {#3} {#4} {#1}
26918             }
26919           }
26920         \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower } {#4}
26921       }
26922 \exp_last_unbraced:NNo \cs_new:Npn \__tl_change_case_space:wnn \c_space_tl
26923   {
26924     \__tl_change_case_output:nwn { ~ }
26925     \__tl_change_case_loop:wnn
26926   }

```

For N-type arguments there are several stages to the approach. First, a simply check for the end-of-input marker, which if found triggers the final clean up and output step. Assuming that is not the case, the first check is for math-mode escaping: this test can encompass control sequences or other N-type tokens so is handled up front.

```

26927 \cs_new:Npn \__tl_change_case_N_type:Nwnn #1#2 \q_recursion_stop
26928   {
26929     \quark_if_recursion_tail_stop_do:Nn #1
26930       { \__tl_change_case_end:wn }
26931     \exp_after:wN \__tl_change_case_N_type:NNNnnn
26932       \exp_after:wN #1 \l_tl_case_change_math_tl
26933       \q_recursion_tail ? \q_recursion_stop {#2}
26934   }

```

Looking for math mode escape first requires a loop over the possible token pairs to see if the current input (#1) matches an open-math case (#2). If it does then this test loop is ended and a new input-gathering one is begun. The latter simply transfers material from the input to the output without any expansion, testing each N-type token to see if it matches the close-math case required. If that is the situation then the “math loop” stops and resumes the main loop: as that might be either the standard case-changing one or the mixed-case alternative, it is not hard-coded into the math loop but is rather passed as argument #3 to `\__tl_change_case_math:NNNnnn`. 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).

```

26935 \cs_new:Npn \__tl_change_case_N_type:NNNnnn #1#2#3
26936   {
26937     \quark_if_recursion_tail_stop_do:Nn #2
26938       { \__tl_change_case_N_type:Nnnm #1 }
26939     \token_if_eq_meaning:NNTF #1 #2
26940       {
26941         \use_i_delimit_by_q_recursion_stop:nw
26942           {
26943             \__tl_change_case_math:NNNnnn
26944               #1 #3 \__tl_change_case_loop:wnn
26945             }
26946       }

```

```

26947      { \__tl_change_case_N_type:NNNnnn #1 }
26948    }
26949 \cs_new:Npn \__tl_change_case_math:NNNnnn #1#2#3#4
26950  {
26951    \__tl_change_case_output:nwn {#1}
26952    \__tl_change_case_math_loop:wNNnn #4 \q_recursion_stop #2 #3
26953  }
26954 \cs_new:Npn \__tl_change_case_math_loop:wNNnn #1 \q_recursion_stop
26955  {
26956    \tl_if_head_is_N_type:nTF {#1}
26957    { \__tl_change_case_math:NwNNnn }
26958    {
26959      \tl_if_head_is_group:nTF {#1}
26960      { \__tl_change_case_math_group:nwNNnn }
26961      { \__tl_change_case_math_space:wNNnn }
26962    }
26963    #1 \q_recursion_stop
26964  }
26965 \cs_new:Npn \__tl_change_case_math:NwNNnn #1#2 \q_recursion_stop #3#4
26966  {
26967    \token_if_eq_meaning:NNTF \q_recursion_tail #1
26968    { \__tl_change_case_end:wn }
26969    {
26970      \__tl_change_case_output:nwn {#1}
26971      \token_if_eq_meaning:NNTF #1 #3
26972      { #4 #2 \q_recursion_stop }
26973      { \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop #3#4 }
26974    }
26975  }
26976 \cs_new:Npn \__tl_change_case_math_group:nwNNnn #1#2 \q_recursion_stop
26977  {
26978    \__tl_change_case_output:nwn { {#1} }
26979    \__tl_change_case_math_loop:wNNnn #2 \q_recursion_stop
26980  }
26981 \exp_last_unbraced:NNo
26982   \cs_new:Npn \__tl_change_case_math_space:wNNnn \c_space_tl
26983  {
26984    \__tl_change_case_output:nwn { ~ }
26985    \__tl_change_case_math_loop:wNNnn
26986  }

```

Once potential math-mode cases are filtered out the next stage is to test if the token grabbed is a control sequence: they cannot be used in the lookup table and also may require expansion. At this stage the loop code starting `\__tl_change_case_loop:wnn` is inserted: all subsequent steps in the code which need a look-ahead are coded to rely on this and thus have w-type arguments if they may do a look-ahead.

```

26987 \cs_new:Npn \__tl_change_case_N_type:Nnnn #1#2#3#4
26988  {
26989    \token_if_cs:NTF #1
26990    { \__tl_change_case_cs_letterlike:Nn #1 {#3} }
26991    { \use:c { __tl_change_case_char_ #3 :Nnn } #1 {#3} {#4} }
26992    \__tl_change_case_loop:wnn #2 \q_recursion_stop {#3} {#4}
26993  }

```

For character tokens there are some special cases to deal with then the majority of

changes are covered by using the T<sub>E</sub>X data as a lookup along with expandable character generation. This avoids needing a very large number of macros or (as seen in earlier versions) a somewhat tricky split of the characters into various blocks. Notice that the special case code may do a look-ahead so requires a final w-type argument whereas the core lookup table does not and also guarantees an output so f-type expansion may be used to obtain the case-changed result.

```

26994 \cs_new:Npn \__tl_change_case_char_lower:Nnn #1#2#3
26995 {
26996   \cs_if_exist_use:cF { __tl_change_case_ #2 _ #3 :Nnw }
26997   { \use_i:nn }
26998   #1
26999   {
27000     \use:c { __tl_change_case_ #2 _ sigma:Nnw } #1
27001     { \__tl_change_case_char:nN {#2} #1 }
27002   }
27003 }
27004 \cs_new_eq:NN \__tl_change_case_char_upper:Nnn
27005   \__tl_change_case_char_lower:Nnn

```

For mixed case, the code is somewhat different: there is a need to look up both mixed and upper case chars and we have to cover the situation where there is a character to skip over.

```

27006 \cs_new:Npn \__tl_change_case_char_mixed:Nnn #1#2#3
27007 {
27008   \__tl_change_case_mixed_switch:w
27009   \cs_if_exist_use:cF { __tl_change_case_mixed_ #3 :Nnw }
27010   {
27011     \cs_if_exist_use:cF { __tl_change_case_upper_ #3 :Nnw }
27012     { \use_i:nn }
27013   }
27014   #1
27015   { \__tl_change_case_mixed_skip:N #1 }
27016 }

```

For Unicode engines we can handle all characters directly. However, for the 8-bit engines the aim is to deal with (a subset of) Unicode (UTF-8) input. They deal with that by making the upper half of the range active, so we look for that and if found work out how many UTF-8 octets there are to deal with. Those can then be grabbed to reconstruct the full Unicode character, which is then used in a lookup. (As will become obvious below, there is no intention here of covering all of Unicode.)

```

27017 \bool_lazy_or:nnTF
27018   { \sys_if_engine_luatex_p: }
27019   { \sys_if_engine_xetex_p: }
27020   {
27021     \cs_new:Npn \__tl_change_case_char:nN #1#2
27022     {
27023       \__tl_change_case_output:fwn
27024       { \use:c { char_ #1 _case:N } #2 }
27025     }
27026   }
27027   {
27028     \cs_new:Npn \__tl_change_case_char:nN #1#2
27029     {
27030       \int_compare:nNnTF { '#2 } > { "80 }

```

```

27031   {
27032     \int_compare:nNnTF { '#2 } < { "E0 "
27033       { \__tl_change_case_char_UTFviii:nNNN {#1} #2 }
27034       {
27035         \int_compare:nNnTF { '#2 } < { "F0 "
27036           { \__tl_change_case_char_UTFviii:nNNNN {#1} #2 }
27037           { \__tl_change_case_char_UTFviii:nNNNNN {#1} #2 }
27038         }
27039       }
27040     {
27041       \__tl_change_case_output:fwn
27042         { \use:c { char_ #1 _case:N } #2 }
27043     }
27044   }
27045 }
```

To allow for the special case of mixed case, we insert here a action-dependent auxiliary.

```

27046 \bool_lazy_or:nnF
27047   { \sys_if_engine_luatex_p: }
27048   { \sys_if_engine_xetex_p: }
27049   {
27050     \cs_new:Npn \__tl_change_case_char_UTFviii:nNNN #1#2#3#4
27051       { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4} #3 }
27052     \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNN #1#2#3#4#5
27053       { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5} #3 }
27054     \cs_new:Npn \__tl_change_case_char_UTFviii:nNNNNN #1#2#3#4#5#6
27055       { \__tl_change_case_char_UTFviii:nnN {#1} {#2#4#5#6} #3 }
27056     \cs_new:Npn \__tl_change_case_char_UTFviii:nnN #1#2#3
27057       {
27058         \cs_if_exist:cTF { c__tl_ #1 _case_ \tl_to_str:n {#2} _tl }
27059           {
27060             \__tl_change_case_output:vvn
27061               { c__tl_ #1 _case_ \tl_to_str:n {#2} _tl }
27062             }
27063             { \__tl_change_case_output:nvn {#2} }
27064           #3
27065         }
27066       }
```

Before dealing with general control sequences there are the special ones to deal with. Letter-like control sequences are a simple look-up, while for accents the loop is much as done elsewhere. Notice that we have a no-op test to make sure there is no unexpected expansion of letter-like input. The split into two parts here allows us to insert the “switch” code for mixed casing.

```

27067 \cs_new:Npn \__tl_change_case_cs_letterlike:Nn #1#2
27068   {
27069     \str_if_eq:nnTF {#2} { mixed }
27070     {
27071       \__tl_change_case_cs_letterlike:NnN #1 { upper }
27072         \__tl_change_case_mixed_switch:w
27073     }
27074     { \__tl_change_case_cs_letterlike:NnN #1 {#2} \prg_do_nothing: }
27075   }
27076 \cs_new:Npn \__tl_change_case_cs_letterlike:NnN #1#2#3
27077   {
```

```

27078     \cs_if_exist:cTF { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
27079     {
27080         \__tl_change_case_output:vvn
27081             { c__tl_change_case_ #2 _ \token_to_str:N #1 _tl }
27082         #3
27083     }
27084     {
27085         \cs_if_exist:cTF
27086             {
27087                 c__tl_change_case_
27088                     \str_if_eq:nnTF {#2} { lower } { upper } { lower }
27089                         _ \token_to_str:N #1 _tl
27090             }
27091             {
27092                 \__tl_change_case_output:nwn {#1}
27093                     #3
27094             }
27095             {
27096                 \exp_after:wN \__tl_change_case_cs_accents:NN
27097                     \exp_after:wN #1 \l_tl_case_change_accents_tl
27098                         \q_recursion_tail \q_recursion_stop
27099             }
27100         }
27101     }
27102     \cs_new:Npn \__tl_change_case_cs_accents:NN #1#2
27103     {
27104         \quark_if_recursion_tail_stop_do:Nn #2
27105             { \__tl_change_case_cs:N #1 }
27106         \str_if_eq:nnTF {#1} {#2}
27107             {
27108                 \use_i_delimit_by_q_recursion_stop:nw
27109                     { \__tl_change_case_output:nwn {#1} }
27110             }
27111             { \__tl_change_case_cs_accents:NN #1 }
27112     }

```

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: that comes *after* the loop function which is therefore rearranged. In a L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  context, \protect needs to be treated specially, to prevent expansion of the next token but output it without braces.

```

27113 \cs_new:Npn \__tl_change_case_cs:N #1
27114     {
27115     (*package)
27116         \str_if_eq:nnTF {#1} { \protect } { \__tl_change_case_protect:wNN }
27117     (/package)
27118         \exp_after:wN \__tl_change_case_cs:NN
27119             \exp_after:wN #1 \l_tl_case_change_exclude_tl
27120                 \q_recursion_tail \q_recursion_stop
27121     }
27122     \cs_new:Npn \__tl_change_case_cs:NN #1#2
27123     {
27124         \quark_if_recursion_tail_stop_do:Nn #2
27125             {

```

```

27126     \__tl_change_case_cs_expand:Nnw #1
27127         { \__tl_change_case_output:nwn {#1} }
27128     }
27129 \str_if_eq:nnTF {#1} {#2}
27130     {
27131         \use_i_delimit_by_q_recursion_stop:nw
27132             { \__tl_change_case_cs>NNn #1 }
27133     }
27134     { \__tl_change_case_cs:NN #1 }
27135 }
27136 \cs_new:Npn \__tl_change_case_cs>NNn #1#2#3
27137     {
27138         \__tl_change_case_output:nwn { #1 {#3} }
27139         #2
27140     }
27141 (*package)
27142 \cs_new:Npn \__tl_change_case_protect:wNN #1 \q_recursion_stop #2 #3
27143     { \__tl_change_case_output:nwn { \protect #3 } #2 }
27144 
```

When a control sequence is not on the exclude list the other test if to see if it is expandable. Once again, if there is a hit then the loop function is grabbed as part of the clean-up and reinserted before the now expanded material. The test for expandability has to check for end-of-recursion as it is needed by the look-ahead code which might hit the end of the input. The test is done in two parts as \bool\_if:nTF would choke if #1 was (!

```

27145 \cs_new:Npn \__tl_change_case_if_expandable:NTF #1
27146     {
27147         \token_if_expandable:NTF #1
27148         {
27149             \bool_lazy_any:nTF
27150             {
27151                 { \token_if_eq_meaning_p:NN \q_recursion_tail #1 }
27152                 { \token_if_protected_macro_p:N #1 }
27153                 { \token_if_protected_long_macro_p:N #1 }
27154             }
27155             { \use_ii:nn }
27156             { \use_i:nn }
27157         }
27158         { \use_ii:nn }
27159     }
27160 \cs_new:Npn \__tl_change_case_cs_expand:Nnw #1#2
27161     {
27162         \__tl_change_case_if_expandable:NTF #1
27163             { \__tl_change_case_cs_expand:NN #1 }
27164             { #2 }
27165     }
27166 \cs_new:Npn \__tl_change_case_cs_expand:NN #1#2
27167     { \exp_after:wN #2 #1 }

```

For mixed case, there is an additional list of exceptions to deal with: once that is sorted, we can move on back to the main loop.

```

27168 \cs_new:Npn \__tl_change_case_mixed_skip:N #1
27169     {
27170         \exp_after:wN \__tl_change_case_mixed_skip>NN

```

```

27171     \exp_after:wN #1 \l_tl_mixed_case_ignore_tl
27172     \q_recursion_tail \q_recursion_stop
27173   }
27174 \cs_new:Npn \__tl_change_case_mixed_skip:NN #1#2
27175   {
27176     \quark_if_recursion_tail_stop_do:nn {#2}
27177     { \__tl_change_case_char:nN { mixed } #1 }
27178     \int_compare:nNnT { '#1 } = { '#2 }
27179     {
27180       \use_i_delimit_by_q_recursion_stop:nw
27181       {
27182         \__tl_change_case_output:nwn {#1}
27183         \__tl_change_case_mixed_skip_tidy:Nwn
27184       }
27185     }
27186   \__tl_change_case_mixed_skip:NN #1
27187 }
27188 \cs_new:Npn \__tl_change_case_mixed_skip_tidy:Nwn #1#2 \q_recursion_stop #3
27189 {
27190   \__tl_change_case_loop:wnn #2 \q_recursion_stop { mixed }
27191 }

```

Needed to switch from mixed to lower casing when we have found a first character in the former mode.

```

27192 \cs_new:Npn \__tl_change_case_mixed_switch:w
27193   #1 \__tl_change_case_loop:wnn #2 \q_recursion_stop #3
27194   {
27195     #1
27196     \__tl_change_case_loop:wnn #2 \q_recursion_stop { lower }
27197   }

```

*(End definition for \\_\_tl\_change\_case:nnn and others.)*

\\_\_tl\_change\_case\_lower\_sigma:Nnw  
\\_\_tl\_change\_case\_lower\_sigma:w  
\\_\_tl\_change\_case\_lower\_sigma:Nw  
\\_\_tl\_change\_case\_upper\_sigma:Nnw

If the current char is an upper case 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.

```

27198 \cs_new:Npn \__tl_change_case_lower_sigma:Nnw #1#2#3#4 \q_recursion_stop
27199   {
27200     \int_compare:nNnTF { '#1 } = { "03A3" }
27201     {
27202       \__tl_change_case_output:fwn
27203       { \__tl_change_case_lower_sigma:w #4 \q_recursion_stop }
27204     }
27205     {#2}
27206     #3 #4 \q_recursion_stop
27207   }
27208 \cs_new:Npn \__tl_change_case_lower_sigma:w #1 \q_recursion_stop
27209   {
27210     \tl_if_head_is_N_type:nTF {#1}
27211     { \__tl_change_case_lower_sigma:Nw #1 \q_recursion_stop }
27212     { \c__tl_final_sigma_tl }
27213   }
27214 \cs_new:Npn \__tl_change_case_lower_sigma:Nw #1#2 \q_recursion_stop
27215   {
27216     \__tl_change_case_if_expandable:NTF #1
27217   }

```

```

27218     \exp_after:wN \__tl_change_case_lower_sigma:w #1
27219         #2 \q_recursion_stop
27220     }
27221     {
27222         \token_if_letter:NTF #1
27223             { \c__tl_std_sigma_tl }
27224             { \c__tl_final_sigma_tl }
27225     }
27226 }
```

Simply skip to the final step for upper casing.

```
27227 \cs_new_eq:NN \__tl_change_case_upper_sigma:Nnw \use_ii:nn
```

*(End definition for \\_\_tl\_change\_case\_lower\_sigma:Nnw and others.)*

\\_\_tl\_change\_case\_lower\_tr:Nnw  
\\_\_tl\_change\_case\_lower\_tr\_auxi:Nw  
\\_\_tl\_change\_case\_lower\_tr\_auxii:Nw

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.

```

27228 \bool_lazy_or:nnTF
27229   { \sys_if_engine_luatex_p: }
27230   { \sys_if_engine_xetex_p: }
27231   {
27232     \cs_new:Npn \__tl_change_case_lower_tr:Nnw #1#2
27233     {
27234       \int_compare:nNnTF { '#1 } = { "0049 }
27235           { \__tl_change_case_lower_tr_auxi:Nw }
27236           {
27237             \int_compare:nNnTF { '#1 } = { "0130 }
27238               { \__tl_change_case_output:nwn { i } }
27239               {#2}
27240           }
27241     }
```

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, which is done by the \use\_i:nn (it grabs \\_\_tl\_change\_case\_loop:wn and the dot-above char and discards the latter).

```

27242 \cs_new:Npn \__tl_change_case_lower_tr_auxi:Nw #1#2 \q_recursion_stop
27243   {
27244     \tl_if_head_is_N_type:nTF {#2}
27245       { \__tl_change_case_lower_tr_auxii:Nw #2 \q_recursion_stop }
27246       { \__tl_change_case_output:Vwn \c__tl_dotless_i_tl }
27247       #1 #2 \q_recursion_stop
27248   }
27249 \cs_new:Npn \__tl_change_case_lower_tr_auxii:Nw #1#2 \q_recursion_stop
27250   {
27251     \__tl_change_case_if_expandable:NTF #1
27252     {
27253       \exp_after:wN \__tl_change_case_lower_tr_auxi:Nw #1
27254         #2 \q_recursion_stop
27255     }
27256     {
27257       \bool_lazy_or:nnTF
27258         { \token_if_cs_p:N #1 }
```

```

27259     { ! \int_compare_p:nNn { '#1 } = { "0307" } }
27260     { \_tl_change_case_output:Vwn \c__tl_dotless_i_lt }
27261     {
27262         \_tl_change_case_output:nwn { i }
27263         \use_i:nn
27264     }
27265 }
27266 }
27267 }

```

For 8-bit engines, dot-above is not available so there is a simple test for an upper-case I. Then we can look for the UTF-8 representation of an upper case dotted-I without the combining char. If it's not there, preserve the UTF-8 sequence as-is.

```

27268   {
27269     \cs_new:Npn \_tl_change_case_lower_tr:Nnw #1#2
27270     {
27271         \int_compare:nNnTF { '#1 } = { "0049" }
27272         { \_tl_change_case_output:Vwn \c__tl_dotless_i_lt }
27273         {
27274             \int_compare:nNnTF { '#1 } = { 196 }
27275             { \_tl_change_case_lower_tr_auxi:Nw #1 {#2} }
27276             {#2}
27277         }
27278     }
27279     \cs_new:Npn \_tl_change_case_lower_tr_auxi:Nw #1#2#3#4
27280     {
27281         \int_compare:nNnTF { '#4 } = { 176 }
27282         {
27283             \_tl_change_case_output:nwn { i }
27284             #3
27285         }
27286         {
27287             #2
27288             #3 #4
27289         }
27290     }
27291 }

```

Upper casing is easier: just one exception with no context.

```

27292 \cs_new:Npn \_tl_change_case_upper_tr:Nnw #1#2
27293 {
27294     \int_compare:nNnTF { '#1 } = { "0069" }
27295     { \_tl_change_case_output:Vwn \c__tl_dotted_I_lt }
27296     {#2}
27297 }

```

Straight copies.

```

27298 \cs_new_eq:NN \_tl_change_case_lower_az:Nnw \_tl_change_case_lower_tr:Nnw
27299 \cs_new_eq:NN \_tl_change_case_upper_az:Nnw \_tl_change_case_upper_tr:Nnw

```

*(End definition for \\_tl\_change\_case\_lower\_tr:Nnw and others.)*

\\_tl\_change\_case\_lower\_lt:Nnw  
\\_tl\_change\_case\_lower\_lt:nNnw  
\\_tl\_change\_case\_lower\_lt:nnw  
\\_tl\_change\_case\_lower\_lt:Nw  
\\_tl\_change\_case\_lower\_lt:NNw  
\\_tl\_change\_case\_upper\_lt:Nnw  
\\_tl\_change\_case\_upper\_lt:nnw  
\\_tl\_change\_case\_upper\_lt:Nw  
\\_tl\_change\_case\_upper\_lt:NNw

For Lithuanian, the issue to be dealt with is dots over lower case letters: these should be present if there is another accent. That means that there is some work to do when lower casing I and J. The first step is a simple match attempt: \c\_\_tl\_accents\_lt\_lt

contains accented upper case letters which should gain a dot-above char in their lower case form. This is done using f-type expansion so only one pass is needed to find if it works or not. If there was no hit, the second stage is to check for I, J and I-ogonek, and if the current char is a match to look for a following accent.

```

27300 \cs_new:Npn \__tl_change_case_lower_lt:Nnw #1
27301 {
27302     \exp_args:Nf \__tl_change_case_lower_lt:nNnw
27303     { \str_case:nVF #1 \c__tl_accents_lt_tl \exp_stop_f: }
27304     #1
27305 }
27306 \cs_new:Npn \__tl_change_case_lower_lt:nNnw #1#2
27307 {
27308     \tl_if_blank:nTF {#1}
27309     {
27310         \exp_args:Nf \__tl_change_case_lower_lt:nnw
27311         {
27312             \int_case:nnF {'#2}
27313             {
27314                 { "0049 } i
27315                 { "004A } j
27316                 { "012E } \c__tl_i_ogonek_tl
27317             }
27318             \exp_stop_f:
27319         }
27320     }
27321     {
27322         \__tl_change_case_output:nwn {#1}
27323         \use_none:n
27324     }
27325 }
27326 \cs_new:Npn \__tl_change_case_lower_lt:nnw #1#2
27327 {
27328     \tl_if_blank:nTF {#1}
27329     {#2}
27330     {
27331         \__tl_change_case_output:nwn {#1}
27332         \__tl_change_case_lower_lt:Nw
27333     }
27334 }

```

Grab the next char and see if it is one of the accents used in Lithuanian: if it is, add the dot-above char into the output.

```

27335 \cs_new:Npn \__tl_change_case_lower_lt:Nw #1#2 \q_recursion_stop
27336 {
27337     \tl_if_head_is_N_type:nT {#2}
27338     { \__tl_change_case_lower_lt:NNw }
27339     #1 #2 \q_recursion_stop
27340 }
27341 \cs_new:Npn \__tl_change_case_lower_lt:NNw #1#2#3 \q_recursion_stop
27342 {
27343     \__tl_change_case_if_expandable:NTF #2
27344     {
27345         \exp_after:wN \__tl_change_case_lower_lt:Nw \exp_after:wN #1 #2
27346         #3 \q_recursion_stop

```

```

27347 }
27348 {
27349     \bool_lazy_and:nnt
27350         { ! \token_if_cs_p:N #2 }
27351     {
27352         \bool_lazy_any_p:n
27353         {
27354             { \int_compare_p:nNn { '#2 } = { "0300" } }
27355             { \int_compare_p:nNn { '#2 } = { "0301" } }
27356             { \int_compare_p:nNn { '#2 } = { "0303" } }
27357         }
27358     }
27359     { \__tl_change_case_output:Vwn \c__tl_dot_above_tl }
27360     #1 #2#3 \q_recursion_stop
27361 }
27362 }
```

For upper casing, the test required is for a dot-above char after an I, J or I-ogonek. First a test for the appropriate letter, and if found a look-ahead and potentially one token dropped.

```

27363 \cs_new:Npn \__tl_change_case_upper_lt:Nnw #1
27364 {
27365     \exp_args:Nf \__tl_change_case_upper_lt:nnw
27366     {
27367         \int_case:nnF {'#1}
27368         {
27369             { "0069" } I
27370             { "006A" } J
27371             { "012F" } \c__tl_I_ogonek_tl
27372         }
27373         \exp_stop_f:
27374     }
27375 }
27376 \cs_new:Npn \__tl_change_case_upper_lt:nnw #1#2
27377 {
27378     \tl_if_blank:nTF {#1}
27379         {#2}
27380         {
27381             \__tl_change_case_output:nnn {#1}
27382             \__tl_change_case_upper_lt:Nw
27383         }
27384 }
27385 \cs_new:Npn \__tl_change_case_upper_lt:Nw #1#2 \q_recursion_stop
27386 {
27387     \tl_if_head_is_N_type:nT {#2}
27388         { \__tl_change_case_upper_lt>NNw }
27389         #1 #2 \q_recursion_stop
27390     }
27391 \cs_new:Npn \__tl_change_case_upper_lt>NNw #1#2#3 \q_recursion_stop
27392 {
27393     \__tl_change_case_if_expandable:NTF #2
27394     {
27395         \exp_after:wN \__tl_change_case_upper_lt:Nw \exp_after:wN #1 #2
27396             #3 \q_recursion_stop
27397 }
```

```

27397 }
27398 {
27399     \bool_lazy_and:nTF
27400         { ! \token_if_cs_p:N #2 }
27401         { \int_compare_p:nNn { '#2 } = { "0307" } }
27402         { #1 }
27403         { #1 #2 }
27404     #3 \q_recursion_stop
27405   }
27406 }

```

(End definition for `\_tl_change_case_lower_lt:Nnw` and others.)

`\_tl_change_case_upper_de-alt:Nnw` A simple alternative version for German.

```

27407 \cs_new:cpn { \_tl_change_case_upper_de-alt:Nnw } #1#2
27408   {
27409     \int_compare:nNnTF { '#1 } = { 223 }
27410     { \_tl_change_case_output:Vwn \c__tl_upper_Eszett_tl }
27411     { '#2 }
27412   }

```

(End definition for `\_tl_change_case_upper_de-alt:Nnw`.)

`\c__tl_std_sigma_tl` The above needs various special token lists containing pre-formed characters. This set  
`\c__tl_final_sigma_tl` are only available in Unicode engines, with no-op definitions for 8-bit use.

```

27413 \bool_lazy_or:nnTF
27414   { \sys_if_engine_luatex_p: }
27415   { \sys_if_engine_xetex_p: }
27416   {
27417     \group_begin:
27418       \cs_set:Npn \_tl_tmp:n #1
27419         { \char_generate:nn {#1} { \char_value_catcode:n {#1} } }
27420       \tl_const:Nx \c__tl_std_sigma_tl { \_tl_tmp:n { "03C3" } }
27421       \tl_const:Nx \c__tl_final_sigma_tl { \_tl_tmp:n { "03C2" } }
27422       \tl_const:Nx \c__tl_accents_lt_tl
27423         {
27424           \_tl_tmp:n { "00CC" }
27425             {
27426               \_tl_tmp:n { "0069" }
27427               \_tl_tmp:n { "0307" }
27428               \_tl_tmp:n { "0300" }
27429             }
27430           \_tl_tmp:n { "00CD" }
27431             {
27432               \_tl_tmp:n { "0069" }
27433               \_tl_tmp:n { "0307" }
27434               \_tl_tmp:n { "0301" }
27435             }
27436           \_tl_tmp:n { "0128" }
27437             {
27438               \_tl_tmp:n { "0069" }
27439               \_tl_tmp:n { "0307" }
27440               \_tl_tmp:n { "0303" }
27441             }

```

```

27442     }
27443     \tl_const:Nx \c__tl_dot_above_tl    { \_tl_tmp:n { "0307" } }
27444     \tl_const:Nx \c__tl_upper_Eszett_tl { \_tl_tmp:n { "1E9E" } }
27445     \group_end:
27446   }
27447   {
27448     \tl_const:Nn \c__tl_std_sigma_tl    { }
27449     \tl_const:Nn \c__tl_final_sigma_tl { }
27450     \tl_const:Nn \c__tl_accents_lt_tl { }
27451     \tl_const:Nn \c__tl_dot_above_tl    { }
27452     \tl_const:Nn \c__tl_upper_Eszett_tl { }
27453   }

```

(End definition for `\c__tl_std_sigma_tl` and others.)

`\c__tl_dotless_i_tl` For cases where there is an 8-bit option in the T1 font set up, a variant is provided in both cases.

```

27454 \group_begin:
27455   \bool_lazy_or:nnTF
27456   { \sys_if_engine_luatex_p: }
27457   { \sys_if_engine_xetex_p: }
27458   {
27459     \cs_set_protected:Npn \_tl_tmp:w #1#2
27460     {
27461       \tl_const:Nx #1
27462       {
27463         \exp_after:wN \exp_after:wN \exp_after:wN
27464           \exp_not:N \char_generate:nn
27465             {"#2} { \char_value_catcode:n {"#2} }
27466       }
27467     }
27468   }
27469   {
27470     \cs_set_protected:Npn \_tl_tmp:w #1#2
27471     {
27472       \group_begin:
27473         \cs_set_protected:Npn \_tl_tmp:w ##1##2##3##4
27474         {
27475           \tl_const:Nx #1
27476           {
27477             \exp_after:wN \exp_after:wN \exp_after:wN
27478               \exp_not:N \char_generate:nn {##1} { 13 }
27479             \exp_after:wN \exp_after:wN \exp_after:wN
27480               \exp_not:N \char_generate:nn {##2} { 13 }
27481           }
27482         }
27483         \tl_set:Nx \l__tl_internal_a_tl
27484           { \char_codepoint_to_bytes:n {"#2} }
27485           \exp_after:wN \_tl_tmp:w \l__tl_internal_a_tl
27486         \group_end:
27487       }
27488     }
27489   \_tl_tmp:w \c__tl_dotless_i_tl { 0131 }
27490   \_tl_tmp:w \c__tl_dotted_I_tl { 0130 }

```

```

27491   \__tl_tmp:w \c__tl_i_ogonek_tl { 012F }
27492   \__tl_tmp:w \c__tl_I_ogonek_tl { 012E }
27493 \group_end:
```

(End definition for \c\_\_tl\_dotless\_i\_t1 and others.)

For 8-bit engines we now need to define the case-change data for the multi-octet mappings. These need a list of what code points are doable in T1 so the list is hard coded (there's no saving in loading the mappings dynamically). All of the straight-forward ones have two octets, so that is taken as read.

```

27494 \group_begin:
27495   \bool_lazy_or:nnT
27496   { \sys_if_engine_pdftex_p: }
27497   { \sys_if_engine_uptex_p: }
27498   {
27499     \cs_set_protected:Npn \__tl_loop:nn #1#2
27500     {
27501       \quark_if_recursion_tail_stop:n {#1}
27502       \tl_set:Nx \l__tl_internal_a_tl
27503       {
27504         \char_codepoint_to_bytes:n {"#1}
27505         \char_codepoint_to_bytes:n {"#2}
27506       }
27507       \exp_after:wN \__tl_tmp:w \l__tl_internal_a_tl
27508       \__tl_loop:nn
27509     }
27510     \cs_set_protected:Npn \__tl_tmp:nnnn #1#2#3#4#5
27511     {
27512       \tl_const:cx
27513       {
27514         c__tl_ #1 _case_
27515         \char_generate:nn {#2} { 12 }
27516         \char_generate:nn {#3} { 12 }
27517         _tl
27518       }
27519       {
27520         \exp_after:wN \exp_after:wN \exp_after:wN
27521           \exp_not:N \char_generate:nn {#4} { 13 }
27522         \exp_after:wN \exp_after:wN \exp_after:wN
27523           \exp_not:N \char_generate:nn {#5} { 13 }
27524       }
27525     }
27526     \cs_set_protected:Npn \__tl_tmp:w #1#2#3#4#5#6#7#8
27527     {
27528       \tl_const:cx
27529       {
27530         c__tl_lower_case_
27531         \char_generate:nn {#1} { 12 }
27532         \char_generate:nn {#2} { 12 }
27533         _tl
27534       }
27535       {
27536         \exp_after:wN \exp_after:wN \exp_after:wN
27537           \exp_not:N \char_generate:nn {#5} { 13 }
27538         \exp_after:wN \exp_after:wN \exp_after:wN
```

```

27539           \exp_not:N \char_generate:nn {#6} { 13 }
27540           }
27541           \__tl_tmp:nnnn { upper } {#5} {#6} {#1} {#2}
27542           \__tl_tmp:nnnn { mixed } {#5} {#6} {#1} {#2}
27543           }
27544           \__tl_loop:nn
27545           { OOC0 } { OOE0 }
27546           { OOC2 } { OOE2 }
27547           { OOC3 } { OOE3 }
27548           { OOC4 } { OOE4 }
27549           { OOC5 } { OOE5 }
27550           { OOC6 } { OOE6 }
27551           { OOC7 } { OOE7 }
27552           { OOC8 } { OOE8 }
27553           { OOC9 } { OOE9 }
27554           { OOCA } { OOEA }
27555           { OOCD } { OOEB }
27556           { OOCC } { OOEC }
27557           { OOCD } { OOED }
27558           { OOCE } { OOEE }
27559           { OOCF } { OOEF }
27560           { OODO } { OOF0 }
27561           { OOD1 } { OOF1 }
27562           { OOD2 } { OOF2 }
27563           { OOD3 } { OOF3 }
27564           { OOD4 } { OOF4 }
27565           { OOD5 } { OOF5 }
27566           { OOD6 } { OOF6 }
27567           { OOD8 } { OOF8 }
27568           { OOD9 } { OOF9 }
27569           { OODA } { OOFA }
27570           { OODB } { OOFB }
27571           { OODC } { OOFC }
27572           { OODD } { OOFD }
27573           { OODE } { OOFE }
27574           { O100 } { O101 }
27575           { O102 } { O103 }
27576           { O104 } { O105 }
27577           { O106 } { O107 }
27578           { O108 } { O109 }
27579           { O10A } { O10B }
27580           { O10C } { O10D }
27581           { O10E } { O10F }
27582           { O110 } { O111 }
27583           { O112 } { O113 }
27584           { O114 } { O115 }
27585           { O116 } { O117 }
27586           { O118 } { O119 }
27587           { O11A } { O11B }
27588           { O11C } { O11D }
27589           { O11E } { O11F }
27590           { O120 } { O121 }
27591           { O122 } { O123 }
27592           { O124 } { O125 }

```

```

27593      { 0128 } { 0129 }
27594      { 012A } { 012B }
27595      { 012C } { 012D }
27596      { 012E } { 012F }
27597      { 0132 } { 0133 }
27598      { 0134 } { 0135 }
27599      { 0136 } { 0137 }
27600      { 0139 } { 013A }
27601      { 013B } { 013C }
27602      { 013E } { 013F }
27603      { 0141 } { 0142 }
27604      { 0143 } { 0144 }
27605      { 0145 } { 0146 }
27606      { 0147 } { 0148 }
27607      { 014A } { 014B }
27608      { 014C } { 014D }
27609      { 014E } { 014F }
27610      { 0150 } { 0151 }
27611      { 0152 } { 0153 }
27612      { 0154 } { 0155 }
27613      { 0156 } { 0157 }
27614      { 0158 } { 0159 }
27615      { 015A } { 015B }
27616      { 015C } { 015D }
27617      { 015E } { 015F }
27618      { 0160 } { 0161 }
27619      { 0162 } { 0163 }
27620      { 0164 } { 0165 }
27621      { 0168 } { 0169 }
27622      { 016A } { 016B }
27623      { 016C } { 016D }
27624      { 016E } { 016F }
27625      { 0170 } { 0171 }
27626      { 0172 } { 0173 }
27627      { 0174 } { 0175 }
27628      { 0176 } { 0177 }
27629      { 0178 } { 00FF }
27630      { 0179 } { 017A }
27631      { 017B } { 017C }
27632      { 017D } { 017E }
27633      { 01CD } { 01CE }
27634      { 01CF } { 01DO }
27635      { 01D1 } { 01D2 }
27636      { 01D3 } { 01D4 }
27637      { 01E2 } { 01E3 }
27638      { 01E6 } { 01E7 }
27639      { 01E8 } { 01E9 }
27640      { 01EA } { 01EB }
27641      { 01F4 } { 01F5 }
27642      { 0218 } { 0219 }
27643      { 021A } { 021B }
27644      \q_recursion_tail ?
27645      \q_recursion_stop
27646      \cs_set_protected:Npn \__tl_tmp:w #1#2#3

```

```

27647 {
27648   \group_begin:
27649     \cs_set_protected:Npn \__tl_tmp:w ##1##2##3##4
27650     {
27651       \tl_const:cx
27652       {
27653         c__tl_ #3 _case_
27654         \char_generate:nn {##1} { 12 }
27655         \char_generate:nn {##2} { 12 }
27656         _tl
27657       }
27658       {#2}
27659     }
27660     \tl_set:Nx \l__tl_internal_a_tl
27661       { \char_codepoint_to_bytes:n { "#1" } }
27662       \exp_after:wn \__tl_tmp:w \l__tl_internal_a_tl
27663     \group_end:
27664   }
27665   \__tl_tmp:w { OODF } { SS } { upper }
27666   \__tl_tmp:w { OODF } { Ss } { mixed }
27667   \__tl_tmp:w { 0131 } { I } { upper }
27668 }
27669 \group_end:

```

The (fixed) look-up mappings for letter-like control sequences.

```

27670 \group_begin:
27671   \cs_set_protected:Npn \__tl_change_case_setup:NN #1#2
27672   {
27673     \quark_if_recursion_tail_stop:N #1
27674     \tl_const:cn { c__tl_change_case_lower_ \token_to_str:N #1 _tl }
27675       { #2 }
27676     \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N #2 _tl }
27677       { #1 }
27678     \__tl_change_case_setup:NN
27679   }
27680   \__tl_change_case_setup:NN
27681   \AA \aa
27682   \AE \ae
27683   \DH \dh
27684   \DJ \dj
27685   \IJ \ij
27686   \L \l
27687   \NG \ng
27688   \O \o
27689   \OE \oe
27690   \SS \ss
27691   \TH \th
27692   \q_recursion_tail ?
27693   \q_recursion_stop
27694   \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \i _tl } { I }
27695   \tl_const:cn { c__tl_change_case_upper_ \token_to_str:N \j _tl } { J }
27696 \group_end:

```

**\l\_tl\_case\_change\_accents\_tl** A list of accents to leave alone.

```

27697 \tl_new:N \l_tl_case_change_accents_tl
27698 \tl_set:Nn \l_tl_case_change_accents_tl
27699 { \" \' \. \^ \` \~ \c \H \k \r \t \u \v }

```

(End definition for `\l_tl_case_change_accents_tl`. This variable is documented on page 256.)

`\_tl_change_case_mixed_nl:Nnw` 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.

```

27700 \cs_new:Npn \_tl_change_case_mixed_nl:Nnw #1
27701 {
27702     \bool_lazy_or:nnTF
27703     { \int_compare_p:nNn { '#1 } = { 'i } }
27704     { \int_compare_p:nNn { '#1 } = { 'I } }
27705     {
27706         \_tl_change_case_output:nwn { I }
27707         \_tl_change_case_mixed_nl:Nw
27708     }
27709 }
27710 \cs_new:Npn \_tl_change_case_mixed_nl:Nw #1#2 \q_recursion_stop
27711 {
27712     \tl_if_head_is_N_type:nT {#2}
27713     { \_tl_change_case_mixed_nl>NNw }
27714     #1 #2 \q_recursion_stop
27715 }
27716 \cs_new:Npn \_tl_change_case_mixed_nl>NNw #1#2#3 \q_recursion_stop
27717 {
27718     \_tl_change_case_if_expandable:NTF #2
27719 {
27720     \exp_after:wN \_tl_change_case_mixed_nl:Nw \exp_after:wN #1 #2
27721     #3 \q_recursion_stop
27722 }
27723 {
27724     \bool_lazy_and:nnTF
27725     { ! ( \token_if_cs_p:N #2 ) }
27726     {
27727         \bool_lazy_or_p:nn
27728         { \int_compare_p:nNn { '#2 } = { 'j } }
27729         { \int_compare_p:nNn { '#2 } = { 'J } }
27730     }
27731     {
27732         \_tl_change_case_output:nwn { J }
27733         #1
27734     }
27735     { #1 #2 }
27736     #3 \q_recursion_stop
27737 }
27738 }

```

(End definition for `\_tl_change_case_mixed_nl:Nnw`, `\_tl_change_case_mixed_nl:Nw`, and `\_tl_change_case_mixed_nl>NNw`.)

`\l_tl_case_change_math_tl` The list of token pairs which are treated as math mode and so not case changed.

```

27739 \tl_new:N \l_tl_case_change_math_tl
27740 (*package)
27741 \tl_set:Nn \l_tl_case_change_math_tl

```

```

27742   { $ $ \\( \\) }
27743   </package>

```

(End definition for `\l_tl_case_change_math_tl`. This variable is documented on page 255.)

**\l\_tl\_case\_change\_exclude\_tl** The list of commands for which an argument is not case changed.

```

27744 \tl_new:N \l_tl_case_change_exclude_tl
27745 (*package)
27746 \tl_set:Nn \l_tl_case_change_exclude_tl
27747   { \cite \ensuremath \label \ref }
27748 </package>

```

(End definition for `\l_tl_case_change_exclude_tl`. This variable is documented on page 255.)

**\l\_tl\_mixed\_case\_ignore\_tl** Characters to skip over when finding the first letter in a word to be mixed cased.

```

27749 \tl_new:N \l_tl_mixed_case_ignore_tl
27750 \tl_set:Nx \l_tl_mixed_case_ignore_tl
27751   {
27752     ( % )
27753     [ % ]
27754     \cs_to_str:N \{ % \}
27755     '
27756     -
27757   }

```

(End definition for `\l_tl_mixed_case_ignore_tl`. This variable is documented on page 256.)

#### 44.14.3 Building a token list

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 <assignment> <next tl>
{<left>} <right>
```

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 `\cs_set_nopar:Npx` if the variable is local, and `\cs_gset_nopar:Npx` if it is global.

**\tl\_build\_begin:N**  
**\tl\_build\_gbegin:N**  
`\__tl_build_begin:NN`  
`\__tl_build_begin:NNN`

First construct the  $\langle next tl \rangle$ : using a prime here conflicts with the usual `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 `\cs_(g)set_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 `\__tl_build_begin:NNN` could use `\tl_(g)clear_new: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.

```

27758 \__kernel_patch:nnNpn { \__kernel_chk_var_local:N #1 } {}
27759 \cs_new_protected:Npn \tl_build_begin:N #1
27760   { \__tl_build_begin:NN \cs_set_nopar:Npx #1 }
27761 \__kernel_patch:nnNpn { \__kernel_chk_var_global:N #1 } {}
27762 \cs_new_protected:Npn \tl_build_gbegin:N #1

```

```

27763 { \__tl_build_begin:N \cs_gset_nopar:Npx #1 }
27764 \cs_new_protected:Npn \__tl_build_begin:NN #1#2
27765 { \exp_args:Nc \__tl_build_begin:NNN { \cs_to_str:N #2 ' } #2 #1 }
27766 \cs_new_protected:Npn \__tl_build_begin:NNN #1#2#3
27767 {
27768     #3 #1 { }
27769     #3 #2
27770     {
27771         \exp_not:n { \exp_end: \exp_end: \exp_end: \exp_end: }
27772         \exp_not:n { \__tl_build_last:NNn #3 #1 { } }
27773     }
27774 }
```

(End definition for `\tl_build_begin:N` and others. These functions are documented on page 258.)

`\tl_build_clear:N`  
`\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 `\tl_build_end:N`. This only affects memory usage.

```

27775 \cs_new_eq:NN \tl_build_clear:N \tl_build_begin:N
27776 \cs_new_eq:NN \tl_build_gclear:N \tl_build_gbegin:N
```

(End definition for `\tl_build_clear:N` and `\tl_build_gclear:N`. These functions are documented on page 258.)

`\tl_build_put_right:Nn`  
`\tl_build_put_right:Nx`  
`\tl_build_gput_right:Nn`  
`\tl_build_gput_right:Nx`  
`\__tl_build_last:NNn`  
`\__tl_build_put:nn`  
`\__tl_build_put:nw`

Similar to `\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, `\__tl_build_last:NNn` is expanded instead. It resets the definition of the `\tl var` by ending the `\exp_not:n` and the definition early. Then it makes sure the `\next tl` (its argument `#1`) is set-up and starts a new definition. Then `\__tl_build_put:nn` and `\__tl_build_put:nw` place the `\left` part of the original `\tl var` as appropriate for the definition of the `\next tl` (the `\right` 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 `\cs_(g)set_nopar:Npx` rather than `\tl_(g)set: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.

```

27777 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
27778 \cs_new_protected:Npn \tl_build_put_right:Nn #1#2
27779 {
2780     \cs_set_nopar:Npx #1
2781     { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
2782 }
2783 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
2784 \cs_new_protected:Npn \tl_build_put_right:Nx #1#2
2785 {
2786     \cs_set_nopar:Npx #1
2787     { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
2788 }
2789 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
2790 \cs_new_protected:Npn \tl_build_gput_right:Nn #1#2
2791 {
2792     \cs_gset_nopar:Npx #1
2793     { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 #2 } }
```

```

27794      }
27795  \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
27796  \cs_new_protected:Npn \tl_build_gput_right:Nx #1#2
27797  {
27798      \cs_gset_nopar:Npx #1
27799      { \exp_after:wN \exp_not:n \exp_after:wN { \exp:w #1 } #2 }
27800  }
27801 \cs_new_protected:Npn \__tl_build_last:NNn #1#2
27802  {
27803      \if_false: { { \fi:
27804          \exp_end: \exp_end: \exp_end: \exp_end: \exp_end:
27805          \__tl_build_last:NNn #1 #2 { }
27806      }
27807  }
27808  \if_meaning:w \c_empty_tl #2
27809  \__tl_build_begin:NN #1 #2
27810 \fi:
27811 #1 #2
27812  {
27813      \exp_after:wN \exp_not:n \exp_after:wN
27814      {
27815          \exp:w \if_false: } } \fi:
27816          \exp_after:wN \__tl_build_put:nn \exp_after:wN {#2}
27817  }
27818 \cs_new_protected:Npn \__tl_build_put:nn #1#2 { \__tl_build_put:nw {#2} #1 }
27819 \cs_new_protected:Npn \__tl_build_put:nw #1#2 \__tl_build_last:NNn #3#4#5
27820  { #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 259.)

**\tl\_build\_put\_left:Nn** See `\tl_build_put_right:Nn` for all the machinery. We could easily provide `\tl_build_put_left_right:Nnn`, by just add the `\tl_build_right:Nnn` material after the `{(left)}` in the x-expanding assignment.

```

27821 \__kernel_patch:nnNNpn { \__kernel_chk_var_local:N #1 } { }
27822 \cs_new_protected:Npn \tl_build_put_left:Nn #1
27823  { \__tl_build_put_left:NNn \cs_set_nopar:Npx #1 }
27824 \cs_generate_variant:Nn \tl_build_put_left:Nn { Nx }
27825 \__kernel_patch:nnNNpn { \__kernel_chk_var_global:N #1 } { }
27826 \cs_new_protected:Npn \tl_build_gput_left:Nn #1
27827  { \__tl_build_put_left:NNn \cs_gset_nopar:Npx #1 }
27828 \cs_generate_variant:Nn \tl_build_gput_left:Nn { Nx }
27829 \cs_new_protected:Npn \__tl_build_put_left:NNn #1#2#3
27830  {
27831      #1 #2
27832  {
27833      \exp_after:wN \exp_not:n \exp_after:wN
27834      {
27835          \exp:w \exp_after:wN \__tl_build_put:nn
27836          \exp_after:wN {#2} {#3}
27837      }
27838  }
27839  }

```

(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 259.)

**\tl\_build\_get:NN** 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.

```

27840 \cs_new_protected:Npn \tl_build_get:NN
27841   { \__tl_build_get:NNN \tl_set:Nx }
27842 \cs_new_protected:Npn \__tl_build_get:NNN #1#2#3
27843   { #1 #3 { \if_false: { \fi: \exp_after:wN \__tl_build_get:w #2 } } }
27844 \cs_new:Npn \__tl_build_get:w #1 \__tl_build_last:NNn #2#3#4
27845   {
27846     \exp_not:n {#4}
27847     \if_meaning:w \c_empty_tl #3
27848       \exp_after:wN \__tl_build_get_end:w
27849     \fi:
27850       \exp_after:wN \__tl_build_get:w #3
27851   }
27852 \cs_new:Npn \__tl_build_get_end:w #1#2#3
27853   { \exp_after:wN \exp_not:n \exp_after:wN { \if_false: } \fi: }
```

(End definition for  $\text{\tl\_build\_get:NN}$  and others. This function is documented on page 259.)

**\tl\_build\_end:N** Get the data then clear the  $\langle next\ tl \rangle$  recursively until finding an empty one. It is perhaps wasteful to repeatedly use  $\text{\cs\_to\_sr:N}$ . The local/global scope is checked by  $\text{\tl\_set:Nx}$  or  $\text{\tl\_gset:Nx}$ .

```

27854 \cs_new_protected:Npn \tl_build_end:N #1
27855   {
27856     \__tl_build_get:NNN \tl_set:Nx #1 #1
27857     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_clear:N
27858   }
27859 \cs_new_protected:Npn \tl_build_gend:N #1
27860   {
27861     \__tl_build_get:NNN \tl_gset:Nx #1 #1
27862     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } \tl_gclear:N
27863   }
27864 \cs_new_protected:Npn \__tl_build_end_loop:NN #1#2
27865   {
27866     \if_meaning:w \c_empty_tl #1
27867       \exp_after:wN \use_none:nnnnnn
27868     \fi:
27869     #2 #1
27870     \exp_args:Nc \__tl_build_end_loop:NN { \cs_to_str:N #1 ' } #2
27871   }
```

(End definition for  $\text{\tl\_build\_end:N}$ ,  $\text{\tl\_build\_gend:N}$ , and  $\text{\__tl\_build\_end\_loop:NN}$ . These functions are documented on page 259.)

#### 44.14.4 Other additions to l3tl

**\tl\_range\_braced:Nnn** For the braced version  $\text{\__tl\_range\_braced:w}$  sets up  $\text{\__tl\_range\_collect\_braced:w}$  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  $\text{\__tl\_range\_collect:nn}$  with a first argument of the form  $\{ \{ \langle collected \rangle \} \langle tokens \rangle \}$ , whose head is the collected

**\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**

tokens and whose tail is what remains of the original token list. This form makes it easier to move tokens to the *(collected)* tokens.

```

27872 \cs_new:Npn \tl_range_braced:Nnn { \exp_args:No \tl_range_braced:n nn }
27873 \cs_generate_variant:Nn \tl_range_braced:Nnn { c }
27874 \cs_new:Npn \tl_range_braced:nnn { \__tl_range:Nnnn \__tl_range_braced:w }
27875 \cs_new:Npn \tl_range_unbraced:Nnn
27876   { \exp_args:No \tl_range_unbraced:nnn }
27877 \cs_generate_variant:Nn \tl_range_unbraced:Nnn { c }
27878 \cs_new:Npn \tl_range_unbraced:nnn
27879   { \__tl_range:Nnnn \__tl_range_unbraced:w }
27880 \cs_new:Npn \__tl_range_braced:w #1 ; #2
27881   { \__tl_range_collect_braced:w #1 ; { } #2 }
27882 \cs_new:Npn \__tl_range_unbraced:w #1 ; #2
27883   { \__tl_range_collect_unbraced:w #1 ; { } #2 }
27884 \cs_new:Npn \__tl_range_collect_braced:w #1 ; #2#3
27885   {
27886     \if_int_compare:w #1 > 1 \exp_stop_f:
27887       \exp_after:wN \__tl_range_collect_braced:w
27888       \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
27889     \fi:
27890     { #2 {#3} }
27891   }
27892 \cs_new:Npn \__tl_range_collect_unbraced:w #1 ; #2#3
27893   {
27894     \if_int_compare:w #1 > 1 \exp_stop_f:
27895       \exp_after:wN \__tl_range_collect_unbraced:w
27896       \int_value:w \int_eval:n { #1 - 1 } \exp_after:wN ;
27897     \fi:
27898     { #2 #3 }
27899   }

```

(End definition for `\tl_range_braced:Nnn` and others. These functions are documented on page 258.)

## 44.15 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.

```

27900 \group_begin:
27901   \char_set_catcode_active:N *
27902   \char_set_lccode:nn { '*' } { '\ }
27903   \tex_lowercase:D { \tl_const:Nn \c_catcode_active_space_tl { * } }
27904 \group_end:

```

(End definition for `\c_catcode_active_space_tl`. This variable is documented on page 259.)

`\@=\_peek`

### `\_peek_N_type:TF`

`\_peek_execute_branches_N_type:`

`\__peek_N_type:w`

`\__peek_N_type_aux:nww`

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 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, `\use_none_delimit_by_q_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.

```

27906 \group_begin:
27907   \cs_set_protected:Npn \__peek_tmp:w #1 \q_stop
27908   {
27909     \cs_new_protected:Npn \__peek_execute_branches_N_type:
27910     {
27911       \if_int_odd:w
27912         \if_catcode:w \exp_not:N \l_peek_token { 0 \exp_stop_f: \fi:
27913         \if_catcode:w \exp_not:N \l_peek_token } 0 \exp_stop_f: \fi:
27914         \if_meaning:w \l_peek_token \c_space_token 0 \exp_stop_f: \fi:
27915         1 \exp_stop_f:
27916         \exp_after:wN \__peek_N_type:w
27917           \token_to_meaning:N \l_peek_token
27918           \q_mark \__peek_N_type_aux:nnw
27919           #1 \q_mark \use_none_delimit_by_q_stop:w
27920           \q_stop
27921         \exp_after:wN \__peek_true:w
27922       \else:
27923         \exp_after:wN \__peek_false:w
27924       \fi:
27925     }
27926     \cs_new_protected:Npn \__peek_N_type:w ##1 #1 ##2 \q_mark ##3
27927     { ##3 {##1} {##2} }
27928   }
27929   \exp_after:wN \__peek_tmp:w \tl_to_str:n { outer } \q_stop
27930 \group_end:
27931 \cs_new_protected:Npn \__peek_N_type_aux:nnw #1 #2 #3 \fi:
27932   {
27933     \fi:
27934     \tl_if_in:noTF {#1} { \tl_to_str:n {ma} }
27935     { \__peek_true:w }
27936     { \tl_if_empty:nTF {#2} { \__peek_true:w } { \__peek_false:w } }
27937   }
27938 \cs_new_protected:Npn \peek_N_type:TF
27939   {
27940     \__peek_token_generic:NNTF
27941       \__peek_execute_branches_N_type: \scan_stop:
27942   }
27943 \cs_new_protected:Npn \peek_N_type:T
27944   { \__peek_token_generic:NNT \__peek_execute_branches_N_type: \scan_stop: }
27945 \cs_new_protected:Npn \peek_N_type:F
27946   { \__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 260.)

```
\l_peek_collect_tl
27947 \tl_new:N \l_peek_collect_tl

(End definition for \l_peek_collect_tl.)
```

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.

```
27948 \cs_new_protected:Npn \peek_catcode_collect_inline:Nn
27949   { \__peek_collect>NNn \__peek_execute_branches_catcode: }
27950 \cs_new_protected:Npn \peekCharCode_collect_inline:Nn
27951   { \__peek_collect>NNn \__peek_execute_branches_charcode: }
27952 \cs_new_protected:Npn \peek_meaning_collect_inline:Nn
27953   { \__peek_collect>NNn \__peek_execute_branches_meaning: }
27954 \cs_new_protected:Npn \__peek_collect:NNn #1#2#3
27955   {
27956     \group_align_safe_begin:
27957     \cs_set_eq:NN \l_peek_search_token #2
27958     \tl_set:Nn \l_peek_search_tl {#2}
27959     \tl_clear:N \l_peek_collect_tl
27960     \cs_set:Npn \__peek_false:w
27961       { \exp_args:No \__peek_false_aux:n \l_peek_collect_tl }
27962     \cs_set:Npn \__peek_false_aux:n ##1
27963       {
27964         \group_align_safe_end:
27965         #3
27966       }
27967     \cs_set_eq:NN \__peek_true:w \__peek_collect_true:w
27968     \cs_set:Npn \__peek_true_aux:w { \peek_after:Nw #1 }
27969     \__peek_true_aux:w
27970   }
27971 \cs_new_protected:Npn \__peek_collect_true:w
27972   {
27973     \if_case:w
27974       \if_catcode:w \exp_not:N \l_peek_token { 1 \exp_stop_f: \fi:
27975       \if_catcode:w \exp_not:N \l_peek_token } 2 \exp_stop_f: \fi:
27976       \if_meaning:w \l_peek_token \c_space_token 3 \exp_stop_f: \fi:
27977       0 \exp_stop_f:
27978       \exp_after:wN \__peek_collect:N
27979       \or: \__peek_collect_remove:nw { \c_group_begin_token }
27980       \or: \__peek_collect_remove:nw { \c_group_end_token }
27981       \or: \__peek_collect_remove:nw { ~ }
27982       \fi:
27983   }
27984 \cs_new_protected:Npn \__peek_collect:N #1
27985   {
27986     \tl_put_right:Nn \l_peek_collect_tl {#1}
```

```

27987     \__peek_true_aux:w
27988 }
27989 \cs_new_protected:Npn \__peek_collect_remove:nw #1
27990 {
27991     \tl_put_right:Nn \l__peek_collect_tl {#1}
27992     \exp_after:wN \__peek_true_remove:w
27993 }

```

(End definition for `\peek_catcode_collect_inline:Nn` and others. These functions are documented on page 260.)

27994 ⟨/initex | package⟩

## 45 l3drivers Implementation

```

27995 (*initex | package)
27996 (@@=driver)

```

Whilst there is a reasonable amount of code overlap between drivers, it is much clearer to have the blocks more-or-less separated than run in together and DocStripped out in parts. As such, most of the following is set up on a per-driver basis, though there is some common code (again given in blocks not interspersed with other material).

All the file identifiers are up-front so that they come out in the right place in the files.

```

27997 (*package)
27998 \ProvidesExplFile
27999 (*dvipdfmx)
28000 {l3dvipdfmx.def}{2019-01-28}{}
28001 {L3 Experimental driver: dvipdfmx}
28002 (/dvipdfmx)
28003 (*dvips)
28004 {l3dvips.def}{2019-01-28}{}
28005 {L3 Experimental driver: dvips}
28006 (/dvips)
28007 (*dvisvgm)
28008 {l3dvisvgm.def}{2019-01-28}{}
28009 {L3 Experimental driver: dvisvgm}
28010 (/dvisvgm)
28011 (*pdfmode)
28012 {l3pdfmode.def}{2019-01-28}{}
28013 {L3 Experimental driver: PDF mode}
28014 (/pdfmode)
28015 (*xdvipdfmx)
28016 {l3xdvipdfmx.def}{2019-01-28}{}
28017 {L3 Experimental driver: xdvipdfmx}
28018 (/xdvipdfmx)
28019 (/package)

```

The order of the driver code here is such that we get somewhat logical outcomes in terms of code sharing whilst keeping things readable. (Trying to mix all of the code by concept is almost unmanageable.) The key parts which are shared are

- Color support is either `dvips`-like or `pdfmode`-like.
- `pdfmode` and `(x)dvipdfmx` share drawing routines.

- `xdvipdfmx` is largely the same as `dvipdfmx` so takes most of the same code.

`\__driver_literal:e` The one shared function for all drivers is access to the basic `\special` primitive: it has slightly odd expansion behaviour so a wrapper is provided.

```

28020 \cs_new_eq:NN \__driver_literal:e \tex_special:D
28021 \cs_new_protected:Npn \__driver_literal:n #1
28022   { \__driver_literal:e { \exp_not:n {#1} } }
28023 \cs_generate_variant:Nn \__driver_literal:n { x }
```

(End definition for `\__driver_literal:e` and `\__driver_literal:n`.)

## 45.1 Color support

Color support is split into two parts: a “general” concept and one directly linked to drawings (or rather the split between filling and stroking). General color is relatively easy to handle: we have a color stack available with all modern drivers, and can use that. Whilst (x)`dvipdfmx` does have its own approach to color specials, it is easier to use `dvips`-like ones for all cases except direct PDF output.

### 45.1.1 dvips-style

```
28024 (*dvisvgm | dvipdfmx | dvips | xdvipdfmx)
```

`\driver_color_pickup:N`  
`\__driver_color_pickup:w` Allow for L<sup>A</sup>T<sub>E</sub>X 2 <sub>$\varepsilon$</sub>  color. Here, the possible input values are limited: `dvips`-style colors can mainly be taken as-is with the exception spot ones (here we need a model and a tint).

```

28025 (*package)
28026 \cs_new_protected:Npn \driver_color_pickup:N #1 { }
28027 \AtBeginDocument
28028 {
28029   \@ifpackageloaded {color}
28030   {
28031     \cs_set_protected:Npn \driver_color_pickup:N #1
28032       {
28033         \exp_args:NV \tl_if_head_is_space:nTF \current@color
28034           {
28035             \tl_set:Nx #1
28036             {
28037               spot ~
28038               \exp_after:wN \use:n \current@color \c_space_tl 1
28039             }
28040           }
28041           {
28042             \exp_last_unbraced:Nx \__driver_color_pickup:w
28043               { \current@color } \q_stop #1
28044           }
28045       }
28046     \cs_new_protected:Npn \__driver_color_pickup:w #1 ~ #2 \q_stop #3
28047       { \tl_set:Nn #3 { #1 ~ #2 } }
28048   }
28049   { }
28050 }
28051 
```

(End definition for `\driver_color_pickup:N` and `\__driver_color_pickup:w`. This function is documented on page 262.)

```

\driver_color_cmyk:nnnn
  \driver_color_gray:n
  \driver_color_rgb:nnn
  \driver_color_spot:nn
__driver_color_select:n
\_driver_color_select:x
\driver_color_reset:
  \_driver_color_select:n
  \_driver_color_select:x
    {
      \_driver_color_select:x
        {
          cmyk~
            \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
            \fp_eval:n {#3} ~ \fp_eval:n {#4}
        }
    }
  \cs_new_protected:Npn \driver_color_gray:n #1
    {
      \_driver_color_select:x { gray~ \fp_eval:n {#1} } }
  \cs_new_protected:Npn \driver_color_rgb:nnn #1#2#3
    {
      \_driver_color_select:x
        {
          rgb~ \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} }
    }
  \cs_new_protected:Npn \driver_color_spot:nn #1#2
    {
      \_driver_color_select:n { #1 } }
  \cs_new_protected:Npn \_driver_color_select:n #1
    {
      \_driver_literal:n { color~push~ #1 }
      (*dvips)
        \_driver_literal_postscript:n { /13fc~{ }~def }
      
```

(End definition for \driver\_color\_cmyk:nnnn and others. These functions are documented on page 262.)

```

28081  </dvisvgm | dvipdfmx | dvips | xdvipdfmx>

```

#### 45.1.2 pdfmode

```

28082  (*pdfmode)

```

\driver\_color\_pickup:N  
 \\_driver\_color\_pickup:w

The current color in driver-dependent format: pick up the package-mode data if available. We end up converting back and forward in this route as we store our color data in dvips format. The \current@color needs to be x-expanded before \\_driver\_color\_pickup:w breaks it apart, because for instance xcolor sets it to be instructions to generate a colour

```

28083  (*package)
28084  \cs_new_protected:Npn \driver_color_pickup:N #1 {
28085  \AtBeginDocument
28086  {
28087  \c_ifpackageloaded { color }
28088  {
28089  \cs_set_protected:Npn \driver_color_pickup:N #1
28090  {
28091  \exp_last_unbraced:Nx \_driver_color_pickup:w

```

```

28092           { \current@color } ~ 0 ~ 0 ~ 0 \q_stop #1
28093       }
28094   \cs_new_protected:Npn \__driver_color_pickup:w
28095     #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 \q_stop #7
28096   {
28097     \str_if_eq:nnTF {#2} { g }
28098       { \tl_set:Nn #7 { gray ~ #1 } }
28099       {
28100         \str_if_eq:nnTF {#4} { rg }
28101           { \tl_set:Nn #7 { rgb ~ #1 ~ #2 ~ #3 } }
28102           {
28103             \str_if_eq:nnTF {#5} { k }
28104               { \tl_set:Nn #7 { cmyk ~ #1 ~ #2 ~ #3 ~ #4 } }
28105               {
28106                 \str_if_eq:nnTF {#2} { cs }
28107                   {
28108                     \tl_set:Nx #7 { spot ~ \use_none:n #1 ~ #5 }
28109                   }
28110                   {
28111                     \tl_set:Nn #7 { gray ~ 0 }
28112                   }
28113               }
28114           }
28115       }
28116   }
28117 }
28118 {
28119 }
28120 </package>

```

(End definition for `\driver_color_pickup:N` and `\__driver_color_pickup:w`. This function is documented on page 262.)

`\l__driver_color_stack_int` pdfTeX and LuaTeX have multiple stacks available, and to track which one is in use a variable is required.

```
28121 \int_new:N \l__driver_color_stack_int
```

(End definition for `\l__driver_color_stack_int`.)

`\driver_color_cmyk:nnnn` Simply dump the data, but allowing for LuaTeX.

```

28122 \cs_new_protected:Npn \driver_color_cmyk:nnnn #1#2#3#4
28123   {
28124     \use:x
28125       {
28126         \__driver_color_cmyk:nnnn
28127           { \fp_eval:n {#1} }
28128           { \fp_eval:n {#2} }
28129           { \fp_eval:n {#3} }
28130           { \fp_eval:n {#4} }
28131       }
28132     }
28133 \cs_new_protected:Npn \__driver_color_cmyk:nnnn #1#2#3#4
28134   {
28135     \__driver_color_select:n
28136       { #1 ~ #2 ~ #3 ~ #4 ~ k ~ #1 ~ #2 ~ #3 ~ #4 ~ K }

```

```

28137      }
28138  \cs_new_protected:Npn \driver_color_gray:n #1
28139    { \exp_args:Nx \__driver_color_gray:n { \fp_eval:n {#1} } }
28140  \cs_new_protected:Npn \__driver_color_gray:n #1
28141    { \__driver_color_select:n { #1 ~ g ~ #1 ~ G } }
28142  \cs_new_protected:Npn \driver_color_rgb:nnn #1#2#3
28143    {
28144      \use:x
28145      {
28146        \__driver_color_rgb:nnn
28147          { \fp_eval:n {#1} }
28148          { \fp_eval:n {#2} }
28149          { \fp_eval:n {#3} }
28150      }
28151    }
28152  \cs_new_protected:Npn \__driver_color_rgb:nnn #1#2#3
28153    { \__driver_color_select:n { #1 ~ #2 ~ #3 ~ rg ~ #1 ~ #2 ~ #3 ~ RG } }
28154  \cs_new_protected:Npn \driver_color_spot:nn #1#2
28155    { \__driver_color_select:n { /#1 ~ cs ~ /#1 ~ CS ~ #2 ~ sc ~ #2 ~ SC } }
28156  \cs_new_protected:Npx \__driver_color_select:n #1
28157    {
28158      \cs_if_exist:NTF \tex_pdfextension:D
28159        { \tex_pdfextension:D colorstack }
28160        { \tex_pdfcolorstack:D }
28161        \exp_not:N \l__driver_color_stack_int push {#1}
28162        \group_insert_after:N \exp_not:N \__driver_color_reset:
28163    }
28164  \cs_generate_variant:Nn \__driver_color_select:n { x }
28165  \cs_new_protected:Npx \__driver_color_reset:
28166    {
28167      \cs_if_exist:NTF \tex_pdfextension:D
28168        { \tex_pdfextension:D colorstack }
28169        { \tex_pdfcolorstack:D }
28170        \exp_not:N \l__driver_color_stack_int pop \scan_stop:
28171    }

```

(End definition for `\driver_color_cmyk:nnnn` and others. These functions are documented on page 262.)

28172 `</pdfmode>`

## 45.2 dvips driver

28173 `<*dvips>`

### 45.2.1 Basics

`\__driver_literal_postscript:n` Literal PostScript can be included using a few low-level formats. Here, we use the form with no positioning: this is overall more convenient as a wrapper. Note that this does require that where position is important, an appropriate wrapper is included.

```

28174  \cs_new_protected:Npn \__driver_literal_postscript:n #1
28175    { \__driver_literal:n { ps:: #1 } }
28176  \cs_generate_variant:Nn \__driver_literal_postscript:n { x }

```

(End definition for `\__driver_literal_postscript:n`.)

```
\_driver_align_currentpoint_begin:  
\_driver_align_currentpoint_end:
```

In dvips there is no build-in saving of the current position, and so some additional PostScript is required to set up the transformation matrix and also to restore it afterwards. Notice the use of the stack to save the current position “up front” and to move back to it at the end of the process. Notice that the [begin]/[end] pair here mean that we can use a run of PostScript statements in separate lines: not *required* but does make the code and output more clear.

```
28177 \cs_new_protected:Npn \_driver_align_currentpoint_begin:  
28178 {  
28179     \_driver_literal:n { ps::[begin] }  
28180     \_driver_literal_postscript:n { currentpoint }  
28181     \_driver_literal_postscript:n { currentpoint~translate }  
28182 }  
28183 \cs_new_protected:Npn \_driver_align_currentpoint_end:  
28184 {  
28185     \_driver_literal_postscript:n { neg~exch~neg~exch~translate }  
28186     \_driver_literal:n { ps::[end] }  
28187 }
```

(End definition for `\_driver_align_currentpoint_begin:` and `\_driver_align_currentpoint_end:..`)

```
\_driver_scope_begin:  
\_driver_scope_end:
```

Saving/restoring scope for general operations needs to be done with dvips positioning (try without to see this!). Thus we need the `ps:` version of the special here. As only the graphics state is ever altered within this pairing, we use the lower-cost `g`-versions.

```
28188 \cs_new_protected:Npn \_driver_scope_begin:  
28189 { \_driver_literal:n { ps:gsave } }  
28190 \cs_new_protected:Npn \_driver_scope_end:  
28191 { \_driver_literal:n { ps:grestore } }
```

(End definition for `\_driver_scope_begin:` and `\_driver_scope_end:..`)

#### 45.2.2 Box operations

`\driver_box_use_clip:N`

The dvips driver scales all absolute dimensions based on the output resolution selected and any TeX magnification. Thus for any operation involving absolute lengths there is a correction to make. See `normalscale` from `special.pro` for the variables, noting that here everything is saved on the stack rather than as a separate variable. Once all of that is done, the actual clipping is trivial.

```
28192 \cs_new_protected:Npn \driver_box_use_clip:N #1  
28193 {  
28194     \_driver_scope_begin:  
28195     \_driver_align_currentpoint_begin:  
28196     \_driver_literal_postscript:n { matrix~currentmatrix }  
28197     \_driver_literal_postscript:n  
28198     { Resolution~72~div~Vresolution~72~div~scale }  
28199     \_driver_literal_postscript:n { DVImag~dup~scale }  
28200     \_driver_literal_postscript:x  
28201     {  
28202         0 ~  
28203         \dim_to_decimal_in_bp:n { \box_dp:N #1 } ~  
28204         \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~  
28205         \dim_to_decimal_in_bp:n { -\box_ht:N #1 - \box_dp:N #1 } ~  
28206         rectclip  
28207 }
```

```

28208     \__driver_literal_postscript:n { setmatrix }
28209     \__driver_align_currentpoint_end:
28210     \hbox_overlap_right:n { \box_use:N #1 }
28211     \__driver_scope_end:
28212     \skip_horizontal:n { \box_wd:N #1 }
28213 }

```

(End definition for `\driver_box_use_clip:N`. This function is documented on page 261.)

**\driver\_box\_use\_rotate:Nn**  
`\__driver_box_use_rotate:Nn`

Rotating using dvips does not require that the box dimensions are altered and has a very convenient built-in operation. Zero rotation must be written as 0 not -0 so there is a quick test.

```

28214 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
28215   { \exp_args:NNf \__driver_box_use_rotate:Nn #1 { \fp_eval:n {#2} } }
28216 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
28217   {
28218     \__driver_scope_begin:
28219     \__driver_align_currentpoint_begin:
28220     \__driver_literal_postscript:x
28221     {
28222       \fp_compare:nNnTF {#2} = \c_zero_fp
28223         { 0 }
28224         { \fp_eval:n { round ( -(#2) , 5 ) } } ~
28225         rotate
28226     }
28227     \__driver_align_currentpoint_end:
28228     \box_use:N #1
28229     \__driver_scope_end:
28230   }

```

(End definition for `\driver_box_use_rotate:Nn` and `\__driver_box_use_rotate:Nn`. This function is documented on page 262.)

**\driver\_box\_use\_scale:Nnn**

The dvips driver once again has a dedicated operation we can use here.

```

28231 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
28232   {
28233     \__driver_scope_begin:
28234     \__driver_align_currentpoint_begin:
28235     \__driver_literal_postscript:x
28236     {
28237       \fp_eval:n { round ( #2 , 5 ) } ~
28238       \fp_eval:n { round ( #3 , 5 ) } ~
28239       scale
28240     }
28241     \__driver_align_currentpoint_end:
28242     \hbox_overlap_right:n { \box_use:N #1 }
28243     \__driver_scope_end:
28244   }

```

(End definition for `\driver_box_use_scale:Nnn`. This function is documented on page 262.)

### 45.2.3 Images

\\_\\_driver\\_image\\_getbb\\_eps:n Simply use the generic function.  
28245 \cs\_new\_eq:NN \\_\\_driver\_image\_getbb\_eps:n \image\_read\_bb:n  
(End definition for \\_\\_driver\_image\_getbb\_eps:n.)

\\_\\_driver\\_image\\_include\\_eps:n The special syntax is relatively clear here: remember we need PostScript sizes here.  
28246 \cs\_new\_protected:Npn \\_\\_driver\_image\_include\_eps:n #1  
28247 { \\_\\_driver\_literal:n { PSfile = #1 } }  
(End definition for \\_\\_driver\_image\_include\_eps:n.)

### 45.2.4 Drawing

\\_\\_driver\\_draw\\_literal:n The same as literal PostScript: same arguments about positioning apply here.  
\\_\\_driver\\_draw\\_literal:x  
28248 \cs\_new\_eq:NN \\_\\_driver\_draw\_literal:n \\_\\_driver\_literal\_postscript:n  
28249 \cs\_generate\_variant:Nn \\_\\_driver\_draw\_literal:n { x }  
(End definition for \\_\\_driver\_draw\_literal:n.)

\driver\_draw\_begin: \driver\_draw\_end: The ps::[begin] special here deals with positioning but allows us to continue on to a matching ps::[end]: contrast with ps:, which positions but where we can't split material between separate calls. The @beginspecial/@endspecial pair are from special.pro and correct the scale and y-axis direction. The definition of /13fc deals with fill color in paths. In contrast to pgf, we don't save the current point: discussion with Tom Rokici suggested a better way to handle the necessary translations (see \driver\_draw\_box\_use:Nnnnn). (Note that @beginspecial/@endspecial forms a driver scope.) The [begin]/[end] lines are handled differently from the rest as they are conceptually different: not really drawing literals but instructions to dvips itself.

```
28250 \cs_new_protected:Npn \driver_draw_begin:  
28251 {  
28252     \_\_driver_literal:n { ps::[begin] }  
28253     \_\_driver_draw_literal:n { @beginspecial }  
28254     \_\_driver_draw_literal:n { /13fc-{ }~def }  
28255 }  
28256 \cs_new_protected:Npn \driver_draw_end:  
28257 {  
28258     \_\_driver_draw_literal:n { @endspecial }  
28259     \_\_driver_literal:n { ps::[end] }  
28260 }
```

(End definition for \driver\_draw\_begin: and \driver\_draw\_end:. These functions are documented on page 263.)

\driver\_draw\_scope\_begin: \driver\_draw\_scope\_end: Scope here may need to contain saved definitions, so the entire memory rather than just the graphic state has to be sent to the stack.

```
28261 \cs_new_protected:Npn \driver_draw_scope_begin:  
28262 { \_\_driver_draw_literal:n { save } }  
28263 \cs_new_protected:Npn \driver_draw_scope_end:  
28264 { \_\_driver_draw_literal:n { restore } }
```

(End definition for \driver\_draw\_scope\_begin: and \driver\_draw\_scope\_end:. These functions are documented on page 263.)

```
\driver_draw_moveto:nn
\driver_draw_lineto:nn
\driver_draw_rectangle:nnnn
\driver_draw_curveto:nnnnnn
```

Path creation operations mainly resolve directly to PostScript primitive steps, with only the need to convert to bp. Notice that x-type expansion is included here to ensure that any variable values are forced to literals before any possible caching. There is no native rectangular path command (without also clipping, filling or stroking), so that task is done using a small amount of PostScript.

```
28265 \cs_new_protected:Npn \driver_draw_moveto:nn #1#2
28266 {
28267     \__driver_draw_literal:x
28268     {
28269         \dim_to_decimal_in_bp:n {#1} ~
28270         \dim_to_decimal_in_bp:n {#2} ~ moveto
28271     }
28272 }
28273 \cs_new_protected:Npn \driver_draw_lineto:nn #1#2
28274 {
28275     \__driver_draw_literal:x
28276     {
28277         \dim_to_decimal_in_bp:n {#1} ~
28278         \dim_to_decimal_in_bp:n {#2} ~ lineto
28279     }
28280 }
28281 \cs_new_protected:Npn \driver_draw_rectangle:nnnn #1#2#3#4
28282 {
28283     \__driver_draw_literal:x
28284     {
28285         \dim_to_decimal_in_bp:n {#4} ~ \dim_to_decimal_in_bp:n {#3} ~
28286         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
28287         moveto~dup~0~rlineto~exch~0~exch~rlineto~neg~0~rlineto~closepath
28288     }
28289 }
28290 \cs_new_protected:Npn \driver_draw_curveto:nnnnnn #1#2#3#4#5#6
28291 {
28292     \__driver_draw_literal:x
28293     {
28294         \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
28295         \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
28296         \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
28297         curveto
28298     }
28299 }
```

(End definition for `\driver_draw_moveto:nn` and others. These functions are documented on page 263.)

```
\driver_draw_evenodd_rule:
\driver_draw_nonzero_rule:
```

```
\g__driver_draw_eor_bool
```

The even-odd rule here can be implemented as a simply switch.

```
28300 \cs_new_protected:Npn \driver_draw_evenodd_rule:
28301     { \bool_gset_true:N \g__driver_draw_eor_bool }
28302 \cs_new_protected:Npn \driver_draw_nonzero_rule:
28303     { \bool_gset_false:N \g__driver_draw_eor_bool }
28304 \bool_new:N \g__driver_draw_eor_bool
```

(End definition for `\driver_draw_evenodd_rule:`, `\driver_draw_nonzero_rule:`, and `\g__driver_draw_eor_bool`. These functions are documented on page 264.)

`\driver_draw_closepath:` Unlike PDF, PostScript doesn't track separate colors for strokes and other elements. It is also desirable to have the `clip` keyword after a stroke or fill. To achieve those outcomes,

```
\driver_draw_closestroke:
\driver_draw_fill:
\driver_draw_fillstroke:
\driver_draw_clip:
\driver_draw_discardpath:
\g__driver_draw_clip_bool
```

there is some work to do. For color, the stroke color is simple but the fill one has to be inserted by hand. For clipping, the required ordering is achieved using a TeX switch. All of the operations end with a new path instruction as they do not terminate (again in contrast to PDF).

```

28305 \cs_new_protected:Npn \driver_draw_closepath:
28306   { \__driver_draw_literal:n { closepath } }
28307 \cs_new_protected:Npn \driver_draw_stroke:
28308   {
28309     \__driver_draw_literal:n { stroke }
28310     \bool_if:NT \g__driver_draw_clip_bool
28311       {
28312         \__driver_draw_literal:x
28313           {
28314             \bool_if:NT \g__driver_draw_eor_bool { eo }
28315             clip
28316           }
28317         }
28318       \__driver_draw_literal:n { newpath }
28319       \bool_gset_false:N \g__driver_draw_clip_bool
28320     }
28321 \cs_new_protected:Npn \driver_draw_closestroke:
28322   {
28323     \driver_draw_closepath:
28324     \driver_draw_stroke:
28325   }
28326 \cs_new_protected:Npn \driver_draw_fill:
28327   {
28328     \__driver_draw_literal:n { gsave }
28329     \__driver_draw_literal:n { 13fc }
28330     \__driver_draw_literal:x
28331       {
28332         \bool_if:NT \g__driver_draw_eor_bool { eo }
28333         fill
28334       }
28335     \__driver_draw_literal:n { grestore }
28336     \bool_if:NT \g__driver_draw_clip_bool
28337       {
28338         \__driver_draw_literal:x
28339           {
28340             \bool_if:NT \g__driver_draw_eor_bool { eo }
28341             clip
28342           }
28343         }
28344       \__driver_draw_literal:n { newpath }
28345       \bool_gset_false:N \g__driver_draw_clip_bool
28346     }
28347 \cs_new_protected:Npn \driver_draw_fillstroke:
28348   {
28349     \__driver_draw_literal:n { gsave }
28350     \__driver_draw_literal:n { 13fc }
28351     \__driver_draw_literal:x
28352       {
28353         \bool_if:NT \g__driver_draw_eor_bool { eo }
28354         fill

```

```

28355 }
28356 \__driver_draw_literal:n { grestore }
28357 \__driver_draw_literal:n { stroke }
28358 \bool_if:NT \g__driver_draw_clip_bool
28359 {
28360     \__driver_draw_literal:x
28361     {
28362         \bool_if:NT \g__driver_draw_eor_bool { eo }
28363         clip
28364     }
28365 }
28366 \__driver_draw_literal:n { newpath }
28367 \bool_gset_false:N \g__driver_draw_clip_bool
28368 }
28369 \cs_new_protected:Npn \driver_draw_clip:
28370     { \bool_gset_true:N \g__driver_draw_clip_bool }
28371 \bool_new:N \g__driver_draw_clip_bool
28372 \cs_new_protected:Npn \driver_draw_discardpath:
28373 {
28374     \bool_if:NT \g__driver_draw_clip_bool
28375     {
28376         \__driver_draw_literal:x
28377         {
28378             \bool_if:NT \g__driver_draw_eor_bool { eo }
28379             clip
28380         }
28381     }
28382     \__driver_draw_literal:n { newpath }
28383     \bool_gset_false:N \g__driver_draw_clip_bool
28384 }

```

(End definition for `\driver_draw_closepath:` and others. These functions are documented on page 263.)

`\driver_draw_dash_pattern:nn` Converting paths to output is again a case of mapping directly to PostScript operations.

```

\__driver_draw_dash:n
\driver_draw_linewidth:n
\driver_draw_miterlimit:n
\driver_draw_cap_but:
\driver_draw_cap_round:
\driver_draw_cap_rectangle:
\driver_draw_join_miter:
\driver_draw_join_round:
\driver_draw_join_bevel:
28385 \cs_new_protected:Npn \driver_draw_dash_pattern:nn #1#2
28386 {
28387     \__driver_draw_literal:x
28388     {
28389         [
28390             \exp_args:Nf \use:n
28391                 { \clist_map_function:nN {#1} \__driver_draw_dash:n }
28392         ] ~
28393         \dim_to_decimal_in_bp:n {#2} ~ setdash
28394     }
28395 }
28396 \cs_new:Npn \__driver_draw_dash:n #1
28397     { ~ \dim_to_decimal_in_bp:n {#1} }
28398 \cs_new_protected:Npn \driver_draw_linewidth:n #1
28399 {
28400     \__driver_draw_literal:x
28401     { \dim_to_decimal_in_bp:n {#1} ~ setlinewidth }
28402 }
28403 \cs_new_protected:Npn \driver_draw_miterlimit:n #1

```

```

28404 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ setmiterlimit } }
28405 \cs_new_protected:Npn \driver_draw_cap_butt:
28406 { \__driver_draw_literal:n { 0 ~ setlinecap } }
28407 \cs_new_protected:Npn \driver_draw_cap_round:
28408 { \__driver_draw_literal:n { 1 ~ setlinecap } }
28409 \cs_new_protected:Npn \driver_draw_cap_rectangle:
28410 { \__driver_draw_literal:n { 2 ~ setlinecap } }
28411 \cs_new_protected:Npn \driver_draw_join_miter:
28412 { \__driver_draw_literal:n { 0 ~ setlinejoin } }
28413 \cs_new_protected:Npn \driver_draw_join_round:
28414 { \__driver_draw_literal:n { 1 ~ setlinejoin } }
28415 \cs_new_protected:Npn \driver_draw_join_bevel:
28416 { \__driver_draw_literal:n { 2 ~ setlinejoin } }

```

(End definition for `\driver_draw_dash_pattern:nn` and others. These functions are documented on page 265.)

For dvips, we can use the standard color stack to deal with stroke color, but for fills have to switch to raw PostScript. This is thus not handled by the stack, but the context is very restricted. See also how fills are implemented.

```

28417 \cs_new_protected:Npn \driver_draw_color_fill_cmyk:nnnn #1#2#3#4
28418 {
28419   \__driver_draw_color_fill:x
28420   {
28421     \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
28422     \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
28423     setcmykcolor
28424   }
28425 }
28426 \cs_new_protected:Npn \driver_draw_color_stroke_cmyk:nnnn #1#2#3#4
28427 {
28428   \__driver_draw_color_stroke:x
28429   {
28430     cmyk ~
28431     \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
28432     \fp_eval:n {#3} ~ \fp_eval:n {#4}
28433   }
28434 }
28435 \cs_new_protected:Npn \driver_draw_color_fill_gray:n #1
28436 { \__driver_draw_color_fill:x { \fp_eval:n {#1} ~ setgray } }
28437 \cs_new_protected:Npn \driver_draw_color_stroke_gray:n #1
28438 { \__driver_draw_color_stroke:x { gray ~ \fp_eval:n {#1} } }
28439 \cs_new_protected:Npn \driver_draw_color_fill_rgb:nnn #1#2#3
28440 {
28441   \__driver_draw_color_fill:x
28442   { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ setrgbcolor }
28443 }
28444 \cs_new_protected:Npn \driver_draw_color_stroke_rgb:nnn #1#2#3
28445 {
28446   \__driver_draw_color_stroke:x
28447   { rgb ~ \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} }
28448 }
28449 \cs_new_protected:Npn \__driver_draw_color_fill:n #1
28450 { \__driver_draw_literal:n { /13fc ~ { #1 } ~ def } }

```

```

28451 \cs_generate_variant:Nn \__driver_draw_color_fill:n { x }
28452 \cs_new_protected:Npn \__driver_draw_color_stroke:n #1
28453 {
28454     \__driver_literal:n { color~push~#1 }
28455     \group_insert_after:N \__driver_color_reset:
28456 }
28457 \cs_generate_variant:Nn \__driver_draw_color_stroke:n { x }

```

(End definition for `\driver_draw_color_fill_cmyk:nnnn` and others. These functions are documented on page 265.)

**\driver\_draw\_cm:nnnn** In dvips, keeping the transformations in line with the engine is unfortunately not possible for scaling and rotations: even if we decompose the matrix into those operations, there is still no driver tracking (*cf.* (x)dvipdfmx). Thus we take the shortest path available and simply dump the matrix as given.

```

28458 \cs_new_protected:Npn \driver_draw_cm:nnnn #1#2#3#4
28459 {
28460     \__driver_draw_literal:n
28461     {
28462         [
28463             \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
28464             \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
28465             0 ~ 0
28466         ] ~
28467         concat
28468     }
28469 }

```

(End definition for `\driver_draw_cm:nnnn`. This function is documented on page 266.)

**\driver\_draw\_box\_use:Nnnnn** Inside a picture `@beginspecial/@endspecial` are active, which is normally a good thing but means that the position and scaling would be off if the box was inserted directly. To deal with that, there are a number of possible approaches. The implementation here was suggested by Tom Rokici (author of dvips). We end the current special placement, then set the current point with a literal `[begin]`. As for general literals, we then use the stack to store the current point and move to it. To insert the required transformation, we have to flip the *y*-axis, once before and once after it. Then we get back to the TeX reference point to insert our content. The clean up has to happen in the right places, hence the `[begin]/[end]` pair around `restore`. Finally, we can return to “normal” drawing mode. Notice that the set up here is very similar to that in `\__driver_align_currentpoint-...`, but the ordering of saving and restoring is different (intermixed).

```

28470 \cs_new_protected:Npn \driver_draw_box_use:Nnnnn #1#2#3#4#5
28471 {
28472     \__driver_draw_literal:n { @endspecial }
28473     \__driver_draw_literal:n { [end] }
28474     \__driver_draw_literal:n { [begin] }
28475     \__driver_draw_literal:n { save }
28476     \__driver_draw_literal:n { currentpoint }
28477     \__driver_draw_literal:n { currentpoint~translate }
28478     \driver_draw_cm:nnnn { 1 } { 0 } { 0 } { -1 }
28479     \driver_draw_cm:nnnn {#2} {#3} {#4} {#5}
28480     \driver_draw_cm:nnnn { 1 } { 0 } { 0 } { -1 }
28481     \__driver_draw_literal:n { neg~exch~neg~exch~translate }

```

```

28482     \__driver_draw_literal:n { [end] }
28483     \hbox_overlap_right:n { \box_use:N #1 }
28484     \__driver_draw_literal:n { [begin] }
28485     \__driver_draw_literal:n { restore }
28486     \__driver_draw_literal:n { [end] }
28487     \__driver_draw_literal:n { [begin] }
28488     \__driver_draw_literal:n { @beginspecial }
28489 }

```

(End definition for `\driver_draw_box_use:Nnnnn`. This function is documented on page 266.)

#### 45.2.5 PDF Features

`\g_driver_pdf_object_int` For tracking objects to allow finalisation.

```

\g_driver_pdf_object_prop
28490 \int_new:N \g_driver_pdf_object_int
28491 \prop_new:N \g_driver_pdf_object_prop

```

(End definition for `\g_driver_pdf_object_int` and `\g_driver_pdf_object_prop`.)

`\driver_pdf_object_new:nn` Tracking objects is similar to dvipdfmx.

```

\driver_pdf_object_ref:n
28492 \cs_new_protected:Npn \driver_pdf_object_new:nn #1#2
28493 {
28494     \int_gincr:N \g_driver_pdf_object_int
28495     \int_const:cn
28496         { g_driver_pdf_object_ \tl_to_str:n {#1} _int }
28497         { \g_driver_pdf_object_int }
28498     \prop_gput:Nnn \g_driver_pdf_object_prop {#1} {#2}
28499 }
28500 \cs_new:Npn \driver_pdf_object_ref:n #1
28501 { { 13obj \int_use:c { g_driver_pdf_object_ \tl_to_str:n {#1} _int } } }

```

(End definition for `\driver_pdf_object_new:nn` and `\driver_pdf_object_ref:n`. These functions are documented on page 266.)

`\driver_pdf_object_write:nn` This is where we choose the actual type: some work to get things right.

```

\__driver_pdf_object_write_array:nn
\__driver_pdf_object_write_dict:nn
\__driver_pdf_object_write_stream:nn
\__driver_pdf_object_write_stream:nnn
28502 \cs_new_protected:Npn \driver_pdf_object_write:nn #1#2
28503 {
28504     \__driver_literal_postscript:x
28505     {
28506         mark ~ /objdef ~ \driver_pdf_object_ref:n {#1} ~
28507         /type
28508         \str_case_e:nn
28509             { \prop_item:Nn \g_driver_pdf_object_prop {#1} }
28510             {
28511                 { array } { /array }
28512                 { dict } { /dict }
28513                 { fstream } { /stream }
28514                 { stream } { /stream }
28515             }
28516             /OBJ ~ pdfmark
28517         }
28518         \use:c
28519             { \__driver_pdf_object_write_ \prop_item:Nn \g_driver_pdf_object_prop {#1} :nn }
28520             {#1} {#2}
28521     }

```

```

28522 \cs_new_protected:Npn \__driver_pdf_object_write_array:nn #1#2
28523 {
28524   \__driver_literal_postscript:x
28525   {
28526     mark ~ \driver_pdf_object_ref:n {#1} ~
28527     [ ~ \exp_not:n {#2} ~ ] ~ /PUTINTERVAL ~ pdfmark
28528   }
28529 }
28530 \cs_new_protected:Npn \__driver_pdf_object_write_dict:nn #1#2
28531 {
28532   \__driver_literal_postscript:x
28533   {
28534     mark ~ \driver_pdf_object_ref:n {#1} ~
28535       << ~ \exp_not:n {#2} ~ >> ~ /PUT ~ pdfmark
28536   }
28537 }
28538 \cs_new_protected:Npn \__driver_pdf_object_write_stream:nn #1#2
28539 {
28540   \exp_args:Nx
28541   \__driver_pdf_object_write_stream:nnn
28542   { \driver_pdf_object_ref:n {#1} }
28543   #2
28544 }
28545 \cs_new_protected:Npn \__driver_pdf_object_write_stream:nnn #1#2#3
28546 {
28547   \__driver_literal_postscript:n
28548   {
28549     [nobreak] ~
28550     mark ~ #1 ~ ( #3 ) ~ /PUT ~ pdfmark ~
28551     mark ~ #1 ~ << #2 >> ~ /PUT ~ pdfmark
28552   }
28553 }

```

(End definition for `\driver_pdf_object_write:nn` and others. This function is documented on page 267.)

`\driver_pdf_compresslevel:n` These are all no-ops.

`\driver_pdf_objects_enable:`  
`\driver_pdf_objects_disable:`

```

28554 \cs_new_protected:Npn \driver_pdf_compresslevel:n #1 { }
28555 \cs_new_protected:Npn \driver_pdf_objects_enable: { }
28556 \cs_new_protected:Npn \driver_pdf_objects_disable: { }

```

(End definition for `\driver_pdf_compresslevel:n`, `\driver_pdf_objects_enable:`, and `\driver_pdf_objects_disable:`. These functions are documented on page 267.)

28557 ⟨/dvips⟩

### 45.3 pdfmode driver

28558 ⟨\*pdfmode⟩

The direct PDF driver covers both pdfTEX and LuaTEX. The latter renames and restructures the driver primitives but this can be handled at one level of abstraction. As

such, we avoid using two separate drivers for this material at the cost of some x-type definitions to get everything expanded up-front.

#### 45.3.1 Basics

`\__driver_literal_pdf:n`  
`\__driver_literal_pdf:x`

This is equivalent to `\special{pdf:}` but the engine can track it. Without the `direct` keyword everything is kept in sync: the transformation matrix is set to the current point automatically. Note that this is still inside the text (BT ... ET block).

```
28559 \cs_new_protected:Npx \__driver_literal_pdf:n #1
28560 {
28561     \cs_if_exist:NTF \tex_pdfextension:D
28562         { \tex_pdfextension:D literal }
28563         { \tex_pdfliteral:D }
28564         { \exp_not:N \exp_not:n {#1} }
28565     }
28566 \cs_generate_variant:Nn \__driver_literal_pdf:n { x }

(End definition for \__driver_literal_pdf:n.)
```

`\__driver_scope_begin:`  
`\__driver_scope_end:`

Higher-level interfaces for saving and restoring the graphic state.

```
28567 \cs_new_protected:Npx \__driver_scope_begin:
28568 {
28569     \cs_if_exist:NTF \tex_pdfextension:D
28570         { \tex_pdfextension:D save \scan_stop: }
28571         { \tex_pdfsave:D }
28572     }
28573 \cs_new_protected:Npx \__driver_scope_end:
28574 {
28575     \cs_if_exist:NTF \tex_pdfextension:D
28576         { \tex_pdfextension:D restore \scan_stop: }
28577         { \tex_pdfrestore:D }
28578 }
```

(End definition for `\__driver_scope_begin:` and `\__driver_scope_end:)`)

`\__driver_matrix:n`  
`\__driver_matrix:x`

Here the appropriate function is set up to insert an affine matrix into the PDF. With pdfTeX and LuaTeX in direct PDF output mode there is a primitive for this, which only needs the rotation/scaling/skew part.

```
28579 \cs_new_protected:Npx \__driver_matrix:n #1
28580 {
28581     \cs_if_exist:NTF \tex_pdfextension:D
28582         { \tex_pdfextension:D setmatrix }
28583         { \tex_pdfsetmatrix:D }
28584         { \exp_not:N \exp_not:n {#1} }
28585     }
28586 \cs_generate_variant:Nn \__driver_matrix:n { x }
```

(End definition for `\__driver_matrix:n`.)

#### 45.3.2 Box operations

`\driver_box_use_clip:N`

The general method is to save the current location, define a clipping path equivalent to the bounding box, then insert the content at the current position and in a zero width box. The “real” width is then made up using a horizontal skip before tidying up. There are other approaches that can be taken (for example using XForm objects), but the logic here

shares as much code as possible and uses the same conversions (and so same rounding errors) in all cases.

```

28587 \cs_new_protected:Npn \driver_box_use_clip:N #1
28588 {
28589   \__driver_scope_begin:
28590   \__driver_literal_pdf:x
28591   {
28592     O~
28593     \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
28594     \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
28595     \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
28596     re~W~n
28597   }
28598   \hbox_overlap_right:n { \box_use:N #1 }
28599   \__driver_scope_end:
28600   \skip_horizontal:n { \box_wd:N #1 }
28601 }
```

(End definition for `\driver_box_use_clip:N`. This function is documented on page 261.)

`\driver_box_use_rotate:Nn`

```
\__driver_box_use_rotate:Nn
  \l_driver_cos_fp
  \l_driver_sin_fp
```

Rotations are set using an affine transformation matrix which therefore requires sine/cosine values not the angle itself. We store the rounded values to avoid rounding twice. There are also a couple of comparisons to ensure that  $-0$  is not written to the output, as this avoids any issues with problematic display programs. Note that numbers are compared to 0 after rounding.

```

28602 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
28603   { \exp_args:NNF \__driver_box_use_rotate:Nn #1 { \fp_eval:n {#2} } }
28604 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
28605   {
28606     \__driver_scope_begin:
28607     \box_set_wd:Nn #1 { Opt }
28608     \fp_set:Nn \l_driver_cos_fp { round ( cosd ( #2 ) , 5 ) }
28609     \fp_compare:nNnT \l_driver_cos_fp = \c_zero_fp
28610       { \fp_zero:N \l_driver_cos_fp }
28611     \fp_set:Nn \l_driver_sin_fp { round ( sind ( #2 ) , 5 ) }
28612     \__driver_matrix:x
28613   {
28614     \fp_use:N \l_driver_cos_fp \c_space_tl
28615     \fp_compare:nNnTF \l_driver_sin_fp = \c_zero_fp
28616       { 0~0 }
28617     {
28618       \fp_use:N \l_driver_sin_fp
28619       \c_space_tl
28620       \fp_eval:n { -\l_driver_sin_fp }
28621     }
28622     \c_space_tl
28623     \fp_use:N \l_driver_cos_fp
28624   }
28625   \box_use:N #1
28626   \__driver_scope_end:
28627 }
```

```

28628 \fp_new:N \l_driver_cos_fp
28629 \fp_new:N \l_driver_sin_fp
```

(End definition for `\driver_box_use_rotate:Nn` and others. This function is documented on page 262.)

### \driver\_box\_use\_scale:Nnn

The same idea as for rotation but without the complexity of signs and cosines.

```
28630 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
28631 {
28632     \__driver_scope_begin:
28633     \__driver_matrix:x
28634     {
28635         \fp_eval:n { round ( #2 , 5 ) } ~
28636         0~0~
28637         \fp_eval:n { round ( #3 , 5 ) }
28638     }
28639     \hbox_overlap_right:n { \box_use:N #1 }
28640     \__driver_scope_end:
28641 }
```

(End definition for `\driver_box_use_scale:Nnn`. This function is documented on page 262.)

### 45.3.3 Images

#### \l\_driver\_image\_attr\_tl

In PDF mode, additional attributes of an image (such as page number) are needed both to obtain the bounding box and when inserting the image: this occurs as the image dictionary approach means they are read as part of the bounding box operation. As such, it is easier to track additional attributes using a dedicated `tl` rather than build up the same data twice.

```
28642 \tl_new:N \l_driver_image_attr_tl
```

(End definition for `\l_driver_image_attr_tl`.)

#### \\_\_driver\_image\_getbb\_jpg:n \\_\_driver\_image\_getbb\_pdf:n \\_\_driver\_image\_getbb\_png:n \\_\_driver\_image\_getbb\_auxi:n \\_driver\_image\_getbb\_auxii:n

Getting the bounding box here requires us to box up the image and measure it. To deal with the difference in feature support in bitmap and vector images but keeping the common parts, there is a little work to do in terms of auxiliaries. The key here is to notice that we need two forms of the attributes: a “short” set to allow us to track for caching, and the full form to pass to the primitive.

```
28643 \cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
28644 {
28645     \int_zero:N \l_image_page_int
28646     \tl_clear:N \l_image_pagebox_tl
28647     \tl_set:Nx \l_driver_image_attr_tl
28648     {
28649         \tl_if_empty:NF \l_image_decode_tl
28650             { :D \l_image_decodearray_tl }
28651         \bool_if:NT \l_image_interpolate_bool
28652             { :I }
28653     }
28654     \tl_clear:N \l_driver_image_attr_tl
28655     \__driver_image_getbb_auxi:n {#1}
28656 }
28657 \cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
28658 \cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
28659 {
28660     \tl_clear:N \l_image_decode_tl
28661     \bool_set_false:N \l_image_interpolate_bool
```

```

28662     \tl_set:Nx \l__driver_image_attr_tl
28663     {
28664         : \l_image_pagebox_tl
28665         \int_compare:nNnT \l_image_page_int > 1
28666             { :P \int_use:N \l_image_page_int }
28667     }
28668     \__driver_image_getbb_auxi:n {#1}
28669 }
28670 \cs_new_protected:Npn \__driver_image_getbb_auxi:n #1
28671 {
28672     \image_bb_restore:xF {#1 \l__driver_image_attr_tl }
28673         { \__driver_image_getbb_auxii:n {#1} }
28674 }
28675 % \begin{macrocode}
28676 % Measuring the image is done by boxing up: for PDF images we could
28677 % use |\tex_pdximage:D|, but if doesn't work for other types.
28678 % As the box always starts at $(0,0)$ there is no need to worry about
28679 % the lower-left position.
28680 % \begin{macrocode}
28681 \cs_new_protected:Npn \__driver_image_getbb_auxii:n #1
28682 {
28683     \tex_immediate:D \tex_pdximage:D
28684         \bool_lazy_or:nnT
28685             { \l_image_interpolate_bool }
28686             { ! \tl_if_empty_p:N \l_image_decodearray_tl }
28687         {
28688             attr ~
28689             {
28690                 \tl_if_empty:NF \l_image_decode_tl
28691                     { /Decode~[ \l_image_decodearray_tl ] }
28692                     \bool_if:NT \l_image_interpolate_bool
28693                         { /Interpolate~true }
28694             }
28695         }
28696         \int_compare:nNnT \l_image_page_int > 0
28697             { page ~ \int_use:N \l_image_page_int }
28698             \tl_if_empty:NF \l_image_pagebox_tl
28699                 { \l_image_pagebox_tl }
28700             {#1}
28701             \hbox_set:Nn \l__driver_tmp_box
28702                 { \tex_pdfrefximage:D \tex_pdflastximage:D }
28703                 \dim_set:Nn \l_image_urx_dim { \box_wd:N \l__driver_tmp_box }
28704                 \dim_set:Nn \l_image_ury_dim { \box_ht:N \l__driver_tmp_box }
28705                 \int_const:cn { c__driver_image_ #1 \l__driver_image_attr_tl _int }
28706                     { \tex_the:D \tex_pdflastximage:D }
28707                     \image_bb_save:x {#1 \l__driver_image_attr_tl }
28708     }

```

(End definition for `\__driver_image_getbb_jpg:n` and others.)

`\__driver_image_include_jpg:n`  
`\__driver_image_include_pdf:n`  
`\__driver_image_include_png:n`

Images are already loaded for the measurement part of the code, so inclusion is straightforward, with only any attributes to worry about. The latter carry through from determination of the bounding box.

```
28709 \cs_new_protected:Npn \__driver_image_include_jpg:n #1
```

```

28710   {
28711     \tex_pfdrefximage:D
28712       \int_use:c { c__driver_image_ #1 \l__driver_image_attr_tl _int }
28713   }
28714 \cs_new_eq:NN \__driver_image_include_pdf:n \__driver_image_include_jpg:n
28715 \cs_new_eq:NN \__driver_image_include_png:n \__driver_image_include_jpg:n

(End definition for \__driver_image_include_jpg:n, \__driver_image_include_pdf:n, and \__driver-
image_include_png:n.)

```

#### 45.3.4 PDF Objects

\g\_\_driver\_pdf\_object\_prop

For tracking objects to allow finalisation.

```
28716 \prop_new:N \g__driver_pdf_object_prop
```

(End definition for \g\_\_driver\_pdf\_object\_prop.)

\driver\_pdf\_object\_new:nn

\driver\_pdf\_object\_ref:n

Declaring objects means reserving at the PDF level plus starting tracking.

```

28717 \group_begin:
28718   \cs_set_protected:Npn \__driver_tmp:w #1#2
28719   {
28720     \cs_new_protected:Npx \driver_pdf_object_new:nn ##1##2
28721     {
28722       #1 reserveobjnum ~
28723       \int_const:cn
28724         { g__driver_pdf_object_ \exp_not:N \tl_to_str:n {##1} _int }
28725         {##2}
28726       \prop_gput:Nnn \exp_not:N \g__driver_pdf_object_prop {##1} {##2}
28727     }
28728   }
28729   \cs_if_exist:NTF \tex_pdfextension:D
28730   {
28731     \__driver_tmp:w
28732       { \tex_pdfextension:D obj ~ }
28733       { \tex_pdffeedback:D lastobj \scan_stop: }
28734   }
28735   { \__driver_tmp:w { \tex_pdfobj:D } { \tex_pdflastobj:D } }
28736 \group_end:
28737 \cs_new:Npn \driver_pdf_object_ref:n #1
28738   { \int_use:c { g__driver_pdf_object_ \tl_to_str:n {##1} _int } ~ 0 ~ R }

(End definition for \driver_pdf_object_new:nn and \driver_pdf_object_ref:n. These functions are
documented on page 266.)
```

\driver\_pdf\_object\_write:nn

\\_\_driver\_exp\_not\_i:nn

\\_\_driver\_exp\_not\_ii:nn

Writing the data needs a little information about the structure of the object.

```

28739 \group_begin:
28740   \cs_set_protected:Npn \__driver_tmp:w #1
28741   {
28742     \cs_new_protected:Npn \driver_pdf_object_write:nn ##1##2
28743     {
28744       \tex_immediate:D #1 useobjnum ~
28745       \int_use:c
28746         { g__driver_pdf_object_ \tl_to_str:n {##1} _int }
28747         { \prop_item:Nn \g__driver_pdf_object_prop {##1} }
```

```

28749 {
28750   { array } { { [ ~ \exp_not:n {##2} ~ ] } }
28751   { dict } { { << ~ \exp_not:n {##2} ~ >> } }
28752   { fstream }
28753   {
28754     stream ~ attr ~ { \_driver_exp_not_i:nn ##2 } ~
28755       file ~ { \_driver_exp_not_ii:nn ##2 }
28756   }
28757   { stream }
28758   {
28759     stream ~ attr ~ { \_driver_exp_not_i:nn ##2 } ~
28760       { \_driver_exp_not_ii:nn ##2 }
28761   }
28762 }
28763 }
28764 }
28765 \cs_if_exist:NTF \tex_pdfextension:D
28766   { \_driver_tmp:w { \tex_pdfextension:D obj ~ } }
28767   { \_driver_tmp:w { \tex_pdfobj:D } }
28768 \group_end:
28769 \cs_new:Npn \_driver_exp_not_i:nn #1#2 { \exp_not:n {#1} }
28770 \cs_new:Npn \_driver_exp_not_ii:nn #1#2 { \exp_not:n {#2} }

(End definition for \driver_pdf_object_write:nn, \_driver_exp_not_i:nn, and \_driver_exp-not_ii:nn. This function is documented on page 267.)

```

#### 45.3.5 PDF Structure

`\driver_pdf_compresslevel:n` Simply pass data to the engine.

```

\driver_pdf_objects_enable:
\driver_pdf_objects_disable:
\_\_driver_pdf_objectlevel:n
28771 \cs_new_protected:Npx \driver_pdf_compresslevel:n #1
28772   {
28773     \cs_if_exist:NTF \tex_pdfcompresslevel:D
28774       { \tex_pdfcompresslevel:D }
28775       { \tex_pdfvariable:D compresslevel }
28776       \exp_not:N \int_value:w \exp_not:N \int_eval:n {#1} \scan_stop:
28777   }
28778 \cs_new_protected:Npn \driver_pdf_objects_enable:
28779   { \_driver_pdf_objectlevel:n { 2 } }
28780 \cs_new_protected:Npn \driver_pdf_objects_disable:
28781   { \_driver_pdf_objectlevel:n { 0 } }
28782 \cs_new_protected:Npx \_driver_pdf_objectlevel:n #1
28783   {
28784     \cs_if_exist:NTF \tex_pdfobjcompresslevel:D
28785       { \tex_pdfobjcompresslevel:D }
28786       { \tex_pdfvariable:D objcompresslevel }
28787       #1 \scan_stop:
28788   }

```

(End definition for \driver\_pdf\_compresslevel:n and others. These functions are documented on page 267.)

28789 `</pdfmode>`

## 45.4 dvipdfmx driver

```
28790  {*dvipdfmx | xdvipdfmx}
```

The `dvipdfmx` shares code with the PDF mode one (using the common section to this file) but also with `xdvipdfmx`. The latter is close to identical to `dvipdfmx` and so all of the code here is extracted for both drivers, with some `clean` up for `xdvipdfmx` as required.

### 45.4.1 Basics

`\__driver_literal_pdf:n` Equivalent to `pdf:content` but favored as the link to the pdfTeX primitive approach is clearer.

```
28791  \cs_new_protected:Npn \__driver_literal_pdf:n #1
28792    { \__driver_literal:n { pdf:literal~ #1 } }
28793  \cs_generate_variant:Nn \__driver_literal_pdf:n { x }
```

(*End definition for `\__driver_literal_pdf:n`.*)

`\__driver_scope_begin:` Scoping is done using the driver-specific specials.

```
28794  \cs_new_protected:Npn \__driver_scope_begin:
28795    { \__driver_literal:n { x:gsave } }
28796  \cs_new_protected:Npn \__driver_scope_end:
28797    { \__driver_literal:n { x:grestore } }
```

(*End definition for `\__driver_scope_begin:` and `\__driver_scope_end::`*)

### 45.4.2 Box operations

`\driver_box_use_clip:N` The code here is identical to that for `pdfmode`: unlike rotation and scaling, there is no higher-level support in the driver for clipping.

```
28798  \cs_new_protected:Npn \driver_box_use_clip:N #1
28799    {
28800      \__driver_scope_begin:
28801      \__driver_literal_pdf:x
28802        {
28803          0~
28804          \dim_to_decimal_in_bp:n { -\box_dp:N #1 } ~
28805          \dim_to_decimal_in_bp:n { \box_wd:N #1 } ~
28806          \dim_to_decimal_in_bp:n { \box_ht:N #1 + \box_dp:N #1 } ~
28807          re~W~n
28808        }
28809        \hbox_overlap_right:n { \box_use:N #1 }
28810        \__driver_scope_end:
28811        \skip_horizontal:n { \box_wd:N #1 }
28812    }
```

(*End definition for `\driver_box_use_clip:N`. This function is documented on page 261.*)

`\driver_box_use_rotate:Nn` Rotating in `(x)dvipdfmx` can be implemented using either PDF or driver-specific code. The former approach however is not “aware” of the content of boxes: this means that any embedded links would not be adjusted by the rotation. As such, the driver-native approach is preferred: the code therefore is similar (though not identical) to the `dvips` version (notice the rotation angle here is positive). As for `dvips`, zero rotation is written as `0` not `-0`.

```

28813 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
28814   { \exp_args:NNf \__driver_box_use_rotate:Nn #1 { \fp_eval:n {#2} } }
28815 \cs_new_protected:Npn \__driver_box_use_rotate:Nn #1#2
28816   {
28817     \__driver_scope_begin:
28818     \__driver_literal:x
28819     {
28820       x:rotate-
28821       \fp_compare:nNnTF {#2} = \c_zero_fp
28822         { 0 }
28823         { \fp_eval:n { round ( #2 , 5 ) } }
28824     }
28825     \box_use:N #1
28826     \__driver_scope_end:
28827   }

(End definition for \driver_box_use_rotate:Nn and \__driver_box_use_rotate:Nn. This function is documented on page 262.)
```

**\driver\_box\_use\_scale:Nnn** Much the same idea for scaling: use the higher-level driver operation to allow for box content.

```

28828 \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
28829   {
28830     \__driver_scope_begin:
28831     \__driver_literal:x
28832     {
28833       x:scale-
28834       \fp_eval:n { round ( #2 , 5 ) } ~
28835       \fp_eval:n { round ( #3 , 5 ) }
28836     }
28837     \hbox_overlap_right:n { \box_use:N #1 }
28838     \__driver_scope_end:
28839   }
```

(End definition for \driver\_box\_use\_scale:Nnn. This function is documented on page 262.)

#### 45.4.3 Images

**\\_\_driver\_image\_getbb\_eps:n** Simply use the generic functions: only for dvipdfmx in the extraction cases.  
**\\_\_driver\_image\_getbb\_jpg:n**  
**\\_\_driver\_image\_getbb\_pdf:n**  
**\\_\_driver\_image\_getbb\_png:n**

```

28840 \cs_new_eq:NN \__driver_image_getbb_eps:n \image_read_bb:n
28841 (*dvipdfmx)
28842 \cs_new_protected:Npn \__driver_image_getbb_jpg:n #1
28843   {
28844     \int_zero:N \l_image_page_int
28845     \tl_clear:N \l_image_pagebox_tl
28846     \image_extract_bb:n {#1}
28847   }
28848 \cs_new_eq:NN \__driver_image_getbb_png:n \__driver_image_getbb_jpg:n
28849 \cs_new_protected:Npn \__driver_image_getbb_pdf:n #1
28850   {
28851     \tl_clear:N \l_image_decode_tl
28852     \bool_set_false:N \l_image_interpolate_bool
28853     \image_extract_bb:n {#1}
28854   }
28855 
```

(End definition for `\_driver_image_getbb_eps:n` and others.)

`\g_driver_image_int` Used to track the object number associated with each image.

28856 `\int_new:N \g_driver_image_int`

(End definition for `\g_driver_image_int`.)

`\_driver_image_include_eps:n`  
`\_driver_image_include_jpg:n`  
`\_driver_image_include_pdf:n`  
`\_driver_image_include_png:n`  
`\_driver_image_include_auxi:nn`  
`\_driver_image_include_auxii:nnn`  
`\_driver_image_include_auxii:xnn`  
`\_driver_image_include_auxiii:nn`

The special syntax depends on the file type. There is a difference in how PDF images are best handled between `dvipdfmx` and `xvipdfmx`: for the latter it is better to use the primitive route. The relevant code for that is included later in this file.

28857 `\cs_new_protected:Npn \_driver_image_include_eps:n #1`  
28858   {  
28859     `\_driver_literal:n { PSfile = #1 }`  
28860   }  
28861 `\cs_new_protected:Npn \_driver_image_include_jpg:n #1`  
28862   {  
28863     `\_driver_image_include_auxi:nn {#1} { image }`  
28864   }  
28865 `\cs_new_protected:Npn \_driver_image_include_pdf:n #1`  
28866   {  
28867     `\_driver_image_include_auxi:nn {#1} { epdf }`  
28868   }

Image inclusion is set up to use the fact that each image is stored in the PDF as an XObject. This means that we can include repeated images only once and refer to them. To allow that, track the nature of each image: much the same as for the direct PDF mode case.

28868 `\cs_new_protected:Npn \_driver_image_include_auxi:nn #1#2`  
28869   {  
28870     `\_driver_image_include_auxii:xnn`  
28871     {  
28872       `\tl_if_empty:NF \l_image_pagebox_tl`  
28873       {  
28874         `\int_compare:nNnT \l_image_page_int > 1`  
28875         {  
28876           `\tl_if_empty:NF \l_image_decode_tl`  
28877           {  
28878             `\int_use:c \l_image_decodearray_tl`  
28879             {  
28880               `\bool_if:NT \l_image_interpolate_bool`  
28881               {  
28882                 `\int_use:c \l_image_interpolate_bool`  
28883                 {  
28884                 `\cs_new_protected:Npn \_driver_image_include_auxii:nnn #1#2#3`  
28885                 {  
28886                 `\int_if_exist:cTF { c_driver_image_ #2#1 _int }`  
28887                 {  
28888                 `\_driver_literal:x`  
28889                 {  
28890                 `\_driver_image_include_auxiii:nn {#2} {#1} {#3}`   
28891         }  
28892 `\cs_generate_variant:Nn \_driver_image_include_auxii:nnn { x }`

Inclusion using the specials is relatively straight-forward, but there is one wrinkle. To get the `pagebox` correct for PDF images in all cases, it is necessary to provide both that information and the `bbox` argument: odd things happen otherwise!

```

28893 \cs_new_protected:Npn \__driver_image_include_auxiii:n { #1#2#3
28894 {
28895     \int_gincr:N \g__driver_image_int
28896     \int_const:cn { c__driver_image_ #1#2 _int } { \g__driver_image_int }
28897     \__driver_literal:x
28898 {
28899     pdf:#3-
28900     @image \int_use:c { c__driver_image_ #1#2 _int }
28901     \int_compare:nNnT \l_image_page_int > 1
28902     { page ~ \int_use:N \l_image_page_int \c_space_tl }
28903     \tl_if_empty:NF \l_image_pagebox_tl
28904     {
28905         pagebox ~ \l_image_pagebox_tl \c_space_tl
28906         bbox ~
28907             \dim_to_decimal_in_bp:n \l_image_llx_dim \c_space_tl
28908             \dim_to_decimal_in_bp:n \l_image_lly_dim \c_space_tl
28909             \dim_to_decimal_in_bp:n \l_image_urx_dim \c_space_tl
28910             \dim_to_decimal_in_bp:n \l_image_ury_dim \c_space_tl
28911     }
28912     (#1)
28913     \bool_lazy_or:nnT
28914     { \l_image_interpolate_bool }
28915     { ! \tl_if_empty_p:N \l_image_decodearray_tl }
28916     {
28917         <<
28918         \tl_if_empty:NF \l_image_decode_tl
28919         { /Decode~[ \l_image_decodearray_tl ] }
28920         \bool_if:NT \l_image_interpolate_bool
28921         { /Interpolate~true> }
28922     >>
28923     }
28924 }
28925 }

```

(End definition for `\__driver_image_include_eps:n` and others.)

#### 45.4.4 PDF Objects

`\g__driver_pdf_object_int` For tracking objects to allow finalisation.

```

28926 \int_new:N \g__driver_pdf_object_int
28927 \prop_new:N \g__driver_pdf_object_prop

```

(End definition for `\g__driver_pdf_object_int` and `\g__driver_pdf_object_prop`.)

`\driver_pdf_object_new:nn` Objects are tracked at the macro level, but we don't have to do anything at this stage.

```

28928 \cs_new_protected:Npn \driver_pdf_object_new:nn { #1#2
28929 {
28930     \int_gincr:N \g__driver_pdf_object_int
28931     \int_const:cn
28932     { g__driver_pdf_object_ \tl_to_str:n {#1} _int }
28933     { \g__driver_pdf_object_int }
28934     \prop_gput:Nnn \g__driver_pdf_object_prop {#1} {#2}
28935 }
28936 \cs_new:Npn \driver_pdf_object_ref:n {#1
28937     { \g__driver_pdf_object_ \int_use:c { g__driver_pdf_object_ \tl_to_str:n {#1} _int } }

```

(End definition for `\driver_pdf_object_new:nn` and `\driver_pdf_object_ref:n`. These functions are documented on page 266.)

#### `\driver_pdf_object_write:nn`

This is where we choose the actual type.

```
28938 \cs_new_protected:Npn \driver_pdf_object_write:nn #1#2
28939   {
28940     \exp_args:Nx \__driver_pdf_object_write:nnn
28941       { \prop_item:Nn \g__driver_pdf_object_prop {#1} } {#1} {#2}
28942   }
28943 \cs_new_protected:Npn \__driver_pdf_object_write:nnn #1#2#3
28944   { \use:c { __driver_pdf_object_write_ #1 :nn } {#2} {#3} }
28945 \cs_new_protected:Npn \__driver_pdf_object_write_array:nn #1#2
28946   {
28947     \__driver_literal:x
28948     {
28949       pdf:obj ~ \driver_pdf_object_ref:n {#1} ~
28950         [ ~ \exp_not:n {#2} ~ ]
28951     }
28952   }
28953 \cs_new_protected:Npn \__driver_pdf_object_write_dict:nn #1#2
28954   {
28955     \__driver_literal:x
28956     {
28957       pdf:obj ~ \driver_pdf_object_ref:n {#1} ~
28958         << ~ \exp_not:n {#2} ~ >>
28959     }
28960   }
28961 \cs_new_protected:Npn \__driver_pdf_object_write_fstream:nn #1#2
28962   { \__driver_pdf_object_write_stream:nnnn { f } {#1} #2 }
28963 \cs_new_protected:Npn \__driver_pdf_object_write_stream:nn #1#2
28964   { \__driver_pdf_object_write_stream:nnnn { } {#1} #2 }
28965 \cs_new_protected:Npn \__driver_pdf_object_write_stream:nnnn #1#2#3#4
28966   {
28967     \__driver_literal:x
28968     {
28969       pdf: #1 stream ~ \driver_pdf_object_ref:n {#2} ~
28970         ( \exp_not:n {#4} ) ~ << \exp_not:n {#3} >>
28971     }
28972   }
28973 }
```

(End definition for `\driver_pdf_object_write:nn` and others. This function is documented on page 267.)

#### 45.4.5 PDF Structure

##### `\driver_pdf_compresslevel:n`

##### `\driver_pdf_objects_enable:`

##### `\driver_pdf_objects_disable:`

Pass data to the driver: these are a one-shot.

```
28974 \cs_new_protected:Npn \driver_pdf_compresslevel:n #1
28975   { \__driver_literal:x { dvipdfmx:config~z~ \int_eval:n {#1} } }
28976 \cs_new_protected:Npn \driver_pdf_objects_enable: { }
28977 \cs_new_protected:Npn \driver_pdf_objects_disable:
28978   { \__driver_literal:n { dvipdfmx:config~C~0x40 } }
```

(End definition for `\driver_pdf_compresslevel:n`, `\driver_pdf_objects_enable:`, and `\driver_pdf_objects_disable:`. These functions are documented on page 267.)

28979 `</dvipdfmx | xdvipdfmx>`

## 45.5 xdvipdfmx driver

28980 `<*xdvipdfmx>`

### 45.5.1 Images

`\_driver_image_getbb_jpg:n`  
`\_driver_image_getbb_pdf:n`  
`\_driver_image_getbb_png:n`  
  `\_driver_image_getbb_auxi:nN`  
  `\_driver_image_getbb_auxii:nNnN`  
  `\_driver_image_getbb_auxiii:nNnn`  
  `\_driver_image_getbb_auxiv:mNnn`  
  `\_driver_image_getbb_auxv:vNnn`  
  `\_driver_image_getbb_auxv:vNnnn`  
  `\_driver_image_getbb_pagebox:w`

28981 `\cs_new_protected:Npn \_driver_image_getbb_jpg:n #1`  
28982   `{`  
28983     `\int_zero:N \l_image_page_int`  
28984     `\tl_clear:N \l_image_pagebox_tl`  
28985     `\_driver_image_getbb_auxi:nN {#1} \tex_XeTeXpicfile:D`  
28986   `}`  
28987 `\cs_new_eq:NN \_driver_image_getbb_png:n \_driver_image_getbb_jpg:n`  
28988 `\cs_new_protected:Npn \_driver_image_getbb_pdf:n #1`  
28989   `{`  
28990     `\tl_clear:N \l_image_decode_tl`  
28991     `\bool_set_false:N \l_image_interpolate_bool`  
28992     `\_driver_image_getbb_auxi:nN {#1} \tex_XeTeXpdffile:D`  
28993   `}`  
28994 `\cs_new_protected:Npn \_driver_image_getbb_auxi:nN #1#2`  
28995   `{`  
28996     `\int_compare:nNnTF \l_image_page_int > 1`  
28997       `{ \_driver_image_getbb_auxii:vNn \l_image_page_int {#1} #2 }`  
28998       `{ \_driver_image_getbb_auxiii:nNnn {#1} #2 }`  
28999   `}`  
29000 `\cs_new_protected:Npn \_driver_image_getbb_auxii:nnN #1#2#3`  
29001   `{ \_driver_image_getbb_aux:nNnn {#2} #3 { :P #1 } { page #1 } }`  
29002 `\cs_generate_variant:Nn \_driver_image_getbb_auxii:nnN { V }`  
29003 `\cs_new_protected:Npn \_driver_image_getbb_auxiii:nNnn #1#2#3#4`  
29004   `{`  
29005     `\tl_if_empty:NTF \l_image_pagebox_tl`  
29006       `{ \_driver_image_getbb_auxiv:vNnnn \l_image_pagebox_tl }`  
29007       `{ \_driver_image_getbb_auxv:nNnn }`  
29008       `{#1} #2 {#3} {#4}`  
29009   `}`  
29010 `\cs_new_protected:Npn \_driver_image_getbb_auxiv:nnNnn #1#2#3#4#5`  
29011   `{`  
29012     `\use:x`  
29013       `{`  
29014         `\_driver_image_getbb_auxv:nNnn {#2} #3 { : #1 #4 }`  
29015         `{ #5 ~ \_driver_image_getbb_pagebox:w #1 }`  
29016       `}`  
29017   `}`  
29018 `\cs_generate_variant:Nn \_driver_image_getbb_auxiv:nnNnn { V }`  
29019 `\cs_new_protected:Npn \_driver_image_getbb_auxv:nNnn #1#2#3#4`  
29020   `{`

```

29021      \image_bb_restore:nF {#1#3}
29022      { \__driver_image_getbb_auxvi:nNnn {#1} #2 {#3} {#4} }
29023    }
29024 \cs_new_protected:Npn \__driver_image_getbb_auxvi:nNnn #1#2#3#4
29025  {
29026    \hbox_set:Nn \l__driver_tmp_box { #2 #1 ~ #4 }
29027    \dim_set:Nn \l_image_utm_dim { \box_wd:N \l__driver_tmp_box }
29028    \dim_set:Nn \l_image_ury_dim { \box_ht:N \l__driver_tmp_box }
29029    \image_bb_save:n {#1#3}
29030  }
29031 \cs_new:Npn \__driver_image_getbb_pagebox:w #1 box {#1}

```

(End definition for `\__driver_image_getbb_jpg:n` and others.)

`\__driver_image_include_pdf:n`

For PDF images, properly supporting the `pagebox` concept in X<sub>E</sub>T<sub>E</sub>X is best done using the `\tex_XeTeXpdffile:D` primitive. The syntax here is the same as for the image measurement part, although we know at this stage that there must be some valid setting for `\l_image_pagebox_tl`.

```

29032 \cs_new_protected:Npn \__driver_image_include_pdf:n #1
29033  {
29034    \tex_XeTeXpdffile:D "#1" ~
29035    \int_compare:nNnT \l_image_page_int > 0
29036    { page~ \int_use:N \l_image_page_int }
29037    \__driver_image_getbb_auxiv:VnNnn \l_image_pagebox_tl
29038  }

```

(End definition for `\__driver_image_include_pdf:n`.)

29039 ⟨/xdvipdfmx⟩

## 45.6 Drawing commands: pdfmode and (x)dvipdfmx

Both `pdfmode` and `(x)dvipdfmx` directly produce PDF output and understand a shared set of specials for drawing commands.

29040 ⟨\*dvipdfmx | pdfmode | xdvipdfmx⟩

### 45.6.1 Drawing

`\__driver_draw_literal:n` Pass data through using a dedicated interface.

```

\__driver_draw_literal:x
29041 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal_pdf:n
29042 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for `\__driver_draw_literal:n`.)

`\driver_draw_begin:`

No special requirements here, so simply set up a drawing scope.

```

\driver_draw_end:
29043 \cs_new_protected:Npn \driver_draw_begin:
29044  { \driver_scope_begin: }
29045 \cs_new_protected:Npn \driver_draw_end:
29046  { \driver_scope_end: }

```

(End definition for `\driver_draw_begin:` and `\driver_draw_end:`. These functions are documented on page 263.)

\driver\_draw\_scope\_begin: Use the driver-level scope mechanisms.

```
29047 \cs_new_eq:NN \driver_draw_scope_begin: \__driver_scope_begin:
29048 \cs_new_eq:NN \driver_draw_scope_end: \__driver_scope_end:
```

(End definition for \driver\_draw\_scope\_begin: and \driver\_draw\_scope\_end:. These functions are documented on page 263.)

\driver\_draw\_moveto:nn  
\driver\_draw\_lineto:nn Path creation operations all resolve directly to PDF primitive steps, with only the need to convert to bp.

```
29049 \cs_new_protected:Npn \driver_draw_moveto:nn #1#2
29050 {
29051     \__driver_draw_literal:x
29052         { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ m }
29053     }
29054 \cs_new_protected:Npn \driver_draw_lineto:nn #1#2
29055 {
29056     \__driver_draw_literal:x
29057         { \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~ 1 }
29058     }
29059 \cs_new_protected:Npn \driver_draw_curveto:nnnnnn #1#2#3#4#5#6
29060 {
29061     \__driver_draw_literal:x
29062         {
29063             \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
29064             \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
29065             \dim_to_decimal_in_bp:n {#5} ~ \dim_to_decimal_in_bp:n {#6} ~
29066             c
29067         }
29068     }
29069 \cs_new_protected:Npn \driver_draw_rectangle:nnnn #1#2#3#4
29070 {
29071     \__driver_draw_literal:x
29072         {
29073             \dim_to_decimal_in_bp:n {#1} ~ \dim_to_decimal_in_bp:n {#2} ~
29074             \dim_to_decimal_in_bp:n {#3} ~ \dim_to_decimal_in_bp:n {#4} ~
29075             re
29076         }
29077     }
```

(End definition for \driver\_draw\_moveto:nn and others. These functions are documented on page 263.)

\driver\_draw\_evenodd\_rule:  
\driver\_draw\_nonzero\_rule:  
\g\_\_driver\_draw\_eor\_bool The even-odd rule here can be implemented as a simply switch.

```
29078 \cs_new_protected:Npn \driver_draw_evenodd_rule:
29079     { \bool_gset_true:N \g__driver_draw_eor_bool }
29080 \cs_new_protected:Npn \driver_draw_nonzero_rule:
29081     { \bool_gset_false:N \g__driver_draw_eor_bool }
29082 \bool_new:N \g__driver_draw_eor_bool
```

(End definition for \driver\_draw\_evenodd\_rule:, \driver\_draw\_nonzero\_rule:, and \g\_\_driver\_draw\_eor\_bool. These functions are documented on page 264.)

\driver\_draw\_closepath:  
\driver\_draw\_stroke:  
\driver\_draw\_closestroke:  
\driver\_draw\_fill:  
\driver\_draw\_fillstroke:  
\driver\_draw\_clip:  
\driver\_draw\_discardpath: Converting paths to output is again a case of mapping directly to PDF operations.

```
29083 \cs_new_protected:Npn \driver_draw_closepath:
29084     { \__driver_draw_literal:n { h } }
29085 \cs_new_protected:Npn \driver_draw_stroke:
```

```

29086 { \__driver_draw_literal:n { S } }
29087 \cs_new_protected:Npn \driver_draw_closestroke:
29088 { \__driver_draw_literal:n { s } }
29089 \cs_new_protected:Npn \driver_draw_fill:
29090 {
29091     \__driver_draw_literal:x
29092     { f \bool_if:NT \g__driver_draw_eor_bool * }
29093 }
29094 \cs_new_protected:Npn \driver_draw_fillstroke:
29095 {
29096     \__driver_draw_literal:x
29097     { B \bool_if:NT \g__driver_draw_eor_bool * }
29098 }
29099 \cs_new_protected:Npn \driver_draw_clip:
29100 {
29101     \__driver_draw_literal:x
29102     { W \bool_if:NT \g__driver_draw_eor_bool * }
29103 }
29104 \cs_new_protected:Npn \driver_draw_discardpath:
29105 { \__driver_draw_literal:n { n } }

```

(End definition for `\driver_draw_closepath:` and others. These functions are documented on page 263.)

`\driver_draw_dash_pattern:nn` Converting paths to output is again a case of mapping directly to PDF operations.

```

\__driver_draw_dash:n
\driver_draw_linewidth:n
\driver_draw_miterlimit:n
\driver_draw_cap_but:
\driver_draw_cap_round:
\driver_draw_cap_rectangle:
\driver_draw_join_miter:
\driver_draw_join_round:
\driver_draw_join_bevel:
29106 \cs_new_protected:Npn \driver_draw_dash_pattern:nn #1#2
29107 {
29108     \__driver_draw_literal:x
29109     {
29110         [
29111             \exp_args:Nf \use:n
29112             { \clist_map_function:nN {#1} \__driver_draw_dash:n }
29113         ] ~
29114             \dim_to_decimal_in_bp:n {#2} ~ d
29115     }
29116 }
29117 \cs_new:Npn \__driver_draw_dash:n #1
29118 { ~ \dim_to_decimal_in_bp:n {#1} }
29119 \cs_new_protected:Npn \driver_draw_linewidth:n #1
29120 {
29121     \__driver_draw_literal:x
29122     { \dim_to_decimal_in_bp:n {#1} ~ w }
29123 }
29124 \cs_new_protected:Npn \driver_draw_miterlimit:n #1
29125 { \__driver_draw_literal:x { \fp_eval:n {#1} ~ M } }
29126 \cs_new_protected:Npn \driver_draw_cap_but:
29127 { \__driver_draw_literal:n { 0 ~ J } }
29128 \cs_new_protected:Npn \driver_draw_cap_round:
29129 { \__driver_draw_literal:n { 1 ~ J } }
29130 \cs_new_protected:Npn \driver_draw_cap_rectangle:
29131 { \__driver_draw_literal:n { 2 ~ J } }
29132 \cs_new_protected:Npn \driver_draw_join_miter:
29133 { \__driver_draw_literal:n { 0 ~ j } }
29134 \cs_new_protected:Npn \driver_draw_join_round:

```

```

29135   { \__driver_draw_literal:n { 1 ~ j } }
29136 \cs_new_protected:Npn \driver_draw_join_bevel:
29137   { \__driver_draw_literal:n { 2 ~ j } }

```

(End definition for `\driver_draw_dash_pattern:nn` and others. These functions are documented on page 265.)

`\driver_draw_color_fill_cmyk:nnnn`  
`\driver_draw_color_stroke_cmyk:nnnn`  
`\driver_draw_color_fill_gray:n`  
`\driver_draw_color_stroke_gray:n`  
`\driver_draw_color_fill_rgb:nnn`  
`\driver_draw_color_stroke_rgb:nnn`

```

29138 \cs_new_protected:Npn \driver_draw_color_fill_cmyk:nnnn #1#2#3#4
29139   {
29140     \__driver_color_fill_select:x
29141     {
29142       \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
29143       \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
29144       k
29145     }
29146   }
29147 \cs_new_protected:Npn \driver_draw_color_stroke_cmyk:nnnn #1#2#3#4
29148   {
29149     \__driver_color_select:x
29150     {
29151       \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
29152       \fp_eval:n {#3} ~ \fp_eval:n {#4} ~
29153       k
29154     }
29155   }
29156 \cs_new_protected:Npn \driver_draw_color_fill_gray:n #1
29157   { \__driver_color_fill_select:x { \fp_eval:n {#1} ~ g } }
29158 \cs_new_protected:Npn \driver_draw_color_stroke_gray:n #1
29159   { \__driver_color_select:x { \fp_eval:n {#1} ~ G } }
29160 \cs_new_protected:Npn \driver_draw_color_fill_rgb:nnn #1#2#3
29161   {
29162     \__driver_color_fill_select:x
29163     { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ rg }
29164   }
29165 \cs_new_protected:Npn \driver_draw_color_stroke_rgb:nnn #1#2#3
29166   {
29167     \__driver_color_select:x
29168     { \fp_eval:n {#1} ~ \fp_eval:n {#2} ~ \fp_eval:n {#3} ~ RG }
29169   }
29170 (*pdfmode)
29171 \cs_new_eq:NN \__driver_color_fill_select:n \__driver_color_select:n
29172 (/pdfmode)
29173 (*dvipdfmx | xdvipdfmx)
29174 \cs_new_eq:NN \__driver_color_fill_select:n \__driver_draw_literal:n
29175 (/dvipdfmx | xdvipdfmx)
29176 \cs_generate_variant:Nn \__driver_color_fill_select:n { x }

```

(End definition for `\driver_draw_color_fill_cmyk:nnnn` and others. These functions are documented on page 265.)

`\driver_draw_cm:nnnn`  
`\__driver_draw_cm:nnnn`

Another split here between `pdfmode` and `(x)dvipdfmx`. In the former, we have a direct method to maintain alignment: the driver can use a matrix itself. For `(x)dvipdfmx`,

we can to decompose the matrix into rotations and a scaling, then use those operations as they are handled by the driver. (There is driver support for matrix operations in `(x)dvipdfmx`, but as a matched pair so not suitable for the “stand alone” transformation set up here.)

```

29177 \cs_new_protected:Npn \driver_draw_cm:nnnn #1#2#3#4
29178   {
29179   (*pdfmode)
29180     __driver_matrix:x
29181     {
29182       \fp_eval:n {#1} ~ \fp_eval:n {#2} ~
29183       \fp_eval:n {#3} ~ \fp_eval:n {#4}
29184     }
29185   
```

(End definition for `\driver_draw_cm:nnnn` and `\_driver_draw_cm:nnnn`. This function is documented on page [266](#).)

Internally, transformations for drawing are tracked as a matrix. Not all engines provide a way of dealing with this: if we use a raw matrix, the engine loses track of positions (for example for hyperlinks), and this is not desirable. They do, however, allow us to track rotations and scalings. Luckily, we can decompose any (two-dimensional) matrix into two rotations and a single scaling:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} \cos \beta & \sin \beta \\ -\sin \beta & \cos \beta \end{bmatrix} \begin{bmatrix} w_1 & 0 \\ 0 & w_2 \end{bmatrix} \begin{bmatrix} \cos \gamma & \sin \gamma \\ -\sin \gamma & \cos \gamma \end{bmatrix}$$

The parent matrix can be converted to

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} E & H \\ -H & E \end{bmatrix} + \begin{bmatrix} F & G \\ G & -F \end{bmatrix}$$

From these, we can find that

$$\begin{aligned} \frac{w_1 + w_2}{2} &= \sqrt{E^2 + H^2} \\ \frac{w_1 - w_2}{2} &= \sqrt{F^2 + G^2} \\ \gamma - \beta &= \tan^{-1}(G/F) \\ \gamma + \beta &= \tan^{-1}(H/E) \end{aligned}$$

at which point we just have to do various pieces of re-arrangement to get all of the values. (See J. Blinn, *IEEE Comput. Graph. Appl.*, 1996, **16**, 82–88.) There is one wrinkle: the PostScript (and PDF) way of specifying a transformation matrix exchanges where one would normally expect  $B$  and  $C$  to be.

```

29216  {*dvipdfmx | xdvipdfmx}
29217  \cs_new_protected:Npn \__driver_draw_cm_decompose:nnnnN #1#2#3#4#5
29218  {
29219    \use:x
29220    {
29221      \__driver_draw_cm_decompose_auxi:nnnnN
29222      { \fp_eval:n { (#1 + #4) / 2 } }
29223      { \fp_eval:n { (#1 - #4) / 2 } }
29224      { \fp_eval:n { (#3 + #2) / 2 } }
29225      { \fp_eval:n { (#3 - #2) / 2 } }
29226    }
29227    #5
29228  }
29229  \cs_new_protected:Npn \__driver_draw_cm_decompose_auxi:nnnnN #1#2#3#4#5
29230  {
29231    \use:x
29232    {
29233      \__driver_draw_cm_decompose_auxii:nnnnN
29234      { \fp_eval:n { 2 * sqrt ( #1 * #1 + #4 * #4 ) } }
29235      { \fp_eval:n { 2 * sqrt ( #2 * #2 + #3 * #3 ) } }
29236      { \fp_eval:n { atan ( #3 , #2 ) } }
29237      { \fp_eval:n { atan ( #4 , #1 ) } }
29238    }
29239    #5
29240  }
29241  \cs_new_protected:Npn \__driver_draw_cm_decompose_auxii:nnnnN #1#2#3#4#5
29242  {
29243    \use:x
29244    {
29245      \__driver_draw_cm_decompose_auxiii:nnnnN
29246      { \fp_eval:n { ( #4 - #3 ) / 2 } }
29247      { \fp_eval:n { ( #1 + #2 ) / 2 } }
29248      { \fp_eval:n { ( #1 - #2 ) / 2 } }
29249      { \fp_eval:n { ( #4 + #3 ) / 2 } }
29250    }

```

```

29251      #5
29252    }
29253 \cs_new_protected:Npn \__driver_draw_cm_decompose_auxiii:nnnnN #1#2#3#4#5
29254  {
29255    \fp_compare:nNnTF { abs( #2 ) } > { abs ( #3 ) }
29256    { #5 {#1} {#2} {#3} {#4} }
29257    { #5 {#1} {#3} {#2} {#4} }
29258  }
29259 
```

(End definition for `\__driver_draw_cm_decompose:nnnnN` and others.)

### \driver\_draw\_box\_use:Nnnnn

Inserting a TeX box transformed to the requested position and using the current matrix is done using a mixture of TeX and low-level manipulation. The offset can be handled by TeX, so only any rotation/skew/scaling component needs to be done using the matrix operation. As this operation can never be cached, the scope is set directly not using the `draw` version.

```

29260 \cs_new_protected:Npn \driver_draw_box_use:Nnnnn #1#2#3#4#5
29261  {
29262    \__driver_scope_begin:
29263    (*pdfmode)
29264      \driver_draw_cm:nnnn {#2} {#3} {#4} {#5}
29265    
```

```

29266    (*dvipdfmx | xdvipdfmx)
29267      \__driver_literal:x
29268    {
29269      pdf:btrans~matrix~
29270      \fp_eval:n {#2} ~ \fp_eval:n {#3} ~
29271      \fp_eval:n {#4} ~ \fp_eval:n {#5} ~
29272      0 ~ 0
29273    }
29274 
```

(End definition for `\driver_draw_box_use:Nnnnn`. This function is documented on page 266.)

```

29281 
```

## 45.7 dvisvgm driver

```

29282 (*dvisvgm)
```

### 45.7.1 Basics

```

\__driver_literal_svg:n
\__driver_literal_svg:x
```

Unlike the other drivers, the requirements for making SVG files mean that we can't conveniently transform all operations to the current point. That makes life a bit more tricky later as that needs to be accounted for. A new line is added after each call to help to keep the output readable for debugging.

```

29283 \cs_new_protected:Npn \__driver_literal_svg:n #1
29284  { \__driver_literal:n { dvisvgm:raw~ #1 { ?nl } } }
29285 \cs_generate_variant:Nn \__driver_literal_svg:n { x }
```

(End definition for `\_driver_literal_svg:n`.)

`\_driver_scope_begin:` A scope in SVG terms is slightly different to the other drivers as operations have to be “tied” to these not simply inside them.

```
29286 \cs_new_protected:Npn \_driver_scope_begin:
29287   { \_driver_literal_svg:n { <g> } }
29288 \cs_new_protected:Npn \_driver_scope_end:
29289   { \_driver_literal_svg:n { </g> } }
```

(End definition for `\_driver_scope_begin:` and `\_driver_scope_end:.`)

#### 45.7.2 Driver-specific auxiliaries

`\_driver_scope_begin:n` In SVG transformations, clips and so on are attached directly to scopes so we need a way or allowing for that. This is rather more useful than `\_driver_scope_begin:` as a result. No assumptions are made about the nature of the scoped operation(s).

```
29290 \cs_new_protected:Npn \_driver_scope_begin:n #
29291   { \_driver_literal_svg:n { <g~ #1 > } }
29292 \cs_generate_variant:Nn \_driver_scope_begin:n { x }
```

(End definition for `\_driver_scope_begin:n`.)

#### 45.7.3 Box operations

`\driver_box_use_clip:N` Clipping in SVG is more involved than with other drivers. The first issue is that the clipping path must be defined separately from where it is used, so we need to track how many paths have applied. The naming here uses `13cp` as the namespace with a number following. Rather than use a rectangular operation, we define the path manually as this allows it to have a depth: easier than the alternative approach of shifting content up and down using scopes to allow for the depth of the TeX box and keep the reference point the same!

```
29293 \cs_new_protected:Npn \driver_box_use_clip:N #1
29294   {
29295     \int_gincr:N \g_driver_clip_path_int
29296     \_driver_literal_svg:x
29297     { < clipPath~id = " 13cp \int_use:N \g_driver_clip_path_int " > }
29298     \_driver_literal_svg:x
29299     {
29300       <
29301         path ~ d =
29302           "
29303             M ~ O ~
29304               \dim_to_decimal:n { -\box_dp:N #1 } ~
29305             L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
29306               \dim_to_decimal:n { -\box_dp:N #1 } ~
29307             L ~ \dim_to_decimal:n { \box_wd:N #1 } ~
29308               \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
29309             L ~ O ~
29310               \dim_to_decimal:n { \box_ht:N #1 + \box_dp:N #1 } ~
29311             Z
29312           "
29313     />
29314 }
```

```

29315     \__driver_literal_svg:n
29316     { < /clipPath > }

```

In general the SVG set up does not try to transform coordinates to the current point. For clipping we need to do that, so have a transformation here to get us to the right place, and a matching one just before the TeX box is inserted to get things back on track. The clip path needs to come between those two such that if lines up with the current point, as does the TeX box.

```

29317     \__driver_scope_begin:n
29318     {
29319         transform =
29320         "
29321             translate ( { ?x } , { ?y } ) ~
29322                 scale ( 1 , -1 )
29323             "
29324     }
29325     \__driver_scope_begin:x
29326     {
29327         clip-path =
29328             "url ( \c_hash_str 13cp \int_use:N \g__driver_clip_path_int ) "
29329     }
29330     \__driver_scope_begin:n
29331     {
29332         transform =
29333         "
29334             scale ( -1 , 1 ) ~
29335                 translate ( { ?x } , { ?y } ) ~
29336                     scale ( -1 , -1 )
29337             "
29338     }
29339     \box_use:N #1
29340     \__driver_scope_end:
29341     \__driver_scope_end:
29342     \__driver_scope_end:
29343 %     \skip_horizontal:n { \box_wd:N #1 }
29344     }
29345 \int_new:N \g__driver_clip_path_int

```

(End definition for \driver\_box\_use\_clip:N and \g\_\_driver\_clip\_path\_int. This function is documented on page 261.)

**\driver\_box\_use\_rotate:Nn** Rotation has a dedicated operation which includes a centre-of-rotation optional pair. That can be picked up from the driver syntax, so there is no need to worry about the transformation matrix.

```

29346 \cs_new_protected:Npn \driver_box_use_rotate:Nn #1#2
29347     {
29348         \__driver_scope_begin:x
29349         {
29350             transform =
29351             "
29352                 rotate
29353                     ( \fp_eval:n { round ( -(#2) , 5 ) } , ~ { ?x } , ~ { ?y } )
29354             "
29355     }

```

```

29356      \box_use:N #1
29357      \__driver_scope_end:
29358  }

```

(End definition for `\driver_box_use_rotate:Nn`. This function is documented on page 262.)

### `\driver_box_use_scale:Nnn`

In contrast to rotation, we have to account for the current position in this case. That is done using a couple of translations in addition to the scaling (which is therefore done backward with a flip).

```

29359  \cs_new_protected:Npn \driver_box_use_scale:Nnn #1#2#3
29360  {
29361      \__driver_scope_begin:x
29362      {
29363          transform =
29364          "
29365          translate ( { ?x } , { ?y } ) ~
29366          scale
29367          (
29368              \fp_eval:n { round ( -#2 , 5 ) } ,
29369              \fp_eval:n { round ( -#3 , 5 ) }
29370          ) ~
29371          translate ( { ?x } , { ?y } ) ~
29372          scale ( -1 )
29373          "
29374      }
29375      \hbox_overlap_right:n { \box_use:N #1 }
29376      \__driver_scope_end:
29377  }

```

(End definition for `\driver_box_use_scale:Nnn`. This function is documented on page 262.)

#### 45.7.4 Images

These can be included by extracting the bounding box data.

```

29378  \cs_new_eq:NN \__driver_image_getbb_png:n \image_extract_bb:n
29379  \cs_new_eq:NN \__driver_image_getbb_jpg:n \image_extract_bb:n

```

(End definition for `\__driver_image_getbb_png:n` and `\__driver_image_getbb_jpg:n`.)

`\_driver_image_include_png:n`

`\_driver_image_include_jpg:n`

`\_driver_image_include_bitmap_quote:w`

The driver here has built-in support for basic image inclusion (see `dvisvgm.def` for a more complex approach, needed if clipping, *etc.*, is covered at the image driver level). The only issue is that #1 must be quote-corrected. The `dvisvgm:img` operation quotes the file name, but if it is already quoted (contains spaces) then we have an issue: we simply strip off any quotes as a result.

```

29380  \cs_new_protected:Npn \__driver_image_include_png:n #1
29381  {
29382      \__driver_literal:x
29383      {
29384          dvisvgm:img~
29385          \dim_to_decimal:n { \l_image_ury_dim } ~
29386          \dim_to_decimal:n { \l_image_ury_dim } ~
29387          \__driver_image_include_bitmap_quote:w #1 " " \q_stop
29388      }
29389  }

```

```

29390 \cs_new_eq:NN \__driver_image_include_jpg:n \__driver_image_include_png:n
29391 \cs_new:Npn \__driver_image_include_bitmap_quote:w #1 " #2 " #3 \q_stop
29392 { #1#2 }

(End definition for \__driver_image_include_png:n, \__driver_image_include_jpg:n, and \__driver-
image_include_bitmap_quote:w.)

```

#### 45.7.5 PDF Features

\driver\_pdf\_object\_new:n These are all no-ops.

```

29393 \cs_new_protected:Npn \driver_pdf_object_new:n #1 { }
29394 \cs_new:Npn \driver_pdf_object_ref:n #1 { }
29395 \cs_new_protected:Npn \driver_pdf_object_write:nn #1#2 { }
29396 \cs_new_protected:Npn \driver_pdf_compresslevel:n #1 { }
29397 \cs_new_protected:Npn \driver_pdf_objects_enable: { }
29398 \cs_new_protected:Npn \driver_pdf_objects_disable: { }

```

(End definition for \driver\_pdf\_object\_new:n and others. These functions are documented on page ??.)

#### 45.7.6 Drawing

\\_\_driver\_draw\_literal:n The same as the more general literal call.

```

29399 \cs_new_eq:NN \__driver_draw_literal:n \__driver_literal_svg:n
29400 \cs_generate_variant:Nn \__driver_draw_literal:n { x }

```

(End definition for \\_\_driver\_draw\_literal:n.)

\driver\_draw\_begin: A drawing needs to be set up such that the co-ordinate system is translated. That is done inside a scope, which as described below

```

29401 \cs_new_protected:Npn \driver_draw_begin:
29402 {
29403     \driver_draw_scope_begin:
29404     \__driver_draw_scope:n { transform="translate({?x},{?y})~scale(1,-1)" }
29405 }
29406 \cs_new_protected:Npn \driver_draw_end:
29407 { \driver_draw_scope_end: }

```

(End definition for \driver\_draw\_begin: and \driver\_draw\_end:. These functions are documented on page 263.)

\driver\_draw\_scope\_begin: Several settings that with other drivers are “stand alone” have to be given as part of a scope in SVG. As a result, there is a need to provide a mechanism to automatically close these extra scopes. That is done using a dedicated function and a pair of tracking variables. Within each graphics scope we use a global variable to do the work, with a group used to save the value between scopes. The result is that no direct action is needed when creating a scope.

```

29408 \cs_new_protected:Npn \driver_draw_scope_begin:
29409 {
29410     \int_set_eq:NN
29411     \l_driver_draw_scope_int
29412     \g_driver_draw_scope_int
29413     \group_begin:
29414         \int_gzero:N \g_driver_draw_scope_int

```

```

29415     }
29416 \cs_new_protected:Npn \driver_draw_scope_end:
29417 {
29418     \prg_replicate:nn
29419         { \g_driver_draw_scope_int }
29420         { \__driver_draw_literal:n { </g> } }
29421     \group_end:
29422     \int_gset_eq:NN
29423         \g_driver_draw_scope_int
29424         \l_driver_draw_scope_int
29425 }
29426 \cs_new_protected:Npn \__driver_draw_scope:n #1
29427 {
29428     \__driver_draw_literal:n { <g~ #1 > }
29429     \int_gincr:N \g_driver_draw_scope_int
29430 }
29431 \cs_generate_variant:Nn \__driver_draw_scope:n { x }
29432 \int_new:N \g_driver_draw_scope_int
29433 \int_new:N \l_driver_draw_scope_int

```

(End definition for `\driver_draw_scope_begin:` and others. These functions are documented on page 263.)

`\driver_draw_moveto:nn`  
`\driver_draw_lineto:nn`  
`\driver_draw_rectangle:nnnn`  
`\driver_draw_curveto:nnnnnn`  
`\__driver_draw_add_to_path:n`

Once again, some work is needed to get path constructs correct. Rather than write the values as they are given, the entire path needs to be collected up before being output in one go. For that we use a dedicated storage routine, which adds spaces as required. Since paths should be fully expanded there is no need to worry about the internal x-type expansion.

```

29434 \cs_new_protected:Npn \driver_draw_moveto:nn #1#2
29435 {
29436     \__driver_draw_add_to_path:n
29437         { M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
29438 }
29439 \cs_new_protected:Npn \driver_draw_lineto:nn #1#2
29440 {
29441     \__driver_draw_add_to_path:n
29442         { L ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} }
29443 }
29444 \cs_new_protected:Npn \driver_draw_rectangle:nnnn #1#2#3#4
29445 {
29446     \__driver_draw_add_to_path:n
29447         {
29448             M ~ \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2}
29449             h ~ \dim_to_decimal:n {#3} ~
29450             v ~ \dim_to_decimal:n {#4} ~
29451             h ~ \dim_to_decimal:n { -#3 } ~
29452             Z
29453         }
29454     }
29455 \cs_new_protected:Npn \driver_draw_curveto:nnnnnn #1#2#3#4#5#6
29456 {
29457     \__driver_draw_add_to_path:n
29458         {
29459             C ~

```

```

29460     \dim_to_decimal:n {#1} ~ \dim_to_decimal:n {#2} ~
29461     \dim_to_decimal:n {#3} ~ \dim_to_decimal:n {#4} ~
29462     \dim_to_decimal:n {#5} ~ \dim_to_decimal:n {#6}
29463   }
29464 }
29465 \cs_new_protected:Npn \__driver_draw_add_to_path:n #1
29466 {
29467   \tl_gset:Nx \g__driver_draw_path_tl
29468   {
29469     \g__driver_draw_path_tl
29470     \tl_if_empty:NF \g__driver_draw_path_tl { \c_space_tl }
29471     #1
29472   }
29473 }
29474 \tl_new:N \g__driver_draw_path_tl

```

(End definition for `\driver_draw_moveto:nn` and others. These functions are documented on page 263.)

`\driver_draw_evenodd_rule:` The fill rules here have to be handled as scopes.

`\driver_draw_nonzero_rule:`

```

29475 \cs_new_protected:Npn \driver_draw_evenodd_rule:
29476   { \__driver_draw_scope:n { fill-rule="evenodd" } }
29477 \cs_new_protected:Npn \driver_draw_nonzero_rule:
29478   { \__driver_draw_scope:n { fill-rule="nonzero" } }

```

(End definition for `\driver_draw_evenodd_rule:` and `\driver_draw_nonzero_rule:`. These functions are documented on page 264.)

`\__driver_draw_path:n` Setting fill and stroke effects and doing clipping all has to be done using scopes. This means setting up the various requirements in a shared auxiliary which deals with the bits and pieces. Clipping paths are reused for path drawing; not essential but avoids constructing them twice. Discarding a path needs a separate function as it's not quite the same.

```

29479 \cs_new_protected:Npn \driver_draw_closepath:
29480   { \__driver_draw_add_to_path:n { Z } }
29481 \cs_new_protected:Npn \__driver_draw_path:n #1
29482 {
29483   \bool_if:NTF \g__driver_draw_clip_bool
29484   {
29485     \int_gincr:N \g__driver_clip_path_int
29486     \__driver_draw_literal:x
29487     {
29488       < clipPath~id = " 13cp \int_use:N \g__driver_clip_path_int " >
29489       { ?nl }
29490       <path~d=" \g__driver_draw_path_tl "/> { ?nl }
29491       < /clipPath > { ? nl }
29492       <
29493         use~xlink:href =
29494           "\c_hash_str 13path \int_use:N \g__driver_path_int " ~
29495           #1
29496       />
29497     }
29498   \__driver_draw_scope:x
29499   {
29500     clip-path =

```

```

29501           "url( \c_hash_str 13cp \int_use:N \g_driver_clip_path_int)"
29502       }
29503   }
29504   {
29505       \__driver_draw_literal:x
29506           { <path ~ d=" \g_driver_draw_path_tl " ~ #1 /> }
29507   }
29508   \tl_gclear:N \g_driver_draw_path_tl
29509   \bool_gset_false:N \g_driver_draw_clip_bool
29510   }
29511 \int_new:N \g_driver_path_int
29512 \cs_new_protected:Npn \driver_draw_stroke:
29513     { \__driver_draw_path:n { style="fill:none" } }
29514 \cs_new_protected:Npn \driver_draw_closesstroke:
29515     {
29516         \driver_draw_closepath:
29517         \driver_draw_stroke:
29518     }
29519 \cs_new_protected:Npn \driver_draw_fill:
29520     { \__driver_draw_path:n { style="stroke:none" } }
29521 \cs_new_protected:Npn \driver_draw_fillstroke:
29522     { \__driver_draw_path:n { } }
29523 \cs_new_protected:Npn \driver_draw_clip:
29524     { \bool_gset_true:N \g_driver_draw_clip_bool }
29525 \bool_new:N \g_driver_draw_clip_bool
29526 \cs_new_protected:Npn \driver_draw_discardpath:
29527     {
29528         \bool_if:NT \g_driver_draw_clip_bool
29529     }
29530         \int_gincr:N \g_driver_clip_path_int
29531         \__driver_draw_literal:x
29532         {
29533             < clipPath~id = " 13cp \int_use:N \g_driver_clip_path_int " >
29534                 { ?nl }
29535                 <path~d=" \g_driver_draw_path_tl " /> { ?nl }
29536                 < /clipPath >
29537             }
29538         \__driver_draw_scope:x
29539         {
29540             clip-path =
29541                 "url( \c_hash_str 13cp \int_use:N \g_driver_clip_path_int)"
29542         }
29543     }
29544     \tl_gclear:N \g_driver_draw_path_tl
29545     \bool_gset_false:N \g_driver_draw_clip_bool
29546 }

```

(End definition for `\__driver_draw_path:n` and others. These functions are documented on page [263](#).)

```

\driver_draw_dash_pattern:nn
    \__driver_draw_dash:n
\__driver_draw_dash_aux:nn
\driver_draw_linewidth:n
\driver_draw_miterlimit:n
\driver_draw_cap_buttt:
\driver_draw_cap_round:
\driver_draw_cap_rectangle:
\driver_draw_join_miter:
\driver_draw_join_round:
\driver_draw_join_bevel:

```

All of these ideas are properties of scopes in SVG. The only slight complexity is converting the dash array properly (doing any required maths).

```

29547 \cs_new_protected:Npn \driver_draw_dash_pattern:nn #1#2
29548 {
29549     \use:x

```

```

29550   {
29551     \__driver_draw_dash_aux:nn
29552     { \clist_map_function:nn {#1} \__driver_draw_dash:n }
29553     { \dim_to_decimal:n {#2} }
29554   }
29555   }
29556 \cs_new:Npn \__driver_draw_dash:n #1
29557   { , \dim_to_decimal_in_bp:n {#1} }
29558 \cs_new_protected:Npn \__driver_draw_dash_aux:nn #1#2
29559   {
29560     \__driver_draw_scope:x
29561     {
29562       stroke-dasharray =
29563         "
29564         \tl_if_empty:oTF { \use_none:n #1 }
29565           { none }
29566           { \use_none:n #1 }
29567         " ~
29568         stroke-offset=" #2 "
29569     }
29570   }
29571 \cs_new_protected:Npn \driver_draw_linewidth:n #1
29572   { \__driver_draw_scope:x { stroke-width=" \dim_to_decimal:n {#1} " } }
29573 \cs_new_protected:Npn \driver_draw_miterlimit:n #1
29574   { \__driver_draw_scope:x { stroke-miterlimit=" \fp_eval:n {#1} " } }
29575 \cs_new_protected:Npn \driver_draw_cap_butt:
29576   { \__driver_draw_scope:n { stroke-linecap="butt" } }
29577 \cs_new_protected:Npn \driver_draw_cap_round:
29578   { \__driver_draw_scope:n { stroke-linecap="round" } }
29579 \cs_new_protected:Npn \driver_draw_cap_rectangle:
29580   { \__driver_draw_scope:n { stroke-linecap="square" } }
29581 \cs_new_protected:Npn \driver_draw_join_miter:
29582   { \__driver_draw_scope:n { stroke-linejoin="miter" } }
29583 \cs_new_protected:Npn \driver_draw_join_round:
29584   { \__driver_draw_scope:n { stroke-linejoin="round" } }
29585 \cs_new_protected:Npn \driver_draw_join_bevel:
29586   { \__driver_draw_scope:n { stroke-linejoin="bevel" } }

```

(End definition for \driver\_draw\_dash\_pattern:nn and others. These functions are documented on page 265.)

\driver\_draw\_color\_fill\_cmyk:nnnn  
\driver\_draw\_color\_stroke\_cmyk:nnnn  
SVG fill color has to be covered outside of the stack, as for dvips. Here, we are only allowed RGB colors so there is some conversion to do.

```

29587 \cs_new_protected:Npn \driver_draw_color_stroke_cmyk:nnnn #1#2#3#4
29588   {
29589     \use:x
29590     {
29591       \__driver_draw_color_fill:nnn
29592         { \fp_eval:n { -100 * ( (#1) * ( 1 - (#4) ) - 1 ) } }
29593         { \fp_eval:n { -100 * ( (#2) * ( 1 - (#4) ) + #4 - 1 ) } }
29594         { \fp_eval:n { -100 * ( (#3) * ( 1 - (#4) ) + #4 - 1 ) } }
29595     }
29596   }
29597 \cs_new_eq:NN \driver_draw_color_stroke_cmyk:nnnn \driver_color_cmyk:nnnn

```

```

29598 \cs_new_protected:Npn \driver_draw_color_gray:n #1
29599   {
29600     \use:x
29601     {
29602       \__driver_draw_color_gray_aux:n
29603         { \fp_eval:n { 100 * (#3) } }
29604     }
29605   }
29606 \cs_new_protected:Npn \__driver_draw_color_gray_aux:n #1
29607   { \__driver_draw_color_fill:nnn {#1} {#1} {#1} }
29608 \cs_new_eq:NN \driver_draw_color_stroke_gray:n \driver_color_gray:n
29609 \cs_new_protected:Npn \driver_draw_color_rgb:nnn #1#2#3
29610   {
29611     \use:x
29612     {
29613       \__driver_draw_color_fill:nnn
29614         { \fp_eval:n { 100 * (#1) } }
29615         { \fp_eval:n { 100 * (#2) } }
29616         { \fp_eval:n { 100 * (#3) } }
29617     }
29618   }
29619 \cs_new_protected:Npn \__driver_draw_color_fill:nnn #1#2#3
29620   {
29621     \__driver_draw_scope:x
29622     {
29623       fill =
29624         "
29625         rgb
29626         (
29627           #1 \c_percent_str ,
29628           #2 \c_percent_str ,
29629           #3 \c_percent_str
29630         )
29631         "
29632     }
29633   }
29634 \cs_new_eq:NN \driver_draw_color_stroke_rgb:nnn \driver_color_rgb:nnn

```

(End definition for `\driver_draw_color_fill_cmyk:nnnn` and others. These functions are documented on page [265](#).)

### `\driver_draw_cm:nnnn`

The four arguments here are floats (the affine matrix), the last two are a displacement vector.

```

29635 \cs_new_protected:Npn \driver_draw_cm:nnnn #1#2#3#4
29636   {
29637     \__driver_draw_scope:n
29638     {
29639       transform =
29640         "
29641         matrix
29642         (
29643           \fp_eval:n {#1} , \fp_eval:n {#2} ,
29644           \fp_eval:n {#3} , \fp_eval:n {#4} ,
29645           Opt , Opt

```

```

29646      )
29647      "
29648    }
29649  }

```

(End definition for `\driver_draw_cm:nnnn`. This function is documented on page 266.)

`\driver_draw_box_use:Nnnnn` No special savings can be made here: simply displace the box inside a scope. As there is nothing to re-box, just make the box passed of zero size.

```

29650 \cs_new_protected:Npn \driver_draw_box_use:Nnnnn #1#2#3#4#5#6#7
29651 {
29652   \__driver_scope_begin:
29653   \driver_draw_cm:nnnn {#2} {#3} {#4} {#5}
29654   \__driver_literal_svg:n
29655   {
29656     < g~
29657       stroke="none"-
29658       transform="scale(-1,1)~translate({?x},{?y})~scale(-1,-1)"
29659     >
29660   }
29661   \box_set_wd:Nn #1 { 0pt }
29662   \box_set_ht:Nn #1 { 0pt }
29663   \box_set_dp:Nn #1 { 0pt }
29664   \box_use:N #1
29665   \__driver_literal_svg:n { </g> }
29666   \__driver_scope_end:
29667 }

```

(End definition for `\driver_draw_box_use:Nnnnn`. This function is documented on page 266.)

```

29668 </dvisvgm>
29669 </initex | package>

```

## 46 I3deprecation implementation

```

29670 <*initex | package>
29671 <@=deprecation>

```

`\__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.

```

29672 \cs_new_protected:Npn \__kernel_deprecation_error:Nnn #1#2#3
29673 {
29674   \tex_protected:D \tex_outer:D \tex_edef:D #1
29675   {
29676     \exp_not:N \__kernel_msg_expandable_error:nnnnn
29677     { kernel } { deprecated-command }
29678     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
29679     \exp_not:N \__kernel_msg_error:nnxxx
29680     { kernel } { deprecated-command }
29681     { \tl_to_str:n {#3} } { \token_to_str:N #1 } { \tl_to_str:n {#2} }
29682   }
29683 }
29684 \__kernel_deprecation_error:Nnn \box_resize:cnn

```

```

29685 { \box_resize_to_wd_and_ht_plus_dp:cnn } { 2018-12-27 }
29686 \__kernel_deprecation_error:Nnn \box_resize:Nnn
29687 { \box_resize_to_wd_and_ht_plus_dp:Nnn } { 2018-12-27 }
29688 \__kernel_deprecation_error:Nnn \box_use_clear:c
29689 { \box_use_drop:c } { 2018-12-27 }
29690 \__kernel_deprecation_error:Nnn \box_use_clear:N
29691 { \box_use_drop:N } { 2018-12-27 }
29692 \__kernel_deprecation_error:Nnn \c_job_name_tl
29693 { \c_sys_jobname_str } { 2017-01-01 }
29694 \__kernel_deprecation_error:Nnn \c_minus_one
29695 { -1 } { 2018-12-27 }
29696 \__kernel_deprecation_error:Nnn \dim_case:nnn
29697 { \dim_case:nnF } { 2015-07-14 }
29698 \__kernel_deprecation_error:Nnn \file_add_path:nN
29699 { \file_get_full_name:nN } { 2018-12-27 }
29700 \__kernel_deprecation_error:Nnn \file_if_exist_input:nT
29701 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
29702 \__kernel_deprecation_error:Nnn \file_if_exist_input:nTF
29703 { \file_if_exist:nT and~ \file_input:n } { 2018-03-05 }
29704 \__kernel_deprecation_error:Nnn \file_list:
29705 { \file_log_list: } { 2018-12-27 }
29706 \__kernel_deprecation_error:Nnn \file_path_include:n
29707 { \seq_put_right:Nn \l_file_search_path_seq } { 2018-12-27 }
29708 \__kernel_deprecation_error:Nnn \file_path_remove:n
29709 { \seq_remove_all:Nn \l_file_search_path_seq } { 2018-12-27 }
29710 \__kernel_deprecation_error:Nnn \g_file_current_name_tl
29711 { \g_file_current_name_str } { 2018-12-27 }
29712 \__kernel_deprecation_error:Nnn \int_case:nnn
29713 { \int_case:nnF } { 2015-07-14 }
29714 \__kernel_deprecation_error:Nnn \int_from_binary:n
29715 { \int_from_bin:n } { 2016-01-05 }
29716 \__kernel_deprecation_error:Nnn \int_from_hexadecimal:n
29717 { \int_from_hex:n } { 2016-01-05 }
29718 \__kernel_deprecation_error:Nnn \int_from_octal:n
29719 { \int_from_oct:n } { 2016-01-05 }
29720 \__kernel_deprecation_error:Nnn \int_to_binary:n
29721 { \int_to_bin:n } { 2016-01-05 }
29722 \__kernel_deprecation_error:Nnn \int_to_hexadecimal:n
29723 { \int_to_hex:n } { 2016-01-05 }
29724 \__kernel_deprecation_error:Nnn \int_to_octal:n
29725 { \int_to_oct:n } { 2016-01-05 }
29726 \__kernel_deprecation_error:Nnn \ior_get_str:NN
29727 { \ior_str_get:NN } { 2018-03-05 }
29728 \__kernel_deprecation_error:Nnn \ior_list_streams:
29729 { \ior_show_list: } { 2018-12-27 }
29730 \__kernel_deprecation_error:Nnn \ior_log_streams:
29731 { \ior_log_list: } { 2018-12-27 }
29732 \__kernel_deprecation_error:Nnn \luatex_if_engine_p:
29733 { \sys_if_engine_luatex_p: } { 2017-01-01 }
29734 \__kernel_deprecation_error:Nnn \luatex_if_engine:F
29735 { \sys_if_engine_luatex:F } { 2017-01-01 }
29736 \__kernel_deprecation_error:Nnn \luatex_if_engine:T
29737 { \sys_if_engine_luatex:T } { 2017-01-01 }
29738 \__kernel_deprecation_error:Nnn \luatex_if_engine:TF

```

```

29739   { \sys_if_engine_luatex:TF } { 2017-01-01 }
29740 \__kernel_deprecation_error:Nnn \pdftex_if_engine_p:
29741   { \sys_if_engine_pdftex_p: } { 2017-01-01 }
29742 \__kernel_deprecation_error:Nnn \pdftex_if_engine:F
29743   { \sys_if_engine_pdftex:F } { 2017-01-01 }
29744 \__kernel_deprecation_error:Nnn \pdftex_if_engine:T
29745   { \sys_if_engine_pdftex:T } { 2017-01-01 }
29746 \__kernel_deprecation_error:Nnn \pdftex_if_engine:TF
29747   { \sys_if_engine_pdftex:TF } { 2017-01-01 }
29748 \__kernel_deprecation_error:Nnn \prop_get:cn
29749   { \prop_item:cn } { 2016-01-05 }
29750 \__kernel_deprecation_error:Nnn \prop_get:Nn
29751   { \prop_item:Nn } { 2016-01-05 }
29752 \__kernel_deprecation_error:Nnn \quark_if_recursion_tail_break:N
29753   { } { 2015-07-14 }
29754 \__kernel_deprecation_error:Nnn \quark_if_recursion_tail_break:n
29755   { } { 2015-07-14 }
29756 \__kernel_deprecation_error:Nnn \scan_align_safe_stop:
29757   { protected~commands } { 2017-01-01 }
29758 \__kernel_deprecation_error:Nnn \sort_ordered:
29759   { \sort_return_same: } { 2018-12-27 }
29760 \__kernel_deprecation_error:Nnn \sort_reversed:
29761   { \sort_return_swapped: } { 2018-12-27 }
29762 \__kernel_deprecation_error:Nnn \str_case:nnn
29763   { \str_case:nnF } { 2015-07-14 }
29764 \__kernel_deprecation_error:Nnn \str_case:onn
29765   { \str_case:onF } { 2015-07-14 }
29766 \__kernel_deprecation_error:Nnn \str_case_x:nnn
29767   { \str_case_e:nnF } { 2015-07-14 }
29768 \__kernel_deprecation_error:Nnn \tl_case:cnn
29769   { \tl_case:cnF } { 2015-07-14 }
29770 \__kernel_deprecation_error:Nnn \tl_case:Nnn
29771   { \tl_case:NnF } { 2015-07-14 }
29772 \__kernel_deprecation_error:Nnn \tl_to_lowercase:n
29773   { \tex_lowercase:D } { 2018-03-05 }
29774 \__kernel_deprecation_error:Nnn \tl_to_uppercase:n
29775   { \tex_uppercase:D } { 2018-03-05 }
29776 \__kernel_deprecation_error:Nnn \token_new:Nn
29777   { \cs_new_eq:NN } { 2018-12-29 }
29778 \__kernel_deprecation_error:Nnn \xetex_if_engine_p:
29779   { \sys_if_engine_xetex_p: } { 2017-01-01 }
29780 \__kernel_deprecation_error:Nnn \xetex_if_engine:F
29781   { \sys_if_engine_xetex:F } { 2017-01-01 }
29782 \__kernel_deprecation_error:Nnn \xetex_if_engine:T
29783   { \sys_if_engine_xetex:T } { 2017-01-01 }
29784 \__kernel_deprecation_error:Nnn \xetex_if_engine:TF
29785   { \sys_if_engine_xetex:TF } { 2017-01-01 }

```

(End definition for `\__kernel_deprecation_error:Nnn`.)

```
\etex_beginL:D  
\__deprecation_primitive:NN  
\__deprecation_primitive:w
```

We renamed all primitives to `\tex_...:D` so all others are deprecated. In l3names, `\__kernel_primitives:` is defined to contain `\__kernel_primitive:NN \beginL \etex_beginL:D` and so on, one for each deprecated primitive. We apply `\exp_not:N` to the second argument of `\__kernel_primitive:NN` because it may be outer (both when doing

and undoing deprecation actually), then `\_deprecation_primitive:NN` uses `\tex_-let:D` to change the meaning of this potentially outer token. Then, either turn it into an error or make it equal to the primitive #1. To be more precise, #1 may not be defined, so try a `\tex_...:D` command as well.

```

29786 \cs_new_protected:Npn \_deprecation_primitive:NN #1#2 { }
29787 \exp_last_unbraced:NNNNo
29788   \cs_new:Npn \_deprecation_primitive:w #1 { \token_to_str:N _ } { }
29789 \_kernel_deprecation_code:nn
29790   {
29791     \cs_set_protected:Npn \_kernel_primitive:NN #1
29792       {
29793         \exp_after:wN \_deprecation_primitive:NN
29794         \exp_after:wN #1
29795         \exp_not:N
29796       }
29797     \cs_set_protected:Npn \_deprecation_primitive:NN #1#2
29798   {
29799     \tex_let:D #2 \scan_stop:
29800     \exp_args:NNx \_kernel_deprecation_error:Nnn #2
29801       {
29802         \iow_char:N \\
29803         \cs_if_exist:NTF #1
29804           { \cs_to_str:N #1 }
29805           {
29806             tex_
29807             \exp_last_unbraced:Nf
29808             \_deprecation_primitive:w { \cs_to_str:N #2 }
29809           }
29810         }
29811       { 2019-12-31 }
29812     }
29813   \_kernel_primitives:
29814 }
29815 {
29816   \cs_set_protected:Npn \_kernel_primitive:NN #1
29817   {
29818     \exp_after:wN \_deprecation_primitive:NN
29819     \exp_after:wN #1
29820     \exp_not:N
29821   }
29822   \cs_set_protected:Npn \_deprecation_primitive:NN #1#2
29823   {
29824     \tex_let:D #2 #1
29825     \cs_if_exist:cT { tex_ \cs_to_str:N #1 :D }
29826       {
29827         \exp_args:NNc \cs_set_eq:NN #2
29828           { tex_ \cs_to_str:N #1 :D }
29829       }
29830     }
29831   \_kernel_primitives:
29832 }

```

(End definition for `\etex_beginL:D`, `\_deprecation_primitive:NN`, and `\_deprecation_primitive:w`.)  
29833 `</initex | 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	
!	197
\!	10343
\"	8439, 8442, 27699
\#	6, 5679, 8439, 10759
\\$	5678, 8439, 8442, 21262
\%	5680, 8439, 10761
\&	5671, 8439, 8442, 9489, 10343, 10344
\&&	196
\'	27699
\(	27742
\)	27742
*	197
\*	5364, 5387, 8611, 8613, 8617, 8625
**	197
+	196, 197
\,	12002
-	196, 197
\-	292
\.	27699
/	197
\/	291
\:	5677
\:::	34, 349, 373, 374, 3153, 3154, 3155, 3156, 3157, 3158, 3159, 3161, 3165, 3166, 3169, 3172, 3179, 3182, 3185, 3191, 3278, 3344, 3345, 3346, 3347, 3348, 3349, 3350, 3351, 3352, 3353, 3354, 3355, 3356, 3357, 3358, 3359, 3360, 3361, 3362, 3363, 3364, 3365, 3366, 3367, 3368, 3369, 3370, 3371, 3372, 3373, 3374, 3375, 3376, 3377, 3378, 3379, 3380, 3381, 3382, 3383, 3384, 3385, 3386, 3388, 3390, 3395, 3402, 3406, 3409, 3414, 3429, 3470, 3471, 3472, 3473, 3484
\:::N	34, <u>3157</u> , 3278, 3357, 3363, 3364, 3365, 3366, 3367, 3379, 3380, 3381, 3473
\:::V	34, 3185, 3347, 3354, 3365, 3367, 3369, 3371
\:::V_unbraced	34, 3387
\:::c	34, 3159, 3345, 3351, 3358, 3363, 3368, 3369, 3370, 3371, 3372, 3384, 3385
\:::e	34, <u>3163</u> , 3278, 3349
\:::e_unbraced	34, <u>3387</u> , 3429
\:::error	248, 26201
\:::f	34, <u>3172</u> , 3350, 3353, 3355, 3356, 3363, 3374, 3375, 3377, 3378, 3472
\:::f_unbraced	34, <u>3387</u>
\:::n	34, <u>3156</u> , 3345, 3346, 3347, 3348, 3349, 3350, 3359, 3364, 3365, 3368, 3369, 3372, 3373, 3374, 3375, 3380, 3382, 3383, 3385, 3470, 3473
\:::o	34, <u>3161</u> , 3346, 3351, 3352, 3353, 3354, 3355, 3360, 3361, 3364, 3366, 3368, 3370, 3373, 3376, 3377, 3378, 3381, 3383, 3386, 3471
\:::o_unbraced	34, <u>3387</u> , 3470, 3471, 3472, 3473
\:::p	34, 349, <u>3158</u>
\:::v	34, <u>3185</u> , 3348
\:::v_unbraced	34, <u>3387</u>
\:::x	34, <u>3179</u> , 3344, 3357, 3358, 3359, 3360, 3361, 3362, 3379, 3380, 3381, 3382, 3383, 3384, 3385, 3386
\:::x_unbraced	34, <u>3387</u> , 3484
<	196
=	196
>	196
?	196
?:	196
\\"	2340, 3131, 5673, 8439, 9405, 9423, 9425, 9430, 9431, 9459, 9474, 9927, 9935, 9942, 9954, 9955, 9980, 9981, 9988, 10014, 10016, 10017, 10062, 10063, 10076, 10120, 10141, 10142, 10163, 10164, 10165, 10166, 10167, 10219, 10223, 10228, 10235, 10318, 10319, 10322, 10323, 10764, 11319, 11325, 11332, 12911, 12923, 12929, 13721, 13724, 13725, 13726, 13733, 13736, 13737, 19839, 19842, 19843, 19868, 19869, 19876, 19877, 20332, 21802, 21915, 21916, 21917, 21938, 23291, 23295, 23300, 23334, 23343, 23347, 23352, 23372, 23374, 23375, 23377, 23380, 23382, 23389, 23393, 23396, 23400, 23402, 23406, 23408, 23414, 23416, 23420, 23422, 23426, 23431, 23433, 23475, 23477, 23482, 23484, 23490, 23495, 23496, 23500, 23504, 23514, 23517, 23521, 23522, 23526, 23534, 23597, 25313, 29802

\{ . . . . .	4, 5674, 8439, 10164, 10341, 10758, 20579, 23295, 23300, 23347, 23396, 23433, 23522, 23526, 27754	
\} . . . . .	5, 5675, 8439, 10164, 10342, 10760, 23294, 23300, 23397, 23433, 23522, 23526, 27754	
\^ . . . . .	7, 10, 110, 201, 202, 203, 204, 2814, 3514, 5676, 8439, 8442, 8444, 8450, 8502, 9496, 10357, 10676, 10712, 17556, 19923, 19996, 19999, 20525, 20530, 20531, 20532, 20533, 20536, 20547, 20584, 20647, 20649, 20651, 20653, 20655, 20657, 21261, 22896, 22899, 22913, 22916, 22925, 22928, 22931, 22934, 22948, 22951, 27699	
\~ . . . . .	197	
\_ . . . . .	5682, 8439, 8442	
\` . . . . .	27699	
\  . . . . .	196	
\~ . . . . .	4293, 5681, 8439, 8442, 10762, 20569, 20573, 20579, 27699	
\u . . . . .	290, 2718, 5364, 5387, 8439, 8604, 9982, 10006, 10150, 10156, 10164, 10166, 10264, 10341, 10342, 10765, 20040, 20524, 20529, 20573, 20583, 20767, 22945, 26385, 27902	
 <b>A</b>		
\A . . . . .	5365, 5388	
\AA . . . . .	27681	
\aa . . . . .	27681	
\above . . . . .	293	
\abovedisplayshortskip . . . . .	294	
\abovedisplayskip . . . . .	295	
\abovewithdelims . . . . .	296	
\abs . . . . .	197	
\accent . . . . .	297	
\acos . . . . .	199	
\acosd . . . . .	199	
\acot . . . . .	200	
\acotd . . . . .	200	
\acsc . . . . .	199	
\acscl . . . . .	199	
\adjdemerits . . . . .	298	
\adjustspacing . . . . .	1012, 1621	
\advance . . . . .	175, 191, 299	
\AE . . . . .	27682	
\ae . . . . .	27682	
\afterassignment . . . . .	300	
\aftergroup . . . . .	301	
\alignmark . . . . .	889, 1713	
\aligntab . . . . .	890, 1714	
 <b>B</b>		
\badness . . . . .	304	
\baselineskip . . . . .	305	
\batchmode . . . . .	306	
\begin . . . . .	208, 212, 15642, 20328, 20330, 28675, 28680	
begin internal commands:		
__regex_begin . . . . .	23032	
\begincsname . . . . .	898, 1723	
\begingroup . . . . .	13, 20, 38, 42, 48, 73, 149, 169, 280, 307	
\beginL . . . . .	615, 1119, 1428	
\beginR . . . . .	616, 1429	
\belowdisplayshortskip . . . . .	308	
\belowdisplayskip . . . . .	309	
\binoppenalty . . . . .	310	
\bodydir . . . . .	899, 1812	
\bodydirection . . . . .	900	
bool commands:		
\bool_const:Nn . . . . .	248, 26226	
\bool_do_until:Nn . . . . .	101, 7576	
\bool_do_until:nn . . . . .	101, 7582	
\bool_do_while:Nn . . . . .	101, 7576	
\bool_do_while:nn . . . . .	102, 7582	
.bool_gset:N . . . . .	171, 12454	
\bool_gset:Nn . . . . .	98, 7404	
\bool_gset_eq:NN . . . . .	98, 7400, 20515, 22436	
\bool_gset_false:N . . . . .	98, 7384, 22395, 26237, 28303, 28319, 28345, 28367, 28383, 29081, 29509, 29545	
.bool_gset_inverse:N . . . . .	171, 12462	
\bool_gset_inverse:N . . . . .	249, 26233	
\bool_gset_true:N . . . . .	98, 7384, 22446, 26237, 28301, 28370, 29079, 29524	

\bool\_if:NTF . . . . . 98, 244, 3000,  
7418, 7571, 7573, 7577, 7579, 10979,  
10986, 12231, 12426, 12435, 12601,  
12626, 12640, 12657, 12691, 12719,  
12738, 12740, 12745, 12752, 12773,  
12827, 21395, 21404, 21806, 21884,  
21902, 21931, 22052, 22278, 22286,  
23564, 23569, 24755, 25195, 26234,  
26237, 28310, 28314, 28332, 28336,  
28340, 28353, 28358, 28362, 28374,  
28378, 28651, 28692, 28878, 28920,  
29092, 29097, 29102, 29483, 29528  
\bool\_if:nTF . . . . . 98, 100, 101, 102,  
102, 102, 1054, 1071, 7432, 7450,  
7521, 7528, 7547, 7554, 7563, 7584,  
7593, 7597, 7606, 7704, 21887, 26310  
\bool\_if\_exist:NTF . . . . .  
. . . . . 99, 7446, 12249, 12265  
\bool\_if\_exist\_p:N . . . . . 99, 7446  
\bool\_if\_p:N . . . . . 98, 7418  
\bool\_if\_p:n . . . . .  
. . . . . 100, 476, 7408, 7414, 7450,  
7458, 7528, 7554, 7560, 7564, 26230  
\bool\_lazy\_all:nTF . . . . .  
. . . . . 99, 100, 100, 7508, 21671  
\bool\_lazy\_all\_p:n . . . . . 100, 7508  
\bool\_lazy\_and:nnTF . . . . . 99, 100,  
7525, 7724, 10564, 27349, 27399, 27724  
\bool\_lazy\_and\_p:nn . . . . . 100, 100, 7525  
\bool\_lazy\_any:nTF . . . . .  
. . . . . 99, 100, 101, 7534, 27149  
\bool\_lazy\_any\_p:n . . . . .  
. . . . . 100, 100, 7534, 27352  
\bool\_lazy\_or:nnTF . . . . . 99, 100,  
101, 7551, 19929, 25542, 26767,  
27017, 27046, 27228, 27257, 27413,  
27455, 27495, 27702, 28684, 28913  
\bool\_lazy\_or\_p:nn . . . . . 101, 7551, 27727  
\bool\_log:N . . . . . 98, 7433  
\bool\_log:n . . . . . 98, 7427  
\bool\_new:N . . . . .  
. . . . . 97, 7382, 7442, 7443, 7444, 7445,  
10707, 12144, 12145, 12149, 12150,  
12154, 12249, 12265, 20373, 20802,  
22337, 22338, 22340, 22341, 22342,  
24356, 28304, 28371, 29082, 29525  
\bool\_not\_p:n . . . . . 101, 7560  
.bool\_set:N . . . . . 171, 12454  
\bool\_set:Nn . . . . . 98, 1031, 7404  
\bool\_set\_eq:NN 98, 7400, 20509, 22595  
\bool\_set\_false:N . . . . .  
. . . . . 98, 259, 7384, 10813,  
10955, 10963, 10971, 10981, 10988,  
12171, 12611, 12636, 12648, 12668,  
12677, 12726, 21369, 21574, 22316,  
22408, 22422, 22468, 22536, 24751,  
25193, 26234, 28661, 28852, 28991  
.bool\_set\_inverse:N . . . . . 171, 12462  
\bool\_set\_inverse:N . . . . . 249, 26233  
\bool\_set\_true:N . . . . .  
. . . . . 98, 273, 7384, 10941, 12166,  
12609, 12634, 12650, 12666, 12672,  
12733, 21374, 21578, 22310, 22534,  
22594, 24769, 24784, 24815, 26234  
\bool\_show:N . . . . . 98, 7433  
\bool\_show:n . . . . . 98, 7427  
\bool\_until\_do:Nn . . . . . 101, 7570  
\bool\_until\_do:nn . . . . . 102, 7582  
\bool\_while\_do:Nn . . . . . 101, 7570  
\bool\_while\_do:nn . . . . . 102, 7582  
\bool\_xor:nnTF . . . . . 101, 7561  
\bool\_xor\_p:nn . . . . . 101, 7561  
\c\_false\_bool . . . . . 20, 97, 336, 368,  
470, 473, 473, 473, 474, 2564, 2616,  
2617, 2648, 2667, 2672, 2712, 2731,  
2937, 2944, 3797, 3991, 7382, 7389,  
7395, 7499, 7522, 7528, 7546, 7715,  
21046, 21064, 21278, 21314, 21613,  
21754, 21771, 21827, 21970, 23211  
\g\_tmpta\_bool . . . . . 99, 7442  
\l\_tmpta\_bool . . . . . 99, 7442  
\g\_tmpb\_bool . . . . . 99, 7442  
\l\_tmpb\_bool . . . . . 99, 7442  
\c\_true\_bool . . . . . 20, 97, 336, 387,  
470, 473, 473, 473, 474, 2616, 2648,  
2712, 2730, 2958, 7386, 7392, 7500,  
7501, 7520, 7548, 7554, 7709, 20380,  
20512, 20950, 21010, 21060, 21244,  
21269, 21313, 21320, 21752, 21762,  
21825, 22038, 22216, 22227, 22242,  
22412, 22445, 23042, 23119, 23172  
bool internal commands:  
\\_\_bool\_!:\_Nw . . . . . 7479  
\\_\_bool\_&\_0: . . . . . 7491  
\\_\_bool\_&\_1: . . . . . 7491  
\\_\_bool\_&\_2: . . . . . 7491  
\\_\_bool\_(:\_Nw . . . . . 7484  
\\_\_bool\_)\_0: . . . . . 7491  
\\_\_bool\_)\_1: . . . . . 7491  
\\_\_bool\_)\_2: . . . . . 7491  
\\_\_bool\_choose:NNN . . . . . 7486, 7490, 7491  
\\_\_bool\_get\_next:NN . . . . .  
. . . . . 473, 473, 7466, 7469, 7481, 7487,  
7502, 7503, 7504, 7505, 7506, 7507  
\\_\_bool\_if\_p:n . . . . . 7458  
\\_\_bool\_if\_p\_aux:w . . . . . 473, 7458  
\\_\_bool\_lazy\_all:n . . . . . 7508  
\\_\_bool\_lazy\_any:n . . . . . 7534

\\_\_bool\_p:Nw ..... 7489  
 \\_\_bool\_show>NN ..... 7433  
 \\_\_bool\_to\_str:n ..... 7427, 7440  
 \\_\_bool\_l\_0: ..... 7491  
 \\_\_bool\_l\_1: ..... 7491  
 \\_\_bool\_l\_2: ..... 7491  
 \botmark ..... 311  
 \botmarks ..... 617, 1430  
 \box ..... 312  
 box commands:  
   \box\_autosize\_to\_wd\_and\_ht:Nnn ..  
     ..... 229, 24246  
   \box\_autosize\_to\_wd\_and\_ht\_plus\_-  
     dp:Nnn ..... 229, 24246  
   \box\_clear:N ..... 221,  
     221, 23623, 23630, 24391, 24481, 24551  
   \box\_clear\_new:N ..... 221, 23629  
   \box\_clip:N ..... 243, 243, 243, 25688  
   \box\_dp:N ..... 222, 14202, 23655,  
     23664, 23668, 23991, 24120, 24235,  
     24254, 24260, 24588, 24589, 24695,  
     24700, 24728, 24742, 24927, 24948,  
     25265, 25708, 25715, 25720, 25864,  
     28203, 28205, 28593, 28595, 28804,  
     28806, 29304, 29306, 29308, 29310  
   \box\_gautosize\_to\_wd\_and\_ht:Nnn ..  
     ..... 229, 24246  
   \box\_gautosize\_to\_wd\_and\_ht\_-  
     plus\_dp:Nnn ..... 229, 24246  
   \box\_gclear:N 221, 23623, 23632, 24400  
   \box\_gclear\_new:N ..... 221, 23629  
   \box\_gclip:N ..... 243, 25688  
   \box\_gresize\_to\_ht:Nn .... 229, 24139  
   \box\_gresize\_to\_ht\_plus\_dp:Nn ...  
     ..... 230, 24139  
   \box\_gresize\_to\_wd:Nn .... 230, 24139  
   \box\_gresize\_to\_wd\_and\_ht:Nnn ...  
     ..... 230, 24139  
   \box\_gresize\_to\_wd\_and\_ht\_plus\_-  
     dp:Nnn ..... 230, 24090, 25978  
   \box\_grotate:Nn .... 231, 23972, 25811  
   \box\_gscales:Nnn .... 231, 24217, 26020  
   \box\_gset\_dp:Nn .... 222, 23667, 23669  
   \box\_gset\_eq:NN .....  
     221, 23626, 23635, 24573, 25698, 25749  
   \box\_gset\_eq\_clear:NN ..... 24313  
   \box\_gset\_eq\_drop:N ..... 24316  
   \box\_gset\_eq\_drop:NN ..... 228, 23643  
   \box\_gset\_ht:Nn .... 222, 23676, 23678  
   \box\_gset\_to\_last:N ..... 223, 23715  
   \box\_gset\_trim:Nnnnn ..... 243, 25694  
   \box\_gset\_viewport:Nnnnn .. 243, 25745  
   \box\_gset\_wd:Nn .... 223, 23685, 23687  
  
 \box\_ht:N .....  
   ..... 222, 14201, 23655, 23673, 23677,  
   23990, 24119, 24234, 24247, 24250,  
   24254, 24260, 24476, 24546, 24590,  
   24591, 24686, 24691, 24728, 24735,  
   24926, 24947, 25263, 25725, 25733,  
   25738, 25858, 25862, 28205, 28595,  
   28704, 28806, 29028, 29308, 29310  
 \box\_if\_empty:NTF ..... 223, 23711  
 \box\_if\_empty\_p:N ..... 223, 23711  
 \box\_if\_exist:NTF .....  
   ..... 221, 23630, 23632, 23651, 23748  
 \box\_if\_exist\_p:N ..... 221, 23651  
 \box\_if\_horizontal:NTF ..... 223, 23703  
 \box\_if\_horizontal\_p:N ..... 223, 23703  
 \box\_if\_vertical:NTF ..... 223, 23703  
 \box\_if\_vertical\_p:N ..... 223, 23703  
 \box\_log:N ..... 224, 23734  
 \box\_log:Nnn ..... 224, 23734  
 \box\_move\_down:nn 222, 1017, 23692,  
   25712, 25720, 25763, 25770, 25836  
 \box\_move\_left:nn ..... 222, 23692  
 \box\_move\_right:nn ..... 222, 23692  
 \box\_move\_up:nn 222, 23692, 24967,  
   25260, 25729, 25738, 25777, 25790  
 \box\_new:N .....  
   ..... 221,  
     221, 23615, 23723, 23724, 23725,  
     23726, 23727, 23971, 24332, 24407  
 \box\_resize:Nnn ..... 29684, 29686  
 \box\_resize\_to\_ht:Nn ..... 229, 24139  
 \box\_resize\_to\_ht\_plus\_dp:Nn ...  
   ..... 230, 24139  
 \box\_resize\_to\_wd:Nn .... 230, 24139  
 \box\_resize\_to\_wd\_and\_ht:Nnn ...  
   ..... 230, 24139  
 \box\_resize\_to\_wd\_and\_ht\_plus\_-  
   dp:Nnn .....  
     ..... 230, 24090, 25971, 29685, 29687  
 \box\_rotate:Nn .... 231, 23972, 25809  
 \box\_scale:Nnn .... 231, 24217, 26017  
 \box\_set\_dp:Nn .....  
   ..... 222, 1017, 23661, 24017, 24288,  
   24291, 24927, 24948, 25264, 25715,  
   25723, 25766, 25771, 25841, 29663  
 \box\_set\_eq:NN . 221, 23624, 23635,  
   24561, 24950, 25268, 25695, 25746  
 \box\_set\_eq\_clear:NN ..... 24313  
 \box\_set\_eq\_drop:N ..... 24313  
 \box\_set\_eq\_drop:NN ..... 228, 23643  
 \box\_set\_ht:Nn .....  
   ..... 222, 23661, 24016, 24287,  
   24292, 24926, 24947, 25262, 25732,  
   25741, 25780, 25793, 25839, 29662  
 \box\_set\_to\_last:N ..... 223, 23715

```

\box_set_trim:Nnnnn . . . . . 243, 25694
\box_set_viewport:Nnnnn . . 243, 25745
\box_set_wd:Nn . . . . . .
. 223, 23661, 24018, 24304, 24928,
24949, 25266, 25842, 28607, 29661
\box_show:N . . . . . 224, 227, 23728
\box_show:Nnn . . . . . 224, 23728
\box_use:N . . . . . 221, 222,
222, 23688, 24005, 24964, 24967,
25257, 25260, 25705, 25756, 25837,
28210, 28228, 28242, 28483, 28598,
28625, 28639, 28809, 28825, 28837,
29275, 29339, 29356, 29375, 29664
\box_use_clear:N . . . . . 29688, 29690
\box_use_drop:N . . . . . 228, 23688, 24020,
24299, 24308, 25050, 25187, 25713,
25721, 25730, 25739, 25764, 25770,
25778, 25791, 25844, 29689, 29691
\box_wd:N . . . . . 222, 14200,
23655, 23682, 23686, 23992, 24121,
24236, 24268, 24592, 24593, 24690,
24699, 24717, 24722, 24881, 24928,
24949, 24965, 25258, 25267, 25757,
25861, 25869, 26063, 26070, 28204,
28212, 28594, 28600, 28703, 28805,
28811, 29027, 29305, 29307, 29343
\c_empty_box . . . . .
. 221, 223, 223, 23624, 23626, 23723
\g_tmpa_box . . . . . 224, 23724
\l_tmpa_box . . . . . 223, 23724
\g_tmpb_box . . . . . 224, 23724
\l_tmpb_box . . . . . 223, 23724
box internal commands:
\l_box_angle_fp . . . . .
. 23960, 23982, 23983, 23984, 24013
\__box_autosize:NnnnN . . . . . 24246
\l_box_bottom_dim . . . . . 23963,
23991, 24048, 24052, 24057, 24063,
24068, 24072, 24081, 24083, 24112,
24120, 24129, 24173, 24235, 24241
\l_box_bottom_new_dim . . . . .
23967, 24017, 24049, 24060, 24071,
24082, 24128, 24240, 24288, 24292
\l_box_cos_fp . . . . . 23961,
23984, 23996, 24001, 24028, 24040
\__box_dim_eval:n . . . . .
. 23605, 23664, 23668, 23673,
23677, 23682, 23686, 23693, 23695,
23697, 23699, 23783, 23789, 23819,
23826, 23834, 23856, 23895, 23901,
23932, 23939, 23951, 23957, 25704,
25706, 25755, 25757, 25766, 25790
\__box_dim_eval:w . . . . . 23605
\l_box_internal_box . . . . . 23971, 24005,
24006, 24012, 24016, 24017, 24018,
24020, 24278, 24287, 24288, 24291,
24292, 24299, 24304, 24308, 25702,
25710, 25713, 25715, 25718, 25721,
25723, 25725, 25727, 25730, 25732,
25733, 25736, 25738, 25739, 25741,
25743, 25753, 25761, 25764, 25766,
25769, 25770, 25771, 25775, 25778,
25780, 25788, 25791, 25793, 25795
\l_box_left_dim . . . . . 23963, 23993,
24048, 24050, 24059, 24063, 24068,
24074, 24079, 24083, 24122, 24237
\l_box_left_new_dim . . . . . 23967, 24008,
24019, 24051, 24062, 24073, 24084
\__box_log:nNnn . . . . . 23734
\__box_resize:N . . . . .
. 24090, 24156, 24176, 24193, 24214
\__box_resize:NNN . . . . . 24090
\__box_resize_common:N . . . . .
. 24132, 24244, 24276
\__box_resize_set_corners:N . . . .
. 24090, 24149, 24169, 24189, 24206
\__box_resize_to_ht:NnN . . . . . 24139
\__box_resize_to_ht_plus_dp:NnN .
. 24139
\__box_resize_to_wd:NnN . . . . . 24139
\__box_resize_to_wd_and_ht:NnnN .
. 24197, 24200, 24202
\__box_resize_to_wd_and_ht_plus_dp:NnnN .
. 24090
\__box_resize_to_wd_ht:NnnN . . . . . 24139
\l_box_right_dim . . . . . 23963, 23992,
24046, 24052, 24057, 24061, 24070,
24072, 24081, 24085, 24108, 24121,
24127, 24191, 24208, 24236, 24243
\l_box_right_new_dim . . . . . 23967,
24019, 24053, 24064, 24075, 24086,
24126, 24242, 24296, 24298, 24304
\__box_rotate:N . . . . . 23972
\__box_rotate:NnN . . . . . 23972
\__box_rotate_quadrant_four: . . .
. 23972, 24077
\__box_rotate_quadrant_one: . . .
. 23972, 24044
\__box_rotate_quadrant_three: . . .
. 23972, 24066
\__box_rotate_quadrant_two: . . .
. 23972, 24055
\__box_rotate_xdir:nnN . . . . .
23972, 24022, 24050, 24052, 24061,
24063, 24072, 24074, 24083, 24085
\__box_rotate_ydir:nnN . . . . .
23972, 24033, 24046, 24048, 24057,

```

24059, 24068, 24070, 24079, 24081  
`\__box_scale:N` ..... [24217](#), [24273](#)  
`\__box_scale:NnnN` ..... [24217](#)  
`\l__box_scale_x_fp` ..... [24088](#),  
 24107, 24127, 24155, 24175, 24190,  
 24192, 24207, 24227, 24243, 24268,  
 24270, 24271, 24272, 24282, 24294  
`\l__box_scale_y_fp` .....  
 .... [24088](#), 24109, 24129, 24131,  
 24150, 24155, 24170, 24175, 24192,  
 24209, 24228, 24239, 24241, 24269,  
 24270, 24271, 24272, 24283, 24285  
`\__box_set_trim:NnnnnN` ..... [25694](#)  
`\__box_set_viewport:NnnnnN` .....  
 .... [25746](#), 25749, 25751  
`\__box_show>NNnn` .. [23732](#), [23742](#), [23746](#)  
`\l__box_sin_fp` .....  
 .. [23961](#), 23983, 23994, 24029, 24039  
`\l__box_top_dim` [23963](#), 23990, 24046,  
 24050, 24059, 24061, 24070, 24074,  
 24079, 24085, 24112, 24119, 24131,  
 24153, 24173, 24212, 24234, 24239  
`\l__box_top_new_dim` .....  
[23967](#), 24016, 24047, 24058, 24069,  
 24080, 24130, 24238, 24287, 24291  
`\__box_viewport:NnnnnN` ..... [25745](#)  
`\boxdir` ..... 901, 1813  
`\boxdirection` ..... 902  
`\boxmaxdepth` ..... 313  
`bp` ..... 201  
`\breakafterdirmode` ..... 903, 1724  
`\brokenpenalty` ..... 314

**C**

`\c` ..... 27699  
`\catcode` 4, 5, 6, 7, 10, 218, 219, 220, 221,  
 222, 223, 224, 225, 226, 231, 232,  
 233, 234, 235, 236, 237, 238, 239, 315  
 catcode commands:  
`\c_catcode_active_space_tl` [259](#), [27900](#)  
`\c_catcode_active_tl` .....  
 .... [122](#), [510](#), [8624](#), [8684](#)  
`\c_catcode_letter_token` .....  
 .... [122](#), [509](#), [8606](#), [8674](#), [20119](#)  
`\c_catcode_other_space_tl` .....  
 .... [118](#), [570](#), [8604](#),  
 10721, 10765, 10845, 10934, 11010  
`\c_catcode_other_token` .....  
 .... [122](#), [510](#), [8606](#), [8679](#), [20117](#)  
`\catcodetable` ..... 904, 1725  
`cc` ..... 201  
`ceil` ..... 198  
`\char` ..... 316, 8789

## char commands:

`\l_char_active_seq` .....  
 .... [121](#), [144](#), [8437](#), [11072](#)  
`\char_codepoint_to_bytes:n` .....  
[259](#), [26771](#), [27484](#), [27504](#), [27505](#), [27661](#)  
`\char_fold_case:N` ..... [259](#), [26743](#)  
`\char_generate:nn` ..... 37,  
 118, 382, 943, 1071, 4275, 4290,  
 8464, 8604, 10264, 20672, 21681,  
 26765, 27419, 27464, 27478, 27480,  
 27515, 27516, 27521, 27523, 27531,  
 27532, 27537, 27539, 27654, 27655  
`\char_gset_active_eq:NN` ... [117](#), [8443](#)  
`\char_gset_active_eq:NN` ... [117](#), [8443](#)  
`\char_lower_case:N` ..... [259](#), [26743](#)  
`\char_mixed_case:N` ..... [259](#), [26743](#)  
`\char_set_active_eq:NN` .....  
 .... [117](#), [8443](#), [11075](#)  
`\char_set_active_eq:nN` .... [117](#), [8443](#)  
`\char_set_catcode:nn` .....  
 120, 248, 249, 250, 251, 252, 253,  
 254, 255, 256, [8343](#), 8350, 8352,  
 8354, 8356, 8358, 8360, 8362, 8364,  
 8366, 8368, 8370, 8372, 8374, 8376,  
 8378, 8380, 8382, 8384, 8386, 8388,  
 8390, 8392, 8394, 8396, 8398, 8400,  
 8402, 8404, 8406, 8408, 8410, 8412  
`\char_set_catcode_active:N` .....  
 .... [119](#), [8349](#), [8444](#), [8502](#),  
 8625, 9489, 10344, 19923, 22896, 27901  
`\char_set_catcode_active:n` .....  
 .... [119](#), [8381](#), [8565](#), [12001](#), [12002](#)  
`\char_set_catcode_alignment:N` ...  
 .... [119](#), [8349](#), [8613](#), [22948](#)  
`\char_set_catcode_alignment:n` ...  
 .... [119](#), [266](#), [8381](#), [8549](#)  
`\char_set_catcode_comment:N` [119](#), [8349](#)  
`\char_set_catcode_comment:n` [119](#), [8381](#)  
`\char_set_catcode_end_line:N` ...  
 .... [119](#), [8349](#)  
`\char_set_catcode_end_line:n` ...  
 .... [119](#), [8381](#)  
`\char_set_catcode_escape:N` [119](#), [8349](#)  
`\char_set_catcode_escape:n` [119](#), [8381](#)  
`\char_set_catcode_group_begin:N` ...  
 .... [119](#), [8349](#), [19996](#), [22899](#)  
`\char_set_catcode_group_begin:n` ...  
 .... [119](#), [8381](#), [8542](#)  
`\char_set_catcode_group_end:N` ...  
 .... [119](#), [8349](#), [19999](#), [22916](#)  
`\char_set_catcode_group_end:n` ...  
 .... [119](#), [8381](#), [8544](#)  
`\char_set_catcode_ignore:N` [119](#), [8349](#)

```

\char_set_catcode_ignore:n ..... 119, 263, 264, 8381
\char_set_catcode_invalid:N 119, 8349
\char_set_catcode_invalid:n 119, 8381
\char_set_catcode_letter:N ..... 119, 8349, 15783, 15784, 22925
\char_set_catcode_letter:n ..... 119, 267, 269, 8381, 8561
\char_set_catcode_math_subscript:N ..... 119, 8349, 8617, 22913
\char_set_catcode_math_subscript:n ..... 119, 8381, 8556
\char_set_catcode_math_superscript:N ..... 119, 8349, 22951
\char_set_catcode_math_superscript:n ..... 119, 268, 8381, 8554
\char_set_catcode_math_toggle:N ..... 119, 8349, 8611, 22928
\char_set_catcode_math_toggle:n ..... 119, 8381, 8547
\char_set_catcode_other:N ..... 119, 8349, 20156, 22931
\char_set_catcode_other:n ..... 119, 265, 270, 8381, 8505, 8563
\char_set_catcode_parameter:N ..... 119, 8349, 22934
\char_set_catcode_parameter:n ..... 119, 8381, 8552
\char_set_catcode_space:N . 119, 8349
\char_set_catcode_space:n ..... 119, 271, 8381, 8559
\char_set_lccode:nn ..... 120,
8413, 8450, 8569, 8570, 9485, 9486,
9487, 9488, 10341, 10342, 10343, 27902
\char_set_mathcode:nn ..... 121, 8413
\char_set_sfcode:nn ..... 121, 8413
\char_set_uccode:nn ..... 120, 8413
\char_show_value_catcode:n 120, 8343
\char_show_value_lccode:n . 120, 8413
\char_show_value_mathcode:n 121, 8413
\char_show_value_sfcode:n . 121, 8413
\char_show_value_uccode:n . 121, 8413
\l_char_special_seq ..... 121, 8437
\char_upper_case:N ..... 259, 26743
\char_value_catcode:n 120, 248, 249,
250, 251, 252, 253, 254, 255, 256,
4287, 4290, 8343, 26765, 27419, 27465
\char_value_lccode:n ..... 120, 8413, 25595, 26744, 26754
\char_value_mathcode:n . 121, 8413
\char_value_sfcode:n ..... 121, 8413
\char_value_uccode:n 121, 8413, 26746
char internal commands:
\__char_change_case:nN ..... 26743
\__char_codepoint_to_bytes_-
auxi:n ..... 26771
\__char_codepoint_to_bytes_-
auxii:Nnn ..... 26771
\__char_codepoint_to_bytes_-
auxiii:n ..... 26771
\__char_codepoint_to_bytes_end: .
..... 26771
\__char_codepoint_to_bytes_-
output:nnm ..... 26771
\__char_codepoint_to_bytes_-
outputi:nw ..... 26771
\__char_codepoint_to_bytes_-
outputii:nw ..... 26771
\__char_codepoint_to_bytes_-
outputiii:nw ..... 26771
\__char_codepoint_to_bytes_-
outputiv:nw ..... 26771
\__char_data_auxi:w ..... 25566,
25587, 25591, 25619, 25624, 25657
\__char_data_auxii:w ..... 25568, 25569, 25602,
25606, 25627, 25628, 25630, 25632
\g__char_data_ior .. 25541, 25565,
25582, 25589, 25590, 25616, 25622,
25623, 25646, 25659, 25675, 25676
\__char_generate:n ..... 25546, 25578, 25598, 25610,
25611, 25613, 25639, 25640, 25642
\__char_generate:w ..... 25545, 25577, 25597, 25608, 25637
\__char_generate_aux:nn ..... 8464
\__char_generate_aux:nnw ..... 8464
\__char_generate_aux:w . 8466, 8470
\__char_generate_auxii:nnw . 8464
\__char_generate_invalid_-
catcode: ..... 8464
\__char_int_to_roman:w ..... 8463, 8574, 8598
\__char_tmp:n ..... 8567, 8579, 8582, 8584, 8587
\__char_tmp:NN .. 25664, 25670, 25672
\__char_tmp:nn ..... 8445, 8456, 8457
\l__char_tmp_t1 ..... 8464
\l__char_tmpa_t1 ..... 25572,
25573, 25575, 25582, 25584, 25587
\l__char_tmrb_t1 ..... 25574, 25575
\chardef ..... 228, 241, 317
choice commands:
.choice: ..... 171, 12470
choices commands:
.choices:nn ..... 171, 12472
\cite ..... 27747

```

```

\cleaders ..... 318
\clearmarks ..... 905, 1726
clist commands:
  \clist_clear:N ..... 108,
    108, 7805, 7822, 7985, 12593, 12618
  \clist_clear_new:N ..... 108, 7809
  \clist_concat:NNN 108, 7848, 7874, 7887
  \clist_const:Nn ..... 108, 7802
  \clist_count:N ..... 112, 115, 8185, 8214, 8246, 8312
  \clist_count:n .. 112, 8185, 8277, 8303
  \clist_gclear:N .... 108, 7805, 7824
  \clist_gclear_new:N .... 108, 7809
  \clist_gconcat:NNN .....
    .... 108, 7848, 7876, 7889
  \clist_get:NN ..... 114, 7899
  \clist_get:NNTF ..... 114, 7936
  \clist_gpop:NN ..... 114, 7910
  \clist_gpop:NNTF ..... 114, 7936
  \clist_gpush:Nn ..... 114, 7961
  \clist_gput_left:Nn .....
    .... 109, 7873, 7969, 7970,
      7971, 7972, 7973, 7974, 7975, 7976
  \clist_gput_right:Nn .... 109, 7886
  \clist_gremove_all:Nn .... 110, 7995
  \clist_gremove_duplicates:N 110, 7979
  \clist_greverse:N ..... 110, 8034
  .clist_gset:N ..... 171, 12482
  \clist_gset:Nn ..... 109, 7867
  \clist_gset_eq:NN ... 108, 7813, 7982
  \clist_gset_from_seq:NN .....
    .... 108, 7821, 7998, 19604
  \clist_gsort:Nn .... 110, 8052, 19589
  \clist_if_empty:NTF 110, 7857, 8019,
    8052, 8109, 8139, 8159, 8311, 12357
  \clist_if_empty:nTF ..... 111, 8056
  \clist_if_empty_p:N ..... 110, 8052
  \clist_if_empty_p:n ..... 111, 8056
  \clist_if_exist:NTF .....
    .... 108, 7863, 8212, 11201, 11286
  \clist_if_exist_p:N ..... 108, 7863
  \clist_if_in:NnTF 107, 111, 7988, 8070
  \clist_if_in:nnTF ... 111, 8070, 13594
  \clist_item:Nn 115, 499, 499, 8243, 8312
  \clist_item:nn .. 115, 499, 8274, 8307
  \clist_log:N ..... 115, 8315
  \clist_log:n ..... 115, 8329
  \clist_map_break: .... 112, 8113,
    8118, 8127, 8131, 8147, 8165, 8181
  \clist_map_break:n ..... 112,
    8090, 8181, 12734, 12795, 19597, 19603
  \clist_map_function:NN ..... 63,
    111, 5869, 5879, 8093, 8107, 8190, 8325
  \clist_map_function:Nn ..... 495
  \clist_map_function:nN .. 111, 250, 251, 496, 5874, 5884,
    8123, 8334, 12847, 26341, 28391, 29112
  \clist_map_function:nn ..... 29552
  \clist_map_inline:Nn 111, 111, 494,
    7986, 8137, 12729, 12785, 19597, 19603
  \clist_map_inline:nn ..... .
    111, 2189, 2197, 4029, 8137, 12318,
      12388, 13064, 25361, 25373, 26614
  \clist_map_variable:NNn ... 112, 8157
  \clist_map_variable:nNn ... 112, 8157
  \clist_new:N .. 107, 108, 483, 7800,
    7977, 8336, 8337, 8338, 8339, 12141
  \clist_pop:NN ..... 114, 7910
  \clist_pop:NNTF ..... 114, 7936
  \clist_push:Nn ..... 114, 7961
  \clist_put_left:Nn .....
    .... 109, 7873, 7961, 7962,
      7963, 7964, 7965, 7966, 7967, 7968
  \clist_put_right:Nn .....
    .... 109, 7886, 7989, 12824
  \clist_rand_item:N ..... 115, 8302
  \clist_rand_item:n ... 115, 252, 8302
  \clist_remove_all:Nn .... 110, 7995
  \clist_remove_duplicates:N .....
    .... 107, 110, 7979
  \clist_reverse:N ..... 110, 8034
  \clist_reverse:n .....
    .... 110, 492, 8035, 8037, 8040
  .clist_set:N ..... 171, 12482
  \clist_set:Nn 109, 7867, 7874, 7876,
    7887, 7889, 8076, 8153, 8170, 12356
  \clist_set_eq:NN 108, 7813, 7980, 12714
  \clist_set_from_seq:NN .....
    .... 108, 7821, 7996, 19598
  \clist_show:N ..... 115, 115, 8315
  \clist_show:n ..... 115, 115, 8329
  \clist_sort:Nn ..... 110, 8052, 19589
  \clist_use:Nn ..... 113, 8210
  \clist_use:Nnnn ..... 113, 440, 8210
  \c_empty_clist .....
    ... 116, 7753, 7901, 7916, 7938, 7952
  \l_foo_clist ..... 206
  \g_tmptpa_clist ..... 116, 8336
  \l_tmptpa_clist ..... 116, 8336
  \g_tmptpb_clist ..... 116, 8336
  \l_tmptpb_clist ..... 116, 8336
clist internal commands:
  \__clist_concat>NNNN ..... 7848
  \__clist_count:n ..... 8185
  \__clist_count:w ..... 8185
  \__clist_get:wN ..... 7899, 7941
  \__clist_if_empty_n:w ..... 8056
  \__clist_if_empty_n:wNw ..... 8056

```

\\_\_clist\_if\_in\_return:nnN . . . . . 8070  
 \\_\_clist\_if\_wrap:nTF . . . . .  
   ... 484, 7775, 7799, 7840, 8001, 8082  
 \\_\_clist\_if\_wrap:w . . . . . 484, 7775  
 \l\_\_clist\_internal\_clist . . . . . 487,  
   7754, 7879, 7880, 7892, 7893, 8076,  
   8077, 8078, 8153, 8154, 8170, 8171  
 \l\_\_clist\_internal\_remove\_clist .  
   ... 7977, 7985, 7988, 7989, 7991  
 \l\_\_clist\_internal\_remove\_seq . . . . .  
   ... 7977, 8003, 8004, 8005  
 \\_\_clist\_item:nnnN . . . . . 8243, 8276  
 \\_\_clist\_item\_n:nw . . . . . 8274  
 \\_\_clist\_item\_n\_end:n . . . . . 8274  
 \\_\_clist\_item\_N\_loop:nw . . . . . 8243  
 \\_\_clist\_item\_n\_loop:nw . . . . . 8274  
 \\_\_clist\_item\_n\_strip:n . . . . . 8274  
 \\_\_clist\_item\_n\_strip:w . . . . . 8274  
 \\_\_clist\_map\_function:Nw . . . . .  
   ... 494, 8107, 8144  
 \\_\_clist\_map\_function\_n:Nn . . . . . 495, 8123  
 \\_\_clist\_map\_unbrace:Nw . . . . . 495, 8123  
 \\_\_clist\_map\_variable:Nnw . . . . . 8157  
 \\_\_clist\_pop:NNN . . . . . 7910  
 \\_\_clist\_pop:wN . . . . . 7910  
 \\_\_clist\_pop:wwNNN . . . . . 488, 7910, 7955  
 \\_\_clist\_pop\_TF:NNN . . . . . 7936  
 \\_\_clist\_put\_left:NNNn . . . . . 7873  
 \\_\_clist\_put\_right:NNNn . . . . . 7886  
 \\_\_clist\_rand\_item:nn . . . . . 8302  
 \\_\_clist\_remove\_all: . . . . . 7995  
 \\_\_clist\_remove\_all:NNNn . . . . . 7995  
 \\_\_clist\_remove\_all:w . . . . . 491, 7995  
 \\_\_clist\_remove\_duplicates:NN . . . . . 7979  
 \\_\_clist\_reverse:wwNww . . . . . 492, 8040  
 \\_\_clist\_reverse\_end:ww . . . . . 492, 8040  
 \\_\_clist\_sanitize:n . . . . .  
   ... 7762, 7803, 7868, 7870  
 \\_\_clist\_sanitize:Nn . . . . . 484, 7762  
 \\_\_clist\_set\_from\_seq:n . . . . . 7821  
 \\_\_clist\_set\_from\_seq:NNNN . . . . . 7821  
 \\_\_clist\_show:NN . . . . . 8315  
 \\_\_clist\_show:Nn . . . . . 8329  
 \\_\_clist\_tmp:w . . . . . 491, 7755,  
   8008, 8030, 8084, 8093, 8097, 8099  
 \\_\_clist\_trim\_next:w . . . . .  
   484, 494, 7756, 7765, 7773, 8126, 8134  
 \\_\_clist\_use:nwwn . . . . . 8210  
 \\_\_clist\_use:nwwwnwn . . . . . 497, 8210  
 \\_\_clist\_use:wwn . . . . . 8210  
 \\_\_clist\_wrap\_item:w . . . . . 484, 7771, 7798  
 \closein . . . . . 319  
 \closeout . . . . . 320  
 \clubpenalties . . . . . 618, 1431  
 \clubpenalty . . . . . 321  
 cm . . . . . 201  
 code commands:  
   .code:n . . . . . 171, 12480  
 coffin commands:  
   \coffin\_attach:NnnNnnnn . . . . .  
     ... 234, 1004, 24910  
   \coffin\_clear:N . . . . . 232, 24387  
   \coffin\_display\_handles:Nn . . . . . 235, 25160  
   \coffin\_dp:N . . . . .  
     ... 235, 24588, 25284, 25989, 26028  
   \coffin\_gattach:NnnNnnnn . . . . . 234, 24910  
   \coffin\_gclear:N . . . . . 232, 24387  
   \coffin\_gjoin:NnnNnnnn . . . . . 234, 24859  
   \coffin\_gresize:Nnn . . . . . 243, 25968  
   \coffin\_grotate:Nn . . . . . 243, 25808  
   \coffin\_gsacle:Nnn . . . . . 244, 26016  
   \coffin\_gset\_eq:NN . . . . .  
     ... 232, 24557, 24569, 24580, 24868, 24919  
   \coffin\_gset\_horizontal\_pole:Nnn .  
     ... 233, 24618  
   \coffin\_gset\_vertical\_pole:Nnn . . . . .  
     ... 234, 24618  
   \coffin\_ht:N . . . . .  
     ... 235, 24588, 25283, 25989, 26028  
   \coffin\_if\_exist:NTF . . . . . 232, 24366, 24380  
   \coffin\_if\_exist\_p:N . . . . . 232, 24366  
   \coffin\_join:NnnNnnnn . . . . . 234, 24859  
   \coffin\_log\_structure:N . . . . . 235, 25270  
   \coffin\_mark\_handle:Nnnn . . . . . 235, 25105  
   \coffin\_new:N . . . . . 232, 988, 24405, 24581,  
     24582, 24583, 24584, 24585, 24586,  
     24587, 25043, 25053, 25054, 25055  
   \coffin\_resize:Nnn . . . . . 243, 25968  
   \coffin\_rotate:Nn . . . . . 243, 25808  
   \coffin\_scale:Nnn . . . . . 244, 26016  
   \coffin\_scale:NnnNN . . . . . 26016  
   \coffin\_set\_eq:NN . . . . . 232, 24557,  
     24862, 24913, 24941, 24969, 25181  
   \coffin\_set\_eq:NN\coffin\_gset\_-  
     eq:NN . . . . . 232  
   \coffin\_set\_horizontal\_pole:Nnn . . . . .  
     ... 233, 24618  
   \coffin\_set\_vertical\_pole:Nnn . . . . .  
     ... 234, 24618  
   \coffin\_show\_structure:N . . . . .  
     ... 235, 235, 25270  
   \coffin\_typeset:Nnnnn . . . . . 234, 25045  
   \coffin\_wd:N . . . . .  
     ... 235, 24588, 25285, 25985, 26032  
   \c\_empty\_coffin . . . . . 236, 24581  
   \g\_tmpt\_coffin . . . . . 236, 24584  
   \l\_tmpt\_coffin . . . . . 236, 24584  
   \g\_tmptb\_coffin . . . . . 236, 24584

```

\l_tmpb_coffin ..... 236, 24584
coffin internal commands:
  \__coffin_align:NnnNnnnnN .....
    .. 24873, 24924, 24945, 24952, 25048
  \l__coffin_aligned_coffin .....
    .. 24581, 24874,
    24875, 24879, 24885, 24888, 24891,
    24907, 24908, 24925, 24926, 24927,
    24928, 24929, 24932, 24936, 24940,
    24941, 24946, 24947, 24948, 24949,
    24950, 24983, 24999, 25049, 25050,
    25255, 25262, 25264, 25266, 25268
  \l__coffin_aligned_internal_-
    coffin ..... 24581, 24962, 24969
  \__coffin_attach:NnnNnnnnN ... 24910
  \__coffin_attach_mark:NnnNnnnn ...
    .. 24910, 25117, 25138, 25154
  \l__coffin_bottom_corner_dim ...
    .. 25804, 25836, 25840,
    25920, 25931, 25932, 25952, 25960
  \l__coffin_bounding_prop .....
    .. 25800, 25825, 25857,
    25859, 25865, 25867, 25876, 25939
  \l__coffin_bounding_shift_dim ...
    .. 25803, 25834, 25938, 25944, 25945
  \__coffin_calculate_intersection:Nnn
    .. 24747, 24954, 24957, 25248
  \__coffin_calculate_intersection:nnnnnnnn
    .. 24747, 25194
  \__coffin_calculate_intersection_-
    aux:nnnnnN ..... 24747
  \c__coffin_corners_prop .....
    .. 24335, 24412, 24607, 24614
  \l__coffin_corners_prop .....
    .. 25801, 25816, 25820, 25845,
    25850, 25882, 25922, 25949, 25996,
    26000, 26006, 26012, 26047, 26061
  \l__coffin_cos_fp .....
    1020, 1022, 25798, 25815, 25903, 25912
  \__coffin_display_attach:Nnnnn 25160
  \l__coffin_display_coffin .....
    .. 25053, 25181, 25187, 25257,
    25258, 25263, 25265, 25267, 25268
  \l__coffin_display_coord_coffin .
    .. 25053, 25119,
    25139, 25155, 25202, 25222, 25241
  \l__coffin_display_font_tl .....
    .. 25098, 25127, 25210
  \__coffin_display_handles_-
    aux:nnnn ..... 25160
  \__coffin_display_handles_-
    aux:nnnnnn ..... 25160
  \l__coffin_display_handles_prop .
    .. 25056, 25130, 25134, 25213, 25217
  \l__coffin_display_offset_dim ...
    .. 25093, 25156, 25157, 25242, 25243
  \l__coffin_display_pole_coffin ...
    .. 25053, 25107, 25118, 25162, 25200
  \l__coffin_display_poles_prop ...
    .. 25097, 25172,
    25177, 25180, 25182, 25184, 25191
  \l__coffin_display_x_dim .....
    .. 25095, 25197, 25252
  \l__coffin_display_y_dim .....
    .. 25095, 25198, 25254
  \c__coffin_empty_coffin 25043, 25048
  \l__coffin_error_bool .....
    .. 24356, 24751, 24755,
    24769, 24784, 24815, 25193, 25195
  \__coffin_find_bounding_shift: ..
    .. 25828, 25936
  \__coffin_find_bounding_shift_-
    aux:nn ..... 25936
  \__coffin_find_corner_maxima:N ..
    .. 25827, 25916
  \__coffin_find_corner_maxima_-
    aux:nn ..... 25916
  \__coffin_gattach:NnnNnnnnN .. 24918
  \__coffin_get_pole:NnN .....
    .. 24594, 24749, 24750, 25010, 25011,
    25014, 25015, 25174, 25175, 25178
  \__coffin_greset_structure:N ...
    .. 24401, 24604, 24668
  \__coffin_gupdate:N .....
    .. 24440, 24453, 24505, 24523, 24660
  \__coffin_gupdate_corners:N ...
    .. 24669, 24672
  \__coffin_gupdate_poles:N .....
    .. 24670, 24703
  \__coffin_if_exist:NTF .....
    .. 24378, 24389, 24398, 24420,
    24433, 24458, 24486, 24499, 24528,
    24559, 24571, 24626, 24644, 25278
  \l__coffin_internal_box .....
    .. 24332, 24470,
    24476, 24481, 24540, 24546, 24551,
    25830, 25839, 25841, 25842, 25844
  \l__coffin_internal_dim .....
    .. 24332, 24880, 24882, 24883,
    25864, 25866, 25870, 26027, 26030
  \l__coffin_internal_tl ... 24332,
    24981, 24982, 24984, 25131, 25132,
    25135, 25136, 25144, 25149, 25214,
    25215, 25218, 25219, 25228, 25233
  \__coffin_join:NnnNnnnnN .... 24859
  \l__coffin_left_corner_dim .....
    .. 25804, 25834, 25843,
    25921, 25927, 25928, 25951, 25959

```

```

\__coffin_mark_handle_aux:nnnnNnn
    ..... 25105
\__coffin_offset_corner:Nnnnn . 24990
\__coffin_offset_corners:Nnn ...
    .. 24896, 24897, 24903, 24904, 24990
\__coffin_offset_pole:Nnnnnnn . 24971
\__coffin_offset_poles:Nnn ...
    .. 24894, 24895,
    24900, 24901, 24937, 24938, 24971
\l__coffin_offset_x_dim .....
    ... 24357, 24877, 24878, 24881,
    24892, 24894, 24896, 24902, 24905,
    24939, 24958, 24966, 25251, 25259
\l__coffin_offset_y_dim .....
    ... 24357, 24895, 24897, 24902, 24905,
    24939, 24960, 24967, 25253, 25260
\l__coffin_pole_a_t1 .....
    ... 24359, 24749, 24754, 25010, 25013,
    25014, 25017, 25174, 25176, 25179
\l__coffin_pole_b_t1 .....
    ... 24359,
    24750, 24754, 25011, 25013, 25015,
    25017, 25175, 25176, 25178, 25179
\c__coffin_poles_prop .....
    ... 24342, 24414, 24609, 24616
\l__coffin_poles_prop .....
    ... 25801, 25818, 25822,
    25847, 25852, 25890, 25957, 25998,
    26002, 26008, 26014, 26053, 26068
\__coffin_reset_structure:N ...
    .. 24392, 24604, 24662, 24885, 24929
\__coffin_resize:NnnNN .....
\__coffin_resize_common:NnnN ...
    .. 25992, 25994, 26033
\l__coffin_right_corner_dim ...
    .. 25804, 25843, 25919, 25929, 25930
\__coffin_rotate:NnNN .....
\__coffin_rotate_bounding:nnn ...
    ... 25826, 25873
\__coffin_rotate_corner:Nnnn ...
    ... 25821, 25873
\__coffin_rotate_pole:Nnnnnn ...
    ... 25823, 25885
\__coffin_rotate_vector:nnNN ...
    .. 25875, 25881, 25887, 25888, 25897
\__coffin_scale:NnnNN .....
    ... 26017, 26020, 26022
\__coffin_scale_corner:Nnnn ...
    ... 26001, 26044
\__coffin_scale_pole:Nnnnnn ...
    ... 26003, 26044
\__coffin_scale_vector:nnNN ...
    ... 26037, 26046, 26052
\l__coffin_scale_x_fp 25964, 25984,
    26004, 26024, 26026, 26032, 26040
\l__coffin_scale_y_fp ... 25964,
    25986, 26025, 26026, 26030, 26042
\l__coffin_scaled_total_height_-
    dim ..... 25966, 26029, 26034
\l__coffin_scaled_width_dim .....
    ... 25966, 26031, 26034
\__coffin_set_bounding:N 25824, 25855
\__coffin_set_horizontal_-
    pole:NnnN ..... 24618
\__coffin_set_pole:Nnn ...
    .. 24471, 24541, 24618,
    24983, 25023, 25027, 25035, 25039
\__coffin_set_vertical:NnnNN . 24444
\__coffin_set_vertical:NnNNNNw 24512
\__coffin_set_vertical_pole:NnnN
    ..... 24618
\__coffin_shift_corner:Nnnn ...
    .. 25846, 25947
\__coffin_shift_pole:Nnnnnn ...
    .. 25848, 25947
\__coffin_show_structure:NN .. 25270
\l__coffin_sin_fp .....
    ... 1020, 1022, 25798, 25814, 25904, 25911
\l__coffin_slope_x_fp .....
    ... 24354, 24809, 24814, 24823, 24831
\l__coffin_slope_y_fp .....
    ... 24354, 24811, 24814, 24825, 24832
\__coffin_to_value:N .....
    ... 24365, 24370, 24409, 24410, 24411,
    24413, 24562, 24563, 24564, 24565,
    24574, 24575, 24576, 24577, 24597,
    24606, 24608, 24613, 24615, 24628,
    24646, 24656, 24679, 24710, 24888,
    24932, 24935, 24973, 24992, 24999,
    25173, 25289, 25817, 25819, 25849,
    25851, 25997, 25999, 26011, 26013
\l__coffin_top_corner_dim .....
    ... 25804, 25840, 25918, 25933, 25934
\__coffin_update:N .....
    ... 24427, 24447, 24492, 24516, 24660
\__coffin_update_B:nnnnnnnnN . 25008
\__coffin_update_corners:N .....
    .. 24663, 24672
\__coffin_update_corners:NN .. 24672
\__coffin_update_corners:NNN .. 24672
\__coffin_update_poles:N .....
    .. 24664, 24703, 24891, 24936
\__coffin_update_poles:NN .....
\__coffin_update_poles:NNN .. 24703
\__coffin_update_T:nnnnnnnnN . 25008
\__coffin_update_vertical_-
    poles:NNN .... 24907, 24940, 25008
\l__coffin_x_dim .....
    ... 24361, 24758, 24767, 24787, 24790,
```

24797, 24806, 24817, 24837, 24955,  
 24959, 24978, 24986, 25197, 25249,  
 25875, 25877, 25881, 25883, 25887,  
 25892, 26046, 26048, 26052, 26055  
 $\backslash\text{l\_coffin\_x\_prime\_dim}$  .....  
     ..... [24361](#), 24955,  
 24959, 25249, 25252, 25889, 25893  
 $\backslash\text{l\_coffin\_x\_shift\_corner:Nnnn}$  ..  
     ..... [26007](#), [26059](#)  
 $\backslash\text{l\_coffin\_x\_shift\_pole:Nnnnnn}$  ..  
     ..... [26009](#), [26059](#)  
 $\backslash\text{l\_coffin\_y\_dim}$  .....  
     .... [24361](#), 24759, 24772, 24775,  
 24782, 24799, 24804, 24838, 24956,  
 24961, 24979, 24986, 25198, 25250,  
 25875, 25877, 25881, 25883, 25887,  
 25892, 26046, 26048, 26052, 26055  
 $\backslash\text{l\_coffin\_y\_prime\_dim}$  .....  
     .... [24361](#), 24956,  
 24961, 25250, 25254, 25889, 25894  
 $\backslash\text{color}$  ..... 25113, 25125, 25168, 25208  
 color commands:  
 $\backslash\text{color\_ensure\_current:}$  ..... [237](#),  
     [985](#), [24424](#), [24437](#), [24488](#), [24501](#), [25321](#)  
 $\backslash\text{color\_group\_begin:}$  .....  
     .... [237](#), [237](#), 23765,  
 23770, 23776, 23784, 23790, 23799,  
 23806, 23821, 23828, 23835, 23840,  
 23851, 23853, 23857, 23862, 23868,  
 23874, 23882, 23888, 23896, 23902,  
 23911, 23918, 23934, 23941, [25319](#)  
 $\backslash\text{color\_group\_end:}$  ..... [237](#), [237](#),  
 23765, 23770, 23776, 23784, 23790,  
 23812, 23835, 23840, 23851, 23853,  
 23857, 23862, 23868, 23874, 23882,  
 23888, 23896, 23902, 23925, [25319](#)  
 color internal commands:  
 $\backslash\text{l\_color\_current\_tl}$  .....  
     ... [1007](#), [25319](#), [25324](#), [25326](#), [25341](#)  
 $\backslash\text{l\_color\_select:n}$  ..... [25326](#), [25328](#)  
 $\backslash\text{l\_color\_select:w}$  ..... [25328](#)  
 $\backslash\text{l\_color\_select_cmyk:w}$  ..... [25328](#)  
 $\backslash\text{l\_color\_select_gray:w}$  ..... [25328](#)  
 $\backslash\text{l\_color\_select_rgb:w}$  ..... [25328](#)  
 $\backslash\text{l\_color\_select_spot:w}$  ..... [25328](#)  
 $\backslash\text{columnwidth}$  ..... 24465, 24534  
 $\backslash\text{copy}$  ..... 322  
 $\backslash\text{copyfont}$  ..... 1013, 1622  
 $\cos$  ..... 198  
 $\cosd$  ..... 199  
 $\cot$  ..... 198  
 $\cotd$  ..... 199  
 $\backslash\text{count}$  ..... 171, 173, 174, 175,  
 179, 180, 182, 183, 186, 188, 189,  
 190, 191, 195, 196, 198, 199, 323, 8797  
 $\backslash\text{countdef}$  ..... 324  
 $\backslash\text{cr}$  ..... 325  
 $\backslash\text{crampeddisplaystyle}$  ..... 906, 1727  
 $\backslash\text{crampedscriptscriptstyle}$  ..... 907, 1729  
 $\backslash\text{crampedscriptstyle}$  ..... 909, 1731  
 $\backslash\text{crampedtextstyle}$  ..... 910, 1732  
 $\backslash\text{crcr}$  ..... 326  
 $\backslash\text{creationdate}$  ..... 879  
 $\backslash\text{cs}$  ..... 15637  
 cs commands:  
 $\backslash\text{cs:w}$  ..... 16, [942](#),  
     [942](#), [2063](#), [2085](#), [2087](#), [2136](#), [2760](#),  
 2788, 2986, 3050, 3160, 3209, 3218,  
 3220, 3224, 3225, 3226, 3288, 3294,  
 3300, 3306, 3326, 3328, 3333, 3340,  
 3341, 3447, 3451, 3490, 3978, 6489,  
 6603, 7304, 7374, 7617, 7619, 11358,  
 11692, 11748, 11841, 11882, 12769,  
 12792, 12805, 12817, 13343, 13362,  
 13429, 14258, 14447, 14479, 14896,  
 14922, 14935, 14971, 15482, 17185,  
 18207, 22833, 22836, 23619, 26155  
 $\backslash\text{cs\_end:}$  ..... 16, [373](#), [2063](#),  
 2085, 2087, 2091, 2136, 2754, 2760,  
 2782, 2788, 2913, 2986, 3050, 3160,  
 3209, 3218, 3220, 3224, 3225, 3226,  
 3288, 3294, 3300, 3306, 3326, 3328,  
 3333, 3340, 3341, 3447, 3451, 3490,  
 3978, 6489, 6603, 7304, 7311, 7354,  
 7364, 7374, 7614, 7620, 7622, 7624,  
 7626, 7628, 7630, 7632, 7634, 7636,  
 7638, 7640, 11358, 11692, 11748,  
 11841, 11882, 12442, 12769, 12793,  
 12806, 12817, 13346, 13362, 13437,  
 14261, 14451, 14483, 14902, 14928,  
 14941, 14974, 15488, 17185, 18210,  
 22700, 22850, 23619, 26153, 26155  
 $\backslash\text{cs\_generate\_from\_arg\_count:NNnn}$  .....  
     .... [13](#), [2966](#), 3008  
 $\backslash\text{cs\_generate\_variant:Nn}$  .....  
     .... [9](#), [22](#), [24](#), [25](#), [97](#), [244](#),  
 368, 369, [3756](#), 4020, 4022, 4024,  
 4026, 4060, 4073, 4074, 4079, 4080,  
 4085, 4086, 4106, 4107, 4124, 4125,  
 4157, 4158, 4159, 4160, 4161, 4162,  
 4187, 4188, 4189, 4190, 4191, 4192,  
 4193, 4194, 4219, 4220, 4221, 4222,  
 4223, 4224, 4225, 4226, 4265, 4266,  
 4267, 4268, 4323, 4324, 4325, 4326,  
 4388, 4389, 4394, 4395, 4537, 4555,  
 4569, 4585, 4590, 4592, 4601, 4613,

4614, 4633, 4636, 4641, 4642, 4742,  
 4753, 4754, 4776, 4786, 4921, 4928,  
 4930, 5007, 5028, 5030, 5066, 5081,  
 5082, 5085, 5086, 5097, 5119, 5120,  
 5121, 5122, 5155, 5156, 5161, 5162,  
 5251, 5309, 5327, 5353, 5404, 5465,  
 5543, 5562, 5600, 5615, 5632, 5633,  
 5634, 5647, 5689, 5751, 5752, 5845,  
 5848, 5851, 5854, 5857, 5886, 5887,  
 5888, 5889, 5890, 5891, 5927, 5928,  
 5933, 5934, 5956, 5957, 5958, 5959,  
 5964, 5965, 5966, 5967, 5984, 5985,  
 6010, 6011, 6028, 6029, 6089, 6102,  
 6103, 6121, 6147, 6148, 6200, 6206,  
 6226, 6255, 6266, 6267, 6291, 6314,  
 6328, 6362, 6364, 6492, 6517, 6535,  
 6536, 6541, 6542, 6545, 6548, 6573,  
 6574, 6575, 6576, 6589, 6590, 6591,  
 6592, 6599, 6600, 6602, 7234, 7238,  
 7383, 7396, 7397, 7398, 7399, 7402,  
 7403, 7416, 7417, 7434, 7436, 7574,  
 7575, 7580, 7581, 7804, 7844, 7845,  
 7846, 7847, 7861, 7862, 7871, 7872,  
 7882, 7883, 7884, 7885, 7895, 7896,  
 7897, 7898, 7909, 7934, 7935, 7993,  
 7994, 8032, 8033, 8038, 8039, 8122,  
 8156, 8180, 8193, 8233, 8242, 8266,  
 8273, 8314, 8316, 8318, 8459, 8460,  
 8461, 8462, 9049, 9052, 9055, 9058,  
 9061, 9076, 9079, 9082, 9149, 9150,  
 9151, 9152, 9159, 9160, 9179, 9180,  
 9181, 9182, 9195, 9231, 9233, 9235,  
 9237, 9254, 9255, 9317, 9332, 9338,  
 9340, 9867, 10295, 10296, 10297,  
 10298, 10406, 10411, 10453, 10471,  
 10599, 10624, 10631, 10643, 10658,  
 10661, 10679, 10787, 11166, 11209,  
 11361, 11368, 11374, 11375, 11380,  
 11381, 11400, 11401, 11405, 11409,  
 11419, 11420, 11430, 11431, 11691,  
 11730, 11751, 11758, 11763, 11764,  
 11769, 11770, 11789, 11790, 11792,  
 11794, 11801, 11802, 11809, 11810,  
 11840, 11862, 11863, 11865, 11885,  
 11892, 11899, 11900, 11905, 11906,  
 11931, 11932, 11935, 11938, 11945,  
 11946, 11953, 11954, 11965, 11967,  
 12163, 12262, 12278, 12339, 12453,  
 12586, 12587, 12590, 12598, 12606,  
 12615, 12623, 12631, 12645, 12663,  
 12696, 12850, 13093, 13095, 13129,  
 13711, 13714, 15379, 15386, 15387,  
 15388, 15391, 15392, 15395, 15396,  
 15401, 15402, 15409, 15410, 15411,  
 15412, 15414, 15416, 15632, 15694,  
 18705, 18759, 18837, 18882, 18897,  
 18952, 19567, 19569, 19591, 19594,  
 19600, 19606, 20303, 21033, 23622,  
 23627, 23628, 23633, 23634, 23641,  
 23642, 23649, 23650, 23658, 23659,  
 23660, 23666, 23669, 23675, 23678,  
 23684, 23687, 23690, 23691, 23721,  
 23722, 23730, 23733, 23736, 23745,  
 23763, 23778, 23779, 23792, 23793,  
 23808, 23809, 23830, 23831, 23848,  
 23849, 23876, 23877, 23890, 23891,  
 23904, 23905, 23920, 23921, 23943,  
 23944, 23947, 23948, 23952, 23959,  
 23974, 23977, 24095, 24101, 24141,  
 24144, 24161, 24164, 24181, 24184,  
 24198, 24201, 24219, 24222, 24248,  
 24251, 24257, 24263, 24319, 24320,  
 24324, 24328, 24395, 24404, 24417,  
 24430, 24443, 24449, 24455, 24496,  
 24509, 24518, 24525, 24568, 24580,  
 24620, 24623, 24638, 24641, 24659,  
 24864, 24870, 24915, 24921, 25052,  
 25159, 25245, 25272, 25275, 25330,  
 25690, 25693, 25696, 25699, 25747,  
 25750, 25854, 25974, 25981, 26018,  
 26021, 26193, 26194, 26195, 26196,  
 26232, 26235, 26238, 26249, 26263,  
 26273, 26303, 26343, 26639, 26652,  
 26711, 26720, 26731, 26741, 26847,  
 26850, 26885, 27824, 27828, 27873,  
 27877, 28023, 28078, 28164, 28176,  
 28249, 28451, 28457, 28566, 28586,  
 28793, 28892, 29002, 29018, 29042,  
 29176, 29285, 29292, 29400, 29431  
`\cs_gset:Nn` . . . . . 13, 2981, 3045  
`\cs_gset:Npn` . . . . . 9,  
 11, 313, 2120, 2861, 2875, 2877,  
 6230, 8528, 9395, 9397, 9435, 14137  
`\cs_gset:Npx` . . . . . 11,  
 2120, 2862, 2875, 2878, 6235, 22708  
`\cs_gset_eq:NN` . . . . .  
 . 14, 2893, 2910, 2918, 4058, 4099,  
 4104, 5843, 6240, 6245, 7392, 7395,  
 8457, 9047, 9320, 9328, 10468, 10640  
`\cs_gset_nopar:Nn` . . . . . 13, 2981, 3045  
`\cs_gset_nopar:Npn` . . . . .  
 11, 2120, 2337, 2859, 2867, 2871, 5713  
`\cs_gset_nopar:Npx` . . . . .  
 . . . . . 11, 1067, 2120, 2860, 2867,  
 2872, 4065, 4071, 4150, 4153, 4156,  
 4177, 4180, 4183, 4186, 4209, 4212,  
 4215, 4218, 27763, 27792, 27798, 27827  
`\cs_gset_protected:Nn` 13, 2981, 3045

```

\cs_gset_protected:Npn ..... 11, 2120, 2445, 2865, 2887,
..... 2889, 4559, 5313, 6852, 8142, 9323,
10528, 11665, 15699, 19507, 19515,
19528, 19933, 20201, 20206, 26405
\cs_gset_protected:Npx .. 11, 2120,
2866, 2887, 2890, 6863, 11672, 15706
\cs_gset_protected_nopar:Nn .....
..... 13, 2981, 3045
\cs_gset_protected_nopar:Npn ...
..... 11, 2120, 2863, 2881, 2883
\cs_gset_protected_nopar:Npx ...
..... 11, 2120, 2864, 2881, 2884
\cs_if_eq:NNTF ..... 20,
3077, 3084, 3085, 3088, 3089, 3092,
3093, 9644, 12408, 13910, 13920,
13946, 13948, 13950, 14142, 21657
\cs_if_eq_p:NN ..... 20, 3077, 21673
\cs_if_exist ..... 219
\cs_if_exist:N ..... 20,
4126, 4127, 5935, 5937, 6549, 6551,
7446, 7448, 7863, 7865, 9256, 9258,
11382, 11384, 11771, 11773, 11907,
11909, 15441, 15442, 23651, 23653
\cs_if_exist:NTF .....
..... 15, 20, 310, 366, 512, 541,
2266, 2275, 2740, 2797, 2799, 2801,
2803, 2805, 2807, 2809, 2811, 3097,
3163, 3233, 3269, 3400, 3424, 3492,
3523, 3730, 4045, 5172, 6519, 6520,
6522, 6523, 7246, 7247, 7348, 7720,
7721, 7722, 7730, 8525, 8762, 8781,
9361, 10210, 10394, 10398, 10427,
10487, 10587, 10591, 11025, 11048,
11143, 12177, 12285, 12333, 12712,
12754, 12766, 12776, 12788, 12802,
12815, 12842, 12863, 12871, 13011,
14135, 14310, 15368, 19505, 19523,
19526, 21391, 22822, 24368, 24370,
26118, 26344, 26478, 26496, 26612,
27058, 27078, 27085, 28158, 28167,
28561, 28569, 28575, 28581, 28729,
28765, 28773, 28784, 29803, 29825
\cs_if_exist_p:N .....
..... 20, 310, 310, 2740, 7725, 7749
\cs_if_exist_use:N .....
..... 15, 338, 2796, 12768, 21838
\cs_if_exist_use:NTF .....
..... 15, 2191, 2199, 2796,
2798, 2800, 2806, 2808, 4009, 7740,
12374, 13592, 14268, 14270, 20625,
20632, 20996, 21001, 21038, 21459,
21545, 22757, 26996, 27009, 27011
\cs_if_free:NTF ..... 20, 95,
541, 2768, 2842, 3931, 3945, 3974, 9737
\cs_if_free_p:N .... 19, 20, 95, 2768
\cs_log:N ..... 15, 347, 3122
\cs_meaning:N .....
..... 14, 317, 2072, 2088, 2096, 3134
\cs_new:Nn ..... 11, 96, 2981, 3045
\cs_new:Npn .....
... 9, 10, 13, 95, 95, 313, 314, 326,
327, 847, 2417, 2461, 2519, 2536,
2851, 2875, 2879, 2952, 2954, 2956,
2964, 3016, 3082, 3083, 3084, 3085,
3086, 3087, 3088, 3089, 3090, 3091,
3092, 3093, 3136, 3139, 3148, 3149,
3153, 3154, 3155, 3156, 3157, 3158,
3159, 3161, 3165, 3169, 3172, 3185,
3191, 3197, 3208, 3210, 3217, 3219,
3221, 3228, 3229, 3231, 3235, 3239,
3245, 3247, 3252, 3257, 3263, 3271,
3278, 3279, 3285, 3291, 3297, 3303,
3309, 3316, 3323, 3330, 3337, 3345,
3346, 3347, 3348, 3349, 3350, 3351,
3352, 3353, 3354, 3355, 3356, 3363,
3364, 3365, 3366, 3367, 3368, 3369,
3370, 3371, 3372, 3373, 3374, 3375,
3376, 3377, 3378, 3387, 3388, 3390,
3395, 3402, 3406, 3409, 3419, 3420,
3422, 3426, 3429, 3430, 3432, 3434,
3440, 3446, 3448, 3454, 3456, 3463,
3470, 3471, 3472, 3473, 3474, 3476,
3485, 3487, 3490, 3491, 3494, 3498,
3501, 3503, 3508, 3518, 3521, 3525,
3543, 3544, 3546, 3552, 3557, 3559,
3565, 3585, 3587, 3589, 3595, 3600,
3601, 3624, 3654, 3661, 3667, 3695,
3701, 3706, 3712, 3824, 3845, 3867,
3870, 3879, 3892, 3907, 3918, 3976,
4051, 4260, 4427, 4491, 4494, 4495,
4496, 4497, 4508, 4509, 4514, 4519,
4524, 4529, 4531, 4540, 4542, 4548,
4550, 4586, 4588, 4591, 4593, 4602,
4607, 4612, 4615, 4622, 4629, 4631,
4634, 4645, 4657, 4665, 4671, 4677,
4683, 4690, 4701, 4710, 4712, 4719,
4725, 4727, 4729, 4743, 4745, 4747,
4755, 4760, 4765, 4777, 4778, 4779,
4787, 4831, 4840, 4871, 4892, 4899,
4907, 4913, 4920, 4922, 4927, 4929,
4931, 4932, 4940, 4952, 4961, 4970,
4975, 4981, 5004, 5005, 5006, 5008,
5050, 5171, 5187, 5229, 5234, 5239,
5244, 5249, 5254, 5260, 5265, 5270,
5275, 5280, 5282, 5288, 5290, 5298,
5300, 5302, 5328, 5354, 5356, 5358,
```

5369, 5378, 5381, 5392, 5401, 5403,  
5405, 5413, 5415, 5422, 5443, 5453,  
5458, 5463, 5464, 5466, 5474, 5476,  
5484, 5490, 5496, 5515, 5517, 5526,  
5532, 5539, 5541, 5544, 5554, 5561,  
5563, 5571, 5576, 5581, 5592, 5599,  
5601, 5607, 5609, 5614, 5616, 5622,  
5623, 5628, 5629, 5630, 5631, 5635,  
5640, 5645, 5648, 5650, 5658, 5663,  
5691, 5693, 5695, 5697, 5699, 5701,  
5703, 5705, 5721, 5727, 5735, 5742,  
5749, 5753, 5759, 5792, 5802, 5805,  
5826, 5831, 5919, 5925, 5955, 5968,  
6023, 6057, 6087, 6115, 6120, 6142,  
6177, 6179, 6187, 6193, 6201, 6207,  
6209, 6211, 6220, 6268, 6286, 6290,  
6292, 6315, 6316, 6317, 6324, 6326,  
6389, 6391, 6394, 6400, 6428, 6441,  
6449, 6462, 6469, 6476, 6478, 6603,  
6610, 6624, 6629, 6635, 6646, 6651,  
6658, 6660, 6662, 6664, 6666, 6668,  
6670, 6686, 6691, 6696, 6701, 6706,  
6708, 6714, 6736, 6744, 6752, 6758,  
6764, 6772, 6780, 6786, 6807, 6814,  
6830, 6840, 6842, 6878, 6892, 6898,  
6930, 6962, 6964, 6966, 6972, 6978,  
6990, 6998, 7010, 7018, 7051, 7084,  
7086, 7088, 7090, 7092, 7097, 7102,  
7107, 7112, 7113, 7114, 7115, 7116,  
7117, 7118, 7119, 7120, 7121, 7122,  
7123, 7124, 7125, 7126, 7127, 7128,  
7137, 7138, 7147, 7153, 7155, 7164,  
7171, 7177, 7179, 7181, 7197, 7208,  
7231, 7303, 7335, 7361, 7362, 7371,  
7372, 7431, 7458, 7459, 7468, 7469,  
7479, 7484, 7489, 7491, 7499, 7500,  
7501, 7502, 7503, 7504, 7505, 7506,  
7507, 7508, 7518, 7534, 7544, 7560,  
7570, 7572, 7576, 7578, 7582, 7590,  
7595, 7603, 7609, 7616, 7618, 7620,  
7621, 7623, 7625, 7627, 7629, 7631,  
7633, 7635, 7637, 7639, 7641, 7646,  
7647, 7648, 7649, 7650, 7651, 7652,  
7653, 7654, 7655, 7665, 7667, 7672,  
7674, 7676, 7679, 7681, 7756, 7762,  
7768, 7797, 7798, 7837, 7933, 8029,  
8031, 8040, 8047, 8050, 8063, 8069,  
8107, 8116, 8123, 8129, 8136, 8181,  
8183, 8185, 8203, 8210, 8234, 8235,  
8238, 8240, 8243, 8251, 8267, 8274,  
8282, 8284, 8298, 8300, 8301, 8302,  
8304, 8309, 8345, 8415, 8421, 8427,  
8433, 8464, 8470, 8511, 8519, 8589,  
8711, 8741, 8816, 8828, 8829, 8837,  
8846, 8855, 8868, 8869, 8870, 8871,  
8927, 8935, 8937, 8939, 8949, 8959,  
9006, 9009, 9018, 9027, 9040, 9083,  
9089, 9096, 9101, 9103, 9108, 9110,  
9136, 9183, 9189, 9274, 9280, 9302,  
9311, 9333, 9335, 9434, 9520, 9525,  
9530, 9532, 9534, 9536, 9542, 9551,  
9562, 9568, 9648, 9716, 9718, 9735,  
9868, 10252, 10261, 10266, 10275,  
10280, 10285, 10290, 10512, 10514,  
10684, 10692, 10730, 10747, 10802,  
10853, 10862, 10881, 10882, 10890,  
10896, 10904, 10914, 10919, 10925,  
10931, 11004, 11006, 11008, 11096,  
11298, 11434, 11439, 11446, 11459,  
11467, 11475, 11477, 11495, 11501,  
11514, 11516, 11518, 11520, 11522,  
11530, 11535, 11540, 11545, 11550,  
11552, 11558, 11560, 11568, 11576,  
11582, 11588, 11596, 11604, 11610,  
11631, 11638, 11652, 11688, 11692,  
11695, 11702, 11710, 11719, 11721,  
11832, 11837, 11841, 11850, 11860,  
11962, 12114, 12122, 12124, 12245,  
12813, 12831, 12840, 12846, 12848,  
12851, 12859, 12971, 12972, 12974,  
12988, 13043, 13045, 13051, 13057,  
13074, 13075, 13083, 13178, 13179,  
13180, 13181, 13182, 13183, 13184,  
13185, 13186, 13187, 13218, 13220,  
13222, 13231, 13233, 13245, 13246,  
13248, 13258, 13268, 13278, 13288,  
13296, 13298, 13305, 13307, 13308,  
13313, 13320, 13334, 13336, 13352,  
13353, 13361, 13363, 13372, 13374,  
13386, 13391, 13395, 13400, 13402,  
13404, 13406, 13408, 13415, 13417,  
13425, 13427, 13439, 13441, 13443,  
13445, 13469, 13471, 13473, 13474,  
13475, 13477, 13479, 13481, 13483,  
13501, 13516, 13517, 13523, 13538,  
13545, 13551, 13566, 13572, 13698,  
13699, 13700, 13701, 13702, 13703,  
13704, 13709, 13712, 13758, 13760,  
13762, 13764, 13770, 13774, 13776,  
13785, 13786, 13795, 13808, 13821,  
13828, 13842, 13858, 13870, 13881,  
13891, 13897, 13908, 13918, 13944,  
13955, 13964, 13975, 13980, 14000,  
14002, 14013, 14018, 14031, 14053,  
14054, 14058, 14075, 14076, 14100,  
14108, 14126, 14157, 14183, 14187,  
14190, 14192, 14198, 14210, 14222,  
14229, 14235, 14243, 14266, 14281,

14300, 14308, 14323, 14338, 14349,  
14359, 14369, 14374, 14383, 14400,  
14413, 14418, 14424, 14426, 14433,  
14463, 14491, 14507, 14518, 14523,  
14541, 14559, 14570, 14585, 14590,  
14600, 14610, 14620, 14636, 14684,  
14689, 14696, 14704, 14710, 14715,  
14719, 14736, 14744, 14776, 14793,  
14807, 14826, 14834, 14843, 14852,  
14863, 14865, 14879, 14889, 14890,  
14907, 14914, 14919, 14932, 14945,  
14950, 14997, 14998, 15002, 15019,  
15041, 15043, 15054, 15084, 15088,  
15103, 15120, 15144, 15146, 15148,  
15150, 15160, 15165, 15176, 15188,  
15199, 15212, 15232, 15250, 15252,  
15264, 15270, 15278, 15292, 15299,  
15310, 15317, 15331, 15437, 15439,  
15448, 15470, 15475, 15492, 15519,  
15520, 15521, 15522, 15538, 15549,  
15557, 15569, 15575, 15581, 15589,  
15597, 15603, 15609, 15617, 15625,  
15643, 15656, 15678, 15726, 15732,  
15743, 15767, 15769, 15771, 15773,  
15781, 15785, 15792, 15799, 15800,  
15801, 15802, 15803, 15804, 15807,  
15809, 15838, 15846, 15857, 15859,  
15861, 15863, 15870, 15894, 15896,  
15906, 15921, 15930, 15944, 15952,  
15960, 15967, 15974, 15982, 15992,  
16006, 16017, 16018, 16024, 16041,  
16048, 16050, 16057, 16062, 16079,  
16080, 16081, 16100, 16106, 16116,  
16128, 16135, 16149, 16157, 16195,  
16204, 16225, 16227, 16229, 16238,  
16249, 16261, 16276, 16289, 16302,  
16310, 16328, 16346, 16353, 16361,  
16371, 16372, 16381, 16382, 16391,  
16401, 16415, 16425, 16436, 16444,  
16446, 16457, 16463, 16498, 16519,  
16521, 16523, 16525, 16532, 16541,  
16546, 16553, 16560, 16580, 16585,  
16602, 16613, 16618, 16628, 16630,  
16640, 16647, 16649, 16655, 16657,  
16659, 16663, 16682, 16683, 16688,  
16696, 16697, 16720, 16733, 16740,  
16748, 16749, 16750, 16751, 16752,  
16753, 16761, 16767, 16769, 16771,  
16793, 16798, 16808, 16818, 16829,  
16842, 16853, 16858, 16865, 16874,  
16876, 16885, 16894, 16908, 16910,  
16912, 16925, 16935, 16940, 16949,  
16957, 16964, 16970, 16979, 16981,  
16993, 16998, 17006, 17011, 17021,  
17027, 17033, 17040, 17047, 17049,  
17054, 17056, 17061, 17063, 17077,  
17087, 17099, 17104, 17111, 17121,  
17123, 17134, 17148, 17162, 17182,  
17195, 17197, 17202, 17215, 17220,  
17228, 17233, 17243, 17255, 17285,  
17286, 17287, 17289, 17291, 17293,  
17307, 17313, 17322, 17341, 17347,  
17357, 17376, 17384, 17417, 17423,  
17432, 17434, 17448, 17507, 17515,  
17533, 17550, 17551, 17556, 17581,  
17604, 17633, 17649, 17659, 17670,  
17691, 17706, 17711, 17716, 17718,  
17732, 17738, 17753, 17761, 17771,  
17781, 17815, 17817, 17819, 17821,  
17823, 17838, 17853, 17868, 17883,  
17898, 17913, 17921, 17935, 17937,  
17943, 17955, 17963, 17970, 18196,  
18203, 18240, 18248, 18249, 18260,  
18267, 18269, 18275, 18286, 18296,  
18303, 18310, 18325, 18364, 18377,  
18408, 18414, 18421, 18441, 18443,  
18460, 18475, 18488, 18495, 18500,  
18502, 18511, 18524, 18527, 18548,  
18561, 18576, 18594, 18609, 18619,  
18628, 18641, 18657, 18674, 18687,  
18693, 18695, 18700, 18701, 18702,  
18703, 18706, 18711, 18717, 18722,  
18724, 18747, 18755, 18757, 18760,  
18765, 18771, 18776, 18778, 18801,  
18826, 18835, 18836, 18838, 18843,  
18845, 18850, 18852, 18862, 18870,  
18878, 18880, 18883, 18888, 18893,  
18895, 18896, 18898, 18903, 18908,  
18910, 18916, 18923, 18937, 18942,  
18944, 18954, 18956, 18958, 18960,  
18988, 18990, 18996, 19003, 19009,  
19019, 19024, 19026, 19034, 19035,  
19049, 19056, 19062, 19063, 19076,  
19091, 19097, 19119, 19134, 19144,  
19165, 19174, 19197, 19215, 19226,  
19231, 19242, 19259, 19264, 19310,  
19315, 19329, 19396, 19403, 19410,  
19416, 19423, 19428, 19434, 19446,  
19581, 19736, 19749, 19754, 19760,  
19761, 19768, 19775, 19782, 19789,  
19796, 19797, 19799, 19806, 19812,  
19891, 19896, 19897, 19905, 19911,  
20088, 20093, 20100, 20137, 20142,  
20157, 20180, 20185, 20233, 20239,  
20257, 20262, 20277, 20279, 20281,  
20288, 20333, 20362, 20367, 20623,  
20629, 20640, 20645, 20658, 20663,  
20675, 20687, 20697, 20706, 20726,

20837, 20855, 20863, 20875, 20887,  
 21379, 21655, 21669, 21709, 21750,  
 21910, 22062, 22475, 22607, 22609,  
 22615, 22616, 22623, 22633, 22788,  
 23153, 23158, 23557, 23613, 25352,  
 25353, 25357, 25358, 25390, 25392,  
 26075, 26151, 26161, 26167, 26176,  
 26185, 26187, 26189, 26191, 26197,  
 26240, 26248, 26250, 26257, 26264,  
 26274, 26282, 26284, 26290, 26296,  
 26397, 26412, 26422, 26433, 26439,  
 26452, 26518, 26520, 26668, 26681,  
 26687, 26692, 26704, 26705, 26706,  
 26743, 26745, 26747, 26753, 26755,  
 26761, 26771, 26776, 26820, 26822,  
 26824, 26827, 26830, 26836, 26842,  
 26848, 26851, 26852, 26853, 26854,  
 26855, 26856, 26857, 26865, 26872,  
 26883, 26886, 26892, 26896, 26910,  
 26922, 26927, 26935, 26949, 26954,  
 26965, 26976, 26982, 26987, 26994,  
 27006, 27021, 27028, 27050, 27052,  
 27054, 27056, 27067, 27076, 27102,  
 27113, 27122, 27136, 27142, 27145,  
 27160, 27166, 27168, 27174, 27188,  
 27192, 27198, 27208, 27214, 27232,  
 27242, 27249, 27269, 27279, 27292,  
 27300, 27306, 27326, 27335, 27341,  
 27363, 27376, 27385, 27391, 27407,  
 27700, 27710, 27716, 27844, 27852,  
 27872, 27874, 27875, 27878, 27880,  
 27882, 27884, 27892, 28396, 28500,  
 28737, 28769, 28770, 28936, 29031,  
 29117, 29391, 29394, 29556, 29788  
 \cs\_new:Npx . . . . . 10, 31, 31, 367,  
     367, 2851, 2875, 2880, 3783, 3961,  
     4047, 8194, 8204, 10729, 10741,  
     13327, 14169, 14756, 14980, 15865,  
     17802, 17808, 18420, 18972, 20113,  
     20646, 20648, 20650, 20652, 20654,  
     20656, 26208, 26210, 26212, 26219  
 \cs\_new\_eq:NN . . . . .  
     . . . . . 14, 96, 339, 341, 507, 2652,  
     2893, 3138, 3147, 3178, 3489, 3739,  
     3740, 3741, 3742, 3743, 3744, 3745,  
     3746, 3747, 3748, 3749, 3750, 3751,  
     3752, 3753, 4103, 4104, 4382, 4383,  
     5065, 5079, 5080, 5083, 5084, 5150,  
     5687, 5688, 5822, 5838, 5858, 5859,  
     5860, 5861, 5862, 5863, 5864, 5865,  
     6329, 6330, 6331, 6332, 6333, 6334,  
     6335, 6336, 6337, 6338, 6339, 6340,  
     6341, 6342, 6343, 6344, 6345, 6346,  
     6347, 6348, 6349, 6350, 6351, 6352,  
     6353, 6354, 6382, 6383, 6384, 6385,  
     6386, 6524, 6525, 6528, 6601, 6877,  
     7233, 7237, 7297, 7379, 7380, 7382,  
     7400, 7401, 7706, 7707, 7708, 7709,  
     7712, 7713, 7714, 7715, 7753, 7800,  
     7801, 7805, 7806, 7807, 7808, 7809,  
     7810, 7811, 7812, 7813, 7814, 7815,  
     7816, 7817, 7818, 7819, 7820, 7961,  
     7962, 7963, 7964, 7965, 7966, 7967,  
     7968, 7969, 7970, 7971, 7972, 7973,  
     7974, 7975, 7976, 8463, 8527, 8612,  
     8614, 8615, 8616, 8618, 8621, 8622,  
     8864, 8865, 8866, 9062, 9063, 9064,  
     9065, 9066, 9067, 9068, 9069, 10405,  
     10429, 10484, 10598, 10685, 11225,  
     11233, 11351, 11352, 11353, 11690,  
     11729, 11733, 11734, 11839, 11842,  
     11852, 11864, 11868, 11869, 11964,  
     11966, 11970, 11971, 12942, 12943,  
     13175, 13176, 13177, 13371, 13841,  
     13869, 13877, 13878, 13879, 13888,  
     13890, 14683, 14802, 14803, 14804,  
     15378, 15389, 15390, 18951, 18953,  
     19002, 19884, 19885, 20324, 20336,  
     20458, 20459, 20559, 20567, 20588,  
     20636, 20637, 20638, 20639, 20816,  
     22339, 22961, 23605, 23655, 23656,  
     23657, 23688, 23689, 23700, 23701,  
     23702, 23815, 23846, 23847, 23928,  
     23945, 23946, 24365, 24588, 24589,  
     24590, 24591, 24592, 24593, 25319,  
     25320, 25347, 25348, 25349, 26395,  
     26908, 27004, 27227, 27298, 27299,  
     27775, 27776, 28020, 28245, 28248,  
     28657, 28714, 28715, 28840, 28848,  
     28863, 28987, 29041, 29047, 29048,  
     29171, 29174, 29378, 29379, 29390,  
     29399, 29597, 29608, 29634, 29777  
 \cs\_new\_nopar:Nn . . . . . 12, 2981, 3045  
 \cs\_new\_nopar:Npn . . . . . 10,  
     339, 340, 2851, 2867, 2873, 4681, 4682  
 \cs\_new\_nopar:Npx 10, 2851, 2867, 2874  
 \cs\_new\_protected:Nn . 12, 2981, 3045  
 \cs\_new\_protected:Npn . . . . .  
     . . . . . 10, 326, 2414, 2523,  
     2540, 2851, 2887, 2891, 2893, 2894,  
     2895, 2896, 2897, 2898, 2899, 2900,  
     2901, 2906, 2907, 2908, 2909, 2911,  
     2966, 2976, 2978, 2989, 2998, 3095,  
     3104, 3106, 3108, 3110, 3112, 3120,  
     3122, 3123, 3125, 3126, 3128, 3179,  
     3344, 3357, 3358, 3359, 3360, 3361,  
     3362, 3379, 3380, 3381, 3382, 3383,  
     3384, 3385, 3386, 3414, 3484, 3515,

3757, 3770, 3788, 3792, 3795, 3804,  
3941, 3970, 3981, 3989, 4000, 4019,  
4021, 4023, 4025, 4027, 4043, 4055,  
4062, 4068, 4075, 4077, 4081, 4083,  
4089, 4095, 4114, 4122, 4140, 4143,  
4146, 4149, 4152, 4155, 4164, 4167,  
4170, 4173, 4176, 4179, 4182, 4185,  
4196, 4199, 4202, 4205, 4208, 4211,  
4214, 4217, 4228, 4230, 4232, 4238,  
4251, 4269, 4284, 4301, 4315, 4317,  
4319, 4321, 4327, 4349, 4355, 4384,  
4386, 4390, 4392, 4465, 4466, 4467,  
4556, 4567, 4570, 4576, 4578, 4637,  
4639, 4749, 4751, 5027, 5029, 5031,  
5036, 5038, 5051, 5111, 5113, 5115,  
5117, 5123, 5138, 5151, 5153, 5157,  
5159, 5310, 5325, 5334, 5344, 5346,  
5710, 5813, 5840, 5846, 5849, 5852,  
5855, 5866, 5871, 5876, 5881, 5892,  
5894, 5896, 5929, 5931, 5939, 5947,  
5960, 5962, 5970, 5972, 5974, 5986,  
5988, 5990, 6012, 6014, 6016, 6061,  
6069, 6079, 6090, 6092, 6094, 6096,  
6104, 6122, 6124, 6126, 6227, 6232,  
6237, 6243, 6249, 6256, 6361, 6363,  
6365, 6486, 6497, 6532, 6534, 6537,  
6539, 6544, 6547, 6562, 6565, 6568,  
6571, 6578, 6581, 6584, 6587, 6594,  
6597, 6604, 6844, 6846, 6848, 6855,  
6857, 6859, 6871, 7235, 7239, 7257,  
7301, 7308, 7309, 7320, 7322, 7323,  
7324, 7382, 7385, 7388, 7391, 7394,  
7405, 7411, 7427, 7429, 7433, 7435,  
7437, 7702, 7755, 7802, 7821, 7823,  
7825, 7848, 7850, 7852, 7867, 7869,  
7873, 7875, 7877, 7886, 7888, 7890,  
7899, 7907, 7910, 7912, 7914, 7922,  
7950, 7979, 7981, 7983, 7995, 7997,  
7999, 8034, 8036, 8080, 8137, 8151,  
8157, 8168, 8173, 8315, 8317, 8319,  
8329, 8330, 8331, 8343, 8347, 8349,  
8351, 8353, 8355, 8357, 8359, 8361,  
8363, 8365, 8367, 8369, 8371, 8373,  
8375, 8377, 8379, 8381, 8383, 8385,  
8387, 8389, 8391, 8393, 8395, 8397,  
8399, 8401, 8403, 8405, 8407, 8409,  
8411, 8413, 8417, 8419, 8423, 8425,  
8429, 8431, 8435, 8447, 8872, 8874,  
8876, 8881, 8888, 8897, 8915, 8917,  
8919, 8921, 8923, 8925, 9044, 9050,  
9053, 9056, 9059, 9074, 9077, 9080,  
9126, 9128, 9137, 9143, 9153, 9161,  
9170, 9218, 9219, 9220, 9239, 9241,  
9243, 9318, 9337, 9339, 9341, 9369,  
9377, 9382, 9384, 9391, 9393, 9400,  
9441, 9457, 9477, 9482, 9493, 9577,  
9579, 9642, 9706, 9720, 9745, 9767,  
9768, 9781, 9786, 9812, 9821, 9823,  
9825, 9842, 9869, 9871, 9873, 9875,  
9882, 10300, 10307, 10314, 10326,  
10331, 10348, 10405, 10409, 10425,  
10437, 10454, 10460, 10472, 10473,  
10474, 10501, 10503, 10516, 10518,  
10520, 10526, 10533, 10542, 10544,  
10546, 10551, 10598, 10603, 10607,  
10625, 10632, 10644, 10645, 10646,  
10656, 10659, 10662, 10668, 10674,  
10681, 10683, 10693, 10724, 10735,  
10753, 10788, 10811, 10823, 10842,  
10846, 10935, 10951, 10960, 10968,  
10977, 10984, 10990, 11069, 11083,  
11099, 11116, 11130, 11136, 11167,  
11182, 11188, 11195, 11210, 11227,  
11235, 11241, 11246, 11259, 11280,  
11281, 11282, 11355, 11363, 11370,  
11372, 11376, 11378, 11395, 11398,  
11403, 11407, 11411, 11414, 11422,  
11425, 11524, 11661, 11668, 11680,  
11731, 11735, 11745, 11753, 11760,  
11762, 11765, 11767, 11784, 11787,  
11791, 11793, 11796, 11799, 11804,  
11807, 11866, 11870, 11879, 11887,  
11894, 11897, 11901, 11903, 11926,  
11929, 11934, 11937, 11940, 11943,  
11948, 11951, 11968, 11972, 11986,  
12003, 12009, 12014, 12027, 12033,  
12038, 12052, 12055, 12065, 12077,  
12104, 12115, 12117, 12155, 12157,  
12164, 12169, 12174, 12187, 12193,  
12217, 12229, 12247, 12263, 12279,  
12281, 12283, 12299, 12310, 12312,  
12314, 12337, 12340, 12354, 12367,  
12372, 12376, 12384, 12386, 12396,  
12424, 12433, 12442, 12443, 12454,  
12456, 12458, 12460, 12462, 12464,  
12466, 12468, 12470, 12472, 12474,  
12476, 12478, 12480, 12482, 12484,  
12486, 12488, 12490, 12492, 12494,  
12496, 12498, 12500, 12502, 12504,  
12506, 12508, 12510, 12512, 12514,  
12516, 12518, 12520, 12522, 12524,  
12526, 12528, 12530, 12532, 12534,  
12536, 12538, 12540, 12542, 12544,  
12546, 12548, 12550, 12552, 12554,  
12556, 12558, 12560, 12562, 12564,  
12566, 12568, 12570, 12572, 12574,  
12576, 12578, 12580, 12588, 12591,  
12599, 12607, 12613, 12616, 12624,

12632, 12638, 12646, 12652, 12654,  
12664, 12670, 12675, 12680, 12697,  
12710, 12724, 12750, 12764, 12783,  
12800, 12822, 12876, 12878, 12880,  
12949, 12959, 12993, 12995, 13002,  
13013, 13021, 13028, 13034, 13060,  
13068, 13092, 13094, 13096, 13107,  
13111, 13116, 13130, 13135, 13140,  
13155, 13166, 13188, 13191, 13292,  
13590, 13607, 13609, 13611, 13613,  
13641, 13643, 13645, 13647, 13667,  
13669, 13671, 13673, 13675, 13677,  
13679, 13681, 13683, 15377, 15380,  
15382, 15384, 15393, 15394, 15397,  
15399, 15403, 15404, 15405, 15406,  
15407, 15413, 15415, 15417, 15422,  
15424, 15695, 15702, 15714, 18453,  
19284, 19296, 19334, 19343, 19355,  
19360, 19370, 19378, 19384, 19464,  
19483, 19491, 19503, 19540, 19547,  
19566, 19568, 19570, 19589, 19592,  
19595, 19601, 19607, 19622, 19631,  
19651, 19661, 19672, 19682, 19692,  
19693, 19700, 19706, 19716, 19726,  
19822, 19829, 19831, 19847, 19913,  
19924, 19942, 19952, 19954, 19978,  
19985, 19997, 20000, 20003, 20013,  
20021, 20028, 20037, 20052, 20069,  
20080, 20190, 20197, 20199, 20217,  
20227, 20315, 20319, 20331, 20334,  
20337, 20339, 20348, 20354, 20360,  
20391, 20393, 20394, 20399, 20405,  
20413, 20425, 20441, 20460, 20470,  
20478, 20480, 20488, 20500, 20520,  
20522, 20527, 20535, 20537, 20544,  
20549, 20551, 20553, 20560, 20568,  
20570, 20572, 20574, 20581, 20586,  
20589, 20604, 20831, 20895, 20908,  
20919, 20932, 20965, 20994, 20999,  
21004, 21025, 21034, 21043, 21048,  
21055, 21068, 21070, 21072, 21074,  
21080, 21102, 21115, 21142, 21147,  
21181, 21216, 21237, 21239, 21249,  
21258, 21263, 21272, 21281, 21297,  
21310, 21316, 21327, 21340, 21346,  
21365, 21385, 21416, 21427, 21442,  
21455, 21473, 21481, 21486, 21488,  
21490, 21507, 21526, 21528, 21551,  
21563, 21583, 21604, 21611, 21618,  
21630, 21636, 21693, 21728, 21737,  
21756, 21775, 21781, 21844, 21854,  
21856, 21858, 21865, 21921, 21934,  
21950, 21961, 21980, 21996, 22003,  
22010, 22012, 22026, 22033, 22047,  
22063, 22072, 22086, 22101, 22119,  
22128, 22130, 22142, 22154, 22166,  
22179, 22186, 22206, 22237, 22271,  
22289, 22295, 22304, 22349, 22368,  
22393, 22410, 22432, 22441, 22452,  
22481, 22496, 22511, 22520, 22532,  
22539, 22541, 22543, 22563, 22568,  
22575, 22580, 22585, 22590, 22639,  
22677, 22719, 22735, 22755, 22767,  
22781, 22815, 22829, 22840, 22847,  
22856, 22891, 22897, 22900, 22908,  
22914, 22917, 22926, 22929, 22932,  
22935, 22940, 22949, 22952, 22955,  
22960, 22966, 22971, 22976, 22981,  
22989, 23009, 23011, 23015, 23016,  
23040, 23048, 23057, 23069, 23078,  
23086, 23126, 23170, 23197, 23202,  
23204, 23234, 23262, 23575, 23577,  
23579, 23590, 23616, 23623, 23625,  
23629, 23631, 23636, 23639, 23644,  
23647, 23661, 23667, 23670, 23676,  
23679, 23685, 23692, 23694, 23696,  
23698, 23716, 23719, 23728, 23731,  
23734, 23737, 23739, 23746, 23764,  
23767, 23773, 23781, 23787, 23795,  
23802, 23810, 23817, 23824, 23832,  
23837, 23842, 23844, 23850, 23852,  
23854, 23859, 23865, 23871, 23879,  
23885, 23893, 23899, 23907, 23914,  
23922, 23930, 23937, 23950, 23954,  
23972, 23975, 23978, 23988, 24022,  
24033, 24044, 24055, 24066, 24077,  
24090, 24096, 24102, 24117, 24124,  
24134, 24139, 24142, 24145, 24159,  
24162, 24165, 24179, 24182, 24185,  
24196, 24199, 24202, 24217, 24220,  
24223, 24232, 24246, 24249, 24252,  
24258, 24264, 24276, 24314, 24317,  
24322, 24326, 24378, 24387, 24396,  
24405, 24418, 24431, 24444, 24450,  
24456, 24484, 24497, 24510, 24511,  
24512, 24519, 24526, 24555, 24556,  
24557, 24569, 24594, 24604, 24611,  
24618, 24621, 24624, 24636, 24639,  
24642, 24654, 24660, 24666, 24672,  
24674, 24676, 24682, 24703, 24705,  
24707, 24713, 24747, 24762, 24845,  
24859, 24865, 24871, 24910, 24916,  
24922, 24943, 24952, 24971, 24976,  
24990, 24995, 25008, 25019, 25031,  
25045, 25105, 25152, 25160, 25189,  
25238, 25246, 25270, 25273, 25276,  
25321, 25328, 25331, 25333, 25335,  
25337, 25339, 25354, 25355, 25394,

25681, 25688, 25691, 25694, 25697,  
25700, 25745, 25748, 25751, 25808,  
25810, 25812, 25855, 25873, 25879,  
25885, 25897, 25916, 25925, 25936,  
25942, 25947, 25955, 25968, 25975,  
25982, 25994, 26016, 26019, 26022,  
26037, 26044, 26050, 26059, 26066,  
26078, 26080, 26082, 26084, 26135,  
26141, 26148, 26203, 26205, 26207,  
26227, 26233, 26236, 26304, 26306,  
26308, 26314, 26316, 26318, 26324,  
26326, 26328, 26334, 26336, 26338,  
26349, 26350, 26351, 26371, 26390,  
26402, 26530, 26534, 26553, 26564,  
26587, 26629, 26636, 26642, 26649,  
26709, 26713, 26722, 26733, 27759,  
27762, 27764, 27766, 27778, 27784,  
27790, 27796, 27801, 27818, 27819,  
27822, 27826, 27829, 27840, 27842,  
27854, 27859, 27864, 27909, 27926,  
27931, 27938, 27943, 27945, 27948,  
27950, 27952, 27954, 27971, 27984,  
27989, 28021, 28026, 28046, 28052,  
28061, 28063, 28068, 28070, 28079,  
28084, 28094, 28122, 28133, 28138,  
28140, 28142, 28152, 28154, 28174,  
28177, 28183, 28188, 28190, 28192,  
28214, 28216, 28231, 28246, 28250,  
28256, 28261, 28263, 28265, 28273,  
28281, 28290, 28300, 28302, 28305,  
28307, 28321, 28326, 28347, 28369,  
28372, 28385, 28398, 28403, 28405,  
28407, 28409, 28411, 28413, 28415,  
28417, 28426, 28435, 28437, 28439,  
28444, 28449, 28452, 28458, 28470,  
28492, 28502, 28522, 28530, 28538,  
28545, 28554, 28555, 28556, 28587,  
28602, 28604, 28630, 28643, 28658,  
28670, 28681, 28709, 28742, 28778,  
28780, 28791, 28794, 28796, 28798,  
28813, 28815, 28828, 28842, 28849,  
28857, 28861, 28865, 28868, 28883,  
28893, 28928, 28938, 28943, 28945,  
28953, 28961, 28963, 28965, 28974,  
28976, 28977, 28981, 28988, 28994,  
29000, 29003, 29010, 29019, 29024,  
29032, 29043, 29045, 29049, 29054,  
29059, 29069, 29078, 29080, 29083,  
29085, 29087, 29089, 29094, 29099,  
29104, 29106, 29119, 29124, 29126,  
29128, 29130, 29132, 29134, 29136,  
29138, 29147, 29156, 29158, 29160,  
29165, 29177, 29192, 29217, 29229,  
29241, 29253, 29260, 29283, 29286,  
29288, 29290, 29293, 29346, 29359,  
29380, 29393, 29395, 29396, 29397,  
29398, 29401, 29406, 29408, 29416,  
29426, 29434, 29439, 29444, 29455,  
29465, 29475, 29477, 29479, 29481,  
29512, 29514, 29519, 29521, 29523,  
29526, 29547, 29558, 29571, 29573,  
29575, 29577, 29579, 29581, 29583,  
29585, 29587, 29598, 29606, 29609,  
29619, 29635, 29650, 29672, 29786  
\cs\_new\_protected:Npx . . . . .  
... 10, 367, 367, 372, 2440, 2851,  
2887, 2892, 2983, 3047, 3772, 3776,  
3781, 3953, 3957, 3958, 4047, 5092,  
8453, 8974, 8989, 8994, 8999, 9588,  
9590, 9592, 9594, 9596, 9605, 9607,  
9609, 9891, 9893, 9895, 9897, 9899,  
9908, 9910, 9912, 21210, 21224,  
21226, 28156, 28165, 28559, 28567,  
28573, 28579, 28720, 28771, 28782  
\cs\_new\_protected\_nopar:Nn . . . . .  
... 12, 2981, 3045  
\cs\_new\_protected\_nopar:Npn . . . . .  
... 10, 2851, 2868, 2881, 2885  
\cs\_new\_protected\_nopar:Npx . . . . .  
... 10, 2851, 2881, 2886  
\cs\_set:Nn . . . . . 12, 344, 2981, 3045  
\cs\_set:Npn 9, 10, 95, 95, 313, 332,  
341, 344, 524, 2106, 2136, 2143,  
2145, 2148, 2149, 2150, 2151, 2152,  
2153, 2154, 2155, 2156, 2157, 2158,  
2159, 2160, 2161, 2162, 2163, 2164,  
2165, 2166, 2167, 2169, 2170, 2171,  
2172, 2173, 2174, 2175, 2176, 2177,  
2253, 2350, 2359, 2361, 2459, 2493,  
2499, 2512, 2514, 2517, 2534, 2583,  
2586, 2648, 2698, 2702, 2706, 2710,  
2715, 2721, 2722, 2726, 2733, 2736,  
2796, 2798, 2800, 2802, 2804, 2806,  
2808, 2810, 2829, 2851, 2867, 2875,  
2875, 2981, 3045, 4304, 4358, 4473,  
4643, 5140, 5176, 6407, 6420, 8008,  
8097, 8503, 8885, 9130, 9386, 9388,  
10700, 11035, 12095, 13554, 13616,  
13624, 13633, 13650, 13658, 13686,  
19562, 20608, 20609, 20610, 20927,  
20928, 21494, 21496, 21513, 21515,  
21808, 21835, 21874, 22413, 22721,  
25367, 27418, 27960, 27962, 27968  
\cs\_set:Npx . . . . . 10, 349,  
613, 2106, 2875, 2876, 4360, 8084,  
8883, 8902, 8908, 10758, 10759,  
10760, 10761, 10762, 12080, 12088,  
12349, 20350, 20356, 22049, 25546

\cs\_set\_eq:NN 14, 96, 325, 341, 470,  
 2242, 2250, 2650, 2893, 3776, 3794,  
 3957, 4093, 4103, 5174, 5175, 6018,  
 6019, 6021, 6128, 6129, 6140, 7312,  
 7386, 7389, 8456, 8649, 8879, 8900,  
 8907, 10764, 10765, 10766, 10767,  
 10769, 10771, 10772, 12344, 12359,  
 12363, 12391, 12402, 12412, 19824,  
 19825, 19830, 19981, 20024, 20049,  
 21800, 21809, 21832, 22419, 25545,  
 26360, 26769, 27957, 27967, 29827  
\cs\_set\_nopar:Nn . . . . . 12, 2981, 3045  
\cs\_set\_nopar:Npn . 9, 10, 122, 323,  
 340, 2106, 2135, 2238, 2399, 2400,  
 2867, 2869, 12301, 12370, 25572, 25574  
\cs\_set\_nopar:Npx . . . . .  
 . . . . . 10, 1067, 2106, 2139, 2867,  
 2870, 3181, 3416, 4141, 4144, 4147,  
 4165, 4168, 4171, 4174, 4197, 4200,  
 4203, 4206, 27760, 27780, 27786, 27823  
\cs\_set\_protected:Nn . 12, 2981, 3045  
\cs\_set\_protected:Npn . 9, 10, 260,  
 341, 2106, 2122, 2124, 2126, 2128,  
 2130, 2132, 2137, 2179, 2182, 2187,  
 2195, 2203, 2213, 2225, 2230, 2239,  
 2244, 2256, 2257, 2261, 2263, 2272,  
 2281, 2286, 2292, 2299, 2301, 2302,  
 2303, 2304, 2305, 2307, 2318, 2320,  
 2332, 2348, 2358, 2379, 2381, 2384,  
 2385, 2395, 2397, 2401, 2408, 2412,  
 2428, 2438, 2449, 2464, 2467, 2469,  
 2471, 2473, 2477, 2478, 2480, 2482,  
 2486, 2488, 2491, 2497, 2503, 2507,  
 2508, 2516, 2518, 2520, 2521, 2522,  
 2524, 2533, 2535, 2537, 2538, 2539,  
 2541, 2550, 2562, 2588, 2605, 2624,  
 2632, 2640, 2649, 2651, 2653, 2665,  
 2679, 2724, 2812, 2825, 2827, 2831,  
 2833, 2840, 2848, 2853, 2887, 2887,  
 2921, 2942, 4480, 5061, 5088, 6045,  
 6553, 8445, 8567, 8737, 8755, 9501,  
 9574, 9878, 9880, 10250, 10362,  
 10430, 10680, 10682, 10821, 10902,  
 11002, 11031, 11261, 11386, 11775,  
 11817, 11911, 12338, 13571, 14056,  
 14133, 14717, 14791, 14805, 15000,  
 15017, 15052, 15086, 15101, 15118,  
 16661, 19826, 21222, 21247, 21256,  
 21785, 21794, 21796, 21798, 21801,  
 21803, 21810, 21812, 21817, 21819,  
 21824, 21826, 21828, 21830, 21833,  
 22582, 22583, 23013, 24489, 24502,  
 24536, 25376, 25566, 25569, 25591,  
 25606, 25624, 25632, 25664, 26095,  
 26109, 26120, 27459, 27470, 27473,  
 27499, 27510, 27526, 27646, 27649,  
 27671, 27907, 28031, 28089, 28718,  
 28740, 29791, 29797, 29816, 29822  
\cs\_set\_protected:Npx . . . . .  
 . . . . . 10, 246, 2106, 2887, 2888, 9753  
\cs\_set\_protected\_nopar:Nn . . . . .  
 . . . . . 13, 2981, 3045  
\cs\_set\_protected\_nopar:Npn . . . . .  
 . . . . . 11, 341, 2106, 2881, 2881  
\cs\_set\_protected\_nopar:Npx . . . . .  
 . . . . . 11, 2106, 2881, 2882  
\cs\_show:N . . . . . 15, 15, 20, 347, 3122  
\cs\_split\_function:N . . . . .  
 . . . . . 16, 2529, 2546, 2658,  
 2659, 2724, 2953, 2994, 3763, 3986  
\cs\_to\_sr:N . . . . . 1070  
\cs\_to\_str:N . . . . . 4, 16, 42, 51,  
 335, 336, 336, 366, 421, 2319, 2715,  
 2730, 3733, 5671, 5672, 5673, 5674,  
 5675, 5676, 5677, 5678, 5679, 5680,  
 5681, 5682, 10685, 13569, 20364,  
 21766, 27754, 27765, 27857, 27862,  
 27870, 29804, 29808, 29825, 29828  
\cs\_undefine:N . . . . .  
 . . . . . 14, 467, 614, 2909, 9916,  
 9917, 9918, 10432, 19301, 25350, 25351

\_cs_generate_variant:w . . . . .	3981	\csname . . . . .	14, 21, 39, 43, 49, 68, 90,
\_cs_generate_variant:ww . . . . .	3772	92, 93, 94, 105, 130, 153, 157, 228, 327	
\_cs_generate_variant:wwNN . . . . .	369, 369, 370, 3811, 3929	\csstring . . . . .	911
\_cs_generate_variant:wwNw . . . . .	3772	\currentgrouplevel . . . . .	619, 1432
\_cs_generate_variant_F_- form:nnn . . . . .	3981	\currentgroupype . . . . .	620, 1433
\_cs_generate_variant_loop:nNwN . . . . .	369, 369, 3812, 3824	\currentifbranch . . . . .	621, 1434
\_cs_generate_variant_loop_- base:N . . . . .	3824	\currentiflevel . . . . .	622, 1435
\_cs_generate_variant_loop_- end:nnnnNNnn . . . . .	369, 370, 3814, 3824	\currentiftipe . . . . .	623, 1436
\_cs_generate_variant_loop_- invalid>NNNNnn . . . . .	369, 3824		D
\_cs_generate_variant_loop_- long:wNNnn . . . . .	370, 3817, 3824	\day . . . . .	328
\_cs_generate_variant_loop_- same:w . . . . .	369, 3824	dd . . . . .	201
\_cs_generate_variant_loop_- special>NNNNnn . . . . .	3824, 3924	\deadcycles . . . . .	329
\_cs_generate_variant_p_- form:nnn . . . . .	3981	debug commands:	
\_cs_generate_variant_same:N . . . . .	369, 3869, 3918	\debug_off: . . . . .	311
\_cs_generate_variant_T_- form:nnn . . . . .	3981	\debug_off:n . . . . .	242, 480, 2185
\_cs_generate_variant_TF_- form:nnn . . . . .	3981	\debug_on: . . . . .	311
\_cs_get_function_name:N . . . . .	335	\debug_on:n . . . . .	242, 298, 480, 2185
\_cs_get_function_signature:N . . . . .	335	\debug_resume: . . . . .	242, 321, 984, 2236, 24415
\_cs_parm_from_arg_count_- test:nnTF . . . . .	2921	\debug_suspend: . . . . .	242, 321, 984, 2236, 24408
\_cs_split_function_auxi:w . . . . .	2724		
\_cs_split_function_auxii:w . . . . .	2724	debug internal commands:	
\_cs_tmp:w . . . . .	335, 367, 372, 2724, 2739, 2851, 2867, 2869, 2870, 2871, 2872, 2873, 2874, 2875, 2876, 2877, 2878, 2879, 2880, 2881, 2882, 2883, 2884, 2885, 2886, 2887, 2888, 2889, 2890, 2891, 2892, 2981, 3021, 3022, 3023, 3024, 3025, 3026, 3027, 3028, 3029, 3030, 3031, 3032, 3033, 3034, 3035, 3036, 3037, 3038, 3039, 3040, 3041, 3042, 3043, 3044, 3045, 3053, 3054, 3055, 3056, 3057, 3058, 3059, 3060, 3061, 3062, 3063, 3064, 3065, 3066, 3067, 3068, 3069, 3070, 3071, 3072, 3073, 3074, 3075, 3076, 3776, 3794, 3949, 3957	\_debug_all_off: . . . . .	2185
\_cs_to_str:N . . . . .	335, 2715	\_debug_all_on: . . . . .	2185
\_cs_to_str:w . . . . .	335, 2715	\_debug_check-declarations_off: . . . . .	2259
csc . . . . .	198	\_debug_check-declarations_on: . . . . .	2259
cscd . . . . .	199	\_debug_check-expressions_off: . . . . .	2346
		\_debug_check-expressions_on: . . . . .	2346
		\_debug_chk_expr_aux:nNnN . . . . .	2346
		\_debug_chk_var_scope_aux:NN . . . . .	2284, 2290, 2296, 2316
		\_debug_chk_var_scope_aux:Nn . . . . .	2316
		\_debug_chk_var_scope_aux:NNn . . . . .	323, 2316
		\_debug_deprecation_aux:nnNnn . . . . .	2410
		\_debug_deprecation_expandable:nnNnn . . . . .	2426, 2449
		\_debug_deprecation_off: . . . . .	2393
		\g_debug_deprecation_off_tl . . . . .	326, 2393, 2435, 2456
		\_debug_deprecation_on: . . . . .	2393
		\g_debug_deprecation_on_tl . . . . .	326, 2393, 2430, 2451
		\_debug_log-functions_off: . . . . .	2377
		\_debug_log-functions_on: . . . . .	2377
		\_debug_patch_args_aux:nnnn . . . . .	2480
		\_debug_patch_args_aux:nnnNNnn . . . . .	2480
		\_debug_patch_args_aux:nnnNNnnn . . . . .	2480
		\_debug_patch_aux:nnnn . . . . .	2465
		\_debug_patch_auxii:nnnn . . . . .	2465

```

\__debug_suspended:TF ..... 322, 2236, 2265,
..... 2274, 2283, 2288, 2294, 2352, 2382
\l__debug_suspended_tl ..... 2236
\__debug_tmp:w ..... 2480
\def ..... 74,
..... 75, 76, 112, 129, 131, 132, 150, 151,
..... 154, 170, 185, 213, 217, 242, 281, 330
default commands:
    .default:n ..... 172, 12490
\defaulthyphenchar ..... 331
\defaultskewchar ..... 332
deg ..... 201
\delcode ..... 333
\delimiter ..... 334
\delimiterfactor ..... 335
\delimitershortfall ..... 336
deprecation internal commands:
    \__deprecation_primitive:NN .....
..... 1120, 29786
    \__deprecation_primitive:w ... 29786
\detokenize ..... 68, 228, 624, 1437
\DH ..... 27683
\dh ..... 27683
dim commands:
    \dim_abs:n ..... 155, 1037, 11432
    \dim_add:Nn ..... 155, 11410
    \dim_case:nn ..... 158, 11530
    \dim_case:nnn ..... 29696
    \dim_case:nnTF .....
..... 158, 11530, 11535, 11540, 29697
    \dim_compare:nNnTF ..... 156,
..... 158, 158, 158, 159, 11479, 11554,
..... 11590, 11598, 11607, 11613, 11640,
..... 11643, 11654, 24765, 24768, 24771,
..... 24780, 24783, 24786, 24795, 24802,
..... 24877, 24882, 24892, 25021, 25033,
..... 25708, 25725, 25759, 25773, 25783
    \dim_compare:nTF 157, 159, 159, 159,
..... 159, 11490, 11562, 11570, 11579, 11585
    \dim_compare_p:n ..... 157, 11490
    \dim_compare_p:nNn ..... 156, 11479
    \dim_const:Nn ..... 154,
..... 585, 595, 11362, 11737, 11738, 12945
    \dim_do_until:nn ..... 159, 11560
    \dim_do_until:nNn ..... 158, 11588
    \dim_do_while:nn ..... 159, 11560
    \dim_do_while:nNn ..... 158, 11588
    \dim_eval:n .....
..... 156, 157, 160, 160, 585, 962,
..... 11366, 11533, 11538, 11543, 11548,
..... 11658, 11686, 11732, 11736, 24475,
..... 24545, 24631, 24649, 24686, 24690,
..... 24691, 24695, 24699, 24700, 24717,
..... 24722, 24728, 24735, 24742, 25004,
..... 25005, 25283, 25284, 25285, 25833,
..... 25858, 25861, 25862, 25869, 25951,
..... 25952, 25959, 25960, 26063, 26070
\dim_gadd:Nn ..... 155, 11410
.dim_gset:N ..... 172, 12498
\dim_gset:Nn ..... 155, 585, 11394
\dim_gset_eq:NN ..... 155, 11402
\dim_gsub:Nn ..... 155, 11410
\dim_gzero:N ..... 154, 11369, 11379
\dim_gzero_new:N ..... 154, 11376
\dim_if_exist:NTF .....
..... 154, 11377, 11379, 11382
\dim_if_exist_p:N ..... 154, 11382
\dim_log:N ..... 162, 11733
\dim_log:n ..... 162, 11733
\dim_max:nn .. 155, 11432, 25930, 25934
\dim_min:nn .....
..... 155, 11432, 25928, 25932, 25945
\dim_new:N .....
..... 154, 154, 11354, 11365, 11377,
..... 11379, 11739, 11740, 11741, 11742,
..... 23963, 23964, 23965, 23966, 23967,
..... 23968, 23969, 23970, 24333, 24357,
..... 24358, 24361, 24362, 24363, 24364,
..... 25093, 25095, 25096, 25803, 25804,
..... 25805, 25806, 25807, 25966, 25967
\dim_ratio:nn .. 156, 594, 11475, 11726
.dim_set:N ..... 172, 12498
\dim_set:Nn ..... 155, 11394, 23990,
..... 23991, 23992, 24024, 24035, 24119,
..... 24120, 24121, 24136, 24234, 24235,
..... 24236, 24238, 24240, 24242, 24462,
..... 24531, 24767, 24772, 24782, 24787,
..... 24797, 24804, 24817, 24848, 24880,
..... 24955, 24956, 24958, 24960, 24978,
..... 24979, 25094, 25197, 25198, 25249,
..... 25250, 25251, 25253, 25864, 25899,
..... 25907, 25918, 25919, 25920, 25921,
..... 25927, 25929, 25931, 25933, 25938,
..... 25944, 26027, 26029, 26031, 26039,
..... 26041, 28703, 28704, 29027, 29028
\dim_set_eq:NN .....
..... 155, 11402, 24464, 24465, 24533, 24534
\dim_show:N ..... 161, 11729
\dim_show:n ..... 162, 595, 11731
\dim_sign:n ..... 251, 26431
\dim_step_function:nnnn .....
..... 159, 592, 11616, 11683
\dim_step_inline:nnnn ... 159, 11661
\dim_step_variable:nnnNn .. 160, 11661
\dim_sub:Nn ..... 155, 11410
\dim_to_decimal:n 160, 11693, 11711,
..... 11723, 29304, 29305, 29306, 29307,

```

29308, 29310, 29385, 29386, 29437,  
 29442, 29448, 29449, 29450, 29451,  
 29460, 29461, 29462, 29553, 29572  
`\dim_to_decimal_in_bp:n` . . . . .  
     . . . . . 161, 161, 11710, 28203,  
 28204, 28205, 28269, 28270, 28277,  
 28278, 28285, 28286, 28294, 28295,  
 28296, 28393, 28397, 28401, 28593,  
 28594, 28595, 28804, 28805, 28806,  
 28907, 28908, 28909, 28910, 29052,  
 29057, 29063, 29064, 29065, 29073,  
 29074, 29114, 29118, 29122, 29557  
`\dim_to_decimal_in_sp:n` . . . . . 161,  
     161, 677, 11712, 14196, 14233, 14830  
`\dim_to_decimal_in_unit:nn` 161, 11721  
`\dim_to_fp:n` . . . . .  
     . . . . . 161, 677, 696, 11729, 18915,  
 24028, 24029, 24039, 24040, 24108,  
 24111, 24112, 24137, 24152, 24153,  
 24172, 24173, 24191, 24208, 24211,  
 24212, 24810, 24812, 24822, 24824,  
 24826, 24827, 24852, 24853, 24854,  
 24855, 25903, 25904, 25911, 25912,  
 25985, 25988, 25989, 26040, 26042  
`\dim_until_do:nn` . . . . . 159, 11560  
`\dim_until_do:nNnn` . . . . . 158, 11588  
`\dim_use:N` . . . . .  
     160, 160, 962, 11437, 11448, 11449,  
 11450, 11461, 11462, 11463, 11493,  
 11512, 11689, 11690, 11698, 24986,  
 25866, 25870, 25877, 25883, 25892,  
 25893, 25894, 26048, 26055, 26436  
`\dim_while_do:nn` . . . . . 159, 11560  
`\dim_while_do:nNnn` . . . . . 159, 11588  
`\dim_zero:N` 154, 154, 11369, 11377,  
 23993, 24122, 24237, 24758, 24759  
`\dim_zero_new:N` . . . . . 154, 11376  
`\c_max_dim` . . . . . 162, 165, 629,  
 11737, 11873, 12973, 13015, 13023,  
 25918, 25919, 25920, 25921, 25938  
`\g_tmpa_dim` . . . . . 162, 11739  
`\l_tmpa_dim` . . . . . 162, 11739  
`\g_tmpb_dim` . . . . . 162, 11739  
`\l_tmpb_dim` . . . . . 162, 11739  
`\c_zero_dim` . . . . . 162, 11640,  
 11643, 11737, 11872, 13040, 23839,  
 23861, 24307, 24765, 24768, 24771,  
 24780, 24783, 24786, 24795, 24802,  
 24877, 24882, 24892, 25712, 25723,  
 25729, 25741, 25759, 25763, 25771,  
 25773, 25777, 25783, 25793, 26441  
**dim internal commands:**  
     `\__dim_abs:N` . . . . . 11432  
     `\__dim_case:nnTF` . . . . . 11530  
     `\__dim_case:nw` . . . . . 11530  
     `\__dim_case_end:nw` . . . . . 11530  
     `\__dim_compare:w` . . . . . 11490  
     `\__dim_compare:wNN` . . . . . 588, 11490  
     `\__dim_compare_!w` . . . . . 11490  
     `\__dim_compare_<:w` . . . . . 11490  
     `\__dim_compare_=:w` . . . . . 11490  
     `\__dim_compare_>:w` . . . . . 11490  
     `\__dim_compare_end:w` . . . . . 11498, 11522  
     `\__dim_compare_error:` . . . . . 588, 11490  
     `\__dim_eval:w` . . . . . 594, 11351,  
         11391, 11396, 11399, 11412, 11417,  
         11423, 11428, 11433, 11437, 11443,  
         11444, 11448, 11449, 11450, 11456,  
         11457, 11461, 11462, 11463, 11478,  
         11481, 11483, 11487, 11493, 11512,  
         11517, 11619, 11623, 11627, 11634,  
         11635, 11636, 11687, 11689, 11694,  
         11698, 11715, 11720, 26432, 26436  
     `\__dim_eval_end:` . . . . . 11351, 11396,  
         11399, 11412, 11417, 11423, 11428,  
         11437, 11452, 11465, 11478, 11482,  
         11487, 11689, 11698, 11720, 26436  
     `\__dim_maxmin:wwN` . . . . . 11432  
     `\__dim_ratio:n` . . . . . 11475  
     `\__dim_sign:Nw` . . . . . 26431  
     `\__dim_step:NnnnN` . . . . . 11616  
     `\__dim_step:Nnnnn` . . . . . 11661  
     `\__dim_step:wwwN` . . . . . 11616  
     `\__dim_tmp:w` 586, 586, 11386, 11394,  
         11397, 11410, 11413, 11421, 11424  
     `\__dim_to_decimal:w` . . . . . 11693  
`\dimen` . . . . . 337, 8796  
`\dimendef` . . . . . 338  
`\dimexpr` . . . . . 625, 1438  
`\directlua` . . . . . 16, 23, 53, 59, 61, 912, 1733  
`\disablecjktoken` . . . . . 1251, 2033  
`\discretionary` . . . . . 339  
`\displayindent` . . . . . 340  
`\displaylimits` . . . . . 341  
`\displaystyle` . . . . . 342  
`\displaywidowpenalties` . . . . . 626, 1439  
`\displaywidowpenalty` . . . . . 343  
`\displaywidth` . . . . . 344  
`\divide` . . . . . 345  
`\DJ` . . . . . 27684  
`\dj` . . . . . 27684  
`\do` . . . . . 1291  
`\doublehyphendemerits` . . . . . 346  
`\dp` . . . . . 347  
`\draftmode` . . . . . 1014, 1623  
**driver commands:**  
     `\driver_box_use_clip:N` 261, 25689,  
         25692, 28192, 28587, 28798, 29293

```

\driver_box_use_rotate:Nn . . . .
    262, 24011, 28214, 28602, 28813, 29346
\driver_box_use_scale:Nnn . . . .
    262, 24280, 28231, 28630, 28828, 29359
\driver_color_cmyk:nmmn . . . .
    ... 262, 25334, 28052, 28122, 29597
\driver_color_gray:n . . . .
    ... 262, 25336, 28052, 28122, 29608
\driver_color_pickup:N . . . .
    ... 262, 25324, 28025, 28083
\driver_color_rgb:nnn . . . .
    ... 262, 25338, 28052, 28122, 29634
\driver_color_spot:nn . . . .
    ... 25340, 28052, 28122
\driver_draw_begin: . . . .
    ... 263, 28250, 29043, 29401
\driver_draw_box:Nnnnnnn . . . .
    266
\driver_draw_box_use:Nnnnn . . .
    ... 266, 1081, 28470, 29260, 29650
\driver_draw_cap_butt: . . . .
    ... 265, 265, 28385, 29106, 29547
\driver_draw_cap_rectangle: . . .
    ... 265, 28385, 29106, 29547
\driver_draw_cap_round: . . . .
    ... 265, 28385, 29106, 29547
\driver_draw_clip: . . . .
    ... 264, 28305, 29083, 29479
\driver_draw_closepath: . . . .
    ... 263, 28305, 29083, 29479
\driver_draw_closestroke: . . . .
    ... 264, 28305, 29083, 29479
\driver_draw_cm:nnnn . . . .
    ... 266, 28458, 28478, 28479,
        28480, 29177, 29264, 29635, 29653
\driver_draw_color_fill_cmyk:nnnn
    ... 265, 28417, 29138, 29587
\driver_draw_color_fill_gray:n ..
    ... 265, 28417, 29138, 29587
\driver_draw_color_fill_rgb:nnn .
    ... 265, 28417, 29138, 29587
\driver_draw_color_gray:n . 29598
\driver_draw_color_rgb:nnn . 29609
\driver_draw_color_stroke_-
    cmyk:nnnn 265, 28417, 29138, 29587
\driver_draw_color_stroke_gray:n
    ... 265, 28417, 29138, 29587
\driver_draw_color_stroke_-
    rgb:nnn . 265, 28417, 29138, 29587
\driver_draw_curveto:nmmnn . .
    ... 263, 28265, 29049, 29434
\driver_draw_dash:nn . . . .
    265
\driver_draw_dash_pattern:nn . .
    ... 265, 28385, 29106, 29547
\driver_draw_discardpath: . . . .
    ... 264, 28305, 29083, 29479
\driver_draw_end: . . . .
    ... 263, 28250, 29043, 29401
\driver_draw_evenodd_rule: . . .
    ... 264, 28300, 29078, 29475
\driver_draw_fill: . . . .
    ... 264, 28305, 29083, 29479
\driver_draw_fillstroke: . . . .
    ... 264, 28305, 29083, 29479
\driver_draw_join_bevel: . . . .
    ... 265, 28385, 29106, 29547
\driver_draw_join_miter: . . . .
    ... 265, 28385, 29106, 29547
\driver_draw_join_round: . . . .
    ... 265, 28385, 29106, 29547
\driver_draw_lineto:nn . . . .
    ... 263, 28265, 29049, 29434
\driver_draw_linewidth:n . . . .
    ... 265, 28385, 29106, 29547
\driver_draw_miterlimit:n . . . .
    ... 265, 28385, 29106, 29547
\driver_draw_move:nn . . . .
    263
\driver_draw_moveto:nn . . . .
    ... 263, 28265, 29049, 29434
\driver_draw_nonzero_rule: . . .
    ... 264, 28300, 29078, 29475
\driver_draw_rectangle:nnnn . .
    ... 263, 28265, 29049, 29434
\driver_draw_scope_begin: . . . .
    263, 28261, 29044, 29047, 29403, 29408
\driver_draw_scope_end: . . . .
    263, 28261, 29046, 29047, 29407, 29408
\driver_draw_stroke: . . . .
    ... 264, 264, 28305, 29083, 29479
\driver_draw_stroke_cmyk:nnnn . 262
\driver_draw_stroke_gray:n . 262
\driver_draw_stroke_rgb:nnn . 262
\driver_pdf_compresslevel:n . .
    ... 267, 28554, 28771, 28974, 29393
\driver_pdf_object_new:n . 266, 29393
\driver_pdf_object_new:nn . .
    ... 266, 28492, 28717, 28928
\driver_pdf_object_ref:n . .
    ... 266, 28492,
        28506, 28526, 28534, 28542, 28717,
        28928, 28949, 28957, 28969, 29393
\driver_pdf_object_write:nn . .
    ... 267, 28502, 28739, 28938, 29395
\driver_pdf_objects_disable: . .
    ... 267, 28554, 28771, 28974, 29393
\driver_pdf_objects_enable: . .
    ... 267, 28554, 28771, 28974, 29393
\driver_pdf_object_write:nn . 29393

```

driver internal commands:

- \\_\\_driver\\_align\\_currentpoint\\_...  
..... 1086
- \\_\\_driver\\_align\\_currentpoint\\_ -  
begin: .. 28177, 28195, 28219, 28234
- \\_\\_driver\\_align\\_currentpoint\\_ -  
end: .... 28177, 28209, 28227, 28241
- \\_\\_driver\\_box\\_use\\_rotate:Nn ....  
..... 28214, 28602, 28813
- \g\\_\\_driver\\_clip\\_path\\_int ....  
..... 29293, 29485,  
29488, 29501, 29530, 29533, 29541
- \\_\\_driver\\_color\\_cmyk:nnnn .... 28122
- \\_\\_driver\\_color\\_fill\\_select:n . 29138
- \\_\\_driver\\_color\\_gray:n ..... 28122
- \\_\\_driver\\_color\\_pickup:w .....  
..... 1076, 28025, 28083
- \\_\\_driver\\_color\\_reset: .....  
..... 28052, 28122, 28455
- \\_\\_driver\\_color\\_rgb:nnn ..... 28122
- \\_\\_driver\\_color\\_select:n . 28052,  
28122, 29149, 29159, 29167, 29171
- \l\\_\\_driver\\_color\\_stack\\_int ....  
..... 28121, 28161, 28170
- \l\\_\\_driver\\_cos\\_fp ..... 28602
- \\_\\_driver\\_draw\\_add\\_to\\_path:n ...  
..... 29434, 29480
- \g\\_\\_driver\\_draw\\_clip\\_bool ....  
..... 28305, 29479
- \\_\\_driver\\_draw\\_cm:nnnn .... 29177
- \\_\\_driver\\_draw\\_cm\\_decompose:nnnnN  
..... 29187, 29216
- \\_\\_driver\\_draw\\_cm\\_decompose\_-  
auxi:nnnnN ..... 29216
- \\_\\_driver\\_draw\\_cm\\_decompose\_-  
auxii:nnnnN ..... 29216
- \\_\\_driver\\_draw\\_cm\\_decompose\_-  
auxiii:nnnnN ..... 29216
- \\_\\_driver\\_draw\\_color\\_fill:n .. 28417
- \\_\\_driver\\_draw\\_color\\_fill:nmm . 29587
- \\_\\_driver\\_draw\\_color\\_gray\\_aux:n ..  
..... 29602, 29606
- \\_\\_driver\\_draw\\_color\\_stroke:n . 28417
- \\_\\_driver\\_draw\\_dash:n .....  
..... 28385, 29106, 29547
- \\_\\_driver\\_draw\\_dash\\_aux:nn ... 29547
- \g\\_\\_driver\\_draw\\_eor\\_bool . 28300,  
28314, 28332, 28340, 28353, 28362,  
28378, 29078, 29092, 29097, 29102
- \\_\\_driver\\_draw\\_literal:n . 28248,  
28253, 28254, 28258, 28262, 28264,  
28267, 28275, 28283, 28292, 28306,  
28309, 28312, 28318, 28328, 28329,  
28330, 28335, 28338, 28344, 28349,
- 28350, 28351, 28356, 28357, 28360,  
28366, 28376, 28382, 28387, 28400,  
28404, 28406, 28408, 28410, 28412,  
28414, 28416, 28450, 28460, 28472,  
28473, 28474, 28475, 28476, 28477,  
28481, 28482, 28484, 28485, 28486,  
28487, 28488, 29041, 29051, 29056,  
29061, 29071, 29084, 29086, 29088,  
29091, 29096, 29101, 29105, 29108,  
29121, 29125, 29127, 29129, 29131,  
29133, 29135, 29137, 29174, 29399,  
29420, 29428, 29486, 29505, 29531
- \\_\\_driver\\_draw\\_path:n ..... 29479
- \g\\_\\_driver\\_draw\\_path\\_int ..... 29479
- \g\\_\\_driver\\_draw\\_path\\_tl .. 29434,  
29490, 29506, 29508, 29535, 29544
- \\_\\_driver\\_draw\\_scope:n .....  
..... 29404, 29408,  
29476, 29478, 29498, 29538, 29560,  
29572, 29574, 29576, 29578, 29580,  
29582, 29584, 29586, 29621, 29637
- \g\\_\\_driver\\_draw\\_scope\\_int .... 29408
- \l\\_\\_driver\\_draw\\_scope\\_int .... 29408
- \\_\\_driver\\_exp\\_not\\_i:nn ..... 28739
- \\_\\_driver\\_exp\\_not\\_ii:nn ..... 28739
- \l\\_\\_driver\\_image\\_attr\\_tl .....  
..... 28642, 28647, 28654,  
28662, 28672, 28705, 28707, 28712
- \\_\\_driver\\_image\\_getbb\\_aux:nnn 29001
- \\_\\_driver\\_image\\_getbb\\_auxi:n . 28643
- \\_\\_driver\\_image\\_getbb\\_auxi:nN . 28981
- \\_\\_driver\\_image\\_getbb\\_auxii:n . 28643
- \\_\\_driver\\_image\\_getbb\\_auxii:nnN .  
..... 28981
- \\_\\_driver\\_image\\_getbb\\_auxiii:nnn  
..... 28981
- \\_\\_driver\\_image\\_getbb\\_auxiv:nnNn ..  
..... 28981, 29037
- \\_\\_driver\\_image\\_getbb\\_auxv:nnNn ..  
..... 28981
- \\_\\_driver\\_image\\_getbb\\_auxvi:nnn ..  
..... 29022, 29024
- \\_\\_driver\\_image\\_getbb\\_eps:n ....  
..... 28245, 28840
- \\_\\_driver\\_image\\_getbb\\_jpg:n ..  
..... 28643, 28840, 28981, 29378
- \\_\\_driver\\_image\\_getbb\\_pagebox:w ..  
..... 28981
- \\_\\_driver\\_image\\_getbb\\_pdf:n ....  
..... 28643, 28840, 28981
- \\_\\_driver\\_image\\_getbb\\_png:n ....  
..... 28643, 28840, 28981, 29378
- \\_\\_driver\\_image\\_include\\_auxi:nn ..  
..... 28857

```

\__driver_image_include_auxii:nnn
    ..... 28857
\__driver_image_include_auxiii:nn
    ..... 28857
\__driver_image_include_auxiii:nmn
    ..... 28893
\__driver_image_include_bitmap_-
    quote:w ..... 29380
\__driver_image_include_eps:n ...
    ..... 28246, 28857
\__driver_image_include_jpg:n ...
    ..... 28709, 28857, 29380
\__driver_image_include_pdf:n ...
    ..... 28709, 28857, 29032
\__driver_image_include_png:n ...
    ..... 28709, 28857, 29380
\g__driver_image_int ...
    ..... 28856, 28895, 28896
\__driver_literal:e ..... 28020
\__driver_literal:n 28020, 28072,
    28080, 28175, 28179, 28186, 28189,
    28191, 28247, 28252, 28259, 28454,
    28792, 28795, 28797, 28818, 28831,
    28859, 28887, 28897, 28947, 28955,
    28967, 28975, 28978, 29194, 29201,
    29207, 29267, 29277, 29284, 29382
\__driver_literal_pdf:n ...
    .. 28559, 28590, 28791, 28801, 29041
\__driver_literal_postscript:n ...
    ..... 28074, 28174,
    28180, 28181, 28185, 28196, 28197,
    28199, 28200, 28208, 28220, 28235,
    28248, 28504, 28524, 28532, 28547
\__driver_literal_svg:n ...
    29283, 29287, 29289, 29291, 29296,
    29298, 29315, 29399, 29654, 29665
\__driver_matrix:n ...
    ..... 28579, 28612, 28633, 29180
\g__driver_path_int ... 29494, 29511
\g__driver_pdf_object_int 28490,
    28494, 28497, 28926, 28930, 28933
\g__driver_pdf_object_prop ...
    28490, 28498, 28509, 28519, 28716,
    28726, 28748, 28926, 28934, 28941
\__driver_pdf_object_write:nnn 28938
\__driver_pdf_object_write_-
    array:nn ..... 28502, 28938
\__driver_pdf_object_write_-
    dict:nn ..... 28502, 28938
\__driver_pdf_object_write_-
    fstream:nn ..... 28938
\__driver_pdf_object_write_-
    stream:nn ..... 28502, 28938
\__driver_pdf_object_write_-
    stream:nnn ..... 28502
\__driver_scope_begin: 1108, 28188,
    28194, 28218, 28233, 28567, 28589,
    28606, 28632, 28794, 28800, 28817,
    28830, 29047, 29262, 29286, 29652
\__driver_scope_begin:n ... 29290,
    29317, 29325, 29330, 29348, 29361
\__driver_scope_end: ..... 28188,
    28211, 28229, 28243, 28567, 28599,
    28626, 28640, 28794, 28810, 28826,
    28838, 29048, 29279, 29286, 29340,
    29341, 29342, 29357, 29376, 29666
\l__driver_sin_fp ..... 28602
\__driver_tmp:w ..... 28718,
    28731, 28735, 28740, 28766, 28767
\l__driver_tmp_box ..... 28701,
    28703, 28704, 29026, 29027, 29028
\dtou ..... 1213, 1995
\dump ..... 348
\divextension ..... 913, 1734
\divfeedback ..... 914, 1735
\divvariable ..... 915, 1736

E
\edef ..... 4, 113, 138, 215, 349
\efcode ..... 793, 1607
\elapsedtime ..... 880
\else ..... 15, 22, 44, 46, 91, 95, 98,
    101, 102, 106, 107, 168, 172, 187, 350
else commands:
\else: .... 20, 91, 91, 92, 96, 103,
    103, 151, 168, 231, 231, 231, 329,
    330, 337, 367, 387, 399, 399, 475,
    720, 2049, 2093, 2311, 2326, 2335,
    2338, 2388, 2416, 2419, 2576, 2584,
    2610, 2744, 2747, 2756, 2762, 2772,
    2775, 2784, 2790, 2915, 2937, 2946,
    2960, 3018, 3019, 3080, 3203, 3517,
    3628, 3656, 3671, 3679, 3716, 3777,
    3828, 3829, 3831, 3835, 3847, 3848,
    3849, 3850, 3851, 3852, 3853, 3854,
    3855, 3920, 3921, 3923, 4008, 4400,
    4410, 4421, 4436, 4444, 4459, 4504,
    4770, 4799, 4820, 4836, 4844, 4854,
    4867, 4883, 4954, 4966, 5015, 5018,
    5021, 5198, 5205, 5211, 5447, 5503,
    5506, 5509, 5521, 5536, 5731, 5769,
    5777, 5788, 5798, 6034, 6065, 6074,
    6401, 6432, 6453, 6456, 6482, 6527,
    6649, 6682, 6722, 6732, 7048, 7081,
```

7132, 7249, 7315, 7356, 7366, 7422,  
 7454, 7474, 7496, 7514, 7530, 7540,  
 7556, 7566, 7658, 7660, 7662, 7664,  
 7903, 7918, 7940, 7954, 8475, 8478,  
 8486, 8492, 8533, 8540, 8631, 8636,  
 8641, 8646, 8653, 8660, 8665, 8670,  
 8675, 8680, 8685, 8690, 8695, 8700,  
 8722, 8728, 8731, 8766, 8769, 8833,  
 8842, 8850, 8859, 8892, 8931, 8945,  
 8954, 8964, 9284, 10150, 10153,  
 10154, 10493, 11440, 11471, 11488,  
 11498, 11523, 12017, 12041, 12059,  
 12070, 12082, 12094, 12110, 12979,  
 12983, 13226, 13236, 13237, 13252,  
 13262, 13357, 13433, 13495, 13498,  
 13512, 13530, 13534, 13799, 13812,  
 13832, 13860, 13861, 13883, 13904,  
 13927, 13928, 13970, 13988, 14023,  
 14027, 14062, 14079, 14085, 14089,  
 14093, 14254, 14287, 14295, 14328,  
 14332, 14344, 14354, 14364, 14395,  
 14408, 14443, 14453, 14472, 14485,  
 14498, 14502, 14513, 14536, 14553,  
 14565, 14579, 14595, 14603, 14605,  
 14615, 14626, 14642, 14658, 14664,  
 14669, 14676, 14698, 14728, 14751,  
 14779, 14782, 14958, 14962, 14969,  
 14989, 15008, 15025, 15031, 15063,  
 15093, 15109, 15129, 15170, 15185,  
 15218, 15220, 15226, 15241, 15294,  
 15455, 15466, 15504, 15507, 15510,  
 15513, 15544, 15553, 15562, 15565,  
 15736, 15749, 15752, 15759, 15777,  
 15801, 15802, 15817, 15827, 15876,  
 15879, 15888, 15900, 15911, 15925,  
 15938, 15978, 16012, 16032, 16069,  
 16087, 16090, 16096, 16110, 16145,  
 16163, 16166, 16169, 16172, 16233,  
 16306, 16376, 16377, 16386, 16421,  
 16504, 16508, 16512, 16574, 16609,  
 16624, 16889, 16918, 16922, 17082,  
 17091, 17143, 17154, 17170, 17178,  
 17237, 17317, 17328, 17333, 17367,  
 17380, 17392, 17398, 17519, 17527,  
 17566, 17573, 17595, 17623, 17638,  
 17642, 17664, 17695, 17698, 17723,  
 17726, 17767, 17775, 17786, 17789,  
 17834, 17849, 17864, 17879, 17894,  
 17909, 17930, 17975, 18281, 18319,  
 18320, 18329, 18373, 18428, 18429,  
 18430, 18534, 18556, 18571, 18589,  
 18637, 18653, 18859, 18927, 18932,  
 19101, 19137, 19150, 19180, 19184,  
 19192, 19219, 19245, 19253, 19270,  
 19273, 19320, 19324, 19375, 19431,  
 19443, 19496, 19497, 19959, 19962,  
 19965, 19975, 19990, 20017, 20032,  
 20059, 20075, 20108, 20116, 20118,  
 20120, 20122, 20124, 20126, 20128,  
 20130, 20148, 20169, 20173, 20245,  
 20249, 20445, 20446, 20451, 20452,  
 20467, 20474, 20681, 20691, 20735,  
 20744, 20756, 20757, 20759, 20761,  
 20764, 20765, 20768, 20769, 20778,  
 20780, 20782, 20785, 20786, 20788,  
 20824, 20827, 20848, 20851, 20859,  
 20867, 20870, 20879, 20882, 20891,  
 20899, 20902, 20912, 21018, 21125,  
 21169, 21173, 21176, 21187, 21192,  
 21291, 21437, 21450, 21539, 21568,  
 21607, 21625, 21733, 21767, 22041,  
 22059, 22078, 22116, 22172, 22219,  
 22223, 22230, 22251, 22262, 22418,  
 22540, 22626, 22687, 22761, 22796,  
 22808, 22834, 22852, 23044, 23188,  
 23213, 23265, 23704, 23706, 23712,  
 25600, 26154, 26171, 26443, 26446,  
 26664, 26784, 26795, 26815, 27922  
`em` ..... 201  
`\emergencystretch` ..... 351  
`\enablecjktoken` ..... 1252, 2034  
`\end` ..... 125, 297, 352, 15633, 20325, 20326  
`end internal commands:`  
`__regex_end` ..... 23032  
`\endcsname` .. 14, 21, 39, 43, 49, 68, 90,  
 92, 93, 94, 105, 130, 153, 157, 228, 353  
`\endgroup` ..... 13, 36,  
 38, 42, 48, 74, 124, 142, 161, 210, 354  
`\endinput` ..... 143, 355  
`\endL` ..... 627, 1441  
`\endlinechar` ..... 227, 240, 356  
`\endR` ..... 628, 1442  
`\enquote` ..... 15635  
`\ensuremath` ..... 27747  
`\epTeXinputencoding` ..... 1214, 1996  
`\epTeXversion` ..... 1215, 1997  
`\eqno` ..... 357  
`\errhelp` ..... 115, 134, 358  
`\errmessage` ..... 123, 135, 359  
`\ERROR` ..... 8581  
`\errorcontextlines` ..... 360  
`\errorstopmode` ..... 361  
`\escapechar` ..... 362  
`\escapehex` ..... 25427  
`\ETC` ..... 20305  
`etex commands:`  
`\etex_beginL:D` .... 1119, 1428, 29786  
`\etex_beginR:D` ..... 1429

\etex_botmarks:D .....	1430	\etex_TeXXeTstate:D .....	1485
\etex_clubpenalties:D .....	1431	\etex_topmarks:D .....	1486
\etex_currentgrouplevel:D .....	1432	\etex_tracingassigns:D .....	1487
\etex_currentgroupype:D .....	1433	\etex_tracinggroups:D .....	1488
\etex_currentifbranch:D .....	1434	\etex_tracingifs:D .....	1489
\etex_currentiflevel:D .....	1435	\etex_tracingnesting:D .....	1490
\etex_currentiftype:D .....	1436	\etex_tracingscantokens:D .....	1491
\etex_detokenize:D .....	1437	\etex_unexpanded:D .....	1492
\etex_dimexpr:D .....	1438	\etex_unless:D .....	1493
\etex_displaywidowpenalties:D .....	1440	\etex_widowpenalties:D .....	1494
\etex_endL:D .....	1441	\eTeXrevision .....	629, 1443
\etex_endR:D .....	1442	\eTeXversion .....	630, 1444
\etex_eTeXrevision:D .....	1443	\etoksapp .....	916, 1737
\etex_eTeXversion:D .....	1444	\etokspre .....	917, 1738
\etex_everyeof:D .....	1445	\euc .....	1216, 1998
\etex_firstmarks:D .....	1446	\everycr .....	363
\etex_fontchardp:D .....	1447	\everydisplay .....	364
\etex_fontcharht:D .....	1448	\everyeof .....	631, 1445
\etex_fontcharic:D .....	1449	\everyhbox .....	365
\etex_fontcharwd:D .....	1450	\everyjob .....	66, 67, 366
\etex_glueexpr:D .....	1451	\everymath .....	367
\etex_glueshrink:D .....	1452	\everypar .....	368
\etex_glueshrinkorder:D .....	1453	\everyvbox .....	369
\etex_gluestretch:D .....	1454	ex .....	201
\etex_gluestretchorder:D .....	1455	\exceptionpenalty .....	918
\etex_gluetomu:D .....	1456	\exhyphenpenalty .....	370
\etex_ifcsname:D .....	1457	exp .....	197
\etex_ifdefined:D .....	1458	exp commands:	
\etex_iffontchar:D .....	1459	\exp:w .....	33,
\etex_interactionmode:D .....	1460	33, 34, 329, 335, 350, 351, 352,	
\etex_interlinepenalties:D .....	1461	360, 395, 395, 401, 415, 478, 484,	
\etex_lastlinefit:D .....	1462	553, 571, 667, 670, 673, 674, 692,	
\etex_lastnodetype:D .....	1463	697, 698, 1048, 1068, 2070, 2513,	
\etex_marks:D .....	1464	2515, 3175, 3188, 3194, 3242, 3246,	
\etex_middle:D .....	1465	3250, 3255, 3261, 3267, 3283, 3295,	
\etex_muexpr:D .....	1466	3301, 3307, 3312, 3314, 3321, 3393,	
\etex_mutoglue:D .....	1467	3398, 3407, 3412, 3421, 3423, 3431,	
\etex_numexpr:D .....	1468	3438, 3444, 3452, 3461, 3468, 3482,	
\etex_pagediscards:D .....	1469	3502, 3506, 3511, 3513, 3550, 3691,	
\etex_parshapedimen:D .....	1470	4511, 4516, 4521, 4526, 4733, 4889,	
\etex_parshapeindent:D .....	1471	4959, 5231, 5236, 5241, 5246, 5262,	
\etex_parshapeLength:D .....	1472	5267, 5272, 5277, 5430, 5439, 5494,	
\etex_predisplaydirection:D .....	1473	6688, 6693, 6698, 6703, 7465, 7611,	
\etex_protected:D .....	1474	7765, 7773, 7832, 8126, 8134, 8466,	
\etex_readline:D .....	1475	10254, 10867, 11497, 11532, 11537,	
\etex_savinghyphcodes:D .....	1476	11542, 11547, 13265, 13380, 13384,	
\etex_savingvdiscards:D .....	1477	13547, 13775, 13901, 13902, 13903,	
\etex_scantokens:D .....	1478	13904, 14015, 14033, 14061, 14105,	
\etex_showgroups:D .....	1479	14117, 14122, 14130, 14139, 14161,	
\etex_showifs:D .....	1480	14167, 14239, 14252, 14253, 14262,	
\etex_showtokens:D .....	1481	14275, 14293, 14294, 14314, 14327,	
\etex_splitbotmarks:D .....	1482	14331, 14353, 14381, 14394, 14407,	
\etex_splitediscards:D .....	1483	14431, 14442, 14452, 14471, 14484,	
\etex_splitfirstmarks:D .....	1484	14497, 14500, 14512, 14535, 14564,	

14578, 14594, 14614, 14625, 14631,  
 14641, 14687, 14694, 14725, 14740,  
 14748, 14765, 14781, 14785, 14794,  
 14831, 14840, 14849, 14854, 14856,  
 14867, 14869, 14884, 14887, 14894,  
 14905, 14992, 15012, 15030, 15033,  
 15047, 15060, 15108, 15126, 15197,  
 15209, 15238, 15240, 15244, 15246,  
 15304, 15314, 15324, 15336, 15446,  
 15463, 15473, 15628, 15629, 15630,  
 15821, 15824, 15832, 15842, 15850,  
 16862, 17391, 17413, 17568, 17745,  
 17951, 18694, 18709, 18726, 18763,  
 18780, 18822, 18841, 18854, 18886,  
 18901, 18912, 18979, 19040, 19087,  
 19127, 19163, 19341, 19437, 26661,  
 26672, 26861, 26902, 26916, 27781,  
 27787, 27793, 27799, 27815, 27835  
  
`\exp_after:wN` 30, 32, 33, 329, 332,  
 349, 351, 400, 405, 459, 540, 644,  
 667, 670, 733, 734, 794, 853, 876,  
 944, 1068, 2067, 2085, 2087, 2092,  
 2094, 2323, 2353, 2354, 2366, 2415,  
 2418, 2422, 2494, 2500, 2513, 2515,  
 2567, 2591, 2609, 2611, 2630, 2638,  
 2646, 2670, 2675, 2682, 2719, 2723,  
 2728, 2739, 2755, 2757, 2760, 2783,  
 2785, 2788, 2914, 2916, 2925, 2945,  
 2947, 2986, 3050, 3143, 3153, 3160,  
 3162, 3174, 3175, 3187, 3188, 3193,  
 3194, 3199, 3204, 3206, 3209, 3218,  
 3220, 3223, 3224, 3225, 3228, 3230,  
 3232, 3236, 3241, 3246, 3249, 3254,  
 3259, 3260, 3261, 3265, 3266, 3267,  
 3273, 3274, 3281, 3282, 3283, 3287,  
 3288, 3289, 3293, 3294, 3295, 3299,  
 3300, 3301, 3305, 3306, 3307, 3311,  
 3312, 3313, 3314, 3318, 3319, 3320,  
 3321, 3325, 3326, 3327, 3332, 3333,  
 3334, 3335, 3339, 3340, 3341, 3342,  
 3389, 3392, 3393, 3397, 3398, 3411,  
 3412, 3419, 3421, 3423, 3427, 3431,  
 3433, 3436, 3437, 3442, 3443, 3447,  
 3450, 3451, 3455, 3458, 3459, 3460,  
 3465, 3466, 3467, 3475, 3478, 3479,  
 3480, 3481, 3486, 3488, 3490, 3491,  
 3502, 3505, 3510, 3535, 3536, 3537,  
 3548, 3549, 3561, 3562, 3563, 3568,  
 3576, 3577, 3578, 3579, 3580, 3581,  
 3626, 3627, 3629, 3651, 3656, 3657,  
 3669, 3670, 3672, 3676, 3677, 3678,  
 3681, 3683, 3686, 3690, 3691, 3692,  
 3697, 3698, 3709, 3715, 3717, 3721,  
 3723, 3724, 3733, 3774, 3778, 3800,  
 3807, 3827, 3978, 3996, 4007, 4235,  
 4253, 4254, 4255, 4256, 4310, 4311,  
 4312, 4369, 4370, 4371, 4376, 4418,  
 4429, 4430, 4455, 4500, 4502, 4591,  
 4704, 4731, 4762, 4767, 4768, 4769,  
 4771, 4784, 4794, 4811, 4833, 4843,  
 4846, 4863, 4864, 4874, 4879, 4880,  
 4895, 4896, 4897, 4955, 4957, 4958,  
 4959, 4964, 4965, 4967, 5045, 5046,  
 5292, 5293, 5305, 5360, 5383, 5417,  
 5418, 5429, 5430, 5438, 5446, 5448,  
 5455, 5460, 5478, 5479, 5480, 5492,  
 5493, 5520, 5522, 5528, 5534, 5548,  
 5568, 5579, 5595, 5603, 5611, 5618,  
 5625, 5637, 5724, 5730, 5732, 5756,  
 5807, 5808, 5930, 5932, 5944, 5952,  
 6049, 6083, 6095, 6108, 6109, 6110,  
 6132, 6133, 6178, 6213, 6214, 6215,  
 6272, 6273, 6299, 6300, 6303, 6396,  
 6401, 6409, 6410, 6422, 6423, 6444,  
 6445, 6471, 6472, 6481, 6621, 6626,  
 6631, 6654, 6656, 6809, 6810, 6811,  
 6836, 6837, 7020, 7048, 7053, 7081,  
 7094, 7104, 7131, 7133, 7134, 7142,  
 7159, 7203, 7292, 7293, 7304, 7313,  
 7365, 7367, 7374, 7463, 7481, 7486,  
 7490, 7612, 7686, 7764, 7772, 7832,  
 7904, 7919, 7941, 7955, 8016, 8024,  
 8030, 8125, 8133, 8217, 8218, 8221,  
 8222, 8466, 8467, 8514, 8522, 8593,  
 8594, 8595, 8707, 8726, 8773, 8811,  
 8840, 8841, 8843, 8849, 8852, 8891,  
 8894, 8930, 8932, 8942, 8943, 8944,  
 8946, 8952, 8953, 8955, 8962, 8963,  
 8965, 9013, 9022, 9031, 9122, 9133,  
 9304, 9305, 9306, 9510, 9729, 9730,  
 10255, 10256, 10257, 10258, 10270,  
 10371, 10426, 10537, 10555, 10604,  
 10797, 10800, 10850, 10859, 10862,  
 10865, 10866, 10868, 10908, 10974,  
 11005, 11018, 11039, 11049, 11054,  
 11123, 11124, 11125, 11230, 11243,  
 11436, 11440, 11448, 11449, 11461,  
 11462, 11492, 11497, 11508, 11511,  
 11633, 11634, 11635, 11697, 11828,  
 11995, 11996, 12005, 12007, 12024,  
 12029, 12031, 12048, 12057, 12061,  
 12062, 12069, 12071, 12072, 12073,  
 12074, 12085, 12108, 12111, 12190,  
 12233, 12380, 12381, 12689, 12769,  
 12770, 12793, 12806, 12817, 12818,  
 12997, 12998, 12999, 13016, 13024,  
 13047, 13048, 13089, 13225, 13227,  
 13228, 13239, 13240, 13241, 13251,

13253, 13261, 13263, 13270, 13271,  
13272, 13273, 13274, 13275, 13280,  
13281, 13282, 13283, 13284, 13285,  
13286, 13329, 13342, 13345, 13356,  
13358, 13373, 13377, 13378, 13379,  
13382, 13383, 13447, 13449, 13476,  
13480, 13505, 13509, 13526, 13533,  
13535, 13547, 13630, 13638, 13655,  
13664, 13710, 13775, 13844, 13845,  
13846, 13906, 13916, 13935, 13941,  
13967, 13968, 13971, 13982, 13986,  
13993, 13994, 14005, 14006, 14015,  
14022, 14024, 14025, 14033, 14061,  
14079, 14080, 14083, 14084, 14086,  
14087, 14091, 14092, 14094, 14095,  
14104, 14105, 14110, 14116, 14122,  
14130, 14139, 14159, 14160, 14163,  
14164, 14166, 14173, 14174, 14176,  
14194, 14195, 14223, 14226, 14231,  
14232, 14237, 14238, 14240, 14249,  
14250, 14251, 14252, 14255, 14256,  
14257, 14260, 14275, 14292, 14293,  
14303, 14304, 14314, 14326, 14330,  
14343, 14345, 14353, 14363, 14365,  
14371, 14376, 14378, 14380, 14386,  
14387, 14391, 14393, 14405, 14406,  
14428, 14430, 14436, 14439, 14441,  
14445, 14450, 14455, 14456, 14466,  
14467, 14469, 14470, 14473, 14477,  
14482, 14496, 14499, 14511, 14520,  
14527, 14528, 14529, 14530, 14532,  
14534, 14545, 14546, 14547, 14548,  
14550, 14552, 14554, 14555, 14556,  
14562, 14563, 14573, 14577, 14578,  
14580, 14581, 14582, 14587, 14593,  
14604, 14606, 14613, 14614, 14616,  
14617, 14624, 14630, 14640, 14708,  
14721, 14722, 14723, 14724, 14738,  
14739, 14741, 14746, 14747, 14762,  
14764, 14781, 14785, 14794, 14828,  
14829, 14830, 14836, 14837, 14838,  
14839, 14845, 14846, 14847, 14848,  
14855, 14868, 14876, 14882, 14883,  
14885, 14886, 14892, 14893, 14895,  
14921, 14934, 14956, 14957, 14959,  
14960, 14967, 14968, 14970, 14973,  
14985, 14987, 14988, 14990, 14991,  
14993, 15005, 15006, 15007, 15010,  
15011, 15012, 15022, 15023, 15024,  
15027, 15028, 15029, 15032, 15036,  
15045, 15046, 15057, 15058, 15059,  
15062, 15064, 15065, 15066, 15091,  
15092, 15094, 15095, 15096, 15106,  
15107, 15108, 15110, 15111, 15112,  
15124, 15125, 15128, 15130, 15131,  
15132, 15152, 15153, 15154, 15155,  
15156, 15157, 15158, 15168, 15169,  
15171, 15172, 15173, 15179, 15190,  
15191, 15192, 15193, 15194, 15195,  
15196, 15197, 15202, 15203, 15204,  
15205, 15206, 15207, 15208, 15224,  
15225, 15227, 15228, 15235, 15236,  
15237, 15242, 15243, 15245, 15261,  
15276, 15285, 15295, 15301, 15302,  
15303, 15308, 15321, 15322, 15323,  
15329, 15445, 15462, 15472, 15497,  
15498, 15545, 15627, 15735, 15737,  
15776, 15778, 15781, 15816, 15818,  
15820, 15823, 15830, 15831, 15834,  
15835, 15840, 15841, 15848, 15849,  
15884, 15885, 15886, 15888, 15899,  
15924, 15926, 15932, 15933, 15937,  
15940, 15962, 15964, 15977, 15979,  
15985, 15987, 15990, 15996, 15998,  
16000, 16001, 16002, 16004, 16009,  
16011, 16013, 16017, 16020, 16026,  
16027, 16031, 16033, 16034, 16035,  
16043, 16045, 16046, 16053, 16059,  
16066, 16067, 16072, 16073, 16074,  
16075, 16094, 16095, 16096, 16102,  
16103, 16104, 16109, 16111, 16119,  
16121, 16123, 16124, 16126, 16137,  
16139, 16141, 16142, 16147, 16198,  
16199, 16206, 16207, 16209, 16211,  
16213, 16216, 16219, 16221, 16223,  
16232, 16234, 16240, 16242, 16244,  
16245, 16246, 16252, 16254, 16256,  
16257, 16258, 16279, 16280, 16283,  
16291, 16293, 16297, 16298, 16299,  
16300, 16305, 16307, 16314, 16317,  
16320, 16323, 16332, 16335, 16338,  
16341, 16348, 16350, 16356, 16364,  
16366, 16368, 16385, 16387, 16394,  
16396, 16399, 16405, 16407, 16409,  
16410, 16411, 16413, 16427, 16428,  
16431, 16449, 16451, 16453, 16465,  
16468, 16471, 16474, 16477, 16480,  
16483, 16486, 16490, 16502, 16506,  
16510, 16513, 16528, 16534, 16536,  
16538, 16548, 16572, 16575, 16587,  
16589, 16593, 16594, 16595, 16597,  
16598, 16600, 16607, 16615, 16616,  
16622, 16623, 16629, 16632, 16633,  
16634, 16635, 16643, 16685, 16690,  
16692, 16699, 16702, 16705, 16708,  
16711, 16714, 16722, 16723, 16735,  
16743, 16745, 16755, 16757, 16764,  
16773, 16775, 16778, 16781, 16784,

16787, 16800, 16802, 16810, 16812,  
16820, 16822, 16832, 16835, 16838,  
16845, 16860, 16861, 16878, 16880,  
16881, 16938, 16951, 16953, 16959,  
16972, 16974, 16976, 17000, 17014,  
17016, 17023, 17025, 17066, 17067,  
17068, 17070, 17071, 17072, 17074,  
17075, 17081, 17083, 17084, 17090,  
17092, 17093, 17094, 17095, 17107,  
17113, 17115, 17150, 17157, 17164,  
17184, 17185, 17187, 17189, 17191,  
17204, 17209, 17210, 17211, 17212,  
17213, 17217, 17222, 17224, 17230,  
17236, 17238, 17239, 17245, 17246,  
17247, 17248, 17249, 17250, 17251,  
17252, 17257, 17259, 17261, 17263,  
17265, 17270, 17272, 17274, 17276,  
17278, 17280, 17298, 17302, 17310,  
17311, 17316, 17318, 17327, 17330,  
17331, 17332, 17334, 17335, 17336,  
17344, 17350, 17362, 17365, 17366,  
17368, 17369, 17393, 17394, 17397,  
17399, 17415, 17419, 17420, 17421,  
17437, 17443, 17509, 17510, 17511,  
17518, 17520, 17521, 17526, 17528,  
17529, 17538, 17539, 17541, 17544,  
17547, 17563, 17567, 17568, 17572,  
17574, 17610, 17616, 17617, 17619,  
17621, 17622, 17624, 17625, 17635,  
17636, 17639, 17640, 17641, 17643,  
17644, 17645, 17662, 17663, 17665,  
17666, 17672, 17674, 17677, 17680,  
17683, 17686, 17694, 17697, 17699,  
17702, 17709, 17713, 17721, 17722,  
17725, 17727, 17729, 17734, 17735,  
17741, 17746, 17747, 17755, 17756,  
17757, 17758, 17923, 17924, 17925,  
17927, 17945, 17946, 17947, 17948,  
17949, 17950, 17957, 17966, 17973,  
17974, 18198, 18199, 18205, 18206,  
18209, 18214, 18217, 18220, 18223,  
18226, 18229, 18232, 18235, 18251,  
18252, 18262, 18271, 18279, 18280,  
18282, 18283, 18288, 18289, 18298,  
18305, 18314, 18315, 18328, 18330,  
18358, 18359, 18368, 18371, 18396,  
18402, 18403, 18445, 18446, 18448,  
18462, 18463, 18471, 18482, 18516,  
18519, 18529, 18530, 18533, 18535,  
18541, 18555, 18557, 18598, 18601,  
18621, 18694, 18704, 18708, 18726,  
18729, 18750, 18751, 18758, 18762,  
18780, 18783, 18812, 18813, 18819,  
18820, 18821, 18828, 18836, 18840,  
18854, 18857, 18873, 18874, 18881,  
18885, 18896, 18900, 18912, 18918,  
18919, 18920, 18926, 18928, 18931,  
18933, 18975, 18977, 18978, 19000,  
19029, 19039, 19046, 19051, 19052,  
19062, 19089, 19100, 19102, 19104,  
19106, 19111, 19112, 19114, 19125,  
19126, 19146, 19152, 19153, 19155,  
19158, 19163, 19169, 19170, 19200,  
19202, 19205, 19208, 19210, 19218,  
19220, 19224, 19228, 19233, 19238,  
19249, 19312, 19313, 19336, 19337,  
19338, 19339, 19340, 19347, 19358,  
19364, 19390, 19391, 19392, 19398,  
19399, 19405, 19406, 19413, 19414,  
19418, 19419, 19420, 19425, 19430,  
19436, 19439, 19440, 19441, 19442,  
19443, 19487, 19585, 19586, 19628,  
19647, 19648, 19663, 19664, 19665,  
19893, 19899, 19901, 19912, 19972,  
19973, 19974, 19975, 19981, 19982,  
19998, 20016, 20018, 20024, 20025,  
20056, 20058, 20060, 20090, 20105,  
20107, 20109, 20134, 20144, 20152,  
20162, 20172, 20174, 20176, 20211,  
20235, 20244, 20247, 20248, 20250,  
20251, 20259, 20260, 20274, 20338,  
20351, 20352, 20357, 20358, 20361,  
20364, 20402, 20409, 20416, 20422,  
20429, 20437, 20473, 20475, 20484,  
20616, 20619, 20660, 20680, 20682,  
20683, 20690, 20693, 20694, 20718,  
20737, 20746, 20858, 20860, 20866,  
20869, 20871, 20878, 20881, 20883,  
20890, 20892, 20898, 20901, 20904,  
21219, 21292, 21304, 21436, 21439,  
21449, 21451, 21634, 21642, 21716,  
21766, 21992, 22292, 22448, 22471,  
22528, 22554, 22618, 22619, 22627,  
22630, 22794, 22795, 22798, 22799,  
22807, 22809, 22810, 22819, 22833,  
22836, 22906, 23155, 23187, 23189,  
25585, 25587, 25684, 26089, 26114,  
26163, 26199, 26200, 26268, 26278,  
26283, 26286, 26414, 26415, 26417,  
26418, 26426, 26427, 26435, 26578,  
26579, 26580, 26670, 26690, 26859,  
26900, 26914, 26931, 26932, 27096,  
27097, 27118, 27119, 27167, 27170,  
27171, 27218, 27253, 27345, 27395,  
27463, 27477, 27479, 27485, 27507,  
27520, 27522, 27536, 27538, 27662,  
27720, 27781, 27787, 27793, 27799,  
27813, 27816, 27833, 27835, 27836,



10539, 10666, 10696, 10792, 11121,  
 11252, 11834, 12477, 12495, 12523,  
 12545, 20198, 20221, 20278, 20280,  
 21029, 21086, 21106, 21741, 22121,  
 22762, 22805, 23210, 23240, 23738,  
 26561, 26576, 27872, 27876, 27961  
`\exp_args:Noc` ..... 28, 3345  
`\exp_args:Nof` ..... 28, 3345  
`\exp_args:Noo` . 28, 3345, 21128, 22134  
`\exp_args:Noof` ..... 29, 3363  
`\exp_args:Nooo` ..... 29, 3363  
`\exp_args:Noox` ..... 29, 3363  
`\exp_args:Nox` ..... 28, 3345  
`\exp_args:NV` ..... 27, 3245,  
 10934, 11010, 11110, 11153, 12475,  
 12493, 12521, 12543, 20511, 28033  
`\exp_args:Nv` ..... 27, 3245  
`\exp_args:NVo` ..... 28, 3345  
`\exp_args:NVV` ..... 28, 3257, 10845  
`\exp_args:Nx` ..... 28,  
 2923, 3344, 5034, 7328, 7440, 8571,  
 9483, 10333, 10702, 12479, 12497,  
 12525, 12547, 15420, 21261, 21262,  
 21645, 22565, 28139, 28540, 28940  
`\exp_args:Nxo` ..... 28, 3345  
`\exp_args:Nxx` ..... 28, 3345  
`\exp_args_generate:n` ..... 244, 4027  
`\exp_end:` ..... 33, 33, 329, 332,  
 335, 351, 352, 360, 360, 395, 395,  
 401, 415, 479, 553, 667, 697, 1043,  
 1067, 1068, 2071, 2616, 2629, 2637,  
 2645, 3206, 3215, 3513, 3543, 3692,  
 4541, 4711, 4895, 4896, 4967, 5289,  
 5463, 6715, 7643, 7646, 7647, 7648,  
 7649, 7650, 7651, 7652, 7653, 7654,  
 7656, 7760, 8497, 8511, 8514, 8519,  
 8522, 8528, 8536, 8589, 8594, 10258,  
 11559, 13906, 14861, 17415, 19089,  
 19091, 19163, 26889, 27771, 27804  
`\exp_end_continue_f:nw` ..... 34, 3513  
`\exp_end_continue_f:w` .....  
 ..... 33, 34, 350, 670, 3175,  
 3246, 3283, 3307, 3412, 3431, 3444,  
 3468, 3482, 3502, 3513, 7465, 7832,  
 10195, 10867, 11497, 13265, 13380,  
 13384, 13547, 14015, 14033, 14053,  
 14117, 14122, 14130, 14139, 14161,  
 14239, 14275, 14283, 14314, 14687,  
 14694, 14740, 14787, 14794, 14831,  
 14875, 14881, 14884, 14894, 14905,  
 15047, 15238, 15240, 15244, 15246,  
 15304, 15314, 15324, 15336, 15446,  
 15463, 15473, 15628, 15629, 15630,  
 15821, 15832, 15842, 15850, 16862,  
 17568, 17745, 17951, 18694, 18709,  
 18726, 18763, 18780, 18822, 18841,  
 18854, 18886, 18901, 18912, 18980,  
 19040, 19127, 19341, 19437, 26661  
`\exp_last_two_unbraced:Nnn` .....  
 ..... 30, 3485, 24752, 25012, 25016  
`\exp_last_unbraced:Ne` 30, 3426, 3429  
`\exp_last_unbraced:Nn` ..... 30,  
 30, 3419, 6981, 7001, 8597, 13079,  
 13568, 15914, 20671, 25143, 25148,  
 25226, 25232, 28042, 28091, 29807  
`\exp_last_unbraced:NNn` .....  
 ..... 30, 3419, 4718,  
 8111, 8144, 10699, 24983, 26922, 26981  
`\exp_last_unbraced:Nnn` .....  
 .. 30, 3419, 9185, 9269, 19293, 26252  
`\exp_last_unbraced:NNNn` 30, 3419, 7407  
`\exp_last_unbraced:NnNn` ... 30, 3419  
`\exp_last_unbraced:NNNNn` .....  
 ..... 30, 3419, 3787, 3791, 3969,  
 5657, 7413, 12244, 13333, 13351, 29787  
`\exp_not:e` ..... 32, 3490  
`\exp_not:N` .. 31, 31, 150, 202, 351,  
 359, 363, 398, 399, 466, 511, 518,  
 675, 853, 862, 936, 940, 1119, 2067,  
 2442, 2446, 2571, 2657, 2660, 2985,  
 2986, 3049, 3050, 3153, 3199, 3490,  
 3490, 3568, 3572, 3626, 3646, 3656,  
 3669, 3764, 3766, 3767, 3774, 3775,  
 3776, 3777, 3778, 3779, 3780, 3781,  
 3783, 3784, 3785, 3811, 3820, 3874,  
 3875, 3876, 3877, 3949, 3955, 3956,  
 3957, 3959, 3962, 3965, 3966, 3978,  
 4051, 4362, 4791, 4794, 4808, 4811,  
 4842, 4849, 5094, 5095, 5317, 5318,  
 5921, 5923, 6260, 6867, 7130, 7293,  
 8086, 8196, 8199, 8207, 8208, 8454,  
 8541, 8545, 8574, 8626, 8630, 8635,  
 8640, 8645, 8652, 8659, 8664, 8669,  
 8674, 8679, 8684, 8694, 8699, 8704,  
 8707, 8708, 8709, 8711, 8712, 8721,  
 8726, 8741, 8742, 8759, 8764, 8765,  
 8766, 8767, 8768, 8769, 8771, 8773,  
 8774, 8775, 8776, 8779, 8780, 8783,  
 8784, 8805, 8808, 8809, 8811, 8812,  
 8813, 8816, 8817, 8819, 8820, 8822,  
 8823, 8825, 8904, 8910, 8941, 8944,  
 8951, 8952, 8961, 8962, 8976, 8977,  
 8992, 8997, 9002, 9116, 9224, 9247,  
 9589, 9591, 9593, 9595, 9600, 9601,  
 9606, 9608, 9610, 9892, 9894, 9896,  
 9898, 9903, 9904, 9909, 9911, 9913,  
 11676, 11702, 12081, 12090, 12252,  
 12254, 12268, 12270, 12324, 12325,

12448, 12449, 13329, 13330, 13331,  
 14078, 14079, 14176, 14177, 14178,  
 14179, 14180, 14285, 14325, 14329,  
 14351, 14444, 14476, 14561, 14575,  
 14592, 14602, 14612, 14649, 14652,  
 14758, 14759, 14761, 14762, 14763,  
 14764, 14765, 14766, 14769, 14771,  
 14773, 14952, 14954, 14982, 14983,  
 14985, 14986, 14987, 14988, 14989,  
 14990, 14991, 14992, 14993, 14994,  
 14995, 15090, 15105, 15710, 15867,  
 17804, 17805, 17806, 17810, 17811,  
 17812, 18974, 18975, 18977, 18978,  
 18979, 18980, 18981, 18983, 19960,  
 19963, 20115, 20116, 20117, 20118,  
 20119, 20120, 20121, 20122, 20123,  
 20124, 20125, 20126, 20127, 20128,  
 20129, 20130, 20133, 20134, 20135,  
 20647, 20649, 20651, 20653, 20655,  
 20657, 21017, 21019, 21212, 21214,  
 21225, 21229, 21396, 21849, 22559,  
 22776, 22898, 22911, 23274, 25550,  
 25554, 25555, 25560, 26125, 26209,  
 26211, 26215, 26217, 26222, 26224,  
 26320, 26330, 27464, 27478, 27480,  
 27521, 27523, 27537, 27539, 27912,  
 27913, 27974, 27975, 28161, 28162,  
 28170, 28564, 28584, 28724, 28726,  
 28776, 29676, 29679, 29795, 29820  
  
`\exp_stop_f:` . . . . .  
 . . . 32, 33, 91, 350, 397, 432, 571,  
 636, 648, 714, 715, 801, 828, 849,  
 853, 1029, 3172, 3633, 3636, 3651,  
 3659, 3714, 4954, 4963, 5012, 5013,  
 5019, 5197, 5204, 5211, 5445, 5461,  
 5500, 5501, 5507, 5519, 5535, 5536,  
 5955, 6398, 6413, 6426, 6648, 6653,  
 6832, 7295, 8344, 8346, 8414, 8416,  
 8420, 8422, 8426, 8428, 8432, 8434,  
 8472, 8473, 8480, 8481, 8482, 8483,  
 8488, 8489, 8508, 8523, 8531, 8597,  
 9122, 10270, 10426, 10604, 10855,  
 10869, 10881, 11506, 12976, 12980,  
 13146, 13150, 13178, 13373, 13488,  
 13503, 13528, 13775, 13779, 13783,  
 13785, 13789, 13793, 13801, 13806,  
 13819, 13826, 13839, 13850, 13851,  
 13862, 13863, 13872, 13875, 13886,  
 13928, 13984, 13989, 14060, 14090,  
 14248, 14291, 14342, 14362, 14389,  
 14403, 14438, 14465, 14474, 14493,  
 14509, 14525, 14543, 14603, 14622,  
 14638, 14653, 14667, 14876, 14955,  
 14966, 15219, 15223, 15511, 15517,  
 15519, 15534, 15543, 15551, 15559,  
 15560, 15757, 15877, 15883, 15898,  
 15935, 16008, 16030, 16084, 16085,  
 16093, 16430, 16448, 16501, 16505,  
 16509, 16527, 16562, 16563, 16564,  
 16565, 16566, 16592, 16604, 16620,  
 16637, 16915, 16916, 17013, 17106,  
 17139, 17152, 17157, 17166, 17168,  
 17295, 17324, 17329, 17359, 17396,  
 17436, 17512, 17562, 17608, 17609,  
 17614, 17620, 17638, 17661, 17693,  
 17696, 17743, 17763, 17769, 17784,  
 17825, 17840, 17855, 17870, 17885,  
 17900, 17928, 17972, 18238, 18248,

18278, 18430, 18432, 18469, 18481,  
 18513, 18554, 18563, 18578, 18597,  
 18630, 18643, 18727, 18781, 18829,  
 18832, 18855, 19070, 19074, 19081,  
 19082, 19137, 19138, 19139, 19148,  
 19158, 19176, 19245, 19248, 19251,  
 19266, 19317, 19321, 19362, 19431,  
 19438, 19486, 19757, 19926, 19935,  
 19936, 19970, 20040, 20048, 20071,  
 20073, 20074, 20078, 20095, 20146,  
 20149, 20165, 20171, 20243, 20246,  
 20444, 20445, 20446, 20452, 20472,  
 20733, 20753, 20754, 20758, 20762,  
 20763, 20766, 20767, 20775, 20776,  
 20779, 20783, 20784, 20787, 20846,  
 20941, 21185, 21190, 21204, 21205,  
 21218, 21289, 21290, 21329, 21429,  
 21606, 21764, 22056, 22074, 22103,  
 22113, 22168, 22181, 22192, 22208,  
 22259, 22491, 22625, 22629, 22696,  
 22759, 22772, 22792, 22797, 22803,  
 22849, 22902, 22919, 22942, 23160,  
 23165, 23186, 23264, 23276, 23281,  
 25835, 26165, 26437, 26531, 26778,  
 26779, 26785, 27303, 27318, 27373,  
 27886, 27894, 27912, 27913, 27914,  
 27915, 27974, 27975, 27976, 27977  
**exp internal commands:**  
 \\_\_exp\_arg\_last\_unbraced:nn .. 3387  
 \\_\_exp\_arg\_next:Nnn ..... 3153, 3160  
 \\_\_exp\_arg\_next:nmn .....  
     351, 3153, 3162, 3170, 3174, 3187, 3193  
 \\_\_exp\_e:N ..... 3529, 3559  
 \\_\_exp\_e:nn ..... 353, 360,  
     3242, 3407, 3525, 3545, 3550, 3558,  
     3586, 3588, 3593, 3598, 3665, 3683  
 \\_\_exp\_e:Nnn ..... 361, 3561, 3565  
 \\_\_exp\_e\_end:nn ..... 360, 3535, 3543, 3617  
 \\_\_exp\_e\_expandable:Nnn .....  
     362, 3576, 3587  
 \\_\_exp\_e\_group:n ..... 3532, 3546  
 \\_\_exp\_e\_if\_toks\_register:N .. 3726  
 \\_\_exp\_e\_if\_toks\_register:NTF ..  
     3680, 3726  
 \\_\_exp\_e\_noexpand:Nnn ... 3579, 3595  
 \\_\_exp\_e\_primitive:Nnn ... 3581, 3589  
 \\_\_exp\_e\_protected:Nnn 362, 3577, 3585  
 \\_\_exp\_e\_put:nn .....  
     361, 362, 364, 3546, 3598, 3610, 3697  
 \\_\_exp\_e\_put:nnn .... 365, 3546, 3703  
 \\_\_exp\_e\_space:nn ..... 3536, 3544  
 \\_\_exp\_e\_the:N ..... 3661  
 \\_\_exp\_e\_the:Nnn ..... 3580, 3661  
 \\_\_exp\_e\_the\_errhelp: ..... 3740  
 \\_\_exp\_e\_the\_everycr: ..... 3741  
 \\_\_exp\_e\_the\_everydisplay: ... 3742  
 \\_\_exp\_e\_the\_everyeof: ..... 3743  
 \\_\_exp\_e\_the\_everyhbox: ..... 3744  
 \\_\_exp\_e\_the\_everyjob: ..... 3745  
 \\_\_exp\_e\_the\_everymath: ..... 3746  
 \\_\_exp\_e\_the\_everypar: ..... 3747  
 \\_\_exp\_e\_the\_everyvbox: ..... 3748  
 \\_\_exp\_e\_the\_output: ..... 3749  
 \\_\_exp\_e\_the\_pdfpageattr: .... 3750  
 \\_\_exp\_e\_the\_pdffpageresources: 3751  
 \\_\_exp\_e\_the\_pdfpagesattr: ... 3752  
 \\_\_exp\_e\_the\_pdpkmode: ..... 3753  
 \\_\_exp\_e\_the\_toks:N ..... 365, 3701  
 \\_\_exp\_e\_the\_toks:n . 365, 3677, 3701  
 \\_\_exp\_e\_the\_toks:wnn 365, 3676, 3701  
 \\_\_exp\_e\_the\_toks\_reg:N ..... 3661  
 \\_\_exp\_e\_the\_XeTeXinterchartoks:  
     ..... 3739  
 \\_\_exp\_e\_unexpanded:N ..... 3600  
 \\_\_exp\_e\_unexpanded:nN ..... 363, 3600  
 \\_\_exp\_e\_unexpanded:nn ..... 3600  
 \\_\_exp\_e\_unexpanded:Nnn ... 3578, 3600  
 \\_\_exp\_eval\_error\_msg:w ..... 3197  
 \\_\_exp\_eval\_register:N .....  
     ..... 3188, 3194, 3197,  
     3250, 3255, 3261, 3267, 3295, 3301,  
     3313, 3314, 3321, 3393, 3398, 3421,  
     3423, 3438, 3452, 3461, 3506, 3511  
 \l\_\_exp\_internal\_tl .....  
     319, 2135, 2139, 2140,  
     3153, 3153, 3181, 3183, 3416, 3417  
 \\_\_exp\_last\_two\_unbraced:nnN .. 3485  
**\expandafter** ..... 13, 14, 21,  
     38, 39, 42, 43, 48, 49, 66, 67, 90, 92,  
     93, 94, 105, 130, 153, 161, 176, 192, 371  
**\expanded** ..... 920, 1741  
**\expandglyphsinfont** ..... 1015, 1624  
**\ExplFileVersion** ..... 6  
**\ExplFileDescription** ..... 6  
**\ExplFileName** ..... 6  
**\ExplSyntaxOff** ..... 3,  
     6, 6, 107, 213, 246, 260, 273, 274, 311  
**\ExplSyntaxOn** ..... 3,  
     6, 6, 107, 242, 273, 274, 311, 381, 503

**F**

**false** ..... 202  
**\fam** ..... 372

\fi . . . . . 17, 35, 41, 51, 70, 71,  
72, 97, 100, 102, 103, 104, 107, 108,  
137, 146, 159, 160, 177, 193, 211, 373

fi commands:

\fi: . . . . . 20, 91,  
91, 92, 96, 103, 103, 151, 168, 231,  
231, 231, 329, 330, 332, 335, 337,  
380, 385, 387, 415, 417, 420, 449,  
459, 480, 507, 578, 648, 674, 690,  
720, 853, 2049, 2095, 2183, 2313,  
2330, 2341, 2342, 2390, 2423, 2424,  
2568, 2576, 2584, 2592, 2612, 2617,  
2630, 2638, 2646, 2648, 2671, 2676,  
2683, 2718, 2723, 2749, 2750, 2758,  
2764, 2777, 2778, 2786, 2792, 2917,  
2938, 2948, 2962, 3019, 3080, 3144,  
3202, 3205, 3212, 3213, 3520, 3527,  
3537, 3550, 3563, 3568, 3571, 3572,  
3573, 3574, 3582, 3630, 3658, 3664,  
3673, 3678, 3684, 3687, 3691, 3699,  
3709, 3718, 3724, 3785, 3801, 3808,  
3817, 3831, 3832, 3837, 3838, 3839,  
3857, 3858, 3859, 3860, 3861, 3862,  
3863, 3864, 3865, 3873, 3893, 3895,  
3925, 3926, 3927, 3995, 4006, 4016,  
4240, 4249, 4375, 4376, 4402, 4412,  
4423, 4438, 4446, 4461, 4472, 4476,  
4506, 4705, 4758, 4763, 4772, 4774,  
4801, 4822, 4838, 4847, 4856, 4862,  
4869, 4874, 4885, 4889, 4897, 4956,  
4968, 5017, 5023, 5024, 5198, 5205,  
5211, 5306, 5374, 5378, 5379, 5397,  
5450, 5463, 5505, 5511, 5512, 5523,  
5536, 5537, 5558, 5596, 5622, 5725,  
5733, 5757, 5771, 5779, 5790, 5800,  
5996, 5999, 6036, 6050, 6067, 6077,  
6131, 6136, 6401, 6434, 6456, 6457,  
6459, 6481, 6482, 6521, 6530, 6606,  
6614, 6641, 6649, 6655, 6684, 6724,  
6734, 6834, 7049, 7082, 7130, 7135,  
7251, 7317, 7358, 7368, 7424, 7456,  
7474, 7496, 7516, 7532, 7542, 7558,  
7568, 7658, 7660, 7662, 7664, 7666,  
7668, 7905, 7920, 7943, 7957, 8477,  
8480, 8481, 8482, 8483, 8488, 8489,  
8494, 8495, 8496, 8535, 8545, 8592,  
8600, 8602, 8631, 8636, 8641, 8646,  
8653, 8660, 8665, 8670, 8675, 8680,  
8685, 8690, 8695, 8700, 8722, 8733,  
8734, 8783, 8784, 8835, 8844, 8853,  
8861, 8895, 8933, 8947, 8956, 8966,  
9286, 10150, 10154, 10155, 10495,  
10538, 10556, 10807, 10848, 10857,  
10878, 10888, 10892, 10899, 10907,

11118, 11127, 11440, 11473, 11488,  
11505, 11509, 11523, 11526, 12025,  
12049, 12063, 12075, 12100, 12101,  
12112, 12985, 12986, 13017, 13025,  
13087, 13229, 13238, 13242, 13254,  
13264, 13359, 13412, 13415, 13416,  
13421, 13435, 13473, 13474, 13475,  
13476, 13477, 13478, 13479, 13480,  
13481, 13482, 13483, 13484, 13497,  
13499, 13510, 13513, 13527, 13532,  
13536, 13690, 13781, 13782, 13791,  
13792, 13803, 13804, 13805, 13816,  
13817, 13818, 13825, 13836, 13837,  
13838, 13848, 13849, 13853, 13854,  
13862, 13865, 13866, 13874, 13885,  
13905, 13928, 13972, 13991, 13992,  
14001, 14007, 14027, 14028, 14055,  
14064, 14081, 14088, 14096, 14097,  
14200, 14201, 14202, 14205, 14208,  
14247, 14263, 14289, 14290, 14297,  
14305, 14334, 14335, 14338, 14340,  
14341, 14346, 14356, 14359, 14361,  
14366, 14397, 14410, 14415, 14421,  
14424, 14425, 14459, 14460, 14487,  
14488, 14501, 14504, 14515, 14538,  
14557, 14567, 14583, 14597, 14603,  
14607, 14612, 14618, 14633, 14644,  
14663, 14673, 14675, 14681, 14702,  
14730, 14753, 14786, 14788, 14911,  
14961, 14965, 14975, 14976, 14995,  
15013, 15034, 15037, 15067, 15097,  
15113, 15133, 15174, 15186, 15199,  
15201, 15221, 15222, 15229, 15247,  
15286, 15296, 15457, 15468, 15497,  
15498, 15499, 15506, 15508, 15509,  
15515, 15516, 15519, 15546, 15554,  
15555, 15563, 15564, 15566, 15567,  
15738, 15751, 15761, 15762, 15767,  
15768, 15769, 15770, 15771, 15772,  
15779, 15789, 15796, 15807, 15808,  
15819, 15836, 15881, 15882, 15889,  
15902, 15917, 15927, 15941, 15971,  
15980, 16014, 16036, 16054, 16071,  
16088, 16089, 16091, 16092, 16097,  
16112, 16145, 16174, 16175, 16176,  
16177, 16178, 16191, 16235, 16308,  
16375, 16377, 16378, 16388, 16417,  
16420, 16421, 16432, 16452, 16515,  
16516, 16517, 16529, 16568, 16569,  
16570, 16571, 16577, 16580, 16582,  
16592, 16610, 16625, 16637, 16644,  
16891, 16895, 16897, 16901, 16908,  
16909, 16919, 16920, 16923, 17015,  
17085, 17096, 17108, 17138, 17145,

17156, 17172, 17179, 17240, 17297,  
 17307, 17309, 17319, 17337, 17338,  
 17370, 17373, 17382, 17384, 17386,  
 17400, 17414, 17438, 17522, 17530,  
 17561, 17569, 17575, 17586, 17589,  
 17592, 17601, 17611, 17613, 17619,  
 17626, 17629, 17638, 17646, 17667,  
 17700, 17701, 17728, 17730, 17748,  
 17749, 17768, 17779, 17788, 17791,  
 17835, 17850, 17865, 17880, 17895,  
 17910, 17913, 17915, 17932, 17977,  
 18284, 18320, 18321, 18331, 18372,  
 18373, 18397, 18424, 18425, 18428,  
 18430, 18431, 18436, 18448, 18467,  
 18472, 18480, 18483, 18515, 18525,  
 18526, 18536, 18558, 18573, 18591,  
 18599, 18602, 18630, 18638, 18654,  
 18726, 18744, 18780, 18798, 18831,  
 18854, 18860, 18934, 18935, 19071,  
 19072, 19081, 19088, 19093, 19103,  
 19113, 19138, 19141, 19154, 19186,  
 19194, 19195, 19223, 19245, 19246,  
 19247, 19250, 19255, 19275, 19276,  
 19326, 19327, 19367, 19375, 19425,  
 19431, 19444, 19488, 19500, 19501,  
 19556, 19587, 19629, 19640, 19649,  
 19658, 19713, 19723, 19733, 19847,  
 19849, 19903, 19907, 19911, 19938,  
 19949, 19967, 19968, 19969, 19976,  
 19992, 19998, 20001, 20009, 20019,  
 20034, 20042, 20050, 20061, 20077,  
 20097, 20110, 20132, 20150, 20158,  
 20160, 20163, 20170, 20175, 20252,  
 20253, 20403, 20410, 20411, 20417,  
 20420, 20423, 20430, 20431, 20434,  
 20438, 20439, 20449, 20450, 20455,  
 20456, 20468, 20476, 20485, 20486,  
 20514, 20684, 20695, 20747, 20749,  
 20756, 20759, 20760, 20764, 20768,  
 20769, 20770, 20771, 20780, 20781,  
 20785, 20788, 20789, 20790, 20826,  
 20829, 20850, 20853, 20861, 20872,  
 20873, 20884, 20885, 20893, 20905,  
 20906, 20916, 20917, 20930, 20949,  
 20950, 20958, 20959, 21010, 21020,  
 21046, 21060, 21064, 21127, 21175,  
 21178, 21179, 21194, 21197, 21220,  
 21287, 21288, 21293, 21320, 21321,  
 21332, 21336, 21370, 21375, 21383,  
 21418, 21425, 21430, 21440, 21452,  
 21478, 21541, 21570, 21609, 21616,  
 21627, 21700, 21717, 21721, 21735,  
 21769, 21993, 22044, 22060, 22084,  
 22108, 22117, 22177, 22184, 22204,  
 22222, 22233, 22235, 22265, 22268,  
 22293, 22420, 22449, 22472, 22473,  
 22494, 22529, 22555, 22628, 22689,  
 22701, 22764, 22778, 22779, 22800,  
 22811, 22837, 22854, 22904, 22906,  
 22921, 22923, 22944, 23046, 23105,  
 23122, 23123, 23162, 23163, 23167,  
 23168, 23190, 23195, 23232, 23270,  
 23278, 23279, 23283, 23284, 23704,  
 23706, 23712, 25579, 25580, 25586,  
 25599, 25603, 25604, 25620, 25685,  
 26156, 26172, 26173, 26448, 26449,  
 26573, 26582, 26666, 26813, 26814,  
 26817, 27803, 27810, 27815, 27843,  
 27849, 27853, 27868, 27889, 27897,  
 27912, 27913, 27914, 27924, 27931,  
 27933, 27974, 27975, 27976, 27982  
 file commands:  
 \file\_add\_path:nN ..... 29698  
 \g\_file\_curr\_dir\_str .....  
     ... 151, 11014, 11214, 11220, 11237  
 \g\_file\_curr\_ext\_str .....  
     ... 151, 11014, 11216, 11222, 11239  
 \g\_file\_curr\_name\_str . 151, 9410,  
     11014, 11057, 11215, 11221, 11238  
 \g\_file\_current\_name\_str ..... 29711  
 \g\_file\_current\_name\_tl ..... 29710  
 \file\_get:nnN ..... 152,  
     152, 11099, 26708, 26710, 26712,  
     26716, 26721, 26725, 26732, 26736  
 \file\_get:nnNTF ... 152, 11099, 11101  
 \file\_get\_full\_name:nN .....  
     ... 152, 312, 1027, 10415,  
     11106, 11136, 11177, 11190, 26086,  
     26097, 26111, 26137, 26143, 29699  
 \file\_get\_mdfive\_hash:nN . 245, 26078  
 \file\_get\_size:nN ..... 245, 26078  
 \file\_get\_timestamp:nN ... 245, 26078  
 \file\_if\_exist:nTF .....  
     ... 152, 152, 153, 11175,  
     26602, 26604, 26608, 29701, 29703  
 \file\_if\_exist\_input:n ... 245, 26135  
 \file\_if\_exist\_input:ntf .....  
     ... 245, 26135, 29700, 29702  
 \file\_input:n .....  
     153, 153, 245, 246, 11188, 29701, 29703  
 \file\_input\_stop: ..... 246, 26148  
 \file\_list: ..... 29704  
 \file\_log\_list: ... 153, 11280, 29705  
 \file\_parse\_full\_name:nNNN .....  
     ... 152, 11153, 11218, 11241  
 \file\_path\_include:n .....  
     ... 245, 245, 245, 245, 29706  
 \file\_path\_remove:n ..... 29708

\l_file_search_path_seq . . . . .	152, 152, 11065, 11140, 29707, 29709	\filemoddate . . . . .	882
\file_show_list: . . . . .	153, 11280	\filesize . . . . .	883
file internal commands:		\finalhyphendemerits . . . . .	374
\l_file_base_name_str . . . . .	11060, 11138, 11170, 11184, 11186	\firstmark . . . . .	375
\l_file_dir_str . . . . .	11062, 11154, 11219, 11220	\firstmarks . . . . .	632, 1446
\l_file_ext_str . . . . .	11062, 11154, 11155, 11219, 11222	\firstvalidlanguage . . . . .	922, 1744
\l_file_full_name_str . . . . .	11060, 11106, 11107, 11111, 11149, 11151, 11153, 11158, 11160, 11163, 11171, 11172, 11177, 11178, 11190, 11191, 11193, 26086, 26090, 26097, 26103, 26111, 26115, 26137, 26138, 26139, 26143, 26144, 26146	flag commands:	
\l_file_get_aux:nnN . . . . .	11099	\flag_clear:n . . . . .	93, 93, 7306, 7321, 21639, 23128, 23129
\l_file_get_details:nnN . . . . .	26078	\flag_clear_new:n . . . . .	93, 7320
\l_file_get_do:Nw . . . . .	11099	\flag_height:n . . . . .	94, 7329, 7360, 7375, 23138, 23139, 23145, 23146
\l_file_get_full_name_search:nN .	11136	\flag_if_exist:n . . . . .	94
\l_file_input:n . . . . .	11188, 26139, 26146	\flag_if_exist:nTF . . . . .	94, 7321, 7337, 7346
\l_file_input_pop: . . . . .	11188	\flag_if_exist_p:n . . . . .	94, 7346
\l_file_input_pop:nnn . . . . .	11188	\flag_if_raised:n . . . . .	94
\l_file_input_push:n . . . . .	11188	\flag_if_raised:nTF . . . . .	94, 7351, 21647
\g_file_internal_ior . . . . .	11013, 11157, 11159, 11164, 11172, 11173	\flag_if_raised_p:n . . . . .	94, 7351
\l_file_internal_t1 . . . . .	11012, 11074, 11075, 11077, 11078, 11079, 11081, 11229, 11230	\flag_log:n . . . . .	93, 7322
\l_file_list:N . . . . .	11280	\flag_new:n . . . . .	93,
\l_file_list_aux:n . . . . .	11280	93, 467, 7301, 7321, 13586, 13587, 13588, 13589, 21629, 23032, 23033	
\c_file_marker_t1 . . . . .	578, 11098, 11121, 11131	\flag_raise:n . . . . .	94, 7372, 23161, 23166
\l_file_name_quote:nN . . . . .	11069, 11169	\flag_raise_if_clear:n . . . . .	
\l_file_name_quote_aux:n . . . . .	11087, 11096	. . . . .	246, 13620, 13629, 13637, 13654, 13663, 13694, 21665, 21687, 26150
\l_file_name_sanitize_aux:n . . . . .	11069	\flag_show:n . . . . .	93, 7322
\l_file_name_str . . . . .	11062, 11154, 11219, 11221	flag fp commands:	
\l_file_parse_full_name_auxi:w . . . . .	11241	flag_fp_division_by_zero . . . . .	193, 13586
\l_file_parse_full_name_split:nNNNTF .	11241	flag_fp_invalid_operation . . . . .	193, 13586
\g_file_record_seq . . . . .	581, 583, 583, 11052, 11198, 11203, 11292, 11305, 11306	flag_fp_overflow . . . . .	193, 13586
\g_file_stack_seq . . . . .	581, 11028, 11212, 11229	flag_fp_underflow . . . . .	193, 13586
\l_file_tmp:w . . . . .	11031, 11035, 11039, 11045, 11049, 11261, 11276, 11278	flag internal commands:	
\l_file_tmp_seq . . . . .	11066, 11284, 11288, 11292, 11293, 11295, 11302, 11307	\_\_flag_chk_exist:n . . . . .	
\filedump . . . . .	881	. . . . .	7332, 7351, 7360, 26150

\fp\_abs:n . . . . . 197, 202, 823, 18954, 24137,  
   24239, 24241, 24243, 26030, 26032  
\fp\_add:Nn . . . . . 186, 823, 1027, 15403  
\fp\_compare:nNnTF . . . . . 188,  
   189, 189, 190, 190, 15459, 15600,  
   15606, 15611, 15619, 15680, 15686,  
   23994, 23996, 24001, 24270, 24285,  
   24294, 24813, 26004, 28222, 28609,  
   28615, 28821, 29197, 29210, 29255  
\fp\_compare:nTF . . . . .  
   . . . . . 189, 190, 190, 190, 190,  
   196, 15443, 15572, 15578, 15583, 15591  
\fp\_compare\_p:n . . . . . 189, 15443  
\fp\_compare\_p:nNn . . . . . 188, 15459  
\fp\_const:Nn . . . . .  
   185, 15380, 15426, 15427, 15428, 15429  
\fp\_do\_until:nn . . . . . 190, 15569  
\fp\_do\_until:nNnn . . . . . 189, 15597  
\fp\_do\_while:nn . . . . . 190, 15569  
\fp\_do\_while:nNnn . . . . . 189, 15597  
\fp\_eval:n . . . . . 186, 189,  
   196, 196, 196, 196, 196, 197, 197,  
   197, 197, 197, 197, 197, 198, 198,  
   198, 199, 199, 199, 200, 200, 200,  
   201, 201, 202, 202, 244, 244, 710,  
   18951, 26472, 26490, 28057, 28058,  
   28062, 28066, 28127, 28128, 28129,  
   28130, 28139, 28147, 28148, 28149,  
   28215, 28224, 28237, 28238, 28404,  
   28421, 28422, 28431, 28432, 28436,  
   28438, 28442, 28447, 28463, 28464,  
   28603, 28620, 28635, 28637, 28814,  
   28823, 28834, 28835, 29125, 29142,  
   29143, 29151, 29152, 29157, 29159,  
   29163, 29168, 29182, 29183, 29199,  
   29204, 29205, 29212, 29222, 29223,  
   29224, 29225, 29234, 29235, 29236,  
   29237, 29246, 29247, 29248, 29249,  
   29270, 29271, 29353, 29368, 29369,  
   29574, 29592, 29593, 29594, 29603,  
   29614, 29615, 29616, 29643, 29644  
\fp\_format:nn . . . . . 202  
\fp\_gadd:Nn . . . . . 186, 15403  
.fp\_gset:N . . . . . 172, 12506  
\fp\_gset:Nn . . . . . 186, 15380, 15404, 15406  
\fp\_gset\_eq:NN . . . . . 186, 15389, 15394  
\fp\_gsub:Nn . . . . . 186, 15403  
\fp\_gzero:N . . . . . 185, 15393, 15400  
\fp\_gzero\_new:N . . . . . 186, 15397  
\fp\_if\_exist:NTF . . . . .  
   . . . . . 188, 15398, 15400, 15441  
\fp\_if\_exist\_p:N . . . . . 188, 15441  
\fp\_if\_nan:nTF . . . . . 203  
\fp\_log:N . . . . . 193, 15413  
\fp\_log:n . . . . . 193, 15422  
\fp\_max:nn . . . . . 202, 18956  
\fp\_min:nn . . . . . 202, 18956  
\fp\_new:N . . . . . 185, 186,  
   15377, 15398, 15400, 15430, 15431,  
   15432, 15433, 23960, 23961, 23962,  
   24088, 24089, 24354, 24355, 25798,  
   25799, 25964, 25965, 28628, 28629  
.fp\_set:N . . . . . 172, 12506  
\fp\_set:Nn . . . . . 186, 15380, 15403,  
   15405, 23982, 23983, 23984, 24107,  
   24109, 24150, 24170, 24190, 24207,  
   24209, 24227, 24228, 24268, 24269,  
   24809, 24811, 25814, 25815, 25984,  
   25986, 26024, 26025, 28608, 28611  
\fp\_set\_eq:NN . . . . . 186, 15389, 15393,  
   24155, 24175, 24192, 24271, 24272  
\fp\_show:N . . . . . 193, 15413  
\fp\_show:n . . . . . 193, 15422  
\fp\_sign:n . . . . . 244, 26075  
\fp\_step\_function:nnnN . . . . .  
   . . . . . 191, 15625, 15717  
\fp\_step\_inline:nnnn . . . . . 191, 15695  
\fp\_step\_variable:nnnNn . . . . . 191, 15695  
\fp\_sub:Nn . . . . . 186, 15403  
\fp\_to\_decimal:N . . . . .  
   187, 188, 13579, 18757, 18788, 18951  
\fp\_to\_decimal:n . . . . .  
   . . . . . 186, 187, 187, 187, 188, 18757,  
   18953, 18955, 18957, 18959, 26076  
\fp\_to\_dim:N . . . . . 187, 821, 18880  
\fp\_to\_dim:n . . . . . 187, 192,  
   18880, 24026, 24037, 24137, 24819,  
   24850, 25901, 25909, 26040, 26042  
\fp\_to\_int:N . . . . . 187, 18896  
\fp\_to\_int:n . . . . . 187, 18896  
\fp\_to\_scientific:N . . . . .  
   . . . . . 187, 18703, 18734, 18741  
\fp\_to\_scientific:n . . . . . 187, 187, 18703  
\fp\_to\_tl:N . . . . .  
   . . . . . 187, 245, 13580, 15420, 18836  
\fp\_to\_tl:n . . . . . 187,  
   13189, 13619, 13628, 13653, 13662,  
   13691, 15259, 15274, 15423, 15425,  
   15652, 15653, 15672, 15683, 18836  
\fp\_trap:nn . . . . . 193, 193,  
   653, 13590, 13705, 13706, 13707, 13708  
\fp\_until\_do:nn . . . . . 190, 15569  
\fp\_until\_do:nNnn . . . . . 190, 15597  
\fp\_use:N . . . . .  
   188, 245, 18951, 28614, 28618, 28623  
\fp\_while\_do:nn . . . . . 190, 15569  
\fp\_while\_do:nNnn . . . . . 190, 15597  
\fp\_zero:N . . . . . 185, 186, 15393, 15398, 28610

\fp\_zero\_new:N . . . . . 186, [15397](#)  
\c\_inf\_fp . . . . .  
. . . . . 192, 201, [13200](#), 14796, 16188,  
16270, 16608, 17366, 17389, 17591,  
17594, 17598, 17622, 17917, 19442  
\c\_nan\_fp . . . . . 201, 656, 680,  
[13200](#), 13630, 13638, 13710, 13916,  
13935, 13941, 14122, 14130, 14139,  
14218, 14275, 14314, 14708, 14785,  
14797, 15261, 15276, 15676, 17565,  
19000, 19046, 19358, 19414, 19440  
\c\_one\_fp . . . . .  
. . . . . 191, 707, 808, 14800, 15204, 15225,  
15426, 15776, 16629, 17360, 17560,  
17612, 17841, 17871, 18420, 19062  
\c\_pi\_fp . . . . . 192, 201, 689, 14798, [15428](#)  
\g\_tmpa\_fp . . . . . 192, [15430](#)  
\l\_tmpa\_fp . . . . . 192, [15430](#)  
\g\_tmpb\_fp . . . . . 192, [15430](#)  
\l\_tmpb\_fp . . . . . 192, [15430](#)  
\c\_zero\_fp . . . . .  
. . . . . 191, 710, 724, 834, [13200](#), 13247,  
14801, 15216, 15228, 15378, 15393,  
15394, 15778, 15781, 16017, 16184,  
17369, 17390, 17588, 17625, 18635,  
18741, 18926, 19439, 23994, 23996,  
24001, 24285, 24294, 26004, 28222,  
28609, 28615, 28821, 29197, 29210  
fp internal commands:  
\fp\_&\_o:ww . . . . . 712, 720, [15782](#)  
\fp\_&\_tuple\_o:ww . . . . . [15782](#)  
\fp\_\*\_o:ww . . . . . 16149  
\fp\_\*\_tuple\_o:ww . . . . . 16655  
\fp+\_o:ww . . . . . 723, 723, 752, [15870](#)  
\fp-\_o:ww . . . . . 723, 723, [15865](#)  
\fp/\_o:ww . . . . . 732, 773, [16261](#)  
\fp^\_o:ww . . . . . 17556  
\fp\_acos\_o:w . . . . . 812, 815, [18576](#)  
\fp\_acot\_o:Nw . . . . . 17816, 17818, [18408](#)  
\fp\_acotii\_o:Nww . . . . . 18418, [18421](#)  
\fp\_acotii\_o:ww . . . . . 808  
\fp\_acsc\_normal\_o:NnwNnw . . . . .  
. . . . . 814, 18634, 18649, [18657](#)  
\fp\_acsc\_o:w . . . . . [18628](#)  
\fp\_add:NNNn . . . . . [15403](#)  
\fp\_add\_big\_i:wNww . . . . . 725  
\fp\_add\_big\_i\_o:wNww . . . . .  
. . . . . 723, 726, 15937, [15944](#)  
\fp\_add\_big\_ii:wNww . . . . . 725  
\fp\_add\_big\_ii\_o:wNww . . . . . 15940, [15944](#)  
\fp\_add\_inf\_o:Nww . . . . . 15886, [15906](#)  
\fp\_add\_normal\_o:Nww . . . . .  
. . . . . 725, 15885, [15921](#)  
\fp\_add\_npos\_o:NnwNnw . . . . .  
. . . . . 725, 15924, [15930](#)  
\fp\_add\_return\_ii\_o:Nww . . . . .  
. . . . . 15888, [15894](#), 15899  
\fp\_add\_significand\_carry\_o:wwwNN . . . . .  
727, 15977, [15992](#)  
\fp\_add\_significand\_no\_carry\_o:wwwNN . . . . .  
727, 15979, [15982](#)  
\fp\_add\_significand\_o:NnnwnnnN . . . . .  
726, 726, 15947, 15955, [15960](#)  
\fp\_add\_significand\_pack:NNNNNNN . . . . .  
. . . . . 15960  
\fp\_add\_significand\_test\_o:N . . . . . [15960](#)  
\fp\_add\_zeros\_o:Nww . . . . . 15884, [15896](#)  
\fp\_and\_return:wNw . . . . . [15782](#)  
\fp\_array\_bounds:NNnTF . . . . .  
. . . . . 19315, 19345, 19412  
\fp\_array\_bounds\_error>NNn . . . . . [19315](#)  
\fp\_array\_count:n . . . . . 13296,  
13900, 15526, 15527, 16668, 18676  
\fp\_array\_gset:NNNNww . . . . . [19334](#)  
\fp\_array\_gset:w . . . . . [19334](#)  
\fp\_array\_gset\_normal:w . . . . . [19334](#)  
\fp\_array\_gset\_recover:Nw . . . . . [19334](#)  
\fp\_array\_gset\_special:nnNNN . . . . .  
. . . . . 19334, 19390  
\fp\_array\_gzero:N . . . . . 834  
\fp\_array\_if\_all\_fp:nTF . . . . .  
. . . . . 13308, 15254  
\fp\_array\_if\_all\_fp\_loop:w . . . . . [13308](#)  
\g\_fp\_array\_int . . . . .  
. . . . . 19282, 19289, 19291, 19302  
\fp\_array\_item:N . . . . . [19396](#)  
\fp\_array\_item:NNNnN . . . . . [19396](#)  
\fp\_array\_item:NwN . . . . . [19396](#)  
\fp\_array\_item:w . . . . . [19396](#)  
\fp\_array\_item\_normal:w . . . . . [19396](#)  
\fp\_array\_item\_special:w . . . . . [19396](#)  
\l\_fp\_array\_loop\_int . . . . .  
. . . . . 19283, 19386, 19389, 19392  
\fp\_array\_new:nNNN . . . . . [19284](#)  
\fp\_array\_new:nNNNN . . . . . 19293, 19296  
\fp\_array\_to\_clist:n . . . . .  
. . . . . 13961, [18960](#), 19086  
\fp\_array\_to\_clist\_loop:Nw . . . . . [18960](#)  
\fp\_asec\_o:w . . . . . [18641](#)  
\fp\_asin\_auxi\_o:NnNww . . . . .  
. . . . . 813, 815, 18606, [18609](#), 18668  
\fp\_asin\_isqrt:wn . . . . . [18609](#)  
\fp\_asin\_normal\_o:NnwNnnnw . . . . .  
. . . . . 18567, 18583, [18594](#)  
\fp\_asin\_o:w . . . . . [18561](#)  
\fp\_atan\_auxi:w . . . . . 810, 18486, [18500](#)  
\fp\_atan\_auxii:w . . . . . [18500](#)

`\_fp_atan_combine_aux:ww . . . . .` [18527](#)  
`\_fp_atan_combine_o:NwwwN . . . . .` [808](#), [809](#), [18445](#), [18462](#), [18527](#)  
`\_fp_atan_default:w . . . . .` [707](#), [808](#), [18408](#)  
`\_fp_atan_div:wnwwn . . . . .` [809](#), [18473](#), [18475](#)  
`\_fp_atan_inf_o:NNNw . . . . .` [808](#), [18433](#),  
[18434](#), [18435](#), [18443](#), [18579](#), [18652](#)  
`\_fp_atan_near:wwn . . . . .` [18475](#)  
`\_fp_atan_near_aux:wwn . . . . .` [18475](#)  
`\_fp_atan_normal_o:NNnwN . . . . .` [808](#), [18437](#), [18453](#)  
`\_fp_atan_o:Nw . . . . .` [17820](#), [17822](#), [18408](#)  
`\_fp_atan_Taylor_break:w . . . . .` [18511](#)  
`\_fp_atan_Taylor_loop:ww . . . . .` [810](#), [18506](#), [18511](#)  
`\_fp_atan_test_o:NwwNwwN . . . . .` [814](#), [18456](#), [18460](#), [18616](#)  
`\_fp_atanii_o:Nww . . . . .` [18412](#), [18421](#)  
`\_fp_basics_pack_high:NNNNN . . . . .` [727](#), [743](#), [13406](#), [15985](#), [16137](#),  
[16240](#), [16252](#), [16394](#), [16587](#), [17113](#)  
`\_fp_basics_pack_high_carry:w . . . . .` [645](#), [13406](#)  
`\_fp_basics_pack_low:NNNNN . . . . .` [733](#), [743](#),  
[13406](#), [15987](#), [16139](#), [16242](#), [16254](#),  
[16396](#), [16536](#), [16538](#), [16589](#), [17115](#)  
`\_fp_basics_pack_weird_high:NNNNNNNN . . . . .` [203](#), [13417](#), [15996](#), [16405](#)  
`\_fp_basics_pack_weird_low:NNNNN . . . . .` [203](#), [13417](#), [15998](#), [16407](#)  
`\c_fp_big_leading_shift_int . . . . .` [13392](#), [16466](#), [16801](#), [16811](#), [16821](#)  
`\c_fp_big_middle_shift_int . . . . .` [13392](#), [16469](#), [16472](#), [16475](#),  
[16478](#), [16481](#), [16484](#), [16488](#), [16803](#),  
[16813](#), [16823](#), [16833](#), [16836](#), [16839](#)  
`\c_fp_big_trailing_shift_int . . . . .` [13392](#), [16492](#), [16846](#)  
`\c_fp_Bigg_leading_shift_int . . . . .` [13397](#), [16315](#), [16333](#)  
`\c_fp_Bigg_middle_shift_int . . . . .` [13397](#), [16318](#), [16321](#), [16336](#), [16339](#)  
`\c_fp_Bigg_trailing_shift_int . . . . .` [13397](#), [16324](#), [16342](#)  
`\_fp_binary_rev_type_o:Nww . . . . .` [14919](#), [16658](#), [16660](#)  
`\_fp_binary_type_o:Nww . . . . .` [14919](#), [16656](#), [16669](#)  
`\c_fp_block_int . . . . .` [13205](#), [17065](#)  
`\_fp_case_return:nw . . . . .` [648](#), [13474](#), [13504](#), [13507](#),  
[13512](#), [14021](#), [17325](#), [18433](#), [18434](#),  
[18435](#), [18728](#), [18782](#), [18856](#), [18858](#),  
[18859](#), [18926](#), [19363](#), [19365](#), [19366](#)  
`\_fp_case_return_i_o:ww . . . . .` [13481](#),  
[15887](#), [15901](#), [15910](#), [16182](#), [18424](#)  
`\_fp_case_return_ii_o:ww . . . . .` [13481](#), [16183](#), [17610](#), [17628](#), [18425](#)  
`\_fp_case_return_o:Nw . . . . .` [648](#),  
[649](#), [13475](#), [16608](#), [17360](#), [17365](#),  
[17368](#), [17560](#), [17565](#), [17588](#), [17591](#),  
[17594](#), [17841](#), [17871](#), [18635](#), [18637](#)  
`\_fp_case_return_o:Nww . . . . .` [13479](#), [16184](#), [16185](#),  
[16188](#), [16189](#), [17612](#), [17621](#), [17624](#)  
`\_fp_case_return_same_o:w . . . . .` [648](#), [649](#), [13477](#), [16417](#),  
[16421](#), [16609](#), [16621](#), [16624](#), [17144](#),  
[17372](#), [17585](#), [17826](#), [17834](#), [17849](#),  
[17864](#), [17879](#), [17886](#), [17894](#), [17909](#),  
[18564](#), [18572](#), [18590](#), [18636](#), [18653](#)  
`\_fp_case_use:nw . . . . .` [648](#), [13473](#), [15912](#), [16180](#), [16181](#),  
[16186](#), [16187](#), [16269](#), [16272](#), [16419](#),  
[16605](#), [17137](#), [17140](#), [17596](#), [17827](#),  
[17832](#), [17842](#), [17847](#), [17857](#), [17862](#),  
[17872](#), [17877](#), [17887](#), [17892](#), [17902](#),  
[17907](#), [18566](#), [18569](#), [18579](#), [18581](#),  
[18587](#), [18631](#), [18633](#), [18644](#), [18647](#),  
[18652](#), [18731](#), [18738](#), [18785](#), [18792](#)  
`\_fp_change_func_type:NNN . . . . .` [13336](#), [14712](#), [16651](#), [18713](#),  
[18767](#), [18844](#), [18890](#), [18905](#), [19347](#)  
`\_fp_change_func_type_aux:w . . . . .` [13336](#)  
`\_fp_change_func_type_chk:NNN . . . . .` [13336](#)  
`\_fp_chk:w . . . . .` [635](#), [636](#), [637](#), [689](#), [723](#), [725](#),  
[727](#), [733](#), [736](#), [13190](#), [13200](#),  
[13201](#), [13202](#), [13203](#), [13204](#), [13214](#),  
[13219](#), [13221](#), [13222](#), [13243](#), [13246](#),  
[13248](#), [13258](#), [13271](#), [13290](#), [13485](#),  
[13501](#), [13686](#), [13691](#), [13918](#), [13964](#),  
[13973](#), [13975](#), [14810](#), [15493](#), [15494](#),  
[15656](#), [15672](#), [15676](#), [15740](#), [15741](#),  
[15744](#), [15755](#), [15756](#), [15764](#), [15765](#),  
[15773](#), [15785](#), [15788](#), [15792](#), [15795](#),  
[15871](#), [15891](#), [15892](#), [15894](#), [15895](#),  
[15896](#), [15904](#), [15907](#), [15918](#), [15919](#),  
[15921](#), [15930](#), [16006](#), [16158](#), [16192](#),  
[16193](#), [16196](#), [16277](#), [16415](#), [16423](#),  
[16425](#), [16602](#), [16611](#), [16613](#), [16618](#),  
[16626](#), [16628](#), [16630](#), [16634](#), [17134](#),  
[17146](#), [17148](#), [17357](#), [17374](#), [17376](#),  
[17557](#), [17576](#), [17578](#), [17579](#), [17582](#),  
[17599](#), [17602](#), [17605](#), [17630](#), [17631](#),  
[17633](#), [17649](#), [17738](#), [17751](#), [17753](#)

$\text{\_fp\_divide\_n\_pos\_o:Nww}$  . . . . .  
 . . . . . 735, 736, 16266, [16276](#)  
 $\text{\_fp\_divide\_significand\_calc:wnnnnnnn}$   
 . . . . . 739,  
 739, 16293, [16302](#), 16350, 17217, 17224  
 $\text{\_fp\_divide\_significand\_calc\_i:wnnnnnnn}$   
 . . . . . 16302  
 $\text{\_fp\_divide\_significand\_calc\_ii:wnnnnnnn}$   
 . . . . . 16302  
 $\text{\_fp\_divide\_significand\_i\_o:wnw}$  . . . . .  
 . . . . . 736, 739, 16283, [16289](#)  
 $\text{\_fp\_divide\_significand\_ii:wnw}$  . . . . .  
 . . . . . 741, 16297, 16298, 16299, [16346](#)  
 $\text{\_fp\_divide\_significand\_iii:wnnnnn}$   
 . . . . . 741, 16300, [16353](#)  
 $\text{\_fp\_divide\_significand\_iv:wnnnnnnn}$   
 . . . . . 742, 16356, [16361](#)  
 $\text{\_fp\_divide\_significand\_large\_o:wwwNNNNwN}$  . . . . .  
 744, 16387, [16401](#)  
 $\text{\_fp\_divide\_significand\_pack:NNN}$  . . . . .  
 . . . . . 743,  
 775, 16348, [16381](#), 17204, 17222, 17230  
 $\text{\_fp\_divide\_significand\_small\_o:wwwNNNNwN}$  . . . . .  
 743, 16385, [16391](#)  
 $\text{\_fp\_divide\_significand\_test\_o:w}$  . . . . .  
 . . . . . 743, 743, 16291, [16382](#)  
 $\text{\_fp\_divide\_significand\_v:NN}$  . . . . .  
 . . . . . 16366, 16368, [16371](#)  
 $\text{\_fp\_divide\_significand\_v:NNw}$  . . . . .  
[16361](#)  
 $\text{\_fp\_divide\_significand\_vi:Nw}$  . . . . .  
 . . . . . 742, [16361](#)  
 $\text{\_fp\_division\_by\_zero\_o:Nnw}$  . . . . .  
 . . . . . 653, 13650,  
[13698](#), 16606, 17141, 17917, 17918  
 $\text{\_fp\_division\_by\_zero\_o:NNww}$  . . . . .  
 653, 13658, [13698](#), 16270, 16273, 17598  
 $\text{\_c\_fp\_empty\_tuple\_fp}$  . . . . .  
 . . . . . 13291, 14116, 14771, 14781  
 $\text{\_fp\_ep\_compare:wwww}$  . . . . .  
[16910](#), 18469  
 $\text{\_fp\_ep\_compare\_aux:wwww}$  . . . . .  
[16910](#)  
 $\text{\_fp\_ep\_div:wwwn}$  . . . . .  
 . . . . . 805, 16940, 17051,  
 18398, 18485, 18489, 18498, 18665  
 $\text{\_fp\_ep\_div\_eps\_pack:NNNNNw}$  . . . . .  
[16970](#)  
 $\text{\_fp\_ep\_div\_epsi:wnNNNNn}$  . . . . .  
 765  
 $\text{\_fp\_ep\_div\_epsi:wnNNNNNn}$  . . . . .  
 . . . . . 16967, [16970](#)  
 $\text{\_fp\_ep\_div\_epsi:wwNNNNNn}$  . . . . .  
[16970](#)  
 $\text{\_fp\_ep\_div\_esti:wwwn}$  . . . . .  
 . . . . . 765, 16946, [16949](#)  
 $\text{\_fp\_ep\_div\_estii:wwnnwwn}$  . . . . .  
[16949](#)  
 $\text{\_fp\_ep\_div\_estiii:NNNNNwwn}$  . . . . .  
[16949](#)  
 $\text{\_fp\_ep\_inv\_to\_float\_o:wN}$  . . . . .  
 794

`\_fp_ep_inv_to_float_o:wwN` . . . . .  
 .... [804](#), [17047](#), [17055](#), [17860](#), [17875](#)  
`\_fp_ep_isqrt:wwn` . . . . . [16993](#), [18626](#)  
`\_fp_ep_isqrt_aux:wwn` . . . . . [16993](#)  
`\_fp_ep_isqrt_auxi:wwn` [16996](#), [16998](#)  
`\_fp_ep_isqrt_auxii:wwnnwn` . [16993](#)  
`\_fp_ep_isqrt_epsi:wn` . . . . .  
 ..... [768](#), [17030](#), [17033](#)  
`\_fp_ep_isqrt_epsii:wwN` . . . . . [17033](#)  
`\_fp_ep_isqrt_esti:wwnnwn` . . . . .  
 ..... [17008](#), [17011](#)  
`\_fp_ep_isqrt_estii:wwnnwn` . [17011](#)  
`\_fp_ep_isqrt_estiii:NNNNNwwn` . . . . .  
 ..... [17011](#)  
`\_fp_ep_mul:wwwn` . . . . .  
 .. [16925](#), [18355](#), [18385](#), [18613](#), [18624](#)  
`\_fp_ep_mul_raw:wwwN` . . . . .  
 ..... [16925](#), [17939](#), [18305](#)  
`\_fp_ep_to_ep:wwN` . [16876](#), [16927](#),  
[16930](#), [16942](#), [16945](#), [16995](#), [18614](#)  
`\_fp_ep_to_ep_end:www` . . . . . [16876](#)  
`\_fp_ep_to_ep_loop:N` . . . . .  
 ..... [803](#), [16876](#), [18306](#)  
`\_fp_ep_to_ep_zero:ww` . . . . . [16876](#)  
`\_fp_ep_to_fixed:wwn` . . . . . [16858](#),  
[17936](#), [18492](#), [18501](#), [18611](#), [19106](#)  
`\_fp_ep_to_fixed_auxi:www` . . . [16858](#)  
`\_fp_ep_to_fixed_auxii:nnnnnnwn` . . . . .  
 ..... [16858](#)  
`\_fp_ep_to_float_o:wn` . . . . . [794](#)  
`\_fp_ep_to_float_o:wwN` . . . . . [791](#),  
[804](#), [17047](#), [17059](#), [17830](#), [17845](#), [18404](#)  
`\_fp_error:nnnn` . . . . . [13619](#),  
[13627](#), [13636](#), [13653](#), [13661](#), [13689](#),  
[13712](#), [13911](#), [13913](#), [13934](#), [13939](#),  
[14707](#), [15257](#), [15272](#), [15652](#), [15671](#),  
[15682](#), [18719](#), [18773](#), [18847](#), [19357](#)  
`\_fp_exp_after_?_f:nw` [643](#), [676](#), [14100](#)  
`\_fp_exp_after_any_f:Nnw` . . . [13361](#)  
`\_fp_exp_after_any_f:nw` . . . . .  
 .... [643](#), [13361](#), [13387](#), [14102](#), [14876](#)  
`\_fp_exp_after_array_f:w` . . . . .  
 ..... [643](#), [13372](#),  
[14761](#), [15822](#), [15833](#), [15843](#), [15851](#)  
`\_fp_exp_after_f:nw` . . . . .  
 .... [639](#), [676](#), [13248](#), [13366](#), [14809](#), [14947](#)  
`\_fp_exp_after_mark_f:nw` [676](#), [14100](#)  
`\_fp_exp_after_normal:nNNw` . . . . .  
 ..... [13251](#), [13261](#), [13278](#)  
`\_fp_exp_after_normal:Nwwww` . . . . .  
 ..... [13280](#), [13288](#)  
`\_fp_exp_after_o:w` . . . [639](#), [13248](#),  
[13478](#), [13482](#), [13484](#), [13971](#), [14015](#),  
 ..... [14033](#), [15239](#), [15772](#), [15790](#), [15799](#),  
[15808](#), [15895](#), [16632](#), [17750](#), [17755](#)  
`\_fp_exp_after_special:nNNw` . . . . .  
 ..... [640](#), [13253](#), [13263](#), [13268](#)  
`\_fp_exp_after_stop_f:nw` . . . . . [13361](#)  
`\_fp_exp_after_tuple_f:nw` . . . . .  
 ..... [13372](#), [15048](#)  
`\_fp_exp_after_tuple_o:w` . . . . .  
 .. [13372](#), [15797](#), [15800](#), [15803](#), [15805](#)  
`\c_fp_exp_intarray` . . . . .  
 .. [17450](#), [17536](#), [17543](#), [17546](#), [17548](#)  
`\_fp_exp_intarray:w` . . . . . [17507](#)  
`\_fp_exp_intarray_aux:w` . . . . . [17507](#)  
`\_fp_exp_large:NwN` [782](#), [17507](#), [17734](#)  
`\_fp_exp_large_after:wwn` [782](#), [17507](#)  
`\_fp_exp_normal_o:w` . . . [17362](#), [17376](#)  
`\_fp_exp_o:w` . . . . . [17122](#), [17357](#)  
`\_fp_exp_overflow>NN` . . . . . [17376](#)  
`\_fp_exp_pos_large:NnnNwn` . . . . .  
 ..... [17408](#), [17507](#)  
`\_fp_exp_pos_o:NNwnw` . . . . .  
 ..... [17379](#), [17381](#), [17384](#)  
`\_fp_exp_pos_o:Nnwnw` . . . . . [17376](#)  
`\_fp_exp_Taylor:Nnnwn` . . . . .  
 ..... [17404](#), [17423](#), [17553](#)  
`\_fp_exp_Taylor_break:Nww` . . . [17423](#)  
`\_fp_exp_Taylor_ii:ww` . [17429](#), [17432](#)  
`\_fp_exp_Taylor_loop:www` . . . [17423](#)  
`\_fp_expand:n` . . . . . [824](#), [13538](#), [18964](#)  
`\_fp_expand_loop:nwnN` . . . . . [13538](#)  
`\_fp_exponent:w` . . . . . [13222](#)  
`\c_fp_five_int` . . . . . [13773](#),  
[13797](#), [13810](#), [13823](#), [13830](#), [13883](#)  
`\_fp_fixed_(calculation):wwn` . . . [753](#)  
`\_fp_fixed_add:nnNnnwn` . . . . . [16751](#)  
`\_fp_fixed_add:Nnnnnwn` . . . . . [16751](#)  
`\_fp_fixed_add:wwn` . . . . . [754](#),  
[757](#), [16751](#), [16991](#), [17299](#), [17307](#),  
[17318](#), [17336](#), [18497](#), [18557](#), [19121](#)  
`\_fp_fixed_add_after:NNNNNwn` . [16751](#)  
`\_fp_fixed_add_one:wN` [754](#), [16683](#),  
[16984](#), [17440](#), [17449](#), [18623](#), [19112](#)  
`\_fp_fixed_add_pack:NNNNNwn` . [16751](#)  
`\_fp_fixed_continue:wn` . . . . .  
 ..... [16682](#), [16928](#),  
[16933](#), [16943](#), [17518](#), [17709](#), [17974](#),  
[18343](#), [18615](#), [18624](#), [19104](#), [19116](#)  
`\_fp_fixed_div_int:wnN` . . . . . [16720](#)  
`\_fp_fixed_div_int:wwN` . . . . .  
 ..... [755](#), [16720](#), [17298](#), [17439](#), [18516](#)  
`\_fp_fixed_div_int_after:Nw` . . . . .  
 ..... [756](#), [16720](#)  
`\_fp_fixed_div_int_auxi:wnn` . [16720](#)

$\backslash\text{--fp\_fixed\_div\_int\_auxii:wnn}$  . . . . .  
 ..... 756, [16720](#)  
 $\backslash\text{--fp\_fixed\_div\_int\_pack:Nw}$  . . . . .  
 ..... 756, [16720](#)  
 $\backslash\text{--fp\_fixed\_div\_myriad:wn}$  . . . . .  
 ..... 16688, [16988](#)  
 $\backslash\text{--fp\_fixed\_inv\_to\_float\_o:wn}$  . . . . .  
 ..... 17054, [17381](#), [17645](#)  
 $\backslash\text{--fp\_fixed\_mul:nnnnnnnw}$  . . . . .  
 ..... 16771  
 $\backslash\text{--fp\_fixed\_mul:wwn}$  . . . . .  
 .. 754, [755](#), [757](#), [802](#), [804](#), [16771](#),  
 16937, [16968](#), [16983](#), [16985](#), [16989](#),  
 17042, [17045](#), [17058](#), [17300](#), [17310](#),  
 17350, [17441](#), [17539](#), [17554](#), [17655](#),  
 18312, [18366](#), [18504](#), [18537](#), [18539](#)  
 $\backslash\text{--fp\_fixed\_mul\_add:nnnnwnnn}$  . . . . .  
 ..... 760, [16840](#), [16842](#)  
 $\backslash\text{--fp\_fixed\_mul\_add:nnnnwnnnw}$  . . . . .  
 ..... 761, [16847](#), [16853](#)  
 $\backslash\text{--fp\_fixed\_mul\_add:Nwnnnwnnn}$  . . . . .  
 .... 760, [16804](#), [16814](#), [16825](#), [16829](#)  
 $\backslash\text{--fp\_fixed\_mul\_add:wwnn}$  . . . . .  
 ..... 759, [16798](#), [19126](#)  
 $\backslash\text{--fp\_fixed\_mul\_after:wwn}$  . . . . .  
 ..... 758, [16690](#), [16696](#), [16699](#),  
 16773, [16800](#), [16810](#), [16820](#), [17672](#)  
 $\backslash\text{--fp\_fixed\_mul\_one\_minus\_}$   
 $\text{mul:wwn}$  . . . . .  
 ..... 16798  
 $\backslash\text{--fp\_fixed\_mul\_short:wwn}$  . . . . .  
 ..... 755, [16697](#),  
 16966, [16987](#), [17029](#), [17031](#), [18550](#)  
 $\backslash\text{--fp\_fixed\_mul\_sub\_back:wwnn}$  . . . . .  
 ..... 759, [16798](#),  
 17043, [18333](#), [18335](#), [18336](#), [18337](#),  
 18338, [18339](#), [18340](#), [18341](#), [18342](#),  
 18346, [18348](#), [18349](#), [18350](#), [18351](#),  
 18352, [18353](#), [18354](#), [18379](#), [18381](#),  
 18382, [18383](#), [18384](#), [18387](#), [18389](#),  
 18390, [18391](#), [18392](#), [18517](#), [18525](#)  
 $\backslash\text{--fp\_fixed\_one\_minus\_mul:wwn}$  . . . . .  
 ..... 759, [760](#), [16818](#)  
 $\backslash\text{--fp\_fixed\_sub:wwn}$  [16751](#), [17035](#),  
 17316, [17332](#), [17344](#), [17978](#), [18498](#),  
 18555, [18621](#), [19114](#), [19123](#), [19155](#)  
 $\backslash\text{--fp\_fixed\_to\_float\_o:Nw}$  . . . . .  
 ..... 17061, [17325](#)  
 $\backslash\text{--fp\_fixed\_to\_float\_o:wN}$  . . . . .  
 ..... 754, [769](#),  
 811, [17048](#), [17061](#), [17345](#), [17355](#),  
 17379, [17641](#), [18545](#), [19054](#), [19160](#)  
 $\backslash\text{--fp\_fixed\_to\_float\_pack:ww}$  . . . . .  
 ..... 17094, [17104](#)  
 $\backslash\text{--fp\_fixed\_to\_float\_rad\_o:wn}$  . . . . .  
 ..... 17056, [18545](#)  
 $\backslash\text{--fp\_fixed\_to\_float\_round\_}$   
 $\text{up:wnnnmw}$  . . . . .  
 ..... 17107, [17111](#)  
 $\backslash\text{--fp\_fixed\_to\_float\_zero:w}$  . . . . .  
 ..... 17090, [17099](#)  
 $\backslash\text{--fp\_fixed\_to\_loop:N}$  . . . . .  
 ..... 17067, [17077](#), [17081](#)  
 $\backslash\text{--fp\_fixed\_to\_loop\_end:w}$  . . . . .  
 ..... 17083, [17087](#)  
 $\backslash\text{--fp\_from\_dim:wNNnnnnnn}$  . . . . .  
 ..... 18915  
 $\backslash\text{--fp\_from\_dim:wnnnnwNn}$  [18943](#), [18944](#)  
 $\backslash\text{--fp\_from\_dim:wnnnnwNw}$  . . . . .  
 ..... 18915  
 $\backslash\text{--fp\_from\_dim:wNw}$  . . . . .  
 ..... 18915  
 $\backslash\text{--fp\_from\_dim\_test:ww}$  . . . . .  
 .... 822, [14194](#), [14231](#), [14828](#), [18915](#)  
 $\backslash\text{--fp\_func\_to\_name:N}$  . . . . .  
 ..... 13566, [14707](#), [14716](#)  
 $\backslash\text{--fp\_func\_to\_name\_aux:w}$  . . . . .  
 ..... 13566  
 $\backslash\text{c\_fp\_half\_prec\_int}$  . . . . .  
 ..... 13205, [14435](#), [14467](#)  
 $\backslash\text{--fp\_if\_type\_fp:NTwFw}$  . . . . .  
 707, [13307](#), [13315](#), [13322](#), [13338](#),  
 13365, [15266](#), [15280](#), [15451](#), [15477](#),  
 15478, [15645](#), [15646](#), [15647](#), [15813](#)  
 $\backslash\text{--fp\_inf\_fp:N}$  . . . . .  
 ..... 13218, [13674](#)  
 $\backslash\text{--fp\_int:wTF}$  . . . . .  
 ..... 13485, [19065](#)  
 $\backslash\text{--fp\_int\_eval:w}$  . . . . .  
 644, [659](#),  
 660, [661](#), [674](#), [690](#), [725](#), [733](#), [733](#),  
 734, [737](#), [741](#), [769](#), [13175](#), [13232](#),  
 13300, [13431](#), [13434](#), [13847](#), [13851](#),  
 13863, [13864](#), [13900](#), [13983](#), [13987](#),  
 14026, [14241](#), [14246](#), [14288](#), [14377](#),  
 14388, [14437](#), [14468](#), [14474](#), [14475](#),  
 14521, [14531](#), [14533](#), [14549](#), [14551](#),  
 14574, [14576](#), [14742](#), [14964](#), [15179](#),  
 15464, [15934](#), [15942](#), [15963](#), [15965](#),  
 15986, [15988](#), [15997](#), [15999](#), [16028](#),  
 16034, [16044](#), [16046](#), [16120](#), [16122](#),  
 16138, [16140](#), [16144](#), [16160](#), [16200](#),  
 16208, [16210](#), [16212](#), [16214](#), [16217](#),  
 16220, [16222](#), [16241](#), [16243](#), [16253](#),  
 16255, [16281](#), [16284](#), [16292](#), [16294](#),  
 16315, [16318](#), [16321](#), [16324](#), [16333](#),  
 16336, [16339](#), [16342](#), [16349](#), [16351](#),  
 16357, [16365](#), [16367](#), [16369](#), [16375](#),  
 16395, [16397](#), [16406](#), [16408](#), [16429](#),  
 16450, [16454](#), [16466](#), [16469](#), [16472](#),  
 16475, [16478](#), [16481](#), [16484](#), [16487](#),  
 16491, [16503](#), [16507](#), [16511](#), [16514](#),  
 16535, [16537](#), [16539](#), [16549](#), [16588](#),  
 16590, [16599](#), [16686](#), [16691](#), [16693](#),  
 16700, [16703](#), [16706](#), [16709](#), [16712](#),  
 16715, [16724](#), [16736](#), [16744](#), [16746](#),  
 16756, [16758](#), [16765](#), [16774](#), [16776](#),  
 16779, [16782](#), [16785](#), [16788](#), [16801](#)

16803, 16811, 16813, 16821, 16823,  
 16833, 16836, 16839, 16846, 16861,  
 16879, 16882, 16938, 16952, 16954,  
 16960, 16973, 16975, 16977, 17001,  
 17017, 17024, 17025, 17048, 17065,  
 17069, 17114, 17116, 17158, 17169,  
 17188, 17190, 17192, 17205, 17218,  
 17223, 17225, 17231, 17248, 17249,  
 17250, 17251, 17252, 17253, 17258,  
 17260, 17262, 17264, 17266, 17271,  
 17273, 17275, 17277, 17279, 17281,  
 17303, 17311, 17395, 17444, 17521,  
 17529, 17537, 17543, 17546, 17652,  
 17673, 17675, 17678, 17681, 17684,  
 17687, 17703, 17729, 17743, 17759,  
 17926, 17958, 17967, 18199, 18213,  
 18216, 18219, 18222, 18225, 18228,  
 18231, 18234, 18237, 18253, 18263,  
 18272, 18290, 18299, 18306, 18317,  
 18327, 18360, 18370, 18395, 18404,  
 18447, 18464, 18466, 18478, 18479,  
 18520, 18531, 18542, 18600, 18752,  
 18875, 18929, 19030, 19053, 19107,  
 19159, 19181, 19183, 19185, 19190,  
 19209, 19221, 19229, 19234, 19239  
 $\backslash\text{fp\_int\_eval\_end}$ : .....  
 . 13175, 13232, 13303, 13422, 13900,  
 13997, 14001, 15180, 15464, 16144,  
 16179, 16371, 16746, 16882, 17703,  
 17759, 17959, 17968, 18317, 18327,  
 18370, 18395, 18479, 19188, 19190  
 $\backslash\text{fp\_int\_p:w}$ : ..... 13485  
 $\backslash\text{fp\_int\_to\_roman:w}$ : ..... 13175,  
 13434, 14449, 14481, 17185, 19291  
 $\backslash\text{fp\_invalid\_operation:nw}$ : .....  
 . 653, 13616, 13698, 13710, 18733,  
 18740, 18787, 18794, 18894, 18909  
 $\backslash\text{fp\_invalid\_operation\_o:nw}$ : .....  
 . 653, 13709, 14716,  
 16419, 16645, 17137, 17833, 17848,  
 17863, 17878, 17893, 17908, 18570,  
 18588, 18604, 18632, 18645, 18661  
 $\backslash\text{fp\_invalid\_operation\_o:Nw}$ : .....  
 . 653, 13624, 13698,  
 14917, 15914, 16186, 16187, 17744  
 $\backslash\text{fp\_invalid\_operation\_o:nw}$ : 16670  
 $\backslash\text{fp\_invalid\_operation\_tl\_o:nn}$ : .....  
 . 653, 13633, 13698, 13959, 19085  
 $\backslash\text{c\_fp\_leading\_shift\_int}$ : .....  
 . 13388, 16691,  
 16700, 16774, 17673, 18253, 18290  
 $\backslash\text{fp\_ln\_c:NwNw}$ : 777, 777, 17282, 17313  
 $\backslash\text{fp\_ln\_div\_after:Nw}$ : .....  
 . 775, 17184, 17233  
 $\backslash\text{fp\_ln\_div\_i:w}$ : ..... 17206, 17215  
 $\backslash\text{fp\_ln\_div\_ii:wwn}$ : .....  
 . 17209, 17210, 17211, 17212, 17220  
 $\backslash\text{fp\_ln\_div\_vi:wwn}$ : ..... 17213, 17228  
 $\backslash\text{fp\_ln\_exponent:wn}$ : 778, 17160, 17322  
 $\backslash\text{fp\_ln\_exponent\_one:ww}$ : 17327, 17341  
 $\backslash\text{fp\_ln\_exponent\_small:NNww}$ : .....  
 . 17330, 17334, 17347  
 $\backslash\text{c\_fp\_ln\_i\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_ii\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_iii\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_iv\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_ix\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{fp\_ln\_npos\_o:w}$ : .....  
 . 771, 772, 17146, 17148  
 $\backslash\text{fp\_ln\_o:w}$ : .. 771, 786, 17124, 17134  
 $\backslash\text{fp\_ln\_significand:NNNNnnnn}$ : .....  
 . 772, 17159, 17162, 17653  
 $\backslash\text{fp\_ln\_square\_t\_after:w}$ : .....  
 . 17257, 17289  
 $\backslash\text{fp\_ln\_square\_t\_pack:NNNNNw}$ : .....  
 . 17259, 17261, 17263, 17265, 17287  
 $\backslash\text{fp\_ln\_t\_large:NNw}$ : .....  
 . 776, 17238, 17245, 17255  
 $\backslash\text{fp\_ln\_t\_small:Nw}$ : .. 17236, 17243  
 $\backslash\text{fp\_ln\_t\_small:w}$ : ..... 776  
 $\backslash\text{fp\_ln\_Taylor:wwNw}$ : 777, 17290, 17291  
 $\backslash\text{fp\_ln\_Taylor\_break:w}$ : 17296, 17307  
 $\backslash\text{fp\_ln\_Taylor\_loop:www}$ : .....  
 . 17292, 17293, 17302  
 $\backslash\text{fp\_ln\_twice\_t\_after:w}$ : 17270, 17286  
 $\backslash\text{fp\_ln\_twice\_t\_pack:Nw}$ : .. 17272,  
 17274, 17276, 17278, 17280, 17285  
 $\backslash\text{c\_fp\_ln\_vi\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_vii\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_viii\_fixed\_tl}$ : ..... 17125  
 $\backslash\text{c\_fp\_ln\_x\_fixed\_tl}$ : .....  
 . 17125, 17344, 17351  
 $\backslash\text{fp\_ln\_x\_ii:wnnnn}$ : .. 17164, 17182  
 $\backslash\text{fp\_ln\_x\_iii:NNNNNNw}$ : .. 17191, 17195  
 $\backslash\text{fp\_ln\_x\_iii\_var:NNNNNw}$ : .....  
 . 17189, 17197  
 $\backslash\text{fp\_ln\_x\_iv:wnnnnnnn}$ : .....  
 . 774, 17187, 17202  
 $\backslash\text{fp\_logb\_aux\_o:w}$ : ..... 16602  
 $\backslash\text{fp\_logb\_o:w}$ : ..... 15860, 16602  
 $\backslash\text{c\_fp\_max\_exp\_exponent\_int}$ : .....  
 . 13211, 17387  
 $\backslash\text{c\_fp\_max\_exponent\_int}$ : .. 13209,  
 13215, 13236, 16899, 17101, 17708  
 $\backslash\text{c\_fp\_middle\_shift\_int}$ : .....  
 . 13388, 16703,  
 16706, 16709, 16712, 16776, 16779,

16782, 16785, 17675, 17678, 17681,  
 17684, 18256, 18263, 18293, 18299  
 $\backslash\text{--fp\_minmax\_aux\_o:Nw}$  ..... 15726  
 $\backslash\text{--fp\_minmax\_auxi:ww}$  .....  
                        15748, 15760, 15767  
 $\backslash\text{--fp\_minmax\_auxii:ww}$  .....  
                        15750, 15758, 15767  
 $\backslash\text{--fp\_minmax\_break\_o:w}$  . 15741, 15771  
 $\backslash\text{--fp\_minmax\_loop:Nww}$  .....  
                        719, 15735, 15737, 15743  
 $\backslash\text{--fp\_minmax\_o:Nw}$  .....  
                        712, 15438, 15440, 15726  
 $\backslash\text{c\_--fp\_minus\_min\_exponent\_int}$  ...  
                        13209, 13237  
 $\backslash\text{--fp\_misused:n}$  . 13188, 13192, 13293  
 $\backslash\text{--fp\_mul\_cases\_o:NnNnw}$  .....  
                        735, 16151, 16157, 16263  
 $\backslash\text{--fp\_mul\_cases\_o:nNnnww}$  ..... 16157  
 $\backslash\text{--fp\_mul\_npos\_o:Nww}$  .....  
                        732, 733, 735, 822, 16154, 16195, 18946  
 $\backslash\text{--fp\_mul\_significand\_drop:NNNNNw}$   
                        733, 16204  
 $\backslash\text{--fp\_mul\_significand\_keep:NNNNNw}$   
                        16204  
 $\backslash\text{--fp\_mul\_significand\_large\_f:NwwNNNN}$  ..... 16234, 16238  
 $\backslash\text{--fp\_mul\_significand\_o:nnnnNnnnn}$   
                        733, 733, 16202, 16204  
 $\backslash\text{--fp\_mul\_significand\_small\_f:NNwwWN}$  ..... 16232, 16249  
 $\backslash\text{--fp\_mul\_significand\_test\_f:NNN}$   
                        734, 16206, 16229  
 $\backslash\text{c\_--fp\_myriad\_int}$  ..... 13208,  
                        16686, 16717, 16718, 16795, 16856  
 $\backslash\text{--fp\_neg\_sign:N}$  .....  
                        723, 13231, 15868, 16021  
 $\backslash\text{--fp\_not\_o:w}$  ..... 712, 14735, 15773  
 $\backslash\text{c\_--fp\_one\_fixed\_tl}$  ..... 16680,  
                        17298, 17511, 17709, 17736, 18449,  
                        18516, 18621, 19104, 19114, 19155  
 $\backslash\text{--fp\_overflow:w}$  .....  
                        639, 653, 655, 13239, 13698, 17389  
 $\backslash\text{c\_--fp\_overflowing_fp}$  .....  
                        13212, 18734, 18788  
 $\backslash\text{--fp\_pack:NNNNNw}$  .. 13388, 16692,  
                        16702, 16705, 16708, 16711, 16714,  
                        16775, 16778, 16781, 16784, 16787,  
                        17674, 17677, 17680, 17683, 17686  
 $\backslash\text{--fp\_pack\_big:NNNNNNw}$  ... 13392,  
                        16468, 16471, 16474, 16477, 16480,  
                        16483, 16486, 16490, 16802, 16812,  
                        16822, 16832, 16835, 16838, 16845  
 $\backslash\text{--fp\_pack\_Bigg:NNNNNNw}$  .....  
                        13397, 16317,  
                        16320, 16323, 16335, 16338, 16341  
 $\backslash\text{--fp\_pack\_eight:wNNNNNNNN}$  .....  
                        645, 730, 13404,  
                        16130, 16439, 16867, 17945, 17946  
 $\backslash\text{--fp\_pack\_twice\_four:wNNNNNNNN}$  .  
                        645, 13402, 14008, 14009,  
                        16072, 16073, 16868, 16869, 16870,  
                        16902, 16903, 16904, 17092, 17093,  
                        17426, 17427, 17428, 17947, 17948,  
                        18242, 18243, 18244, 18245, 18939  
 $\backslash\text{--fp\_parse:n}$  ..... 665,  
                        677, 689, 697, 709, 710, 716, 823,  
                        834, 1027, 14039, 14191, 14852,  
                        15381, 15383, 15385, 15408, 15446,  
                        15463, 15473, 15630, 15690, 16615,  
                        18709, 18763, 18841, 18886, 18901,  
                        18955, 18957, 18959, 19341, 26076  
 $\backslash\text{--fp\_parse\_after:ww}$  ..... 14852  
 $\backslash\text{--fp\_parse\_apply\_binary:NwNwN}$  ...  
                        669, 670, 673, 674, 701, 14890, 15058  
 $\backslash\text{--fp\_parse\_apply\_binary\_chk:NN}$  .  
                        14890, 14921, 14934  
 $\backslash\text{--fp\_parse\_apply\_binary\_error:NNN}$  ..... 14890  
 $\backslash\text{--fp\_parse\_apply\_comma:NwNwN}$  ...  
                        701, 15017  
 $\backslash\text{--fp\_parse\_apply\_compare:NwNNNNNwN}$   
                        15203, 15212  
 $\backslash\text{--fp\_parse\_apply\_compare\_aux:NNwN}$  ..... 15224, 15227, 15232  
 $\backslash\text{--fp\_parse\_apply\_function:NNNwN}$   
                        693, 14684, 14845  
 $\backslash\text{--fp\_parse\_apply\_unary:NNNwN}$  ...  
                        14689, 14721, 14836  
 $\backslash\text{--fp\_parse\_apply\_unary\_chk:nNNNNw}$   
                        14700, 14701, 14704  
 $\backslash\text{--fp\_parse\_apply\_unary\_chk:nNNNw}$   
                        14689  
 $\backslash\text{--fp\_parse\_apply\_unary\_chk:NwNw}$   
                        14689  
 $\backslash\text{--fp\_parse\_apply\_unary\_error:NNw}$  ...  
                        14689, 16652  
 $\backslash\text{--fp\_parse\_apply\_unary\_type:NNN}$   
                        14689  
 $\backslash\text{--fp\_parse\_caseless\_inf:N}$  ... 14802  
 $\backslash\text{--fp\_parse\_caseless\_infinity:N}$  .  
                        14802  
 $\backslash\text{--fp\_parse\_caseless\_nan:N}$  ... 14802  
 $\backslash\text{--fp\_parse\_compare:NNNNNN}$  .. 15144  
 $\backslash\text{--fp\_parse\_compare\_auxi:NNNNNN}$   
                        15144

`\_fp_parse_compare_auxii:NNNN .` ..... 15144  
`\_fp_parse_compare_end:NNNNw .` 15144  
`\_fp_parse_continue:NwN .` ..... 670, 698, 14879, 14892,  
 15045, 15242, 15830, 15840, 15848  
`\_fp_parse_continue_compare:NNwNN .` ..... 15235, 15250  
`\_fp_parse_digits:_N .` ..... 14056  
`\_fp_parse_digits_i:N .` ..... 14056  
`\_fp_parse_digits_ii:N .` ..... 14056  
`\_fp_parse_digits_iii:N .` ..... 14056  
`\_fp_parse_digits_iv:N .` ..... 14056  
`\_fp_parse_digits_v:N .` ..... 14056  
`\_fp_parse_digits_vi:N .` ..... 14056, 14393, 14441  
`\_fp_parse_digits_vii:N .` ..... 683, 14056, 14380, 14430  
`\_fp_parse_excl_error: .` ..... 15144  
`\_fp_parse_expand:w .` ..... 673, 673, 674, 674, 14053, 14055,  
 14065, 14105, 14167, 14211, 14220,  
 14223, 14227, 14264, 14298, 14336,  
 14338, 14357, 14359, 14381, 14398,  
 14411, 14431, 14461, 14489, 14505,  
 14516, 14539, 14568, 14578, 14585,  
 14598, 14614, 14634, 14645, 14731,  
 14754, 14766, 14841, 14850, 14858,  
 14871, 14994, 15012, 15036, 15062,  
 15108, 15128, 15197, 15210, 15826  
`\_fp_parse_exponent:N .` ..... 687, 14166, 14372, 14521, 14588, 14590  
`\_fp_parse_exponent:Nw .` ..... 14396, 14409,  
 14458, 14486, 14537, 14566, 14585  
`\_fp_parse_exponent_aux:N .` 14590  
`\_fp_parse_exponent_body:N .` ..... 14616, 14620  
`\_fp_parse_exponent_digits:N .` ..... 14624, 14636  
`\_fp_parse_exponent_keep:N .` 14647  
`\_fp_parse_exponent_keep:NTF .` ..... 14627, 14647  
`\_fp_parse_exponent_sign:N .` ..... 14606, 14610  
`\_fp_parse_function:NNN .` ..... 13759, 13761, 13763,  
 13766, 14834, 15438, 15440, 17816,  
 17818, 17820, 17822, 18989, 18991  
`\_fp_parse_function_all_fp_o:nw .` ..... 13893, 15252, 15728  
`\_fp_parse_function_one_two:nnw .` 807, 808, 15264, 18410, 18416, 19058  
`\_fp_parse_function_one_two_- aux:nw .` ..... 15264  
`\_fp_parse_function_one_two_- auxii:nw .` ..... 15264  
`\_fp_parse_function_one_two_- error_o:w .` ..... 15264  
`\_fp_parse_infix>NN .` ..... 676, 679, 695, 700, 14104,  
 14276, 14315, 14763, 14789, 14794,  
 14809, 14831, 14947, 14950, 15010  
`\_fp_parse_infix_!:_N .` ..... 15144  
`\_fp_parse_infix_&:Nw .` ..... 15101  
`\_fp_parse_infix_(:_N .` ..... 15084  
`\_fp_parse_infix_(:N .` ..... 15000  
`\_fp_parse_infix_*:_N .` ..... 15086  
`\_fp_parse_infix_+:_N .` ..... 673, 674, 14053, 15052  
`\_fp_parse_infix_,:N .` ..... 15017  
`\_fp_parse_infix_:-:N .` ..... 15052  
`\_fp_parse_infix_/_:N .` ..... 15052  
`\_fp_parse_infix_::N .` ..... 15118, 15811  
`\_fp_parse_infix_<:_N .` ..... 15144  
`\_fp_parse_infix_=:_N .` ..... 15144  
`\_fp_parse_infix_>:N .` ..... 15144  
`\_fp_parse_infix_?:N .` ..... 15118  
`\_fp_parse_infix_<operation>:N .` 673  
`\_fp_parse_infix_~:_N .` ..... 15052  
`\_fp_parse_infix_after_operand:NwN .` ..... 679, 14159, 14237, 14738, 14945  
`\_fp_parse_infix_and:N .` 15052, 15117  
`\_fp_parse_infix_check:NNN .` ..... 14970, 14980  
`\_fp_parse_infix_comma:w .` 701, 15017  
`\_fp_parse_infix_end:N .` ..... 697, 701, 14859, 14864, 14872, 14998  
`\_fp_parse_infix_mark:NNN .` ..... 14957, 14997  
`\_fp_parse_infix_mul:N .` ..... 703, 14960, 14968, 15052, 15085, 15094  
`\_fp_parse_infix_or:N .` 15052, 15116  
`\_fp_parse_infix_|:Nw .` ..... 15101  
`\_fp_parse_large:N .` 681, 14343, 14426  
`\_fp_parse_large_leading:wwNN .` ..... 685, 14428, 14433  
`\_fp_parse_large_round:NN .` ..... 686, 14469, 14541  
`\_fp_parse_large_round_aux:wNN .` ..... 14541  
`\_fp_parse_large_round_test:NN .` ..... 14541  
`\_fp_parse_large_trailing:wwNN .` ..... 686, 14439, 14463  
`\_fp_parse_letters:N .` ..... 679, 680, 14252, 14266

`\_fp_parse_lparen_after:NwN .` [14744](#)  
`\_fp_parse_o:n .....`  
`.....` [665](#), [14852](#), [15628](#), [15629](#)  
`\_fp_parse_one:Nw .....`  
`.....` [669](#)–[672](#), [674](#), [681](#), [695](#),  
[698](#), [14053](#), [14076](#), [14320](#), [14683](#), [14885](#)  
`\_fp_parse_one_digit>NN .....`  
`.....` [694](#), [14092](#), [14235](#)  
`\_fp_parse_one_fp>NN .....`  
`.....` [675](#), [14084](#), [14100](#)  
`\_fp_parse_one_other>NN` [14095](#), [14243](#)  
`\_fp_parse_one_register>NN .....`  
`.....` [14087](#), [14157](#)  
`\_fp_parse_one_register_aux:Nw .....`  
`.....` [14157](#)  
`\_fp_parse_one_register_- auxii:wwwNw .....`  
`.....` [14157](#)  
`\_fp_parse_one_register_dim:ww .....`  
`.....` [14157](#)  
`\_fp_parse_one_register_int:www .....`  
`.....` [14157](#)  
`\_fp_parse_one_register_- math>NNw .....`  
`.....` [14198](#)  
`\_fp_parse_one_register_mu:www .....`  
`.....` [14157](#)  
`\_fp_parse_one_register_- special:N .....`  
`.....` [14162](#), [14198](#)  
`\_fp_parse_one_register_wd:Nw` [14198](#)  
`\_fp_parse_one_register_wd:ww` [14198](#)  
`\_fp_parse_operand:Nw .....`  
`....` [669](#), [671](#), [672](#), [673](#), [697](#), [701](#),  
[14053](#), [14727](#), [14729](#), [14750](#), [14752](#),  
[14841](#), [14850](#), [14857](#), [14870](#), [14879](#),  
[15035](#), [15061](#), [15127](#), [15210](#), [15825](#)  
`\_fp_parse_pack_carry:ww .` [684](#), [14413](#)  
`\_fp_parse_pack_leading:NNNNNww .....`  
`.....` [14376](#), [14413](#), [14436](#)  
`\_fp_parse_pack_trailing:NNNNNNww .....`  
`.....` [14386](#), [14413](#), [14455](#), [14466](#), [14473](#)  
`\_fp_parse_prefix:NNN .` [14255](#), [14300](#)  
`\_fp_parse_prefix_!:_Nw .....` [14717](#)  
`\_fp_parse_prefix_(:_Nw .....` [14744](#)  
`\_fp_parse_prefix_):_Nw .....` [14776](#)  
`\_fp_parse_prefix_+:_Nw .....` [14683](#)  
`\_fp_parse_prefix_-:_Nw .....` [14717](#)  
`\_fp_parse_prefix_:_Nw .....` [14736](#)  
`\_fp_parse_prefix_unknown:NNN` [14300](#)  
`\_fp_parse_return_semicolon:ww .....`  
`.....` [14054](#), [14063](#), [14296](#),  
[14503](#), [14514](#), [14596](#), [14628](#), [14643](#)  
`\_fp_parse_round:Nw .....` [13764](#)  
`\_fp_parse_round_after:wN .....`  
`.....` [688](#), [14518](#), [14523](#), [14573](#)  
`\_fp_parse_round_loop:N .....` [687](#),  
`.....` [688](#), [14491](#), [14534](#), [14552](#), [14577](#)  
`\_fp_parse_round_up:N .....` [14491](#)  
`\_fp_parse_small:N` [682](#), [14363](#), [14374](#)  
`\_fp_parse_small_leading:wwNN .....`  
`.....` [683](#), [14378](#), [14383](#), [14445](#)  
`\_fp_parse_small_round:NN .....`  
`.....` [14405](#), [14523](#), [14562](#)  
`\_fp_parse_small_trailing:wwNN .....`  
`.....` [684](#), [14391](#), [14400](#), [14477](#)  
`\_fp_parse_strim_end:w .....` [14349](#)  
`\_fp_parse_strim_zeros:N .....`  
`....` [681](#), [694](#), [14330](#), [14349](#), [14742](#)  
`\_fp_parse_trim_end:w .....` [14323](#)  
`\_fp_parse_trim_zeros:N` [14241](#), [14323](#)  
`\_fp_parse_unary_function:NNN .....`  
`....` [14834](#), [15858](#), [15860](#), [15862](#),  
[15864](#), [17122](#), [17124](#), [17804](#), [17810](#)  
`\_fp_parse_word:Nw` [679](#), [14249](#), [14266](#)  
`\_fp_parse_word_abs:N .....` [15857](#)  
`\_fp_parse_word_acos:N .....` [17796](#)  
`\_fp_parse_word_acosd:N .....` [17796](#)  
`\_fp_parse_word_acot:N .....` [17815](#)  
`\_fp_parse_word_acotd:N .....` [17815](#)  
`\_fp_parse_word_acsc:N .....` [17796](#)  
`\_fp_parse_word_acscd:N .....` [17796](#)  
`\_fp_parse_word_asec:N .....` [17796](#)  
`\_fp_parse_word_asecd:N .....` [17796](#)  
`\_fp_parse_word_asin:N .....` [17796](#)  
`\_fp_parse_word_asind:N .....` [17796](#)  
`\_fp_parse_word_atan:N .....` [17815](#)  
`\_fp_parse_word_atand:N .....` [17815](#)  
`\_fp_parse_word_bp:N .....` [14805](#)  
`\_fp_parse_word_cc:N .....` [14805](#)  
`\_fp_parse_word.ceil:N .....` [13758](#)  
`\_fp_parse_word_cm:N .....` [14805](#)  
`\_fp_parse_word_cos:N .....` [17796](#)  
`\_fp_parse_word_cosd:N .....` [17796](#)  
`\_fp_parse_word_cot:N .....` [17796](#)  
`\_fp_parse_word_cotd:N .....` [17796](#)  
`\_fp_parse_word_csc:N .....` [17796](#)  
`\_fp_parse_word_cscd:N .....` [17796](#)  
`\_fp_parse_word_dd:N .....` [14805](#)  
`\_fp_parse_word_deg:N .....` [14791](#)  
`\_fp_parse_word_em:N .....` [14824](#)  
`\_fp_parse_word_ex:N .....` [14824](#)  
`\_fp_parse_word_exp:N .....` [17121](#)  
`\_fp_parse_word_false:N .....` [14791](#)  
`\_fp_parse_word_floor:N .....` [13758](#)  
`\_fp_parse_word_in:N .....` [14805](#)  
`\_fp_parse_word_inf:N .....`  
`.....` [14791](#), [14802](#), [14803](#)  
`\_fp_parse_word_ln:N .....` [17121](#)  
`\_fp_parse_word_logb:N .....` [15857](#)

`\_fp_parse_word_max:N` ..... 15437  
`\_fp_parse_word_min:N` ..... 15437  
`\_fp_parse_word_mm:N` ..... 14805  
`\_fp_parse_word_nan:N` ..... 14791, 14804  
`\_fp_parse_word_nc:N` ..... 14805  
`\_fp_parse_word_nd:N` ..... 14805  
`\_fp_parse_word_pc:N` ..... 14805  
`\_fp_parse_word_pi:N` ..... 14791  
`\_fp_parse_word_pt:N` ..... 14805  
`\_fp_parse_word_rand:N` ..... 18988  
`\_fp_parse_word_randint:N` ..... 18988  
`\_fp_parse_word_round:N` ..... 13764  
`\_fp_parse_word_sec:N` ..... 17796  
`\_fp_parse_word_secd:N` ..... 17796  
`\_fp_parse_word_sign:N` ..... 15857  
`\_fp_parse_word_sin:N` ..... 17796  
`\_fp_parse_word_sind:N` ..... 17796  
`\_fp_parse_word_sp:N` ..... 14805  
`\_fp_parse_word_sqrt:N` ..... 15857  
`\_fp_parse_word_tan:N` ..... 17796  
`\_fp_parse_word_tand:N` ..... 17796  
`\_fp_parse_word_true:N` ..... 14791  
`\_fp_parse_word_trunc:N` ..... 13758  
`\_fp_parse_zero:` .....  
      ..... 681, 14345, 14365, 14369  
`\_fp_pow_B:wwN` ..... 17656, 17691  
`\_fp_pow_C_neg:w` ..... 17694, 17711  
`\_fp_pow_C_overflow:w` .....  
      ..... 17699, 17706, 17727  
`\_fp_pow_C_pack:w` ..... 17713, 17721, 17732  
`\_fp_pow_C_pos:w` ..... 17697, 17716  
`\_fp_pow_C_pos_loop:wN` .....  
      ..... 17717, 17718, 17725  
`\_fp_pow_exponent:Nwnnnnnw` .....  
      ..... 17662, 17665, 17670  
`\_fp_pow_exponent:wnN` ..... 17654, 17659  
`\_fp_pow_neg:www` ..... 788, 17567, 17738  
`\_fp_pow_neg_aux:wNN` ..... 788, 17738  
`\_fp_pow_neg_case:w` ..... 17740, 17761  
`\_fp_pow_neg_case_aux:nnnn` ..... 17761  
`\_fp_pow_neg_case_aux:Nnnw` .....  
      ..... 789, 17761  
`\_fp_pow_normal_o:ww` .....  
      ..... 784, 17572, 17604  
`\_fp_pow_npos_aux:NNnww` .....  
      ..... 17639, 17643, 17649  
`\_fp_pow_npos_o:Nww` ..... 785, 17616, 17633  
`\_fp_pow_zero_or_inf:ww` .....  
      ..... 784, 17574, 17581  
`\c_fp_prec_and_int` ..... 14039, 15081  
`\c_fp_prec_colon_int` .....  
      ..... 14039, 15139, 15825  
`\c_fp_prec_comma_int` .....  
      ..... 694, 14039, 14112,  
      14750, 14778, 15021, 15026, 15035  
`\c_fp_prec_comp_int` .....  
      ..... 14039, 15167, 15210  
`\c_fp_prec_end_int` ..... 697,  
      701, 14039, 14114, 14857, 14870, 15004  
`\c_fp_prec_func_int` .....  
      ..... 694, 14039, 14749, 14841, 14850  
`\c_fp_prec_hat_int` ..... 14039, 15071  
`\c_fp_prec_hatii_int` ..... 14039, 15071  
`\c_fp_prec_int` .....  
      ..... 13205, 13431, 13492, 13519, 13977,  
      17407, 17773, 17776, 18806, 18808,  
      18814, 18865, 19069, 19108, 19159  
`\c_fp_prec_not_int` .....  
      ..... 693, 14039, 14734, 14735  
`\c_fp_prec_or_int` ..... 14039, 15083  
`\c_fp_prec_plus_int` .....  
      ..... 668, 14039, 15077, 15079  
`\c_fp_prec_quest_int` .....  
      ..... 14039, 15122, 15137  
`\c_fp_prec_times_int` .....  
      ..... 14039, 15073, 15075  
`\c_fp_prec_tuple_int` .....  
      ..... 694, 14039, 14113, 14752, 14780  
`\_fp_rand_myriads:n` .....  
      ..... 828, 829, 19024, 19041, 19127  
`\_fp_rand_myriads_get:w` ..... 19024  
`\_fp_rand_myriads_loop:w` ..... 19024  
`\_fp_rand_o:Nw` .....  
      ..... 18989, 18996, 19002, 19035  
`\_fp_rand_o:w` ..... 19035  
`\_fp_randinat_wide_aux:w` ..... 19197  
`\_fp_randinat_wide_auxii:w` ..... 19197  
`\_fp_randint:n` ..... 19259  
`\_fp_randint:ww` ..... 19165, 19269  
`\_fp_randint_auxi_o:ww` ..... 19056  
`\_fp_randint_auxii:wn` ..... 19056  
`\_fp_randint_auxiii_o:ww` ..... 19056  
`\_fp_randint_auxiv_o:ww` ..... 19056  
`\_fp_randint_auxv_o:w` ..... 19056  
`\_fp_randint_badarg:w` ..... 828, 19056  
`\_fp_randint_default:w` ..... 19056  
`\_fp_randint_o:Nw` ..... 18991, 19002, 19056  
`\_fp_randint_o:w` ..... 19056  
`\_fp_randint_split_aux:w` ..... 19197  
`\_fp_randint_split_o:Nw` ..... 832, 19197  
`\_fp_randint_wide_aux:w` .....  
      ..... 832, 19200, 19231  
`\_fp_randint_wide_auxii:w` .....  
      ..... 19233, 19242

$\text{\_fp\_reverse\_args:Nww}$  . . . . .  
 . . . . . 814, 815, 13184,  
 18396, 18471, 18584, 18650, 19153  
 $\text{\_fp\_round:NNN}$  658, 659, 660, 735,  
 750, 13774, 13844, 15989, 16000,  
 16244, 16256, 16398, 16409, 16593  
 $\text{\_fp\_round:Nwn}$  . 13902, 13955, 18913  
 $\text{\_fp\_round:Nww}$  . 13903, 13924, 13955  
 $\text{\_fp\_round:Nwww}$  . . . . . 13904, 13918  
 $\text{\_fp\_round\_aux\_o:Nw}$  . . . . . 13891  
 $\text{\_fp\_round\_digit:Nw}$  . . . . . 647, 661,  
 733, 735, 750, 13449, 13858, 16003,  
 16146, 16247, 16259, 16412, 16598  
 $\text{\_fp\_round\_name\_from\_cs:N}$  . . . . .  
 . . . . . 13894, 13914, 13940, 13944, 13960  
 $\text{\_fp\_round\_neg:NNN}$  . . . . . 659,  
 661, 731, 13869, 16108, 16123, 16141  
 $\text{\_fp\_round\_no\_arg\_o:Nw}$  13901, 13908  
 $\text{\_fp\_round\_normal:NnnwNNnn}$  . . 13955  
 $\text{\_fp\_round\_normal:NNwNnn}$  . . 13955  
 $\text{\_fp\_round\_normal:NwNNnw}$  . . 13955  
 $\text{\_fp\_round\_normal\_end:wwNnn}$  . 13955  
 $\text{\_fp\_round\_o:Nw}$  . . . . .  
 . . . . . 13759, 13761, 13763, 13767, 13891  
 $\text{\_fp\_round\_pack:Nw}$  . . . . . 13955  
 $\text{\_fp\_round\_return\_one:}$  . . . . .  
 . . . . . 659, 13774, 13780,  
 13790, 13798, 13802, 13811, 13815,  
 13824, 13831, 13835, 13873, 13884  
 $\text{\_fp\_round\_s:NNNw}$  . . . . .  
 . . . . . 659, 660, 687, 13842, 14527, 14545  
 $\text{\_fp\_round\_special:NwwNnn}$  . . 13955  
 $\text{\_fp\_round\_special\_aux:Nw}$  . . 13955  
 $\text{\_fp\_round\_to\_nearest:NNN}$  . . . . .  
 . . . . . 662, 663, 13767, 13770,  
 13774, 13878, 13910, 13920, 18913  
 $\text{\_fp\_round\_to\_nearest\_neg:NNN}$  13869  
 $\text{\_fp\_round\_to\_nearest\_ninf:NNN}$  . . . . .  
 . . . . . 663, 13774, 13889  
 $\text{\_fp\_round\_to\_nearest\_ninf\_ - neg:NNN}$  . . . . . 13869  
 $\text{\_fp\_round\_to\_nearest\_pinf:NNN}$  . . . . .  
 . . . . . 663, 13774, 13880  
 $\text{\_fp\_round\_to\_nearest\_pinf\_ - neg:NNN}$  . . . . . 13869  
 $\text{\_fp\_round\_to\_nearest\_zero:NNN}$  . . . . .  
 . . . . . 663, 13774  
 $\text{\_fp\_round\_to\_nearest\_zero\_ - neg:NNN}$  . . . . . 13869  
 $\text{\_fp\_round\_to\_ninf:NNN}$  . . . . .  
 . . . . . 13761, 13774, 13877, 13948  
 $\text{\_fp\_round\_to\_ninf\_neg:NNN}$  . . 13869  
 $\text{\_fp\_round\_to\_pinf:NNN}$  . . . . .  
 . . . . . 13763, 13774, 13869, 13950  
 $\text{\_fp\_round\_to\_pinf\_neg:NNN}$  . . 13869  
 $\text{\_fp\_round\_to\_zero:NNN}$  . . . . .  
 . . . . . 13759, 13774, 13946  
 $\text{\_fp\_round\_to\_zero\_neg:NNN}$  . . 13869  
 $\text{\_fp\_rrot:www}$  . . . . . 13185, 18517  
 $\text{\_fp\_sanitize:Nw}$  . . . . . 725, 728,  
 733, 736, 744, 804, 811, 829, 13233,  
 14016, 14034, 15932, 16026, 16198,  
 16279, 16427, 17150, 17393, 17635,  
 18358, 18402, 18529, 19051, 19146  
 $\text{\_fp\_sanitize:wN}$  . . . . .  
 . . . . . 679, 682, 13233, 14240, 14741  
 $\text{\_fp\_sanitize\_zero:w}$  . . . . . 13233  
 $\text{\_fp\_sec\_o:w}$  . . . . . 17868  
 $\text{\_fp\_set\_sign\_o:w}$  . . . . .  
 . . . . . 14734, 15858, 16629, 16630, 16651  
 $\text{\_fp\_show:NN}$  . . . . . 15413  
 $\text{\_fp\_sign\_aux\_o:w}$  . . . . . 16618  
 $\text{\_fp\_sign\_o:w}$  . . . . . 15862, 16618  
 $\text{\_fp\_sin\_o:w}$  652, 692, 693, 813, 17823  
 $\text{\_fp\_sin\_series\_aux\_o:NNnwww}$  18310  
 $\text{\_fp\_sin\_series\_o:NNwww}$  . . 791,  
 805, 17829, 17844, 17859, 17874, 18310  
 $\text{\_fp\_small\_int:wTF}$  . . . . . 13501, 13957  
 $\text{\_fp\_small\_int\_normal:NnwTF}$  . 13501  
 $\text{\_fp\_small\_int\_test:NnnwNTF}$  . 13501  
 $\text{\_fp\_small\_int\_test:NnnwNw}$  . . . . .  
 . . . . . 13520, 13523  
 $\text{\_fp\_small\_int\_true:wTF}$  . . . . . 13501  
 $\text{\_fp\_sqrt\_auxi\_o:NNNNwnnN}$  . . . . .  
 . . . . . 16449, 16457  
 $\text{\_fp\_sqrt\_auxii\_o:NnnnnnnnN}$  . . . . .  
 . . . . . 746, 748, 16459, 16463, 16543, 16555  
 $\text{\_fp\_sqrt\_auxiii\_o:wnnnnnnnn}$  . . . . .  
 . . . . . 16460, 16498, 16544  
 $\text{\_fp\_sqrt\_auxiv\_o:NNNNNw}$  . . . . . 16498  
 $\text{\_fp\_sqrt\_auxix\_o:wnwnw}$  . . . . . 16532  
 $\text{\_fp\_sqrt\_auxv\_o:NNNNNw}$  . . . . . 16498  
 $\text{\_fp\_sqrt\_auxvi\_o:NNNNNw}$  . . . . . 16498  
 $\text{\_fp\_sqrt\_auxvii\_o:NNNNNw}$  . . . . . 16498  
 $\text{\_fp\_sqrt\_auxviii\_o:nnnnnnn}$  . . . . .  
 . . . . . 16520, 16522, 16524, 16530, 16532  
 $\text{\_fp\_sqrt\_auxx\_o:Nnnnnnnn}$  . . . . .  
 . . . . . 16528, 16546  
 $\text{\_fp\_sqrt\_auxxi\_o:wwnnN}$  . . . . . 16546  
 $\text{\_fp\_sqrt\_auxxii\_o:nnnnnnnnw}$  . . . . .  
 . . . . . 16556, 16560  
 $\text{\_fp\_sqrt\_auxxiii\_o:w}$  . . . . . 16560  
 $\text{\_fp\_sqrt\_auxxiv\_o:wnnnnnnnN}$  . . . . .  
 . . . . . 16572, 16575, 16583, 16585  
 $\text{\_fp\_sqrt\_Newton\_o:wwn}$  . . . . .  
 . . . . . 746, 16434, 16445, 16446  
 $\text{\_fp\_sqrt\_npos\_auxi\_o:wwnnN}$  . 16425

`\_fp_sqrt_npos_auxii_o:wNNNNNNNN`  
 ..... 16425  
`\_fp_sqrt_npos_o:w` 16422, 16425  
`\_fp_sqrt_o:w` ..... 15864, 16415  
`\_fp_step:NNnnnn` ..... 15695  
`\_fp_step:NnnnnN` ..... 15625  
`\_fp_step:wwwN` ..... 15625  
`\_fp_step_fp:wwwN` ..... 15625  
`\_fp_str_if_eq:nn` .....  
 .. 13551, 14652, 14666, 14954, 17607  
`\_fp_sub_back_far_o:NnnwnnnnN` ..  
 ..... 730, 16035, 16081  
`\_fp_sub_back_near_after:wNNNNw`  
 ..... 16041, 16119  
`\_fp_sub_back_near_o:nnnnnnnnnN` ..  
 ..... 728, 16031, 16041  
`\_fp_sub_back_near_pack:NNNNNNw`  
 ..... 16041, 16121  
`\_fp_sub_back_not_far_o:wwwNNN` ..  
 ..... 16096, 16116  
`\_fp_sub_back Quite_far_ii:NN` 16100  
`\_fp_sub_back Quite_far_o:wwNN` ..  
 ..... 16094, 16100  
`\_fp_sub_back_shift:wnnnn` ..  
 ..... 729, 16053, 16057  
`\_fp_sub_back_shift_ii:ww` ..... 16057  
`\_fp_sub_back_shift_iii:NNNNNNNNw`  
 ..... 16057  
`\_fp_sub_back_shift_iv:nnnnw` ..... 16057  
`\_fp_sub_back_very_far_ii_-`  
`o:nnNwwNN` ..... 16128  
`\_fp_sub_back_very_far_o:wwwNNN`  
 ..... 16095, 16128  
`\_fp_sub_eq_o:Nnwnw` ..... 16006  
`\_fp_sub_npos_i_o:Nnwnw` ..  
 ..... 727, 16011, 16020, 16024  
`\_fp_sub_npos_ii_o:Nnwnw` ..... 16006  
`\_fp_sub_npos_o:NnwNnw` ..  
 ..... 727, 15926, 16006  
`\_fp_tan_o:w` ..... 17883  
`\_fp_tan_series_aux_o:Nnwww` .. 18364  
`\_fp_tan_series_o:NNwww` ..  
 ..... 793, 793, 17890, 17905, 18364  
`\_fp_ternary:NwwN` .. 712, 15137, 15809  
`\_fp_ternary_auxi:NwwN` ..  
 ..... 712, 721, 15809  
`\_fp_tmp:w` ..... 647,  
 702, 13443, 13453, 13454, 13455,  
 13456, 13457, 13458, 13459, 13460,  
 13461, 13462, 13463, 13464, 13465,  
 13466, 13467, 13468, 13571, 13573,  
 14056, 14068, 14069, 14070, 14071,  
 14072, 14073, 14074, 14133, 14155,  
 14717, 14734, 14735, 14791, 14796,  
 14797, 14798, 14799, 14800, 14801,  
 14805, 14813, 14814, 14815, 14816,  
 14817, 14818, 14819, 14820, 14821,  
 14822, 14823, 15000, 15016, 15017,  
 15040, 15052, 15070, 15072, 15074,  
 15076, 15078, 15080, 15082, 15086,  
 15100, 15101, 15116, 15117, 15118,  
 15136, 15138, 16661, 16675, 16676  
`\_fp_to_decimal:w` ..  
 .. 18768, 18778, 18895, 18912, 19401  
`\_fp_to_decimal_dispatch:w` 818,  
 820, 821, 15688, 18758, 18762, 18765  
`\_fp_to_decimal_huge:wnnnn` .. 18778  
`\_fp_to_decimal_large:Nnnw` .. 18778  
`\_fp_to_decimal_normal:wnnnnn` ..  
 ..... 18778, 18866  
`\_fp_to_decimal_recover:w` .. 18765  
`\_fp_to_dim:w` ..... 18880  
`\_fp_to_dim_dispatch:w` .. 821, 18880  
`\_fp_to_dim_recover:w` ..... 18880  
`\_fp_to_int:w` .. 821, 18905, 18910  
`\_fp_to_int_dispatch:w` ..... 18896  
`\_fp_to_int_recover:w` ..... 18896  
`\_fp_to_scientific:w` ..  
 ..... 818, 18714, 18724  
`\_fp_to_scientific_dispatch:w` ..  
 .. 816, 820, 18704, 18708, 18711  
`\_fp_to_scientific_normal:wnnnnn`  
 ..... 18724  
`\_fp_to_scientific_normal:wNw` 18724  
`\_fp_to_scientific_recover:w` .. 18711  
`\_fp_to_tl:w` .. 18844, 18852, 19408  
`\_fp_to_tl_dispatch:w` ..  
 .. 815, 820, 18836, 18840, 18843, 18981  
`\_fp_to_tl_normal:nnnnn` ..... 18852  
`\_fp_to_tl_recover:w` ..... 18843  
`\_fp_to_tl_scientific:wnnnnn` .. 18852  
`\_fp_to_tl_scientific:wNw` .. 18852  
`\c_fp_trailing_shift_int` ..  
 ..... 13388, 16693,  
 16715, 16788, 17687, 18256, 18293  
`\_fp_trap_division_by_zero_-`  
`set:N` ..... 13641  
`\_fp_trap_division_by_zero_set_-`  
`error:` ..... 13641  
`\_fp_trap_division_by_zero_set_-`  
`flag:` ..... 13641  
`\_fp_trap_division_by_zero_set_-`  
`none:` ..... 13641  
`\_fp_trap_invalid_operation_-`  
`set:N` ..... 13607

`\_fp_trap_invalid_operation_-  
 set_error: .....` [13607](#)  
`\_fp_trap_invalid_operation_-  
 set_flag: .....` [13607](#)  
`\_fp_trap_invalid_operation_-  
 set_none: .....` [13607](#)  
`\_fp_trap_overflow_set:N .....` [13667](#)  
`\_fp_trap_overflow_set:NnNn .....` [13667](#)  
`\_fp_trap_overflow_set_error: .....` [13667](#)  
`\_fp_trap_overflow_set_flag: .....` [13667](#)  
`\_fp_trap_overflow_set_none: .....` [13667](#)  
`\_fp_trap_underflow_set:N .....` [13667](#)  
`\_fp_trap_underflow_set_error: .....` [13667](#)  
`\_fp_trap_underflow_set_flag: .....` [13667](#)  
`\_fp_trap_underflow_set_none: .....` [13667](#)  
`\_fp_trig:NNNNNwn .....` [17829](#), [17844](#),  
[17859](#), [17874](#), [17889](#), [17904](#), [17921](#)  
`\c_fp_trig_intarray .....` [801](#),  
[17982](#), [18212](#), [18215](#), [18218](#), [18221](#),  
[18224](#), [18227](#), [18230](#), [18233](#), [18236](#)  
`\_fp_trig_large:ww .....` [17929](#), [18196](#)  
`\_fp_trig_large_auxi:w .....` [18196](#)  
`\_fp_trig_large_auxii:w .....` [801](#), [18196](#)  
`\_fp_trig_large_auxiii:w .....` [801](#), [18196](#)  
`\_fp_trig_large_auxix:Nw .....` [18269](#)  
`\_fp_trig_large_auxv:www .....` [18246](#), [18249](#)  
`\_fp_trig_large_auxvi:wnnnnnnnn .....` [18249](#)  
`\_fp_trig_large_auxvii:w .....` [18252](#), [18269](#)  
`\_fp_trig_large_auxviii:w .....` [18269](#)  
`\_fp_trig_large_auxviii:ww .....` [18271](#), [18275](#)  
`\_fp_trig_large_auxxx:wNNNNN .....` [18269](#)  
`\_fp_trig_large_auxxi:w .....` [18269](#)  
`\_fp_trig_large_pack:NNNNNw .....` [18249](#), [18298](#)  
`\_fp_trig_small:ww .....` [795](#), [803](#), [17931](#), [17935](#), [17941](#), [18308](#)  
`\_fp_trigd_large:ww .....` [17929](#), [17943](#)  
`\_fp_trigd_large_auxi:nnnnwNNNN .....` [17943](#)  
`\_fp_trigd_large_auxii:wNw .....` [17943](#)  
`\_fp_trigd_large_auxiii:www .....` [17943](#)  
`\_fp_trigd_small:ww .....` [795](#), [17931](#), [17937](#), [17980](#)  
`\_fp_trim_zeros:w .....` [18695](#), [18819](#), [18828](#), [18879](#)  
`\_fp_trim_zeros_dot:w .....` [18695](#)  
`\_fp_trim_zeros_end:w .....` [18695](#)  
`\_fp_trim_zeros_loop:w .....` [18695](#)  
`\_fp_tuple_ .....` [15799](#), [15800](#), [15803](#), [15804](#)  
`\_fp_tuple_&_o:ww .....` [15782](#)  
`\_fp_tuple_&_tuple_o:ww .....` [15782](#)  
`\_fp_tuple_*_o:ww .....` [16655](#)  
`\_fp_tuple_+_tuple_o:ww .....` [16661](#)  
`\_fp_tuple_/_tuple_o:ww .....` [16661](#)  
`\_fp_tuple_/_o:ww .....` [16655](#)  
`\_fp_tuple_chk:w .....` [641](#),  
[13291](#), [13297](#), [13298](#), [13375](#), [13378](#),  
[15049](#), [15259](#), [15274](#), [15299](#), [15302](#),  
[15318](#), [15319](#), [15322](#), [15523](#), [15524](#),  
[16664](#), [16665](#), [16671](#), [16672](#), [18674](#)  
`\_fp_tuple_compare_back:ww .....` [15520](#)  
`\_fp_tuple_compare_back_loop:w .....` [15520](#)  
`\_fp_tuple_compare_back_-tuple:ww .....` [15520](#)  
`\_fp_tuple_convert:Nw .....` [18674](#), [18723](#), [18777](#), [18851](#)  
`\_fp_tuple_convert_end:w .....` [18674](#)  
`\_fp_tuple_convert_loop:nNw .....` [18674](#)  
`\_fp_tuple_count:w .....` [13296](#)  
`\_fp_tuple_count_loop:Nw .....` [13296](#)  
`\_fp_tuple_map_loop_o:nw .....` [15299](#)  
`\_fp_tuple_map_o:nw .....` [15299](#), [16648](#), [16656](#), [16658](#), [16660](#)  
`\_fp_tuple_mapthread_loop_o:nw .....` [15317](#)  
`\_fp_tuple_mapthread_o:nww .....` [15317](#), [16669](#)  
`\_fp_tuple_not_o:w .....` [15773](#)  
`\_fp_tuple_set_sign_aux_o:Nnw .....` [16640](#)  
`\_fp_tuple_set_sign_aux_o:w .....` [16640](#)  
`\_fp_tuple_set_sign_o:w .....` [16640](#)  
`\_fp_tuple_to_decimal:w .....` [18765](#)  
`\_fp_tuple_to_scientific:w .....` [18711](#)  
`\_fp_tuple_to_tl:w .....` [18843](#)  
`\_fp_tuple_|_o:ww .....` [15782](#)  
`\_fp_tuple_|_tuple_o:ww .....` [15782](#)  
`\_fp_type_from_scan:N .....` [642](#), [13320](#), [14898](#), [14900](#), [14924](#),  
[14926](#), [14937](#), [14939](#), [15484](#), [15486](#)  
`\_fp_type_from_scan:w .....` [13320](#)  
`\_fp_type_from_scan_other:N .....` [13320](#), [13344](#), [13362](#)  
`\_fp_underflow:w .....` [639](#), [653](#), [655](#), [13240](#), [13698](#), [17390](#)  
`\_fp_use_i:ww .....` [762](#), [813](#), [13186](#), [16905](#), [18603](#)  
`\_fp_use_i:www .....` [13186](#)  
`\_fp_use_i_until_s:nw .....` [803](#), [13181](#),  
[13227](#), [13493](#), [13543](#), [17973](#), [18251](#),  
[18257](#), [18288](#), [19069](#), [19140](#), [19348](#)  
`\_fp_use_ii_until_s:nnw .....` [13181](#), [13225](#)

\\_\_fp\_use\_none\_stop\_f:n .....  
     ..... [13178](#), [17070](#), [17071](#), [17072](#)  
 \\_\_fp\_use\_none\_until\_s:w .....  
     .. [13181](#), [16451](#), [17747](#), [18598](#), [18601](#)  
 \\_\_fp\_use\_s:n ..... [13179](#)  
 \\_\_fp\_use\_s:nn ..... [13179](#)  
 \\_\_fp\_zero\_fp:N .. [13218](#), [13682](#), [14022](#)  
 \\_\_fp\_l\_o:ww ..... [712](#), [15782](#)  
 \\_\_fp\_l\_tuple\_o:ww ..... [15782](#)  
 \\_\_fp\_ ..... [15785](#), [15792](#), [15801](#), [15802](#)  
 fparray commands:  
     \fparray\_count:N ..... [244](#),  
         [244](#), [245](#), [19310](#), [19321](#), [19332](#), [19387](#)  
     \fparray\_gset:Nnn ..... [244](#), [835](#), [19334](#)  
     \fparray\_gzero:N ..... [245](#), [19384](#)  
     \fparray\_item:Nn ..... [245](#), [835](#), [19396](#)  
     \fparray\_item\_to\_tl:Nn ..... [245](#), [19396](#)  
     \fparray\_new:Nn ..... [244](#), [19284](#)  
 \futurelet ..... [380](#)

**G**

\gdef ..... [381](#)  
 \GetIdInfo ..... [6](#)  
 \gleaders ..... [930](#), [1752](#)  
 \global ..... [173](#),  
     [174](#), [188](#), [189](#), [190](#), [201](#), [202](#), [203](#),  
     [204](#), [205](#), [206](#), [207](#), [208](#), [209](#), [278](#), [382](#)  
 \globaldefs ..... [383](#)  
 \glueexpr ..... [637](#), [1451](#)  
 \glueshrink ..... [638](#), [1452](#)  
 \glueshrinkorder ..... [639](#), [1453](#)  
 \gluestretch ..... [640](#), [1454](#)  
 \gluestretchorder ..... [641](#), [1455](#)  
 \gluetomu ..... [642](#), [1456](#)  
 group commands:  
     \group\_align\_safe\_begin/end: [480](#), [856](#)  
     \group\_align\_safe\_begin: ... [104](#),  
         [384](#), [388](#), [473](#), [4357](#), [4685](#), [7462](#),  
         [7665](#), [8884](#), [8899](#), [19916](#), [26867](#), [27956](#)  
     \group\_align\_safe\_end: .... [104](#),  
         [384](#), [388](#), [4380](#), [4711](#), [7464](#), [7665](#),  
         [8893](#), [8904](#), [8910](#), [19919](#), [26888](#), [27964](#)  
 \group\_begin: . [8](#), [1007](#), [2077](#), [3124](#),  
     [3127](#), [3130](#), [3513](#), [3947](#), [4129](#), [4241](#),  
     [4452](#), [5060](#), [5087](#), [5363](#), [5386](#), [6043](#),  
     [8443](#), [8449](#), [8500](#), [8580](#), [8606](#), [8624](#),  
     [8648](#), [8736](#), [8754](#), [9460](#), [9484](#), [9500](#),  
     [9573](#), [9877](#), [10249](#), [10340](#), [10361](#),  
     [10709](#), [10755](#), [11030](#), [11071](#), [11119](#),  
     [12000](#), [15782](#), [19572](#), [19609](#), [19915](#),  
     [19922](#), [19995](#), [20155](#), [20504](#), [20598](#),  
     [20606](#), [20921](#), [21420](#), [21783](#), [21876](#),  
     [22297](#), [22679](#), [22890](#), [23050](#), [23059](#),  
     [23071](#), [23080](#), [23088](#), [23206](#), [23236](#),  
     [23750](#), [25319](#), [25544](#), [25663](#), [26359](#),  
     [26574](#), [26715](#), [26724](#), [26735](#), [27417](#),  
     [27454](#), [27472](#), [27494](#), [27648](#), [27670](#),  
     [27900](#), [27906](#), [28717](#), [28739](#), [29413](#)  
 \c\_group\_begin\_token [46](#), [122](#), [260](#),  
     [399](#), [508](#), [4816](#), [4851](#), [8606](#), [8630](#),  
     [19960](#), [23798](#), [23805](#), [23820](#), [23827](#),  
     [23910](#), [23917](#), [23933](#), [23940](#), [27979](#)  
 \group\_end: ..... [8](#), [8](#), [433](#), [1007](#),  
     [2077](#), [3124](#), [3127](#), [3133](#), [3522](#), [3950](#),  
     [4135](#), [4262](#), [4307](#), [4456](#), [5078](#), [5110](#),  
     [5368](#), [5391](#), [6053](#), [6058](#), [8451](#), [8458](#),  
     [8583](#), [8603](#), [8623](#), [8627](#), [8655](#), [8753](#),  
     [8801](#), [9479](#), [9492](#), [9511](#), [9734](#), [9922](#),  
     [10265](#), [10347](#), [10372](#), [10713](#), [10784](#),  
     [11050](#), [11080](#), [11133](#), [12051](#), [15806](#),  
     [19576](#), [19617](#), [19826](#), [19920](#), [19941](#),  
     [20002](#), [20179](#), [20517](#), [20620](#), [20954](#),  
     [20962](#), [21433](#), [21841](#), [21883](#), [21890](#),  
     [21898](#), [22301](#), [22302](#), [22716](#), [22954](#),  
     [23055](#), [23066](#), [23150](#), [23212](#), [23273](#),  
     [23756](#), [25320](#), [25660](#), [25677](#), [26366](#),  
     [26590](#), [26718](#), [26728](#), [26739](#), [27445](#),  
     [27486](#), [27493](#), [27663](#), [27669](#), [27696](#),  
     [27904](#), [27930](#), [28736](#), [28768](#), [29421](#)  
 \c\_group\_end\_token . [122](#), [260](#), [508](#),  
     [8606](#), [8635](#), [19963](#), [23813](#), [23926](#), [27980](#)  
 \group\_insert\_after:N .....  
     . [8](#), [2083](#), [20513](#), [28076](#), [28162](#), [28455](#)  
 groups commands:  
     .groups:n ..... [172](#), [12514](#)

**H**

\H ..... [27699](#)  
 \halign ..... [384](#)  
 \hangafter ..... [385](#)  
 \hangindent ..... [386](#)  
 \hbadness ..... [387](#)  
 \hbox ..... [388](#)  
 hbox commands:  
     \hbox:n .....  
         [224](#), [23764](#), [24009](#), [24305](#), [25110](#), [25165](#)  
     \hbox\_gset:Nn .. [225](#), [23766](#), [23976](#),  
         [24099](#), [24143](#), [24163](#), [24183](#), [24200](#),  
         [24221](#), [24250](#), [24261](#), [24435](#), [25692](#)  
     \hbox\_gset:Nw ..... [225](#), [23794](#), [24501](#)  
     \hbox\_gset\_end: ... [225](#), [23794](#), [24504](#)  
     \hbox\_gset\_to\_wd:Nnn .... [225](#), [23780](#)  
     \hbox\_gset\_to\_wd:Nnw .... [225](#), [23816](#)  
     \hbox\_overlap\_left:n .... [225](#), [23842](#)  
     \hbox\_overlap\_right:n .... [225](#), [23842](#), [28210](#), [28242](#), [28483](#), [28598](#),  
         [28639](#), [28809](#), [28837](#), [29275](#), [29375](#)

\hbox\_set:Nn . . . . .  
   ... 225, 225, 23766, 23973, 24005,  
   24006, 24093, 24140, 24160, 24167,  
   24180, 24197, 24218, 24247, 24255,  
   24278, 24422, 24875, 24962, 25255,  
   25689, 25702, 25710, 25718, 25727,  
   25736, 25753, 25761, 25769, 25775,  
   25788, 25830, 25844, 28701, 29026  
   \hbox\_set:Nw . . . . . 225, 23794, 24488  
   \hbox\_set\_end: 225, 225, 23794, 24491  
   \hbox\_set\_to\_wd:Nnn . 225, 225, 23780  
   \hbox\_set\_to\_wd:Nnw . . . . . 225, 23816  
   \hbox\_to\_wd:nn . . . . . 224, 23832, 24296  
   \hbox\_to\_zero:n . . . . .  
     ... 224, 23832, 23843, 23845  
   \hbox\_unpack:N . . . . . 225, 23846, 24879  
   \hbox\_unpack\_clear:N . . . . . 24321  
   \hbox\_unpack\_drop:N . . . . .  
     ... 228, 23846, 24321, 24323

hcoffin commands:

  \hcoffin\_gset:Nn . . . . . 233, 24418  
   \hcoffin\_gset:Nw . . . . . 233, 24484  
   \hcoffin\_gset\_end: . . . . . 233, 24484  
   \hcoffin\_set:Nn . . . . .  
     ... 233, 24418, 25107, 25119, 25162, 25202  
   \hcoffin\_set:Nw . . . . . 233, 24484  
   \hcoffin\_set\_end: . . . . . 233, 24484

  \hfil . . . . . 389  
   \hfill . . . . . 390  
   \hfilneg . . . . . 391  
   \hfuzz . . . . . 392  
   \hjcode . . . . . 925, 1747  
   \hoffset . . . . . 393  
   \holdinginserts . . . . . 394  
   \hpack . . . . . 926, 1748  
   \hrule . . . . . 395  
   \hsize . . . . . 396  
   \hskip . . . . . 397  
   \hss . . . . . 398  
   \ht . . . . . 399

hundred commands:

  \c\_one\_hundred . . . . . 7257  
   \hyphenation . . . . . 400  
   \hyphenationbounds . . . . . 927, 1749  
   \hyphenationmin . . . . . 928, 1750  
   \hyphenchar . . . . . 401  
   \hyphenpenalty . . . . . 402  
   \hyphenpenaltymode . . . . . 929, 1751

I

  \I . . . . . 202  
   \i . . . . . 205, 27694  
   \if . . . . . 403

if commands:

  \if:w . . . . .  
     ... 21, 117, 334, 335, 370, 885, 2049,  
     2322, 2334, 2336, 2718, 3018, 3019,  
     3826, 3829, 3830, 3831, 3832, 3847,  
     3848, 3849, 3850, 3851, 3852, 3853,  
     3854, 3855, 3920, 3921, 3923, 7130,  
     8848, 10150, 10152, 10154, 12079,  
     12083, 12106, 14325, 14329, 14351,  
     14444, 14476, 14495, 14561, 14575,  
     14592, 14612, 15090, 15105, 17638,  
     19081, 20822, 25593, 25601, 25618  
   \if\_bool:N . . . . . 103, 103, 7379, 7420  
   \if\_box\_empty:N . . . . . 231, 23700, 23712  
   \if\_case:w . . . . . 91, 415, 417, 459, 648,  
     735, 789, 829, 2926, 3567, 5461,  
     5535, 6382, 7021, 7054, 8596, 10869,  
     13235, 13488, 13503, 13899, 13928,  
     15178, 15219, 15873, 16008, 16083,  
     16108, 16160, 16604, 16620, 16637,  
     16914, 17139, 17166, 17324, 17359,  
     17517, 17562, 17614, 17740, 17763,  
     17825, 17840, 17855, 17870, 17885,  
     17900, 18426, 18479, 18563, 18578,  
     18630, 18643, 18727, 18781, 18855,  
     19078, 19362, 19438, 19971, 20165,  
     20463, 20736, 21537, 21566, 21623,  
     22056, 22113, 22491, 22831, 27973  
   \if\_catcode:w . . . . .  
     ... 21, 389, 399, 400, 518, 2049,  
     4500, 4807, 4849, 4861, 4878, 8630,  
     8635, 8640, 8645, 8652, 8659, 8664,  
     8669, 8674, 8679, 8684, 8694, 8721,  
     8936, 8941, 14078, 14285, 14602,  
     14649, 14952, 19960, 19963, 20117,  
     20119, 20121, 20123, 20125, 20127,  
     20129, 27912, 27913, 27974, 27975  
   \if\_charcode:w . . . . .  
     ... 21, 117, 398, 399, 420, 518, 861,  
     2049, 4790, 4842, 5619, 8699, 8938,  
     13491, 15450, 15812, 20030, 20054,  
     20103, 20679, 20689, 21171, 22685  
   \if\_cs\_exist:N . . . . .  
     ... 21, 2063, 2745, 2773, 3516, 8729, 8857  
   \if\_cs\_exist:w 21, 2063, 2091, 2754,  
     2782, 2913, 7311, 7354, 7364, 26153  
   \if\_dim:w . . . . .  
     ... 168, 11351, 11469, 11487, 11510, 26441  
   \if\_eof:w . . . . .  
     ... 151, 561, 10484, 10491, 10536, 10554  
   \if\_false: 20, 97, 380, 385, 388, 397,  
     436, 450, 480, 507, 578, 853, 944,  
     2049, 3527, 3537, 3550, 3563, 3664,  
     3678, 3684, 3691, 3699, 3709, 3724,

4240, 4249, 4375, 4376, 4472, 4476,  
4758, 4763, 4774, 4862, 4874, 4889,  
4897, 5996, 5999, 6131, 6136, 6626,  
7666, 8543, 8592, 10848, 10888,  
10892, 10899, 10907, 11118, 11127,  
11497, 19907, 19949, 19998, 20001,  
20930, 20949, 20950, 20959, 21010,  
21046, 21060, 21064, 21287, 21320,  
21332, 21336, 21370, 21375, 21383,  
21418, 21425, 21430, 21478, 21700,  
21717, 21721, 22906, 22923, 23162,  
23167, 23278, 23283, 26573, 26582,  
26660, 27803, 27815, 27843, 27853  
\if\_hbox:N . . . . . 231, 23700, 23704  
\if\_int\_compare:w 20, 91, 449, 450,  
451, 2081, 3714, 4954, 4963, 5012,  
5013, 5019, 5195, 5204, 5209, 5445,  
5500, 5501, 5507, 5519, 5535, 6382,  
6430, 6481, 6482, 6521, 6606, 6659,  
6661, 6663, 6665, 6667, 6669, 6671,  
6680, 6832, 7666, 7668, 8472, 8473,  
8480, 8481, 8482, 8483, 8488, 8489,  
8531, 8839, 10855, 11526, 12976,  
12980, 13023, 13085, 13236, 13237,  
13431, 13528, 13779, 13789, 13797,  
13810, 13823, 13830, 13851, 13863,  
13872, 13883, 13984, 13989, 14060,  
14090, 14245, 14247, 14284, 14289,  
14342, 14362, 14389, 14403, 14438,  
14465, 14493, 14509, 14525, 14543,  
14602, 14622, 14638, 14651, 14665,  
14726, 14749, 14778, 14780, 14953,  
14963, 14965, 15004, 15021, 15026,  
15056, 15122, 15167, 15461, 15508,  
15511, 15542, 15551, 15554, 15559,  
15560, 15563, 15566, 15753, 15877,  
15898, 15935, 16030, 16084, 16085,  
16088, 16091, 16161, 16170, 16375,  
16448, 16501, 16505, 16509, 16527,  
16562, 16563, 16564, 16565, 16566,  
16592, 16916, 16919, 17013, 17106,  
17152, 17168, 17295, 17329, 17387,  
17396, 17436, 17607, 17609, 17620,  
17638, 17661, 17693, 17696, 17743,  
17773, 17928, 17972, 18430, 18468,  
18477, 18513, 18597, 18600, 18829,  
19068, 19136, 19137, 19138, 19148,  
19176, 19181, 19182, 19245, 19246,  
19247, 19251, 19266, 19271, 19317,  
19321, 19485, 19554, 19583, 19624,  
19635, 19638, 19656, 19711, 19721,  
19731, 19935, 20007, 20040, 20048,  
20071, 20095, 20146, 20161, 20243,  
20246, 20401, 20407, 20408, 20415,  
20418, 20421, 20427, 20428, 20432,  
20435, 20436, 20444, 20445, 20446,  
20452, 20482, 20483, 20733, 20753,  
20754, 20755, 20758, 20762, 20763,  
20766, 20767, 20775, 20776, 20779,  
20783, 20784, 20787, 20846, 20868,  
20880, 20889, 20897, 20900, 20910,  
20913, 20941, 21014, 21119, 21185,  
21190, 21218, 21285, 21318, 21429,  
21446, 21731, 21764, 21991, 22074,  
22103, 22168, 22181, 22192, 22208,  
22259, 22291, 22469, 22470, 22523,  
22550, 22625, 22696, 22759, 22769,  
22772, 22792, 22849, 22902, 22919,  
22942, 23091, 23120, 23160, 23165,  
23186, 23264, 23276, 23281, 25594,  
26778, 26779, 26785, 27886, 27894  
\if\_int\_odd:w . . . . . 92,  
805, 6382, 6518, 6720, 6730, 7245,  
8479, 8487, 8506, 13801, 13848,  
13860, 15215, 16144, 16430, 17784,  
18278, 18317, 18327, 18370, 18394,  
18554, 19244, 20171, 20472, 20857,  
20865, 20877, 21291, 21606, 27911  
\if\_meaning:w . . 21, 386, 387, 399,  
720, 2049, 2414, 2417, 2564, 2590,  
2608, 2667, 2672, 2681, 2742, 2760,  
2770, 2788, 2944, 2958, 3079, 3142,  
3199, 3200, 3568, 3571, 3572, 3573,  
3574, 3626, 3656, 3669, 3675, 3774,  
3797, 3806, 3991, 4003, 4004, 4408,  
4418, 4429, 4442, 4457, 4703, 4767,  
4833, 5304, 5372, 5395, 5556, 5594,  
5723, 5729, 5755, 5767, 5775, 5807,  
6032, 6048, 6063, 6071, 6401, 6451,  
6456, 6457, 6641, 7474, 7496, 7901,  
7916, 7938, 7952, 8689, 8726, 8764,  
8767, 8831, 8890, 8929, 9282, 10805,  
11440, 11503, 12016, 12040, 12057,  
12068, 13224, 13238, 13250, 13260,  
13355, 13410, 13419, 13510, 13525,  
13527, 13690, 13778, 13788, 13800,  
13813, 13814, 13833, 13834, 13848,  
13849, 13860, 13861, 13927, 13966,  
14001, 14004, 14020, 14027, 14079,  
14082, 14200, 14201, 14202, 14203,  
14206, 14302, 14415, 14421, 14650,  
14698, 14909, 14982, 15216, 15234,  
15284, 15294, 15497, 15498, 15499,  
15500, 15501, 15502, 15734, 15746,  
15747, 15775, 15787, 15794, 15811,  
15874, 15909, 15923, 15969, 15976,  
16052, 16064, 16164, 16167, 16178,  
16231, 16304, 16374, 16377, 16384,

16417, 16418, 16421, 16642, 16887,	\ifvmode .....	417	
16898, 17079, 17089, 17136, 17235,	\ifvoid .....	418	
17315, 17364, 17378, 17525, 17559,	\ifx .....	14, 21, 39, 43, 49,	
17571, 17584, 17587, 17590, 17593,	90, 92, 93, 94, 105, 130, 152, 153, 419		
17619, 17720, 17724, 17783, 18320,	\ifybox .....	1222, 2004	
18373, 18424, 18425, 18427, 18428,	\ifydir .....	1223, 2005	
18448, 18465, 18532, 18630, 18726,	\ignoreligaturesinfont .....	1018, 1627	
18780, 18854, 18925, 18930, 19067,	\ignorespaces .....	420	
19099, 19110, 19217, 19375, 19425,	\IJ .....	27685	
19431, 19496, 19497, 19957, 19987,	\ij .....	27685	
20015, 20115, 20512, 20821, 20845,	image commands:		
21167, 21170, 21613, 22038, 22216,	\image_bb_restore:nTF .....	28672, 29021	
22227, 22242, 22412, 22445, 22804,	\image_bb_save:n .....	28707, 29029	
23042, 23119, 23172, 23211, 25573,	\l_image_decode_tl .....	28649, 28660,	
25575, 25584, 26169, 26172, 26444,	28690, 28851, 28876, 28918, 28990		
27808, 27847, 27866, 27914, 27976	\l_image_decodearray_tl .....	28650,	
\if_mode_horizontal: .....	21, 2059, 7660	28686, 28691, 28877, 28915, 28919	
\if_mode_inner: .....	21, 2059, 7662	\image_extract_bb:n .....	
\if_mode_math: .....	21, 2059, 7664	28846, 28853, 29378, 29379	
\if_mode_vertical: .....	21, 2059, 7658, 25683	\l_image_interpolate_bool .....	
\if_predicate:w .....	95, 97, 103, 7379,	28651, 28661, 28685, 28692,	
7452, 7512, 7527, 7538, 7553, 7564	28852, 28878, 28914, 28920, 28991		
\if_true: .....	20, 97, 387, 2049	\l_image_llx_dim .....	28907
\if_vbox:N .....	231, 23700, 23706	\l_image_lly_dim .....	28908
\ifabsdim .....	1016, 1625	\l_image_page_int .....	
\ifabsnum .....	1017, 1626	28645, 28665, 28666, 28696, 28697,	
\ifcase .....	404	28844, 28874, 28875, 28901, 28902,	
\ifcat .....	405	28983, 28996, 28997, 29035, 29036	
\ifcondition .....	931	\l_image_pagebox_tl .....	
\ifcsname .....	643, 1457	1101, 28646, 28664, 28698,	
\ifdbox .....	1217, 1999	28699, 28845, 28872, 28873, 28903,	
\ifddir .....	1218, 2000	28905, 28984, 29005, 29006, 29037	
\ifdefined .....	165, 644, 1458	\image_read_bb:n .....	28245, 28840
\ifdim .....	406	\l_image_urx_dim .....	28703, 28909
\ifeof .....	407	\l_image_ury_dim .....	
\iffalse .....	408	28704, 28910, 29028, 29385, 29386	
\iffontchar .....	645, 1459	\l_image_utx_dim .....	29027
\ifhbox .....	409	\immediate .....	421
\ifhmode .....	410	\immediateassigned .....	932
\ifincsname .....	794, 1608	\immediateassignment .....	933
\ifinner .....	411	in .....	201
\ifmdir .....	1219, 2001	\indent .....	422
\ifmmode .....	412	inf .....	201
\ifnum .....	45, 60, 89, 102, 107, 171, 186, 413	\infinity .....	14203, 14204
\ifodd .....	414	inherit commands:	
\ifpdfabsdim .....	750, 1567	.inherit:n .....	172, 12516
\ifpdfabsnum .....	751, 1568	\inhibitglue .....	1224, 2006
\ifpdfprimitive .....	752, 1569	\inhibitxspcode .....	1225, 2007
\ifprimitive .....	885, 1618	\initcatcodetable .....	934, 1753
\iftbox .....	1220, 2002	initial commands:	
\iftdir .....	1221, 2003	.initial:n .....	173, 12518
\iftrue .....	415	\input .....	50, 166, 167, 423
\ifvbox .....	416	\inputlineno .....	424
		\insert .....	425

\insertht . . . . . 1019, 1629  
 \insertpenalties . . . . . 426  
 int commands:  
     \c\_eight . . . . . 7257  
     \c\_eleven . . . . . 7257  
     \c\_fifteen . . . . . 7257  
     \c\_five . . . . . 7257  
     \l\_foo\_int . . . . . 216  
     \c\_four . . . . . 7257  
     \c\_fourteen . . . . . 7257  
     \int\_abs:n . . . . . 79, 1029, 6392, 13023  
     \int\_add:Nn . . . . . 81, 6561, 10964, 20453,  
                       21608, 22198, 22199, 22454, 22547  
     \int\_case:nn . . . . . 84, 459, 6686, 6884, 6890  
     \int\_case:nnn . . . . . 29712  
     \int\_case:nnTF . . . . . 84,  
                       6296, 6686, 6691, 6696, 8214, 14110,  
                       18676, 23562, 27312, 27367, 29713  
     \int\_compare:nNnTF . . . . . 82,  
                       82, 84, 85, 85, 4244, 4271, 4286,  
                       4293, 4909, 4916, 4983, 5424, 5426,  
                       5435, 6189, 6196, 6499, 6505, 6672,  
                       6710, 6766, 6774, 6783, 6789, 6816,  
                       6819, 6880, 6968, 6974, 6980, 7000,  
                       7154, 7173, 7175, 7217, 7739, 8253,  
                       8255, 8260, 8269, 8289, 8306, 10578,  
                       10664, 10777, 11706, 12963, 12968,  
                       12973, 13077, 13118, 13144, 15526,  
                       16667, 18659, 18804, 18806, 19298,  
                       19472, 20290, 20490, 20502, 20665,  
                       20983, 20985, 21777, 22415, 22643,  
                       22658, 22861, 23136, 23581, 26276,  
                       26353, 26763, 27030, 27032, 27035,  
                       27178, 27200, 27234, 27237, 27271,  
                       27274, 27281, 27294, 27409, 28665,  
                       28696, 28874, 28901, 28996, 29035  
     \int\_compare:nTF . . . . .  
                       . . . . . 83, 85, 85, 85, 85, 189,  
                       588, 6619, 6738, 6746, 6755, 6761,  
                       10462, 10489, 10634, 18864, 21881,  
                       23319, 23541, 23542, 23547, 23549  
     \int\_compare\_p:n . . . . . 83, 6619, 21888  
     \int\_compare\_p:nNn . . . . .  
                       . . . . . 20, 82, 6672, 7726,  
                       10566, 21676, 21677, 26620, 26622,  
                       26624, 27259, 27354, 27355, 27356,  
                       27401, 27703, 27704, 27728, 27729  
     \int\_const:Nn . . . . . 80, 466, 6493,  
                       7183, 7184, 7185, 7186, 7187, 7188,  
                       7189, 7190, 7191, 7192, 7193, 7194,  
                       7195, 7196, 7241, 7242, 7243, 7283,  
                       7695, 7697, 7699, 7700, 7701, 10382,  
                       10561, 10562, 13205, 13206, 13207,  
                       13208, 13209, 13210, 13211, 13388,  
                       13389, 13390, 13392, 13393, 13394,  
                       13397, 13398, 13399, 13773, 14039,  
                       14040, 14041, 14042, 14043, 14044,  
                       14045, 14046, 14047, 14048, 14049,  
                       14050, 14051, 14052, 19018, 19538,  
                       20387, 20388, 20389, 20390, 20794,  
                       20795, 20796, 20797, 20798, 20799,  
                       20803, 20804, 20805, 20806, 20807,  
                       20808, 20809, 20810, 20811, 20812,  
                       20813, 20814, 20815, 26540, 26626,  
                       28495, 28705, 28723, 28896, 28931  
     \int\_decr:N . . . . . 81,  
                       6577, 19644, 19645, 19646, 19709,  
                       19710, 19719, 19720, 19729, 19730,  
                       19950, 22493, 22851, 22920, 23121  
     \int\_div\_round:nn . . . . . 80, 6436  
     \int\_div\_truncate:nn . . . . .  
                       . . . . . 80, 80, 6436, 6895, 6993,  
                       7013, 7698, 26791, 26804, 26809, 26821  
     \int\_do\_until:nn . . . . . 85, 6736  
     \int\_do\_until:nNn . . . . . 84, 6764  
     \int\_do\_while:nn . . . . . 85, 6736  
     \int\_do\_while:nNn . . . . . 84, 6764  
     \int\_eval:n . . . . . 13, 25, 79,  
                       79, 79, 80, 81, 82, 83, 84, 91, 91,  
                       248, 310, 314, 342, 445, 462, 630,  
                       631, 635, 667, 713, 737, 739, 870,  
                       1030, 1043, 2926, 2955, 2971, 4296,  
                       4604, 4609, 4902, 4910, 4918, 4945,  
                       4949, 4958, 4965, 5000, 5010, 5418,  
                       5431, 5456, 5480, 5481, 5493, 5498,  
                       5529, 5546, 5583, 6182, 6190, 6198,  
                       6270, 6387, 6689, 6694, 6699, 6704,  
                       6877, 6963, 6965, 7095, 7105, 7140,  
                       7151, 7157, 7168, 7199, 7236, 7240,  
                       7613, 8187, 8196, 8247, 8257, 8271,  
                       8278, 8293, 8344, 8346, 8414, 8416,  
                       8420, 8422, 8426, 8428, 8432, 8434,  
                       8467, 8468, 10448, 10619, 10900,  
                       10958, 12962, 12999, 13000, 13049,  
                       13065, 13114, 13138, 13147, 13151,  
                       13215, 19007, 19167, 19170, 19171,  
                       19261, 19262, 19294, 19340, 19400,  
                       19407, 19586, 19852, 19853, 20073,  
                       20149, 20153, 20176, 20472, 20737,  
                       21606, 21811, 21815, 21818, 21822,  
                       22011, 22013, 22022, 22039, 22040,  
                       22042, 22043, 22192, 22282, 22313,  
                       22518, 22566, 22641, 22771, 22776,  
                       23323, 23368, 23369, 23568, 23732,  
                       23742, 26242, 26279, 26383, 26531,  
                       26694, 26774, 26826, 26829, 26834,  
                       26840, 27888, 27896, 28776, 28975  
     \int\_eval:w . . . . . 79, 311,

314, 314, 5449, 6387, 7314, 7365,  
 10851, 10860, 10885, 10897, 13090,  
 16616, 19900, 20135, 20145, 26428  
`\int_from_alpha:n` ..... 88, 7138  
`\int_from_base:nn` .....  
 ..... 89, 7155, 7178, 7180, 7182  
`\int_from_bin:n` ..... 88, 7177, 29715  
`\int_from_binary:nn` ..... 29714  
`\int_from_hex:n` ..... 89, 7177, 29717  
`\int_from_hexadecimal:nn` ..... 29716  
`\int_from_oct:n` ..... 89, 7177, 29719  
`\int_from_octal:nn` ..... 29718  
`\int_from_roman:n` ..... 89, 7197  
`\int_gadd:Nn` ..... 81, 6561  
`\int_gdecr:N` ..... 81, 4565,  
 5323, 6247, 6577, 6875, 8148, 9327,  
 10531, 11684, 15718, 20215, 26410  
`\int_gincr:N` ..... 81, 4558, 5312, 6239,  
 6577, 6850, 6861, 8141, 9322, 10522,  
 11663, 11670, 12952, 15697, 15704,  
 19289, 20193, 26404, 28494, 28895,  
 28930, 29295, 29429, 29485, 29530  
`.int_gset:N` ..... 173, 12526  
`\int_gset:Nn` ..... 81, 446, 6593, 9518  
`\int_gset_eq:NN` ..... 81, 6543, 29422  
`\int_gsub:Nn` ..... 81, 6561, 19302  
`\int_gzero:N` ..... 80, 6531, 6540, 29414  
`\int_gzero_new:N` ..... 80, 6537  
`\int_if_even:nTF` ..... 84, 6716, 11086  
`\int_if_even_p:n` ..... 84, 6716  
`\int_if_exist:NTF` ..... 81, 6538, 6540,  
 6549, 7211, 7215, 21532, 21587, 28885  
`\int_if_exist_p:N` ..... 81, 6549  
`\int_if_odd:nTF` ..... 84, 6716, 17002  
`\int_if_odd_p:n` ..... 84, 6716, 21925  
`\int_incr:N` .....  
 ... 81, 6577, 12320, 13039, 13070,  
 13160, 19389, 19558, 19654, 19655,  
 19991, 20033, 20046, 20064, 20344,  
 20345, 21423, 21846, 22031, 22124,  
 22455, 22490, 22492, 22573, 22838,  
 22903, 23062, 23118, 23182, 26373  
`\int_log:N` ..... 90, 7237  
`\int_log:n` ..... 90, 7239  
`\int_max:nn` ..... 80, 823, 6392, 16863,  
 17952, 22148, 22373, 23268, 23269  
`\int_min:nn` ..... 80, 826, 6392  
`\int_mod:nn` ..... 80,  
 6436, 6885, 6984, 7004, 7696, 26823  
`\int_new:N` ..... 80, 80, 6485,  
 6501, 6507, 6538, 6540, 7253, 7254,  
 7255, 7256, 7670, 10686, 10689,  
 10691, 10704, 12139, 12944, 12946,  
 19282, 19283, 19453, 19454, 19455,  
 19456, 19457, 19458, 19459, 19460,  
 19461, 19462, 19463, 19886, 19887,  
 19888, 19889, 20370, 20371, 20372,  
 20385, 20792, 20793, 20800, 20801,  
 20818, 21942, 21944, 21945, 21946,  
 21949, 22319, 22320, 22321, 22322,  
 22323, 22324, 22325, 22326, 22327,  
 22328, 22331, 22332, 22333, 22603,  
 23031, 23034, 23035, 23036, 23587,  
 26346, 26347, 28121, 28490, 28856,  
 28926, 29345, 29432, 29433, 29511  
`\int_rand:n` .....  
 ... 89, 247, 13058, 19009, 19012, 19259  
`\int_rand:nn` ..... 89, 247, 252, 826, 833,  
 1032, 4925, 6204, 7241, 8307, 8312,  
 10049, 19003, 19006, 19165, 26269  
`\int_range:nn` ..... 827  
`.int_set:N` ..... 173, 12526  
`\int_set:Nn` ..... 81, 310, 3131, 4245,  
 4279, 6593, 10507, 10509, 10670,  
 10672, 10687, 10697, 10710, 10757,  
 10763, 10775, 10780, 12325, 19466,  
 19468, 19470, 19493, 19494, 19509,  
 19517, 19518, 19530, 19531, 19542,  
 19543, 19544, 19560, 19563, 19945,  
 20008, 20332, 21614, 21943, 22006,  
 22008, 22089, 22144, 22145, 22158,  
 22169, 22193, 22211, 22260, 22352,  
 22371, 22405, 22426, 22522, 22557,  
 23067, 23215, 23241, 23741, 23743,  
 23751, 23752, 23753, 23754, 26374  
`\int_set_eq:NN` ..... 81, 4242, 4246, 6543,  
 11120, 19510, 19551, 20443, 20911,  
 20915, 20924, 20926, 20969, 21022,  
 21322, 21422, 21435, 21534, 21965,  
 21982, 21983, 22029, 22030, 22080,  
 22190, 22191, 22243, 22298, 22354,  
 22357, 22378, 22385, 22402, 22404,  
 22407, 22421, 22424, 22456, 22457,  
 22461, 22597, 23173, 26575, 29410  
`\int_show:N` ..... 89, 7233  
`\int_show:n` ..... 90, 465, 1030, 7235  
`\int_sign:n` ..... 246, 1037, 26159  
`\int_step...` ..... 218  
`\int_step_function:nN` ..... 86, 6792, 26364  
`\int_step_function:nnN` .....  
 ... 86, 6792, 8579,  
 8584, 8587, 10575, 19613, 22462, 23132  
`\int_step_function:nnnn` ..... 86, 250,  
 251, 455, 716, 6792, 6874, 23244, 23252  
`\int_step_inline:nn` .....  
 ... 86, 631, 6844, 12955  
`\int_step_inline:nnn` .....  
 ... 86, 6844, 10391, 10584, 22396, 23592

\int\_step\_inline:nnn . . . . . 86, 718, 6844  
 \int\_step\_variable:nNn . . . . . 86, 6844  
 \int\_step\_variable:nnNn . . . . . 86, 6844  
 \int\_step\_variable:nnnNn . . . . . 86, 6844  
 \int\_sub:Nn . . . . . 81, 6561, 9461, 10972,  
     20447, 21126, 22246, 22254, 22263  
 \int\_to\_Alph:n . . . . . 87, 88, 6898  
 \int\_to\_alph:n . . . . . 87, 87, 88, 6898  
 \int\_to\_arabic:n . . . . . 87, 6877  
 \int\_to\_Base:n . . . . . 88  
 \int\_to\_base:n . . . . . 88  
 \int\_to\_Base:nn . . . . . 88, 89, 6962, 7089  
 \int\_to\_base:nn . . . . .  
     . . . . . 88, 89, 6962, 7085, 7087, 7091  
 \int\_to\_bin:n . . . . . 88, 88, 88, 7084, 29721  
 \int\_to\_binary:n . . . . . 29720  
 \int\_to\_Hex:n . . . . . 88, 89, 7084, 20668  
 \int\_to\_hex:n . . . . . 88, 89, 7084, 29723  
 \int\_to\_hexadecimal:n . . . . . 29722  
 \int\_to\_oct:n . . . . . 88, 89, 7084, 29725  
 \int\_to\_octal:n . . . . . 29724  
 \int\_to\_Roman:n . . . . . 88, 89, 7092  
 \int\_to\_roman:n . . . . . 88, 89, 7092  
 \int\_to\_symbols:nnn . . . . .  
     . . . . . 87, 87, 6878, 6900, 6932  
 \int\_until\_do:nn . . . . . 85, 6736  
 \int\_until\_do:nNnn . . . . . 85, 6764  
 \int\_use:N . . . . . 79, 82, 660,  
     665, 4560, 4562, 5314, 5318, 6240,  
     6246, 6601, 6853, 6864, 8143, 8145,  
     9321, 9329, 9434, 10006, 10510,  
     10524, 10666, 11666, 11673, 12326,  
     15700, 15707, 20195, 20943, 21016,  
     21087, 21098, 21107, 21111, 21122,  
     21123, 21129, 21130, 21136, 21137,  
     21305, 21925, 21999, 22001, 22019,  
     22020, 22021, 22122, 22135, 22136,  
     22508, 22558, 22588, 22698, 22710,  
     22806, 23067, 23582, 26385, 26406,  
     26408, 26472, 26481, 26482, 26485,  
     26490, 26499, 26500, 26504, 26507,  
     26512, 28501, 28666, 28697, 28712,  
     28738, 28745, 28875, 28888, 28900,  
     28902, 28937, 29036, 29297, 29328,  
     29488, 29494, 29501, 29533, 29541  
 \int\_value:w . . . . .  
     . . . . . 91, 314, 314, 365, 450, 474,  
     588, 630, 631, 639, 644, 648, 661,  
     667, 674, 677, 683, 690, 714, 715,  
     723, 731, 739, 801, 805, 818, 853,  
     1029, 1032, 2367, 2723, 3676, 3678,  
     4958, 4965, 5418, 5419, 5431, 5449,  
     5456, 5479, 5480, 5481, 5493, 5529,  
     6382, 6390, 6391, 6396, 6397, 6409,

17327, 17332, 17336, 17395, 17444,  
 17512, 17521, 17529, 17540, 17542,  
 17545, 17548, 17637, 17673, 17675,  
 17678, 17681, 17684, 17687, 17694,  
 17697, 17699, 17703, 17725, 17727,  
 17759, 17926, 17958, 17967, 18199,  
 18200, 18211, 18214, 18217, 18220,  
 18223, 18226, 18229, 18232, 18235,  
 18253, 18263, 18272, 18290, 18299,  
 18306, 18316, 18360, 18369, 18404,  
 18447, 18464, 18520, 18531, 18542,  
 18752, 18828, 18875, 18921, 18929,  
 18931, 18933, 19030, 19053, 19107,  
 19147, 19159, 19170, 19171, 19201,  
 19204, 19207, 19209, 19211, 19218,  
 19221, 19229, 19234, 19239, 19340,  
 19400, 19407, 19419, 19420, 19421,  
 19431, 19586, 19900, 19912, 20091,  
 20133, 20135, 20145, 20153, 20172,  
 20174, 20182, 20661, 21155, 21161,  
 21193, 21195, 21204, 21205, 21329,  
 21753, 21768, 22619, 22620, 22631,  
 23156, 23187, 23189, 25613, 26163,  
 26164, 26269, 26279, 26416, 26428,  
 26435, 26522, 27888, 27896, 28776  
`\int_while_do:nn` ..... 85, 6736  
`\int_while_do:nNnn` ..... 85, 6764  
`\int_zero:N` ..... 80, 80, 6531,  
 6538, 10817, 12317, 13036, 13063,  
 13157, 19386, 19946, 19947, 19948,  
 20047, 20923, 21124, 21571, 21878,  
 21964, 22351, 22370, 22403, 22681,  
 23061, 26361, 28645, 28844, 28983  
`\int_zero_new:N` ..... 80, 6537  
`\c_max_int` ..... 90, 182, 633,  
 826, 827, 874, 926, 7242, 19212,  
 20418, 20432, 22457, 23729, 23735  
`\c_nine` ..... 7257  
`\c_one` ..... 7257  
`\c_one_int` ..... 90,  
 6579, 6582, 6585, 6588, 7241, 13090  
`\c_seven` ..... 7257  
`\c_six` ..... 7257  
`\c_sixteen` ..... 7257  
`\c_ten` ..... 7257  
`\c_thirteen` ..... 7257  
`\c_three` ..... 7257  
`\g_tmpa_int` ..... 90, 7253  
`\l_tmpa_int` ..... 2, 90, 211, 7253  
`\g_tmpb_int` ..... 90, 7253  
`\l_tmpb_int` ..... 2, 90, 7253  
`\c_twelve` ..... 7257  
`\c_two` ..... 7257  
`\c_zero` ..... 466, 7257  
`\c_zero_int` ..... 90, 317, 335,  
 335, 336, 1043, 2100, 2721, 2723,  
 4242, 6481, 6482, 6499, 6532, 6534,  
 6606, 6614, 6816, 6819, 7241, 7666,  
 7668, 11120, 11526, 12956, 13077,  
 19053, 19182, 19246, 26525, 26575  
**int internal commands:**  
`\_int_abs:N` ..... 6392  
`\_int_case:mnTF` ..... 6686  
`\_int_case:nw` ..... 6686  
`\_int_case_end:nw` ..... 6686  
`\_int_compare:nnN` ..... 450, 6619  
`\_int_compare>NNw` ..... 450, 450, 6619  
`\_int_compare:Nw` ..... 449, 451, 6619  
`\_int_compare:w` ..... 450, 6619  
`\_int_compare_!=:NNw` ..... 6619  
`\_int_compare_<:NNw` ..... 6619  
`\_int_compare_<=:NNw` ..... 6619  
`\_int_compare_=:NNw` ..... 6619  
`\_int_compare_==:NNw` ..... 6619  
`\_int_compare_>:NNw` ..... 6619  
`\_int_compare_>=:NNw` ..... 6619  
`\_int_compare_end_=:NNw` ..... 450, 6619  
`\_int_compare_error:` .....  
 ..... 449, 450, 6604, 6622, 6624  
`\_int_compare_error:Nw` .....  
 ..... 449, 450, 451, 6604, 6644  
`\_int_constdef:Nw` .. 466, 6493, 7293  
`\_int_DEPRECATED_CONSTANTS:nn` 7257  
`\_int_div_truncate:NwNw` ..... 6436  
`\_int_eval:w` ..... 310,  
 443, 444, 450, 6382, 6388, 6390,  
 6391, 6393, 6397, 6404, 6405, 6410,  
 6411, 6417, 6418, 6423, 6424, 6438,  
 6439, 6443, 6445, 6446, 6463, 6466,  
 6467, 6471, 6472, 6473, 6480, 6496,  
 6515, 6558, 6563, 6566, 6569, 6572,  
 6595, 6598, 6622, 6656, 6674, 6676,  
 6680, 6717, 6720, 6727, 6730, 6795,  
 6799, 6803, 6810, 6811, 6812, 6838,  
 7021, 7048, 7054, 7081, 26160, 26164  
`\_int_eval_end:` .....  
 ..... 6382, 6390, 6397, 6447, 6463,  
 6474, 6483, 6515, 6563, 6566, 6569,  
 6572, 6595, 6598, 6675, 6680, 6720,  
 6730, 7021, 7048, 7054, 7081, 26164  
`\_int_from_alpha:N` ..... 462, 7138  
`\_int_from_alpha:nN` ..... 462, 7138  
`\_int_from_base:N` ..... 462, 7155  
`\_int_from_base:nnN` ..... 462, 7155  
`\_int_from_roman>NN` ..... 7197  
`\c__int_from_roman_C_int` ..... 7183  
`\c__int_from_roman_c_int` ..... 7183  
`\c__int_from_roman_D_int` ..... 7183

```

\c__int_from_roman_d_int ..... 7183
\__int_from_roman_error:w ..... 7197
\c__int_from_roman_I_int ..... 7183
\c__int_from_roman_i_int ..... 7183
\c__int_from_roman_L_int ..... 7183
\c__int_from_roman_l_int ..... 7183
\c__int_from_roman_M_int ..... 7183
\c__int_from_roman_m_int ..... 7183
\c__int_from_roman_V_int ..... 7183
\c__int_from_roman_v_int ..... 7183
\c__int_from_roman_X_int ..... 7183
\c__int_from_roman_x_int ..... 7183
\c__int_max_constdef_int ..... 6493
\__int_maxmin:wwN ..... 6392
\__int_mod:ww ..... 6436
\__int_pass_signs:wn ..... .
..... 461, 7128, 7142, 7159
\__int_pass_signs_end:wn ..... 7128
\__int_show:nN ..... 7233
\__int_sign:Nw ..... 26159
\__int_step>NNnnnn ..... 6844
\__int_step:NwnnN ..... 6792
\__int_step:wwwN ..... 6792
\__int_tmp:w ..... 447, 448, 6553,
..... 6561, 6564, 6567, 6570, 6593, 6596
\__int_to_Base:nn ..... 6962
\__int_to_base:nn ..... 6962
\__int_to_Base:nnN ..... 6962
\__int_to_base:nnN ..... 6962
\__int_to_Base:nnnN ..... 6962
\__int_to_base:nnnn ..... 6962
\__int_to_Letter:n ..... 6962
\__int_to_letter:n ..... 6962
\__int_to_roman:N ..... 7092
\__int_to_roman:w ..... .
..... 450, 461, 2081, 6382, 6632, 7095, 7105
\__int_to_Roman_aux:N 7104, 7107, 7110
\__int_to_Roman_c:w ..... 7092
\__int_to_roman_c:w ..... 7092
\__int_to_Roman_d:w ..... 7092
\__int_to_roman_d:w ..... 7092
\__int_to_Roman_i:w ..... 7092
\__int_to_roman_i:w ..... 7092
\__int_to_Roman_l:w ..... 7092
\__int_to_roman_l:w ..... 7092
\__int_to_Roman_m:w ..... 7092
\__int_to_roman_m:w ..... 7092
\__int_to_Roman_Q:w ..... 7092
\__int_to_roman_Q:w ..... 7092
\__int_to_Roman_v:w ..... 7092
\__int_to_roman_v:w ..... 7092
\__int_to_Roman_x:w ..... 7092
\__int_to_roman_x:w ..... 7092
\__int_to_symbols:nnnn ..... 6878
\__int_value:w ..... 7297
intarray commands:
\intarray_const_from_clist:Nn ...
..... 247, 632, 13059, 17450, 17982
\intarray_count:N ..... .
..... 182, 182, 182, 12963, 12966, 12968,
..... 12969, 12971, 12980, 12991, 13037,
..... 13058, 13077, 13102, 13158, 19313
\intarray_gset:Nnn 182, 630, 632, 12993
\intarray_gset_rand:Nn ... 247, 13107
\intarray_gset_rand:Nnn .. 247, 13107
\intarray_gzero:N ..... 182, 13034
\intarray_item:Nn ..... .
..... 182, 630, 632, 13043, 13058
\intarray_log:N ..... 247, 13092
\intarray_new:Nn ..... .
..... 182, 629, 632, 12949, 19305,
..... 19306, 19307, 20382, 20383, 20384,
..... 22334, 22335, 23037, 23038, 23039
\intarray_rand_item:N ... 182, 13057
\intarray_show:N ... 247, 632, 13092
\intarray_to_clist:N ... 247, 13074
intarray internal commands:
\__intarray_bounds:NNnTF ..... .
..... 12974, 13004, 13053
\__intarray_bounds_error:NNn . 12974
\__intarray_const_from_clist:nN .
..... 13059
\__intarray_count:w ..... .
..... 12942, 12962, 12971, 13066, 13085
\__intarray_entry:w ..... .
..... 12942, 12994, 13040, 13044
\g_intarray_font_int ..... .
..... 12946, 12952, 12954
\__intarray_gset:Nnn ..... 12993
\__intarray_gset:Nww .. 12997, 13002
\__intarray_gset_all_same:Nn . 13107
\__intarray_gset_overflow:Nnn . 12993
\__intarray_gset_overflow:NNnn ..
..... 13016, 13024, 13028
\__intarray_gset_overflow_-
..... test:nw ..... 632, 633, 13006,
..... 13013, 13021, 13071, 13125, 13132
\__intarray_gset_rand:Nnn .... 13107
\__intarray_gset_rand_auxi:Nnnn .
..... 13107
\__intarray_gset_rand_auxii:Nnnm
..... 13107
\__intarray_gset_rand_auxiii:Nnnn
..... 13107
\__intarray_item:Nn ..... 13043
\__intarray_item:Nw ... 13047, 13051

```

```

\l__intarray_loop_int . . . . . 12944,
    13036, 13039, 13040, 13063, 13066,
    13070, 13072, 13157, 13160, 13161
\__intarray_new:N . . . . . 12949, 13062
\__intarray_show>NN . . . . . .
    . . . . . 13092, 13094, 13096
\__intarray_signed_max_dim:n . .
    . . . . . 12972, 13031, 13032
\c__intarray_sp_dim . . . . .
    . . . . . 12945, 12954, 12994
\__intarray_to_clist:Nn 13074, 13103
\__intarray_to_clist:w 13074
\interactionmode . . . . . 646, 1460
\interlinepenalties . . . . . 647, 1461
\interlinepenalty . . . . . 427
ior commands:
\ior_close:N . . . . .
    . . . . . 144, 145, 145, 10439, 10460,
    11164, 25589, 25622, 25659, 25676
\ior_get>NN . . . . . 146,
    146, 147, 151, 246, 10501, 10517, 10543
\ior_get_str>NN . . . . . 29726
\ior_if_eof:N . . . . . 561
\ior_if_eof:NTF . . . . .
    148, 10485, 10529, 10548, 11159, 11173
\ior_if_eof_p:N . . . . . 148, 10485
\ior_list_streams: . . . . . 29728
\ior_log_list: . . . . . 145, 10472, 29731
\ior_log_streams: . . . . . 29730
\ior_map_break: 147, 10512, 10530,
    10537, 10549, 10555, 25585, 25655
\ior_map_break:n . . . . . 148, 10512
\ior_map_inline:Nn . . . . . 147, 246, 10516
\ior_map_variable>NNn . . . . .
    . . . . . 246, 10542, 25582
\ior_new:N . . . . .
    . . . . . 144, 10405, 10407, 10408, 11013, 25541
\ior_open:Nn . . . . . 144,
    580, 10409, 25565, 25590, 25623, 25675
\ior_open:NnTF . . . . . 145, 10410, 10413
\ior_show_list: . . . . . 145, 10472, 29729
\ior_str_get>NN . . . . .
    . . . . . 146, 10503, 10519, 10545, 29727
\ior_str_map_inline:Nn . . . . .
    . . . . . 147, 246, 10516, 25616, 25646
\ior_str_map_variable>NNn 246, 10542
\c_term_ior . . . . .
    . . . . . 151, 10382, 10405, 10462, 10468, 10489
\g_tmipa_ior . . . . . 151, 10407
\g_tmpb_ior . . . . . 151, 10407
ior internal commands:
\l__ior_file_name_str . . . .
    . . . . . 10412, 10415, 10416, 10419
\l__ior_internal_tl . . . . .
    . . . . . 10381, 10535, 10539
\__ior_list:N . . . . . 10472
\__ior_map_inline>NNn . . . . . 10516
\__ior_map_inline>NNNn . . . . . 10516
\__ior_map_inline_loop>NNN . . . . . 10516
\__ior_map_variable>NNNn . . . . . 10542
\__ior_map_variable_loop>NNNn . . . . . 10542
\__ior_new:N . . . . . 558, 10424, 10447
\__ior_new_aux:N . . . . . 10429, 10433
\__ior_open_stream:Nn . . . . . 10437
\l__ior_stream_tl . . . . .
    . . . . . 10388, 10440, 10448, 10456
\g__ior_streams_prop . . . . .
    . . . . . 10389, 10457, 10465, 10479
\g__ior_streams_seq . . . . .
    . . . . . 10383, 10440, 10466, 10467
iow commands:
\iow_allow_break: . . . . .
    . . . . . 150, 246, 10724, 10766, 10771
\iow_allow_break:n . . . . . 568
\iow_char:N . . . . . 149,
    2340, 10014, 10016, 10017, 10120,
    10141, 10142, 10685, 11074, 17556,
    19839, 19842, 19843, 19868, 19869,
    19876, 19877, 20647, 20649, 20651,
    20653, 20655, 20657, 21261, 21262,
    21802, 21915, 21916, 21917, 21938,
    23291, 23294, 23295, 23300, 23334,
    23343, 23347, 23352, 23372, 23374,
    23375, 23377, 23380, 23382, 23389,
    23393, 23396, 23397, 23400, 23402,
    23406, 23408, 23414, 23416, 23420,
    23422, 23426, 23431, 23433, 23475,
    23477, 23482, 23484, 23490, 23495,
    23500, 23504, 23514, 23517, 23521,
    23522, 23526, 23534, 23597, 29802
\iow_close:N . . . . . 145, 145, 10610, 10632
\iow_indent:n . . . . .
    . . . . . 150, 150, 568, 569, 9955, 10063,
    10735, 10767, 10772, 13722, 13734
\l_iow_line_count_int . . . . .
    . . . . . 150, 150, 569, 867, 9461, 10686,
    10776, 10781, 10819, 20292, 20296
\iow_log:n . . . . . 148, 311, 325, 2382,
    2831, 5052, 9675, 9690, 9691, 9697,
    10299, 10302, 10303, 10304, 10680
\iow_log_list: . . . . . 145, 10644
\iow_new:N . . . . . 144, 10598, 10600, 10601
\iow_newline: . . . . .
    . . . . . 149, 149, 149, 150, 312,
    405, 535, 566, 9483, 10336, 10352,
    10354, 10684, 10764, 10773, 10779,

```

11298, 20242, 21849, 25283, 25284,  
 25285, 26209, 26211, 26214, 26221  
`\iow_now:Nn` .....  
     ... 148, 148, 148, 149, 149, 10674,  
         10680, 10681, 10682, 10683, 26637  
`\iow_open:Nn` ..... 145, 10607  
`\iow_shipout:Nn` .....  
     ... 149, 149, 149, 566, 10659, 26650  
`\iow_shipout_x:Nn` .....  
     ... 149, 149, 149, 566, 10656  
`\iow_show_list:` ..... 145, 10644  
`\iow_term:n` . 148, 2831, 9495, 9653,  
     9668, 9669, 9723, 10306, 10309,  
     10310, 10311, 10350, 10680, 23583  
`\iow_wrap:nnnN` .....  
     ... 149, 149, 150, 150, 150,  
         246, 405, 569, 1030, 5037, 5052,  
         9459, 9462, 9474, 9654, 9676, 9695,  
         9702, 10303, 10310, 10328, 10329,  
         10727, 10733, 10738, 10750, 10753  
`\iow_wrap:nnnN\iow_wrap:nxnN` .. 150  
`\c_log_iow` .....  
     151, 562, 10561, 10634, 10680, 10681  
`\c_term_iow` .....  
     ... 151, 562, 10561, 10575, 10578,  
         10598, 10634, 10640, 10682, 10683  
`\g_tmipa_iow` ..... 151, 10600  
`\g_tmrb_iow` ..... 151, 10600  
 iow internal commands:  
     `\_iow_allow_break:` 568, 10724, 10766  
     `\_iow_allow_break_error:` .....  
         ... 568, 10724, 10771  
     `\l\_iow_file_name_str` .....  
         ... 10606, 10609, 10612, 10620  
     `\_iow_indent:n` ... 568, 10735, 10767  
     `\_iow_indent_error:n` .....  
         ... 568, 10735, 10772  
     `\l\_iow_indent_int` ..... 10703,  
         10817, 10835, 10947, 10964, 10972  
     `\l\_iow_indent_tl` .. 10703, 10818,  
         10834, 10946, 10965, 10973, 10974  
     `\l\_iow_line_break_bool` .....  
         ... 10707, 10813, 10941, 10955, 10963,  
             10971, 10979, 10981, 10986, 10988  
     `\l\_iow_line_part_tl` .....  
         ... 571, 573, 574, 10705, 10815,  
             10827, 10848, 10906, 10909, 10940,  
             10954, 10956, 10962, 10970, 10993  
     `\l\_iow_line_target_int` .....  
         ... 575, 10689, 10775,  
             10777, 10780, 10942, 10947, 10982  
     `\l\_iow_line_tl` 10705, 10814, 10831,  
         10921, 10937, 10953, 10954, 10962,  
         10970, 10992, 10993, 10998, 11000  
     `\_iow_list:N` ..... 10644  
     `\_iow_new:N` ..... 10602, 10618  
     `\l\_iow_newline_tl` ..... 10688,  
         10773, 10774, 10776, 10779, 10997  
     `\l\_iow_one_indent_int` .....  
         ... 10690, 10964, 10972  
     `\l\_iow_one_indent_tl` .....  
         ... 567, 10690, 10965  
     `\_iow_open_stream:Nn` ..... 10607  
     `\_iow_set_indent:n` ..... 567, 10690  
     `\l\_iow_stream_tl` .....  
         ... 10581, 10611, 10619, 10627  
     `\g\_iow_streams_prop` .....  
         ... 10582, 10628, 10637, 10651  
     `\g\_iow_streams_seq` .....  
         ... 10570, 10611, 10638, 10639  
     `\_iow_tmp:w` ..... 573, 10821,  
         10845, 10902, 10934, 11002, 11010  
     `\_iow_unindent:w` .. 567, 10690, 10974  
     `\_iow_with:nNnn` ..... 10662  
     `\_iow_wrap_allow_break:n` .... 10951  
     `\c\_iow_wrap_allow_break_marker_-`  
         `tl` ..... 10709, 10729  
     `\_iow_wrap_break:w` .... 10888, 10902  
     `\_iow_wrap_break_end:w` .. 573, 10902  
     `\_iow_wrap_break_first:w` .... 10902  
     `\_iow_wrap_break_loop:w` .... 10902  
     `\_iow_wrap_break_none:w` .... 10902  
     `\_iow_wrap_chunk:nw` 10819, 10821,  
         10957, 10958, 10966, 10975, 10982  
     `\_iow_wrap_do:` ..... 10783, 10788  
     `\_iow_wrap_end:n` ..... 10977  
     `\_iow_wrap_end_chunk:w` .....  
         ... 571, 10839, 10846, 10938  
     `\c\_iow_wrap_end_marker_tl` .....  
         ... 10709, 10793  
     `\_iow_wrap_fix_newline:w` .... 10788  
     `\_iow_wrap_indent:n` ..... 10960  
     `\c\_iow_wrap_indent_marker_tl` ..  
         ... 10709, 10743  
     `\_iow_wrap_line:nw` .....  
         ... 571, 573, 10833, 10837, 10846, 10945  
     `\_iow_wrap_line_aux:Nw` ..... 10846  
     `\_iow_wrap_line_end:NnnnnnnnnN` 10846  
     `\_iow_wrap_line_end:nw` .....  
         ... 573, 10846, 10922, 10923, 10932  
     `\_iow_wrap_line_loop:w` ..... 10846  
     `\_iow_wrap_line_seven:nnnnnnnn` 10846  
     `\c\_iow_wrap_marker_tl` .....  
         ... 567, 570, 10709, 10845  
     `\_iow_wrap_newline:n` ..... 10977  
     `\c\_iow_wrap_newline_marker_tl` ..  
         ... 570, 10709, 10808

\\_\_iow\_wrap\_next:nw .....  
     ..... [10821](#), [10900](#), [10942](#)  
 \\_\_iow\_wrap\_next\_line:w [10894](#), [10935](#)  
 \\_\_iow\_wrap\_start:w ..... [10788](#)  
 \\_\_iow\_wrap\_store\_do:n .....  
     ..... [10893](#), [10980](#), [10987](#), [10990](#)  
 \l\_\_iow\_wrap\_tl .....  
     ..... [569](#), [570](#), [575](#), [575](#), [10708](#),  
[10770](#), [10785](#), [10790](#), [10792](#), [10795](#),  
[10797](#), [10800](#), [10816](#), [10994](#), [10996](#)  
 \\_\_iow\_wrap\_trim:N .....  
     ..... [575](#), [10923](#), [10954](#), [10980](#), [10987](#), [11002](#)  
 \\_\_iow\_wrap\_trim:w ..... [11002](#)  
 \\_\_iow\_wrap\_trim\_aux:w ..... [11002](#)  
 \\_\_iow\_wrap\_unindent:n ..... [10960](#)  
 \c\_iow\_wrap\_unindent\_marker\_tl .  
     ..... [10709](#), [10745](#)

**J**

\J ..... [204](#)  
 \j ..... [27695](#)  
 \jcharwidowpenalty ..... [1226](#), [2008](#)  
 \jfam ..... [1227](#), [2009](#)  
 \jfont ..... [1228](#), [2010](#)  
 \jis ..... [1229](#), [2011](#)  
 job commands:  
     \c\_job\_name\_tl ..... [29692](#)  
 \jobname ..... [428](#)

**K**

\k ..... [27699](#)  
 \kanjискip ..... [1230](#), [2012](#)  
 \kansuji ..... [1231](#), [2013](#)  
 \kansujichar ..... [1232](#), [2014](#)  
 \kcatcode ..... [1233](#), [2015](#)  
 \kchar ..... [1254](#), [2036](#)  
 \kchardef ..... [1255](#), [2037](#)  
 \kern ..... [429](#)  
 kernel internal commands:  
     \kernel\_chk\_cs\_exist:N ... [310](#),  
[322](#), [2259](#), [2697](#), [2701](#), [2705](#), [2709](#), [3756](#)  
     \kernel\_chk\_defined:NTF .....  
         ..... [310](#), [500](#),  
[531](#), [3095](#), [3114](#), [5033](#), [6367](#), [7326](#),  
[7439](#), [8321](#), [9343](#), [13098](#), [15419](#), [22991](#)  
     \kernel\_chk\_expr:nNnN .....  
         ..... [310](#), [324](#), [2346](#),  
[6388](#), [6393](#), [6404](#), [6405](#), [6417](#), [6418](#),  
[6438](#), [6439](#), [6466](#), [6467](#), [6496](#), [6558](#),  
[6674](#), [6676](#), [6717](#), [6727](#), [6795](#), [6799](#),  
[6803](#), [11391](#), [11433](#), [11443](#), [11444](#),  
[11456](#), [11457](#), [11481](#), [11483](#), [11619](#),  
[11623](#), [11627](#), [11687](#), [11694](#), [11715](#),  
     \kernel\_chk\_if\_free\_cs:N .....  
         ..... [507](#), [532](#), [2835](#), [2855](#), [2903](#),  
[4057](#), [4064](#), [4070](#), [5712](#), [5842](#), [6488](#),  
[6511](#), [8607](#), [8609](#), [8619](#), [9046](#), [11357](#),  
[11747](#), [11881](#), [12951](#), [23618](#), [26229](#)  
     \kernel\_chk\_var\_exist:N .....  
         .. [310](#), [310](#), [322](#), [468](#), [2259](#), [4092](#),  
[4098](#), [4110](#), [4111](#), [4118](#), [4119](#), [7307](#)  
     \kernel\_chk\_var\_global:N .....  
         ..... [310](#), [322](#),  
[2259](#), [4097](#), [4148](#), [4151](#), [4154](#), [4175](#),  
[4178](#), [4181](#), [4184](#), [4207](#), [4210](#), [4213](#),  
[4216](#), [6533](#), [6546](#), [6567](#), [6570](#), [6583](#),  
[6586](#), [6596](#), [7390](#), [7393](#), [7410](#), [11371](#),  
[11397](#), [11406](#), [11413](#), [11424](#), [11761](#),  
[11786](#), [11798](#), [11806](#), [11896](#), [11928](#),  
[11936](#), [11942](#), [11950](#), [23638](#), [23646](#),  
[23718](#), [23772](#), [23786](#), [23801](#), [23823](#),  
[23870](#), [23884](#), [23898](#), [23913](#), [23936](#),  
[23953](#), [27761](#), [27789](#), [27795](#), [27825](#)  
     \kernel\_chk\_var\_local:N .....  
         ..... [310](#), [322](#),  
[2259](#), [4091](#), [4139](#), [4142](#), [4145](#), [4163](#),  
[4166](#), [4169](#), [4172](#), [4195](#), [4198](#), [4201](#),  
[4204](#), [6531](#), [6543](#), [6561](#), [6564](#), [6577](#),  
[6580](#), [6593](#), [7384](#), [7387](#), [7404](#), [11369](#),  
[11394](#), [11402](#), [11410](#), [11421](#), [11759](#),  
[11783](#), [11795](#), [11803](#), [11893](#), [11925](#),  
[11933](#), [11939](#), [11947](#), [23635](#), [23643](#),  
[23715](#), [23766](#), [23780](#), [23794](#), [23816](#),  
[23864](#), [23878](#), [23892](#), [23906](#), [23929](#),  
[23949](#), [27758](#), [27777](#), [27783](#), [27821](#)  
     \kernel\_chk\_var\_scope>NN .....  
         ..... [310](#), [310](#), [322](#), [2259](#),  
[4061](#), [4067](#), [5709](#), [5812](#), [6494](#), [11362](#),  
[11752](#), [11886](#), [12958](#), [13059](#), [26226](#)  
     \kernel\_cs\_parm\_from\_arg\_-  
         count:nnTF .. [310](#), [2552](#), [2921](#), [2968](#)  
     \kernel\_debug\_log:n .....  
         .. [311](#), [325](#), [2377](#), [2837](#), [3933](#), [9366](#), [12334](#)  
     \kernel\_deprecation\_code:nn ..  
         ..... [311](#), [2393](#), [7284](#), [29789](#)  
     \kernel\_deprecation\_error:Nnn ..  
         . [466](#), [2433](#), [2454](#), [7287](#), [29672](#), [29800](#)  
     \kernel\_exp\_not:w [311](#), [359](#), [3489](#),  
[3491](#), [3495](#), [3499](#), [3502](#), [3505](#), [3510](#),  
[4731](#), [4757](#), [4781](#), [6637](#), [26670](#), [26859](#)  
     \l\_kernel\_expl\_bool .....  
         ..... [241](#), [244](#), [259](#), [273](#), [2048](#)  
     \kernel\_file\_input\_pop: [311](#), [11188](#)  
     \kernel\_file\_input\_push:n ....  
         ..... [311](#), [11188](#)

```

\__kernel_file_missing:n .....
    ... 311, 10410, 11102, 11182, 11192
\__kernel_file_name_sanitize:nN .
    311, 311, 10609, 11069, 11138, 11184
\__kernel_if_debug:TF .....
    ... 311, 2179, 2185, 2236, 2259,
        2316, 2346, 2377, 2393, 2410, 2465,
        2484, 4087, 7333, 10136, 23573, 23588
\__kernel_int_add:nnn 311, 6478, 19212
\__kernel_intarray_gset:Nnn 630,
    12956, 12969, 12993, 13072, 13161,
    19373, 19374, 19376, 19380, 19381,
    19382, 22399, 22483, 22485, 22487,
    22513, 22516, 22571, 23094, 23096,
    23102, 23110, 23112, 23115, 23176,
    23178, 23180, 23193, 23199, 23203
\__kernel_intarray_item:Nn .....
    ... 631, 801, 13043, 13088,
        17536, 17542, 17545, 17548, 18212,
        18215, 18218, 18221, 18224, 18227,
        18230, 18233, 18236, 19419, 19420,
        19421, 22478, 22499, 22502, 22524,
        22551, 22612, 22613, 22636, 22637,
        22645, 22651, 22653, 22660, 22666,
        22668, 22725, 22729, 23099, 23226
\__kernel_iow_open:Nn .....
    ... 312, 10419, 10437, 11157, 11172
\__kernel_iow_with:Nnn . 312, 405,
    535, 565, 5041, 5043, 9496, 9498,
    9725, 9727, 10357, 10359, 10662, 10676
\__kernel_msg_error:nn .....
    ... 312, 2812, 9915, 12097,
        12116, 20903, 20936, 20984, 20987,
        21448, 21704, 22859, 22943, 24757
\__kernel_msg_error:nmn .....
    ... 312, 2192, 2200, 2227, 2232,
        2268, 2277, 2420, 2565, 2620, 2668,
        2673, 2812, 3005, 3012, 3100, 3798,
        3992, 4331, 5127, 5817, 9738, 9915,
        11089, 11185, 12198, 12257, 12273,
        12419, 12437, 12965, 13168, 13189,
        13602, 19300, 19833, 20942, 21144,
        21547, 21560, 21599, 21722, 22697,
        22704, 22957, 23759, 24383, 25378,
        26123, 26355, 26392, 26536, 26568
\__kernel_msg_error:nmmn .....
    ... 312, 2556,
        2596, 2687, 2812, 2844, 2970, 3884,
        4012, 4035, 9373, 9764, 9915, 10726,
        12182, 12237, 12292, 12306, 12428,
        12809, 13598, 19695, 19702, 21121,
        21186, 21410, 22863, 22879, 24599
\__kernel_msg_error:nnnnn .....
    ... 312, 2339, 9915, 10737, 13004,
        19345, 19850, 23143, 23266, 29679
\__kernel_msg_error:nnnnnn .....
    ... 312, 3899, 3913, 9915, 13030
\__kernel_msg_expandable_-
    error:nn ..... 313,
    3519, 5833, 7644, 8474, 8476, 8484,
    8490, 8532, 9041, 10266, 14121, 20642
\__kernel_msg_expandable_-
    error:nnn ..... 313, 3214, 3591, 3614, 3638,
        4597, 6310, 6615, 6821, 7339, 8229,
        9120, 10266, 11645, 14128, 14144,
        14149, 14216, 14273, 14312, 14318,
        14655, 14660, 14671, 14678, 14769,
        14783, 14983, 15009, 15667, 18998,
        19005, 19011, 20728, 25369, 26523
\__kernel_msg_expandable_-
    error:nnnn ..... 313, 2370, 10266, 10732, 13120, 15141,
        15162, 15828, 19177, 19267, 20667
\__kernel_msg_expandable_-
    error:nnnnn . 313, 10266, 10749,
        13053, 13713, 19044, 19412, 29676
\__kernel_msg_expandable_-
    error:nnnnnn ..... 313, 10266
\__kernel_msg_fatal:nn .....
    ... 312, 9915, 10443, 10614
\__kernel_msg_fatal:nnn ... 312, 9915
\__kernel_msg_fatal:nnnn .. 312, 9915
\__kernel_msg_fatal:nnnnn . 312, 9915
\__kernel_msg_fatal:nnnnnn 312, 9915
\__kernel_msg_info:nn .... 313, 9920
\__kernel_msg_info:nnn .... 313, 9920
\__kernel_msg_info:nnnn .. 313, 9920
\__kernel_msg_info:nnnnn .. 313, 9920
\__kernel_msg_info:nnnnnn . 313, 9920
\__kernel_msg_new:nnn .....
    ... 312, 9869, 9965, 9967, 9969,
        9971, 9973, 9991, 10048, 10124,
        10145, 10194, 10196, 10198, 10200,
        10202, 10204, 10206, 10208, 10212,
        10215, 10222, 10224, 10231, 10238,
        11316, 12932, 12947, 13741, 13743,
        13745, 13747, 13749, 13751, 13753,
        15339, 15341, 15343, 15345, 15347,
        15349, 15351, 15353, 15355, 15357,
        15359, 15361, 15363, 15365, 15370,
        15720, 15722, 15724, 18994, 20306,
        23290, 23292, 23297, 23551, 25311
\__kernel_msg_new:nnnn . 312, 9869,
    9923, 9931, 9939, 9946, 9957, 9975,
    9984, 9996, 10003, 10010, 10019,
    10028, 10035, 10041, 10050, 10057,
    10066, 10072, 10079, 10086, 10093,
```



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, 603,  
604, 605, 606, 607, 608, 609, 610,  
611, 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,  
695, 696, 697, 698, 699, 700, 701,  
702, 703, 704, 705, 707, 708, 709,  
710, 711, 712, 713, 715, 716, 717,  
718, 719, 720, 721, 722, 723, 724,  
725, 726, 727, 728, 729, 730, 731,  
732, 733, 734, 735, 736, 737, 738,  
739, 740, 741, 742, 743, 744, 745,  
746, 747, 748, 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,  
788, 789, 790, 791, 792, 793, 794,  
795, 796, 797, 798, 799, 800, 801,  
802, 807, 819, 821, 822, 823, 824,  
825, 826, 827, 828, 829, 830, 831,  
833, 835, 837, 838, 839, 841, 842,  
843, 844, 845, 846, 848, 850, 851,  
853, 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, 895, 896,  
898, 899, 900, 901, 902, 903, 904,  
905, 906, 907, 909, 910, 911, 912,  
913, 914, 915, 916, 917, 918, 919,  
920, 921, 922, 923, 924, 925, 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, 970, 971, 972, 973, 974, 975,  
976, 977, 978, 979, 980, 981, 982,  
983, 984, 985, 986, 987, 988, 989,  
990, 991, 992, 993, 994, 995, 996,  
997, 998, 999, 1000, 1001, 1002,  
1003, 1004, 1006, 1007, 1008, 1009,  
1010, 1011, 1012, 1013, 1014, 1015,  
1016, 1017, 1018, 1019, 1020, 1022,  
1024, 1026, 1027, 1028, 1029, 1030,  
1031, 1032, 1033, 1034, 1035, 1036,  
1037, 1038, 1039, 1040, 1041, 1042,  
1043, 1044, 1045, 1046, 1047, 1048,  
1049, 1050, 1051, 1052, 1053, 1054,  
1055, 1056, 1057, 1058, 1059, 1060,  
1061, 1062, 1063, 1064, 1065, 1066,  
1067, 1068, 1069, 1071, 1073, 1074,  
1075, 1076, 1078, 1079, 1080, 1081,  
1083, 1084, 1086, 1088, 1089, 1090,  
1091, 1092, 1094, 1096, 1097, 1098,  
1099, 1101, 1102, 1103, 1104, 1105,  
1106, 1107, 1108, 1109, 1110, 1111,  
1112, 1113, 1114, 1115, 1116, 1117,  
1118, 1119, 1119, 1120, 1121, 1122,  
1123, 1124, 1125, 1126, 1127, 1128,  
1129, 1130, 1131, 1132, 1133, 1134,  
1135, 1136, 1137, 1138, 1140, 1142,  
1143, 1145, 1147, 1148, 1149, 1150,  
1152, 1153, 1154, 1156, 1158, 1160,  
1161, 1162, 1163, 1164, 1165, 1166,  
1167, 1168, 1169, 1170, 1171, 1173,  
1175, 1176, 1177, 1178, 1179, 1180,  
1181, 1182, 1183, 1184, 1185, 1186,  
1187, 1188, 1189, 1190, 1191, 1193,  
1195, 1196, 1197, 1198, 1199, 1200,  
1201, 1202, 1203, 1204, 1205, 1206,  
1207, 1208, 1209, 1210, 1211, 1212,  
1213, 1214, 1215, 1216, 1217, 1218,  
1219, 1220, 1221, 1222, 1223, 1224,  
1225, 1226, 1227, 1228, 1229, 1230,  
1231, 1232, 1233, 1234, 1235, 1236,

1237, 1238, 1239, 1240, 1241, 1242,  
 1243, 1244, 1245, 1246, 1247, 1248,  
 1249, 1250, 1251, 1252, 1253, 1254,  
 1255, 1256, 1257, 1258, 1259, 1417,  
 1428, 1429, 1430, 1431, 1432, 1433,  
 1434, 1435, 1436, 1437, 1438, 1439,  
 1441, 1442, 1443, 1444, 1445, 1446,  
 1447, 1448, 1449, 1450, 1451, 1452,  
 1453, 1454, 1455, 1456, 1457, 1458,  
 1459, 1460, 1461, 1462, 1463, 1464,  
 1465, 1466, 1467, 1468, 1469, 1470,  
 1471, 1472, 1473, 1474, 1475, 1476,  
 1477, 1478, 1479, 1480, 1481, 1482,  
 1483, 1484, 1485, 1486, 1487, 1488,  
 1489, 1490, 1491, 1492, 1493, 1494,  
 1495, 1496, 1497, 1498, 1499, 1500,  
 1501, 1502, 1503, 1504, 1505, 1506,  
 1507, 1508, 1509, 1510, 1511, 1512,  
 1513, 1514, 1515, 1516, 1517, 1518,  
 1520, 1522, 1523, 1524, 1525, 1526,  
 1527, 1528, 1530, 1531, 1532, 1533,  
 1534, 1535, 1536, 1538, 1539, 1540,  
 1541, 1542, 1543, 1544, 1545, 1546,  
 1547, 1548, 1549, 1550, 1551, 1552,  
 1553, 1554, 1556, 1557, 1558, 1559,  
 1560, 1561, 1562, 1563, 1564, 1565,  
 1566, 1567, 1568, 1569, 1570, 1571,  
 1572, 1573, 1574, 1575, 1576, 1577,  
 1578, 1579, 1580, 1581, 1582, 1583,  
 1584, 1585, 1586, 1587, 1588, 1589,  
 1590, 1591, 1592, 1593, 1594, 1595,  
 1596, 1597, 1598, 1599, 1600, 1601,  
 1602, 1603, 1604, 1605, 1606, 1607,  
 1608, 1609, 1610, 1611, 1612, 1613,  
 1614, 1615, 1616, 1617, 1618, 1619,  
 1620, 1621, 1622, 1623, 1624, 1625,  
 1626, 1627, 1629, 1630, 1632, 1634,  
 1636, 1637, 1638, 1639, 1640, 1641,  
 1642, 1643, 1644, 1645, 1646, 1647,  
 1648, 1649, 1650, 1651, 1652, 1653,  
 1655, 1656, 1657, 1658, 1659, 1660,  
 1661, 1662, 1663, 1664, 1665, 1667,  
 1669, 1671, 1672, 1673, 1675, 1676,  
 1677, 1678, 1679, 1680, 1682, 1684,  
 1685, 1687, 1689, 1690, 1691, 1692,  
 1693, 1694, 1695, 1696, 1697, 1698,  
 1699, 1700, 1701, 1702, 1703, 1704,  
 1705, 1706, 1707, 1708, 1709, 1710,  
 1711, 1712, 1713, 1714, 1715, 1716,  
 1717, 1719, 1721, 1723, 1724, 1725,  
 1726, 1727, 1729, 1731, 1732, 1733,  
 1734, 1735, 1736, 1737, 1738, 1739,  
 1741, 1742, 1744, 1745, 1746, 1747,  
 1748, 1749, 1750, 1751, 1752, 1753,  
 1754, 1755, 1756, 1757, 1758, 1759,  
 1760, 1761, 1762, 1763, 1765, 1766,  
 1767, 1768, 1769, 1770, 1771, 1772,  
 1773, 1774, 1775, 1776, 1777, 1778,  
 1779, 1780, 1781, 1782, 1783, 1784,  
 1785, 1786, 1787, 1788, 1789, 1790,  
 1791, 1793, 1794, 1795, 1796, 1797,  
 1798, 1799, 1800, 1802, 1803, 1805,  
 1806, 1808, 1809, 1810, 1811, 1812,  
 1813, 1814, 1815, 1816, 1817, 1819,  
 1820, 1821, 1822, 1823, 1824, 1825,  
 1826, 1827, 1828, 1829, 1830, 1831,  
 1832, 1833, 1834, 1835, 1836, 1837,  
 1838, 1839, 1840, 1841, 1842, 1843,  
 1844, 1845, 1846, 1847, 1848, 1849,  
 1850, 1851, 1852, 1854, 1856, 1857,  
 1858, 1859, 1861, 1862, 1863, 1864,  
 1866, 1867, 1869, 1871, 1872, 1873,  
 1874, 1875, 1877, 1879, 1880, 1881,  
 1882, 1884, 1885, 1886, 1887, 1888,  
 1889, 1890, 1891, 1892, 1893, 1894,  
 1895, 1896, 1897, 1898, 1899, 1900,  
 1901, 1902, 1903, 1904, 1905, 1906,  
 1907, 1908, 1909, 1910, 1911, 1912,  
 1913, 1914, 1915, 1916, 1917, 1918,  
 1919, 1920, 1921, 1923, 1925, 1926,  
 1928, 1930, 1931, 1932, 1933, 1934,  
 1935, 1936, 1938, 1940, 1942, 1943,  
 1944, 1945, 1946, 1947, 1948, 1949,  
 1950, 1951, 1952, 1953, 1955, 1957,  
 1958, 1959, 1960, 1961, 1962, 1963,  
 1964, 1965, 1966, 1967, 1968, 1969,  
 1970, 1971, 1972, 1973, 1975, 1977,  
 1978, 1979, 1980, 1981, 1982, 1983,  
 1984, 1985, 1986, 1987, 1988, 1989,  
 1990, 1991, 1992, 1993, 1994, 1995,  
 1996, 1997, 1998, 1999, 2000, 2001,  
 2002, 2003, 2004, 2005, 2006, 2007,  
 2008, 2009, 2010, 2011, 2012, 2013,  
 2014, 2015, 2016, 2017, 2018, 2019,  
 2020, 2021, 2022, 2023, 2024, 2025,  
 2026, 2027, 2028, 2029, 2030, 2031,  
 2032, 2033, 2034, 2035, 2036, 2037,  
 2038, 2039, 2040, 2041, 29791, 29816  
**\\_\\_kernel\_primitives:** . . . . .  
 . . . . . 298, 1119, 1424, 2044, 29813, 29831  
**\\_\\_kernel\_randint:n** . . . . .  
 . . . . . 314, 314, 314, 633, 827,  
 831, 13151, 19019, 19031, 19189, 19274  
**\\_\\_kernel\_randint:mn** . . . . .  
 . . . . . 314, 633, 13147, 19193, 19197, 19272  
**\c\\_\\_kernel\_randint\_max\_int** . . . . .  
 . . . . . 831, 2048, 13144, 19018, 19187, 19271  
**\\_\\_kernel\_register\_log:N** . . . . .

..... 315, 3104, 7237, 11733,  
 11734, 11868, 11869, 11970, 11971  
`\_kernel_register_show:N` .....  
 ..... 315, 315,  
 404, 3104, 7233, 11729, 11864, 11966  
`\_kernel_register_show_aux:NN` 3104  
`\_kernel_register_show_aux:nNN` 3104  
`\_kernel_show>NN` ..... 3122  
`\_kernel_str_to_other:n` ... 315,  
 315, 411, 414, 418, 5358, 5410, 5471  
`\_kernel_str_to_other_fast:n` ...  
 ..... 315, 5319,  
 5339, 5381, 10696, 10792, 20613, 21741  
`\_kernel_str_to_other_fast_-  
loop:w` ..... 5381  
`\_kernel_tl_to_str:w` .....  
 . 315, 389, 2074, 4430, 4501, 4591,  
 4768, 5190, 5294, 5808, 12085, 12108  
**keys commands:**  
`\l_keys_choice_int` .....  
 ..... 171, 173, 175, 175,  
 176, 12139, 12317, 12320, 12325, 12326  
`\l_keys_choice_tl` .....  
 . 171, 173, 175, 176, 12139, 12324  
`\keys_define:nn` ... 170, 10061, 12155  
`\keys_if_choice_exist:nnnTF` ...  
 ..... 179, 12868  
`\keys_if_choice_exist_p:nnn` ...  
 ..... 179, 12868  
`\keys_if_exist:nnTF` .....  
 ..... 179, 626, 12861, 12885  
`\keys_if_exist_p:nn` ..... 179, 12861  
`\l_keys_key_tl` .....  
 ..... 177, 12142, 12258, 12274,  
 12700, 12789, 12792, 12826, 12837  
`\keys_log:nn` ..... 179, 12876  
`\l_keys_path_tl` ..... 177,  
 12146, 12183, 12203, 12212, 12221,  
 12225, 12239, 12251, 12253, 12255,  
 12267, 12269, 12271, 12286, 12289,  
 12293, 12301, 12303, 12304, 12307,  
 12322, 12345, 12350, 12359, 12363,  
 12370, 12374, 12378, 12385, 12392,  
 12403, 12409, 12413, 12429, 12438,  
 12446, 12481, 12682, 12689, 12712,  
 12715, 12754, 12758, 12766, 12768,  
 12769, 12777, 12786, 12810, 12834  
`\keys_set:nn` ..... 169,  
 173, 177, 177, 178, 12380, 12385,  
 12578, 12602, 12610, 12658, 12667  
`\keys_set_filter:nnn` .... 179, 12613  
`\keys_set_filter:nnnN` ... 179, 12613  
`\keys_set_groups:nnn` .... 179, 12613  
`\keys_set_known:nn` ..... 178, 12588  
`\keys_set_known:nnN` .. 178, 621, 12588  
`\keys_show:nn` ..... 179, 179, 12876  
`\l_keys_value_tl` ..... 177,  
 12153, 12429, 12757, 12760, 12762,  
 12770, 12794, 12806, 12818, 12828  
**keys internal commands:**  
`\_keys_bool_set:Nn` .....  
 .. 12247, 12455, 12457, 12459, 12461  
`\_keys_bool_set_inverse:Nn` ...  
 .. 12263, 12463, 12465, 12467, 12469  
`\_keys_check_groups:` .. 12716, 12724  
`\_keys_choice_find:n` .. 12280, 12831  
`\_keys_choice_find:nn` ..... 12831  
`\_keys_choice_make:` .....  
 .. 12250, 12266, 12279, 12311, 12471  
`\_keys_choice_make:N` ..... 12279  
`\_keys_choice_make_aux:N` ..... 12279  
`\_keys_choices_make:nn` .....  
 .. 12310, 12473, 12475, 12477, 12479  
`\_keys_choices_make:Nnn` ..... 12310  
`\_keys_cmd_set:nn` .....  
 .. 12251, 12253, 12255, 12267,  
 12269, 12271, 12303, 12304, 12321,  
 12331, 12378, 12385, 12446, 12481  
`\c__keys_code_root_tl` .....  
 ..... 12132, 12333, 12388,  
 12374, 12766, 12769, 12789, 12792,  
 12803, 12805, 12815, 12817, 12842,  
 12843, 12844, 12864, 12872, 12891  
`\c__keys_default_root_tl` .....  
 .. 12132, 12345, 12350, 12754, 12758  
`\_keys_default_set:n` 12260, 12276,  
 12340, 12491, 12493, 12495, 12497  
`\_keys_define:n` ..... 12160, 12164  
`\_keys_define:nn` .... 12160, 12164  
`\_keys_define:nnn` ..... 12155  
`\_keys_define_aux:nn` ..... 12164  
`\_keys_define_code:n` .. 12178, 12229  
`\_keys_define_code:w` ..... 12229  
`\_keys_execute:` .....  
 .. 12693, 12720, 12742, 12746, 12764  
`\_keys_execute:nn` ..... 12764  
`\_keys_execute_inherit:` ..... 12764  
`\_keys_execute_unknown:` .. 624, 12764  
`\l_keys_filtered_bool` .....  
 .. 12149, 12626, 12634, 12636, 12640,  
 12648, 12650, 12719, 12740, 12745  
`\_keys_find_key_module:w` .... 12670  
`\l_keys_groups_clist` ... 12141,  
 12356, 12357, 12364, 12714, 12729  
`\c__keys_groups_root_tl` .....  
 .. 12132, 12359, 12363, 12712, 12715  
`\_keys_groups_set:n` .. 12354, 12515  
`\_keys_inherit:n` ..... 12367, 12517

```

\c__keys_inherit_root_tl . . . . .
    . . . . . 12132, 12370, 12777, 12786
\l__keys_inherit_tl . . . . .
    . . . . . 12147, 12791, 12833, 12837
\__keys_initialise:n . . . . .
    . . . . . 12372, 12519, 12521, 12523, 12525
\__keys_keys_set_known:nn . . . . 12588
\__keys_meta_make:n . . . . 12376, 12535
\__keys_meta_make:nn . . . . 12376, 12537
\l__keys_module_tl . . . . .
    12143, 12156, 12159, 12161, 12205,
    12206, 12212, 12381, 12579, 12582,
    12584, 12673, 12678, 12688, 12694,
    12702, 12704, 12803, 12805, 12810
\__keys_multichoice_find:n . . .
    . . . . . 12282, 12831
\__keys_multichoice_make: . . .
    . . . . . 12279, 12313, 12539
\__keys_multichoice_make:nn . . .
    . . . . . 12310, 12541, 12543, 12545, 12547
\l__keys_no_value_bool . . . . .
    12144, 12166, 12171, 12231, 12426,
    12435, 12672, 12677, 12752, 12827
\l__keys_only_known_bool . . . . .
    . . . . . 12145, 12601, 12609, 12611, 12773
\__keys_parent:n . . . . . 12286,
    12289, 12293, 12777, 12786, 12848
\__keys_parent:w . . . . . 12848
\__keys_property_find:n 12176, 12187
\__keys_property_find:w . . . . 12187
\__keys_property_search:w . . .
    . . . . . 12213, 12217, 12226
\l__keys_property_tl . . . . .
    . . . . . 12148, 12177, 12180,
    12183, 12189, 12190, 12197, 12209,
    12222, 12234, 12235, 12238, 12242
\c__keys_props_root_tl . . . 12138,
    12177, 12235, 12242, 12454, 12456,
    12458, 12460, 12462, 12464, 12466,
    12468, 12470, 12472, 12474, 12476,
    12478, 12480, 12482, 12484, 12486,
    12488, 12490, 12492, 12494, 12496,
    12498, 12500, 12502, 12504, 12506,
    12508, 12510, 12512, 12514, 12516,
    12518, 12520, 12522, 12524, 12526,
    12528, 12530, 12532, 12534, 12536,
    12538, 12540, 12542, 12544, 12546,
    12548, 12550, 12552, 12554, 12556,
    12558, 12560, 12562, 12564, 12566,
    12568, 12570, 12572, 12574, 12576
\__keys_remove_spaces:n . . .
    . . . . . 12159, 12189,
    12322, 12582, 12686, 12842, 12843,
    12859, 12864, 12872, 12883, 12892
\l__keys_selective_bool . . . . .
    . . . . . 12149, 12657, 12666, 12668, 12691
\l__keys_selective_seq . . . . .
    . . . . . 12151, 12653, 12656, 12661, 12727
\__keys_set:n . . . . . 12583, 12670
\__keys_set:nn . . . . . 12583, 12670
\__keys_set:nnn . . . . . 12578
\__keys_set_aux: . . . . . 12670
\__keys_set_aux:nnn . . . . . 12670
\__keys_set_filter:nnn . . . . . 12613
\__keys_set_filter:nnnnN . . . . . 12613
\__keys_set_groups:nnn . . . . . 12613
\__keys_set_known:nn . . . . . 12603, 12607
\__keys_set_known:nnnN . . . . . 12588
\__keys_set_selective: . . . . . 12670
\__keys_set_selective:nn . . . . . 12613
\__keys_set_selective:nnn . . . . . 12613
\__keys_set_selective:nnnn . . . . . 12613
\__keys_show:Nnn . . . . . 12876
\__keys_store_unused: . . . . .
    . . . . . 12721, 12741, 12747, 12764
\l__keys_tmp_bool . . . . .
    . . . . . 12154, 12726, 12733, 12738
\c__keys_type_root_tl . . . . .
    . . . . . 12132, 12286, 12289, 12301
\__keys_undefine: 12369, 12386, 12573
\l__keys_unused_clist . . . . . 620,
    12152, 12589, 12593, 12595, 12596,
    12614, 12618, 12620, 12621, 12824
\__keys_validate_cleanup:w . . . 12396
\__keys_validate_forbidden: . . . 12396
\__keys_validate_required: . . . 12396
\c__keys_validate_root_tl . . . .
    . . . . . 12132, 12403, 12409, 12413, 12768
\__keys_value_or_default:n . . .
    . . . . . 12690, 12750
\__keys_value_requirement:nn . . .
    . . . . . 12396, 12575, 12577
\__keys_variable_set:NnnN . . .
    . . . . . 12443, 12483, 12485, 12487,
    12489, 12499, 12501, 12503, 12505,
    12507, 12509, 12511, 12513, 12527,
    12529, 12531, 12533, 12549, 12551,
    12553, 12555, 12557, 12559, 12561,
    12563, 12565, 12567, 12569, 12571
keyval commands:
    \keyval_parse>NNn . . . . .
        . . . . . 181, 11986, 12160, 12583
keyval internal commands:
    \keyval_action: . . . . . 12065
    \keyval_def:Nn . . . . . 12067, 12087, 12117
    \keyval_def_aux:n . . . . . 12117
    \keyval_def_aux:w . . . . . 12117
    \keyval_empty_key: . . . . . 12111, 12115

```

\l__keyval_key_tl . . . . .	937
.. 11983, 12067, 12068, 12081, 12091	
\__keyval_loop>NNw 11989, 11995, 12055	
\__keyval_sanitise_aux:w . . . . .	11999
\__keyval_sanitise_comma: . . . . .	11994, 11999
\__keyval_sanitise_comma_auxi:w . . . . .	11999
\__keyval_sanitise_comma_auxii:w . . . . .	11999
\__keyval_sanitise_equals: . . . . .	11993, 11999
\__keyval_sanitise_equals_auxi:w . . . . .	11999
\__keyval_sanitise_equals_-auxii:w . . . . .	11999
\l__keyval_sanitise_tl . . . . .	11985, 11992, 11996, 12005, 12007, 12011, 12018, 12020, 12029, 12031, 12035, 12042, 12044, 12053
\__keyval_split>NNw . . . . .	12060, 12065
\__keyval_split_tidy:w . . . . .	12065
\__keyval_split_value>NNw . . . . .	12065
\l__keyval_value_tl . . . . .	11983, 12087, 12092
\kuten . . . . .	1234, 1256, 2016, 2038
<b>L</b>	
\L . . . . .	27686
\l . . . . .	27686
13kernel . . . . .	239, 25397
13kernel.charcat . . . . .	239, 25431
13kernel.elapsedtime . . . . .	239, 25437
13kernel.filemdfivesum . . . . .	239, 25450
13kernel.filemoddate . . . . .	239, 25462
13kernel.filesize . . . . .	239, 25507
13kernel.resettimer . . . . .	239, 25437
13kernel.strcmp . . . . .	239, 25517
\label . . . . .	27747
\language . . . . .	430
\lastallocatedtoks . . . . .	19524
\lastbox . . . . .	431
\lastkern . . . . .	432
\lastlinefit . . . . .	648, 1462
\lastnamedcs . . . . .	935, 1754
\lastnode type . . . . .	649, 1463
\lastpenalty . . . . .	433
\lastsavedboxresourceindex . . . . .	1020, 1630
\lastsavedimageresourceindex . . . . .	1022, 1632
\lastsavedimageresourcepages . . . . .	1024, 1634
\lastskip . . . . .	434
\lastxpos . . . . .	1026, 1636
\lastypos . . . . .	1027, 1637
\latelua . . . . .	936, 1755
\lateluafunction . . . . .	937
LaTeX3 error commands:	
\LaTeX3_error: . . . . .	553
\lccode 173, 188, 201, 203, 205, 207, 209, 435	
\leaders . . . . .	436
\left . . . . .	437
left commands:	
\c_left_brace_str . . . . .	62, 885, 5671, 20689, 21074, 21078, 21098, 21111, 21135, 21618, 21698, 22784, 22819, 22843
\leftghost . . . . .	938, 1814
\lefthyphenmin . . . . .	438
\leftmarginkern . . . . .	795, 1609
\leftskip . . . . .	439
\leqno . . . . .	440
\let . . . . .	1, 40, 278, 279, 441
\letcharcode . . . . .	939, 1756
\letterspacefont . . . . .	796, 1610
\limits . . . . .	442
\LineBreak . . . . .	80, 81, 82, 83, 84, 85, 86, 87, 112, 119, 120, 121, 129, 131
\linedir . . . . .	940, 1815
\linedirection . . . . .	941
\linepenalty . . . . .	443
\lineskip . . . . .	444
\lineskiplimit . . . . .	445
\linewidth . . . . .	24464, 24533
\ln . . . . .	17662, 17665
\ln . . . . .	197
\localbrokenpenalty . . . . .	942, 1816
\localinterlinepenalty . . . . .	943, 1817
\localleftbox . . . . .	948, 1819
\localrightbox . . . . .	949, 1820
\locount . . . . .	10398, 10591
\loctoks . . . . .	19496, 19497, 19523
\logb . . . . .	244
\long . . . . .	281, 446, 8791, 8795
\LongText . . . . .	76, 117, 141
\looseness . . . . .	447
\lower . . . . .	448
\lowercase . . . . .	449
\lpcode . . . . .	797, 1611
lua commands:	
\lua_escape:e 238, 5174, 5189, 25350, 25352, 25391, 26103, 26632, 26645	
\lua_escape:n . . . . .	238, 25352
\lua_escape_x:n . . . . .	25389
\lua_now:e . . . . .	238, 5175, 5178, 8515, 25351, 25352, 25389, 25550, 26100, 26631
\lua_now:n . . . . .	238, 25352, 26599
\lua_now_x:n . . . . .	25389
\lua_shipout:n . . . . .	238, 25352

\lua_shipout_e:n . . . . .	238, 25352, 25393, 26644	\luatex_if_engine:TF . . . . .	29734, 29736, 29738
\lua_shipout_x:n . . . . .	25389	\luatex_if_engine_p: . . . . .	29732
lua internal commands:		\luatex_initcatcodetable:D . . . . .	1753
\__lua_escape:n . . . . .	25347, 25357, 25392	\luatex_lastnamedcs:D . . . . .	1754
\__lua_now:n . . . . .	25347, 25352, 25390	\luatex_latelua:D . . . . .	1755
\__lua_shipout:n . . . . .	25347, 25354, 25394	\luatex_leftghost:D . . . . .	1814
\luabytecode . . . . .	944	\luatex_letcharcode:D . . . . .	1756
\luabytecodecall . . . . .	945	\luatex_linedir:D . . . . .	1815
\luacopyinputnodes . . . . .	946	\luatex_localbrokenpenalty:D . . . . .	1816
\luadef . . . . .	947	\luatex_localinterlinepenalty:D . . . . .	1818
\luaeescapestring . . . . .	950, 1757	\luatex_localleftbox:D . . . . .	1819
\luafunction . . . . .	951, 1758	\luatex_localrightbox:D . . . . .	1820
\luafunctioncall . . . . .	952	\luatex_luaeescapestring:D . . . . .	1757
luatex commands:		\luatex_luafunction:D . . . . .	1758
\luatex_alignmark:D . . . . .	1713	\luatex_luatexbanner:D . . . . .	1759
\luatex_aligntab:D . . . . .	1714	\luatex_luatexrevision:D . . . . .	1760
\luatex_attribute:D . . . . .	1715	\luatex_luatexversion:D . . . . .	1761
\luatex_attributedef:D . . . . .	1716	\luatex_mathdelimitersmode:D . . . . .	1762
\luatex_automaticdiscretionary:D . . . . .	1718	\luatex_mathdir:D . . . . .	1821
\luatex_automatichyphenmode:D . . . . .	1720	\luatex_mathdisplayskipmode:D . . . . .	1764
\luatex_automatichyphenpenalty:D . . . . .	1722	\luatex_matheqngapstep:D . . . . .	1765
\luatex_begincsname:D . . . . .	1723	\luatex_mathnolimitsmode:D . . . . .	1766
\luatex_bodydir:D . . . . .	1812	\luatex_mathoption:D . . . . .	1767
\luatex_boxdir:D . . . . .	1813	\luatex_mathpenaltiesmode:D . . . . .	1768
\luatex_breakafterdirmode:D . . . . .	1724	\luatex_mathrulesfam:D . . . . .	1769
\luatex_catcodetable:D . . . . .	1725	\luatex_mathscriptboxmode:D . . . . .	1771
\luatex_clearmarks:D . . . . .	1726	\luatex_mathscriptsmode:D . . . . .	1770
\luatex_crampeddisplaystyle:D . . . . .	1728	\luatex_mathstyle:D . . . . .	1772
\luatex_crampedscriptscriptstyle:D . . . . .	1730	\luatex_mathsurroundmode:D . . . . .	1773
\luatex_crampedscriptstyle:D . . . . .	1731	\luatex_mathsurroundskip:D . . . . .	1774
\luatex_crampedtextstyle:D . . . . .	1732	\luatex_nochrule:D . . . . .	1775
\luatex_directlua:D . . . . .	1733	\luatex_nokerns:D . . . . .	1776
\luatex_dviextension:D . . . . .	1734	\luatex_noligs:D . . . . .	1777
\luatex_dvifeedback:D . . . . .	1735	\luatex_nospaces:D . . . . .	1778
\luatex_dvivariable:D . . . . .	1736	\luatex_novrule:D . . . . .	1779
\luatex_etoksapp:D . . . . .	1737	\luatex_outputbox:D . . . . .	1780
\luatex_etokspre:D . . . . .	1738	\luatex_pagebottomoffset:D . . . . .	1781
\luatex_expanded:D . . . . .	1741	\luatex_pagedir:D . . . . .	1822
\luatex_explicitdiscretionary:D . . . . .	1743	\luatex_pageleftoffset:D . . . . .	1782
\luatex_exphyphenpenalty:D . . . . .	1740	\luatex_pagerightoffset:D . . . . .	1783
\luatex_firstvalidlanguage:D . . . . .	1744	\luatex_pagetopoffset:D . . . . .	1784
\luatex_fontid:D . . . . .	1745	\luatex_pardir:D . . . . .	1823
\luatex_formatname:D . . . . .	1746	\luatex_pdfextension:D . . . . .	1785
\luatex_gleaders:D . . . . .	1752	\luatex_pdffeedback:D . . . . .	1786
\luatex_hjcode:D . . . . .	1747	\luatex_pdfvariable:D . . . . .	1787
\luatex_hpack:D . . . . .	1748	\luatex_postexhyphenchar:D . . . . .	1788
\luatex_hyphenationbounds:D . . . . .	1749	\luatex_posthyphenchar:D . . . . .	1789
\luatex_hyphenationmin:D . . . . .	1750	\luatex_prebinoppenalty:D . . . . .	1790
\luatex_hyphenpenaltytymode:D . . . . .	1751	\luatex_predisplaygapfactor:D . . . . .	1792
		\luatex_preexhyphenchar:D . . . . .	1793
		\luatex_prehyphenchar:D . . . . .	1794
		\luatex_prerelpenalty:D . . . . .	1795
		\luatex_rightghost:D . . . . .	1824

\luatex_savecatcodetable:D . . . . .	1796	\luatexposthyphenchar . . . . .	1338
\luatex_scantexttokens:D . . . . .	1797	\luatexpreexhyphenchar . . . . .	1339
\luatex_setfontid:D . . . . .	1798	\luatexprehyphenchar . . . . .	1340
\luatex_shapemode:D . . . . .	1799	\luatexrevision . . . . .	954, 1760
\luatex_suppressifcsnameerror:D	1801	\luatexrightghost . . . . .	1367
\luatex_suppresslongerror:D ..	1802	\luatexsavecatcodetable . . . . .	1341
\luatex_suppressmathparerror:D	1804	\luatexscantexttokens . . . . .	1342
\luatex_suppressoutererror:D ..	1805	\luatexsuppressfontnotfounderror . . . . .	1312, 1351
\luatex_suppressprimitiveerror:D .....	1807	\luatexsuppressifcsnameerror . . . . .	1344
\luatex_textdir:D . . . . .	1825	\luatexsuppresslongerror . . . . .	1345
\luatex_toksapp:D . . . . .	1808	\luatexsuppressmathparerror . . . . .	1347
\luatex_tokspre:D . . . . .	1809	\luatexsuppressoutererror . . . . .	1348
\luatex_tpack:D . . . . .	1810	\luatextextdir . . . . .	1368
\luatex_vpack:D . . . . .	1811	\luatextracingfonts . . . . .	1308
\luatexalignmark . . . . .	1313	\luatexUchar . . . . .	1349
\luatexaligntab . . . . .	1314	\luatexversion . . . . .	45, 107, 955, 1761
\luatexattribute . . . . .	1315		
\luatexattributedef . . . . .	1316		
\luatexbanner . . . . .	953, 1759		
\luatexbodydir . . . . .	1352		
\luatexboxdir . . . . .	1353		
\luatexcatcodetable . . . . .	1317		
\luatexclearmarks . . . . .	1318		
\luatexcrampeddisplaystyle . . . . .	1319		
\luatexcrampedscriptscriptstyle ..	1321		
\luatexcrampedscriptstyle . . . . .	1322		
\luatexcrampedtextstyle . . . . .	1323		
\luatexfontid . . . . .	1324		
\luatexformatname . . . . .	1325		
\luatexleaders . . . . .	1326		
\luatexinitcatcodetable . . . . .	1327		
\luatexlatelua . . . . .	1328		
\luatexleftghost . . . . .	1354		
\luatexlocalbrokenpenalty . . . . .	1355		
\luatexlocalinterlinepenalty . . .	1357		
\luatexlocalleftbox . . . . .	1358		
\luatexlocalrightbox . . . . .	1359		
\luatexluaescapestring . . . . .	1329		
\luatexluafunction . . . . .	1330		
\luatexmathdir . . . . .	1360		
\luatexmathstyle . . . . .	1331		
\luatexnokerns . . . . .	1332		
\luatexnoligs . . . . .	1333		
\luatexoutputbox . . . . .	1334		
\luatexpagebottomoffset . . . . .	1361		
\luatexpagedir . . . . .	1362		
\luatexpageheight . . . . .	1363		
\luatexpageleftoffset . . . . .	1335		
\luatexpagerightoffset . . . . .	1364		
\luatexpagetopoffset . . . . .	1336		
\luatexpagewidth . . . . .	1365		
\luatexpardir . . . . .	1366		
\luatexpostexhyphenchar . . . . .	1337		

**M**

\mag . . . . .	450
\mark . . . . .	451
\marks . . . . .	650, 1464
math commands:	
\c_math_subscript_token . . . . .	
..... 122, 509, 8606, 8664, 20127	
\c_math_superscript_token . . . . .	
..... 122, 509, 8606, 8659, 20125	
\c_math_toggle_token . . . . .	
..... 122, 508, 8606, 8640, 20121	
\mathaccent . . . . .	452
\mathbin . . . . .	453
\mathchar . . . . .	454, 8790
\mathchardef . . . . .	455
\mathchoice . . . . .	456
\mathclose . . . . .	457
\mathcode . . . . .	458
\mathdelimitersmode . . . . .	956, 1762
\mathdir . . . . .	957, 1821
\mathdirection . . . . .	958
\mathdisplayskipmode . . . . .	959, 1763
\matheqnoreset . . . . .	960, 1765
\mathinner . . . . .	459
\mathnolimitsmode . . . . .	961, 1766
\mathop . . . . .	460
\mathopen . . . . .	461
\mathoption . . . . .	962, 1767
\mathord . . . . .	462
\mathpenaltiesmode . . . . .	963, 1768
\mathpunct . . . . .	463
\mathrel . . . . .	464
\mathrulesfam . . . . .	964, 1769
\mathscriptboxmode . . . . .	966, 1771
\mathscriptchmode . . . . .	967
\mathscriptsmode . . . . .	965, 1770

\mathstyle . . . . .	968, 1772	\msg_error:nnn . . . . .	141, 9630
\mathsurround . . . . .	465	\msg_error:nnnn . . . . .	141, 9630
\mathsurroundmode . . . . .	969, 1773	\msg_error:nnnnn . . . . .	141, 9630
\mathsurroundskip . . . . .	970, 1774	\msg_error:nnnnnn . . . . .	141, 247, 9630
max . . . . .	197	\msg_error_text:n . . . . .	139, 9520, 9637
max commands:		\msg_expandable_error:nn . . . . .	248, 26176
\c_max_char_int . . . . .	90, 7243, 8489, 20665	\msg_expandable_error:nnn . . . . .	248, 26176
\c_max_register_int . . . . .	90, 218, 839, 840, 2101, 6382, 10006, 19468, 19494, 19531, 19539, 19543, 26353, 26383, 26385	\msg_expandable_error:nnnn . . . . .	248, 26176
\maxdeadcycles . . . . .	466	\msg_expandable_error:nnnnn . . . . .	248, 26176
\maxdepth . . . . .	467	\msg_fatal:nn . . . . .	141, 9612
\mdfivesum . . . . .	884, 1617	\msg_fatal:nnn . . . . .	141, 9612
\meaning . . . . .	468	\msg_fatal:nnnn . . . . .	141, 9612
\medmuskip . . . . .	469	\msg_fatal:nnnnn . . . . .	141, 9612
\message . . . . .	470	\msg_fatal_text:n . . . . .	139, 9520, 9615
\MessageBreak . . . . .	129	\msg_gset:nnn . . . . .	138, 9377
meta commands:		\msg_gset:nnnn . . . . .	138, 9377
.meta:n . . . . .	173, 12534	\msg_if_exist:nnTF . . . . .	
.meta:nn . . . . .	173, 12536	. . . . .	139, 9359, 9371, 9748
\middle . . . . .	651, 1465	\msg_if_exist_p:nn . . . . .	139, 9359
min . . . . .	197	\msg_info:nn . . . . .	142, 9671
minus commands:		\msg_info:nnn . . . . .	142, 9671
\c_minus_inf_fp . . . . .	192, 201, 13200, 16189, 16273, 16606, 17141, 17918, 19443	\msg_info:nnnn . . . . .	142, 9671
\c_minus_zero_fp . . . . .	191, 13200, 16185, 18637, 19441	\msg_info:nnnnn . . . . .	142, 9671
\mkern . . . . .	471	\msg_info:nnnnnn . . . . .	142, 142, 9671, 9921
mm . . . . .	201	\msg_info_text:n . . . . .	140, 9520, 9673
mode commands:		\msg_interrupt:nnn . . . . .	10313
\mode_if_horizontal:TF . . . . .	102, 7659	\msg_line_context: . . . . .	
\mode_if_horizontal_p: . . . . .	102, 7659	. . . . .	139, 533, 2829, 2838,
\mode_if_inner:TF . . . . .	102, 7661	3936, 9367, 9434, 10139, 10160, 12334	
\mode_if_inner_p: . . . . .	102, 7661	\msg_line_number: . . . . .	139, 9434, 12126
\mode_if_math:TF . . . . .	102, 7663	\msg_log:n . . . . .	10299
\mode_if_math_p: . . . . .	102, 7663	\msg_log:nn . . . . .	142, 9693
\mode_if_vertical:TF . . . . .	102, 7657	\msg_log:nnn . . . . .	142, 9693
\mode_if_vertical_p: . . . . .	102, 7657	\msg_log:nnnn . . . . .	142, 9693
\mode_leave_vertical: . . . . .	242, 25047, 25681	\msg_log:nnnnn . . . . .	142, 9693
\month . . . . .	472	\msg_log:nnnnnn . . . . .	142,
\moveleft . . . . .	473	6363, 8317, 8330, 9339, 9693, 10473, 10645, 11281, 12879, 13094, 25274	
\ moveright . . . . .	474	\msg_log_eval:Nn . . . . .	248, 7240, 7430,
msg commands:		11736, 11871, 11973, 15425, 26203	
\msg_critical:nn . . . . .	141, 9621	\g_msg_module_documentation_prop . . . . .	140
\msg_critical:nn(nn) . . . . .	246	\msg_module_name:n . . . . .	
\msg_critical:nnn . . . . .	141, 9621	. . . . .	140, 9444, 9539, 9562, 9652, 9674
\msg_critical:nnnn . . . . .	141, 9621	\g_msg_module_name_prop . . . . .	
\msg_critical:nnnnn . . . . .	141, 9621	. . . . .	140, 140, 9547, 9564, 9565
\msg_critical:nnnnnn . . . . .	141, 9621	\msg_module_type:n . . . . .	
\msg_critical_text:n . . . . .	139, 9520, 9624	. . . . .	139, 140, 140, 9538, 9551
\msg_error:nn . . . . .	141, 9630	\g_msg_module_type_prop . . . . .	
		. . . . .	140, 140, 9547, 9553, 9554
		\msg_new:nnn . . . . .	138, 9377, 9872

\msg\_new:nnnn . . . . . 138, 532, 9377, 9870  
\msg\_none:nn . . . . . 142, 9699  
\msg\_none:nnn . . . . . 142, 9699  
\msg\_none:nnnn . . . . . 142, 9699  
\msg\_none:nnnnn . . . . . 142, 9699  
\msg\_redirect\_class:nn . . . . . 143, 9821  
\msg\_redirect\_module:nnn . . . . . 143, 9821  
\msg\_redirect\_name:nnn . . . . . 143, 9812  
\msg\_see\_documentation\_text:n . . . . . 140, 9562  
\msg\_set:nnn . . . . . 138, 9377, 9876  
\msg\_set:nnnn . . . . . 138, 9377, 9874  
\msg\_show:nn . . . . . 248, 9700  
\msg\_show:nnn . . . . . 248, 9700  
\msg\_show:nnnn . . . . . 248, 9700  
\msg\_show:nnnnn . . . . . 248, 9700  
\msg\_show:nnnnnn 248, 248, 442, 500,  
 531, 6361, 8315, 8329, 9337, 9700,  
 10472, 10644, 11280, 12877, 13092,  
 20222, 20230, 22985, 22994, 25271  
\msg\_show\_eval:Nn 248, 7236, 7428,  
 11732, 11867, 11969, 15423, 26203  
\msg\_show\_item:n . . . . .  
 . . . . . 248, 248, 6371, 8325, 8334, 26208  
\msg\_show\_item:nn 248, 531, 9347, 26208  
\msg\_show\_item\_unbraced:n 248, 26208  
\msg\_show\_item\_unbraced:nn . 248,  
 559, 10480, 10652, 12887, 25290, 26208  
\msg\_term:n . . . . . 10299  
\msg\_warning:nn . . . . . 141, 9649  
\msg\_warning:nnn . . . . . 141, 9649  
\msg\_warning:nnnn . . . . . 141, 9649  
\msg\_warning:nnnnn . . . . . 141, 9649  
\msg\_warning:nnnnnn . . . . . 141, 9649, 9920  
\msg\_warning\_text:n . . . . . 139, 9520, 9651

msg internal commands:

\\_msg\_chk\_free:nn . . . . . 9369, 9379  
\\_msg\_chk\_if\_free:nn . . . . . 9364  
\\_msg\_class\_chk\_exist:nTF . . . . .  
 . . . . . 9735, 9750, 9817, 9827, 9832  
\l\_msg\_class\_loop\_seq . 544, 9744,  
 9836, 9844, 9854, 9855, 9858, 9860  
\\_msg\_class\_new:nn . . . . .  
 . . . . . 541, 545, 9573, 9612, 9621,  
 9630, 9649, 9671, 9693, 9699, 9700  
\l\_msg\_class\_t1 . . . . . 542,  
 544, 9740, 9757, 9770, 9791, 9795,  
 9798, 9806, 9845, 9847, 9849, 9863  
\c\_msg\_coding\_error\_text\_t1 9402,  
 9926, 9934, 9960, 9978, 9987, 9999,  
 10013, 10022, 10044, 10053, 10060,  
 10069, 10075, 10082, 10089, 10096,  
 10103, 10111, 10119, 10149, 10162  
\c\_\_msg\_continue\_text\_t1 . . . . .  
 . . . . . 9402, 9448, 10319  
\c\_\_msg\_critical\_text\_t1 . 9402, 9627  
\l\_\_msg\_current\_class\_t1 . . . . .  
 . . . . . 544, 9740, 9752,  
 9790, 9795, 9798, 9806, 9835, 9849  
\\_\_msg\_error:Nnnnn . . . . . 9630  
\\_\_msg\_error\_code:nnnnm . . . . . 9919  
\\_\_msg\_expandable\_error:n . . . . .  
 . . . . . 553, 10249, 10268  
\\_\_msg\_expandable\_error:w 553, 10249  
\\_\_msg\_expandable\_error\_module:nn . . . . .  
 . . . . . 26176  
\\_\_msg\_fatal\_code:nnnnnn . . . . . 9915  
\c\_\_msg\_fatal\_text\_t1 . 9402, 9618  
\c\_\_msg\_help\_text\_t1 9402, 9453, 10323  
\l\_\_msg\_hierarchy\_seq . . . . .  
 . . . . . 542, 543, 9743, 9773, 9783, 9788  
\\_\_msg\_interrupt:n . . . . . 9475, 9484  
\\_\_msg\_interrupt:Nnnn . . . . .  
 . . . . . 9441, 9614, 9623, 9636  
\\_\_msg\_interrupt\_more\_text:n . . . . .  
 . . . . . 534, 9457  
\\_\_msg\_interrupt\_text:n . . . . . 9457  
\\_\_msg\_interrupt\_wrap:nnn . . . . .  
 . . . . . 9447, 9452, 9457  
\\_\_msg\_kernel\_class\_new:nN . . . . .  
 . . . . . 546, 9877, 9915, 9919, 9920, 9921  
\\_\_msg\_kernel\_class\_new\_aux:nN 9877  
\c\_\_msg\_more\_text\_prefix\_t1 . . . . .  
 . . . . . 9357, 9388, 9397, 9633  
\l\_\_msg\_name\_str . . . . .  
 . . . . . 9355, 9444, 9464, 9468,  
 9652, 9660, 9664, 9674, 9682, 9686  
\c\_\_msg\_no\_info\_text\_t1 . . . . .  
 . . . . . 9402, 9449, 10318  
\\_\_msg\_no\_more\_text:nnnn . . . . . 9630  
\\_msg\_old\_interrupt\_more\_text:n . . . . .  
 . . . . . 10328, 10331  
\\_msg\_old\_interrupt\_text:n . . . . .  
 . . . . . 10329, 10348  
\\_msg\_old\_interrupt\_wrap:nn . . . . .  
 . . . . . 10318, 10322, 10326  
\c\_\_msg\_on\_line\_text\_t1 . . 9402, 9437  
\\_\_msg\_redirect:nnn . . . . . 9821  
\\_\_msg\_redirect\_loop\_chk:nnn . . . . .  
 . . . . . 9821, 9863  
\\_\_msg\_redirect\_loop\_list:n . . 9821  
\l\_\_msg\_redirect\_prop . . . . .  
 . . . . . 9742, 9770, 9815, 9818  
\c\_\_msg\_return\_text\_t1 . . . . .  
 . . . . . 9402, 9929, 9937, 9944  
\\_\_msg\_show:n . . . . . 540, 9700  
\\_\_msg\_show:nn . . . . . 9700

\\_\_msg\_show:w ..... 9700  
 \\_\_msg\_show\_dot:w ..... 9700  
 \\_\_msg\_show\_eval:nnN ..... 26203  
 \\_\_msg\_text:n ..... 9520  
 \\_\_msg\_text:nn ..... 9520  
 \c\_\_msg\_text\_prefix\_tl . 553, 9357,  
     9361, 9386, 9395, 9617, 9626, 9639,  
     9657, 9679, 9696, 9703, 10271, 26181  
 \l\_\_msg\_text\_str .....  
     ..... 9355, 9443, 9462, 9467,  
     9651, 9656, 9663, 9673, 9678, 9685  
 \\_\_msg\_tmp:w ..... 10250, 10263  
 \l\_\_msg\_tmp\_tl .....  
     ..... 9354, 9474, 9480, 9724, 9730  
 \c\_\_msg\_trouble\_text\_tl ..... 9402  
 \\_\_msg\_use:nnnnnn ..... 9583, 9745  
 \\_\_msg\_use\_code: ..... 542, 9745  
 \\_\_msg\_use\_hierarchy:nwwN ..... 9745  
 \\_\_msg\_use\_redirect\_module:n ...  
     ..... 543, 9745  
 \\_\_msg\_use\_redirect\_name:n ... 9745  
\mskip ..... 475  
\muexpr ..... 652, 1466  
multichoice commands:  
 .multichoice: ..... 173, 12538  
multicoices commands:  
 .multicoices:nn ..... 173, 12538  
\multiply ..... 476  
\muskip ..... 477, 8798  
muskip commands:  
 \c\_max\_muskip ..... 168, 11974  
 \muskip\_add:Nn ..... 166, 11939  
 \muskip\_const:Nn .....  
     ..... 166, 11886, 11974, 11975  
 \muskip\_eval:n .....  
     ..... 167, 167, 11890, 11955, 11969, 11973  
 \muskip\_gadd:Nn ..... 166, 11939  
 \muskip\_gset:Nn ..... 166, 11925  
 \muskip\_gset\_eq:NN ..... 166, 11933  
 \muskip\_gsub:Nn ..... 167, 11939  
 \muskip\_gzero:N ... 166, 11893, 11904  
 \muskip\_gzero\_new:N ..... 166, 11901  
 \muskip\_if\_exist:NTF .....  
     ..... 166, 11902, 11904, 11907  
 \muskip\_if\_exist\_p:N .... 166, 11907  
 \muskip\_log:N ..... 167, 11970  
 \muskip\_log:n ..... 167, 11970  
 \muskip\_new:N .....  
     ... 166, 166, 11878, 11889, 11902,  
     11904, 11976, 11977, 11978, 11979  
 \muskip\_set:Nn ..... 166, 11925  
 \muskip\_set\_eq:NN ..... 166, 11933  
 \muskip\_show:N ..... 167, 11966  
 \muskip\_show:n ..... 167, 602, 11968  
 \muskip\_sub:Nn ..... 167, 11939  
 \muskip\_use:N . 167, 167, 11963, 11964  
 \muskip\_zero:N 166, 166, 11893, 11902  
 \muskip\_zero\_new:N ..... 166, 11901  
 \g\_tmpa\_muskip ..... 168, 11976  
 \l\_tmpa\_muskip ..... 168, 11976  
 \g\_tmpb\_muskip ..... 168, 11976  
 \l\_tmpb\_muskip ..... 168, 11976  
 \c\_zero\_muskip 168, 11895, 11898, 11974  
\muskipdef ..... 478  
\mutoglu ..... 653, 1467

**N**

nan ..... 201  
 nc ..... 201  
 nd ..... 201  
 \newbox ..... 445  
 \newcatcodetable ..... 52  
 \newcount ..... 445  
 \newdimen ..... 445  
 \newlinechar ..... 110, 479  
 \next ..... 74, 113,  
     129, 129, 129, 138, 147, 151, 154, 162  
 \NG ..... 27687  
 \ng ..... 27687  
 \noalign ..... 480  
 \noautospacing ..... 1235, 2017  
 \noautoxspacing ..... 1236, 2018  
 \noboundary ..... 481  
 \noexpand ..... 125, 129, 140, 143, 482  
 \nohrule ..... 971, 1775  
 \noindent ..... 483  
 \nokerns ..... 972, 1776  
 \noligs ..... 973, 1777  
 \nolimits ..... 484  
 \nonscript ..... 485  
 \nonstopmode ..... 486  
 \normaldeviate ..... 1028, 1638  
 \normalend 1386, 1387, 10394, 10427, 10587  
 \normaleveryjob ..... 1388  
 \normalexpanded ..... 1397  
 \normalhoffset ..... 1400  
 \normalinput ..... 1389  
 \normalitaliccorrection ..... 1399, 1401  
 \normallanguage ..... 1390  
 \normalleft ..... 1407, 1408  
 \normalmathop ..... 1391  
 \normalmiddle ..... 1409  
 \normalmonth ..... 1392  
 \normalouter ..... 1393  
 \normalover ..... 1394  
 \normalright ..... 1410  
 \normalshowtokens ..... 1403  
 \normalunexpanded ..... 1396

\normalvcenter . . . . .	1395	
\normalvoffset . . . . .	1402	
\nospaces . . . . .	974, 1778	
notexpanded commands:		
\notexpanded: <token> . . . . .	130	
\novrule . . . . .	975, 1779	
\nulldelimiterspace . . . . .	487	
\nullfont . . . . .	488	
\num . . . . .	185	
\number . . . . .	55, 489	
\numexpr . . . . .	174, 188, 654, 1468	
<b>O</b>		
\O . . . . .	27688	
\o . . . . .	27688	
\OE . . . . .	27689	
\oe . . . . .	27689	
\omit . . . . .	490	
one commands:		
\c_minus_one . . . . .	29694	
\c_one_degree_fp	192, 201, 14799, 15428	
\openin . . . . .	491	
\openout . . . . .	492	
\or . . . . .	493	
or commands:		
\or: . . . . .	91, 415, 417, 648, 2049, 2928, 2929, 2930, 2931, 2932, 2933, 2934, 2935, 2936, 3577, 3578, 3579, 3580, 3581, 5461, 5537, 6382, 7023, 7024, 7025, 7026, 7027, 7028, 7029, 7030, 7031, 7032, 7033, 7034, 7035, 7036, 7037, 7038, 7039, 7040, 7041, 7042, 7043, 7044, 7045, 7046, 7047, 7056, 7057, 7058, 7059, 7060, 7061, 7062, 7063, 7064, 7065, 7066, 7067, 7068, 7069, 7070, 7071, 7072, 7073, 7074, 7075, 7076, 7077, 7078, 7079, 7080, 8541, 8545, 8548, 8550, 8551, 8553, 8555, 8557, 8558, 8560, 8562, 8564, 8566, 8599, 10871, 10872, 10873, 10874, 10875, 10876, 10877, 13239, 13240, 13241, 13490, 13505, 13506, 13902, 13903, 13928, 15182, 15183, 15184, 15220, 15885, 15886, 15887, 16010, 16095, 16181, 16182, 16183, 16184, 16185, 16186, 16187, 16188, 16189, 16268, 16271, 16607, 16608, 16622, 16623, 16637, 16921, 17142, 17167, 17173, 17174, 17175, 17176, 17177, 17326, 17361, 17363, 17371, 17564, 17615, 17618, 17627, 17742, 17765, 17766, 17827, 17832, 17842, 17847, 17857, 17862, 17872, 17877, 17887,	
<b>P</b>		
\PackageError . . . . .	132, 140	
\pagebottomoffset . . . . .	977, 1781	
\pagedepth . . . . .	501	
\pagedir . . . . .	978, 1822	
\pagedirection . . . . .	979	
\pagediscards . . . . .	655, 1469	
\pagefullstretch . . . . .	502	
\pagefillstretch . . . . .	503	
\pagefilstretch . . . . .	504	
\pagegoal . . . . .	505	
\pageheight . . . . .	1030, 1640	
\pageleftoffset . . . . .	980, 1782	
\pagerightoffset . . . . .	981, 1783	
\pageshrink . . . . .	506	
\pagestretch . . . . .	507	
\pagetopoffset . . . . .	982, 1784	
\pagetotal . . . . .	508	
\pagewidth . . . . .	1031, 1641	
\par . . . . .	9, 10, 10, 11, 11, 11, 12, 12, 12, 13, 13, 13, 146, 341, 509, 969, 23851, 23853, 23857, 23862, 23868, 23874, 23882, 23888, 23896, 23902, 23924	
\pardir . . . . .	983, 1823	
\pardirection . . . . .	984	
\parfillskip . . . . .	510	
\parindent . . . . .	511	
\parshape . . . . .	512	
\parshapedimen . . . . .	656, 1470	
\parshapeindent . . . . .	657, 1471	
\parshapelength . . . . .	658, 1472	
\parskip . . . . .	513	
\patterns . . . . .	514	

\pausing .....	515	\pdflinkmargin .....	716, 1531
pc .....	201	\pdfliteral .....	717, 1532
\pdfadjustspacing .....	753, 1570	\pdfmapfile .....	770, 1585
\pdfannot .....	681, 1495	\pdfmapline .....	771, 1586
\pdfcatalog .....	682, 1496	\pdfmdfivesum .....	772, 1587
\pdfcolorstack .....	684, 1498	\pdfminorversion .....	718, 1533
\pdfcolorstackinit .....	685, 1499	\pdfnames .....	719, 1534
\pdfcompresslevel .....	683, 1497	\pdfnoligatures .....	773, 1588
\pdfcopyfont .....	754, 1571	\pdfnormaldeviate .....	774, 1589
\pdfcreationdate .....	686, 1500	\pdfobj .....	720, 1535
\pdfdecimaldigits .....	687, 1501	\pdfobjcompresslevel .....	721, 1536
\pdfdest .....	688, 1502	\pdfoutline .....	722, 1538
\pdfdestmargin .....	689, 1503	\pdfoutput .....	723, 1539
\pdfdraftmode .....	755, 1572	\pdfpageattr .....	724, 1540
\pdfeachlinedepth .....	756, 1573	\pdfpagebox .....	725, 1541
\pdfeachlineheight .....	757, 1574	\pdfpageheight .....	775, 1590
\pdfelapsetime .....	758	\pdfpageref .....	726, 1542
\pdfendlink .....	690, 1504	\pdfpageresources .....	727, 1543
\pdfendthread .....	691, 1505	\pdfpagesattr .....	728, 1544
\pdfextension .....	985, 1785	\pdfpagewidth .....	776, 1591
\pdffeedback .....	986, 1786	\pdfpkmode .....	777, 1592
\pdffiledump .....	759	\pdfpkresolution .....	778, 1593
\pdffilemoddate .....	760, 1575	\pdfprimitive .....	779, 1594
\pdffilesize .....	761, 1576	\pdfprotrudechars .....	780, 1595
\pdffirstlineheight .....	762, 1577	\pdfpxdimen .....	781, 1596
\pdffontattr .....	692, 1506	\pdfrandomseed .....	782, 1597
\pdffontexpand .....	763, 1578	\pdfrefobj .....	729, 1545
\pdffontname .....	693, 1507	\pdfrefxform .....	730, 1546
\pdffontobjnum .....	694, 1508	\pdfrefiximage .....	731, 1547
\pdffontsize .....	764, 1579	\pdfresettimer .....	783
\pdfgamma .....	695, 1509	\pdfrestore .....	732, 1548
\pdfgentounicode .....	698, 1512	\pdfretval .....	733, 1549
\pdfglyptounicode .....	699, 1513	\pdfsave .....	734, 1550
\pdfhorigin .....	700, 1514	\pdfsavepos .....	784, 1598
\pdfignoreddimen .....	765, 1580	\pdfsetmatrix .....	735, 1551
\pdfimageapplygamma .....	696, 1510	\pdfsetrandomseed .....	786, 1600
\pdfimagegamm .....	697, 1511	\pdfshellescape .....	787, 1601
\pdfimagehicolor .....	701, 1515	\pdfstartlink .....	736, 1552
\pdfimageresolution .....	702, 1516	\pdfstartthread .....	737, 1553
\pdfincludetext .....	703, 1517	\pdfstrcmp .....	40, 409, 785, 1599
\pdfinclusioncopyfonts .....	704, 1518	\pdfsuppressptexinfo .....	738, 1554
\pdfinclusionerrorlevel .....	705, 1520	\pdftex commands:	
\pdfinfo .....	707, 1522	\pdftex_adjustspacing:D ..	1570, 1621
\pdfinsertht .....	766, 1581	\pdftex_copyfont:D .....	1571, 1622
\pdflastannot .....	708, 1523	\pdftex_draftmode:D .....	1572, 1623
\pdflastlinedepth .....	767, 1582	\pdftex_eachlinedepth:D .....	1573
\pdflastlink .....	709, 1524	\pdftex_eachlineheight:D .....	1574
\pdflastobj .....	710, 1525	\pdftex_efcode:D .....	1607
\pdflastxform .....	711, 1526	\pdftex_filemoddate:D .....	1575
\pdflastximage .....	712, 1527	\pdftex_filesize:D .....	1576
\pdflastximagecolordepth .....	713, 1528	\pdftex_firstlineheight:D .....	1577
\pdflastximagepages .....	715, 1530	\pdftex_fontexpand:D .....	1578, 1624
\pdflastxpos .....	768, 1583	\pdftex_fontsize:D .....	1579
\pdflastypos .....	769, 1584		

\pdftex\_if\_engine:TF ..... 29742, 29744, 29746  
\pdftex\_if\_engine\_p: ..... 29740  
\pdftex\_ifabsdim:D ..... 1567, 1625  
\pdftex\_ifabsnum:D ..... 1568, 1626  
\pdftex\_ifincsname:D ..... 1608  
\pdftex\_ifprimitive:D ... 1569, 1618  
\pdftex\_ignoreddimen:D ..... 1580  
\pdftex\_ignoreligaturesinfont:D 1628  
\pdftex\_insertht:D ..... 1581, 1629  
\pdftex\_lastlinedepth:D ..... 1582  
\pdftex\_lastxpos:D ..... 1583, 1636  
\pdftex\_lastypos:D ..... 1584, 1637  
\pdftex\_leftmarginkern:D ..... 1609  
\pdftex\_letterspacefont:D ..... 1610  
\pdftex\_lpcode:D ..... 1611  
\pdftex\_mapfile:D ..... 1585  
\pdftex\_mapline:D ..... 1586  
\pdftex\_mdfivesum:D ..... 1587, 1617  
\pdftex\_noligatures:D ..... 1588  
\pdftex\_normaldeviate:D ... 1589, 1638  
\pdftex\_pageheight:D ..... 1590, 1640  
\pdftex\_pagewidth:D ..... 1591  
\pdftex\_pagewidth:D ..... 1641  
\pdftex\_pdfannot:D ..... 1495  
\pdftex\_pdfcatalog:D ..... 1496  
\pdftex\_pdfcolorstack:D ..... 1498  
\pdftex\_pdfcolorstackinit:D .. 1499  
\pdftex\_pdfcompresslevel:D ... 1497  
\pdftex\_pdfcreationdate:D .... 1500  
\pdftex\_pdfdecimaldigits:D ... 1501  
\pdftex\_pdfdest:D ..... 1502  
\pdftex\_pdfdestmargin:D ..... 1503  
\pdftex\_pdfendlink:D ..... 1504  
\pdftex\_pdfendthread:D ..... 1505  
\pdftex\_pdffontattr:D ..... 1506  
\pdftex\_pdffontname:D ..... 1507  
\pdftex\_pdffontobjnum:D ..... 1508  
\pdftex\_pdfgamma:D ..... 1509  
\pdftex\_pdfgentounicode:D ... 1512  
\pdftex\_pdfglyphtounicode:D .. 1513  
\pdftex\_pdfhorigin:D ..... 1514  
\pdftex\_pdfimageapplygamma:D .. 1510  
\pdftex\_pdfimagegamma:D ..... 1511  
\pdftex\_pdfimagehicolor:D ... 1515  
\pdftex\_pdfimageresolution:D .. 1516  
\pdftex\_pdfincludechars:D ... 1517  
\pdftex\_pdfinclusioncopyfonts:D 1519  
\pdftex\_pdfinclusionerrorlevel:D ..... 1521  
\pdftex\_pdfinfo:D ..... 1522  
\pdftex\_pdflastannot:D ..... 1523  
\pdftex\_pdflastlink:D ..... 1524  
\pdftex\_pdflastobj:D ..... 1525  
\pdftex\_pdflastxform:D ... 1526, 1631  
\pdftex\_pdflastximage:D ... 1527, 1633  
\pdftex\_pdflastximagecolordepth:D ..... 1529  
\pdftex\_pdflastximagepages:D ... ..... 1530, 1635  
\pdftex\_pdflinkmargin:D ..... 1531  
\pdftex\_pdfliteral:D ..... 1532  
\pdftex\_pdfminorversion:D .... 1533  
\pdftex\_pdfnames:D ..... 1534  
\pdftex\_pdfobj:D ..... 1535  
\pdftex\_pdfobjcompresslevel:D . 1537  
\pdftex\_pdfoutline:D ..... 1538  
\pdftex\_pdfoutput:D ..... 1539, 1639  
\pdftex\_pdfpageattr:D ..... 1540  
\pdftex\_pdfpagebox:D ..... 1541  
\pdftex\_pdfpageref:D ..... 1542  
\pdftex\_pdfpageresources:D ... 1543  
\pdftex\_pdfpagesattr:D ..... 1544  
\pdftex\_pdfrefobj:D ..... 1545  
\pdftex\_pdfrefxform:D ... 1546, 1645  
\pdftex\_pdfrefximage:D ... 1547, 1646  
\pdftex\_pdfrestore:D ..... 1548  
\pdftex\_pdfretval:D ..... 1549  
\pdftex\_pdfsave:D ..... 1550  
\pdftex\_pdfsetmatrix:D ..... 1551  
\pdftex\_pdfstartlink:D ..... 1552  
\pdftex\_pdfstartthread:D ..... 1553  
\pdftex\_pdfsuppressptexinfo:D . 1555  
\pdftex\_pdftexbanner:D ..... 1604  
\pdftex\_pdftexrevision:D ..... 1605  
\pdftex\_pdftexversion:D ..... 1606  
\pdftex\_pdfthread:D ..... 1556  
\pdftex\_pdfthreadmargin:D .... 1557  
\pdftex\_pdftrailer:D ..... 1558  
\pdftex\_pdfuniqueresname:D ... 1559  
\pdftex\_pdfvorigin:D ..... 1560  
\pdftex\_pdxform:D ..... 1561, 1648  
\pdftex\_pdxformattr:D ..... 1562  
\pdftex\_pdxformname:D ..... 1563  
\pdftex\_pdxformresources:D .. 1564  
\pdftex\_pdximage:D ..... 1565, 1649  
\pdftex\_pdximagebbox:D ..... 1566  
\pdftex\_pkmode:D ..... 1592  
\pdftex\_pkresolution:D ..... 1593  
\pdftex\_primitive:D ..... 1594, 1619  
\pdftex\_protrudechars:D ... 1595, 1642  
\pdftex\_pxdimen:D ..... 1596, 1643  
\pdftex\_quitvmode:D ..... 1612  
\pdftex\_randomseed:D ..... 1597, 1644  
\pdftex\_rightmarginkern:D .... 1613  
\pdftex\_rpcode:D ..... 1614  
\pdftex\_savepos:D ..... 1598, 1647  
\pdftex\_setrandomseed:D .. 1600, 1650

\pdftex\_shellescape:D . . . . . 1601, 1620  
 \pdftex\_strcmp:D . . . . . 1599  
 \pdftex\_syntex:D . . . . . 1615  
 \pdftex\_tagcode:D . . . . . 1616  
 \pdftex\_tracingfonts:D . . . . . 1602, 1651  
 \pdftex\_uniformdeviate:D . . . . . 1603, 1652  
 \pdftexbanner . . . . . 790, 1604  
 \pdftexrevision . . . . . 791, 1605  
 \pdftexversion . . . . . 102, 792, 1606  
 \pdfthread . . . . . 739, 1556  
 \pdfthreadmargin . . . . . 740, 1557  
 \pdftracingfonts . . . . . 788, 1303, 1304, 1602  
 \pdftrailer . . . . . 741, 1558  
 \pdfuniformdeviate . . . . . 789, 1603  
 \pdfuniqueresname . . . . . 742, 1559  
 \pdfvariable . . . . . 987, 1787  
 \pdfvorigin . . . . . 743, 1560  
 \pdfxform . . . . . 744, 1561  
 \pdfxformattr . . . . . 745, 1562  
 \pdfxformname . . . . . 746, 1563  
 \pdfxformresources . . . . . 747, 1564  
 \pdfximage . . . . . 748, 1565  
 \pdfximagebbox . . . . . 749, 1566  
 peek commands:  
   \peek\_after:Nw . . . . . 104,  
     126, 126, 126, 8872, 8885, 8913, 27968  
   \peek\_catcode:NTF . . . . . 126, 8968  
   \peek\_catcode\_collect\_inline:Nn .  
     . . . . . 260, 27948  
   \peek\_catcode\_ignore\_spaces:NTF .  
     . . . . . 127, 8982  
   \peek\_catcode\_remove:NTF . . 127, 8968  
   \peek\_catcode\_remove\_ignore\_-  
     spaces:NTF . . . . . 127, 8982  
   \peek\_charcode:NTF . . . . . 127, 8968  
   \peek\_charcode\_collect\_inline:Nn .  
     . . . . . 260, 27948  
   \peek\_charcode\_ignore\_spaces:NTF .  
     . . . . . 127, 8982  
   \peek\_charcode\_remove:NTF . . 127, 8968  
   \peek\_charcode\_remove\_ignore\_-  
     spaces:NTF . . . . . 128, 8982  
   \peek\_gafter:Nw . . . . . 126, 126, 8872  
   \peek\_meaning:NTF . . . . . 128, 8968  
   \peek\_meaning\_collect\_inline:Nn .  
     . . . . . 260, 27948  
   \peek\_meaning\_ignore\_spaces:NTF .  
     . . . . . 128, 8982  
   \peek\_meaning\_remove:NTF . . 128, 8968  
   \peek\_meaning\_remove\_ignore\_-  
     spaces:NTF . . . . . 128, 8982  
   \peek\_N\_type:TF . . . . . 260, 27906, 27943, 27945  
   \peek\_remove\_spaces:n . . . . .  
     . . . . . 260, 519, 8881, 8991, 8996, 9001  
 peek internal commands:  
   \\_\_peek\_collect:N . . . . . 1073, 27948  
   \\_\_peek\_collect>NNn . . . . . 27948  
   \\_\_peek\_collect\_remove:nw . . . . . 27948  
   \l\_\_peek\_collect\_tl . . . . . 1073,  
     27947, 27959, 27961, 27986, 27991  
   \\_\_peek\_collect\_true:w . . . . . 1073, 27948  
   \\_\_peek\_execute\_branches\_...: . . 1073  
   \\_\_peek\_execute\_branches\_-  
     catcode: . . . . . 519, 8935, 27949  
   \\_\_peek\_execute\_branches\_-  
     catcode\_aux: . . . . . 8935  
   \\_\_peek\_execute\_branches\_-  
     catcode\_auxii:N . . . . . 8935  
   \\_\_peek\_execute\_branches\_-  
     catcode\_auxiii: . . . . . 8935  
   \\_\_peek\_execute\_branches\_-  
     charcode: . . . . . 519, 8935, 27951  
   \\_\_peek\_execute\_branches\_-  
     meaning: . . . . . 519, 8927, 27953  
   \\_\_peek\_execute\_branches\_N\_type:  
     . . . . . 27906  
   \\_\_peek\_false:w . . . . .  
     . . . . . 1072, 1073, 8868, 8883, 8894, 8908,  
       8932, 8955, 8965, 27923, 27936, 27960  
   \\_\_peek\_false\_aux:n 1073, 27961, 27962  
   \\_\_peek\_get\_prefix\_arg\_replacement:wN  
     . . . . . 9005  
   \\_\_peek\_N\_type:w . . . . . 27906  
   \\_\_peek\_N\_type\_aux:nnw . . . . . 27906  
   \\_\_peek\_remove\_spaces: . . . . . 8881  
   \l\_\_peek\_search\_tl . . . . . 516, 518,  
     1073, 8867, 8901, 8952, 8962, 27958  
   \l\_\_peek\_search\_token . . . . .  
     . . . . . 516, 1073, 8866, 8900, 8929, 27957  
   \\_\_peek\_tmp:w 8868, 8879, 27907, 27929  
   \\_\_peek\_token\_generic:NNTF . . . . .  
     . . . . . 519, 1072,  
       8915, 8917, 8919, 27940, 27944, 27946  
   \\_\_peek\_token\_generic\_aux:NNNTF .  
     . . . . . 8897, 8916, 8922  
   \\_\_peek\_token\_remove\_generic:NNTF .  
     . . . . . 519, 8915, 8923, 8925  
   \\_\_peek\_true:w . . . . .  
     . . . . . 1072, 1073, 8868, 8907, 8930,  
       8953, 8963, 27921, 27935, 27936, 27967  
   \\_\_peek\_true\_aux:w . . . . .  
     . . . . . 516, 517, 8868, 8878, 8885,  
       8886, 8902, 8916, 27968, 27969, 27987  
   \\_\_peek\_true\_remove:w . . . . .  
     . . . . . 516, 517, 8876, 8891, 8922, 27992  
   \penalty . . . . . 516

\pi ..... 14206, 14207  
 pi ..... 201  
 \pm ..... 15635, 15636  
 \postbreakpenalty ..... 1237, 2019  
 \postdisplaypenalty ..... 517  
 \postexhyphenchar ..... 988, 1788  
 \posthyphenchar ..... 989, 1789  
 \prebinoppenalty ..... 990, 1790  
 \prebreakpenalty ..... 1238, 2020  
 \predisplaydirection ..... 659, 1473  
 \predisplaygapfactor ..... 991, 1791  
 \predisplaypenalty ..... 518  
 \predisplaysize ..... 519  
 \preehyphenchar ..... 992, 1793  
 \prehyphenchar ..... 993, 1794  
 \prerelpenalty ..... 994, 1795  
 \pretolerance ..... 520  
 \prevdepth ..... 521  
 \prevgraf ..... 522  
 prg commands:  
 \prg\_break: .....  
     103, 437, 438, 530, 935, 1032, 3147,  
     4915, 6184, 6217, 6823, 7671, 7678,  
     9287, 9308, 11173, 13301, 13310,  
     15306, 15326, 15327, 15531, 15532,  
     15545, 15645, 15646, 15647, 18967,  
     19025, 19249, 19751, 19826, 20162,  
     20236, 20266, 20267, 20268, 20269,  
     20270, 20271, 20639, 20643, 22600,  
     22627, 26254, 26287, 26293, 26420  
 \prg\_break:n ..... 103,  
     103, 3147, 4917, 6058, 6197, 6833,  
     7671, 7680, 9192, 13086, 13317, 22138  
 \prg\_break\_point: ... 103, 103, 579,  
     841, 842, 849, 1033, 3147, 4905,  
     6055, 6185, 6217, 6283, 6290, 6828,  
     7671, 7673, 7674, 9187, 9272, 9308,  
     11150, 13080, 13302, 13311, 15307,  
     15328, 15533, 15649, 18968, 19025,  
     19257, 19579, 19620, 19744, 19751,  
     20085, 20237, 20273, 20617, 22139,  
     22467, 22621, 26254, 26288, 26420  
 \prg\_break\_point:Nn ..... 65, 103,  
     103, 348, 438, 455, 480, 592, 3138,  
     4546, 4564, 4574, 5296, 5322, 5342,  
     6218, 6253, 6264, 6875, 7671, 7671,  
     7672, 7677, 8113, 8127, 8147, 8165,  
     9309, 9325, 10530, 10549, 11684,  
     15718, 20214, 26255, 26400, 26409  
 \prg\_do\_nothing: .....  
     .. 8, 103, 380, 429, 484, 499, 655,  
     822, 890, 938, 3136, 3147, 3499,  
     3841, 3868, 4256, 5112, 5114, 5906,  
     5913, 6150, 6152, 7759, 7765, 7773,  
     7927, 8126, 8134, 8283, 8287, 8294,  
     10576, 10870, 11125, 12058, 13541,  
     13608, 13642, 13668, 13676, 15191,  
     18949, 19666, 19824, 19825, 20056,  
     20105, 20938, 20981, 20982, 20989,  
     20990, 22695, 22858, 26580, 27074  
 \prg\_generate\_conditional\_-  
     variant:Nnn .....  
         .. 97, 3981, 4404, 4414, 4425,  
         4448, 4468, 4478, 4538, 4803, 5200,  
         5213, 5221, 5252, 5781, 5803, 6038,  
         6059, 6153, 6155, 6169, 6171, 6173,  
         6175, 7426, 7945, 7959, 7960, 8103,  
         8105, 9216, 9217, 9265, 9289, 9300,  
         10423, 23707, 23709, 23713, 24376  
 \prg\_map\_break:Nn .....  
     .. 103, 103, 348, 392, 480,  
     496, 531, 3138, 4587, 4589, 5355,  
     5357, 6208, 6210, 7671, 7675, 7677,  
     8182, 8184, 9334, 9336, 10513, 10515  
 \prg\_new\_conditional:Nnn .....  
     .. 95, 2533, 7381  
 \prg\_new\_conditional:Npnn 95, 95,  
     97, 313, 314, 327, 509, 519, 2516,  
     3077, 3726, 4396, 4406, 4416, 4432,  
     4440, 4482, 4498, 4788, 4805, 4824,  
     4859, 4876, 4887, 5193, 5202, 5207,  
     5765, 5773, 5783, 5793, 6030, 6619,  
     6678, 6718, 6728, 7346, 7352, 7381,  
     7418, 7450, 7510, 7525, 7536, 7551,  
     7561, 7657, 7659, 7661, 7663, 7775,  
     8056, 8628, 8633, 8638, 8643, 8650,  
     8656, 8662, 8667, 8672, 8677, 8682,  
     8687, 8692, 8697, 8704, 8719, 8724,  
     8759, 8805, 9260, 9267, 9359, 10485,  
     11485, 11490, 11811, 11826, 12861,  
     12868, 13485, 14647, 15443, 15459,  
     20731, 20751, 20773, 20819, 20843,  
     23703, 23705, 23711, 24366, 26654  
 \prg\_new\_eq\_conditional:NNn ..  
     .. 96, 2649, 4126,  
     4127, 5163, 5165, 5167, 5169, 5935,  
     5937, 6355, 6356, 6357, 6358, 6359,  
     6360, 6549, 6551, 7381, 7446, 7448,  
     7863, 7865, 8052, 8054, 9256, 9258,  
     11382, 11384, 11771, 11773, 11907,  
     11909, 15441, 15442, 23651, 23653  
 \prg\_new\_protected\_conditional:Nnn ..  
     .. 95, 2533, 7381  
 \prg\_new\_protected\_conditional:Npnn ..  
     .. 95, 2516, 4450,  
     4470, 5215, 5223, 6040, 6149, 6151,  
     6157, 6160, 6163, 6166, 7381, 7936,  
     7946, 7948, 8070, 8074, 9196, 9206,

9291, 10413, 11104, 11175, 21165,  
 22999, 23004, 23017, 23019, 26558  
`\prg_replicate:nn` .....  
     ..... 102, 478, 631, 7609, 9465,  
 9661, 9683, 10203, 10702, 13037,  
 13158, 16863, 17714, 17952, 18208,  
 18254, 18291, 18814, 18822, 19287,  
 19387, 20341, 20944, 21680, 22065,  
 22091, 22244, 22252, 22700, 23162,  
 23167, 23174, 23277, 23282, 29418  
`\prg_return_false:` ... 96, 97, 330,  
 398, 433, 449, 450, 493, 493, 949,  
 2512, 2576, 2584, 2743, 2748, 2761,  
 2766, 2774, 2791, 3080, 3736, 4401,  
 4411, 4422, 4437, 4445, 4460, 4475,  
 4489, 4505, 4800, 4821, 4837, 4845,  
 4855, 4868, 4882, 4896, 5198, 5205,  
 5211, 5219, 5227, 5770, 5778, 5789,  
 5799, 6035, 6054, 6073, 6617, 6649,  
 6654, 6683, 6723, 6733, 7349, 7357,  
 7381, 7423, 7455, 7515, 7531, 7541,  
 7557, 7567, 7658, 7660, 7662, 7664,  
 7790, 7793, 7939, 7953, 8059, 8094,  
 8100, 8631, 8636, 8641, 8646, 8653,  
 8660, 8665, 8670, 8675, 8680, 8685,  
 8690, 8695, 8700, 8717, 8722, 8727,  
 8732, 8765, 8768, 8780, 8809, 8834,  
 8851, 8860, 9204, 9214, 9263, 9283,  
 9298, 9362, 10417, 10494, 11108,  
 11179, 11488, 11507, 11522, 11523,  
 11815, 11829, 12866, 12874, 13496,  
 13498, 14662, 14674, 15454, 15467,  
 20745, 20756, 20759, 20764, 20768,  
 20769, 20777, 20780, 20785, 20788,  
 20825, 20828, 20849, 20852, 21172,  
 21177, 23045, 23704, 23706, 23712,  
 24372, 24374, 26562, 26570, 26665  
`\prg_return_true:` 96, 97, 330, 387,  
 399, 399, 433, 526, 578, 947, 949,  
 2512, 2576, 2584, 2746, 2763, 2771,  
 2776, 2789, 2794, 3080, 3728, 3736,  
 4399, 4409, 4420, 4435, 4443, 4458,  
 4475, 4488, 4503, 4798, 4819, 4835,  
 4853, 4866, 4884, 4895, 5198, 5205,  
 5211, 5219, 5227, 5768, 5776, 5787,  
 5797, 6033, 6058, 6076, 6649, 6681,  
 6721, 6731, 7349, 7355, 7381, 7421,  
 7453, 7513, 7529, 7539, 7555, 7565,  
 7658, 7660, 7662, 7664, 7786, 7789,  
 7795, 7942, 7956, 8060, 8090, 8100,  
 8631, 8636, 8641, 8646, 8653, 8660,  
 8665, 8670, 8675, 8680, 8685, 8690,  
 8695, 8700, 8716, 8722, 8730, 8779,  
 8832, 8858, 9202, 9212, 9263, 9285,  
 9296, 9362, 10420, 10492, 10497,  
 10499, 11113, 11180, 11488, 11523,  
 11814, 11830, 12865, 12873, 13489,  
 13494, 14657, 14680, 15456, 15465,  
 20734, 20748, 20756, 20759, 20764,  
 20768, 20780, 20785, 20788, 20823,  
 20847, 21168, 21174, 23043, 23704,  
 23706, 23712, 24371, 26583, 26663  
`\prg_set_conditional:Nnn` .....  
     ..... 95, 2533, 7381  
`\prg_set_conditional:Npnn` 95, 96,  
 97, 2516, 2740, 2752, 2768, 2780, 7381  
`\prg_set_eq_conditional>NNn` ....  
     ..... 96, 2649, 7381  
`\prg_set_protected_conditional:Nnn`  
     ..... 95, 2533, 7381  
`\prg_set_protected_conditional:Npnn`  
     ..... 95, 2516, 7381  
 prg internal commands:  
`\__prg_break:` ..... 7671  
`\__prg_break:n` ..... 7671  
`\__prg_break_point:` ..... 7671  
`\__prg_break_point:Nn` 348, 480, 7671  
`\__prg_generate_conditional:nnNNnnn`  
     ..... 2528, 2553, 2562  
`\__prg_generate_conditional:NNnnnnNw`  
     ..... 2562  
`\__prg_generate_conditional_-`  
     `count:NNNnm` ..... 2533  
`\__prg_generate_conditional_-`  
     `count:nnNNNnn` ..... 2533  
`\__prg_generate_conditional_-`  
     `fast:nw` ..... 331, 332, 2562  
`\__prg_generate_conditional_-`  
     `parm:NNNpn` ..... 2516  
`\__prg_generate_conditional_-`  
     `test:w` ..... 2562  
`\__prg_generate_F_form:wNNnnnnN` 2605  
`\__prg_generate_p_form:wNNnnnnN` .  
     ..... 331, 2605  
`\__prg_generate_T_form:wNNnnnnN` 2605  
`\__prg_generate_TF_form:wNNnnnnN`  
     ..... 2605  
`\__prg_map_break:Nn` ..... 480, 7671  
`\__prg_p_true:w` ..... 332, 2605  
`\__prg_replicate:N` ..... 7609  
`\__prg_replicate_` ..... 7609  
`\__prg_replicate_0:n` ..... 7609  
`\__prg_replicate_1:n` ..... 7609  
`\__prg_replicate_2:n` ..... 7609  
`\__prg_replicate_3:n` ..... 7609  
`\__prg_replicate_4:n` ..... 7609  
`\__prg_replicate_5:n` ..... 7609  
`\__prg_replicate_6:n` ..... 7609

\\_\_prg\_replicate\_7:n . . . . . 7609  
 \\_\_prg\_replicate\_8:n . . . . . 7609  
 \\_\_prg\_replicate\_9:n . . . . . 7609  
 \\_\_prg\_replicate\_first:N . . . . . 7609  
 \\_\_prg\_replicate\_first\_-:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_0:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_1:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_2:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_3:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_4:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_5:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_6:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_7:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_8:n . . . . . 7609  
 \\_\_prg\_replicate\_first\_9:n . . . . . 7609  
 \\_\_prg\_set\_eq\_conditional:NNNn 2649  
 \\_\_prg\_set\_eq\_conditional:nnNnnNNw  
     . . . . . 2657, 2665  
 \\_\_prg\_set\_eq\_conditional\_F\_-  
     form:nnn . . . . . 2665  
 \\_\_prg\_set\_eq\_conditional\_F\_-  
     form:wNnnnn . . . . . 2710  
 \\_\_prg\_set\_eq\_conditional\_-  
     loop:nnnnNw . . . . . 2665  
 \\_\_prg\_set\_eq\_conditional\_p\_-  
     form:nnn . . . . . 2665  
 \\_\_prg\_set\_eq\_conditional\_p\_-  
     form:wNnnnn . . . . . 2698  
 \\_\_prg\_set\_eq\_conditional\_T\_-  
     form:nnn . . . . . 2665  
 \\_\_prg\_set\_eq\_conditional\_T\_-  
     form:wNnnnn . . . . . 2706  
 \\_\_prg\_set\_eq\_conditional\_TF\_-  
     form:nnn . . . . . 2665  
 \\_\_prg\_set\_eq\_conditional\_TF\_-  
     form:wNnnnn . . . . . 2702  
\primitive . . . . . 886, 1619, 3592  
prop commands:  
 \c\_empty\_prop . . . . .  
     137, 522, 9043, 9047, 9051, 9054, 9262  
 \prop\_clear:N . . . . .  
     .. 132, 132, 9050, 9057, 22423, 24886  
 \prop\_clear\_new:N . . . . . 132, 9056  
 \prop\_const\_from\_keyval:Nn . . . . .  
     . . . . . 249, 9074, 24335, 24342  
 \prop\_count:N . . . . . 249, 26240, 26269  
 \prop\_gclear:N . . . . . 132, 9050, 9060  
 \prop\_gclear\_new:N . . . . .  
     . . . . . 132, 984, 9056, 24409, 24410  
 \prop\_get:Nn . . . . . 103, 29748, 29750  
 \prop\_get:NnN . . . . . 63, 64, 133,  
     134, 9153, 25130, 25134, 25213, 25217  
 \prop\_get:NnNTF . . . . . 133,  
     134, 135, 9291, 9770, 9790, 9845, 24596  
 \prop\_gpop:NnN . . . . . 133, 9161  
 \prop\_gpop:NnNTF . . . . . 133, 135, 9196  
 \prop\_gput:Nnn . . . . .  
     .. 133, 9218, 9548, 9550, 10402,  
     10457, 10595, 10628, 24622, 24640,  
     24675, 24706, 28498, 28726, 28934  
 \prop\_gput\_if\_new:Nnn . . . . . 133, 9239  
 \prop\_gremove:Nn . . . . .  
     . . . . . 134, 9137, 10465, 10637  
 \prop\_gset\_eq:NN . . . . . 132, 9054,  
     9062, 24411, 24413, 24574, 24576,  
     24613, 24615, 25811, 25979, 26020  
 \prop\_gset\_from\_keyval:Nn . . . . . 249, 9074  
 \prop\_if\_empty:NTF . . . . . 134, 9260, 26266  
 \prop\_if\_empty\_p:N . . . . . 134, 9260  
 \prop\_if\_exist:NTF . . . . .  
     . . . . . 134, 9057, 9060, 9256  
 \prop\_if\_exist\_p:N . . . . . 134, 9256  
 \prop\_if\_in:NnTF . . . . . 134, 9267, 9553, 9564  
 \prop\_if\_in\_p:Nn . . . . . 134, 9267  
 \prop\_item:Nn . . . . .  
     134, 249, 9183, 9554, 9565, 28509,  
     28519, 28748, 28941, 29749, 29751  
 \prop\_log:N . . . . . 137, 9337  
 \prop\_map\_break: . . . . .  
     .. 136, 1032, 9309, 9325, 9333, 26255  
 \prop\_map\_break:n . . . . . 136, 9333  
 \prop\_map\_function:NN . . . . .  
     . . . . . 135, 248, 249, 529, 1032,  
     9302, 9347, 10479, 10651, 25288, 26245  
 \prop\_map\_inline:Nn . . . . .  
     . . . . . 135, 9318, 23184,  
     24973, 24992, 25182, 25191, 25820,  
     25822, 25825, 25845, 25847, 25922,  
     25939, 26000, 26002, 26006, 26008  
 \prop\_map\_tokens:Nn . . . . . 249, 26250  
 \prop\_new:N . . . . . 132, 132, 9044, 9057,  
     9060, 9070, 9071, 9072, 9073, 9547,  
     9549, 9576, 9742, 10389, 10582,  
     22329, 22330, 25056, 25097, 25800,  
     25801, 25802, 28491, 28716, 28927  
 \prop\_pop:NnN . . . . . 133, 9161  
 \prop\_pop:NnNTF . . . . . 133, 135, 9196  
 \prop\_put:Nnn . . . . .  
     .. 133, 368, 521, 522, 9218, 9818,  
     9834, 9851, 22587, 24619, 24637,  
     24656, 24673, 24704, 24997, 25057,  
     25059, 25061, 25063, 25065, 25067,  
     25069, 25071, 25073, 25075, 25077,  
     25079, 25081, 25083, 25085, 25087,  
     25089, 25091, 25857, 25859, 25865,  
     25867, 25876, 25882, 25890, 25949,  
     25957, 26047, 26053, 26061, 26068  
 \prop\_put\_if\_new:Nnn . . . . . 133, 9239

\prop_rand_key_value:N . . . . .	249, 26264	\ptex_autoxspacing:D . . . . .	1994
\prop_remove:Nn . . . . .	134, 9137, 9815, 9830, 25177, 25180, 25184	\ptex_dtou:D . . . . .	1995
\prop_set_eq:NN . . . . .	132, 9051, 9062, 22598, 24562, 24564, 24606, 24608, 24930, 25172, 25809, 25816, 25818, 25972, 25996, 25998, 26017	\ptex_epTeXversion:D . . . . .	1997
\prop_set_from_keyval:Nn . . . . .	249, 9074	\ptex_euc:D . . . . .	1998
\prop_show:N . . . . .	136, 9337	\ptex_ifdbox:D . . . . .	1999
\g_tmpa_prop . . . . .	137, 9070	\ptex_ifddir:D . . . . .	2000
\l_tmpa_prop . . . . .	137, 9070	\ptex_ifmdir:D . . . . .	2001
\g_tmpb_prop . . . . .	137, 9070	\ptex_iftbox:D . . . . .	2002
\l_tmpb_prop . . . . .	137, 9070	\ptex_iftdir:D . . . . .	2003
prop internal commands:		\ptex_ifybox:D . . . . .	2004
\__prop_count:nn . . . . .	26240	\ptex_ifydir:D . . . . .	2005
\__prop_from_keyval:n . . . . .	9074	\ptex_inhibitglue:D . . . . .	2006
\__prop_from_keyval_key:n . . . . .	9074	\ptex_inhibitxspcode:D . . . . .	2007
\__prop_from_keyval_key:w . . . . .	523, 9074	\ptex_inputencoding:D . . . . .	1996
\__prop_from_keyval_loop:w . . . . .	9074	\ptex_jcharwidowpenalty:D . . . . .	2008
\__prop_from_keyval_split:Nw . . . . .	9074	\ptex_jfam:D . . . . .	2009
\__prop_from_keyval_value:n . . . . .	9074	\ptex_jfont:D . . . . .	2010
\__prop_from_keyval_value:w . . . . .	523, 9074	\ptex_jis:D . . . . .	2011
\__prop_if_in:N . . . . .	529, 9267	\ptex_kanjiskip:D . . . . .	2012
\__prop_if_in:nwwn . . . . .	529, 9267	\ptex_kansuji:D . . . . .	2013
\l__prop_internal_tl . . . . .	527, 9039, 9042, 9222, 9228, 9229, 9245, 9252	\ptex_kansujichar:D . . . . .	2014
\__prop_item_Nn:nwn . . . . .	526	\ptex_kcatcode:D . . . . .	2015
\__prop_item_Nn:nwwn . . . . .	9183	\ptex_kuten:D . . . . .	2016
\__prop_map_function:Nwwn . . . . .	9302	\ptex_noautospacing:D . . . . .	2017
\__prop_map_tokens:nwwn . . . . .	26250	\ptex_noautoxspacing:D . . . . .	2018
\__prop_pair:wn . . . . .	521, 521, 521, 524, 529, 530, 530, 1032, 9039, 9040, 9116, 9131, 9134, 9186, 9189, 9224, 9247, 9270, 9274, 9308, 9311, 9321, 9323, 9328, 26254, 26257, 26274	\ptex_postbreakpenalty:D . . . . .	2019
\__prop_put>NNnn . . . . .	9218	\ptex_pbreackpenalty:D . . . . .	2020
\__prop_put_if_new>NNnn . . . . .	9239	\ptex_ptexminorversion:D . . . . .	2021
\__prop_rand_item:w . . . . .	26264	\ptex_ptexrevision:D . . . . .	2022
\__prop_show>NN . . . . .	9337, 9339, 9341	\ptex_ptexversion:D . . . . .	2023
\__prop_split:NnTF . . . . .	521, 527, 528, 529, 9126, 9139, 9145, 9155, 9163, 9172, 9198, 9208, 9227, 9250, 9293	\ptex_showmode:D . . . . .	2024
\__prop_split_aux:NnTF . . . . .	9126	\ptex_sjis:D . . . . .	2025
\__prop_split_aux:w . . . . .	524, 9126	\ptex_tate:D . . . . .	2026
\protect . . . . .	10769, 14140, 27116, 27143	\ptex_tbaselineoffset:D . . . . .	2027
\protected . . . . .	213, 215, 217, 242, 660, 1474, 8793, 8795	\ptex_tfnt:D . . . . .	2028
\protrudechars . . . . .	1032, 1642	\ptex_xkanjiskip:D . . . . .	2029
\ProvidesExplClass . . . . .	6	\ptex_xspcode:D . . . . .	2030
\ProvidesExplFile . . . . .	6, 27998	\ptex_ybaselineoffset:D . . . . .	2031
\ProvidesExplPackage . . . . .	6	\ptex_yoko:D . . . . .	2032
pt . . . . .	201	\ptexminorversion . . . . .	1239, 2021
ptex commands:		\ptexrevision . . . . .	1240, 2022
\ptex_autospacing:D . . . . .	1993	\ptexversion . . . . .	1241, 2023
		\pxdimen . . . . .	1033, 1643

**Q**

quark commands:

\q_mark . . . . .	64, 107, 336, 367, 369, 383, 390, 393, 394, 407, 413, 420, 484, 488, 491, 492, 497, 523, 524, 603, 844, 848, 849, 850, 853, 2574, 2576, 2584, 2730, 2731, 2734, 2735, 2736, 3780, 3781, 3783, 3789, 3793, 3815, 3824, 3843, 3871, 3874, 3883, 3898, 3942, 3956, 3959, 3971, 3998, 4001,
-------------------	--

```

4017, 4316, 4318, 4320, 4322, 4530,
4540, 4618, 4619, 4622, 4625, 4626,
4632, 4635, 4650, 4651, 4657, 4661,
4663, 4666, 5250, 5281, 5288, 5361,
5378, 5626, 5628, 5715, 6304, 6305,
6319, 6322, 6632, 6635, 6707, 6714,
7780, 7797, 7919, 7929, 7933, 7955,
8011, 8017, 8031, 8043, 8044, 8045,
8048, 8049, 8050, 8059, 8060, 8069,
8223, 8224, 8236, 8237, 9086, 9094,
9106, 9131, 9133, 9134, 9710, 9775,
9776, 9781, 9784, 10912, 10919,
10931, 11005, 11006, 11007, 11008,
11551, 11558, 11989, 11996, 12006,
12030, 12052, 12062, 12074, 13330,
13331, 13335, 19669, 19673, 19679,
19683, 19689, 19692, 19757, 19797,
19799, 19802, 19806, 19809, 19812,
19814, 19817, 27918, 27919, 27926
\q_nil ..... 18, 19, 64,
64, 64, 319, 384, 386, 387, 394, 423,
425, 523, 2162, 2165, 4004, 4342,
4418, 4419, 4429, 4430, 4649, 4653,
4671, 4674, 4677, 4767, 4768, 5715,
5767, 5786, 5792, 5807, 5808, 9099,
9103, 9110, 10798, 10805, 12006,
12016, 12030, 12040, 21643, 21661
\q_no_value ..... 63,
64, 64, 64, 69, 70, 70, 70, 70, 70,
75, 75, 75, 114, 133, 133, 133, 423,
424, 434, 435, 488, 525, 525, 5715,
5775, 5796, 5802, 6064, 6072, 6084,
6110, 7902, 7917, 9157, 9168, 9177
\quark_if_nil:n ..... 425
\quark_if_nil:NTF ..... 64, 5765, 21665, 21685
\quark_if_nil:nTF . 64, 423, 4340, 5783
\quark_if_nil_p:N ..... 64, 5765
\quark_if_nil_p:n ..... 64, 5783
\quark_if_no_value:NTF ..... 64, 5765, 25132, 25136, 25215, 25219
\quark_if_no_value:nTF .... 64, 5783
\quark_if_no_value_p:N ..... 64, 5765
\quark_if_no_value_p:n ..... 64, 5783
\quark_if_recursion_tail_break:N
..... 29752
\quark_if_recursion_tail_break:n
..... 29754
\quark_if_recursion_tail_-
break:NN 65, 4581, 5330, 5348, 5753
\quark_if_recursion_tail_-
break:nN ..... 65, 4552, 4915, 5753, 8118, 8131
\quark_if_recursion_tail_stop:N .
.. 65, 5721, 7210, 8176, 25666, 27673
\quark_if_recursion_tail_stop:n .
..... 65,
401, 424, 5735, 7770, 8206, 9091, 27501
\quark_if_recursion_tail_stop_-
do:Nn .... 65, 5665, 5721, 7149,
7166, 7213, 26929, 26937, 27104, 27124
\quark_if_recursion_tail_stop_-
do:nn .. 65, 5735, 7520, 7546, 27176
\quark_new:N ..... 63, 5709,
5715, 5716, 5717, 5718, 5719, 5720
\q_recursion_stop ..... 18, 19, 64,
65, 65, 65, 65, 66, 319, 333, 423,
496, 2164, 2168, 2580, 2662, 3767,
3997, 5642, 5650, 5655, 5719, 7144,
7161, 7204, 7231, 7509, 7535, 7766,
8164, 8200, 9087, 21643, 21661,
25674, 26869, 26872, 26881, 26892,
26906, 26920, 26927, 26933, 26952,
26954, 26963, 26965, 26972, 26973,
26976, 26979, 26992, 27098, 27120,
27142, 27172, 27188, 27190, 27193,
27196, 27198, 27203, 27206, 27208,
27211, 27214, 27219, 27242, 27245,
27247, 27249, 27254, 27335, 27339,
27341, 27346, 27360, 27385, 27389,
27391, 27396, 27404, 27645, 27693,
27710, 27714, 27716, 27721, 27736
\q_recursion_tail ..... 64, 65,
65, 65, 65, 65, 66, 333, 412, 423,
423, 494, 529, 2580, 2590, 2662,
2681, 4545, 4563, 4573, 4904, 5295,
5304, 5321, 5341, 5642, 5719, 5723,
5729, 5738, 5745, 5750, 5755, 5762,
7144, 7161, 7204, 7509, 7535, 7766,
8112, 8126, 8146, 8164, 8200, 9087,
9271, 9282, 11989, 11996, 12057,
25674, 26869, 26933, 26967, 27098,
27120, 27151, 27172, 27644, 27692
\q_stop ..... 18,
19, 30, 45, 63, 63, 64, 64, 107,
319, 336, 369, 384, 393, 413, 417,
423, 450, 461, 462, 492, 497, 512,
514, 523, 524, 553, 641, 848, 849,
850, 2163, 2166, 2322, 2324, 2601,
2606, 2625, 2633, 2641, 2692, 2698,
2702, 2706, 2710, 2731, 2734, 2735,
2736, 3784, 3793, 3819, 3871, 3875,
3879, 3887, 3893, 3902, 3908, 3910,
3942, 3962, 3971, 3998, 4341, 4502,
4508, 4530, 4540, 4620, 4622, 4627,
4629, 4655, 4677, 4758, 4760, 4777,
4795, 4812, 4834, 5040, 5050, 5147,

```

5250, 5281, 5288, 5361, 5370, 5376,  
 5378, 5384, 5401, 5420, 5482, 5539,  
 5551, 5589, 5605, 5612, 5620, 5622,  
 5626, 5628, 5715, 6084, 6087, 6095,  
 6097, 6178, 6179, 6306, 6319, 6322,  
 6324, 6611, 6627, 6629, 6633, 6646,  
 6707, 6714, 7137, 7143, 7160, 7904,  
 7907, 7919, 7922, 7930, 7933, 7941,  
 7955, 8017, 8045, 8048, 8049, 8061,  
 8069, 8225, 8236, 8237, 8238, 8264,  
 8298, 8709, 8712, 8742, 8776, 8813,  
 8817, 8823, 8846, 9007, 9014, 9023,  
 9032, 9093, 9103, 9106, 9110, 9131,  
 9134, 9712, 9716, 9718, 9777, 10798,  
 10839, 10897, 10935, 10948, 11005,  
 11008, 11035, 11040, 11244, 11247,  
 11261, 11276, 11278, 11498, 11522,  
 11551, 11558, 11830, 11832, 12060,  
 12077, 12104, 12123, 12124, 12191,  
 12193, 12213, 12217, 12226, 12234,  
 12245, 12689, 12697, 12707, 12849,  
 12851, 12856, 13307, 13318, 13325,  
 13331, 13335, 13349, 13368, 14180,  
 14184, 14691, 14696, 15267, 15289,  
 15452, 15453, 15478, 15479, 15645,  
 15646, 15647, 15814, 15815, 19745,  
 19754, 19758, 19760, 19797, 19798,  
 19799, 19804, 19806, 19810, 19812,  
 19820, 25329, 25331, 25332, 25333,  
 25335, 25337, 25339, 25570, 25587,  
 25591, 25602, 25606, 25618, 25619,  
 25625, 25627, 25628, 25630, 25633,  
 25649, 25657, 26201, 26270, 26283,  
 26284, 26286, 26290, 26294, 26296,  
 26301, 27907, 27920, 27929, 28043,  
 28046, 28092, 28095, 29387, 29391  
**\s\_stop** 66, 67, 849, 5825, 5826, 17433,  
 17448, 18698, 18702, 19758, 19760  
 quark internal commands:  
**\s\_fp** . . . . . 635, 636, 637,  
 641, 642, 665, 672, 673, 676, 689,  
 691, 693, 720, 723, 725, 725, 727,  
 733, 736, 822, 13190, 13200, 13201,  
 13202, 13203, 13204, 13214, 13219,  
 13221, 13222, 13243, 13246, 13248,  
 13258, 13270, 13290, 13307, 13310,  
 13317, 13324, 13340, 13367, 13473,  
 13475, 13477, 13478, 13479, 13481,  
 13482, 13483, 13485, 13501, 13686,  
 13691, 13918, 13964, 13973, 13975,  
 14652, 14810, 15267, 15282, 15306,  
 15326, 15327, 15453, 15478, 15479,  
 15493, 15494, 15531, 15532, 15645,  
 15646, 15647, 15656, 15672, 15676,  
 15740, 15741, 15744, 15755, 15756,  
 15764, 15765, 15767, 15768, 15769,  
 15771, 15772, 15773, 15785, 15788,  
 15792, 15795, 15815, 15865, 15868,  
 15871, 15891, 15892, 15894, 15895,  
 15896, 15904, 15907, 15918, 15919,  
 15921, 15930, 16006, 16158, 16192,  
 16193, 16196, 16277, 16415, 16423,  
 16425, 16602, 16611, 16613, 16618,  
 16626, 16628, 16630, 16633, 17134,  
 17146, 17148, 17357, 17374, 17376,  
 17557, 17576, 17578, 17579, 17582,  
 17599, 17602, 17605, 17630, 17631,  
 17633, 17649, 17738, 17751, 17753,  
 17756, 17761, 17823, 17836, 17838,  
 17851, 17853, 17866, 17868, 17881,  
 17883, 17896, 17898, 17911, 17921,  
 18422, 18438, 18439, 18443, 18454,  
 18561, 18574, 18576, 18592, 18595,  
 18605, 18628, 18639, 18641, 18655,  
 18657, 18662, 18724, 18745, 18748,  
 18778, 18799, 18802, 18852, 18868,  
 18871, 18947, 18948, 19063, 19065,  
 19097, 19360, 19368, 19371, 19447  
**\s\_fp\_<type>** . . . . . 665  
**\s\_fp\_division** . . . . . 13195  
**\s\_fp\_exact** . . . . . 13195, 13200,  
 13201, 13202, 13203, 13204, 15740  
**\s\_fp\_invalid** . . . . . 13195  
**\s\_fp\_mark** . . . . . 651, 672,  
 676, 697, 700, 13193, 13542, 13543,  
 13545, 13549, 14859, 14872, 14954  
**\s\_fp\_overflow** . . . . . 13195, 13221  
**\s\_fp\_stop** . . . . . 643,  
 13193, 13381, 14761, 14860, 14864,  
 14873, 15822, 15833, 15843, 15851  
**\s\_fp\_tuple** . . . . . 641,  
 13291, 13297, 13298, 13375, 13377,  
 15049, 15259, 15274, 15299, 15301,  
 15318, 15319, 15321, 15523, 15524,  
 16664, 16665, 16671, 16672, 18674  
**\s\_fp\_underflow** . . . . . 13195, 13219  
**\s\_prop** . . . . . 521, 521, 524,  
 530, 1032, 1032, 9039, 9039, 9040,  
 9043, 9085, 9117, 9131, 9134, 9186,  
 9189, 9225, 9248, 9270, 9274, 9308,  
 9311, 9323, 26254, 26257, 26274, 26279  
**\\_\\_quark\_if\_empty\_if:n** . . . . . 5783  
**\\_\\_quark\_if\_nil:w** . . . . . 425, 5783  
**\\_\\_quark\_if\_no\_value:w** . . . . . 5783  
**\\_\\_quark\_if\_recursion\_tail:w** . . . .  
 . . . . . 423, 5735, 5762  
**\s\_seq** 426, 429, 430, 430, 435, 440,  
 1033, 5830, 5839, 5869, 5874, 5879,

5884, 5917, 5943, 5951, 5955, 6131,  
 6179, 6316, 26284, 26290, 26331, 26341  
`\s__tl` ..... 852, 853, 853, 854,  
 855, 862, 863, 863, 19882, 19883,  
 20102, 20133, 20139, 20164, 20182,  
 20187, 20201, 20213, 20236, 20239  
`\q__tl_act_mark` .....  
 ..... 395, 395, 395, 4681, 4686, 4703  
`\q__tl_act_stop` .....  
 ..... 395, 4681, 4686, 4690, 4699,  
 4701, 4707, 4712, 4715, 4719, 4722  
`\quitvmode` ..... 798, 1612

**R**

`\r` ..... 27699  
`\radical` ..... 523  
`\raise` ..... 524  
`rand` ..... 201  
`randint` ..... 201  
`\randomseed` ..... 1034, 1644  
`\read` ..... 525  
`\readline` ..... 661, 1475  
`\ref` ..... 27747  
 regex commands:  
`\c_foo_regex` ..... 208  
`\regex_(g)set:Nn` ..... 215  
`\regex_const:Nn` ..... 208, 215, 22966  
`\regex_count:NnN` ..... 216, 23009  
`\regex_count:nnN` ..... 216, 948, 23009  
`\regex_extract_all:NnN` ..... 216, 23013  
`\regex_extract_all:nnN` .....  
 ..... 209, 216, 869, 23013  
`\regex_extract_all:NnTF` ..... 216, 23013  
`\regex_extract_all:nnTF` ..... 216, 23013  
`\regex_extract_once:NnN` ..... 216, 23013  
`\regex_extract_once:nnN` .....  
 ..... 216, 216, 23013  
`\regex_extract_once:NnTF` ..... 216, 23013  
`\regex_extract_once:nnTF` .....  
 ..... 213, 216, 23013  
`\regex_gset:Nn` ..... 215, 22966  
`\regex_match:NnTF` ..... 215, 22999  
`\regex_match:nnTF` ..... 215, 22999  
`\regex_new:N` ..... 215,  
 872, 22960, 22962, 22963, 22964, 22965  
`\regex_replace_all:NnN` ..... 217, 23013  
`\regex_replace_all:nnN` ..... 209, 217, 23013  
`\regex_replace_all:NnTF` ..... 217, 23013  
`\regex_replace_all:nnTF` ..... 217, 23013  
`\regex_replace_once:NnN` ..... 217, 23013  
`\regex_replace_once:nnN` .....  
 ..... 216, 217, 23013  
`\regex_replace_once:NnTF` ..... 217, 23013  
`\regex_replace_once:nnTF` ..... 217, 23013

`\regex_set:Nn` ..... 215, 22966  
`\regex_show:N` ..... 215, 22981  
`\regex_show:n` ..... 208, 215, 22981  
`\regex_split:NnN` ..... 217, 23013  
`\regex_split:nnN` ..... 217, 23013  
`\regex_split:NnNTF` ..... 217, 23013  
`\regex_split:nnNTF` ..... 217, 23013  
`\g_tmpt_regex` ..... 217, 22962  
`\l_tmpt_regex` ..... 217, 22962  
`\g_tmppb_regex` ..... 217, 22962  
`\l_tmppb_regex` ..... 217, 22962

## regex internal commands:

`\__regex_action_cost:n` ..... 914, 918,  
 926, 22053, 22054, 22062, 22537, 22563  
`\__regex_action_free:n` .....  
 ..... 218, 914, 926, 22076, 22082,  
 22083, 22094, 22159, 22163, 22188,  
 22213, 22217, 22220, 22248, 22256,  
 22266, 22280, 22311, 22535, 22539  
`\__regex_action_free_aux:nn` ..... 22539  
`\__regex_action_free_group:n` ..... 218,  
 914, 926, 22106, 22228, 22231, 22539  
`\__regex_action_start_wildcard:` ..  
 ..... 914, 21969, 22532  
`\__regex_action_submatch:n` .....  
 ..... 914, 22182, 22183, 22309, 22583, 22585  
`\__regex_action_success:` .....  
 ..... 218, 914, 21972, 21992, 22590  
`\__regex_action_wildcard:` ..... 931  
`\c__regex_all_catcodes_int` .....  
 ..... 20803, 20925, 21015, 21615  
`\__regex_anchor:N` .....  
 ..... 883, 924, 21244, 21809, 22271  
`\c__regex_ascii_lower_int` .....  
 ..... 20390, 20448, 20454  
`\c__regex_ascii_max_control_int` .....  
 ..... 20387, 20564  
`\c__regex_ascii_max_int` .....  
 ..... 20387, 20557, 20565, 20755  
`\c__regex_ascii_min_int` .....  
 ..... 20387, 20556, 20563  
`\__regex_assertion:Nn` ..... 883,  
 924, 21244, 21269, 21278, 21803, 22271  
`\__regex_b_test:` .....  
 ..... 883, 924, 21269, 21278, 21808, 22271  
`\l__regex_balance_int` ..... 872,  
 938, 20385, 22351, 22370, 22488,  
 22492, 22493, 22681, 22710, 22903,  
 22920, 23215, 23241, 23264, 23268,  
 23269, 23276, 23277, 23281, 23282  
`\g__regex_balance_intarray` .....  
 ..... 869, 872, 20382, 22487, 22651, 22666  
`\l__regex_balance_t1` ..... 218,  
 938, 940, 22606, 22682, 22711, 22773

```

\__regex_branch:n ..... 883, 901, 920, 20379, 20930,
                     21425, 21478, 21657, 21785, 22151
\__regex_break_point:TF ..... 873, 896, 918, 20391,
                     20397, 22053, 22054, 22277, 22300
\__regex_break_true:w ..... 873, 873, 20391, 20397, 20402,
                     20409, 20416, 20422, 20429, 20437,
                     20484, 20496, 20513, 21219, 22292
\__regex_build:N ..... 948, 21950, 23006, 23012, 23016, 23020
\__regex_build:n ..... 948, 21950, 23001, 23010, 23015, 23018
\__regex_build_for_cs:n 20508, 21974
\__regex_build_new_state: ..... 21966, 21967,
                     21984, 21985, 22014, 22035, 22067,
                     22104, 22109, 22156, 22171, 22176,
                     22215, 22234, 22269, 22273, 22306
\l__regex_build_tl ..... 901,
                     20376, 20922, 20929, 20947, 20952,
                     20955, 20956, 20959, 20960, 20963,
                     21009, 21012, 21045, 21059, 21063,
                     21188, 21202, 21243, 21268, 21277,
                     21287, 21319, 21332, 21336, 21418,
                     21421, 21424, 21430, 21431, 21434,
                     21477, 21744, 21760, 21778, 21784,
                     21839, 21842, 21847, 21877, 21892,
                     21896, 21899, 21905, 22680, 22699,
                     22714, 22717, 22738, 22770, 22832,
                     22835, 22850, 22894, 22910, 22946
\__regex_build_transition_-
left:NNN 22010, 22217, 22231, 22248
\__regex_build_transition_-
right:nNn ..... 22010,
                     22068, 22106, 22159, 22163, 22188,
                     22213, 22220, 22228, 22256, 22266
\__regex_build_transitions_-
lazyness:NNNNN ..... 22033, 22075, 22081, 22093
\l__regex_capturing_group_int ...
                     870, 914, 954, 21949, 21964, 22122,
                     22124, 22135, 22136, 22144, 22145,
                     22148, 22769, 23174, 23246, 23254
\l__regex_case_changed_char_int ...
                     874,
                     20418, 20421, 20432, 20435, 20436,
                     20443, 20447, 20453, 22324, 22457
\c__regex_catcode_A_int ..... 20803
\c__regex_catcode_B_int ..... 20803
\c__regex_catcode_C_int ..... 20803
\c__regex_catcode_D_int ..... 20803
\c__regex_catcode_E_int ..... 20803
\c__regex_catcode_in_class_mode_-
int ..... 20793,
                     20914, 21286, 21447, 21540, 21569
\g__regex_catcode_intarray .....
                     869, 872, 20382, 22485, 22502
\c__regex_catcode_L_int ..... 20803
\c__regex_catcode_M_int ..... 20803
\c__regex_catcode_mode_int 20793,
                     20910, 20983, 21318, 21538, 21567
\c__regex_catcode_O_int ..... 20803
\c__regex_catcode_P_int ..... 20803
\c__regex_catcode_S_int ..... 20803
\c__regex_catcode_T_int ..... 20803
\c__regex_catcode_U_int ..... 20803
\l__regex_catcodes_bool .....
                     20800, 21574, 21578, 21613
\l__regex_catcodes_int ..... 884,
                     20800, 20926, 21014, 21016, 21022,
                     21305, 21322, 21422, 21435, 21534,
                     21571, 21606, 21608, 21614, 21615
\__regex_char_if_alphanumeric:N .
                     ..... 20773
\__regex_char_if_alphanumeric:NTF
                     ..... 20751, 20976, 22877
\__regex_char_if_special:N ... 20751
\__regex_char_if_special:NTF ...
                     ..... 20751, 20972
\g__regex_charcode_intarray .....
                     869, 872, 20382, 22483, 22499
\__regex_chk_c_allowed:TF .....
                     ..... 20895, 21527
\__regex_class:NnnnN .....
                     883, 891, 892, 898,
                     20380, 21010, 21313, 21314, 21320,
                     21673, 21752, 21762, 21800, 22047
\c__regex_class_mode_int .....
                     ..... 20793, 20900, 20915
\__regex_class_repeat:n .....
                     918, 22057, 22063, 22079, 22088
\__regex_class_repeat:nN 22058, 22072
\__regex_class_repeat:nnN .....
                     ..... 22059, 22086
\__regex_command_K: .....
                     883, 21778, 21801, 22304
\__regex_compile:n ..... 20965,
                     21952, 22968, 22973, 22978, 22983
\__regex_compile:w .....
                     890, 20919, 20967, 21620
\__regex_compile_$: .....
                     21239
\__regex_compile_(: .....
                     21442
\__regex_compile_(): .....
                     21481
\__regex_compile_.: .....
                     21210
\__regex_compile/_A: .....
                     21239
\__regex_compile/_B: .....
                     21263

```

\\_regex\_compile/\_b: ..... 21263  
\\_regex\_compile/\_c: ..... 21526  
\\_regex\_compile/\_D: ..... 21222  
\\_regex\_compile/\_d: ..... 21222  
\\_regex\_compile/\_G: ..... 21239  
\\_regex\_compile/\_H: ..... 21222  
\\_regex\_compile/\_h: ..... 21222  
\\_regex\_compile/\_K: ..... 21775  
\\_regex\_compile/\_N: ..... 21222  
\\_regex\_compile/\_S: ..... 21222  
\\_regex\_compile/\_s: ..... 21222  
\\_regex\_compile/\_u: ..... 21693  
\\_regex\_compile/\_V: ..... 21222  
\\_regex\_compile/\_v: ..... 21222  
\\_regex\_compile/\_W: ..... 21222  
\\_regex\_compile/\_w: ..... 21222  
\\_regex\_compile/\_Z: ..... 21239  
\\_regex\_compile/\_z: ..... 21239  
\\_regex\_compile[\_: ..... 21297  
\\_regex\_compile\_]: ..... 21281  
\\_regex\_compile^: ..... 21239  
\\_regex\_compile\_abort\_tokens:n .  
..... 21025, 21052, 21402, 21412  
\\_regex\_compile\_anchor:NTF .. 21239  
\\_regex\_compile\_c[:w ..... 21563  
\\_regex\_compile\_c:C>NN 21542, 21551  
\\_regex\_compile\_c\_lbrack\_add:N .  
..... 21563  
\\_regex\_compile\_c\_lbrack\_end: 21563  
\\_regex\_compile\_c\_lbrack\_-  
loop:NN ..... 21563  
\\_regex\_compile\_c\_test:NN ... 21526  
\\_regex\_compile\_class:NN .... 21327  
\\_regex\_compile\_class:TFNN ....  
..... 898, 21312, 21323, 21327  
\\_regex\_compile\_class\_catcode:w  
..... 21304, 21316  
\\_regex\_compile\_class\_normal:w .  
..... 21307, 21310  
\\_regex\_compile\_class\_posix:NNNNw  
..... 21346  
\\_regex\_compile\_class\_posix\_-  
end:w ..... 21346  
\\_regex\_compile\_class\_posix\_-  
loop:w ..... 21346  
\\_regex\_compile\_class\_posix\_-  
test:w ..... 21300, 21346  
\\_regex\_compile\_cs\_aux:Nn ... 21629  
\\_regex\_compile\_cs\_aux:NNnnnN 21629  
\\_regex\_compile\_end: .....  
..... 890, 20919, 20992, 21638  
\\_regex\_compile\_end\_cs: 20988, 21629  
\\_regex\_compile\_escaped:N ....  
..... 20977, 20994  
\\_regex\_compile\_group\_begin:N ..  
.. 21416, 21464, 21469, 21487, 21489  
\\_regex\_compile\_group\_end: ....  
..... 21416, 21484  
\\_regex\_compile\_lparen:w .....  
..... 21451, 21455  
\\_regex\_compile\_one:n .....  
.... 21004, 21154, 21160, 21214,  
21225, 21228, 21238, 21393, 21645  
\\_regex\_compile\_quantifier:w ...  
.... 21023, 21034, 21292, 21436  
\\_regex\_compile\_quantifier\_\*:w .  
..... 21068  
\\_regex\_compile\_quantifier\_+:w .  
..... 21068  
\\_regex\_compile\_quantifier\_?:w .  
..... 21068  
\\_regex\_compile\_quantifier\_-  
abort:nNN .....  
.. 21043, 21078, 21097, 21110, 21133  
\\_regex\_compile\_quantifier\_-  
braced\_auxi:w ..... 21074  
\\_regex\_compile\_quantifier\_-  
braced\_auxii:w ..... 21074  
\\_regex\_compile\_quantifier\_-  
braced\_auxiii:w ..... 21074  
\\_regex\_compile\_quantifier\_-  
lazyness:nnNN 892, 21055, 21069,  
21071, 21073, 21086, 21106, 21128  
\\_regex\_compile\_quantifier\_-  
none: ..... 21039, 21041, 21043  
\\_regex\_compile\_range:Nw .....  
..... 21152, 21165  
\\_regex\_compile\_raw:N .....  
.... 20845, 20973, 20977, 20979,  
20997, 21002, 21030, 21145, 21147,  
21167, 21213, 21259, 21295, 21343,  
21363, 21381, 21439, 21444, 21449,  
21465, 21475, 21483, 21501, 21502,  
21503, 21509, 21520, 21521, 21522,  
21530, 21585, 21634, 21705, 21711  
\\_regex\_compile\_raw\_error:N ...  
..... 21142,  
21250, 21266, 21275, 21696, 21779  
\\_regex\_compile\_special:N ....  
.... 885, 20973, 20994,  
21036, 21057, 21084, 21089, 21104,  
21117, 21151, 21170, 21330, 21348,  
21367, 21387, 21388, 21457, 21492,  
21510, 21553, 21572, 21698, 21714  
\\_regex\_compile\_special\_group\_-  
::w ..... 21490  
\\_regex\_compile\_special\_group\_-  
::w ..... 21486

```

\__regex_compile_special_group_-  

    i:w ..... 21490  

\__regex_compile_special_group_-  

    |:w ..... 21486  

\__regex_compile_u_end: .....  

    ..... 21717, 21723, 21728  

\__regex_compile_u_in_cs: .....  

    ..... 21734, 21737  

\__regex_compile_u_in_cs_aux:n ..  

    ..... 21747, 21750  

\__regex_compile_u_loop>NN ... 21693  

\__regex_compile_u_not_cs: .....  

    ..... 21732, 21756  

\__regex_compile_|: ..... 21473  

\__regex_compute_case_changed_-  

    char: ..... 20419, 20433, 20441  

\__regex_count:nnN 23010, 23012, 23057  

\c__regex_cs_in_class_mode_int ..  

    ..... 20793, 21626  

\c__regex_cs_mode_int . 20793, 21624  

\l__regex_cs_name_tl .....  

    ..... 20386, 20505, 20511  

\l__regex_curr_catcode_int 20463,  

    20482, 20490, 20502, 22324, 22501  

\l__regex_curr_char_int .....  

    ..... 20401, 20407,  

    20408, 20415, 20427, 20428, 20443,  

    20444, 20445, 20446, 20452, 20483,  

    21218, 22298, 22324, 22456, 22498  

\__regex_curr_cs_to_str: .....  

    ..... 20362, 20493, 20505  

\l__regex_curr_pos_int .....  

    . 871, 20365, 21991, 22291, 22319,  

    22352, 22354, 22357, 22371, 22378,  

    22385, 22416, 22426, 22455, 22470,  

    22484, 22486, 22488, 22489, 22490,  

    22500, 22503, 22588, 22597, 23109  

\l__regex_curr_state_int .....  

    ..... 926, 933, 22328, 22508,  

    22514, 22515, 22517, 22522, 22525,  

    22547, 22552, 22557, 22558, 22566  

\l__regex_curr_submatches_prop ..  

    ..... 22329, 22423, 22527,  

    22559, 22560, 22578, 22587, 22599  

\l__regex_default_catcodes_int ..  

    ..... 884, 20800, 20924,  

    20926, 21022, 21322, 21422, 21435  

\__regex_disable_submatches: ...  

    .. 20507, 21621, 22580, 23051, 23060  

\l__regex_empty_success_bool ...  

    .. 22337, 22408, 22412, 22595, 23119  

\__regex_escape_`w ..... 20639  

\__regex_escape_`a:w ..... 20639  

\__regex_escape_`break:w ..... 20639  

\__regex_escape_`e:w ..... 20639  

\__regex_escape_`f:w ..... 20639  

\__regex_escape_`n:w ..... 20639  

\__regex_escape_`r:w ..... 20639  

\__regex_escape_`t:w ..... 20639  

\__regex_escape_`x:w ..... 20658  

\__regex_escape_`\:w ..... 20623  

\__regex_escape_break:w ..... 20639  

\__regex_escape_escaped:N .....  

    ..... 20609, 20633, 20636  

\__regex_escape_loop:N .....  

    ..... 879, 20616, 20623, 20658,  

    20694, 20702, 20703, 20720, 20729  

\__regex_escape_raw:N .....  

    . 880, 20610, 20636, 20647, 20649,  

    20651, 20653, 20655, 20657, 20671  

\__regex_escape_unescaped:N .....  

    ..... 20608, 20626, 20636  

\__regex_escape_use:nnnn .....  

    ..... 878, 890, 20595, 20970, 22683  

\__regex_escape_x:N 880, 20693, 20697  

\__regex_escape_x_end:w .. 880, 20658  

\__regex_escape_x_large:n .... 20658  

\__regex_escape_x_loop:N .....  

    ..... 880, 20690, 20706  

\__regex_escape_x_loop_error: .. 20706  

\__regex_escape_x_loop_error:n ..  

    ..... 20709, 20721, 20726  

\__regex_escape_x_test:N .....  

    ..... 880, 20661, 20675  

\__regex_escape_x_testii:N ... 20675  

\l__regex_every_match_tl .....  

    ..... 22336, 22430, 22434, 22443  

\__regex_extract: .... 950, 23075,  

    23081, 23093, 23170, 23214, 23237  

\__regex_extract_all:nnN 23024, 23069  

\__regex_extract_b:wn ..... 23170  

\__regex_extract_e:wn ..... 23170  

\__regex_extract_once:nnN .....  

    ..... 23022, 23069  

\__regex_extract_seq_aux:n .....  

    ..... 23135, 23153  

\__regex_extract_seq_aux:ww .. 23153  

\l__regex_fresh_thread_bool ...  

    ..... 927, 933, 22310, 22316,  

    22337, 22468, 22534, 22536, 22596  

\__regex_get_digits:NTFw .....  

    ..... 20831, 21076, 21091  

\__regex_get_digits_loop:nw ...  

    ..... 20834, 20837, 20840  

\__regex_get_digits_loop:w ... 20831  

\__regex_group:nnnN ..... 883,  

    901, 21464, 21469, 21794, 21970, 22119

```

\\_\_regex\_group\_aux:nnnnN . . . . .  
    . . . . . 920, 22098, 22121, 22129, 22132  
\\_\_regex\_group\_aux:nnnnN . . . . . 920  
\\_\_regex\_group\_end\_extract\_seq:N . . . . .  
    . . . . . 23076, 23084, 23124, 23126  
\\_\_regex\_group\_end\_replace:N . . . . .  
    . . . . . 23231, 23260, 23262  
\l\_\_regex\_group\_level\_int . . . . .  
    . . . . . 20792, 20923,  
        20941, 20943, 20945, 21423, 21429  
\\_\_regex\_group\_no\_capture:nnnN . . . . .  
    . . . . . 883, 21487, 21796, 22119  
\\_\_regex\_group\_repeat:nn 22114, 22166  
\\_\_regex\_group\_repeat:nnN . . . . .  
    . . . . . 22115, 22206  
\\_\_regex\_group\_repeat:nnnN . . . . .  
    . . . . . 22116, 22237  
\\_\_regex\_group\_repeat\_aux:n . . . . .  
    921, 922, 22173, 22186, 22224, 22241  
\\_\_regex\_group\_resetting:nnnnN . . . . .  
    . . . . . 883, 21489, 21798, 22130  
\\_\_regex\_group\_resetting\_-  
    loop:nnNn . . . . . 22130  
\\_\_regex\_group\_submatches:nNN . . . . .  
    . . . . . 22174, 22179, 22209, 22225, 22239  
\\_\_regex\_hexadecimal\_use:N . . . . . 20731  
\\_\_regex\_hexadecimal\_use:NTF . . . . .  
    . . . . . 20692, 20701, 20711, 20731  
\\_\_regex\_if\_end\_range:NN . . . . . 21165  
\\_\_regex\_if\_end\_range:NNTF . . . . . 21165  
\\_\_regex\_if\_in\_class:TF . . . . .  
    . . . . . 20855, 20934, 21007,  
        21023, 21149, 21212, 21283, 21299,  
        21444, 21475, 21483, 23337, 23350  
\\_\_regex\_if\_in\_class\_or\_catcode:TF . . . . .  
    . . . . . 20875, 21241, 21265, 21274, 21695  
\\_\_regex\_if\_in\_cs:TF . . . . .  
    . . . . . 20863, 21632, 23335, 23344  
\\_\_regex\_if\_match:nn . . . . .  
    . . . . . 23001, 23006, 23048  
\\_\_regex\_if\_raw\_digit:NN . . . . . 20843  
\\_\_regex\_if\_raw\_digit:NNTF . . . . .  
    . . . . . 20833, 20839, 20843  
\\_\_regex\_if\_two\_empty\_matches:TF . . . . .  
    . . . . . 927, 22337, 22413, 22419, 22592  
\\_\_regex\_if\_within\_catcode:TF . . . . .  
    . . . . . 20887, 21302  
\\_\_regex\_int\_eval:w . . . . .  
    . . . . . 20324, 22619, 22620, 22631, 23191  
\l\_\_regex\_internal\_a\_int . . . . .  
    . . . . . 893, 940, 20368,  
        21076, 21087, 21098, 21107, 21111,  
        21119, 21122, 21126, 21129, 21136,  
        22080, 22083, 22089, 22094, 22175,  
    22190, 22196, 22202, 22211, 22214,  
        22218, 22221, 22226, 22229, 22232,  
        22247, 22255, 22264, 22785, 22806  
\l\_\_regex\_internal\_a\_t1 . . . . .  
    . . . . . 878, 908, 909, 910, 951, 954,  
        20368, 20492, 20495, 20599, 20607,  
        20614, 20621, 21370, 21375, 21391,  
        21396, 21401, 21405, 21411, 21412,  
        21640, 21651, 21700, 21730, 21742,  
        21758, 21788, 21791, 21842, 21857,  
        21899, 21906, 22005, 22006, 22007,  
        22008, 22157, 22158, 22162, 22164,  
        22987, 22996, 23220, 23250, 23280  
\l\_\_regex\_internal\_b\_int . . . . .  
    . . . . . 20368, 21091, 21120, 21123,  
        21124, 21126, 21130, 21137, 22191,  
        22196, 22201, 22247, 22255, 22264  
\l\_\_regex\_internal\_b\_t1 . . . . . 20368  
\l\_\_regex\_internal\_bool . . . . .  
    . . . . . 20368, 21369, 21374, 21395, 21404  
\l\_\_regex\_internal\_c\_int . . . . .  
    . . . . . 20368, 22193, 22198, 22199, 22203  
\l\_\_regex\_internal\_regex . . . . .  
    . . . . . 889, 20816, 20963, 21642, 21648,  
        21953, 22969, 22974, 22979, 22984  
\l\_\_regex\_internal\_seq . . . . . 20368,  
    21923, 21924, 21929, 21936, 21937,  
        21938, 21940, 23130, 23148, 23151  
\g\_\_regex\_internal\_tl . . . . .  
    . . . . . 20368, 20612, 20616, 21739, 21746  
\\_\_regex\_item\_caseful\_equal:n . . . . .  
    . . . . . 883, 20399, 20524,  
        20525, 20529, 20530, 20531, 20532,  
        20533, 20542, 20547, 20565, 20583,  
        20927, 21514, 21675, 21753, 21810  
\\_\_regex\_item\_caseful\_range:nn . . . . .  
    . . . . . 883, 20399, 20521,  
        20536, 20539, 20540, 20541, 20555,  
        20562, 20569, 20571, 20573, 20576,  
        20577, 20578, 20579, 20584, 20587,  
        20592, 20593, 20928, 21516, 21812  
\\_\_regex\_item\_caseless\_equal:n . . . . .  
    . . . . . 883, 20413, 21495, 21817  
\\_\_regex\_item\_caseless\_range:nn . . . . .  
    . . . . . 883, 20413, 21497, 21819  
\\_\_regex\_item\_catcode: . . . . . 20460  
\\_\_regex\_item\_catcode:nTF . . . . .  
    . . . . . 883, 899, 20460, 21016, 21324, 21824  
\\_\_regex\_item\_catcode\_reverse:nTF . . . . .  
    . . . . . 884, 20460, 21325, 21826  
\\_\_regex\_item\_cs:n . . . . .  
    . . . . . 872, 884, 20500, 21648, 21833  
\\_\_regex\_item\_equal:n . . . . .  
    . . . . . 20458, 20927, 21155, 21161,

21191, 21204, 21205, 21494, 21513  
\\_regex\_item\_exact:nn .....  
..... 884, 909, 20480, 21768, 21830  
\\_regex\_item\_exact\_ns:n .....  
..... 884, 906, 20480, 21650, 21765, 21832  
\\_regex\_item\_range:nn .....  
.. 20458, 20928, 21193, 21496, 21515  
\\_regex\_item\_reverse:n .....  
.... 218, 884, 899, 20394, 20479,  
20546, 21229, 21395, 21828, 22301  
\l\\_regex\_last\_char\_int .....  
..... 22298, 22324, 22456  
\l\\_regex\_left\_state\_int .....  
21945, 21968, 21999, 22006, 22019,  
22029, 22036, 22039, 22040, 22042,  
22043, 22069, 22077, 22080, 22107,  
22158, 22160, 22170, 22190, 22210,  
22212, 22240, 22243, 22246, 22249,  
22261, 22274, 22283, 22307, 22314  
\l\\_regex\_left\_state\_seq .....  
..... 21945, 21998, 22005, 22157  
\\_regex\_match:n .....  
..... 22343, 23054, 23064,  
23074, 23083, 23108, 23210, 23240  
\l\\_regex\_match\_count\_int .....  
948, 949, 23031, 23061, 23062, 23067  
\\_regex\_match\_cs:n ... 20511, 22343  
\\_regex\_match\_init: ..... 22343  
\\_regex\_match\_loop: .....  
..... 930, 933, 22429, 22452  
\\_regex\_match\_once: .....  
929, 931, 22360, 22388, 22410, 22448  
\\_regex\_match\_one\_active:n .. 22452  
\l\\_regex\_match\_success\_bool ...  
928, 22340, 22422, 22438, 22445, 22594  
\l\\_regex\_max\_active\_int ... 915,  
916, 21982, 22332, 22424, 22461,  
22464, 22469, 22572, 22573, 22577  
\l\\_regex\_max\_pos\_int ..... 936,  
21254, 21255, 21262, 21917, 21991,  
22319, 22357, 22374, 22385, 22470,  
23109, 23114, 23120, 23229, 23258  
\l\\_regex\_max\_state\_int .....  
..... 914, 915, 916,  
961, 21942, 21965, 21983, 22021,  
22022, 22028, 22030, 22031, 22090,  
22105, 22169, 22189, 22191, 22199,  
22243, 22249, 22257, 22267, 22352,  
22373, 22397, 22402, 22406, 23594  
\l\\_regex\_min\_active\_int .....  
..... 915, 916, 22332,  
22402, 22424, 22461, 22463, 22469  
\l\\_regex\_min\_pos\_int .....  
..... 936, 21252, 21261,  
21915, 22319, 22354, 22378, 22404  
\l\\_regex\_min\_state\_int .....  
.. 915, 916, 21942, 21965, 21982,  
21983, 22373, 22397, 22425, 23593  
\l\\_regex\_min\_submatch\_int .....  
..... 948, 951, 954, 22405,  
22407, 23034, 23132, 23245, 23253  
\l\\_regex\_mode\_int ..... 20793,  
20857, 20865, 20868, 20877, 20880,  
20889, 20897, 20900, 20910, 20911,  
20913, 20915, 20969, 20983, 20985,  
21285, 21289, 21290, 21291, 21318,  
21329, 21446, 21536, 21537, 21565,  
21566, 21622, 21623, 21731, 21777  
\\_regex\_mode\_quit\_c: .....  
..... 20908, 21006, 21419  
\\_regex\_msg\_repeated:nnN .....  
..... 21872, 21893, 21903, 23557  
\\_regex\_multi\_match:n .....  
927, 22432, 23062, 23081, 23089, 23237  
\c\\_regex\_no\_match\_regex .....  
..... 20377, 20816, 22961  
\c\\_regex\_outer\_mode\_int .....  
20793, 20868, 20880, 20889, 20897,  
20911, 20969, 20985, 21731, 21777  
\\_regex\_pop\_lr\_states: .....  
..... 21988, 21996, 22112  
\\_regex\_posix\_alnum: ..... 20549  
\\_regex\_posix\_alpha: ..... 20549  
\\_regex\_posix\_ascii: ..... 20549  
\\_regex\_posix\_blank: ..... 20549  
\\_regex\_posix\_cntrl: ..... 20549  
\\_regex\_posix\_digit: ..... 20549  
\\_regex\_posix\_graph: ..... 20549  
\\_regex\_posix\_lower: ..... 20549  
\\_regex\_posix\_print: ..... 20549  
\\_regex\_posix\_punct: ..... 20549  
\\_regex\_posix\_space: ..... 20549  
\\_regex\_posix\_upper: ..... 20549  
\\_regex\_posix\_word: ..... 20549  
\\_regex\_posix\_xdigit: ..... 20549  
\\_regex\_prop\_-: ..... 896, 21210  
\\_regex\_prop\_d: ... 896, 20520, 20567  
\\_regex\_prop\_h: ..... 20520, 20559  
\\_regex\_prop\_N: ..... 20520, 21238  
\\_regex\_prop\_s: ..... 20520  
\\_regex\_prop\_v: ..... 20520  
\\_regex\_prop\_w: .....  
... 20520, 20588, 22299, 22301, 22302  
\\_regex\_push\_lr\_states: .....  
..... 21986, 21996, 22110  
\\_regex\_query\_get: .....  
..... 22428, 22458, 22496

\\_\\_regex\\_query\\_range:nn 936, 22611,  
     22616, 22635, 22723, 23224, 23257  
 \\_\\_regex\\_query\\_range\\_loop:ww . 22616  
 \\_\\_regex\\_query\\_set:nnn . . . . .  
     928, 22353, 22356,  
     22358, 22377, 22381, 22386, 22481  
 \\_\\_regex\\_query\\_submatch:n . . . . .  
     22633, 22771, 23164  
 \\_\\_regex\\_replace\\_all:nnN 23028, 23234  
 \\_\\_regex\\_replace\\_once:nnN . . . . .  
     23026, 23204  
 \\_\\_regex\\_replacement:n . . . . .  
     22674, 23209, 23239  
 \\_\\_regex\\_replacement\\_aux:n . . 22674  
 \\_\\_regex\\_replacement\\_balance\_-  
     one\\_match:n . . . . . 218,  
     935, 936, 22607, 22708, 23217, 23248  
 \\_\\_regex\\_replacement\_c:w . . . . . 22815  
 \\_\\_regex\\_replacement\_c\_A:w 939, 22896  
 \\_\\_regex\\_replacement\_c\_B:w . . 22899  
 \\_\\_regex\\_replacement\_c\_C:w . . 22908  
 \\_\\_regex\\_replacement\_c\_D:w . . 22913  
 \\_\\_regex\\_replacement\_c\_E:w . . 22916  
 \\_\\_regex\\_replacement\_c\_L:w . . 22925  
 \\_\\_regex\\_replacement\_c\_M:w . . 22928  
 \\_\\_regex\\_replacement\_c\_O:w . . 22931  
 \\_\\_regex\\_replacement\_c\_P:w . . 22934  
 \\_\\_regex\\_replacement\_c\_S:w . . 22940  
 \\_\\_regex\\_replacement\_c\_T:w . . 22948  
 \\_\\_regex\\_replacement\_c\_U:w . . 22951  
 \\_\\_regex\\_replacement\\_cat:NNN . . . . .  
     22823, 22856  
 \l\\_\\_regex\\_replacement\\_category\_-  
     seq . . . . . 22604,  
     22702, 22705, 22706, 22742, 22870  
 \l\\_\\_regex\\_replacement\\_category\_-  
     tl . . . . . 939, 22604,  
     22737, 22743, 22749, 22871, 22872  
 \\_\\_regex\\_replacement\\_char:nNN . . . . .  
     946, 22891,  
     22898, 22905, 22915, 22922, 22927,  
     22930, 22933, 22937, 22950, 22953  
 \l\\_\\_regex\\_replacement\\_csnames\_-  
     int . . . . . 935, 22603, 22696,  
     22698, 22700, 22772, 22831, 22838,  
     22849, 22851, 22861, 22902, 22919  
 \\_\\_regex\\_replacement\\_cu\\_aux:Nw . . . . .  
     22820, 22829, 22844  
 \\_\\_regex\\_replacement\\_do\\_one\_-  
     match:n . . 22609, 22721, 23222, 23256  
 \\_\\_regex\\_replacement\\_error:NNN . . . . .  
     22786, 22798,  
     22809, 22824, 22827, 22845, 22955  
 \\_\\_regex\\_replacement\\_escaped:N . . . . .  
     22692, 22755, 22875  
 \\_\\_regex\\_replacement\\_exp\\_not:N . . . . .  
     942, 22615, 22820  
 \\_\\_regex\\_replacement\_g:w . . . . . 22781  
 \\_\\_regex\\_replacement\_g\_digits:NN . . . . .  
     22781  
 \\_\\_regex\\_replacement\\_normal:n . . . . .  
     22688, 22693, 22735, 22762, 22784,  
     22790, 22817, 22843, 22853, 22868  
 \\_\\_regex\\_replacement\\_put\_-  
     submatch:n . . . . . 22760, 22767, 22805  
 \\_\\_regex\\_replacement\_rbrace:N . . . . .  
     22686, 22804, 22847  
 \\_\\_regex\\_replacement\_u:w . . . . . 22840  
 \\_\\_regex\\_return: . . . . .  
     948, 23002, 23007, 23018, 23020, 23040  
 \l\\_\\_regex\\_right\\_state\\_int . . . . .  
     . . . . . 21945, 21971, 21989, 22001,  
     22008, 22020, 22029, 22030, 22069,  
     22076, 22082, 22095, 22105, 22107,  
     22160, 22164, 22175, 22189, 22198,  
     22210, 22214, 22218, 22221, 22226,  
     22229, 22232, 22240, 22254, 22257,  
     22260, 22263, 22267, 22283, 22314  
 \l\\_\\_regex\\_right\\_state\\_seq . . . . .  
     . . . . . 21945, 22000, 22007, 22162  
 \l\\_\\_regex\\_saved\\_success\\_bool . . . . .  
     . . . . . 928, 20509, 20516, 22340  
 \\_\\_regex\\_show:N . . 21781, 22984, 22993  
 \\_\\_regex\\_show\\_anchor\\_to\\_str:N . . . . .  
     . . . . . 21809, 21910  
 \\_\\_regex\\_show\\_class:NnnnN . . . . .  
     . . . . . 21800, 21874  
 \\_\\_regex\\_show\\_group\\_aux:nnnnN . . . . .  
     . . . . . 21795, 21797, 21799, 21865  
 \\_\\_regex\\_show\\_item\\_catcode:NnTF . . . . .  
     . . . . . 21825, 21827, 21921  
 \\_\\_regex\\_show\\_item\\_exact\\_cs:n . . . . .  
     . . . . . 21832, 21934  
 \l\\_\\_regex\\_show\\_lines\\_int . . . . .  
     . . . . . 20818, 21846, 21878, 21881, 21888  
 \\_\\_regex\\_show\\_one:n . . . . . 21789,  
     21802, 21805, 21811, 21814, 21818,  
     21821, 21831, 21835, 21844, 21860,  
     21867, 21871, 21884, 21900, 21939  
 \\_\\_regex\\_show\\_pop: . . . . . 21854, 21870  
 \l\\_\\_regex\\_show\\_prefix\\_seq . . . . .  
     . . . . . 20817, 21787,  
     21790, 21836, 21850, 21855, 21857  
 \\_\\_regex\\_show\\_push:n . . . . .  
     . . . . . 21837, 21854, 21868, 21879  
 \\_\\_regex\\_show\\_scope:nn . . . . .  
     . . . . . 21829, 21834, 21854, 21926

\\_\_regex\_single\_match: .....  
     927, 20506, [22432](#), 23052, 23072, 23207  
 \\_\_regex\_split:nnN .... 23030, [23086](#)  
 \\_\_regex\_standard\_escapechar: ...  
     20328, 20331, 20611, 20968, 21963  
 \l\_\_regex\_start\_pos\_int .....  
     21253, 21916,  
     22319, 22416, 22421, 22427, 23092,  
     23104, 23117, 23120, 23194, 23258  
 \g\_\_regex\_state\_active\_intarray .  
     .. 869, 915, 927, 927, 928, [22334](#),  
     22400, 22513, 22516, 22524, 22551  
 \l\_\_regex\_step\_int .....  
     869, [22331](#), 22403, 22454,  
     22514, 22518, 22526, 22540, 22542  
 \\_\_regex\_store\_state:n .....  
     22425, 22565, [22568](#)  
 \\_\_regex\_store\_submatches: ....  
     22568, 22582  
 \\_\_regex\_submatch\_balance:n ....  
     .. 22608, [22639](#), 22712, 22775, 23156  
 \g\_\_regex\_submatch\_begin\_-  
     intarray .... 869, 935, 22613,  
     22636, 22661, 22669, 22730, [23037](#),  
     23099, 23102, 23115, 23176, 23200  
 \g\_\_regex\_submatch\_end\_intarray .  
     869, 22637, 22646, 22654, [23037](#),  
     23096, 23112, 23178, 23203, 23226  
 \l\_\_regex\_submatch\_int .....  
     870, 948, 950, 951, 954,  
     22407, [23034](#), 23111, 23113, 23116,  
     23118, 23121, 23133, 23173, 23177,  
     23179, 23181, 23182, 23247, 23255  
 \g\_\_regex\_submatch\_prev\_intarray  
     .... 869, 948, 952, 22612, 22726,  
     [23037](#), 23094, 23110, 23180, 23193  
 \\_\_regex\_success\_bool .... 928,  
     20510, 20512, 20515, [22340](#), 22395,  
     22437, 22446, 23042, 23172, 23211  
 \l\_\_regex\_success\_pos\_int .....  
     .. [22319](#), 22404, 22421, 22597, 23092  
 \l\_\_regex\_success\_submatches\_-  
     prop .. 926, 952, [22329](#), 22598, 23184  
 \\_\_regex\_tests\_action\_cost:n ...  
     .... [22047](#), 22068, 22077, 22095  
 \g\_\_regex\_thread\_state\_intarray .  
     ..... 869, 915,  
     925, 927, 927, 934, [22334](#), 22478, 22571  
 \\_\_regex\_tmp:w .....  
     20350, 20352, 20356, 20358, [20367](#),  
     21222, 21232, 21233, 21234, 21235,  
     21236, 21247, 21252, 21253, 21254,  
     21255, 21256, 21261, 21262, 23013,  
     23022, 23024, 23026, 23028, 23030  
 \\_\_regex\_toks\_clear:N . [20334](#), 22028  
 \\_\_regex\_toks\_memcpy:NNn [20339](#), 22200  
 \\_\_regex\_toks\_put\_left:Nn .....  
     ... [20348](#), 22011, 22182, 22183  
 \\_\_regex\_toks\_put\_right:Nn .....  
     .. 871, [20348](#), 21968, 21971,  
     21989, 22013, 22036, 22274, 22307  
 \\_\_regex\_toks\_set:Nn .....  
     ... [20334](#), 22489, 22577  
 \\_\_regex\_toks\_use:w .....  
     .. [20333](#), 22479, 22515, 22629, 23597  
 \\_\_regex\_trace:nnN . 22016, 22346,  
     22365, 22391, 22507, [23573](#), 23596  
 \\_\_regex\_trace\_pop:nnN .....  
     ... 20600, 21959, 21978, 22100,  
     22153, 22348, 22367, 22676, [23573](#)  
 \\_\_regex\_trace\_push:nnN .....  
     ... 20597, 21956, 21975, 22099,  
     22152, 22345, 22364, 22675, [23573](#)  
 \g\_\_regex\_trace\_regex\_int .... [23587](#)  
 \\_\_regex\_trace\_states:n .....  
     ... 21958, 21977, [23588](#)  
 \\_\_regex\_two\_if\_eq:NNNN ..... 20819  
 \\_\_regex\_two\_if\_eq:NNNNTF .....  
     ... [20819](#), 21057, 21104,  
     21117, 21151, 21330, 21367, 21387,  
     21388, 21457, 21492, 21509, 21510,  
     21572, 21698, 22783, 22842, 22868  
 \\_\_regex\_use\_state: .....  
     ... [22505](#), 22528, 22554  
 \\_\_regex\_use\_state\_and\_submatches:nn  
     ... [931](#), 22477, [22520](#)  
 \l\_\_regex\_zeroth\_submatch\_int ...  
     ... 948, 952, [23034](#),  
     23095, 23097, 23100, 23103, 23173,  
     23191, 23194, 23218, 23223, 23227  
 \regular\_expression ..... 217  
 \relax ..... 14, 21, 39, 43, 49, 90, 92,  
     93, 94, 105, 130, 153, 174, 188, 218,  
     219, 220, 221, 222, 223, 224, 225,  
     226, 227, 228, 231, 232, 233, 234,  
     235, 236, 237, 238, 239, 240, 241, 526  
 \relpenalty ..... 527  
 \RequirePackage ..... 156  
 \resettimer ..... 887  
 reverse commands:  
     \reverse\_if:N .....  
         ... 20, 420, 449, 451, 675, [2049](#),  
         5619, 6481, 6659, 6661, 6663, 6665,  
         6730, 7563, 11510, 11515, 11519,  
         11521, 14080, 17619, 18371, 18394,  
         20407, 20408, 20427, 20428, 20435,  
         20436, 25573, 25575, 25594, 25618  
     \right ..... 528

right commands:

- \c\_right\_brace\_str ..... 62, 5671, 20719, 21084, 21104, 21117, 21630, 21634, 21716, 22685
- \rightghost ..... 995, 1824
- \righthyphenmin ..... 529
- \rightmarginkern ..... 799, 1613
- \rightskip ..... 530
- \romannumeral ..... 531
- round ..... 198
- \rppcode ..... 800, 1614
- \rule ..... 25114, 25169

**S**

s@ internal commands:

- \s\_..... 855
- \saveboxresource ..... 1038, 1648
- \savecatcodetable ..... 996, 1796
- \saveimageresource ..... 1039, 1649
- \savepos ..... 1037, 1647
- \savinghyphcodes ..... 662, 1476
- \savingvdiscards ..... 663, 1477

scan commands:

- \scan\_align\_safe\_stop: ..... 29756
- \scan\_new:N ..... 66, 5812, 5825, 5830, 9039, 13190, 13193, 13194, 13195, 13196, 13197, 13198, 13199, 13291, 19883
- \scan\_stop: ..... 8, 66, 66, 79, 128, 129, 129, 129, 258, 272, 311, 314, 314, 323, 337, 337, 339, 351, 352, 363, 377, 380, 392, 420, 426, 450, 455, 470, 511, 511, 518, 520, 521, 530, 553, 586, 586, 586, 592, 672, 675, 676, 677, 681, 884, 906, 1072, 2077, 2336, 2354, 2367, 2432, 2437, 2453, 2458, 2742, 2760, 2770, 2788, 2814, 3200, 3635, 3766, 3806, 3832, 3856, 3873, 3997, 4003, 4247, 5620, 5822, 6129, 6875, 8649, 8721, 8941, 9016, 9025, 9034, 10456, 10458, 10627, 10629, 11122, 11126, 11366, 11396, 11399, 11404, 11408, 11412, 11417, 11423, 11428, 11684, 11756, 11785, 11788, 11797, 11800, 11805, 11808, 11838, 11851, 11861, 11890, 11927, 11930, 11941, 11944, 11949, 11952, 11963, 12016, 12040, 12079, 12084, 12086, 12107, 12109, 12954, 12962, 13176, 13355, 14078, 14082, 14285, 14302, 14602, 14649, 14650, 14909, 14952, 14982, 15718, 17529, 17537, 18213, 18216, 18219, 18222, 18225, 18228, 18231, 18234, 18237, 19497, 20005, 20045, 20049, 20055, 20057, 20104, 20106, 20493, 20494, 20841, 21659, 21936, 22500, 22503, 22530, 22893, 22945, 23614, 23765, 25110, 25165, 26456, 26457, 26577, 26581, 26726, 26737, 27941, 27944, 27946, 28170, 28570, 28576, 28733, 28776, 28787, 29799

scan internal commands:

- \g\_\_scan\_marks\_tl ... 5811, 5815, 5821
- \scantextokens ..... 997, 1797
- \scantokens ..... 664, 1478
- \scriptfont ..... 532
- \scriptscriptfont ..... 533
- \scriptscriptstyle ..... 534
- \scriptspace ..... 535
- \scriptstyle ..... 536
- \scrollmode ..... 537
- sec ..... 198
- secd ..... 199

seq commands:

- \c\_empty\_seq ..... 77, 427, 5839, 5843, 5847, 5850, 6032, 6063, 6071
- \l\_foo\_seq ..... 213
- \seq\_clear:N 68, 68, 77, 5846, 5853, 5976, 9773, 9836, 11284, 21836, 22706
- \seq\_clear\_new:N ..... 68, 5852
- \seq\_concat:NNN .. 69, 77, 5929, 11292
- \seq\_const\_from\_clist:Nn .. 250, 26338
- \seq\_count:N ..... 70, 74, 76, 182, 6190, 6204, 6268, 6296, 22705, 26353
- \seq\_elt:w ..... 426
- \seq\_elt\_end: ..... 426
- \seq\_gclear:N ..... 68, 842, 5846, 5856, 19619, 26368
- \seq\_gclear\_new:N ..... 68, 5852
- \seq\_gconcat:NNN ..... 69, 5929, 11304
- \seq\_get:NN ... 75, 6349, 22157, 22162
- \seq\_get:NNTF ..... 75, 6355
- \seq\_get\_left:NN ..... 69, 6079, 6349, 6350, 6355, 6356
- \seq\_get\_left:NNTF ..... 71, 6149
- \seq\_get\_right:NN ..... 70, 6104
- \seq\_get\_right:NNTF ..... 71, 6149
- \seq\_gpop:NN ..... 75, 6349, 11229
- \seq\_gpop:NNTF 76, 6355, 10440, 10611
- \seq\_gpop\_left:NN ..... 70, 6090, 6353, 6354, 6359, 6360
- \seq\_gpop\_left:NNTF ..... 71, 6157
- \seq\_gpop\_right:NN ..... 70, 6122
- \seq\_gpop\_right:NNTF ..... 72, 6157
- \seq\_gpush:Nn ..... 22, 76, 6329, 10467, 10639, 11212

```

\seq_gput_left:Nn ..... 69, 5939, 6339, 6340, 6341, 6342,
6343, 6344, 6345, 6346, 6347, 6348
\seq_gput_right:Nn ..... 69, 5960, 11037, 11044, 11057, 11198, 11203
\seq_gremove_all:Nn . 72, 5986, 10579
\seq_gremove_duplicates:N .. 72, 5970
\seq_greverse:N ..... 72, 6012
\seq_gset_eq>NN ..... 68, 5850, 5858, 5973, 19593, 26350
\seq_gset_filter>NNn .... 250, 26304
\seq_gset_from_clist>NN .... 68, 5866
\seq_gset_from_clist:Nn .... 68, 5866
\seq_gset_from_function:Nnn .... 250, 26334
\seq_gset_from_inline_x:Nnn .... 251, 19611, 26324, 26337, 26363
\seq_gset_map>NNn .... 250, 26314
\seq_gset_split:Nnn ..... 69, 5892, 10385, 10573
\seq_gshuffle:N ..... 251, 26344
\seq_gsort:Nn ..... 72, 6030, 19589
\seq_if_empty:NTF ..... 72, 6030, 6203, 7827, 22702
\seq_if_empty_p:N ..... 72, 6030
\seq_if_exist:NTF ..... 69, 5853, 5856, 5935, 6294
\seq_if_exist_p:N ..... 69, 5935
\seq_if_in:Nn ..... 493
\seq_if_in:NnTF ..... 73, 76, 77, 5979, 6040, 10466, 10638
\seq_indexed_map_function>NNN ... 251, 26397
\seq_indexed_map_inline:Nn 251, 26397
\seq_item:Nn ..... 70, 216, 437, 6177, 6204, 9854, 9855, 9860
\seq_log:N ..... 78, 6361
\seq_map_break: 73, 250, 250, 6207, 6218, 6253, 6264, 12734, 26400, 26409
\seq_map_break:n ..... 74, 438, 6207, 9793, 9807, 11141, 19590, 19593
\seq_map_function>NN ..... 4, 73, 248, 1036, 6211, 6371, 7833, 9858, 11295, 21850, 21929
\seq_map_inline:Nn ..... 73, 73, 77, 1033, 5977, 6249, 9788, 11072, 11140, 12727, 19590, 19593
\seq_map_variable>NNn .... 73, 6256
\seq_mapthread_function>NNN ... 250, 26282
\seq_new:N ..... 4, 68, 68, 5840, 5853, 5856, 5969, 6375, 6376, 6377, 6378, 7978, 8437, 8440, 9743, 9744, 10383, 10570, 11028, 11052, 11065, 11067, 12151, 19451, 20374, 20817, 21947, 21948, 22605, 26348
\seq_pop>NN ..... 75, 6349, 22005, 22007, 22742
\seq_pop>NNTF ..... 76, 6355
\seq_pop_left>NN ..... 70, 6090, 6351, 6352, 6357, 6358
\seq_pop_left>NNTF ..... 71, 6157
\seq_pop_right>NN ..... 70, 6122, 21787, 21857
\seq_pop_right>NNTF ..... 71, 6157
\seq_push:Nn ..... 76, 6329, 6336, 21998, 22000, 22870
\seq_put_left:Nn ..... 69, 5939, 6329, 6330, 6331, 6332, 6333, 6334, 6335, 6336, 6337, 6338, 9783
\seq_put_right:Nn ..... 69, 76, 77, 5960, 5980, 9844, 21790, 21855, 29707
\seq_rand_item:N ..... 70, 6201
\seq_remove_all:Nn ..... 69, 72, 76, 77, 5986, 8004, 29709
\seq_remove_duplicates:N ..... 72, 76, 77, 5970, 11293
\seq_reverse:N ..... 72, 432, 6012
\seq_set_eq>NN ..... 68, 77, 5847, 5858, 5971, 19590, 26349
\seq_set_filter>NNn ..... 250, 1034, 21924, 26304
\seq_set_from_clist>NN 68, 5866, 8003
\seq_set_from_clist:Nn ..... 68, 107, 1035, 5866, 11288, 11302, 12656
\seq_set_from_function>NNN ... 250, 23130, 26334
\seq_set_from_inline_x:Nnn .... 251, 1034, 26324, 26335
\seq_set_map>NNn ..... 250, 21937, 23148, 26314
\seq_set_split:Nnn ..... 69, 5892, 8438, 8441, 21923, 21936
\seq_show:N ..... 78, 540, 6361
\seq_shuffle:N ..... 251, 26344
\seq_sort:Nn ..... 72, 206, 6030, 19589
\seq_use:Nn ..... 75, 6292, 21940
\seq_use:Nnnn ..... 74, 6292
\g_tmpr_seq ..... 78, 6375
\l_tmpr_seq ..... 78, 6375
\g_tmprb_seq ..... 78, 6375
\l_tmprb_seq ..... 78, 6375
seq internal commands:
\__seq_count:w ..... 439, 6268
\__seq_count_end:w ..... 439, 6268
\__seq_get_left:wnw ..... 6079
\__seq_get_right_end:Nnn ..... 6104
\__seq_get_right_loop:nw .. 435, 6104

```

```

\__seq_if_in: ..... 6040
\__seq_indexed_map>NN ..... 26399, 26407, 26412
\__seq_indexed_map:nNN ..... 26397
\__seq_indexed_map:Nw ... 1036, 26397
\l__seq_internal_a_int ..... 26344
\l__seq_internal_a_t1 ..... 429, 5836, 5900, 5904, 5910,
                           5915, 5917, 6001, 6006, 6044, 6048
\l__seq_internal_b_int ..... 26344
\l__seq_internal_b_t1 ..... 5836, 5997, 6001, 6047, 6048
\g__seq_internal_seq ..... 26344
\__seq_item:n . 426, 426, 426, 427,
                430, 433, 434, 435, 437, 438, 438,
                439, 439, 440, 1033, 1033, 1034,
                5831, 5943, 5951, 5961, 5963, 5968,
                6018, 6019, 6021, 6026, 6045, 6084,
                6087, 6097, 6112, 6115, 6128, 6129,
                6140, 6184, 6193, 6217, 6220, 6230,
                6235, 6241, 6245, 6275, 6276, 6277,
                6278, 6279, 6280, 6281, 6282, 6287,
                6288, 6303, 6318, 6321, 6324, 26320,
                26330, 26331, 26360, 26420, 26422
\__seq_item:nN ..... 6177
\__seq_item:nwn ..... 6177
\__seq_item:wNn ..... 6177
\__seq_map_function>NNn ..... 6211
\__seq_map_function:Nw ..... 6214, 6220, 6224
\__seq_mapthread_function:Nnnwnn
                ..... 26282
\__seq_mapthread_function:wNN . 26282
\__seq_mapthread_function:wNw . 26282
\__seq_pop>NNNN ..... 6061, 6091, 6093, 6123, 6125
\__seq_pop_item_def: ..... 426, 427, 6008,
                           6227, 6253, 6264, 26312, 26322, 26332
\__seq_pop_left>NNN . 6090, 6159, 6162
\__seq_pop_left:wnwNNN ..... 6090
\__seq_pop_right>NNN ..... 431, 6122, 6165, 6168
\__seq_pop_right_loop:nn ..... 6122
\__seq_pop_TF>NNNN ..... 436, 6061,
                           6150, 6152, 6159, 6162, 6165, 6168
\__seq_push_item_def: ..... 6227
\__seq_push_item_def:n ..... 426, 427, 5992,
                           6227, 6251, 6258, 26310, 26320, 26330
\__seq_put_left_aux:w .... 430, 5939
\__seq_remove_all_aux>NNn .... 5986
\__seq_remove_duplicates>NN .. 5970
\l__seq_remove_seq ..... 5969, 5976, 5979, 5980, 5982
\__seq_reverse>NN ..... 6012
\__seq_reverse_item:nw ..... 432
\__seq_reverse_item:nwn ..... 6012
\__seq_set_filter>NNNn ..... 26304
\__seq_set_from_inline_x>NNnn . 26324
\__seq_set_map>NNNn ..... 26314
\__seq_set_split>NNnn ..... 5892
\__seq_set_split_auxi:w ... 429, 5892
\__seq_set_split_auxii:w ... 429, 5892
\__seq_set_split_end: ... 429, 5892
\__seq_show>NN ..... 6361
\__seq_shuffle>NN ..... 26344
\__seq_shuffle_item:n ..... 26344
\__seq_tmp:w ..... 5838, 6018, 6021, 6128, 6140
\__seq_use>NNnNnn ..... 6292
\__seq_use:nwnn ..... 6292
\__seq_use:nwwwnwn ..... 6292
\__seq_use_setup:w ..... 6292
\__seq_wrap_item:n ..... 429, 1033, 5869, 5874, 5879, 5884,
                     5901, 5926, 5968, 6004, 26310, 26341
\setbox ..... 538
\setfontid ..... 998, 1798
\setlanguage ..... 539
\setrandomseed ..... 1040, 1650
\sfcode ..... 190, 540
\sffamily ..... 25103
\shapemode ..... 999, 1799
\shellescape ..... 888, 1620
\Shipout ..... 1289
\shipout ..... 541, 1276, 1277
\ShortText ..... 75, 123, 140
\show ..... 542
\showbox ..... 543
\showboxbreadth ..... 544
\showboxdepth ..... 545
\showgroups ..... 665, 1479
\showifs ..... 666, 1480
\showlists ..... 546
\showmode ..... 1242, 2024
\showthe ..... 547
\ShowTokens ..... 207
\showtokens ..... 667, 1481
sign ..... 198
sin ..... 198
sind ..... 199
\sjis ..... 1243, 2025
\skewchar ..... 548
\skip ..... 549, 8799
skip commands:
               \c_max_skip ..... 165, 11872

```

\skip\_add:Nn ..... 163, 11795  
 \skip\_const:Nn ..... 162, 600, 11752, 11872, 11873  
 \skip\_eval:n ..... 164, 164, 164,  
     164, 11756, 11813, 11835, 11867, 11871  
 \skip\_gadd:Nn ..... 163, 11795  
 .skip\_gset:N ..... 173, 12548  
 \skip\_gset:Nn ..... 163, 595, 11783  
 \skip\_gset\_eq:NN ..... 163, 11791  
 \skip\_gsub:Nn ..... 163, 11795  
 \skip\_gzero:N ..... 163, 11759, 11768  
 \skip\_gzero\_new:N ..... 163, 11765  
 \skip\_horizontal:N ..... 165, 11842  
 \skip\_horizontal:n .....  
     165, 11842, 28212, 28600, 28811, 29343  
 \skip\_if\_eq:nnTF ..... 164, 11811  
 \skip\_if\_eq\_p:nn ..... 164, 11811  
 \skip\_if\_exist:NTF .....  
     163, 11766, 11768, 11771  
 \skip\_if\_exist\_p:N ..... 163, 11771  
 \skip\_if\_finite:nTF ..... 164, 11817, 26454  
 \skip\_if\_finite\_p:n ..... 164, 11817  
 \skip\_log:N ..... 165, 11868  
 \skip\_log:n ..... 165, 11868  
 \skip\_new:N .....  
     .. 162, 163, 11744, 11755, 11766,  
         11768, 11874, 11875, 11876, 11877  
 .skip\_set:N ..... 173, 12548  
 \skip\_set:Nn ..... 163, 11783  
 \skip\_set\_eq:NN ..... 163, 11791  
 \skip\_show:N ..... 164, 11864  
 \skip\_show:n ..... 164, 599, 11866  
 \skip\_split\_finite\_else\_action:nnNN .....  
     251, 26452  
 \skip\_sub:Nn ..... 163, 11795  
 \skip\_use:N .....  
     .. 164, 164, 11829, 11838, 11839  
 \skip\_vertical:N ..... 165, 11842  
 \skip\_vertical:n ..... 165, 11842  
 \skip\_zero:N ..... 163, 163, 166, 11759, 11766  
 \skip\_zero\_new:N ..... 163, 11765  
 \g\_tmpa\_skip ..... 165, 11874  
 \l\_tmpa\_skip ..... 165, 11874  
 \g\_tmpb\_skip ..... 165, 11874  
 \l\_tmpb\_skip ..... 165, 11874  
 \c\_zero\_skip ..... 165, 585, 11370, 11373,  
     11760, 11762, 11872, 26460, 26461  
 skip internal commands:  
 \\_\_skip\_if\_finite:wwNw ..... 11817  
 \\_\_skip\_tmp:w ..... 596, 11775,  
     11783, 11786, 11795, 11798, 11803,  
     11806, 11817, 11834, 11911, 11925,  
     11928, 11939, 11942, 11947, 11950  
 \skipdef ..... 550

sort commands:  
 \sort\_ordered: ..... 29758  
 \sort\_return\_same: .....  
     .. 206, 206, 844, 19672, 29759  
 \sort\_return\_swapped: .....  
     .. 206, 206, 844, 19672, 29761  
 \sort\_reversed: ..... 29760  
 sort internal commands:  
 \\_\_sort:nnNnn ..... 846, 847  
 \l\_\_sort\_A\_int .....  
     .. 844, 19461, 19466, 19473, 19476,  
         19485, 19636, 19641, 19644, 19664,  
         19696, 19703, 19718, 19720, 19721  
 \l\_\_sort\_B\_int .....  
     .. 844, 844, 19461, 19641, 19645,  
         19653, 19655, 19656, 19708, 19709,  
         19718, 19719, 19728, 19729, 19731  
 \l\_\_sort\_begin\_int .....  
     .. 838, 844, 19459, 19633, 19721, 19731  
 \l\_\_sort\_block\_int .....  
     .. 838, 839, 843, 19458, 19468,  
         19473, 19477, 19480, 19485, 19486,  
         19563, 19624, 19627, 19634, 19637  
 \l\_\_sort\_C\_int .....  
     .. 844, 844, 19461, 19642, 19646,  
         19653, 19654, 19665, 19697, 19704,  
         19708, 19710, 19711, 19728, 19730  
 \\_\_sort\_compare:nn .....  
     .. 841, 845, 19562, 19663  
 \\_\_sort\_compute\_range: .....  
     .. 838, 839, 839, 19490, 19550  
 \\_\_sort\_copy\_block: ..... 843, 19643, 19651  
 \\_\_sort\_disable\_toksdef: ..... 19549, 19828  
 \\_\_sort\_disabled\_toksdef:n ..... 19828  
 \l\_\_sort\_end\_int ..... 838, 843, 844, 844,  
     19459, 19625, 19633, 19634, 19635,  
     19636, 19637, 19638, 19639, 19656  
 \\_\_sort\_error: ..... 19822, 19835, 19854  
 \\_\_sort\_i:nnnnNn ..... 848  
 \g\_\_sort\_internal\_seq .....  
     .. 841, 842, 19451, 19611, 19618, 19619  
 \g\_\_sort\_internal\_tl .....  
     .. 19451, 19574, 19577, 19578  
 \l\_\_sort\_length\_int .....  
     .. 838, 838, 19453, 19560, 19624  
 \\_\_sort\_level: .....  
     .. 841, 851, 19564, 19622, 19826  
 \\_\_sort\_loop:wNn ..... 847, 848  
 \\_\_sort\_main:NNNn .....  
     .. 841, 842, 19547, 19573, 19610  
 \l\_\_sort\_max\_int .....  
     .. 838, 839, 19453, 19470, 19544, 19554  
 \c\_\_sort\_max\_length\_int ..... 19490

\\_\_sort\_merge\_blocks: .....  
     ..... 19626, 19631, 19825  
 \\_\_sort\_merge\_blocks\_aux: .....  
     843, 19647, 19661, 19714, 19724, 19824  
 \\_\_sort\_merge\_blocks\_end: .....  
     ..... 846, 19722, 19726  
 \l\_\_sort\_min\_int 838, 839, 841, 841,  
     19453, 19467, 19475, 19493, 19509,  
     19517, 19530, 19542, 19551, 19561,  
     19575, 19614, 19625, 19852, 19853  
 \\_\_sort\_quick\_cleanup:w ..... 19736  
 \\_\_sort\_quick\_end:nnTFNn .....  
     ..... 849, 850, 19756, 19796  
 \\_\_sort\_quick\_only\_i:NnnnnNn . 19761  
 \\_\_sort\_quick\_only\_i\_end:nnwnw ..  
     ..... 19772, 19796  
 \\_\_sort\_quick\_only\_ii:NnnnnNn . 19761  
 \\_\_sort\_quick\_only\_ii\_end:nnwnw ..  
     ..... 19779, 19796  
 \\_\_sort\_quick\_prepare:Nnnn ... 19736  
 \\_\_sort\_quick\_prepare\_end:NNNnw ..  
     ..... 19736  
 \\_\_sort\_quick\_single\_end:nnwnw ..  
     ..... 19765, 19796  
 \\_\_sort\_quick\_split:NnNn ..  
     ..... 848, 849, 19756,  
     19761, 19801, 19808, 19814, 19816  
 \\_\_sort\_quick\_split\_end:nnwnw ..  
     ..... 19786, 19793, 19796  
 \\_\_sort\_quick\_split\_i:NnnnnNn ..  
     ..... 847, 19761  
 \\_\_sort\_quick\_split\_ii:NnnnnNn 19761  
 \\_\_sort\_redefine\_compute\_range: ..  
     ..... 19490  
 \\_\_sort\_return\_mark:w ..  
     ..... 844, 19667, 19668, 19672  
 \\_\_sort\_return\_none\_error: ..  
     ..... 844, 19670, 19672, 19706, 19716  
 \\_\_sort\_return\_same:w ..  
     ..... 844, 19680, 19698, 19706  
 \\_\_sort\_return\_swapped:w 19690, 19716  
 \\_\_sort\_return\_two\_error: 844, 19672  
 \\_\_sort\_seq:NNNNn ..... 841, 19589  
 \\_\_sort\_shrink\_range: ..... 839,  
     840, 19464, 19495, 19511, 19519, 19532  
 \\_\_sort\_shrink\_range\_loop: ... 19464  
 \\_\_sort\_t1>NNn ..... 841, 19566  
 \\_\_sort\_t1\_toks:w ..... 841, 19566  
 \\_\_sort\_too\_long\_error:NNw ..  
     ..... 19555, 19847  
 \l\_\_sort\_top\_int 838, 841, 841, 844,  
     844, 19453, 19551, 19554, 19557,  
     19558, 19561, 19583, 19614, 19635,  
     19638, 19639, 19642, 19711, 19853  
 \l\_\_sort\_true\_max\_int ..... 838,  
     839, 19453, 19467, 19480, 19494,  
     19510, 19518, 19531, 19543, 19852  
 sp ..... 201  
 spac commands:  
     \spac\_directions\_normal\_body\_dir ..  
         ..... 1404  
     \spac\_directions\_normal\_page\_dir ..  
         ..... 1405  
 \space ..... 55  
 \spacefactor ..... 551  
 \spaceskip ..... 552  
 \span ..... 553  
 \special ..... 554  
 \splitbotmark ..... 555  
 \splitbotmarks ..... 668, 1482  
 \spliddiscards ..... 669, 1483  
 \splitfirstmark ..... 556  
 \splitfirstmarks ..... 670, 1484  
 \splitmaxdepth ..... 557  
 \splittopskip ..... 558  
 sqrt ..... 200  
 \SS ..... 27690  
 \ss ..... 27690  
 str commands:  
     \c\_ampersand\_str ..... 62, 5671  
     \c\_atsign\_str ..... 62, 5671  
     \c\_backslash\_str 62, 5671, 20629, 21201  
     \c\_circumflex\_str ..... 62, 5671  
     \c\_colon\_str .....  
         ... 62, 5671, 8712, 8817, 8823, 12245  
     \c\_dollar\_str ..... 62, 5671  
     \c\_hash\_str .....  
         ... 62, 5671, 25618, 25649, 25650,  
         25654, 29328, 29494, 29501, 29541  
     \c\_percent\_str .....  
         ... 62, 5671, 29627, 29628, 29629  
 \str\_case:nn ..  
     ..... 55, 5229, 10241, 21350, 26468  
 \str\_case:nnn ..... 29762, 29764  
 \str\_case:nnTF ..  
     . 55, 589, 5229, 5234, 5239, 10128,  
     12398, 21913, 27303, 29763, 29765  
 \str\_case\_e:nn ..  
     .. 55, 5229, 5690, 5691, 28508, 28747  
 \str\_case\_e:nnTF ..  
     . 55, 5229, 5265, 5270, 5692, 5693,  
     5694, 5695, 5696, 5697, 21082, 29767  
 \str\_case\_x:nn .. 5690  
 \str\_case\_x:nnn ..... 29766  
 \str\_case\_x:nnTF ..... 5690, 5693, 5695  
 \str\_clear:N ..  
     ... 52, 52, 5060, 11149, 11256, 11268  
 \str\_clear\_new:N ..... 52, 5060

```

\str_concat:NNN ..... 52, 5060
\str_const:Nn ..... . 51, 5087, 5671, 5672, 5673, 5674,  

   5675, 5676, 5677, 5678, 5679, 5680,  

   5681, 5682, 7689, 7693, 7718, 7737,  

   26466, 26598, 26605, 26609, 26613
\str_count:N ..... . . . . . 57, 5561, 9467, 9468, 9663,  

   9664, 9685, 9686, 10698, 10776, 20297
\str_count:n ..... 57, 5561, 20291
\str_count_ignore_spaces:n ..... . . . . . 57, 418, 5561, 19912
\str_count_spaces:N ..... 57, 5541
\str_count_spaces:n 57, 418, 5541, 5567
\str_fold_case:n ..... . . . . . 60, 61, 254, 259, 5629, 14271
\str_gclear:N ..... 52, 5060
\str_gclear_new:N ..... 5060
\str_gconcat:NNN ..... 52, 5060
\str_gput_left:Nn ..... 52, 5087
\str_gput_right:Nn ..... 53, 5087
\str_gremove_all:Nn ..... 53, 5157
\str_gremove_once:Nn ..... 53, 5151
\str_greplace_all:Nnn 53, 5111, 5160
\str_greplace_once:Nnn 53, 5111, 5154
\str_gset:Nn ..... . . . . . 52, 5087, 11021, 11237, 11238, 11239
\str_gset_eq:NN ..... . . . . . 52, 5060, 11026, 11220, 11221, 11222
\str_head:N ..... 58, 419, 5599
\str_head:n ..... . . . . . 58, 398, 419, 4797, 4842, 5599
\str_head_ignore_spaces:n .. 58, 5599
\str_if_empty:NTF ..... 54, 5163, 10416, 11107, 11151, 11155,  

   11178, 11191, 11250, 26138, 26144
\str_if_empty_p:N ..... 54, 5163
\str_if_eq:ee ..... 529
\str_if_eq:eeTF ..... 54, 512,  

   3002, 5193, 5284, 5700, 5701, 5702,  

   5703, 5704, 5705, 8771, 9191, 9276,  

   11813, 23429, 23559, 25648, 25652
\str_if_eq:NN ..... 410
\str_if_eq:nn ..... 132, 521
\str_if_eq:NNTF ..... 54, 5207
\str_if_eq:nnTF ..... . . . . . 54, 55, 55, 134, 249, 431, 4485,  

   5193, 5256, 5994, 8715, 9114, 9804,  

   9847, 12201, 12219, 12288, 12731,  

   14140, 14214, 20677, 20699, 20708,  

   25629, 27069, 27088, 27106, 27116,  

   27129, 28097, 28100, 28103, 28106
\str_if_eq_p:ee .. 54, 5193, 5698, 5699
\str_if_eq_p:NN ..... 54, 5207
\str_if_eq_p:nn ..... . . . . . 54, 5193, 7735, 7745, 7747, 26617
\str_if_eq_x:nnTF ... 5690, 5701, 5703
\str_if_eq_x_p:nn ..... 5690
\str_if_exist:NTF ..... 53, 5163
\str_if_exist_p:N ..... 53, 5163
\str_if_in:Nntf ..... 54, 5215
\str_if_in:nntf .. 54, 4033, 4047, 5215
\str_item:Nn ..... . . . . . 58, 5403
\str_item:nn ..... 58, 414, 418, 5403
\str_item_ignore_spaces:nn ..... . . . . . 58, 414, 5403
\str_lower_case:n ..... 60, 254, 5629
\str_map_break: ..... 56, 5290
\str_map_break:n ... 56, 57, 4037, 5290
\str_map_function:NN ..... . . . . . 55, 55, 56, 56, 5290
\str_map_function:nNn 55, 55, 412, 5290
\str_map_inline:Nn .. 56, 56, 56, 5290
\str_map_inline:nn ..... . . . . . 56, 4031, 5290, 22379
\str_map_variable:NNn ..... 56, 5290
\str_map_variable:nNn ..... 56, 5290
\str_new:N ..... . . . . . 51, 52, 5060,  

   5683, 5684, 5685, 5686, 9355, 9356,  

   10412, 10606, 11014, 11015, 11016,  

   11060, 11061, 11062, 11063, 11064
\str_put_left:Nn ... 52, 5087, 11253
\str_put_right:Nn ... 53, 5087, 11160
\str_range:Nnn ..... . . . . . 59, 5464
\str_range:nnn .. 59, 418, 5464, 20294
\str_range_ignore_spaces:nnn 59, 5464
\str_remove_all:Nn ..... 53, 53, 5157
\str_remove_once:Nn ..... 53, 5151
\str_replace_all:Nnn .. 53, 5111, 5158
\str_replace_once:Nnn 53, 5111, 5152
\str_set:Nn ..... . . . . . 52, 53,  

   5087, 5349, 9443, 9444, 9651, 9652,  

   9673, 9674, 11081, 11085, 11094,  

   11250, 11265, 11272, 26087, 26098
\str_set_eq:NN 52, 5060, 11163, 11255
\str_show:N ..... . . . . . 61, 5687
\str_show:n ..... . . . . . 61, 5687
\str_tail:N ..... . . . . . 58, 5614
\str_tail:n ..... 58, 857, 5614, 11272
\str_tail_ignore_spaces:n .. 58, 5614
\str_upper_case:n ..... 60, 254, 5629
\str_use:N ..... . . . . . 57, 5060
\c_tilde_str ..... . . . . . 62, 5671
\g_tmpa_str ..... . . . . . 62, 5683
\l_tmpa_str ..... . . . . . 53, 62, 5683
\g_tmpb_str ..... . . . . . 62, 5683
\l_tmpb_str ..... . . . . . 62, 5683
\c_underscore_str ..... . . . . . 62, 5671

```

str internal commands:  
 \\_\_str\_case:nNTF ..... 5229  
 \\_\_str\_case:nw ..... 5229  
 \\_\_str\_case\_e:nnTF ..... 5229  
 \\_\_str\_case\_e:nw ..... 5229  
 \\_\_str\_case\_end:nw ..... 5229  
 \\_\_str\_change\_case:nn ..... 5629  
 \\_\_str\_change\_case\_aux:nn ..... 5629  
 \\_\_str\_change\_case\_char:nN ... 5629  
 \\_\_str\_change\_case\_end:nw ..... 5629  
 \\_\_str\_change\_case\_end:wn 5648, 5666  
 \\_\_str\_change\_case\_loop:nw ... 5629  
 \\_\_str\_change\_case\_output:nw .. 5629  
 \\_\_str\_change\_case\_result:n .. 5629  
 \\_\_str\_change\_case\_space:n ... 5629  
 \\_\_str\_collect\_delimit\_by\_q\_-  
     stop:w ..... 5492, 5515  
 \\_\_str\_collect\_end:nnnnnnnnw ...  
     ..... 417, 5515  
 \\_\_str\_collect\_end:wn ..... 5515  
 \\_\_str\_collect\_loop:wn ..... 5515  
 \\_\_str\_collect\_loop:wnNNNNNNNN . 5515  
 \\_\_str\_count:n 418, 5419, 5479, 5561  
 \\_\_str\_count\_aux:n ..... 5561  
 \\_\_str\_count\_loop:NNNNNNNNNN .. 5561  
 \\_\_str\_count\_spaces\_loop:w ... 5541  
 \\_\_str\_escape:n ..... 5171  
 \\_\_str\_head:w ..... 419, 5599  
 \\_\_str\_if\_eq:nn 5171, 5196, 5204, 5210  
 \\_\_str\_item:nn ..... 414, 5403  
 \\_\_str\_item:w ..... 414, 5403  
 \\_\_str\_map\_function:Nn ..... 411, 5290  
 \\_\_str\_map\_function:w ..... 411, 5290  
 \\_\_str\_map\_inline>NN ..... 5290  
 \\_\_str\_map\_variable:NnN ..... 5290  
 \\_\_str\_range:nnn ..... 5464  
 \\_\_str\_range:nnw ..... 5464  
 \\_\_str\_range:w ..... 5464  
 \\_\_str\_range\_normalize:nn .....  
     ..... 5487, 5488, 5496  
 \\_\_str\_replace>NNNnn ..... 5111  
 \\_\_str\_replace\_aux:NNNnnn ..... 5111  
 \\_\_str\_replace\_next:w ..... 5111  
 \\_\_str\_skip\_end:NNNNNNNNN .. 415, 5443  
 \\_\_str\_skip\_end:w ..... 5443  
 \\_\_str\_skip\_exp\_end:w .....  
     ... 415, 417, 5430, 5439, 5443, 5494  
 \\_\_str\_skip\_loop:wNNNNNNNNN ... 5443  
 \\_\_str\_tail\_auxi:w ..... 5614  
 \\_\_str\_tail\_auxii:w ..... 420, 5614  
 \\_\_str\_tmp:n .....  
     .. 5061, 5067, 5070, 5088, 5098, 5101  
 \\_\_str\_to\_other\_end:w ..... 413, 5358  
 \\_\_str\_to\_other\_fast\_end:w ... 5381  
 \\_\_str\_to\_other\_fast\_loop:w .....  
     ..... 5383, 5392, 5399  
 \\_\_str\_to\_other\_loop:w .... 413, 5358  
 \strcmp ..... 40  
 \string ..... 559  
 \suppressfontnotfounderror ... 819, 1653  
 \suppressifcsnameerror ..... 1000, 1800  
 \suppresslongerror ..... 1001, 1802  
 \suppressmathparerror ..... 1002, 1803  
 \suppressoutererror ..... 1003, 1805  
 \suppressprimitiveerror ..... 1004, 1806  
 \syntex ..... 801, 1615  
 sys commands:  
     \c\_sys\_day\_int ..... 105, 7695  
     \c\_sys\_engine\_str ... 105, 7718, 26468  
     \c\_sys\_engine\_version\_str 252, 26466  
     \sys\_gset\_rand\_seed:n 201, 252, 26528  
     \c\_sys\_hour\_int ..... 105, 7695  
     \sys\_if\_engine\_luatex:TF 105, 238,  
         7718, 8507, 8509, 13552, 25359,  
         25548, 26093, 26542, 26596, 26625,  
         26627, 26640, 29735, 29737, 29739  
     \sys\_if\_engine\_luatex\_p: ... 105,  
         7718, 10565, 25542, 26767, 27018,  
         27047, 27229, 27414, 27456, 29733  
     \sys\_if\_engine\_pdftex:TF .....  
         ... 105, 7718, 29743, 29745, 29747  
     \sys\_if\_engine\_pdftex\_p: .....  
         ... 105, 7718, 27496, 29741  
     \sys\_if\_engine\_ptex:TF ... 105, 7718  
     \sys\_if\_engine\_ptex\_p: .....  
         ... 105, 7718, 19930, 27497  
     \sys\_if\_engine\_uptex:TF ... 105, 7718  
     \sys\_if\_engine\_uptex\_p: .....  
         ... 105, 7718, 19931, 27497  
     \sys\_if\_engine\_xetex:TF ..... 4,  
         105, 7718, 8508, 29781, 29783, 29785  
     \sys\_if\_engine\_xetex\_p: .....  
         .. 105, 7718, 25542, 26767, 27019,  
           27048, 27230, 27415, 27457, 29779  
     \sys\_if\_output\_dvi:TF ... 106, 7737  
     \sys\_if\_output\_dvi\_p: ... 106, 7737  
     \sys\_if\_output\_pdf:TF ... 106, 7737  
     \sys\_if\_output\_pdf\_p: ... 106, 7737  
     \sys\_if\_platform\_unix:TF . 253, 26614  
     \sys\_if\_platform\_unix\_p: . 253, 26614  
     \sys\_if\_platform\_windows:TF .....  
         ..... 253, 26614  
     \sys\_if\_platform\_windows\_p: .....  
         ... 253, 26614  
     \sys\_if\_rand\_exist:TF .....  
         ..... 252, 483, 7748,  
         13109, 18992, 19016, 26517, 26528  
     \sys\_if\_rand\_exist\_p: ... 252, 7748

\sys\_if\_shell: ..... 253  
 \sys\_if\_shell:TF ... 253, 26560, 26619  
 \sys\_if\_shell\_p: ..... 253, 26619  
 \sys\_if\_shell\_restricted:TF ...  
       ..... 253, 26619  
 \sys\_if\_shell\_restricted\_p: ...  
       ..... 253, 26619  
 \sys\_if\_shell\_unrestricted:TF ...  
       ..... 253, 26619  
 \sys\_if\_shell\_unrestricted\_p: ...  
       ..... 253, 26619  
 \c\_sys\_jobname\_str .....  
       ..... 105, 151, 7685, 29693  
 \c\_sys\_minute\_int ..... 105, 7695  
 \c\_sys\_month\_int ..... 105, 7695  
 \c\_sys\_output\_str ..... 106, 7737  
 \c\_sys\_platform\_str 253, 26596, 26617  
 \sys\_rand\_seed: . 201, 251, 252, 26517  
 \c\_sys\_shell\_escape\_int ...  
       .... 253, 26540, 26620, 26622, 26624  
 \sys\_shell\_get:nnN .. 252, 252, 26553  
 \sys\_shell\_get:nnNTF 252, 26553, 26555  
 \sys\_shell\_now:n ..... 253, 26627  
 \sys\_shell\_shipout:n .... 253, 26640  
 \c\_sys\_year\_int ..... 105, 7695

sys internal commands:  
 \\_\_sys\_const:nn 7702, 7734, 7744,  
       7746, 7748, 26616, 26619, 26621, 26623  
 \\_\_sys\_get:nnN ..... 26553  
 \\_\_sys\_get\_do:Nw ..... 26553  
 \I\_\_sys\_internal\_tl ..... 26551  
 \c\_sys\_marker\_tl 26552, 26576, 26588  
 \c\_sys\_shell\_stream\_int ...  
       ..... 26625, 26637, 26650

syst commands:  
 \c\_syst\_last\_allocated\_toks .. 19524

**T**

\t ..... 27699  
 \tabskip ..... 560  
 \tagcode ..... 802, 1616  
 tan ..... 198  
 tand ..... 199  
 \tate ..... 1244, 2026  
 \tbaseline shift ..... 1245, 2027  
 \temp . 170, 176, 181, 184, 185, 192, 197, 200  
 T<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X 2<sub>E</sub> commands:  
 \@ ..... 5672  
 \@@end ..... 294, 1264, 1265  
 \@@hyph ..... 1268  
 \@@input ..... 1269  
 \@@italiccorr ..... 1270  
 \@@shipout ..... 1272, 1273  
 \@@tracingfonts ..... 295

\@@underline ..... 1271  
 \@addtofilelist ..... 11202  
 \@currname ..... 576, 11025, 11026  
 \@currnamestack ..... 11048, 11049  
 \@filelist ..... 153, 577, 581,  
       583, 583, 11201, 11286, 11289, 11303  
 \@firstoftwo ..... 319  
 \@ifpackageloaded ..... 28029, 28087  
 \@secondoftwo ..... 319  
 \@tempa .... 150, 152, 1280, 1294, 1297  
 \@tfor ..... 294, 1280  
 \@unexpandable@protect ..... 676  
 \AtBeginDocument ..... 294  
 \botmark ..... 514  
 \box ..... 228  
 \char ..... 131  
 \chardef ..... 125, 125, 446, 470  
 \conditionally@traceoff ..... 569  
 \copy ..... 221  
 \count ..... 131, 839  
 \cr ..... 480  
 \CROP@shipout ..... 1281  
 \csname ..... 16  
 \cstring ..... 336  
 \current@color .....  
       262, 1076, 28033, 28038, 28043, 28092  
 \currentgrouplevel ..... 347, 1030  
 \currentgroupype ..... 347, 1030  
 \def ..... 131  
 \detokenize ..... 42  
 \dimen ..... 512  
 \dimendef ..... 512  
 \directlua ..... 238  
 \dp ..... 222, 677, 678  
 \dup@shipout ..... 1282  
 \e@alloc@top ..... 839, 19510  
 \edef ..... 1, 375  
 \endcsname ..... 16  
 \endinput ..... 141  
 \endlinechar ..... 38, 38, 146, 381, 514  
 \endtemplate ..... 104, 480  
 \errhelp ..... 534, 534  
 \errmessage ..... 534, 534, 535  
 \errorcontextlines 312, 405, 535, 966  
 \escapechar ..... 42, 335, 347, 567  
 \everypar ..... 242, 366, 1016  
 \expandafter ..... 30, 32  
 \expanded ..... 18, 25, 27, 350, 353, 360, 366  
 \fi ..... 130  
 \firstmark ..... 367, 514  
 \font ..... 130  
 \fontdimen .....  
       ... 183, 219, 629, 630, 631, 631, 632  
 \frozen@everydisplay ..... 1266

\frozen@everymath . . . . .	1267	\nullfont . . . . .	514
\futurelet . . . . .		\number . . . . .	91, 729
... 480, 516, 518, 854, 856, 857, 857, 857		\numexpr . . . . .	364
\global . . . . .	275	\open@shipout . . . . .	1286
\GPTorg@shipout . . . . .	1283	\or . . . . .	91
\halign . . . . .	104, 480, 507, 1016	\outer . . . . .	131, 271, 558, 563, 1072, 1117
\hskip . . . . .	165	\parindent . . . . .	242
\ht . . . . .	222, 677, 678	\pdfmapfile . . . . .	296
\hyphen . . . . .	514	\pdfmapline . . . . .	296
\hyphenchar . . . . .	629	\pdfstrcmp . . . . .	xiii, 268, 269, 271, 286, 1012
\ifcase . . . . .	91	\pdfuniformdeviate . . . . .	201
\ifdim . . . . .	168	\pgfpages@originalshipout . . . . .	1287
\ifeof . . . . .	151	\pgfsys@... . . . . .	262
\iffalse . . . . .	97	\pi . . . . .	194
\ifhbox . . . . .	231	\pr@shipout . . . . .	1288
\ifnum . . . . .	91	\primitive . . . . .	294, 361, 362
\ifodd . . . . .	92, 1071	\protect . . . . .	569, 675, 676, 1053
\iftrue . . . . .	97	\protected . . . . .	131, 361
\ifvbox . . . . .	231	\ProvidesClass . . . . .	6
\ifvoid . . . . .	231	\ProvidesFile . . . . .	6
\ifx . . . . .	21, 271	\ProvidesPackage . . . . .	6
\indent . . . . .	1016	\quitvmode . . . . .	1016
\infty . . . . .	194	\read . . . . .	146, 561
\input@path . . . . .	152, 11143, 11145	\readline . . . . .	146, 561
\italiccorr . . . . .	514	\relax . . . . .	20, 130, 271, 331,
\jobname . . . . .	105	337, 347, 467, 467, 635, 637, 661, 691	
\l@expl@check@declarations@bool . . . . .	2309	\RequirePackage . . . . .	6, 271, 576
\l@expl@enable@debug@bool . . . . .	2181	\reserveinserts . . . . .	271
\l@expl@log@functions@bool . . . . .	2386	\robustify . . . . .	255
\lastnamedcs . . . . .	338	\romannumeral . . . . .	33, 635
\lccode . . . . .	272, 465, 856, 860, 1013	\scantokens . . . . .	380, 381, 578
\leavevmode . . . . .	242	\sfcode . . . . .	272
\let . . . . .	275	\shipout . . . . .	294
\letcharcode . . . . .	503	\show . . . . .	15, 49, 347
\LL@shipout . . . . .	1284	\showbox . . . . .	966
\loctoks . . . . .	839	\showthe . . . . .	347, 464, 594, 599, 602
\long . . . . .	131, 361	\showtokens . . . . .	49, 405, 540
\lower . . . . .	1017	\sin . . . . .	194
\lowercase . . . . .	943, 944, 945	\skip . . . . .	861, 862
\luaescapestring . . . . .	238	\space . . . . .	514
\makeatletter . . . . .	6	\special . . . . .	1075
\mathchar . . . . .	131	\splitbotmark . . . . .	514
\mathchardef . . . . .	125, 446	\splitfirstmark . . . . .	514
\meaning . . . . .	14,	\strcmp . . . . .	268, 286
122, 130, 131, 512, 512, 518, 855, 1072		\string . . . . .	122, 856, 857, 858
\mem@oldshipout . . . . .	1285	\tenrm . . . . .	130
\message . . . . .	25	\tex_lowercase:D . . . . .	506
\newif . . . . .	97	\tex_mdfivesum:D . . . . .	1027
\newlinechar . . . . .	38,	\tex_unexpanded:D . . . . .	359
38, 312, 339, 381, 405, 535, 565, 566		\the . . . . .	82,
\newread . . . . .	558	130, 160, 164, 167, 351, 361, 362, 364	
\newtoks . . . . .	206, 851, 869	\toks . . . . .	xxi, 91, 206, 251, 364, 365,
\newwrite . . . . .	563	366, 838, 839, 839, 839, 840, 841,	
\noexpand . . . . .	31, 130, 361, 362, 362, 363	841, 842, 843, 844, 845, 846, 846,	

851, 855, 857, 859, 860, 862, 864,  
 869, 870, 870, 870, 871, 872, 876,  
 914, 916, 921, 922, 926, 926, 927,  
 927, 928, 931, 934, 936, 944, 961, 1035  
`\toks@` ..... 366  
`\toksdef` ..... 851  
`\topmark` ..... 131, 514  
`\tracingfonts` ..... 295  
`\tracingnesting` ..... 380, 578  
`\tracingonline` ..... 966  
`\typeout` ..... 569  
`\uccode` ..... 1013  
`\Ucharcat` ..... 505, 506  
`\ucharcat@table` ..... 52, 55  
`\unexpanded` ..... 31, 43, 43,  
 44, 47, 47, 48, 70, 70, 74, 75, 110,  
 113, 113, 115, 115, 134, 249, 254,  
 258, 361, 362, 363, 375, 397, 398, 450  
`\unhbox` ..... 228  
`\unhcopy` ..... 225  
`\uniformdeviate` ..... 201  
`\unless` ..... 20  
`\unvbox` ..... 228  
`\unvcopy` ..... 227  
`\uppercase` ..... 943  
`\usepackage` ..... 576  
`\valign` ..... 480  
`\verso@orig@shipout` ..... 1290  
`\vskip` ..... 165  
`\vtop` ..... 985  
`\wd` ..... 222, 677, 678  
`\write` ..... 149, 562, 565

tex commands:

`\tex_above:D` ..... 293  
`\tex_abovedisplayshortskip:D` ..... 294  
`\tex_abovedisplayskip:D` ..... 295  
`\tex_abovewithdelims:D` ..... 296  
`\tex Accent:D` ..... 297  
`\tex adjdemerits:D` ..... 298  
`\tex adjustspacing:D` ..... 753, 1012  
`\tex advance:D` ..... 299, 6563, 6566,  
 6569, 6572, 6579, 6582, 6585, 6588,  
 11412, 11416, 11423, 11427, 11797,  
 11800, 11805, 11808, 11941, 11944,  
 11949, 11952, 19627, 19634, 19637,  
 20039, 20041, 20074, 20076, 21289  
`\tex afterassignment:D` .....  
 ..... 300, 8878, 19980, 20023  
`\tex aftergroup:D` ..... 301, 2083  
`\tex alignmark:D` ..... 889, 1313  
`\tex aligntab:D` ..... 890, 1314  
`\tex atop:D` ..... 302  
`\tex atopwithdelims:D` ..... 303  
`\tex attribute:D` ..... 891, 1315

`\tex_attributedef:D` ..... 892, 1316  
`\tex automaticdiscretionary:D` ..... 894  
`\tex automatichyphenmode:D` ..... 895  
`\tex automatichyphenpenalty:D` ..... 897  
`\tex autospacing:D` ..... 1211  
`\tex autoxspacing:D` ..... 1212  
`\tex badness:D` ..... 304  
`\tex baselineskip:D` ..... 305  
`\tex batchmode:D` ..... 306  
`\tex begincsname:D` ..... 898  
`\tex beginingroup:D` ..... 307, 1275, 1414, 2078  
`\tex beginL:D` ..... 615  
`\tex beginR:D` ..... 616  
`\tex belowdisplayshortskip:D` ..... 308  
`\tex belowdisplayskip:D` ..... 309  
`\tex binoppenalty:D` ..... 310  
`\tex bodydir:D` ..... 899, 1352, 1404  
`\tex bodydirection:D` ..... 900  
`\tex botmark:D` ..... 311  
`\tex botmarks:D` ..... 617  
`\tex box:D` .....  
 ..... 312, 23645, 23648, 23688, 24315, 24318  
`\tex boxdir:D` ..... 901, 1353  
`\tex boxdirection:D` ..... 902  
`\tex boxmaxdepth:D` ..... 313  
`\tex breakafterdirmode:D` ..... 903  
`\tex brokenpenalty:D` ..... 314  
`\tex catcode:D` .....  
 ..... 315, 3514, 8344, 8346, 25555, 25562  
`\tex catcodetable:D` ..... 904, 1317  
`\tex char:D` ..... 316  
`\tex chardef:D` ..... 317, 317,  
 2071, 2100, 2102, 2712, 2713, 6525,  
 7408, 7414, 8802, 10456, 10627, 26230  
`\tex cleaders:D` ..... 318  
`\tex clearmarks:D` ..... 905, 1318  
`\tex closein:D` ..... 319, 10464  
`\tex closeout:D` ..... 320, 10636  
`\tex clubpenalties:D` ..... 618  
`\tex clubpenalty:D` ..... 321  
`\tex copy:D` ..... 322, 23637,  
 23640, 23663, 23672, 23681, 23689  
`\tex copyfont:D` ..... 754, 1013  
`\tex count:D` .....  
 ..... 323, 10395, 10397, 10588,  
 10590, 19493, 19509, 19517, 19518  
`\tex countdef:D` ..... 324  
`\tex cr:D` ..... 325  
`\tex crampeddisplaystyle:D` ..... 906, 1319  
`\tex crampedscriptscriptstyle:D` .....  
 ..... 908, 1320  
`\tex crampedscriptstyle:D` ..... 909, 1322  
`\tex crampedtextstyle:D` ..... 910, 1323  
`\tex crcr:D` ..... 326

```

\tex_creationdate:D ..... 879
\tex_csname:D ..... 327, 2065
\tex_cssstring:D ..... 911
\tex_currentgrouplevel:D ..... 619
\tex_currentgroupype:D ..... 620
\tex_currentifbranch:D ..... 621
\tex_currentiflevel:D ..... 622
\tex_currentiftype:D ..... 623
\tex_day:D ..... 328, 7699
\tex_deadcycles:D ..... 329
\tex_def:D ..... 330, 805, 806,
               807, 1415, 1416, 1417, 2084, 2086,
               2088, 2089, 2106, 2108, 2109, 2110,
               2112, 2113, 2114, 2116, 2117, 2118
\tex_defaulthyphenchar:D ..... 331
\tex_defaultskewchar:D ..... 332
\tex_delcode:D ..... 333
\tex_delimiter:D ..... 334
\tex_delimiterfactor:D ..... 335
\tex_delimitershortfall:D ..... 336
\tex_detokenize:D .... 624, 2074, 2076
\tex_dimen:D ..... 337
\tex_dimendef:D ..... 338
\tex_dimexpr:D .... 625, 11352, 23605
\tex_directlua:D ..... 912,
               1306, 1307, 5175, 13556, 25348, 26544
\tex_disablecjktoken:D .....
               ..... 1251, 6520, 6523, 7725
\tex_discretionary:D ..... 339
\tex_displayindent:D ..... 340
\tex_displaylimits:D ..... 341
\tex_displaystyle:D ..... 342
\tex_displaywidowpenalties:D .. 626
\tex_displaywidowpenalty:D .... 343
\tex_displaywidth:D ..... 344
\tex_divide:D .... 345, 19486, 21290
\tex_doublehyphendemerits:D ... 346
\tex_dp:D ..... 347, 23656
\tex_draftmode:D ..... 755, 1014
\tex_dtou:D ..... 1213
\tex_dump:D ..... 348
\tex_dviextension:D ..... 913
\tex_dvifeedback:D ..... 914
\tex_dvivvariable:D ..... 915
\tex_eachlinedepth:D ..... 756
\tex_eachlineheight:D ..... 757
\tex_eodef:D .....
               .... 349, 1276, 1277, 1293, 2107,
               2111, 2115, 2119, 10848, 10906, 29674
\tex_efcode:D ..... 793
\tex_elapsedtime:D ..... 758, 880
\tex_else:D .....
               .... 350, 1279, 1305, 2052, 2103, 2144
\tex_emergencystretch:D ..... 351
\tex_enablecjktoken:D ..... 1252
\tex_end:D . 352, 1265, 1387, 2823, 9619
\tex_endcsname:D ..... 353, 2066
\tex_endgroup:D .....
               ..... 354, 1262, 1301, 2045, 2079
\tex_endinput:D .... 355, 9628, 26148
\tex_endL:D ..... 627
\tex_endlinechar:D .....
               ..... 257, 258, 272, 356, 4244,
               4245, 4246, 4279, 10507, 10509, 10510
\tex_endR:D ..... 628
\tex_epTeXinputencoding:D .... 1214
\tex_epTeXversion:D 1215, 26485, 26507
\tex_eqno:D ..... 357
\tex_errhelp:D .... 358, 9483, 10333
\tex_errmessage:D .....
               ..... 359, 2815, 9503, 10364
\tex_errorcontextlines:D .... 360,
               5043, 9498, 9518, 9727, 10359, 23754
\tex_errorstopmode:D ..... 361
\tex_escapechar:D .....
               ..... 362, 3131, 10710, 10757,
               10763, 19945, 20007, 20008, 20332
\tex_eTeXrevision:D ..... 629
\tex_eTeXversion:D ..... 630
\tex_etoksapp:D ..... 916
\tex_etokspre:D ..... 917
\tex_euc:D ..... 1216
\tex_everycr:D ..... 363
\tex_everydisplay:D ..... 364, 1266
\tex_everyeof:D 631, 4243, 11121, 26576
\tex_everyhbox:D ..... 365
\tex_everyjob:D ..... 366, 1388,
               7686, 7688, 11018, 11020, 11054, 11056
\tex_everymath:D ..... 367, 1267
\tex_everypar:D ..... 368
\tex_everyvbox:D ..... 369
\tex_exceptionpenalty:D ..... 918
\tex_exhyphenpenalty:D ..... 370
\tex_expandafter:D ..... 371,
               810, 1280, 1294, 1296, 1297, 1420, 2067
\tex_expanded:D ..... 920,
               1397, 2143, 2144, 3163, 3166, 3233,
               3236, 3269, 3275, 3400, 3403, 3424,
               3427, 3492, 3495, 3523, 8525, 10210
\tex_explicitdiscretionary:D ... 921
\tex_explicithyphenpenalty:D ... 919
\tex_fam:D ..... 372
\tex_fi:D ..... 373, 811,
               1274, 1298, 1300, 1309, 1310, 1311,
               1369, 1371, 1372, 1376, 1383, 1398,
               1406, 1411, 1421, 2053, 2105, 2146
\tex_filedump:D ..... 759, 881
\tex_filemoddate:D ..... 760, 882

```

\tex\_filesize:D ..... 761, 883, 26118  
\tex\_finalhyphendemerits:D ..... 374  
\tex\_firstlineheight:D ..... 762  
\tex\_firstmark:D ..... 375  
\tex\_firstmarks:D ..... 632  
\tex\_firstvalidlanguage:D ..... 922  
\tex\_floatingpenalty:D ..... 376  
\tex\_font:D ..... 377, 12953  
\tex\_fontchardp:D ..... 633  
\tex\_fontcharht:D ..... 634  
\tex\_fontcharic:D ..... 635  
\tex\_fontcharwd:D ..... 636  
\tex\_fontdimen:D ..... 378, 12942  
\tex\_fontexpand:D ..... 763, 1015  
\tex\_fontid:D ..... 923, 1324  
\tex\_fontname:D ..... 379  
\tex\_fontsize:D ..... 764  
\tex\_forcecjktoken:D ..... 1253  
\tex\_formatname:D ..... 924, 1325  
\tex\_futurelet:D .....  
..... 380, 8873, 8875, 19953, 20011  
\tex\_gdef:D 381, 2120, 2123, 2127, 2131  
\tex\_gleaders:D ..... 930, 1326  
\tex\_global:D ..... 278, 283,  
285, 382, 586, 812, 814, 1296, 1422,  
2897, 2904, 6502, 6508, 6512, 6534,  
6547, 6569, 6572, 6585, 6588, 6598,  
7414, 8608, 8610, 8620, 8875, 10456,  
10627, 11366, 11373, 11399, 11408,  
11416, 11427, 11756, 11762, 11788,  
11793, 11800, 11808, 11890, 11898,  
11930, 11937, 11944, 11952, 12953,  
23640, 23648, 23720, 23775, 23789,  
23804, 23826, 23873, 23887, 23901,  
23916, 23939, 23956, 24318, 26230  
\tex\_globaldefs:D ..... 383  
\tex\_glueexpr:D ..... 637, 11780,  
11785, 11788, 11797, 11800, 11805,  
11808, 11823, 11829, 11836, 11838,  
11846, 11851, 11856, 11861, 18921  
\tex\_glueshrink:D ..... 638, 26457  
\tex\_glueshrinkorder:D ..... 639  
\tex\_gluestretch:D .....  
..... 640, 20165, 20171, 26456  
\tex\_gluestretchorder:D ..... 641  
\tex\_gluetomu:D ..... 642  
\tex\_halign:D ..... 384  
\tex\_hangafter:D ..... 385  
\tex\_hangindent:D ..... 386  
\tex\_hbadness:D ..... 387  
\tex\_hbox:D ..... 388, 23765, 23769,  
23775, 23783, 23789, 23797, 23804,  
23819, 23826, 23834, 23839, 25044  
\tex\_hfil:D ..... 389  
\tex\_hfill:D ..... 390  
\tex\_hfilneg:D ..... 391  
\tex\_hfuzz:D ..... 392  
\tex\_hjcode:D ..... 925  
\tex\_hoffset:D ..... 393, 1400  
\tex\_holdinginserts:D ..... 394  
\tex\_hpack:D ..... 926  
\tex\_hrule:D ..... 395  
\tex\_hsize:D ..... 396, 24462,  
24464, 24465, 24531, 24533, 24534  
\tex\_hskip:D ..... 397, 11842  
\tex\_hss:D .....  
..... 398, 23843, 23845, 24300, 24309  
\tex\_ht:D ..... 399, 23655  
\tex\_hyphen:D ..... 292, 1268  
\tex\_hyphenation:D ..... 400  
\tex\_hyphenationbounds:D ..... 927  
\tex\_hyphenationmin:D ..... 928  
\tex\_hyphenchar:D ..... 401, 12943  
\tex\_hyphenpenalty:D ..... 402  
\tex\_hyphenpenaltymode:D ..... 929  
\tex\_if:D ..... 117, 403, 2055, 2056  
\tex\_ifabsdim:D ..... 750, 1016  
\tex\_ifabsnum:D 751, 1017, 13011, 13015  
\tex\_ifcase:D ..... 404, 6386  
\tex\_ifcat:D ..... 405, 2057  
\tex\_ifcondition:D ..... 931  
\tex\_ifcsname:D ..... 643, 2064  
\tex\_ifdbox:D ..... 1217  
\tex\_ifddir:D ..... 1218  
\tex\_ifdefined:D .....  
..... 644, 809, 1264, 1272, 1303,  
1306, 1312, 1371, 1372, 1379, 1386,  
1399, 1407, 1419, 2063, 2101, 2144  
\tex\_ifdim:D ..... 406, 11351  
\tex\_ifeof:D ..... 407, 10484  
\tex\_iffalse:D ..... 408, 2050  
\tex\_iffontchar:D ..... 645  
\tex\_ifhbox:D ..... 409, 23700  
\tex\_ifhmode:D ..... 410, 2060  
\tex\_ifincsname:D ..... 794  
\tex\_ifinner:D ..... 411, 2062  
\tex\_ifmdir:D ..... 1219  
\tex\_ifmmode:D ..... 412, 2059  
\tex\_ifnum:D ..... 413, 1370, 2081  
\tex\_ifodd:D .....  
..... 414, 2181, 2309, 2386, 6385, 7379, 7380  
\tex\_ifprimitive:D ..... 752, 885  
\tex\_iftbox:D ..... 1220  
\tex\_iftdir:D ..... 1221  
\tex\_iftrue:D ..... 415, 2049  
\tex\_ifvbox:D ..... 416, 23701  
\tex\_ifvmode:D ..... 417, 2061  
\tex\_ifvoid:D ..... 418, 23702

\tex\_ifx:D ..... 419, 1278, 1295, 2058  
 \tex\_ifybox:D ..... 1222  
 \tex\_ifydir:D ..... 1223  
 \tex\_ignoredimensions:D ..... 765  
 \tex\_ignoreligaturesinfont:D . 1018  
 \tex\_ignorespaces:D ..... 420  
 \tex\_immediate:D . 421, 2832, 2834,  
     10629, 10636, 10677, 28683, 28744  
 \tex\_immediateassigned:D ..... 932  
 \tex\_immediateassignment:D .... 933  
 \tex\_indent:D ..... 422, 25684  
 \tex\_inhibitglue:D ..... 1224  
 \tex\_inhibitxspcode:D ..... 1225  
 \tex\_initcatcodetable:D ... 934, 1327  
 \tex\_input:D .....  
     . 423, 1269, 1389, 11126, 11206, 26581  
 \tex\_inputlineno:D ... 424, 2830, 9434  
 \tex\_insert:D ..... 425  
 \tex\_insertht:D ..... 766, 1019  
 \tex\_insertpenalties:D ..... 426  
 \tex\_interactionmode:D .....  
     . 646, 23738, 23741, 23743  
 \tex\_interlinepenalties:D ..... 647  
 \tex\_interlinepenalty:D ..... 427  
 \tex\_italiccorrection:D .....  
     . 291, 1270, 1401  
 \tex\_jcharwidowpenalty:D ..... 1226  
 \tex\_jfam:D ..... 1227  
 \tex\_jfont:D ..... 1228  
 \tex\_jis:D ..... 1229, 6521, 7726  
 \tex\_jobname:D .....  
     . 428, 7689, 7693, 11021, 11039, 11040  
 \tex\_kanjiskip:D ..... 1230, 7722  
 \tex\_kansuji:D ..... 1231  
 \tex\_kansujichar:D ..... 1232  
 \tex\_kcatcode:D ..... 1233  
 \tex\_kchar:D ..... 1254  
 \tex\_kchardef:D ..... 1255, 6524  
 \tex\_kern:D .....  
     . 429, 24008, 24298, 24307, 24878,  
     24883, 24965, 24966, 25258, 25259,  
     25704, 25706, 25755, 25757, 25832  
 \tex\_kuten:D ..... 1234, 1256  
 \tex\_language:D ..... 430, 1390  
 \tex\_lastbox:D ... 431, 23717, 23720  
 \tex\_lastkern:D ..... 432  
 \tex\_lastlinedepth:D ..... 767  
 \tex\_lastlinefit:D ..... 648  
 \tex\_lastnameedcs:D ..... 935  
 \tex\_lastnodestype:D ..... 649  
 \tex\_lastpenalty:D ..... 433  
 \tex\_lastskip:D ..... 434  
 \tex\_lastxpos:D ..... 768, 1026  
 \tex\_lastypos:D ..... 769, 1027  
 \tex\_latelua:D ..... 936, 1328, 25349  
 \tex\_lateluafunction:D ..... 937  
 \tex\_lccode:D ..... 435, 4130,  
     4131, 4132, 5364, 5365, 5387, 5388,  
     8420, 8422, 19926, 19936, 20005,  
     20007, 20010, 20040, 22893, 22945  
 \tex\_leaders:D ..... 436  
 \tex\_left:D ..... 437, 1408  
 \tex\_leftghost:D ..... 938, 1354  
 \tex\_lefthyphenmin:D ..... 438  
 \tex\_leftmarginkern:D ..... 795  
 \tex\_leftskip:D ..... 439  
 \tex\_leqno:D ..... 440  
 \tex\_let:D ..... 279, 283,  
     285, 326, 441, 812, 814, 1120, 1265,  
     1266, 1267, 1268, 1269, 1270, 1271,  
     1273, 1296, 1302, 1304, 1308, 1313,  
     1314, 1315, 1316, 1317, 1318, 1319,  
     1320, 1322, 1323, 1324, 1325, 1326,  
     1327, 1328, 1329, 1330, 1331, 1332,  
     1333, 1334, 1335, 1336, 1337, 1338,  
     1339, 1340, 1341, 1342, 1343, 1345,  
     1346, 1348, 1349, 1350, 1352, 1353,  
     1354, 1355, 1356, 1358, 1359, 1360,  
     1361, 1362, 1363, 1364, 1365, 1366,  
     1367, 1368, 1374, 1375, 1380, 1381,  
     1382, 1387, 1388, 1389, 1390, 1391,  
     1392, 1393, 1394, 1395, 1396, 1397,  
     1400, 1401, 1402, 1403, 1404, 1405,  
     1408, 1409, 1410, 1422, 2049, 2050,  
     2051, 2052, 2053, 2054, 2055, 2056,  
     2057, 2058, 2059, 2060, 2061, 2062,  
     2063, 2064, 2065, 2066, 2067, 2068,  
     2069, 2070, 2072, 2073, 2074, 2075,  
     2076, 2077, 2078, 2079, 2081, 2082,  
     2083, 2099, 2106, 2107, 2120, 2121,  
     2432, 2437, 2453, 2458, 2893, 8608,  
     8610, 8620, 19927, 19937, 29799, 29824  
 \tex\_letcharcode:D ..... 939  
 \tex\_letterspacefont:D ..... 796  
 \tex\_limits:D ..... 442  
 \tex\_linedir:D ..... 940  
 \tex\_linedirection:D ..... 941  
 \tex\_linepenalty:D ..... 443  
 \tex\_lineskip:D ..... 444  
 \tex\_lineskiplimit:D ..... 445  
 \tex\_localbrokenpenalty:D . 942, 1355  
 \tex\_localinterlinepenalty:D ...  
     . 943, 1356  
 \tex\_localleftbox:D ..... 948, 1358  
 \tex\_localrightbox:D ..... 949, 1359  
 \tex\_long:D .....  
     . 446, 805, 806, 807, 1415, 1416,  
     1417, 2084, 2086, 2089, 2108, 2109,

\tex_looseness:D . . . . .	447	465
\tex_lower:D . . . . .	448, 23699	969
\tex_lowercase:D . . . . .	449, 1071, 4133, 5366, 5389, 8451, 8571, 9490, 10345, 19927, 19937, 20006, 22894, 22946, 27903, 29773	970
\tex_lpcode:D . . . . .	797	466
\tex_luabytecode:D . . . . .	944	467
\tex_luabytecodecall:D . . . . .	945	26114
\tex_luacopyinputnodes:D . . . . .	946	\tex_meaning:D . . . . .
\tex_luadef:D . . . . .	947	468, 1277, 1294, 2072, 2073
\tex_luaescapestring:D . . . . .	409, 950, 1329, 5174, 13560, 13561, 25347	469
\tex_luafunction:D . . . . .	951, 1330	\tex_message:D . . . . .
\tex_luafunctioncall:D . . . . .	952	470
\tex_luatexbanner:D . . . . .	953	\tex_middle:D . . . . .
\tex_luatexrevision:D . . . . .	954, 26492	651, 1409
\tex_luatexversion:D . . . . .	955, 1372, 1379, 2101, 5172, 6519, 7246, 7720, 10566, 26490	471
\tex_mag:D . . . . .	450	\tex_month:D . . . . .
\tex_mapfile:D . . . . .	770, 1374	472, 1392, 7700
\tex_mapline:D . . . . .	771, 1375	\tex_moveleft:D . . . . .
\tex_mark:D . . . . .	451	473, 23693
\tex_marks:D . . . . .	650	\tex_moveright:D . . . . .
\tex_mathaccent:D . . . . .	452	474, 23695
\tex_mathbin:D . . . . .	453	\tex_msip:D . . . . .
\tex_mathchar:D . . . . .	454	475
\tex_mathchardef:D . . . . .	317, 455, 2104, 6528, 6529	\tex_muexpr:D . . . . .
\tex_mathchoice:D . . . . .	456	652, 11920, 11927, 11930, 11941, 11944, 11949, 11952, 11958, 11963
\tex_mathclose:D . . . . .	457	\tex_multiply:D . . . . .
\tex_mathcode:D . . . . .	458, 8414, 8416	476
\tex_mathdelimitersmode:D . . . . .	956	\tex_muskip:D . . . . .
\tex_mathdir:D . . . . .	957, 1360	477
\tex_mathdirection:D . . . . .	958	\tex_muskipdef:D . . . . .
\tex_mathdisplayskipmode:D . . . . .	959	478
\tex_matheqngapstep:D . . . . .	960	\tex_mutoglu:D . . . . .
\tex_mathinner:D . . . . .	459	310, 324, 653, 11920, 11959
\tex_mathnolimitsmode:D . . . . .	961	\tex_newlinechar:D . . . . .
\tex_mathop:D . . . . .	460, 1391	479, 2814, 4246, 4271, 4275, 5041, 9496, 9725, 10357, 10676
\tex_mathopen:D . . . . .	461	\tex_noalign:D . . . . .
\tex_mathoption:D . . . . .	962	480
\tex_mathord:D . . . . .	462	\tex_noautospanning:D . . . . .
\tex_mathpenaltiesmode:D . . . . .	963	1235
\tex_mathpunct:D . . . . .	463	\tex_noautoxspanning:D . . . . .
\tex_mathrel:D . . . . .	464	1236
\tex_mathrulesfam:D . . . . .	964	\tex_noboundary:D . . . . .
\tex_mathscriptboxmode:D . . . . .	966	481
\tex_mathscriptcharmode:D . . . . .	967	\tex_noexpand:D . . . . .
\tex_mathscriptsmode:D . . . . .	965	482, 2068
\tex_mathstyle:D . . . . .	968, 1331	\tex_noheight:D . . . . .
		971
		\tex_noindent:D . . . . .
		483
		\tex_nokerns:D . . . . .
		972, 1332
		\tex_noligatures:D . . . . .
		773
		\tex_noligs:D . . . . .
		973, 1333
		\tex_nolimits:D . . . . .
		484
		\tex_nonscript:D . . . . .
		485
		\tex_nonstopmode:D . . . . .
		486
		\tex_normaldeviate:D . . . . .
		774, 1028
		\tex_nospaces:D . . . . .
		974
		\tex_noheight:D . . . . .
		975
		\tex_nulldelimiterspace:D . . . . .
		487
		\tex_nullfont:D . . . . .
		488, 8831
		\tex_number:D . . . . .
		489, 6382, 24365, 25554
		\tex_numexpr:D . . . . .
		654, 6383, 13175, 20324
		\tex_omit:D . . . . .
		490
		\tex_openin:D . . . . .
		491, 10458
		\tex_openout:D . . . . .
		492, 10629
		\tex_or:D . . . . .
		493, 2051
		\tex_outer:D . . . . .
		494, 1393, 29674
		\tex_output:D . . . . .
		495
		\tex_outputbox:D . . . . .
		976, 1334

\tex\_outputpenalty:D ..... 496  
\tex\_over:D ..... 497, 1394  
\tex\_overfullrule:D ..... 498  
\tex\_overline:D ..... 499  
\tex\_overwithdelims:D ..... 500  
\tex\_pagebottomoffset:D .... 977, 1361  
\tex\_pagedepth:D ..... 501  
\tex\_pagedir:D ..... 978, 1362, 1405  
\tex\_pagedirection:D ..... 979  
\tex\_pagediscards:D ..... 655  
\tex\_pagefillstretch:D ..... 502  
\tex\_pagefillstretch:D ..... 503  
\tex\_pagegoal:D ..... 505  
\tex\_pageheight:D .... 775, 1030, 1363  
\tex\_pageleftoffset:D .... 980, 1335  
\tex\_pagerightoffset:D .... 981, 1364  
\tex\_pageshrink:D ..... 506  
\tex\_pagestretch:D ..... 507  
\tex\_pagetopoffset:D .... 982, 1336  
\tex\_pagetotal:D ..... 508  
\tex\_pagewidth:D .... 776, 1365  
\tex\_pagewidth:D ..... 1031  
\tex\_par:D ..... 509  
\tex\_pardir:D ..... 983, 1366  
\tex\_pardirection:D ..... 984  
\tex\_parfillskip:D ..... 510  
\tex\_parindent:D ..... 511  
\tex\_parshape:D ..... 512  
\tex\_parshapedimen:D ..... 656  
\tex\_parshapeindent:D ..... 657  
\tex\_parshapelenh:D ..... 658  
\tex\_parskip:D ..... 513  
\tex\_patterns:D ..... 514  
\tex\_pausing:D ..... 515  
\tex\_pdfannot:D ..... 681  
\tex\_pdfcatalog:D ..... 682  
\tex\_pdfcolorstack:D 684, 28160, 28169  
\tex\_pdfcolorstackinit:D ..... 685  
\tex\_pdfcompresslevel:D .....  
..... 683, 28773, 28774  
\tex\_pdfcreationdate:D ..... 686  
\tex\_pdfdecimaldigits:D ..... 687  
\tex\_pdfdest:D ..... 688  
\tex\_pdfdestmargin:D ..... 689  
\tex\_pdfendlink:D ..... 690  
\tex\_pdfendthread:D ..... 691  
\tex\_pdfextension:D ... 985, 28158,  
28159, 28167, 28168, 28561, 28562,  
28569, 28570, 28575, 28576, 28581,  
28582, 28729, 28732, 28765, 28766  
\tex\_pdffeedback:D ..... 986, 28733  
\tex\_pdffontattr:D ..... 692  
\tex\_pdffontname:D ..... 693  
\tex\_pdffontobjnum:D ..... 694  
\tex\_pdffgamma:D ..... 695  
\tex\_pdffgentounicode:D ..... 698  
\tex\_pdffglyptounicode:D ..... 699  
\tex\_pdfforigin:D ..... 700  
\tex\_pdfimageapplygamma:D ..... 696  
\tex\_pdfimagegamma:D ..... 697  
\tex\_pdfimagehicolor:D ..... 701  
\tex\_pdfimageresolution:D ..... 702  
\tex\_pdfincludchars:D ..... 703  
\tex\_pdfinclusioncopyfonts:D ... 704  
\tex\_pdfinclusionerrorlevel:D .. 706  
\tex\_pdffinfo:D ..... 707  
\tex\_pdflastannot:D ..... 708  
\tex\_pdflastlink:D ..... 709  
\tex\_pdflastobj:D ..... 710, 28735  
\tex\_pdflastxform:D ..... 711, 1021  
\tex\_pdflastximage:D .....  
..... 712, 1023, 28702, 28706  
\tex\_pdflastximagecolordepth:D .. 714  
\tex\_pdflastximagepages:D .. 715, 1025  
\tex\_pdflinkmargin:D ..... 716  
\tex\_pdfliteral:D ..... 717, 28563  
\tex\_pdfminorversion:D ..... 718  
\tex\_pdfnames:D ..... 719  
\tex\_pdfobj:D ..... 720, 28735, 28767  
\tex\_pdfobjcompresslevel:D .....  
..... 721, 28784, 28785  
\tex\_pdfoutline:D ..... 722  
\tex\_pdfoutput:D .... 723, 1029, 7740  
\tex\_pdfpageattr:D ..... 724  
\tex\_pdfpagebox:D ..... 725  
\tex\_pdfpageref:D ..... 726  
\tex\_pdfpageresources:D ..... 727  
\tex\_pdfpagesattr:D ..... 728  
\tex\_pdfrefobj:D ..... 729  
\tex\_pdfrefxform:D ..... 730, 1035  
\tex\_pdfrefximage:D .....  
..... 731, 1036, 28702, 28711  
\tex\_pdfrestore:D ..... 732, 28577  
\tex\_pdfretval:D ..... 733  
\tex\_pdfsave:D ..... 734, 28571  
\tex\_pdfsetmatrix:D ..... 735, 28583  
\tex\_pdfstartlink:D ..... 736  
\tex\_pdfstartthread:D ..... 737  
\tex\_pdfsuppressptexinfo:D ..... 738  
\tex\_pdftexbanner:D ..... 790, 1380  
\tex\_pdftexrevision:D 791, 1381, 26474  
\tex\_pdftexversion:D .....  
... 297, 792, 1371, 1382, 7721, 26472  
\tex\_pdfthread:D ..... 739  
\tex\_pdfthreadmargin:D ..... 740  
\tex\_pdftrailer:D ..... 741  
\tex\_pdfuniqueresname:D ..... 742

\tex\_pdfvariable:D . 987, 28775, 28786  
\tex\_pdfvorigin:D ..... 743  
\tex\_pdfxform:D ..... 744, 1038  
\tex\_pdxformattr:D ..... 745  
\tex\_pdxformname:D ..... 746  
\tex\_pdxformresources:D ..... 747  
\tex\_pdximage:D .... 748, 1039, 28683  
\tex\_pdximagebbox:D .... 749, 28677  
\tex\_penalty:D ..... 516  
\tex\_pkmode:D ..... 777  
\tex\_pkresolution:D ..... 778  
\tex\_postbreakpenalty:D ..... 1237  
\tex\_postdisplaypenalty:D ..... 517  
\tex\_postexhyphenchar:D ... 988, 1337  
\tex\_posthyphenchar:D .... 989, 1338  
\tex\_prebinoppenalty:D ..... 990  
\tex\_prebreakpenalty:D ..... 1238  
\tex\_predisplaydirection:D .... 659  
\tex\_predisplaygapfactor:D .... 991  
\tex\_predisplaypenalty:D ..... 518  
\tex\_predisplaysize:D ..... 519  
\tex\_preehyphenchar:D .... 992, 1339  
\tex\_prehyphenchar:D ..... 993, 1340  
\tex\_prerelpenalty:D ..... 994  
\tex\_pretolerance:D ..... 520  
\tex\_prevdepth:D ..... 521  
\tex\_prevgraf:D ..... 522  
\tex\_primitive:D .... 779, 886, 3574  
\tex\_protected:D .....  
..... 660, 2108, 2110, 2112,  
2113, 2114, 2115, 2116, 2117, 2118,  
2119, 2127, 2129, 2131, 2133, 29674  
\tex\_protrudechars:D .... 780, 1032  
\tex\_ptexminorversion:D .....  
..... 1239, 26482, 26500  
\tex\_ptexrevision:D 1240, 26483, 26501  
\tex\_ptexversion:D .....  
... 1241, 26478, 26481, 26496, 26499  
\tex\_pxdimen:D ..... 781, 1033  
\tex\_quitvmode:D ..... 798  
\tex\_radical:D ..... 523  
\tex\_raise:D ..... 524, 23697  
\tex\_randomseed:D .... 782, 1034, 26518  
\tex\_read:D ..... 525, 10502  
\tex\_readline:D ..... 661, 10508  
\tex\_relax:D .....  
310, 324, 526, 637, 2077, 6384, 11353  
\tex\_relpriority:D ..... 527  
\tex\_resettimer:D ..... 783, 887  
\tex\_right:D ..... 528, 1410  
\tex\_rightghost:D ..... 995, 1367  
\tex\_righthyphenmin:D ..... 529  
\tex\_rightmarginkern:D ..... 799  
\tex\_rightsip:D ..... 530  
\tex\_roman numeral:D .....  
..... 324, 335, 335, 360, 531,  
2070, 2082, 2365, 2717, 8463, 13177  
\tex\_rpcode:D ..... 800  
\tex\_savecatcodetable:D ... 996, 1341  
\tex\_savepos:D ..... 784, 1037  
\tex\_savinghyphcodes:D ..... 662  
\tex\_savingvdiscards:D ..... 663  
\tex\_scantexttokens:D ..... 997, 1342  
\tex\_scantokens:D .... 664, 4257, 4313  
\tex\_scriptfont:D ..... 532  
\tex\_scriptscriptfont:D ..... 533  
\tex\_scriptscriptstyle:D ..... 534  
\tex\_scriptspace:D ..... 535  
\tex\_scriptstyle:D ..... 536  
\tex\_scrollmode:D ..... 537  
\tex\_setbox:D .. 538, 23637, 23640,  
23645, 23648, 23663, 23672, 23681,  
23717, 23720, 23769, 23775, 23783,  
23789, 23797, 23804, 23819, 23826,  
23867, 23873, 23881, 23887, 23895,  
23901, 23909, 23916, 23932, 23939,  
23951, 23956, 24315, 24318, 25044  
\tex\_setfontid:D ..... 998  
\tex\_setlanguage:D ..... 539  
\tex\_setrandomseed:D 786, 1040, 26531  
\tex\_sfcode:D ..... 540, 8432, 8434  
\tex\_shapemode:D ..... 999  
\tex\_shellescape:D ... 787, 888, 26548  
\tex\_shipout:D ..... 541, 1273, 1297  
\tex\_show:D ..... 542  
\tex\_showbox:D ..... 543, 23755  
\tex\_showboxbreadth:D .... 544, 23751  
\tex\_showboxdepth:D .... 545, 23752  
\tex\_showgroups:D ..... 665  
\tex\_showifs:D ..... 666  
\tex\_showlists:D ..... 546  
\tex\_showmode:D ..... 1242  
\tex\_showthe:D ..... 547  
\tex\_showtokens:D .....  
..... 405, 667, 1403, 5045, 9729  
\tex\_sjis:D ..... 1243  
\tex\_skewchar:D ..... 548  
\tex\_skip:D ..... 549, 20043,  
20072, 20091, 20149, 20165, 20171  
\tex\_skipdef:D ..... 550  
\tex\_space:D ..... 290  
\tex\_spacefactor:D ..... 551  
\tex\_spaceskip:D ..... 552  
\tex\_span:D ..... 553  
\tex\_special:D ..... 554, 28020  
\tex\_splitbotmark:D ..... 555  
\tex\_splitbotmarks:D ..... 668  
\tex\_splitdiscards:D ..... 669

\tex_splitfirstmark:D .....	556	\tex_tpack:D .....	1010
\tex_splitfirstmarks:D .....	670	\tex_tracingassigns:D .....	673
\tex_splitmaxdepth:D .....	557	\tex_tracingcommands:D .....	572
\tex_splittopskip:D .....	558	\tex_tracingfonts:D .....	573
\tex_strcmp:D .....	785, 5171, 13551	..... 788, 1041, 1302, 1304, 1308	
\tex_string:D .....	559, 1276, 1280, 2075	\tex_tracinggroups:D .....	674
\tex_suppressfontnotfounderror:D .....	820, 1350	\tex_tracingifs:D .....	675
\tex_suppressifcsnameerror:D .....	1000, 1343	\tex_tracinglostchars:D .....	573
\tex_suppresslongerror:D .....	1001, 1345	\tex_tracingmacros:D .....	574
\tex_suppressmathparerror:D .....	1002, 1346	\tex_tracingnesting:D .....	575
\tex_suppressoutererror:D .....	1003, 1348	..... 676, 4242, 11120, 26575	
\tex_suppressprimitiveerror:D .....	1005	\tex_tracingonline:D .....	575, 23753
\tex_syncTeX:D .....	801	\tex_tracingoutput:D .....	576
\tex_tabskip:D .....	560	\tex_tracingpages:D .....	577
\tex_tagcode:D .....	802	\tex_tracingparagraphs:D .....	578
\tex_tate:D .....	1244	\tex_tracingrestores:D .....	579
\tex_tbaselineshift:D .....	1245	\tex_tracingscantokens:D .....	677
\tex_textdir:D .....	1006, 1368	\tex_tracingstats:D .....	580
\tex_textdirection:D .....	1007	\tex_uccode:D .....	581, 8426, 8428
\tex_textfont:D .....	561	\tex_Uchar:D .....	1043, 1349, 25545
\tex_textstyle:D .....	562	\tex_Ucharcat:D .....	1044, 8523, 25560
\tex_TeXXeTstate:D .....	671	\tex_uchyph:D .....	582
\tex_tfont:D .....	1246	\tex_ucs:D .....	1257
\tex_the:D .....	258, 310, 352, 563, 672, 677, 678, 2354, 2830, 3117, 3206, 3210, 3573, 3665, 3684, 3699, 3704, 6601, 6603, 7688, 8346, 8416, 8422, 8428, 8434, 11020, 11056, 11634, 11635, 11636, 11690, 11692, 11839, 11841, 11964, 14176, 14666, 19584, 19616, 19664, 19665, 19696, 19697, 19703, 19704, 20164, 20274, 20333, 20352, 20358, 20361, 20365, 23738, 25555, 26365, 26518, 28706	\tex_Udelcode:D .....	1045
\tex_thickmuskip:D .....	564	\tex_Udelcodenum:D .....	1046
\tex_thinmuskip:D .....	565	\tex_Udelimiter:D .....	1047
\tex_time:D .....	566, 7696, 7698	\tex_Udelimiterover:D .....	1048
\tex_toks:D .....	567, 3675, 3704, 19557, 19584, 19616, 19653, 19664, 19665, 19696, 19697, 19703, 19704, 19708, 19718, 19728, 19988, 20006, 20164, 20333, 20336, 20343, 20351, 20352, 20357, 20358, 20361, 20365, 26365, 26376, 26377, 26378	\tex_Udelimiterunder:D .....	1049
\tex_toksapp:D .....	1008	\tex_Uhextensible:D .....	1050
\tex_toksdef:D .....	568, 19836	\tex_Umathaccent:D .....	1051
\tex_tokspre:D .....	1009	\tex_Umathaxis:D .....	1052
\tex_tolerance:D .....	569	\tex_Umathbinbinspacing:D .....	1053
\tex_topmark:D .....	570	\tex_Umathbinclosespacing:D .....	1054
\tex_topmarks:D .....	672	\tex_Umathbininnerspacing:D .....	1055
\tex_topskip:D .....	571	\tex_Umathbinopenspacing:D .....	1056
		\tex_Umathbinopspace:D .....	1057
		\tex_Umathbinordspacing:D .....	1058
		\tex_Umathbinpunctspacing:D .....	1059
		\tex_Umathbinrelspacing:D .....	1060
		\tex_Umathchar:D .....	1061
		\tex_Umathcharclass:D .....	1062
		\tex_Umathchardef:D .....	1063
		\tex_Umathcharfam:D .....	1064
		\tex_Umathcharnum:D .....	1065
		\tex_Umathcharnumdef:D .....	1066
		\tex_Umathcharslot:D .....	1067
		\tex_Umathclosebinspacing:D .....	1068
		\tex_Umathcloseclosespacing:D .....	1070
		\tex_Umathcloseinnerspacing:D .....	1072
		\tex_Umathcloseopenspacing:D .....	1073
		\tex_Umathcloseopspace:D .....	1074
		\tex_Umathcloseordspacing:D .....	1075
		\tex_Umathclosepunctspacing:D .....	1077
		\tex_Umathcloserelspacing:D .....	1078

\tex_Umathcode:D .....	1079	\tex_Umathoverdelimitervgap:D .....	1141
\tex_Umathcodenum:D .....	1080	\tex_Umathpunctbinspacing:D .....	1142
\tex_Umathconnectoroverlapmin:D .....	1082	\tex_Umathpunctclosespacing:D .....	1144
\tex_Umathfractiondelsize:D .....	1083	\tex_Umathpunctinnerspacing:D .....	1146
\tex_Umathfractiondenomdown:D .....	1085	\tex_Umathpunctopenspacing:D .....	1147
\tex_Umathfractiondenomvgap:D .....	1087	\tex_Umathpuncttopspacing:D .....	1148
\tex_Umathfractionnumup:D .....	1088	\tex_Umathpunctordspacing:D .....	1149
\tex_Umathfractionnumvgap:D .....	1089	\tex_Umathpunctpunctspacing:D .....	1151
\tex_Umathfractionrule:D .....	1090	\tex_Umathpunctrelspacing:D .....	1152
\tex_Umathinnerbinspacing:D .....	1091	\tex_Umathquad:D .....	1153
\tex_Umathinnerclosespacing:D .....	1093	\tex_Umathradicaldegreeafter:D .....	1155
\tex_Umathinnerinnerspacing:D .....	1095	\tex_Umathradicaldegreebefore:D .....	1157
\tex_Umathinneropenspacing:D .....	1096	\tex_Umathradicaldegreeraise:D .....	1159
\tex_Umathinneropspacing:D .....	1097	\tex_Umathradicalkern:D .....	1160
\tex_Umathinnerordspacing:D .....	1098	\tex_Umathradicalrule:D .....	1161
\tex_Umathinnerpunctspacing:D .....	1100	\tex_Umathradicalvgap:D .....	1162
\tex_Umathinnerrelspacing:D .....	1101	\tex_Umathrelbinspacing:D .....	1163
\tex_Umathlimitabovebgap:D .....	1102	\tex_Umathrelclosespacing:D .....	1164
\tex_Umathlimitabovekern:D .....	1103	\tex_Umathrelinnerspacing:D .....	1165
\tex_Umathlimitabovevgap:D .....	1104	\tex_Umathrelopspacing:D .....	1166
\tex_Umathlimitbelowbgap:D .....	1105	\tex_Umathrellopspacing:D .....	1167
\tex_Umathlimitbelowkern:D .....	1106	\tex_Umathrelordspacing:D .....	1168
\tex_Umathlimitbelowvgap:D .....	1107	\tex_Umathrelpunctspacing:D .....	1169
\tex_Umathnolimitsfactor:D .....	1108	\tex_Umathrelrelspacing:D .....	1170
\tex_Umathnolimitsupfactor:D .....	1109	\tex_Umathskewedfractionhgap:D .....	1172
\tex_Umathopbinspacing:D .....	1110	\tex_Umathskewedfractionvgap:D .....	1174
\tex_Umathopclosespacing:D .....	1111	\tex_Umathspaceafterscript:D .....	1175
\tex_Umathopenbinspacing:D .....	1112	\tex_Umathstackdenomdown:D .....	1176
\tex_Umathopenclosespacing:D .....	1113	\tex_Umathstacknumup:D .....	1177
\tex_Umathopeninnerspacing:D .....	1114	\tex_Umathstackvgap:D .....	1178
\tex_Umathopenopenspacing:D .....	1115	\tex_Umathsubshiftdown:D .....	1179
\tex_Umathopenopspacing:D .....	1116	\tex_Umathsubshiftdrop:D .....	1180
\tex_Umathopenordspacing:D .....	1117	\tex_Umathsubsupshiftdown:D .....	1181
\tex_Umathopenpunctspacing:D .....	1118	\tex_Umathsubsupvgap:D .....	1182
\tex_Umathopenrelspacing:D .....	1119	\tex_Umathsubtopmax:D .....	1183
\tex_Umathoperatorsize:D .....	1120	\tex_Umathsupbottommin:D .....	1184
\tex_Umathopinnerspacing:D .....	1121	\tex_Umathsupshiftdrop:D .....	1185
\tex_Umathopopenspacing:D .....	1122	\tex_Umathsupshiftup:D .....	1186
\tex_Umathopopspacing:D .....	1123	\tex_Umathsupsubbottommax:D .....	1187
\tex_Umathopordspacing:D .....	1124	\tex_Umathunderbarkern:D .....	1188
\tex_Umathoppunctspacing:D .....	1125	\tex_Umathunderbarrule:D .....	1189
\tex_Umathoprelspacing:D .....	1126	\tex_Umathunderbarvgap:D .....	1190
\tex_Umathordbinspacing:D .....	1127	\tex_Umathunderdelimiterbgap:D .....	1192
\tex_Umathordclosespacing:D .....	1128	\tex_Umathunderdelimitervgap:D .....	1194
\tex_Umathordinnerspacing:D .....	1129	\tex_undefined:D .....	285, 514, 814, 1302, 1374,
\tex_Umathordopenspacing:D .....	1130		1375, 1380, 1381, 1382, 2910, 2918,
\tex_Umathordopspace:D .....	1131		7312, 12346, 12360, 12393, 12414,
\tex_Umathordordspacing:D .....	1132		19496, 19927, 19937, 20015, 20115
\tex_Umathordpunctspacing:D .....	1133	\tex_underline:D .....	583, 1271
\tex_Umathordrelspacing:D .....	1134	\tex_unexpanded:D .....	678, 1396, 1426, 2069, 3489
\tex_Umathoverbarkern:D .....	1135	\tex_unhbox:D .....	584, 23847
\tex_Umathoverbarrule:D .....	1136	\tex_unhcopy:D .....	585, 23846
\tex_Umathoverbarvgap:D .....	1137		
\tex_Umathoverdelimiterbgap:D .....	1139		

```

\tex_uniformdeviate:D ..... 789,
  825, 826, 1035, 1042, 7749, 19021,
  19022, 19203, 19206, 26344, 26375
\tex_unkern:D ..... 586
\tex_unless:D ..... 679, 2054
\tex_Unosubscript:D ..... 1195
\tex_Unosuperscript:D ..... 1196
\tex_unpenalty:D ..... 587
\tex_unskip:D ..... 588
\tex_unvbox:D ..... 589, 23946
\tex_unvcopy:D ..... 590, 23945
\tex_Uoverdelimiter:D ..... 1197
\tex_uppercase:D ..... 591, 29775
\tex_uptexrevision:D ... 1258, 26505
\tex_uptexversion:D ... 1259, 26504
\tex_Uradical:D ..... 1198
\tex_Uroot:D ..... 1199
\tex_Uskewed:D ..... 1200
\tex_Uskewedwithdelims:D ..... 1201
\tex_Ustack:D ..... 1202
\tex_Ustartdisplaymath:D ..... 1203
\tex_Ustartmath:D ..... 1204
\tex_Ustopdisplaymath:D ..... 1205
\tex_Ustopmath:D ..... 1206
\tex_Usubscript:D ..... 1207
\tex_Usuperscript:D ..... 1208
\tex_Uunderdelimiter:D ..... 1209
\tex_Uvextensible:D ..... 1210
\tex_vadjust:D ..... 592
\tex_valign:D ..... 593
\tex_vbadness:D ..... 594
\tex_vbox:D ..... 595, 23851,
  23856, 23861, 23867, 23873, 23895,
  23901, 23909, 23916, 23932, 23939
\tex_vceter:D ..... 596, 1395
\tex_vfil:D ..... 597
\tex_vfill:D ..... 598
\tex_vfilneg:D ..... 599
\tex_vfuzz:D ..... 600
\tex_voffset:D ..... 601, 1402
\tex_vpack:D ..... 1011
\tex_vrule:D ..... 602, 25110, 25165
\tex_vsized:D ..... 603
\tex_vskip:D ..... 604, 11852
\tex_vsplit:D ..... 605, 23951, 23957
\tex_vss:D ..... 606
\tex_vtop:D .. 607, 23853, 23881, 23887
\tex_wd:D ..... 608, 23657
\tex_widowpenalties:D ..... 680
\tex_widowpenalty:D ..... 609
\tex_write:D ..... 610, 2832, 2834, 10657, 10660, 10677
\tex_xdef:D ..... 611, 1424, 2121, 2125, 2129, 2133
\tex_XeTeXcharclass:D ..... 821
\tex_XeTeXcharglyph:D ..... 822
\tex_XeTeXcountfeatures:D ..... 823
\tex_XeTeXcountglyphs:D ..... 824
\tex_XeTeXcountselectors:D .... 825
\tex_XeTeXcountvariations:D ... 826
\tex_XeTeXdashbreakstate:D .... 828
\tex_XeTeXdefaultencoding:D ... 827
\tex_XeTeXfeaturecode:D ..... 829
\tex_XeTeXfeaturename:D ..... 830
\tex_XeTeXfindfeaturebyname:D ... 832
\tex_XeTeXfindselectorbyname:D .. 834
\tex_XeTeXfindvariationbyname:D .. 836
\tex_XeTeXfirstfontchar:D ..... 837
\tex_XeTeXfonttype:D ..... 838
\tex_XeTeXgenerateactualtext:D .. 840
\tex_XeTeXglyph:D ..... 841
\tex_XeTeXglyphbounds:D ..... 842
\tex_XeTeXglyphindex:D ..... 843
\tex_XeTeXglyphname:D ..... 844
\tex_XeTeXinputencoding:D ..... 845
\tex_XeTeXinputnormalization:D .. 847
\tex_XeTeXinterchartokenstate:D .. 849
\tex_XeTeXinterchartoks:D ..... 850
\tex_XeTeXisdefaultselector:D .. 852
\tex_XeTeXisexclusivefeature:D .. 854
\tex_XeTeXlastfontchar:D ..... 855
\tex_XeTeXlinebreaklocale:D ... 857
\tex_XeTeXlinebreakpenalty:D ... 858
\tex_XeTeXlinebreakskip:D ..... 856
\tex_XeTeXOTcountfeatures:D ... 859
\tex_XeTeXOTcountlanguages:D ... 860
\tex_XeTeXOTcountsheets:D ..... 861
\tex_XeTeXOTfeaturetag:D ..... 862
\tex_XeTeXOTlanguagetag:D ..... 863
\tex_XeTeXOTscripttag:D ..... 864
\tex_XeTeXpdffile:D 865, 28992, 29034
\tex_XeTeXpdfpagecount:D ..... 866
\tex_XeTeXpicfile:D ..... 867, 28985
\tex_XeTeXrevision:D ..... 868, 26513
\tex_XeTeXselectorname:D ..... 869
\tex_XeTeXtracingfonts:D ..... 870
\tex_XeTeXupwardsmode:D ..... 871
\tex_XeTeXuseglyphmetrics:D ... 872
\tex_XeTeXvariation:D ..... 873
\tex_XeTeXvariationdefault:D ... 874
\tex_XeTeXvariationmax:D ..... 875
\tex_XeTeXvariationmin:D ..... 876
\tex_XeTeXvariationname:D ..... 877
\tex_XeTeXversion:D ..... 878, 6522, 7247, 7730, 26512
\tex_xkanjiskip:D ..... 1247
\tex_xleaders:D ..... 612
\tex_xspaceskip:D ..... 613

```

```

\tex_xspcode:D ..... 1248
\tex_ybaselineshift:D ..... 1249
\tex_year:D ..... 614, 7701
\tex_yoko:D ..... 1250
\textradir ..... 1006, 1825
\textdirection ..... 1007
\textfont ..... 561
\textstyle ..... 562
\texttt ..... 15636
\TeXeTstate ..... 671, 1485
\tfond ..... 1246, 2028
\TH ..... 27691
\th ..... 27691
\the ..... 67, 218, 219, 220,
         221, 222, 223, 224, 225, 226, 227, 563
\thickmuskip ..... 564
\thimskip ..... 565
thousand commands:
  \c_one_thousand ..... 7257
  \c_ten_thousand ..... 7257
\time ..... 566
\tiny ..... 25103
tl commands:
  \c_empty_tl ..... .
    .. 49, 458, 484, 4058, 4076, 4078,
      4128, 4408, 6970, 6976, 7753, 7764,
      12068, 25573, 27808, 27847, 27866
  \l_my_tl ..... 208, 214
  \c_novalue_tl ..... 39, 49, 4129, 4493
  \c_space_tl .. 49, 4138, 4719, 5658,
    8199, 8208, 9438, 11206, 18978,
    24889, 24933, 25000, 25584, 25654,
    26209, 26211, 26922, 26982, 28038,
    28614, 28619, 28622, 28902, 28905,
    28907, 28908, 28909, 28910, 29470
  \tl_analysis_map_inline:Nn .....
    .. 207, 20190, 21758
  \tl_analysis_map_inline:nn .....
    .. 207, 20190, 22355
  \tl_analysis_show:N .....
    .. 207, 20217, 20314, 20316
  \tl_analysis_show:n .....
    .. 207, 20217, 20318, 20320
  \tl_build_begin:N ..... 258, 259,
    259, 910, 1067, 20922, 21421,
    21784, 21877, 22680, 27758, 27775
  \tl_build_clear:N ..... 258, 27775
  \tl_build_end:N ..... 258,
    259, 910, 1067, 1068, 20952, 20960,
    21431, 21839, 21896, 22714, 27854
  \tl_build_gbegin:N .....
    .. 258, 259, 259, 259, 27758, 27776
  \tl_build_gclear:N ..... 258, 27775
  \tl_build_gend:N ..... 259, 27854
  \tl_build_get:N ..... 259
  \tl_build_get>NN ..... 259, 27840
  \tl_build_gput_left:Nn ... 259, 27821
  \tl_build_gput_right:Nn .. 259, 27777
  \tl_build_put_left:Nn ... 259, 27821
  \tl_build_put_right:Nn . 259, 939,
    1069, 20929, 20947, 20955, 20959,
    21009, 21012, 21045, 21059, 21063,
    21188, 21202, 21243, 21268, 21277,
    21287, 21319, 21332, 21336, 21418,
    21424, 21430, 21434, 21477, 21744,
    21760, 21778, 21847, 21892, 21905,
    22699, 22738, 22770, 22832, 22835,
    22850, 22894, 22910, 22946, 27777
  \tl_case:Nn ..... 40, 4509
  \tl_case:nn ..... 410
  \tl_case:nn(TF) ..... 452
  \tl_case:Nnn ..... 29768, 29770
  \tl_case:NnTF ..... .
    .. 40, 4509, 4514, 4519, 29769, 29771
  \l_tl_case_change_accents_tl ...
    .. 256, 27097, 27697
  \l_tl_case_change_exclude_tl ...
    .. 255, 256, 27119, 27744
  \l_tl_case_change_math_tl ...
    .. 255, 26932, 27739
  \tl_clear:N 35, 36, 4075, 4082, 4235,
    7805, 7806, 10814, 10815, 10818,
    10827, 10937, 10940, 11000, 12197,
    12688, 12760, 20607, 22682, 26122,
    26556, 27857, 27959, 28646, 28654,
    28660, 28845, 28851, 28984, 28990
  \tl_clear_new:N .. 36, 4081, 7809, 7810
  \tl_concat:NNN ..... 36, 4108, 5083
  \tl_const:Nn ..... 35,
    1031, 4061, 4128, 4136, 4138, 4227,
    5839, 7803, 8573, 8604, 8626, 9043,
    9081, 9357, 9358, 9402, 9407, 9409,
    9411, 9413, 9415, 9420, 9421, 9428,
    10711, 10717, 11098, 12132, 12133,
    12134, 12135, 12136, 12137, 12138,
    13200, 13201, 13202, 13203, 13204,
    13212, 13294, 15385, 16680, 17125,
    17126, 17127, 17128, 17129, 17130,
    17131, 17132, 17133, 20304, 20377,
    22979, 25576, 25596, 25608, 25637,
    25667, 25668, 25669, 26340, 26552,
    27420, 27421, 27422, 27443, 27444,
    27448, 27449, 27450, 27451, 27452,
    27461, 27475, 27512, 27528, 27651,
    27674, 27676, 27694, 27695, 27903
  \tl_count:N ..... 24, 40, 43, 43, 4602
  \tl_count:n ... 24, 40, 43, 43, 343,
    418, 444, 641, 2554, 2558, 2959,

```

3009, 4602, 4910, 4925, 4937, 21676  
\tl\_count\_tokens:n ..... 254, 26692  
\tl\_gclear:N .. 35, 841, 4075, 4084,  
7807, 7808, 19578, 27862, 29508, 29544  
\tl\_gclear\_new:N .. 36, 4081, 7811, 7812  
\tl\_gconcat:NNN ..... 36, 4108, 5084  
\tl\_gput\_left:Nn ..... 36, 4163  
\tl\_gput\_right:Nn ..... 36, 2403, 2404, 2430, 2435,  
2451, 2456, 4195, 5821, 5963, 19290  
\tl\_gremove\_all:Nn ..... 37, 4390  
\tl\_gremove\_once:Nn ..... 37, 4384  
\tl\_greplace\_all:Nnn .. 37, 4315, 4393  
\tl\_greplace\_once:Nnn .. 37, 4315, 4387  
\tl\_greverse:N ..... 43, 4749  
.tl\_gset:N ..... 173, 12556  
\tl\_gset:Nn ..... 36, 69,  
383, 1070, 4123, 4139, 4231, 4318,  
4322, 4640, 4752, 5114, 5118, 5878,  
5883, 5895, 5932, 5949, 5989, 6015,  
6093, 6125, 6162, 6168, 7824, 7851,  
7870, 7913, 7949, 7998, 8037, 9078,  
9146, 9175, 9211, 9219, 9242, 15383,  
19574, 20082, 20612, 21739, 26307,  
26317, 26327, 26738, 27861, 29467  
\tl\_gset\_eq>NN .. 36, 4078, 4087,  
5080, 5862, 5863, 5864, 5865, 7401,  
7817, 7818, 7819, 7820, 9066, 9067,  
9068, 9069, 15390, 19568, 22974, 26717  
\tl\_gset\_from\_file:Nnn ..... 26708  
\tl\_gset\_from\_file\_x:Nnn ..... 26708  
\tl\_gset\_rescan:Nnn ..... 38, 4228  
.tl\_gset\_x:N ..... 173, 12556  
\tl\_gsort:Nn ..... 44, 4681, 19566  
\tl\_gtrim\_spaces:N ..... 44, 4631  
\tl\_head:N ..... 45, 4755  
\tl\_head:n .. 45, 45, 361, 397, 398,  
402, 3555, 4755, 4934, 25593, 25601  
\tl\_head:w ..... 45, 398, 399,  
4755, 4795, 4812, 4834, 25618, 25649  
\tl\_if\_blank:nTF ..... 38, 45, 45,  
45, 4396, 4782, 4924, 5063, 5090,  
8207, 8286, 9544, 11033, 12195,  
12684, 12699, 12853, 19740, 22382,  
25612, 25641, 27308, 27328, 27378  
\tl\_if\_blank\_p:n ..... 38, 4396  
\tl\_if\_empty:N 5167, 5169, 8052, 8054  
\tl\_if\_empty:NTF .....  
... 39, 2248, 4406, 10831, 10921,  
10956, 12180, 12205, 12704, 12833,  
22737, 28649, 28690, 28698, 28872,  
28876, 28903, 28918, 29005, 29470  
\tl\_if\_empty:nTF ..... 39, 387, 388,  
389, 484, 493, 2363, 2594, 2685,  
3534, 3612, 4329, 4416, 4427, 4474,  
4873, 4894, 5125, 5737, 5744, 5761,  
5898, 7461, 7758, 7777, 7785, 7788,  
8065, 8098, 8819, 9112, 9445, 9722,  
9814, 9829, 9949, 9953, 9994, 10032,  
10217, 10218, 10227, 10234, 10240,  
10247, 10316, 10825, 11263, 11266,  
11346, 12246, 12342, 12935, 13922,  
14771, 18962, 19037, 20308, 20309,  
23554, 25635, 26659, 27936, 29564  
\tl\_if\_empty\_p:N 39, 4406, 28686, 28915  
\tl\_if\_empty\_p:n ..... 39, 4416, 4427  
\tl\_if\_eq:NN ..... 410  
\tl\_if\_eq:nn(TF) ..... 110, 110  
\tl\_if\_eq:NNTF .....  
... 39, 40, 63, 431, 4440, 4533,  
6001, 9262, 9795, 9849, 25176, 25179  
\tl\_if\_eq:nnTF .....  
... 39, 72, 72, 431, 4450, 8086  
\tl\_if\_eq\_p:NN ..... 39, 4440  
\tl\_if\_exist:N ..... 5163, 5165  
\tl\_if\_exist:NTF ..... 36, 4082,  
4084, 4126, 4595, 20219, 26749, 26757  
\tl\_if\_exist\_p:N ..... 36, 4126  
\tl\_if\_head\_eq\_catcode:nN ..... 399  
\tl\_if\_head\_eq\_catcode:nTF .....  
... 46, 4788, 26850, 26850  
\tl\_if\_head\_eq\_catcode\_p:nN 46, 4788  
\tl\_if\_head\_eq\_charcode:nN ..... 398  
\tl\_if\_head\_eq\_charcode:nNTF 46, 4788  
\tl\_if\_head\_eq\_charcode\_p:nN 46, 4788  
\tl\_if\_head\_eq\_meaning:nN ..... 399  
\tl\_if\_head\_eq\_meaning:nNTF 46, 4788  
\tl\_if\_head\_eq\_meaning\_p:nN .....  
... 46, 4788, 21675  
\tl\_if\_head\_is\_group:nTF .....  
... 46, 3531, 3609, 4695,  
4815, 4850, 4876, 7783, 26877, 26959  
\tl\_if\_head\_is\_group\_p:n .. 46, 4876  
\tl\_if\_head\_is\_N\_type:n ..... 399  
\tl\_if\_head\_is\_N\_type:nTF .....  
... 46, 3528, 3597, 3603, 3648,  
3663, 3708, 4692, 4792, 4809, 4826,  
4859, 4995, 26656, 26874, 26956,  
27210, 27244, 27337, 27387, 27712  
\tl\_if\_head\_is\_N\_type\_p:n .. 46, 4859  
\tl\_if\_head\_is\_space:nTF .....  
... 46, 4887, 4977, 4986, 5652, 28033  
\tl\_if\_head\_is\_space\_p:n .. 46, 4887  
\tl\_if\_in:Nn ..... 493  
\tl\_if\_in:nn ..... 389  
\tl\_if\_in:NnTF ..... 39,  
4351, 4465, 4465, 4466, 5815, 11093

```

\tl_if_in:nTF ..... 39, 388, 410, 4274, 4335, 4337,
4465, 4466, 4467, 4470, 5218, 5226,
9708, 9710, 20494, 24980, 26566, 27934
\tl_if_novalue:nTF ..... 39, 4480
\tl_if_novalue_p:n ..... 39, 4480
\tl_if_single:n ..... 389
\tl_if_single:NTF 40, 4494, 4495, 4496
\tl_if_single:nTF ..... 40, 473, 4495, 4496, 4497, 4498
\tl_if_single_p:N ..... 40, 4494
\tl_if_single_p:n .... 40, 4494, 4498
\tl_if_single_token:nTF .. 254, 26654
\tl_if_single_token_p:n .. 254, 26654
\tl_item:Nn ..... 47, 4899
\tl_item:nn ..... 47, 402, 4899, 4925
\tl_log:N ..... 49, 5027
\tl_log:n ..... 49, 347, 347, 711, 3109, 3125,
5029, 5051, 7323, 7435, 15415, 26206
\tl_lower_case:n .. 60, 120, 254, 26851
\tl_lower_case:nn ..... 254, 26851
\tl_map_break: ..... 41, 219, 865, 4546, 4552,
4564, 4574, 4581, 4586, 20213, 20214
\tl_map_break:n ..... 41, 42, 4586, 11146, 19573
\tl_map_function:NN ..... 40, 40,
40, 250, 251, 4542, 4610, 11087, 21746
\tl_map_function:nN ..... 40, 40,
41, 3003, 4048, 4542, 4605, 5901, 21029
\tl_map_inline:Nn ..... 40, 41, 4556, 11145, 19573
\tl_map_inline:nn ..... 41, 41, 65, 4556, 7732, 8968, 8970,
8972, 8982, 10714, 14824, 17796, 19524
\tl_map_variable:NNn ..... 41, 4570
\tl_map_variable:nNn .. 41, 391, 4570
\tl_mixed_case:n ..... 60, 254, 256, 259, 26851
\tl_mixed_case:nn ..... 254, 26851
\l_tl_mixed_case_ignore_tl ..... 256, 27171, 27749
\l_tl_mixed_change_ignore_tl .. 256
\tl_new:N ..... 35, 36,
122, 377, 4055, 4082, 4084, 4463,
4464, 5053, 5054, 5055, 5056, 5811,
5836, 5837, 7754, 7800, 7801, 8499,
8867, 9042, 9354, 9740, 9741, 10166,
10381, 10388, 10581, 10688, 10690,
10703, 10705, 10706, 10708, 11012,
11983, 11984, 11985, 12140, 12142,
12143, 12146, 12147, 12148, 12152,
12153, 19286, 19452, 19890, 20368,
20369, 20375, 20376, 20386, 22336,
22604, 22606, 24334, 24359, 24360,
25098, 25341, 26551, 27697, 27739,
27744, 27749, 27947, 28642, 29474
\tl_put_left:Nn ..... 36, 4163
\tl_put_right:Nn ..... 36, 1068, 2241, 4195,
5961, 8543, 8545, 8548, 8550, 8551,
8553, 8555, 8557, 8558, 8560, 8562,
8564, 8566, 10962, 10965, 10970,
12702, 20614, 22773, 27986, 27991
\tl_rand_item:N ..... 47, 4922
\tl_rand_item:n ..... 47, 4922
\tl_range:Nnn ..... 48, 4929
\tl_range:nnn ..... 48, 59, 258, 4929
\tl_range_braced:Nnn .... 258, 27872
\tl_range_braced:nnn .. 48, 258, 27872
\tl_range_unbraced:Nnn ... 258, 27872
\tl_range_unbraced:nnn 48, 258, 27872
\tl_remove_all:Nn .. 37, 37, 4390, 11092
\tl_remove_once:Nn ..... 37, 4384
\tl_replace_all:Nnn ..... 37, 429, 490, 4315, 4391, 5910
\tl_replace_once:Nnn ..... 37, 4315, 4385, 8581
\tl_rescan:nn ..... 38, 38, 380, 4228
\tl_reverse:N ..... 43, 43, 4749
\tl_reverse:n ..... 43, 43, 43, 1043, 4729, 4750, 4752
\tl_reverse_items:n .. 43, 43, 43, 4615
\tl_reverse_tokens:n .... 254, 26668
.tl_set:N ..... 173, 12556
\tl_set:Nn ..... 36, 37, 38, 69, 173, 259, 259, 366,
378, 383, 542, 1070, 2246, 4115,
4139, 4229, 4316, 4320, 4453, 4454,
4580, 4638, 4750, 5040, 5112, 5116,
5868, 5873, 5893, 5900, 5904, 5915,
5930, 5941, 5987, 5997, 6006, 6013,
6044, 6047, 6064, 6072, 6081, 6091,
6100, 6106, 6123, 6137, 6159, 6165,
6260, 6867, 7822, 7849, 7868, 7902,
7908, 7911, 7917, 7924, 7947, 7996,
8035, 8175, 8541, 8546, 8901, 9075,
9140, 9156, 9157, 9165, 9166, 9168,
9174, 9177, 9200, 9201, 9210, 9218,
9222, 9240, 9245, 9295, 9480, 9724,
9752, 9835, 10164, 10448, 10619,
10695, 10770, 10773, 10774, 10779,
10790, 10795, 10816, 10953, 10973,
10992, 10994, 11074, 11077, 11078,
11134, 11676, 11992, 12011, 12018,
12035, 12042, 12053, 12119, 12159,
12161, 12189, 12203, 12209, 12212,
```

12221, 12222, 12225, 12324, 12582,  
 12584, 12595, 12596, 12620, 12621,  
 12661, 12682, 12694, 12700, 12762,  
 12791, 15381, 15710, 20492, 20505,  
 20599, 20963, 21370, 21375, 21640,  
 21700, 21730, 21842, 21899, 22434,  
 22443, 22527, 22559, 22872, 23151,  
 23220, 23250, 23274, 24601, 24981,  
 24982, 25100, 25103, 25342, 26112,  
 26305, 26315, 26325, 26591, 26727,  
 26729, 27483, 27502, 27660, 27698,  
 27741, 27746, 27750, 27841, 27856,  
 27958, 28035, 28047, 28098, 28101,  
 28104, 28108, 28111, 28647, 28662  
`\tl_set_eq:N` ..... 36, 470, 4076, 4087, 5079, 5858,  
 5859, 5860, 5861, 7400, 7813, 7814,  
 7815, 7816, 9062, 9063, 9064, 9065,  
 9798, 9806, 12756, 15389, 19566, 22969  
`\tl_set_from_file:Nnn` ..... 26708  
`\tl_set_from_file_x:Nnn` ..... 26708  
`\tl_set_rescan:Nnn` .. 38, 38, 578, 4228  
`.tl_set_x:N` ..... 173, 12556  
`\tl_show:N` 49, 49, 865, 5027, 5688, 20225  
`\tl_show:n` .. 49, 49, 248, 347, 347,  
 405, 405, 711, 1030, 3105, 3122,  
 5027, 5036, 5687, 7322, 7433, 8348,  
 8418, 8424, 8430, 8436, 15413, 26204  
`\tl_show_analysis:N` ..... 20313  
`\tl_show_analysis:n` ..... 20313  
`\tl_sort:Nn` ..... 44, 4681, 19566  
`\tl_sort:nN` .. 44, 846, 848, 4681, 19736  
`\tl_tail:N` ..... 45, 2247, 4755, 21651  
`\tl_tail:n` ..... 45, 4755  
`\tl_to_lowercase:n` ..... 29772  
`\tl_to_str:N` .... 42, 51, 150, 407,  
 569, 942, 4591, 5133, 5210, 5218,  
 10774, 10785, 11079, 11289, 11303  
`\tl_to_str:n` .. 38, 38, 42, 42, 51, 60,  
 61, 133, 133, 150, 170, 177, 213,  
 214, 252, 315, 330, 389, 407, 413,  
 419, 512, 528, 529, 942, 942, 2074,  
 2097, 2189, 2197, 2228, 2233, 2444,  
 2579, 2661, 3765, 3779, 3782, 3789,  
 3793, 3997, 4029, 4248, 4332, 4419,  
 4590, 5037, 5052, 5095, 5134, 5218,  
 5226, 5361, 5383, 5407, 5414, 5468,  
 5475, 5549, 5568, 5579, 5604, 5612,  
 5620, 5626, 5638, 5649, 7142, 7159,  
 7203, 7329, 8708, 8712, 8742, 8743,  
 8776, 8791, 8793, 8795, 8813, 9007,  
 9116, 9127, 9185, 9186, 9224, 9247,  
 9269, 9270, 9584, 9585, 9887, 9888,  
 10696, 10712, 11170, 11244, 11501,  
 11703, 11834, 12079, 12860, 13331,  
 13335, 13352, 13573, 13574, 14178,  
 14179, 14184, 14188, 18719, 18773,  
 18847, 19357, 21029, 22836, 22986,  
 25128, 25211, 26180, 26209, 26211,  
 26215, 26217, 26222, 26224, 26561,  
 26632, 26645, 27058, 27061, 27929,  
 27934, 28496, 28501, 28724, 28738,  
 28746, 28932, 28937, 29678, 29681  
`\tl_to_uppercase:n` ..... 29774  
`\tl_trim_spaces:N` ..... 44, 4631  
`\tl_trim_spaces:n` 44, 4631, 5922, 12860  
`\tl_trim_spaces_apply:nN` .....  
 ... 44, 4631, 7760, 8299, 9098, 12120  
`\tl_trim_spacs:n` ..... 394  
`\tl_upper_case:n` .....  
 ... 60, 120, 254, 259, 26851  
`\tl_upper_case:nn` ..... 254, 26851  
`\tl_use:N` ..... 43, 79, 160,  
 164, 167, 4593, 12392, 26750, 26758  
`\g_tmpa_tl` ..... 50, 5053  
`\l_tmpa_tl` .....  
 ... 4, 37, 50, 1276, 1278, 1295, 5055  
`\g_tmpb_tl` ..... 50, 5053  
`\l_tmpb_tl` .....  
 ... 50, 1277, 1278, 1293, 1295, 5055  
`\tl` internal commands:  
`\c__tl_accents_lt_tl` .....  
 ... 1057, 27303, 27413  
`\__tl_act:NNNnn` ..... 395, 395,  
 396, 1043, 4683, 4734, 26673, 26696  
`\__tl_act_count_group:nn` ..... 26692  
`\__tl_act_count_normal:nN` ..... 26692  
`\__tl_act_count_space:n` ..... 26692  
`\__tl_act_end:w` ..... 4683  
`\__tl_act_end:wn` .. 1043, 4704, 4710  
`\__tl_act_group:nwnNNN` ..... 4683  
`\__tl_act_group_recuse:Nnn` .....  
 ... 26683, 26687  
`\__tl_act_loop:w` ..... 4683  
`\__tl_act_normal:NwnNNN` ..... 4683  
`\__tl_act_output:n` ..... 396, 4683  
`\__tl_act_result:n` .....  
 395, 4688, 4710, 4725, 4726, 4727, 4728  
`\__tl_act_reverse` ..... 396  
`\__tl_act_reverse_output:n` .....  
 ... 4683, 4744, 4746, 4748, 26684  
`\__tl_act_space:wwnNNN` ..... 395, 4683  
`\__tl_analysis:n` .....  
 856, 865, 19913, 20192, 20221, 20229  
`\__tl_analysis_a:n` ..... 19917, 19942  
`\__tl_analysis_a_bgroup:w` .....  
 ... 19973, 19995  
`\__tl_analysis_a_cs:ww` ..... 20052

```

\__tl_analysis_a_egroup:w ..... 19975, 19995
\__tl_analysis_a_group:nw .... 19995
\__tl_analysis_a_group_aux:w .. 19995
\__tl_analysis_a_group_auxii:w 19995
\__tl_analysis_a_group_test:w . 19995
\__tl_analysis_a_loop:w .. 19949,
    19952, 19993, 20035, 20049, 20067
\__tl_analysis_a_safe:N ..... 19974, 20016, 20052
\__tl_analysis_a_space:w 19972, 19978
\__tl_analysis_a_space_test:w ... 858, 19978
\__tl_analysis_a_store: ..... 859, 19989, 20031, 20037
\__tl_analysis_a_type:w 19953, 19954
\__tl_analysis_b:n .... 19918, 20080
\__tl_analysis_b_char:Nww ..... 20107, 20113
\__tl_analysis_b_cs:Nww 20109, 20137
\__tl_analysis_b_cs_test:ww .. 20137
\__tl_analysis_b_loop:w ..... 864, 20080, 20183, 20188
\__tl_analysis_b_normal:wwN .... 20093, 20158
\__tl_analysis_b_normals:ww ... 863, 863, 20090, 20093, 20134, 20144
\__tl_analysis_b_special:w ..... 20096, 20155
\__tl_analysis_b_special_char:wn ... 20155
\__tl_analysis_b_special_space:w ... 20155
\l__tl_analysis_char_token ..... 854, 858,
    859, 19884, 19982, 19987, 20025, 20030
\__tl_analysis_cs_space_count>NN ..... 19897, 20066, 20140
\__tl_analysis_cs_space_count:w .. 19897
\__tl_analysis_cs_space_count_- end:w ..... 19897
\__tl_analysis_disable:n ..... 19922, 19944, 20010, 20063
\__tl_analysis_extract_charcode: ..... 19891, 20005
\__tl_analysis_extract_charcode_- aux:w ..... 19891
\l__tl_analysis_index_int ..... 860, 861,
    19887, 19947, 19950, 19988, 20006,
    20043, 20046, 20072, 20074, 20161
\__tl_analysis_map_inline_aux:Nn ..... 20190
\__tl_analysis_map_inline_- aux:nnm ..... 20190
\l__tl_analysis_nesting_int ... 857, 19888, 19948, 20039, 20048
\l__tl_analysis_normal_int .... 19886, 19946, 19991, 20033,
    20044, 20047, 20064, 20073, 20078
\g__tl_analysis_result_tl ..... 865, 19890, 20082, 20212, 20235
\__tl_analysis_show: ..... 20223, 20231, 20233
\__tl_analysis_show_active:n ... 20248, 20277
\__tl_analysis_show_cs:n 20244, 20277
\c__tl_analysis_show_etc_str ... 867, 20297, 20299, 20304
\__tl_analysis_show_long:nn .. 20277
\__tl_analysis_show_long_- aux:nnmn ..... 20277, 20283
\__tl_analysis_show_loop:wNw . 20233
\__tl_analysis_show_normal:n ... 20251, 20257
\__tl_analysis_show_value:N ... 20262, 20286
\l__tl_analysis_token ..... 854, 854, 858,
    859, 19884, 19894, 19953, 19957,
    19960, 19963, 20011, 20015, 20030
\l__tl_analysis_type_int ..... 858, 860, 861, 19889,
    19956, 19971, 20039, 20041, 20045
\__tl_build_begin>NN .. 27758, 27809
\__tl_build_begin>NNN .. 1067, 27758
\__tl_build_end_loop>NN ..... 27854
\__tl_build_get>NNN ..... 27840, 27856, 27861
\__tl_build_get:w ..... 27840
\__tl_build_get_end:w ..... 27840
\__tl_build_last>NNn ..... 1067, 1068, 27772, 27777, 27844
\__tl_build_put:nn 1068, 27777, 27835
\__tl_build_put:nw ..... 1068, 27777
\__tl_build_put_left>NNn ..... 27821
\__tl_case:NnTF ..... 4512, 4517, 4522, 4527, 4529
\__tl_case:nNTF ..... 4509
\__tl_case:Nw ..... 4509
\__tl_case_end:nw ..... 4509
\__tl_change_case:nnn 26851, 26852,
    26853, 26854, 26855, 26856, 26857
\__tl_change_case_aux:nnn .... 26857
\__tl_change_case_char:nN .... 26857
\__tl_change_case_char_lower:Nnn ..... 26857

```

`\__tl_change_case_char_mixed:Nnn` ..... [26857](#)  
`\__tl_change_case_char_upper:Nnn` ..... [26857](#)  
`\__tl_change_case_char_UTFviii:nn` ..... [26857](#)  
`\__tl_change_case_char_UTFviii:nNN` ..... [26857](#)  
`\__tl_change_case_char_UTFviii:nnN` ..... [27051, 27053, 27055, 27056](#)  
`\__tl_change_case_char_UTFviii:nNNN` ..... [26857](#)  
`\__tl_change_case_char_UTFviii:nNNNN` ..... [26857](#)  
`\__tl_change_case_char_UTFviii:nNNNNN` ..... [27037, 27054](#)  
`\__tl_change_case_cs:N` ..... [26857](#)  
`\__tl_change_case_cs:NN` ..... [26857](#)  
`\__tl_change_case_cs:NNn` ..... [26857](#)  
`\__tl_change_case_cs_accents:NN` ..... [26857](#)  
`\__tl_change_case_cs_expand:NN` ..... [26857](#)  
`\__tl_change_case_cs_expand:Nnw` ..... [26857](#)  
`\__tl_change_case_cs_letterlike:Nn` ..... [26857](#)  
`\__tl_change_case_cs_letterlike:NnN` ..... [26857](#)  
`\__tl_change_case_end:wn` ..... [26857](#)  
`\__tl_change_case_group:nwnn` ..... [26857](#)  
`\__tl_change_case_group_lower:nnnn` ..... [26857](#)  
`\__tl_change_case_group_mixed:nnnn` ..... [26857](#)  
`\__tl_change_case_group_upper:nnnn` ..... [26857](#)  
`\__tl_change_case_if_expandable:NTF` ..... [26857, 27216, 27251, 27343, 27393, 27718](#)  
`\__tl_change_case_loop:wn` ..... [1056](#)  
`\__tl_change_case_loop:wnn` ..... [1048, 1050](#), [26857](#)  
`\__tl_change_case_lower_az:Nnw` ..... [27228](#)  
`\__tl_change_case_lower_lt:nNnw` ..... [27300](#)  
`\__tl_change_case_lower_lt:NNw` ..... [27300](#)  
`\__tl_change_case_lower_lt:Nnw` ..... [27300](#)  
`\__tl_change_case_lower_lt:nnn` ..... [27300](#)  
`\__tl_change_case_lower_lt:Nw` ..... [27300](#)  
`\__tl_change_case_lower_sigma:Nnw` ..... [27198](#)  
`\__tl_change_case_lower_sigma:Nw` ..... [27198](#)  
`\__tl_change_case_lower_sigma:w` ..... [27198](#)  
`\__tl_change_case_lower_tr:Nnw` ..... [27228](#)  
`\__tl_change_case_lower_tr_-auxi:Nw` ..... [27228](#)  
`\__tl_change_case_lower_tr_-auxii:Nw` ..... [27228](#)  
`\__tl_change_case_math:NNNnnn` ..... [1049, 26857](#)  
`\__tl_change_case_math:NwNNnn` ..... [26857](#)  
`\__tl_change_case_math_group:nwNNnn` ..... [26857](#)  
`\__tl_change_case_math_loop:wNNnn` ..... [26857](#)  
`\__tl_change_case_math_space:wNNnn` ..... [26857](#)  
`\__tl_change_case_mixed_nl:NNw` ..... [27700](#)  
`\__tl_change_case_mixed_nl:Nnw` ..... [27700](#)  
`\__tl_change_case_mixed_nl:Nw` ..... [27700](#)  
`\__tl_change_case_mixed_skip:N` ..... [26857](#)  
`\__tl_change_case_mixed_skip>NN` ..... [26857](#)  
`\__tl_change_case_mixed_skip_-tidy:Nwn` ..... [26857](#)  
`\__tl_change_case_mixed_switch:w` ..... [26857](#)  
`\__tl_change_case_N_type:Nnnn` ..... [26857](#)  
`\__tl_change_case_N_type:NNNnnn` ..... [26857](#)  
`\__tl_change_case_N_type:Nwnn` ..... [26857](#)  
`\__tl_change_case_output:nwn` ..... [26857, 27202, 27238, 27246, 27260, 27262, 27272, 27283, 27295, 27322, 27331, 27359, 27381, 27410, 27706, 27732](#)  
`\__tl_change_case_protect:wNN` ..... [26857](#)  
`\__tl_change_case_result:n` ..... [26870, 26883, 26884, 26886](#)  
`\__tl_change_case_setup:NN` ..... [27671, 27678, 27680](#)  
`\__tl_change_case_space:wnn` ..... [26857](#)  
`\__tl_change_case_upper_az:Nnw` ..... [27228](#)  
`\__tl_change_case_upper_de-alt:Nnw` ..... [27407](#)  
`\__tl_change_case_upper_lt:NNw` ..... [27300](#)  
`\__tl_change_case_upper_lt:Nnw` ..... [27300](#)  
`\__tl_change_case_upper_lt:nnw` ..... [27300](#)  
`\__tl_change_case_upper_lt:Nw` ..... [27300](#)  
`\__tl_change_case_upper_sigma:Nnw` ..... [27198](#)  
`\__tl_change_case_upper_sigma:Nw` ..... [27198](#)  
`\__tl_change_case_upper_tr:Nnw` ..... [27228](#)  
`\__tl_count:n` ..... [393, 4602](#)  
`\c__tl_dot_above_tl` ..... [27359, 27413](#)

```

\c__tl_dotless_i_tl ..... 27246, 27260, 27272, 27454
\c__tl_dotted_I_tl .... 27295, 27454
\c__tl_final_sigma_tl ..... 27212, 27224, 27413
\__tl_head_auxi:nw ..... 4755
\__tl_head_auxii:n ..... 4755
\c__tl_I_ogonek_tl .... 27371, 27454
\c__tl_i_ogonek_tl .... 27316, 27454
\__tl_if_blank_p:NNw ..... 4396
\__tl_if_empty_if:n ..... 386, 387, 425, 4398, 4427, 26657, 26661
\__tl_if_head_eq_meaning_- normal:nN ..... 4827, 4831
\__tl_if_head_eq_meaning_- special:nN ..... 4828, 4840
\__tl_if_head_is_N_type:w . 400, 4859
\__tl_if_head_is_space:w .... 4887
\__tl_if_novalue:w ..... 4480
\__tl_if_single:nnw .. 389, 4500, 4508
\__tl_if_single:nTF ..... 4498
\__tl_if_single_p:n ..... 4498
\l__tl_internal_a_t1 ..... 405, 4234, 4235, 4236, 4450, 5040, 5046, 26716, 26717, 26725, 26727, 26729, 26736, 26738, 27483, 27485, 27502, 27507, 27660, 27662
\l__tl_internal_b_t1 ..... 4450
\__tl_item:nn ..... 4899
\__tl_item_aux:nn ..... 4899
\__tl_loop:nn ... 27499, 27508, 27544
\__tl_map_function:Nn .. 391, 4542, 4561
\__tl_map_variable:Nnn ..... 4570
\__tl_range:Nnnn .. 4929, 27874, 27879
\__tl_range:nnNn ..... 4929
\__tl_range:nnnNn ..... 4929
\__tl_range:w ..... 402, 4929
\__tl_range_braced:w .. 402, 1070, 27872
\__tl_range_collect:nn .. 1070, 4929
\__tl_range_collect_braced:w .. 402, 1070, 27872
\__tl_range_collect_group:nN .. 4929
\__tl_range_collect_group:nn .. 4997, 5006
\__tl_range_collect_N:nN .. 4929
\__tl_range_collect_space:nw .. 4929
\__tl_range_collect_unbraced:w .. 27872
\__tl_range_items:nnNn ..... 402
\__tl_range_normalize:nn ..... 4944, 4948, 5008
\__tl_range_skip:w ..... 402, 4929
\__tl_range_skip_spaces:n .. 4929
\__tl_range_unbraced:w ..... 27872
\__tl_replace:NnNNnn ..... 383, 384, 4316, 4318, 4320, 4322, 4327
\__tl_replace_auxi:NnnNNNnn .. 384, 4327
\__tl_replace_auxii:nNNNnn ..... 383, 384, 4327
\__tl_replace_next:w ..... 383, 384, 385, 4320, 4322, 4327
\__tl_replace_wrap:w ..... 383, 384, 385, 4316, 4318, 4327
\__tl_rescan>NNw .. 382, 4228, 4304, 4310
\c__tl_rescan_marker_t1 ..... 4227, 4243, 4260, 4305
\__tl_reverse_group:nn ..... 26668
\__tl_reverse_group_preserve:nn .. 4729
\__tl_reverse_items:nwNwn .. 4615
\__tl_reverse_items:wn ..... 4615
\__tl_reverse_normal:nn .. 4729, 26674
\__tl_reverse_space:n .. 4729, 26676
\__tl_set_rescan:nNN .. 380, 4248, 4269
\__tl_set_rescan>NNnn .. 380, 4228
\__tl_set_rescan>NNw ..... 380
\__tl_set_rescan_multi:nNN .. 380, 381, 382, 4228, 4277, 4298
\__tl_set_rescan_single:nnNN .. 381, 4269
\__tl_set_rescan_single_aux:nnNN .. 4269
\__tl_show:n ..... 5036
\__tl_show:NN ..... 5027
\__tl_show:w ..... 5036
\c__tl_std_sigma_tl .. 27223, 27413
\__tl_tmp:n ..... 27418, 27420, 27421, 27424, 27426, 27427, 27428, 27430, 27432, 27433, 27434, 27436, 27438, 27439, 27440, 27443, 27444
\__tl_tmp:nnnn .. 27510, 27541, 27542
\__tl_tmp:w ..... 388, 394, 4473, 4474, 4480, 4493, 4643, 4680, 27459, 27470, 27473, 27485, 27489, 27490, 27491, 27492, 27507, 27526, 27646, 27649, 27662, 27665, 27666, 27667
\__tl_trim_spaces:nn .. 4632, 4635, 4643
\__tl_trim_spaces_auxi:w .. 394, 4643
\__tl_trim_spaces_auxii:w .. 394, 4643
\__tl_trim_spaces_auxiii:w .. 394, 4643
\__tl_trim_spaces_auxiv:w .. 394, 4643
\c__tl_upper_Eszett_t1 .. 27410, 27413
\tn ..... 20329
token commands:
\c_alignment_token ..... 122, 508, 8606, 8645, 20123
\c_parameter_token ..... 122, 509, 936, 8606, 8649, 8652
\g_peek_token .. 126, 126, 8864, 8875

```

\l\_peek\_token . . . . . 126, 126, 518,  
 1071, 1072, 1073, 8864, 8873, 8890,  
 8929, 8941, 8961, 27912, 27913,  
 27914, 27917, 27974, 27975, 27976  
\c\_space\_token . . . . .  
 . . . . . 31, 46, 49, 122, 260, 260,  
 399, 509, 3632, 4817, 4852, 8606,  
 8669, 8890, 10886, 19957, 19987,  
 20129, 20679, 20714, 27914, 27976  
\token\_get\_arg\_spec:N . . . . . 129, 9005  
\token\_get\_prefix\_spec:N . . . . . 129, 9005  
\token\_get\_replacement\_spec:N . . . . .  
 . . . . . 129, 9005, 12889  
\token\_if\_active:NTF . . . . . 124, 8682  
\token\_if\_active\_p:N . . . . . 124, 8682  
\token\_if\_alignment:NTF 123, 123, 8643  
\token\_if\_alignment\_p:N . . . . . 123, 8643  
\token\_if\_chardef:NTF 125, 8754, 20266  
\token\_if\_chardef\_p:N . . . . . 125, 8754  
\token\_if\_cs:NTF . . . . . 124, 8719, 26989  
\token\_if\_cs\_p:N . . . . .  
 . . . . . 124, 8719, 27258, 27350, 27400, 27725  
\token\_if\_dim\_register:NTF . . . . .  
 . . . . . 125, 8754, 20268  
\token\_if\_dim\_register\_p:N 125, 8754  
\token\_if\_eq\_catcode:NNTF . . . . . 124,  
 126, 127, 127, 127, 260, 3632, 8692  
\token\_if\_eq\_catcode\_p:NN . . . . . 124, 8692  
\token\_if\_eq\_charcode:NNTF . . . . .  
 . . . . . 124, 127, 127, 127, 128, 260,  
 8697, 10130, 10886, 11097, 17388,  
 20714, 20719, 21342, 21542, 21555,  
 21557, 21595, 21716, 22740, 22819  
\token\_if\_eq\_charcode\_p:NN 124, 8697  
\token\_if\_eq\_meaning:NNTF . . . . .  
 . . . . . 124, 128, 128,  
 128, 128, 260, 3635, 3646, 8687,  
 10938, 13692, 14699, 14758, 15658,  
 15660, 15665, 15729, 15915, 17916,  
 21036, 21348, 21381, 21530, 21553,  
 21585, 21711, 21714, 22790, 22817,  
 22858, 22875, 26939, 26967, 26971  
\token\_if\_eq\_meaning\_p:NN . . . . .  
 . . . . . 124, 8687, 27151  
\token\_if\_expandable:NTF . . . . .  
 . . . . . 124, 8724, 20264, 27147  
\token\_if\_expandable\_p:N . . . . . 124, 8724  
\token\_if\_group\_begin:NTF . . . . . 123, 8628  
\token\_if\_group\_begin\_p:N . . . . . 123, 8628  
\token\_if\_group\_end:NTF . . . . . 123, 8633  
\token\_if\_group\_end\_p:N . . . . . 123, 8633  
\token\_if\_int\_register:NTF . . . . .  
 . . . . . 125, 8754, 20269  
\token\_if\_int\_register\_p:N 125, 8754

\token\_if\_letter:N . . . . . 511  
\token\_if\_letter:NTF 124, 8672, 27222  
\token\_if\_letter\_p:N . . . . . 124, 8672  
\token\_if\_long\_macro:NTF . . . . . 124, 8754  
\token\_if\_long\_macro\_p:N . . . . . 124, 8754  
\token\_if\_macro:NTF . . . . .  
 . . . . . 124, 8702, 8808, 9011, 9020, 9029  
\token\_if\_macro\_p:N . . . . . 124, 8702  
\token\_if\_math\_subscript:NTF . . . . .  
 . . . . . 123, 8662  
\token\_if\_math\_subscript\_p:N . . . . .  
 . . . . . 123, 8662  
\token\_if\_math\_superscript:NTF . . . . .  
 . . . . . 123, 8656  
\token\_if\_math\_superscript\_p:N . . . . .  
 . . . . . 123, 8656  
\token\_if\_math\_toggle:NTF . . . . . 123, 8638  
\token\_if\_math\_toggle\_p:N . . . . . 123, 8638  
\token\_if\_mathchardef:NTF . . . . .  
 . . . . . 125, 8754, 20267  
\token\_if\_mathchardef\_p:N . . . . . 125, 8754  
\token\_if\_muskip\_register:NTF . . . . .  
 . . . . . 125, 8754  
\token\_if\_muskip\_register\_p:N . . . . .  
 . . . . . 125, 8754  
\token\_if\_other:NTF . . . . . 124, 8677  
\token\_if\_other\_p:N . . . . . 124, 8677  
\token\_if\_parameter:NTF . . . . . 123, 8648  
\token\_if\_parameter\_p:N . . . . . 123, 8648  
\token\_if\_primitive:NTF . . . . . 126, 8802  
\token\_if\_primitive\_p:N . . . . . 126, 8802  
\token\_if\_protected\_long\_-  
 macro:NTF . . . . . 125, 3570, 8754  
\token\_if\_protected\_long\_macro\_-  
 p:N . . . . . 125, 8754, 27153  
\token\_if\_protected\_macro:NTF . . . . .  
 . . . . . 124, 3569, 8754  
\token\_if\_protected\_macro\_p:N . . . . .  
 . . . . . 124, 8754, 27152  
\token\_if\_skip\_register:NTF . . . . .  
 . . . . . 125, 8754, 20270  
\token\_if\_skip\_register\_p:N 125, 8754  
\token\_if\_space:NTF . . . . . 123, 8667  
\token\_if\_space\_p:N . . . . . 123, 8667  
\token\_if\_toks\_register:NTF . . . . .  
 . . . . . 126, 366, 3728, 8754, 20271  
\token\_if\_toks\_register\_p:N 126, 8754  
\token\_new:Nn . . . . . 29776  
\token\_to\_meaning:N . . . . . 14, 122,  
 510, 514, 2072, 2088, 2845, 3734,  
 3779, 8606, 8708, 8775, 8812, 9014,  
 9023, 9032, 19894, 20260, 20285, 27917  
\token\_to\_str:N . . . . . 5,  
 16, 51, 122, 150, 335, 400, 450,

512, 672, 673, 674, 940, 2074, 2088, 2088, 2269, 2278, 2421, 2444, 2557, 2566, 2598, 2621, 2669, 2674, 2689, 2718, 2719, 2739, 2838, 2845, 2971, 3006, 3013, 3101, 3121, 3134, 3714, 3799, 3885, 3900, 3915, 3935, 3959, 3971, 3993, 4014, 4227, 4864, 4880, 5034, 5331, 5818, 6370, 6640, 7440, 7495, 8324, 8606, 8789, 8790, 8795, 8796, 8797, 8798, 8799, 8800, 9346, 10758, 10759, 10760, 10761, 10762, 10769, 11098, 12021, 12045, 12991, 13031, 13101, 13330, 13345, 13579, 13580, 14060, 14061, 14090, 14259, 14310, 14342, 14362, 14377, 14389, 14390, 14403, 14404, 14429, 14438, 14440, 14465, 14468, 14493, 14495, 14509, 14525, 14543, 14612, 14622, 14623, 14638, 14639, 14972, 15179, 15420, 19332, 19834, 19851, 19902, 19983, 20026, 20056, 20105, 20116, 20118, 20120, 20130, 20166, 20177, 20223, 20259, 20284, 20305, 20625, 20632, 20733, 20737, 21460, 22763, 22995, 23576, 23578, 23760, 24384, 24600, 25281, 26130, 26133, 26356, 26552, 26749, 26750, 26757, 26758, 27078, 27081, 27089, 27674, 27676, 27694, 27695, 29678, 29681, 29788	\tokspre . . . . . 1009, 1809 \tolerance . . . . . 569 \topmark . . . . . 570 \topmarks . . . . . 672, 1486 \topskip . . . . . 571 \tpack . . . . . 1010, 1810 \tracingassigns . . . . . 673, 1487 \tracingcommands . . . . . 572 \tracingfonts . . . . . 1041, 1651 \tracinggroups . . . . . 674, 1488 \tracingifs . . . . . 675, 1489 \tracinglostchars . . . . . 573 \tracingmacros . . . . . 574 \tracingnesting . . . . . 676, 1490 \tracingonline . . . . . 575 \tracingoutput . . . . . 576 \tracingpages . . . . . 577 \tracingparagraphs . . . . . 578 \tracingrestores . . . . . 579 \tracingscantokens . . . . . 677, 1491 \tracingstats . . . . . 580 true . . . . . 202 trunc . . . . . 198 two commands: \c_thirty_two . . . . . 7257 \c_two_hundred_fifty_five . . . . . 7257 \c_two_hundred_fifty_six . . . . . 7257
token internal commands:	
\c_token_A_int . . . . . 8802, 8839 \__token_delimit_by_char":w . . . 8736 \__token_delimit_by_count:w . . . 8736 \__token_delimit_by_dimen:w . . . 8736 \__token_delimit_by_macro:w . . . 8736 \__token_delimit_by_muskip:w . . . 8736 \__token_delimit_by_skip:w . . . 8736 \__token_delimit_by_toks:w . . . 8736 \__token_if_macro_p:w . . . . . 8702 \__token_if_primitive:NNw . . . . . 8802 \__token_if_primitive:Nw . . . . . 8802 \__token_if_primitive_primitive_loop:N . . . . . 8802 \__token_if_primitive_nullfont:N . . . . . 8802 \__token_if_primitive_space:w . . . . . 8802 \__token_if_primitive_undefined:N . . . . . 8802 \__token_tmp:w . . . . . 512, 8737, 8746, 8747, 8748, 8749, 8750, 8751, 8752, 8755, 8789, 8790, 8791, 8792, 8794, 8796, 8797, 8798, 8799, 8800 \toks . . . . . 567, 8800 \toksapp . . . . . 1008, 1808 \toksdef . . . . . 568, 19830	
	U
	\u . . . . . <i>xxi</i> , 907, 27699 \uccode . . . 174, 189, 202, 204, 206, 208, 581 \Uchar . . . . . 1043, 1826 \Ucharcat . . . . . 1044, 1827 \uchyph . . . . . 582 \ucs . . . . . 1257, 2039 \Udelcode . . . . . 1045, 1828 \Udelcodenum . . . . . 1046, 1829 \Udelimiter . . . . . 1047, 1830 \Udelimiterover . . . . . 1048, 1831 \Udelimiterunder . . . . . 1049, 1832 \Uhextensible . . . . . 1050, 1833 \Umathaccent . . . . . 1051, 1834 \Umathaxis . . . . . 1052, 1835 \Umathbinbinspacing . . . . . 1053, 1836 \Umathbinclosespacing . . . . . 1054, 1837 \Umathbininnerspacing . . . . . 1055, 1838 \Umathbinopenspacing . . . . . 1056, 1839 \Umathbinopspacing . . . . . 1057, 1840 \Umathbinordspacing . . . . . 1058, 1841 \Umathbinpunctspacing . . . . . 1059, 1842 \Umathbinrelspacing . . . . . 1060, 1843 \Umathchar . . . . . 1061, 1844 \Umathcharclass . . . . . 1062, 1845 \Umathchardef . . . . . 1063, 1846

\Umathcharfam . . . . .	1064, 1847	\Umathordbinspacing . . . . .	1127, 1910
\Umathcharnum . . . . .	1065, 1848	\Umathordclosespacing . . . . .	1128, 1911
\Umathcharnumdef . . . . .	1066, 1849	\Umathordinnerspacing . . . . .	1129, 1912
\Umathcharslot . . . . .	1067, 1850	\Umathordopenspacing . . . . .	1130, 1913
\Umathclosebinspacing . . . . .	1068, 1851	\Umathorddopsspacing . . . . .	1131, 1914
\Umathcloseclosespacing . . . . .	1069, 1852	\Umathordordspacing . . . . .	1132, 1915
\Umathcloseinnerspacing . . . . .	1071, 1854	\Umathordpunctspacing . . . . .	1133, 1916
\Umathcloseopenspacing . . . . .	1073, 1856	\Umathordrelspacing . . . . .	1134, 1917
\Umathcloseopspacing . . . . .	1074, 1857	\Umathoverbarkern . . . . .	1135, 1918
\Umathcloseordspacing . . . . .	1075, 1858	\Umathoverbarrule . . . . .	1136, 1919
\Umathclosepunctspacing . . . . .	1076, 1859	\Umathoverbarvgap . . . . .	1137, 1920
\Umathcloserelspacing . . . . .	1078, 1861	\Umathoverdelimiterbgap . . . . .	1138, 1921
\Umathcode . . . . .	165, 1079, 1862	\Umathoverdelimitervgap . . . . .	1140, 1923
\Umathcodenum . . . . .	1080, 1863	\Umathpunctbinspacing . . . . .	1142, 1925
\Umathconnectoroverlapmin . . . . .	1081, 1864	\Umathpunctclosespacing . . . . .	1143, 1926
\Umathfractiondelsize . . . . .	1083, 1866	\Umathpunctinnerspacing . . . . .	1145, 1928
\Umathfractiondenomdown . . . . .	1084, 1867	\Umathpunctopenspacing . . . . .	1147, 1930
\Umathfractiondenomvgap . . . . .	1086, 1869	\Umathpuncttopspacing . . . . .	1148, 1931
\Umathfractionnumup . . . . .	1088, 1871	\Umathpunctordspacing . . . . .	1149, 1932
\Umathfractionnumvgap . . . . .	1089, 1872	\Umathpunctpunctspacing . . . . .	1150, 1933
\Umathfractionrule . . . . .	1090, 1873	\Umathpunctrelspacing . . . . .	1152, 1934
\Umathinnerbinspacing . . . . .	1091, 1874	\Umathquad . . . . .	1153, 1935
\Umathinnerclosespacing . . . . .	1092, 1875	\Umathradicaldegreeafter . . . . .	1154, 1936
\Umathinnerinnerspacing . . . . .	1094, 1877	\Umathradicaldegreebefore . . . . .	1156, 1938
\Umathinneropenspacing . . . . .	1096, 1879	\Umathradicaldegreeraise . . . . .	1158, 1940
\Umathinneropspace . . . . .	1097, 1880	\Umathradicalkern . . . . .	1160, 1942
\Umathinnerordspacing . . . . .	1098, 1881	\Umathradicalrule . . . . .	1161, 1943
\Umathinnerpunctspacing . . . . .	1099, 1882	\Umathradicalvgap . . . . .	1162, 1944
\Umathinnerrelspacing . . . . .	1101, 1884	\Umathrelbinspacing . . . . .	1163, 1945
\Umathlimitabovebgap . . . . .	1102, 1885	\Umathrelclosespacing . . . . .	1164, 1946
\Umathlimitabovekern . . . . .	1103, 1886	\Umathrelinnerspacing . . . . .	1165, 1947
\Umathlimitabovevgap . . . . .	1104, 1887	\Umathrelopspacing . . . . .	1166, 1948
\Umathlimitbelowbgap . . . . .	1105, 1888	\Umathrelordspacing . . . . .	1167, 1949
\Umathlimitbelowkern . . . . .	1106, 1889	\Umathrelrelspacing . . . . .	1168, 1950
\Umathlimitbelowvgap . . . . .	1107, 1890	\Umathrelpunctspacing . . . . .	1169, 1951
\Umathnolimitsfactor . . . . .	1108, 1891	\Umathrelrelrelspacing . . . . .	1170, 1952
\Umathnolimitsupfactor . . . . .	1109, 1892	\Umathskewedfractionhgap . . . . .	1171, 1953
\Umathopbinspacing . . . . .	1110, 1893	\Umathskewedfractionvgap . . . . .	1173, 1955
\Umathopclosespacing . . . . .	1111, 1894	\Umathspaceafterscript . . . . .	1175, 1957
\Umathopenbinspacing . . . . .	1112, 1895	\Umathstackdenomdown . . . . .	1176, 1958
\Umathopenclosespacing . . . . .	1113, 1896	\Umathstacknumup . . . . .	1177, 1959
\Umathopeninnerspacing . . . . .	1114, 1897	\Umathstackvgap . . . . .	1178, 1960
\Umathopenopenspacing . . . . .	1115, 1898	\Umathsubshiftdown . . . . .	1179, 1961
\Umathopenopspacing . . . . .	1116, 1899	\Umathsubshiftdrop . . . . .	1180, 1962
\Umathopenordspacing . . . . .	1117, 1900	\Umathsubsupshiftdown . . . . .	1181, 1963
\Umathopenpunctspacing . . . . .	1118, 1901	\Umathsubsupvgap . . . . .	1182, 1964
\Umathopenrelspacing . . . . .	1119, 1902	\Umathsubtopmax . . . . .	1183, 1965
\Umathoperatorsize . . . . .	1120, 1903	\Umathsupbottommin . . . . .	1184, 1966
\Umathopinnerspacing . . . . .	1121, 1904	\Umathsupshiftdrop . . . . .	1185, 1967
\Umathopinspace . . . . .	1122, 1905	\Umathsupshiftup . . . . .	1186, 1968
\Umathopinspace . . . . .	1123, 1906	\Umathsupsubbottommax . . . . .	1187, 1969
\Umathopordspacing . . . . .	1124, 1907	\Umathunderbarkern . . . . .	1188, 1970
\Umathoppunctspacing . . . . .	1125, 1908	\Umathunderbarrule . . . . .	1189, 1971
\Umathoprelspacing . . . . .	1126, 1909	\Umathunderbarvgap . . . . .	1190, 1972

\Umathunderdelimterbgap . . . . . 1191, 1973  
 \Umathunderdelimtervgap . . . . . 1193, 1975  
 undefined commands:  
     .undefine: . . . . . 174, 12572  
 \underline . . . . . 583  
 \unexpanded . . . . . 678, 1492, 3616, 3640  
 \unhbox . . . . . 584  
 \unhcopy . . . . . 585  
 \uniformdeviate . . . . . 1042, 1652  
 \unkern . . . . . 586  
 \unless . . . . . 679, 1493  
 \Unosubscript . . . . . 1195, 1977  
 \Unosuperscript . . . . . 1196, 1978  
 \unpenalty . . . . . 587  
 \unskip . . . . . 588  
 \unvbox . . . . . 589  
 \unvcopy . . . . . 590  
 \Uoverdelimter . . . . . 1197, 1979  
 \uppercase . . . . . 591  
 uptex commands:  
     \uptex\_disablecjktoken:D . . . . . 2033  
     \uptex\_enablecjktoken:D . . . . . 2034  
     \uptex\_forcecjktoken:D . . . . . 2035  
     \uptex\_kchar:D . . . . . 2036  
     \uptex\_kchardef:D . . . . . 2037  
     \uptex\_kuten:D . . . . . 2038  
     \uptex\_ucs:D . . . . . 2039  
     \uptex\_uptexrevision:D . . . . . 2040  
     \uptex\_uptexversion:D . . . . . 2041  
 \uptexrevision . . . . . 1258, 2040  
 \uptexversion . . . . . 1259, 2041  
 \Uradical . . . . . 1198, 1980  
 \Uroot . . . . . 1199, 1981  
 use commands:  
     \use:e . . . . . 18, 2143, 5190  
     \use:N . . . . . 15, 93, 331,  
         2136, 2310, 2312, 2360, 2387, 2389,  
         2593, 2684, 2805, 2807, 2809, 2811,  
         5668, 6638, 7099, 7109, 7214, 7218,  
         7220, 7222, 7223, 7227, 7471, 7493,  
         9617, 9626, 9639, 9646, 9657, 9679,  
         9696, 9703, 9747, 9757, 9765, 10756,  
         10843, 11504, 12235, 12242, 12445,  
         12843, 12844, 20937, 22746, 22885,  
         25332, 26089, 26181, 26894, 26991,  
         27000, 27024, 27042, 28518, 28944  
     \use:n . . . . . 17, 18, 35, 129, 255, 318,  
         380, 496, 553, 651, 673, 838, 848,  
         928, 966, 2137, 2145, 2148, 2242,  
         2526, 2543, 2569, 2629, 2638, 2655,  
         2914, 2991, 3760, 3809, 3983, 4582,  
         4843, 5130, 5217, 5225, 5315, 5336,  
         5350, 7706, 7713, 8177, 8560, 8620,  
         8702, 8739, 8757, 8803, 9581, 9598,  
     \use:n . . . . . 17, 2148, 3178, 4259, 4303,  
         8161, 9005, 11129, 11500, 14173,  
         14182, 14186, 17568, 19376, 20242,  
         26214, 26216, 26221, 26223, 26586  
 \use:nn . . . . . 17, 2148, 2968, 7292  
 \use:nnnn . . . . . 17, 2148  
 \use\_i:nn . . . . . 17, 317, 329, 331,  
     332, 524, 791, 794, 807, 811, 812,  
     1032, 1056, 2092, 2152, 2322, 2415,  
     2513, 2587, 2609, 2755, 2783, 2947,  
     3627, 3657, 3670, 3715, 3966, 4769,  
     5930, 5932, 6272, 7708, 9133, 11273,  
     13311, 14142, 15478, 15814, 16109,  
     16597, 16880, 17397, 17563, 17806,  
     17816, 17820, 18328, 18533, 19100,  
     19125, 19657, 19712, 19722, 19732,  
     20058, 20858, 20869, 20878, 20881,  
     20890, 26253, 26417, 27156, 27263  
 \use\_i:nnn . . . . .  
     . 17, 2154, 4005, 6132, 7316, 9014,  
     14110, 16066, 17538, 19312, 22794  
 \use\_i:nnnn . . . . . 17, 310, 473, 474, 2154,  
     7466, 7468, 7482, 7487, 7503, 7505,  
     15648, 16084, 16091, 16284, 19111  
 \use\_i\_delimit\_by\_q\_nil:nw . 19, 2165  
 \use\_i\_delimit\_by\_q\_recursion\_-  
     stop:nw . . . . . 19,  
     2165, 5730, 5746, 7522, 7548,  
     21688, 26941, 27108, 27131, 27180  
 \use\_i\_delimit\_by\_q\_stop:nw . . . . .  
     . 19, 419, 2165, 2322,  
     2324, 3994, 5429, 5438, 5557, 5608,  
     5611, 8270, 10806, 15452, 15814, 26277  
 \use\_i\_ii:nnn . . . . . 18, 331,  
     332, 2154, 2578, 3204, 6108, 6213, 9304  
 \use\_ii:nn . . . . . 17, 103, 317, 329,  
     524, 791, 794, 805, 807, 810, 811,  
     812, 824, 912, 1415, 1420, 2094,  
     2152, 2418, 2515, 2611, 2630, 2646,  
     2757, 2785, 2945, 3138, 3629, 3672,  
     3717, 4272, 4771, 7714, 9134, 11269,  
     13512, 13535, 15297, 15478, 15479,  
     16111, 17399, 17812, 17818, 17822,  
     18330, 18535, 18966, 19102, 20060,

20860, 20866, 20871, 20883, 20892,  
 21389, 21511, 21681, 21869, 26769,  
 26997, 27012, 27155, 27158, 27227  
`\use_ii:nnn .` 17, 332, 2154, 2646, 9023  
`\use_ii:nnnn .` 17, 473, 474, 2154, 7482  
`\use_iii:nnn` 17, 2154, 3143, 9032, 13317  
`\use_iii:nnnn . . . . .` 17,  
 473, 474, 2154, 7482, 7504, 7506, 7507  
`\use_iv:nnnn . . . . .`  
 17, 473, 474, 2154, 7482, 7502, 15285  
`\use_none:n . . . . .`  
 18, 311, 386, 393, 394, 488,  
 492, 535, 570, 669, 670, 674, 674,  
 806, 812, 865, 1416, 1422, 2169,  
 2250, 2366, 2385, 2577, 2629, 2630,  
 2916, 2972, 3558, 3648, 4308, 4376,  
 4398, 4678, 4768, 4784, 4846, 4863,  
 4873, 4874, 4879, 4894, 4897, 4986,  
 4995, 5732, 5747, 5834, 6117, 6969,  
 6975, 7304, 7707, 7712, 7788, 7832,  
 7929, 8024, 8051, 8090, 8852, 9091,  
 9093, 9098, 9506, 9723, 9949, 9953,  
 10367, 10816, 10871, 10927, 12058,  
 12085, 12108, 12120, 12795, 12854,  
 13079, 13306, 13455, 13459, 13463,  
 13467, 14773, 14999, 15006, 15023,  
 15042, 15065, 15131, 15172, 15297,  
 15312, 15333, 15334, 15540, 15541,  
 16085, 16088, 17068, 18689, 18974,  
 20056, 20105, 20203, 20241, 20475,  
 20746, 20904, 21556, 26298, 26299,  
 26657, 27323, 28108, 29564, 29566  
`\use_none:nn` 18, 324, 385, 389, 398,  
 399, 436, 2169, 2382, 2422, 2559,  
 2567, 2638, 4359, 4502, 4630, 4795,  
 4812, 6002, 6139, 7461, 7758, 8065,  
 10872, 10916, 13371, 13454, 13458,  
 13462, 13466, 18684, 22144, 22807  
`\use_none:nnn . . . . .` 18,  
 399, 2169, 2265, 2274, 2283, 3886,  
 3901, 4834, 10873, 13453, 13457,  
 13461, 13465, 14110, 20272, 20601  
`\use_none:nnnn . . . . .`  
 18, 2169, 4037, 10874, 11646  
`\use_none:nnnnn . . . . .`  
 18, 2169, 2288, 2294, 10875, 10885,  
 13610, 13644, 13670, 13678, 15674  
`\use_none:nnnnnn . . . . .`  
 18, 2169, 2691, 10876, 27867  
`\use_none:nnnnnnn . . . . .`  
 18, 655, 2169, 2352, 13612,  
 13646, 13672, 13680, 13995, 16125  
`\use_none:nnnnnnnn . . . . .` 18, 331, 2169, 2600  
`\use_none:nnnnnnnnn . . . . .` 18, 2169

`\use_none_delimit_by_q_nil:w` 18, 2162  
`\use_none_delimit_by_q_recursion_stop:w . . . . .` 18, 65, 65, 65, 65,  
 2162, 2591, 2670, 2675, 2682, 3800,  
 3807, 4007, 5724, 5739, 21666, 21690  
`\use_none_delimit_by_q_stop:w . . . . .`  
 18, 426, 491, 1072, 2162, 3998,  
 5145, 5427, 5436, 5595, 6654, 8011,  
 8256, 8261, 9776, 11508, 26200, 27919  
`\use_none_delimit_by_s_stop:w . . . . .`  
 66, 67, 5826  
`\useboxresource . . . . .` 1035, 1645  
`\useimageresource . . . . .` 1036, 1646  
`\Uskewed . . . . .` 1200, 1982  
`\Uskewedwithdelims . . . . .` 1201, 1983  
`\Ustack . . . . .` 1202, 1984  
`\Ustartdisplaymath . . . . .` 1203, 1985  
`\Ustartmath . . . . .` 1204, 1986  
`\Ustopdisplaymath . . . . .` 1205, 1987  
`\Ustopmath . . . . .` 1206, 1988  
`\Usubscript . . . . .` 1207, 1989  
`\Usuperscript . . . . .` 1208, 1990  
 utex commands:  
`\utex_binbinspacing:D . . . . .` 1836  
`\utex_binclosespacing:D . . . . .` 1837  
`\utex_bininnerspacing:D . . . . .` 1838  
`\utex_binopenspacing:D . . . . .` 1839  
`\utex_binopspacing:D . . . . .` 1840  
`\utex_binordspacing:D . . . . .` 1841  
`\utex_binpunctspacing:D . . . . .` 1842  
`\utex_binrelspacing:D . . . . .` 1843  
`\utex_char:D . . . . .` 1826  
`\utex_charcat:D . . . . .` 1827  
`\utex_closebinspacing:D . . . . .` 1851  
`\utex_closeclosespacing:D . . . . .` 1853  
`\utex_closeinnerspacing:D . . . . .` 1855  
`\utex_closeopenspacing:D . . . . .` 1856  
`\utex_closeopspaceing:D . . . . .` 1857  
`\utex_closeordspacing:D . . . . .` 1858  
`\utex_closepunctspacing:D . . . . .` 1860  
`\utex_closerelspacing:D . . . . .` 1861  
`\utex_connectoroverlapmin:D . . . . .` 1865  
`\utex_delcode:D . . . . .` 1828  
`\utex_delcodenum:D . . . . .` 1829  
`\utex_delimiter:D . . . . .` 1830  
`\utex_delimiterover:D . . . . .` 1831  
`\utex_delimiterunder:D . . . . .` 1832  
`\utex_fractiondelsize:D . . . . .` 1866  
`\utex_fractiondenomdown:D . . . . .` 1868  
`\utex_fractiondenomvgap:D . . . . .` 1870  
`\utex_fractionnumup:D . . . . .` 1871  
`\utex_fractionnumvgap:D . . . . .` 1872  
`\utex_fractionrule:D . . . . .` 1873  
`\utex_hextensible:D . . . . .` 1833

\utex_innerbinspacing:D .....	1874	\utex_overbarkern:D .....	1918
\utex_innerclosespacing:D .....	1876	\utex_overbarrule:D .....	1919
\utex_innerinnerspacing:D .....	1878	\utex_overbarvgap:D .....	1920
\utex_inneropenspacing:D .....	1879	\utex_overdelimiter:D .....	1979
\utex_inneropspacing:D .....	1880	\utex_overdelimiterbgap:D .....	1922
\utex_innerordspacing:D .....	1881	\utex_overdelimitervgap:D .....	1924
\utex_innerpunctspacing:D .....	1883	\utex_punctbinspacing:D .....	1925
\utex_innerrelspacing:D .....	1884	\utex_punctclosespacing:D .....	1927
\utex_limitabovebgap:D .....	1885	\utex_punctinnerspacing:D .....	1929
\utex_limitabovekern:D .....	1886	\utex_punctopenspacing:D .....	1930
\utex_limitabovevgap:D .....	1887	\utex_puncttopspacing:D .....	1931
\utex_limitbelowbgap:D .....	1888	\utex_punctordspacing:D .....	1932
\utex_limitbelowkern:D .....	1889	\utex_punctpunctspacing:D .....	1933
\utex_limitbelowvgap:D .....	1890	\utex_punctrelspacing:D .....	1934
\utex_mathaccent:D .....	1834	\utex_quad:D .....	1935
\utex_mathaxis:D .....	1835	\utex_radical:D .....	1980
\utex_mathchar:D .....	1844	\utex_radicaldegreeafter:D .....	1937
\utex_mathcharclass:D .....	1845	\utex_radicaldegreebefore:D .....	1939
\utex_mathchardef:D .....	1846	\utex_radicaldegreeraise:D .....	1941
\utex_mathcharfam:D .....	1847	\utex_radicalkern:D .....	1942
\utex_mathcharnum:D .....	1848	\utex_radicalrule:D .....	1943
\utex_mathcharnumdef:D .....	1849	\utex_radicalvgap:D .....	1944
\utex_mathcharslot:D .....	1850	\utex_relbinspacing:D .....	1945
\utex_mathcode:D .....	1862	\utex_relclosespacing:D .....	1946
\utex_mathcodenum:D .....	1863	\utex_relinnerspacing:D .....	1947
\utex_nolimitsfactor:D .....	1891	\utex_reopenspacing:D .....	1948
\utex_nolimitsupfactor:D .....	1892	\utex_relopspacing:D .....	1949
\utex_nosubscript:D .....	1977	\utex_relordspacing:D .....	1950
\utex_nosuperscript:D .....	1978	\utex_relpunctspacing:D .....	1951
\utex_opbinspacing:D .....	1893	\utex_relrelspacing:D .....	1952
\utex_opclosespacing:D .....	1894	\utex_root:D .....	1981
\utex_openbinspacing:D .....	1895	\utex_skewed:D .....	1982
\utex_openclosespacing:D .....	1896	\utex_skewedfractionhgap:D .....	1954
\utex_openinnerspacing:D .....	1897	\utex_skewedfractionvgap:D .....	1956
\utex_openopenspacing:D .....	1898	\utex_skewedwithdelims:D .....	1983
\utex_openopspacing:D .....	1899	\utex_spaceafterscript:D .....	1957
\utex_openordspacing:D .....	1900	\utex_stack:D .....	1984
\utex_openpunctspacing:D .....	1901	\utex_stackdenomdown:D .....	1958
\utex_openrelspacing:D .....	1902	\utex_stacknumup:D .....	1959
\utex_operatorsize:D .....	1903	\utex_stackvgap:D .....	1960
\utex_opinnerspacing:D .....	1904	\utex_startdisplaymath:D .....	1985
\utex_opopenspacing:D .....	1905	\utex_startmath:D .....	1986
\utex_opopspacing:D .....	1906	\utex_stopdisplaymath:D .....	1987
\utex_opordspacing:D .....	1907	\utex_stopmath:D .....	1988
\utex_oppunctspacing:D .....	1908	\utex_subscript:D .....	1989
\utex_oprelspacing:D .....	1909	\utex_subshiftdown:D .....	1961
\utex_ordbinspacing:D .....	1910	\utex_subshiftdrop:D .....	1962
\utex_ordclosespacing:D .....	1911	\utex_subsupshiftdown:D .....	1963
\utex_ordinnerspacing:D .....	1912	\utex_subsupvgap:D .....	1964
\utex_ordopenspacing:D .....	1913	\utex_subtopmax:D .....	1965
\utex_ordopspace:D .....	1914	\utex_supbottommin:D .....	1966
\utex_ordordspacing:D .....	1915	\utex_superscript:D .....	1990
\utex_ordpunctspacing:D .....	1916	\utex_supshiftdrop:D .....	1967
\utex_ordrelspacing:D .....	1917	\utex_supshiftup:D .....	1968

\utex_supsubbottommax:D . . . . .	1969	\vfilneg . . . . .	599
\utex_underbarkern:D . . . . .	1970	\vfuzz . . . . .	600
\utex_underbarrule:D . . . . .	1971	\voffset . . . . .	601
\utex_underbarvgap:D . . . . .	1972	\vpack . . . . .	1011, 1811
\utex_underdelimter:D . . . . .	1991	\vrule . . . . .	602
\utex_underdelimitergap:D . . .	1974	\vsize . . . . .	603
\utex_underdelimitervgap:D . . .	1976	\vskip . . . . .	604
\utex_vextensible:D . . . . .	1992	\vsplit . . . . .	605
\Uunderdelimter . . . . .	1209, 1991	\vss . . . . .	606
\Uvextensible . . . . .	1210, 1992	\vtop . . . . .	607
<b>V</b>			
\v . . . . .	27699		
\vadjust . . . . .	592		
\valign . . . . .	593		
value commands:			
.value_forbidden:n . . . . .	174, 12574		
.value_required:n . . . . .	174, 12574		
\vbadness . . . . .	594		
\vbox . . . . .	595		
vbox commands:			
\vbox:n . . . . .	222, 222, 226, 23850		
\vbox_gset:Nn . . . . .	226, 23864, 24453		
\vbox_gset:Nw . . . . .	226, 23906, 24521		
\vbox_gset_end: . . . . .	226, 23906, 24523		
\vbox_gset_split_to_ht:NNn	227, 23949		
\vbox_gset_to_ht:Nnn . . . . .	226, 23892		
\vbox_gset_to_ht:Nnw . . . . .	227, 23929		
\vbox_gset_top:Nn . . . . .	226, 23878		
\vbox_set:Nn . . . . .	226, 226, 23864, 24447		
\vbox_set:Nw . . . . .	226, 23906, 24514		
\vbox_set_end: . . . . .	226, 227, 23906, 24516		
\vbox_set_split_to_ht:NNn	227, 23949		
\vbox_set_to_ht:Nnn . . . . .	226, 227, 23892		
\vbox_set_to_ht:Nnw . . . . .	227, 23929		
\vbox_set_top:Nn . . . . .			
. . . . .	226, 23878, 24470, 24540		
\vbox_to_ht:nn . . . . .	226, 23854		
\vbox_to_zero:n . . . . .	226, 23854		
\vbox_top:n . . . . .	226, 23850		
\vbox_unpack:N . . . . .	227, 23945, 24470, 24540		
\vbox_unpack_clear:N . . . . .	24325		
\vbox_unpack_drop:N . . . . .			
. . . . .	228, 23945, 24325, 24327		
\vcenter . . . . .	596		
vcoffin commands:			
\vcoffin_gset:Nnn . . . . .	233, 24444		
\vcoffin_gset:Nnw . . . . .	233, 24512		
\vcoffin_gset_end: . . . . .	233, 24512		
\vcoffin_set:Nnn . . . . .	233, 24444		
\vcoffin_set:Nnw . . . . .	233, 24512		
\vcoffin_set_end: . . . . .	233, 24512		
\vfil . . . . .	597		
\vfill . . . . .	598		
<b>W</b>			
\wd . . . . .	608		
\widowpenalties . . . . .	680, 1494		
\widowpenalty . . . . .	609		
\write . . . . .	610		
<b>X</b>			
\xdef . . . . .	611		
xetex commands:			
\xetex_charclass:D . . . . .	1655		
\xetex_charglyph:D . . . . .	1656		
\xetex_countfeatures:D . . . . .	1657		
\xetex_countglyphs:D . . . . .	1658		
\xetex_countselectors:D . . . . .	1659		
\xetex_countvariations:D . . . . .	1660		
\xetex_dashbreakstate:D . . . . .	1662		
\xetex_defaultencoding:D . . . . .	1661		
\xetex_featurecode:D . . . . .	1663		
\xetex_featurename:D . . . . .	1664		
\xetex_findfeaturebyname:D . . . . .	1666		
\xetex_findselectorbyname:D . . . . .	1668		
\xetex_findvariationbyname:D . . . . .	1670		
\xetex_firstfontchar:D . . . . .	1671		
\xetex_fonttype:D . . . . .	1672		
\xetex_generateactualtext:D . . . . .	1674		
\xetex_glyph:D . . . . .	1675		
\xetex_glyphbounds:D . . . . .	1676		
\xetex_glyphindex:D . . . . .	1677		
\xetex_glyphname:D . . . . .	1678		
\xetex_if_engine:TF . . . . .			
. . . . .	29780, 29782, 29784		
\xetex_if_engine_p: . . . . .	29778		
\xetex_inputencoding:D . . . . .	1679		
\xetex_inputnormalization:D . . . . .	1681		
\xetex_interchartokenstate:D . . . . .	1683		
\xetex_interchartoks:D . . . . .	1684		
\xetex_isdefaultselector:D . . . . .	1686		
\xetex_isexclusivefeature:D . . . . .	1688		
\xetex_lastfontchar:D . . . . .	1689		
\xetex_linebreaklocale:D . . . . .	1691		
\xetex_linebreakpenalty:D . . . . .	1692		
\xetex_linebreakskip:D . . . . .	1690		
\xetex_OTcountfeatures:D . . . . .	1693		

\xetex_OTcountlanguages:D . . . . .	1694	\XeTeXglyphname . . . . .	844, 1678
\xetex_OTcountsheets:D . . . . .	1695	\XeTeXinputencoding . . . . .	845, 1679
\xetex_OTfeaturetag:D . . . . .	1696	\XeTeXinputnormalization . . . . .	846, 1680
\xetex_OTlanguagetag:D . . . . .	1697	\XeTeXinterchartokenstate . . . . .	848, 1682
\xetex_OTscripttag:D . . . . .	1698	\XeTeXinterchartoks . . . . .	850, 1684
\xetex_pdffile:D . . . . .	1699	\XeTeXisdefaultselector . . . . .	851, 1685
\xetex_pdfpagecount:D . . . . .	1700	\XeTeXisexclusivefeature . . . . .	853, 1687
\xetex_picfile:D . . . . .	1701	\XeTeXlastfontchar . . . . .	855, 1689
\xetex_selectorname:D . . . . .	1702	\XeTeXlinebreaklocale . . . . .	857, 1691
\xetex_suppressfontnotfounderror:D . . . . .	1654	\XeTeXlinebreakpenalty . . . . .	858, 1692
		\XeTeXlinebreakskip . . . . .	856, 1690
\xetex_tracingfonts:D . . . . .	1703	\XeTeXOTcountfeatures . . . . .	859, 1693
\xetex_upwardsmode:D . . . . .	1704	\XeTeXOTcountlanguages . . . . .	860, 1694
\xetex_useglyphmetrics:D . . . . .	1705	\XeTeXOTcountsheets . . . . .	861, 1695
\xetex_variation:D . . . . .	1706	\XeTeXOTfeaturetag . . . . .	862, 1696
\xetex_variationdefault:D . . . . .	1707	\XeTeXOTlanguagetag . . . . .	863, 1697
\xetex_variationmax:D . . . . .	1708	\XeTeXOTscripttag . . . . .	864, 1698
\xetex_variationmin:D . . . . .	1709	\XeTeXpdffile . . . . .	865, 1699
\xetex_variationname:D . . . . .	1710	\XeTeXpdfpagecount . . . . .	866, 1700
\xetex_XeTeXrevision:D . . . . .	1711	\XeTeXpicfile . . . . .	867, 1701
\xetex_XeTeXversion:D . . . . .	1712	\XeTeXrevision . . . . .	868, 1711
\XeTeXcharclass . . . . .	821, 1655	\XeTeXselectorname . . . . .	869, 1702
\XeTeXcharglyph . . . . .	822, 1656	\XeTeXtracingfonts . . . . .	870, 1703
\XeTeXcountfeatures . . . . .	823, 1657	\XeTeXupwardsmode . . . . .	871, 1704
\XeTeXcountglyphs . . . . .	824, 1658	\XeTeXuseglyphmetrics . . . . .	872, 1705
\XeTeXcountselectors . . . . .	825, 1659	\XeTeXvariation . . . . .	873, 1706
\XeTeXcountvariations . . . . .	826, 1660	\XeTeXvariationdefault . . . . .	874, 1707
\XeTeXdashbreakstate . . . . .	828, 1662	\XeTeXvariationmax . . . . .	875, 1708
\XeTeXdefaultencoding . . . . .	827, 1661	\XeTeXvariationmin . . . . .	876, 1709
\XeTeXfeaturecode . . . . .	829, 1663	\XeTeXvariationname . . . . .	877, 1710
\XeTeXfeaturename . . . . .	830, 1664	\XeTeXversion . . . . .	878, 1712
\XeTeXfindfeaturebyname . . . . .	831, 1665	\xkanjiskip . . . . .	1247, 2029
\XeTeXfindselectorbyname . . . . .	833, 1667	\xleaders . . . . .	612
\XeTeXfindvariationbyname . . . . .	835, 1669	\xspaceskip . . . . .	613
\XeTeXfirstfontchar . . . . .	837, 1671	\xspcode . . . . .	1248, 2030
\XeTeXfonttype . . . . .	838, 1672		
\XeTeXgenerateactualtext . . . . .	839, 1673		<b>Y</b>
\XeTeXglyph . . . . .	841, 1675	\ybaselineshift . . . . .	1249, 2031
\XeTeXglyphbounds . . . . .	842, 1676	\year . . . . .	614
\XeTeXglyphindex . . . . .	843, 1677	\yoko . . . . .	1250, 2032