

# **T<sub>E</sub>X by Topic, A T<sub>E</sub>Xnician's Reference**

Victor Eijkhout

document revision 1.4, December 2013



Copyright © 1991-2013 Victor Eijkhout.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

This document is based on the book  $\text{\TeX}$  by Topic, copyright 1991-2008 Victor Eijkhout. This book was printed in 1991 by Addison-Wesley UK, ISBN 0-201-56882-9, reprinted in 1993, pdf version first made freely available in 2001.

Cover design (lulu.com version): Joanna K. Wozniak (jokwoz@gmail.com)

# 目录

许可证	9
前言	17
<b>1 TeX 处理器的结构</b>	<b>19</b>
1.1 TeX 的 4 个处理器	19
1.2 输入处理器	20
1.3 展开处理器	21
1.4 执行处理器	23
1.5 可视化处理器	23
1.6 示例	24
<b>2 类别码与内部状态</b>	<b>26</b>
2.1 概述	26
2.2 初始处理	26
2.3 类别码	27
2.4 从字符到记号	29
2.5 输入处理器视为有限状态自动机	29
2.6 所有字符皆可信手拈来	30
2.7 内部状态切换	31
2.8 字母符与其他字符	32
2.9 \par 记号	33
2.10 空格	34
2.11 行结束符的更多知识	37
2.12 输入处理器的更多知识	39
2.13 @ 约定	40
<b>3 字符</b>	<b>41</b>
3.1 字符编码	41
3.2 用于字符的控制序列	42
3.3 重音	44

3.4	字符测试	45
3.5	大写和小写	46
3.6	字符相关编码	47
3.7	将记号转换为字符串	47
<b>4</b>	<b>Fonts</b>	<b>49</b>
4.1	Fonts	49
4.2	Font declaration	50
4.3	Font information	51
<b>5</b>	<b>Boxes</b>	<b>55</b>
5.1	Boxes	56
5.2	Box registers	57
5.3	Natural dimensions of boxes	59
5.4	More about box dimensions	62
5.5	Overfull and underfull boxes	65
5.6	Opening and closing boxes	66
5.7	Unboxing	67
5.8	Text in boxes	68
5.9	Assorted remarks	69
<b>6</b>	<b>Horizontal and Vertical Mode</b>	<b>73</b>
6.1	Horizontal and vertical mode	73
6.2	Horizontal and vertical commands	75
6.3	The internal modes	76
6.4	Boxes and modes	77
6.5	Modes and glue	78
6.6	Migrating material	78
6.7	Testing modes	79
<b>7</b>	<b>Numbers</b>	<b>81</b>
7.1	Numbers and $\langle \text{number} \rangle$ s	81
7.2	Integers	82
7.3	Numbers	86
7.4	Integer registers	86
7.5	Arithmetic	87
7.6	Number testing	88
7.7	Remarks	88
<b>8</b>	<b>Dimensions and Glue</b>	<b>90</b>
8.1	Definition of $\langle \text{glue} \rangle$ and $\langle \text{dimen} \rangle$	91
8.2	More about dimensions	95

8.3	More about glue	97
<b>9</b>	<b>Rules and Leaders</b>	<b>103</b>
9.1	Rules	103
9.2	Leaders	105
9.3	Assorted remarks	108
<b>10</b>	<b>编组</b>	<b>111</b>
10.1	编组机制	111
10.2	局部和全局赋值	112
10.3	编组定界符	112
10.4	花括号进阶	113
<b>11</b>	<b>宏定义</b>	<b>115</b>
11.1	介绍	115
11.2	宏定义的结构	116
11.3	前缀	116
11.4	定义的类型	117
11.5	参数文本	117
11.6	构造控制序列	121
11.7	用 <code>\let</code> 和 <code>\futurelet</code> 给出记号赋值	122
11.8	杂项注记	123
11.9	宏的技巧	125
<b>12</b>	<b>Expansion</b>	<b>130</b>
12.1	Introduction	130
12.2	Ordinary expansion	131
12.3	Reversing expansion order	132
12.4	Preventing expansion	135
12.5	<code>\relax</code>	137
12.6	Examples	140
<b>13</b>	<b>Conditionals</b>	<b>147</b>
13.1	The shape of conditionals	148
13.2	Character and control sequence tests	148
13.3	Mode tests	150
13.4	Numerical tests	150
13.5	Other tests	151
13.6	The <code>\newif</code> macro	152
13.7	Evaluation of conditionals	153
13.8	Assorted remarks	154

<b>14 Token Lists</b>	<b>161</b>
14.1 Token lists	161
14.2 Use of token lists	161
14.3 <token parameter>	162
14.4 Token list registers	162
14.5 Examples	163
<b>15 基线距离</b>	<b>166</b>
15.1 行间粘连	166
15.2 盒子深度	168
15.3 术语	169
15.4 补充说明	169
<b>16 Paragraph Start</b>	<b>171</b>
16.1 When does a paragraph start	171
16.2 What happens when a paragraph starts	172
16.3 Assorted remarks	172
16.4 Examples	173
<b>17 Paragraph End</b>	<b>177</b>
17.1 The way paragraphs end	177
17.2 Assorted remarks	178
<b>18 段落形状</b>	<b>181</b>
18.1 文本行的宽度	182
18.2 段落形状参数	182
18.3 杂项注记	183
<b>19 Line Breaking</b>	<b>187</b>
19.1 Paragraph break cost calculation	188
19.2 The process of breaking	192
19.3 Discretionaries	193
19.4 Hyphenation	194
19.5 Switching hyphenation patterns	197
<b>20 Spacing</b>	<b>199</b>
20.1 Introduction	199
20.2 Automatic interword space	200
20.3 User interword space	200
20.4 Control space and tie	201
20.5 More on the space factor	202

<b>21 Characters in Math Mode</b>	<b>205</b>
21.1 Mathematical characters	206
21.2 Delimiters	207
21.3 Radicals	209
21.4 Math accents	210
<b>22 Fonts in Formulas</b>	<b>211</b>
22.1 Determining the font of a character in math mode	211
22.2 Initial family settings	212
22.3 Family definition	212
22.4 Some specific font changes	213
22.5 Assorted remarks	214
<b>23 Mathematics Typesetting</b>	<b>215</b>
23.1 Math modes	217
23.2 Styles in math mode	217
23.3 Classes of mathematical objects	219
23.4 Large operators and their limits	220
23.5 Vertical centring: <code>\vcenter</code>	221
23.6 Mathematical spacing: <code>\mu glue</code>	221
23.7 Generalized fractions	223
23.8 Underlining, overlining	224
23.9 Line breaking in math formulas	224
23.10 Font dimensions of families 2 and 3	224
<b>24 Display Math</b>	<b>227</b>
24.1 Displays	227
24.2 Displays in paragraphs	228
24.3 Vertical material around displays	228
24.4 Glue setting of the display math list	229
24.5 Centring the display formula: <code>displacement</code>	230
24.6 Equation numbers	230
24.7 Non-centred displays	231
<b>25 Alignment</b>	<b>233</b>
25.1 Introduction	233
25.2 Horizontal and vertical alignment	234
25.3 The preamble	235
25.4 The alignment	238
25.5 Example: math alignments	241



<b>26 Page Shape</b>	<b>243</b>
26.1 The reference point for global positioning	243
26.2 <code>\topskip</code>	243
26.3 Page height and depth	244
<b>27 分页</b>	<b>246</b>
27.1 当前页面与备选内容	247
27.2 激活页面构建器	247
27.3 页面长度的记录	247
27.4 分页点	248
27.5 分割竖列	250
27.6 分页的例子	251
<b>28 Output Routines</b>	<b>254</b>
28.1 The <code>\output</code> token list	254
28.2 Output and <code>\box255</code>	255
28.3 Marks	256
28.4 Assorted remarks	257
<b>29 Insertions</b>	<b>261</b>
29.1 Insertion items	261
29.2 Insertion class declaration	262
29.3 Insertion parameters	262
29.4 Moving insertion items from the contributions list	263
29.5 Insertions in the output routine	264
29.6 Plain $\text{\TeX}$ insertions	265
<b>30 File Input and Output</b>	<b>267</b>
30.1 Including files: <code>\input</code> and <code>\endinput</code>	267
30.2 File I/O	268
30.3 Whatsits	270
30.4 Assorted remarks	270
<b>31 Allocation</b>	<b>274</b>
31.1 Allocation commands	275
31.2 Ground rules for macro writers	276
<b>32 Running <math>\text{\TeX}</math></b>	<b>277</b>
32.1 Jobs	277
32.2 Run modes	279

<b>33 T<sub>E</sub>X and the Outside World</b>	<b>280</b>
33.1 T <sub>E</sub> X, IniT <sub>E</sub> X, VirT <sub>E</sub> X	280
33.2 More about formats	283
33.3 The dvi file	285
33.4 Specials	286
33.5 Time	287
33.6 Fonts	287
33.7 T <sub>E</sub> X and web	289
33.8 The T <sub>E</sub> X Users Group	290
<b>34 Tracing</b>	<b>291</b>
34.1 Meaning and content: <code>\show</code> , <code>\showthe</code> , <code>\meaning</code>	292
34.2 Show boxes: <code>\showbox</code> , <code>\tracingoutput</code>	293
34.3 Global statistics	295
<b>35 Errors, Catastrophes, and Help</b>	<b>296</b>
35.1 Error messages	296
35.2 Overflow errors	298
<b>36 The Grammar of T<sub>E</sub>X</b>	<b>302</b>
36.1 Notations	302
36.2 Keywords	303
36.3 Specific grammatical terms	303
36.4 Differences between T <sub>E</sub> X versions 2 and 3	305
<b>37 Glossary of T<sub>E</sub>X Primitives</b>	<b>306</b>
<b>38 编码表格</b>	<b>327</b>
38.1 字符编码表	328
38.2 计算机现代字体	330
38.3 Plain T <sub>E</sub> X 数学符号	335
<b>索引</b>	<b>341</b>
<b>参考文献</b>	<b>346</b>
<b>版本历史</b>	<b>350</b>

# GNU 自由文档许可证

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format

whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the read-

ing or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

### 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the

contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. O. Preserve any Warranty Disclaimers. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are



multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

# 前言

To the casual observer,  $\text{\TeX}$  is not a state-of-the-art typesetting system. No flashy multilevel menus and interactive manipulation of text and graphics dazzle the onlooker. On a less superficial level, however,  $\text{\TeX}$  is a very sophisticated program, first of all because of the ingeniousness of its built-in algorithms for such things as paragraph breaking and make-up of mathematical formulas, and second because of its almost complete programmability. The combination of these factors makes it possible for  $\text{\TeX}$  to realize almost every imaginable layout in a highly automated fashion.

Unfortunately, it also means that  $\text{\TeX}$  has an unusually large number of commands and parameters, and that programming  $\text{\TeX}$  can be far from easy. Anyone wanting to program in  $\text{\TeX}$ , and maybe even the ordinary user, would seem to need two books: a tutorial that gives a first glimpse of the many nuts and bolts of  $\text{\TeX}$ , and after that a systematic, complete reference manual. This book tries to fulfil the latter function. A  $\text{\TeX}$ er who has already made a start (using any of a number of introductory books on the market) should be able to use this book indefinitely thereafter.

In this volume the universe of  $\text{\TeX}$  is presented as about forty different subjects, each in a separate chapter. Each chapter starts out with a list of control sequences relevant to the topic of that chapter and proceeds to treat the theory of the topic. Most chapters conclude with remarks and examples.

Globally, the chapters are ordered as follows. The chapters on basic mechanisms are first, the chapters on text treatment and mathematics are next, and finally there are some chapters on output and aspects of  $\text{\TeX}$ 's connections to the outside world. The book also contains a glossary of  $\text{\TeX}$  commands, tables, and indexes by example, by control sequence, and by subject. The subject index refers for most concepts to only one page, where most of the information on that topic can be found, as well as references to the locations of related information.

This book does not treat any specific  $\text{\TeX}$  macro package. Any parts of the plain format that are treated are those parts that belong to the ‘core’ of plain

$\text{T}_{\text{E}}\text{X}$ : they are also present in, for instance,  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ . Therefore, most remarks about the plain format are true for  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ , as well as most other formats. Putting it differently, if the text refers to the plain format, this should be taken as a contrast to pure  $\text{I}_{\text{N}}\text{T}_{\text{E}}\text{X}$ , not to  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ . By way of illustration, occasionally macros from plain  $\text{T}_{\text{E}}\text{X}$  are explained that do not belong to the core.

### **Acknowledgment**

I am indebted to Barbara Beeton, Karl Berry, and Nico Poppelier, who read previous versions of this book. Their comments helped to improve the presentation. Also I would like to thank the participants of the discussion lists  $\text{T}_{\text{E}}\text{Xhax}$ ,  $\text{T}_{\text{E}}\text{X-nl}$ , and `comp.text.tex`. Their questions and answers gave me much food for thought. Finally, any acknowledgement in a book about  $\text{T}_{\text{E}}\text{X}$  ought to include Donald Knuth for inventing  $\text{T}_{\text{E}}\text{X}$  in the first place. This book is no exception.

Victor Eijkhout

Urbana, Illinois, August 1991

Knoxville, Tennessee, May 2001

Austin, Texas, December 2013

# 第 1 章 $\text{T}_{\text{E}}\text{X}$ 处理器的结构

这本书覆盖了  $\text{T}_{\text{E}}\text{X}$  多个方面的知识，其中每一章都对应着一个相对较小且易于讲述的主题。不过，在开始各个主题之前，有必要对  $\text{T}_{\text{E}}\text{X}$  的工作机理进行一个概述，这就是本章的主题，显然其中许多细节应当忽略，留待后续各章讲述。另外，为了内容的完整性，本章的末尾还会给出几份示例，它们在后续的一些章节中还会被重提。

## 1.1 $\text{T}_{\text{E}}\text{X}$ 的 4 个处理器

$\text{T}_{\text{E}}\text{X}$  处理输入的方式可分为 4 个层面。你可以认为  $\text{T}_{\text{E}}\text{X}$  处理器（很多文档也称为  $\text{T}_{\text{E}}\text{X}$  引擎）是分成了 4 个独立的阶段，每个阶段都接受前一阶段的输出，其处理结果则作为后一阶段的输入。第一个阶段的输入来自 `.tex` 文件，最后一个阶段的输出是 `.dvi` 文件

通常而言，将上述的 4 个层面理解为完整传承的 4 个工作阶段比较直观且易于理解，但事实上这种理解并不正确，因为这 4 个层面是同时活动的，并且彼此之间互通有无。

这 4 个层面（使用 Knuth 最初的术语，可分别大致理解为‘眼睛’、‘嘴巴’、‘胃’和‘肠道’）如下：

1. 输入处理器：这是  $\text{T}_{\text{E}}\text{X}$  从文件中接受文本行输入并将其处理为记号的部分。记号是  $\text{T}_{\text{E}}\text{X}$  的内部对象：有构成所排版文本的字符记号，有可被后续两个层面作为命令进行处理的控制序列记号。
2. 展开处理器：在第一层面所产生的一些（并非全部）记号 – 宏、条件式以及  $\text{T}_{\text{E}}\text{X}$  的一些原始命令 – 都是要被展开的目标。所谓展开，就是将一些记号序列替换为其它记号的过程。
3. 执行处理器：那些不能再被进一步展开的控制序列在这一层会被执行。这里需要关注的部分是  $\text{T}_{\text{E}}\text{X}$  内部状态的变化：赋值（包括宏定义）是该层最典型的行为，另外就是水平列、竖直列和数学列的构建。
4. 可视化处理器：在这一层面完成  $\text{T}_{\text{E}}\text{X}$  排版内容的可视化。诸如水平列会被划

分为段，竖列会被划分为页，数学列会被构建为公式。此外此层面还会输出 dvi 文件【译注：对于现代 TeX 处理器而言，还可以是 pdf 和 xml 等文件】。在这一层面工作的算法对于用户而言是不可见的，但是用户可通过一部分参数去控制它们。

## 1.2 输入处理器

TeX 的输入处理器是 TeX 从输入文件所接受的任何字符翻译为记号的部分，它输出记号流：记号列表。大部分记号可归为两类：字符记号与控制序列记号。还有一类是参数记号，但本章不讨论它。

### 1.2.1 字符的输入

对于所输入的简单文本，字符会被直接转换为字符记号。不过，TeX 会忽略这样一些字符：将多个连续的空格符号仅视为一个空格。还有，TeX 自身会向记号列表中插入一些不太明显的记号，例如行尾的空格记号，每个空行之后的 `\par` 记号。

字符可被输入处理器转换为字符记号，但是这并非意味着所有字符可被排印（排版及印刷）。在 TeX 中，字符被划分为 16 类——每一类都有特定的功能——其中仅有两类字符可被排印。其它字符类，像 `{`、`}`、`&` 和 `#`，都是不可被排印的。一个字符记号可视为一对数字：字符码和类别码。字符码通常使用 `ascii` 编码。一个字符对应的类别码是可以修改的。

有一种字符叫做转义字符，默认是 `\`。当这种字符出现在输入的文本中时，为了生成它的记号，TeX 输入处理器的行为会非常复杂。基本上，输入处理器会将尾随于转义字符的字符序列处理为单个记号，从而构建一个控制序列记号。

对于类别码的处理，TeX 输入处理器的行为可想像为在三种内部状态（*N*，新行；*M*，行内；*S*，忽略空格）中切换的机器。在第 2 章中我们会讨论这些状态及其切换。

### 1.2.2 输入处理器的两个层面

实际上 TeX 的输入处理器自身又可划分为两个层面。由于终端、编辑器或者操作系统的限制，用户可能无法输入一些所需的字符。所以 TeX 提供了使用两个上标字符来获取各个有效字符的编码的机制。这一机制可视为 TeX 输入处理过程中的一个独立的阶段，在时序上要早于上一节中的三态状态机。

例如，`^^+` 这个字串的输入会被处理为 `k`，这是因为 `k` 和 `+` 的 `ascii` 码之差为 64，由于这个替换过程发生在形成字符记号之前，所以输入 `\vs^^+ip 5cm` 与输入 `\vskip 5cm` 是等价的。还有一些例子比这个例子更有用。

上述过程即为 TeX 输入处理器的第一层面，所做的工作是将字符转换为字符，并不考虑类别码的问题，后者是在第二层面（三态状态机）产生的，它与字符码组

合成字符记号。

## 1.3 展开处理器

**T<sub>E</sub>X** 的展开处理器可以接受记号流并且对其中的记号逐一进行展开，直至记号流中所有的记号都是不可展开的。最有代表性的例子是宏的展开：如果一个控制序列记号是一个宏名，那么该记号（可能还包括它的参数记号）会被替换为这个宏的定义文本。

展开处理器所接受的记号流主要来自输入处理器，展开结果是一串不可展开的记号流，将会交给执行处理器。

然而，在处理 `\edef` 或 `\write` 等时也会执行展开处理器。在展开时，这些命令的参数记号列很像在顶层而非在命令参量中的。

### 1.3.1 记号的展开过程

展开处理器展开一个记号的步骤如下：

1. 查看这个记号是否可展开。
2. 如果记号不可展开，那么就将其放入当前构建的记号列表中，然后读入下一个记号。
3. 如果记号可展开，那么它可以展开成什么，就将其替换为什么。对于不带参数的宏以及像 `\jobname` 这样的一些原始命令，只需进行简单的记号序列替换即可。不过，通常 **T<sub>E</sub>X** 需要从记号流中吸收一些参量记号，以形成当前记号的替换文本。例如，对于一个带参数的宏，那么就需要从记号流中析取足够的记号，以形成与它的参数对应的参量。
4. 对于当前记号的展开结果中的第一个记号，返回至步骤 1 继续进行展开。

判断一个记号是否可展开很简单。宏和活动字符、条件式以及一部分 **T<sub>E</sub>X** 原始命令（见第 131 页的列表）都是可展开的，其他记号则都是不可展开的。展开处理器根据这个判定规则将宏的记号替换为它的定义，对条件式进行计算以忽略无关的记号。不过，对于像 `\vskip` 这样的记号和字符记号，包括像美元符和花括号这样的字符，展开处理器会将它们原封不动地传送到执行处理器。

### 1.3.2 几个特例

如上文所述，在一个记号被展开后，**T<sub>E</sub>X** 会对其展开结果中的记号继续进行展开。但是 `\expandafter` 这个控制序列初看破坏了这个游戏规则，因为它只做一步展开。实际发生的事情是：记号列

```
\expandafter(token1)(token2)
```

会被替换为

$\langle token_1 \rangle \langle expansion\ of\ token_2 \rangle$

而这个替换结果还会被展开处理器再次处理。

然而确实存在不遵守展开处理器游戏规则的情况。如果当前处理的记号是 `\noexpand` 控制序列，那么展开处理器把它的下一个记号视为不可展开的：展开处理器如同 `\relax` 那样处理这个记号，直接将其传送到所构建的记号列表中。

例如下面这个宏的定义：

```
\edef\ a{\noexpand\ b}
```

替换文本 `\noexpand\ b` 会在宏定义时被展开。`\noexpand` 的展开结果是其后的那个记号临时改变成 `\relax` 的含义。因此，在展开处理器处理下个记号 `\ b` 时，就将它视为不可展开的，而直接将它扔到在建的记号列表中，从而 `\ b` 就是这个宏的替换文本。

还有一种特例：在 `\edef` 语句里，`\the\langle token\ variable \rangle` 的展开结果是不会被进一步展开的。

### 1.3.3 展开处理器中的花括号

上一节提到，花括号会被展开处理器作为不可展开的字符记号忽略。通常而言这个说法是正确的。例如下面这个 `\romannumeral` 控制序列：

```
\romannumeral1\ number\ count2 3{4 ...
```

它会被 TeX 展开至花括号处停止：如果 `\count2` 的值是 0，那么这个控制序列的展开结果是 103 的罗马数字表示。

另外一个例子，对于

```
\iftrue {\else }\fi
```

展开处理器将使用与

```
\iftrue a\else b\fi
```

完全类似的方式进行处理，结果是 `{` 字符记号，与它的类别码无关。

但是在宏展开的环境中，展开处理器需要识别和处理花括号。首先，配对的花括号可以标定一组记号用于形成一个参量，例如下面这个带有 1 个参数的宏：

```
\def\ macro#1{ ... }
```

你可以使用单个记号作为参量调用这个宏，如下：

```
\macro 1 \macro \$
```

也可以使用配对花括号包围的一组记号作为这个宏的参量，如下：

```
\macro {abc} \macro {d{ef}g}
```

其次，对于带参数的宏，未配对花括号内的表达式不能形成宏参量。例如：

```
\def\ a#1\ stop{ ... }
```

它的参量由第一次出现并且不在配对花括号内的 `\stop` 之前的记号组成：对于

```
\a bc{d\ stop}e\ stop
```

其中 `\a` 的参量是 `bc{d\ stop}e`。这里只接受平衡表达式作为参量。



## 1.4 执行处理器

执行处理器用于构建水平、竖直和数学列表。与这些列表相应，执行处理器在水平、竖直和数学模式中运行。这三种模式每种都有‘内部的’和‘外部的’两个类型。执行处理器在构建列表的过程中，还需要进行一些与模式无关的操作，例如赋值。

执行处理器的输入来自展开处理器的输出，是一个不可展开的记号流。从执行处理器的角度来看，这条记号流所包含的记号有两种类型：

- 用于赋值的记号（包括宏定义）以及像 `\show`、`\aftergroup` 这样执行与模式无关操作的记号。
- 用于构建列表的记号：字符、盒子和粘连。对这些记号的处理方式依赖执行处理器当前处于的模式。

有些记号可以用于任何模式，例如盒子既可以出现在水平模式、竖直模式，也可以出现在数学模式中，但是这些对象的作用与效果需要依赖于具体的模式。其他记号是模式专用的，例如字符记号（确切的说是类别码为 **11** 和 **12** 的字符记号）只能用于水平模式，这意味着：当执行处理器在竖直模式中遇到字符记号时，便会转入水平模式中工作。

并非所有的字符记号都是可排印的，例如在 **T<sub>E</sub>X** 的默认状态中，执行处理器会将 `$` 作为数学模式的切换符，并将 `{` 和 `}` 作为编组的起止符。数学模式切换符用于告知执行处理器进入和退出数学模式，而花括号让执行处理器进入和退出一个编组。

控制序列 `\relax` 需要在此关注一下，它是横跨展开处理器与执行处理器两界的特殊公民，在展开处理器中它是不可展开的，在执行处理器中它什么也不执行，但是它并非一无是处，可以比较下面的两个示例的效果，从中发现 `\relax` 的用途。

示例 1:

```
\count0=1\relax 2
```

示例 2:

```
\def\empty{}
\count0=1\empty 2
```

这两个示例都是在为计数寄存器赋值，但是示例 1 赋的值为 1，而示例 2 赋的值为 12。这是因为在示例 1 中，`\relax` 在执行处理器获得数值 1 的时候阻断了它进一步获取数值 2，而在示例 2 中 `\empty` 的展开结果为空，执行处理器轻而易举地继 1 之后就拿到了 2，所以形成 12。

## 1.5 可视化处理器

**T<sub>E</sub>X** 的可视化处理器包含了用户不可直接控制的一些算法，用于处理断行、阵列、分页、数学排版以及 `dvi` 文件生成等。用户可以通过一些参数间接控制 **T<sub>E</sub>X**

的这部分操作。

可视化处理器中有一部分算法返回的是可被执行处理器处理的结果。例如，已完成断行的段落是一组带有行间粘连和惩罚的水平盒子，并被添加到主竖直列中。再者，分页算法会将其处理结果存储在 `\box255` 中，以使输出例程能够产生页面。另一方面，数学公式不可以被分解，而输送至 `dvi` 文件的盒子也是不可逆的。

## 1.6 示例

### 1.6.1 被忽略的空格

被忽略的空格可以反映数据在 TeX 各层处理器之间的流动情况。例如：

```
\def\af{\penalty200}  
\a 0
```

展开的结果并非是（这将放置值为 200 的惩罚项，并排印数字 0）

```
\penalty200 0
```

而是

```
\penalty2000
```

这是由于 `\a` 后的空格会被输入处理器忽略，从而展开处理器所得到的控制序列是

```
\a0
```

### 1.6.2 内部量值及其表示

TeX 拥有多种内部量值，诸如整数和尺寸。这些内部量值的外部表示方法只有一种，那就是字符串表示，例如 4711 或者 91.44cm。

内部量值与外部表示之间的转换分别发生在两个不同的层面，具体依赖于转换的方向。对于字符串转换为内部量值，例如：

```
\pageno=12 \baselineskip=13pt
```

或者

```
\vskip 5.71pt
```

像这样的语句会在执行处理器中被处理。

另一方面，内部量值到外部表示的转换是由展开处理器完成的。例如：

```
\number\pageno \romannumeral\year  
\the\baselineskip
```

这些语句会被展开处理器处理为内部量值的字符串记号。

最后一个例子，假设 `\count2=45`，看下面的语句：

```
\count0=1\number\count2 3
```

展开处理器可将 `\number\count2` 展开为字符串 45，而 2 之后的空格并不会结束正在赋予的数值：它只用于定界 `\count` 寄存器的数字。从而下一层级的执行处理器看到的是：

```
\count0=1453
```

于是它便奉命行事。

## 第 2 章 类别码与内部状态

在读取字符时， $\text{\TeX}$  以类别码赋之。 $\text{\TeX}$  的输入处理器有三种内部状态，而且输入处理器在这三种内部状态之间的转换以字符的类别码作为表征。本章主要讲述  $\text{\TeX}$  如何读取字符以及类别码如何影响它的读取行为，附加讨论一下有关空格与行尾的问题。

`\endlinechar` 添加到输入行末尾的行结束符的字符码。`\iniTeX` 默认为 13。

`\par` 结束当前段落并进入竖直模式。可以用空行生成。

`\ignorespaces` 读取并展开直到遇到非 `\space token`。

`\catcode` 查询或者设置类别码。

`\ifcat` 检测两个字符的类别码是否相同。

`\_` 控制空格。插入与 `\spacefactor = 1000` 时的空格记号相同大小的空白。

`\obeylines` 用于保留行结束符的 **Plain  $\text{\TeX}$**  宏。

`\obeyspaces` 用于保留（大多数）空格的 **Plain  $\text{\TeX}$**  宏。

### 2.1 概述

$\text{\TeX}$  的输入处理器从文件或终端中扫描输入的文本行，将字符转化为记号。输入处理器可视为一种简单的有限状态自动机，具有三种内部状态，不同的状态对应不同的扫描行为。本章分别从内部状态和类别码这两个角度考察这个自动机。

### 2.2 初始处理

$\text{\TeX}$  对输入文件（也可能是来自终端的输入，但实际很少有人使用，下文不再刻意提它）是逐行处理的，因此首先要讨论  $\text{\TeX}$  输入处理器是如何识别输入行的。

不同的计算机系统对输入行有不同的定义。最常见的方式是采用回车符加换行符作为行终止符，但是有些系统只使用换行符，还有一些系统是固定宽度的输入

行（块存储）而根本不使用终止符。为了对这些系统一视同仁， $\text{\TeX}$  必须要掌控输入行的终止方式，大致步骤如下：

1. 从输入文件读取一行（去掉输入行终止符，如果有的话）。
2. 移除行尾空格（这是针对采用块存储的系统的操作，而且也避免了混乱，因为在编辑器中行尾空格通常是不可见的）。
3. 将 `\endlinechar`（默认为 `\return`，其 `ascii` 码为 13）添加到输入行尾部。如果 `\endlinechar` 的值为负值或者大于 255（在  $\text{\TeX}$  3 之前则为大于 127；见第 305 页介绍的更多差异），行尾不需要添加字符；其效果与该行以注释符结尾相同。

不同的计算机系统可能在字符编码方面也存在区别（最常见的编码是 `ascii` 和 `ebcdic`），因此  $\text{\TeX}$  必须要将文件输入的字符编码转换为它的内部编码，藉此  $\text{\TeX}$  可以兼容任何系统中的字符编码。更多内容详见第 3 章。

## 2.3 类别码

256 个字符码（0–255）的每一个都关联一个不尽相同的类别码。共有 16 个类别，编号从 0 到 15。在扫描输入行的过程中， $\text{\TeX}$  会生成（字符码，类别码）对。 $\text{\TeX}$  的输入处理器的眼里只有（字符码，类别码）对，从中生成字符记号、控制序列记号和参数记号。这些记号随后被传送到  $\text{\TeX}$  的展开处理器与执行处理器。

字符记号是（字符码，类别码）对，它在展开处理器与执行处理器中不会被改变。控制序列记号是由一个或多个前缀为转义符的字符构成，详见下文。参数记号的解释也详见下文。

下面是 16 个类别列表的大致解释，更多的细节知识散布于后文以及后续各章之中。

0. 转义符：用于表示控制序列的开始。`Ini\TeX` 使用反斜线 `\`（ASCII 码为 92）作为转义符。
1. 组开始符：此类字符可让  $\text{\TeX}$  进入新一层的编组。在 `Plain \TeX` 中，组开始符默认是 `{`。
2. 组结束符：此类字符可让  $\text{\TeX}$  结束当前层的编组。在 `Plain \TeX` 中，组结束符默认是 `}`。
3. 数学切换符：置于数学公式两侧，向  $\text{\TeX}$  表示这是数学公式。在 `Plain \TeX` 中，数学切换符默认为 `$`。
4. 制表符：在 `\halign`（`\valign`）制作的表格中作为列（行）的分割符。在 `Plain \TeX` 中，制表符默认为 `&`。
5. 行结束符：用于表示此处为输入行的结束之处。`Ini\TeX` 默认将 `\return` 字符（ASCII 码为 13）视为行结束符，所以 `Ini\TeX` 将 13 作为 `\endlinechar` 的值。

并非巧合；见下面所述。

6. 参数符：用于表示宏的参数。**Plain T<sub>E</sub>X** 默认使用 # 作为参数符。
7. 上标符：在数学模式中用于表示上标，也可用于表示那些无法直接在文本中输入的字符；见下面所述。**Plain T<sub>E</sub>X** 默认使用 ^ 作为上标符。
8. 下标符：在数学模式中用于表示下标。**Plain T<sub>E</sub>X** 使用下划线 \_ 作为下标符。
9. 可忽略符：**T<sub>E</sub>X** 将会从输入中去掉此类字符，因此它不会影响 **T<sub>E</sub>X** 的后续处理。**Plain T<sub>E</sub>X** 使用 `<null>` 字符（ASCII 码为 0）作为可忽略符。
10. 空格符：这个符号会受到 **T<sub>E</sub>X** 的特殊礼遇，它默认被 **IniT<sub>E</sub>X** 赋予 `<space>` 字符（ASCII 码为 32）。
11. 字母符：对于该类字符，**IniT<sub>E</sub>X** 只定义了 `a..z` 和 `A..Z` 这些。通常在写宏包的时候，为了避免宏名冲突，宏包作者通常会将某些非字母符（例如 @）打扮为字母符而使用。
12. 其他字符：**IniT<sub>E</sub>X** 将不属于其他 15 类的字符归到该类，最常见的是数字、标点符号等。
13. 活动符：活动符在功能上相当于 **T<sub>E</sub>X** 控制序列，但是它不需要转义符作为前缀。在 **Plain T<sub>E</sub>X** 中只有 ~ 是活动符，用于产生不可断行的空格；见第 201 页。
14. 注释符：**T<sub>E</sub>X** 将忽略从注释符开始的该行所有字符。**IniT<sub>E</sub>X** 使用分号 % 作为注释符。
15. 无效符：这个字符类是为那些不应该在 **T<sub>E</sub>X** 输入中出现的字符而设置的。**IniT<sub>E</sub>X** 将 `<delete>` 字符（ASCII 码为 127）归入此类。

用户可以修改任意字符的类别码，途径是使用 `\catcode` 命令（见第 36 章对诸如 `<equals>` 的概念的解释）：

```
\catcode<number><equals><number>.
```

此语句的第一个参数是需要修改类别码的字符的编码，它通常可用下面形式给出：

```
`<character> 或 ``\<character>
```

这两种写法都表示该字符的字符码（见第 41 和 82 页）。

**Plain T<sub>E</sub>X** 格式将 `\active` 定义为：

```
\chardef\active=13
```

因此你可以像下面这样写

```
\catcode`\{=\active
```

上面的 `\chardef` 命令将在第 42 和 83 页中介绍。

**L<sup>A</sup>T<sub>E</sub>X** 格式有下面这样的控制序列：

```
\def\makeatletter{\catcode`\@=11 }
\def\makeatother{\catcode`\@=12 }
```

它可用于开启或关闭“隐秘”字符 @（见下述）。

`\catcode` 命令也可用于查询类别码，例如：

```
\count255=\catcode`\{
```

所得类别码存储于第 255 号计数寄存器。

类别码可使用以下命令进行测试：

```
\ifcat<token1><token2>
```

无论 `\ifcat` 之后跟随的是些什么东西，**T<sub>E</sub>X** 都会将其展开，直至发现两个不可展开的记号为止，然后去比较这两个记号的类别码是否相等。控制序列的类别码被视为 16，这样它们的类别码都是相等的，而控制序列与字符记号的类别码总不相等。条件语句在第 13 章中将会详细介绍。

## 2.4 从字符到记号

**T<sub>E</sub>X** 的输入处理器对来自文件或用户终端的输入行进行扫描，将其中的字符转化为记号。记号的类型分为以下三种：

- 字符记号：任何本身会被传递到 **T<sub>E</sub>X** 后续处理器并具有相应的类别码的字符。
- 控制序列记号：这种记号分为两种类型，第一种类型是控制词，由转义符（即类别码为 0 的字符）后跟一串‘字母’而成；第二种类型是控制符，由转义符后跟任何非字母（即类别码不是 11）的单个字符组成。在没必要区分控制词与控制符时，可以将它们统称为控制序列。

由转义符与一个空格字符 `\` 构成的控制序列，称为控制空格。

- 参数记号：由参数字符（类别码为 6，**Plain T<sub>E</sub>X** 中默认为 #）尾随一位在 1..9 中的数字构成。参数记号只能在宏的环境中出现（见第 11 章）。

在宏的替换文本中，如果一个宏参数字符之后又跟随了一个宏参数字符（字符码可以不相同），那么它们会被替换为单个字符记号，其类别码为 6（宏参数），字符码等于第 2 个参数字符的编码。常见情形是输入行内的 `##` 会被替换为 `#6`，这里的下标表示类别码。

## 2.5 输入处理器视为有限状态自动机

**T<sub>E</sub>X** 的输入处理器可视为三态的有限状态自动机，也就是说在任意的瞬间，它都处于这三种内部状态的某一种状态之中，并且在转移到另一种状态之后，对于前一状态没有任何记忆。

### 2.5.1 状态 *N*：新行

在每个输入行的开始处，**T<sub>E</sub>X** 输入处理器便会进入状态 *N*，这是它唯一可进入这一状态的时刻。在这一状态中，所有的空格记号（也就是类别码为 10 的字符）

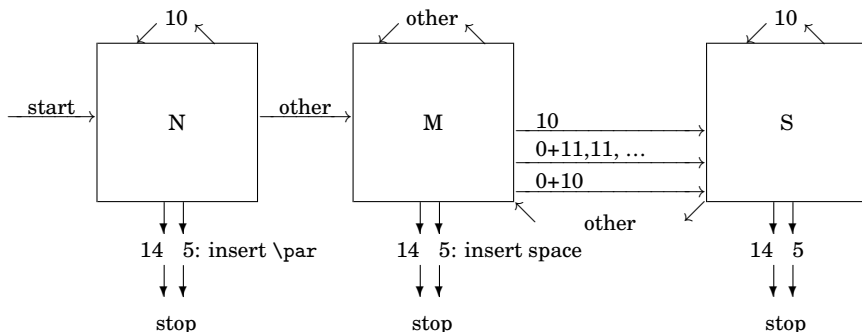
会被忽略；行结束符会被转化为 `\par` 记号。如果遇到其他记号，那么输入处理器所处状态便会切换为状态 *M*。

### 2.5.2 状态 *S*：忽略空格

在任何状态的控制词或控制空格（其他控制符不在这一范畴）之后，或者在状态 *M* 的空格字符之后，输入处理器便会进入状态 *S*。在这一状态中，所有的后续空格或行结束符会被丢弃。

### 2.5.3 状态 *M*：行内

显然状态 *M* 是最寻常的状态。当 **T<sub>E</sub>X** 的输入处理器遇到类别码为 1–4，6–8 以及 11–13 的字符或者控制符（不包括控制空格），在其之后便进入状态 *M*。在状态 *M* 中，如果输入处理器遇到了行结束符，它会将其转化为一个空格记号。



## 2.6 所有字符皆可信手拈来

严格地讲，**T<sub>E</sub>X** 输入处理器并非有限状态自动机。这是因为在扫描输入行期间，两个相同上标字符（类别码为 7）尾随一个编码小于 128 的字符（姑且称之为原字符）组成的三元组会被替换为字符码在 0–127 之间的字符，新字符的编码与原字符的编码相差 64。

这种字符访问机制主要用于访问那些难以输入的字符，例如像 `ascii` 码中的 `<return>` 和 `<delete>` 字符。可分别使用 `^^M` 和 `^^?` 进行访问。不过，由于 `^^?` 的类别码是 15，属于无效符，因此要访问编码为 127 的字符，必须先修改 `^^?` 的类别码。

**T<sub>E</sub>X3** 修改和扩展了这个机制以访问 256 个字符：任何四元组 `^^xy`，其中 `x` 和 `y` 为小写十六进制数字 0–9, `a–f`，被替换为一个在 0–255 之间的字符，即十六进制表示为 `xy` 的字符。这也稍微限制了前面机制的使用：设若键入了 `^^a` 以生成字符 `!`，接着再键入 0–9 或 `a–f` 将被错误理解。



这种字符访问机制使得  $\text{T}_{\text{E}}\text{X}$  的输入处理器比真正的有限状态自动机更强大，并且不会妨碍  $\text{T}_{\text{E}}\text{X}$  输入处理器的其余扫描过程。因而，为了更容易理解此概念，可以假装认为这种对  $\sim$  引导的三元组或四元组的字符替换是提前进行的。实际上这是不可能的，因为输入行内有可能会将非上标符的类别码也设为 7，这样便会影响后续的处理了。

## 2.7 内部状态切换

现在我们来关注一下不同类别码的字符对  $\text{T}_{\text{E}}\text{X}$  的输入处理器内部状态的影响。

### 2.7.1 0: 转义符

在遇到转义符时， $\text{T}_{\text{E}}\text{X}$  便开始形成一个控制序列记号。控制序列记号有三种类型，依赖于转义符后面的字符的类别码。

- 如果转义符之后的字符的类别码为 11，即字母，那么  $\text{T}_{\text{E}}\text{X}$  便会将转义符、类别码为 11 的字符以及后续所有类别码为 11 的字符捆绑为一个控制词，然后进入状态  $S$ ，即忽略空格状态。
- 如果转义符之后的字符的类别码为 10，即空格，那么  $\text{T}_{\text{E}}\text{X}$  便会产生一个控制空格，然后进入状态  $S$ 。
- 如果转义符之后的字符为其他类别码，那么  $\text{T}_{\text{E}}\text{X}$  便形成一个控制符，然后  $\text{T}_{\text{E}}\text{X}$  进入状态  $M$ ，即行内状态。

控制序列的名称所包含的字符必须居于同一行。即使当前行以注释符结束，或者当前行没有行结束符（通过将 `\endlinechar` 设定到 0–255 之外实现），控制序列字符也不能跨过两行。

### 2.7.2 1–4, 7–8, 11–13: 非空字符

类别码属于 1–4、7–8、11–13 的字符会被转化为记号，然后  $\text{T}_{\text{E}}\text{X}$  进入状态  $M$ 。

### 2.7.3 5: 行结束符

遇到行结束符时， $\text{T}_{\text{E}}\text{X}$  会忽略当前行的剩余部分，然后进入状态  $N$  开始处理下一行。如果当前状态是  $N$ ，即当前行只有空格， $\text{T}_{\text{E}}\text{X}$  就插入 `\par` 记号；如果当前状态是  $M$ ，那么就插入一个空格记号；如果当前状态是  $S$ ，就不插入任何记号。

注意“行结束符”是类别码为 5 的字符，它可以不是 `\endlinechar`，也不必出现在行尾。要明白它，请继续阅读下文。

### 2.7.4 6: 参数符

在宏定义中，参数符通常为 `#`，其后可跟随数字 `1..9` 或者另一个参数符，前者产生的是“参数记号”，后者产生的是单个字符记号。这两种情况都会导致 `TEX` 都会进入状态 *M*。

参数符在 Plain `TEX` 中也被用于构建阵列的模板行（见第 25 章）。

### 2.7.5 7: 上标符

上标符会像非空字符那样被处理，除非其后尾随一个相同字符码的上标符。两个上标符及其尾随字符构成的三元或四元组的字符替换功能在前文已有阐述。

### 2.7.6 9: 可忽略符

类别码为 9 的字符会被忽略，并且 `TEX` 会保持其状态不变。

### 2.7.7 10: 空格符

类别码为 10 的记号称为 `<space token>`（空格记号），不管其字符码是什么。在状态 *N* 和 *S* 中，`TEX` 会忽略空格记号（而且其状态不变）；在状态 *M* 中 `TEX` 会将它替换为类别码为 10 字符码为 32 的字符（`ascii` 空格符），并进入状态 *S*。这意味着空格记号的字符码可能与从输入字符的编码不同。

### 2.7.8 14: 注释符

注释符可使 `TEX` 忽略输入行的后续文本，其中包含注释符本身。特别地，注释符将导致 `TEX` 看不到输入行的行结束符，所以即使在状态 *M* 中遇到注释符，`TEX` 也不会插入空格记号。

### 2.7.9 15: 无效符

无效符会导致 `TEX` 报错。`TEX` 的状态会停留在无效字符之前的状态。不过，在控制符中的无效符是可以接受的，譬如 `\^^?` 就不会导致 `TEX` 报错。

## 2.8 字母符与其他字符

大部分编程语言的标识符可由字母与数字构成（也可能包含其他字符，例如下划线），但是 `TEX` 的控制词只能由类别码为 11 的字符形成。默认情况下，数字与标点符号的类别码为 12（其他字符）。不过 `TEX` 可以产生各字符的类别码均为 12 的字符串，尽管这些字符的原始类别码并非 12。

类别码为 12 的字符串可用 `\string`、`\number`、`\romannumeral`、`\jobname`、`\fontname`、`\meaning` 以及 `\the` 等命令生成。这些命令所产生的字符串中如果包含空格符，其类别码为 10。

在极个别情况下十六进制数字会隐藏在控制序列中，因而除了通常的  $A_{11}$ – $F_{11}$  之外，**T<sub>E</sub>X** 还允许  $A_{12}$ – $F_{12}$  作为十六进制数字（这里的下标表示类别码）。

看下面的示例：

`\string\end` 可以得到字符记号 `\12e12n12d12`

注意转义符 `\12` 出现在输出中是因为 `\escapechar` 的值等于反斜线的字符码。将 `\escapechar` 改为另一个值将使得 `\string` 输出另一个字符。这个 `\string` 命令将在第 3 章中进一步介绍。

空格是可以封到控制序列中的，例如

`\csname a b\endcsname`

给出的是一个控制序列记号，其中三个字符有一个是空格符。将这个控制序列转化为字符串

`\expandafter\string\csname a b\endcsname`

可得 `\12a12 10b12`。

举个更实用一些的例子，假设有一系列输入文件 `file1.tex`、`file2.tex` 等。我们想写一个宏统计输入文件的序号，一种方法是：

```
\newcount\filenumber \def\getfilenumber file#1.{\filenumber=#1 }
\expandafter\getfilenumber\jobname.
```

宏参数中的字符 `file`（见第 11.5 节）会吸走 `\jobname` 中的 `file` 部分，从而留下文件编号作为唯一的参数。

但是上述代码有误，宏参数中的 `file` 字符串的类别码为 11，而 `\jobname` 中的 `file` 字符串的类别码为 12，所以需要对上述代码进行以下修正：

```
{\escapechar=-1
 \expandafter\gdef\expandafter\getfilenumber
   \string\file#1.{\filenumber=#1 }
}
```

注意 `\string\file` 得到 `f12i12l12e12` 这 4 个字符，而 `\expandafter` 命令让 `\string\file` 在宏定义之前先行展开，并且 `\escapechar=-1` 让 **T<sub>E</sub>X** 忽略反斜线。由于 `\escapechar` 设定被限制在编组内部，我们需要使用 `\gdef` 进行宏定义。

## 2.9 \par 记号

**T<sub>E</sub>X** 在遇到空白行之后，即在状态 *N* 时遇到类别码为 5 的字符（行结束符）之后，就会向输入中插入一个 `\par` 记号。最好是明白这是如何发生的：因为 **T<sub>E</sub>X** 在遇到空格符之外的任何字符都会离开状态 *N*，所以能够形成 `\par` 的输入行所包含字符的类别码肯定皆为 10；特别地，该行不能包含注释符。此事实常常以另

一种方式被用到：如果输入格式中需要保留空白行，我们可以给该行加上一个注释符。

连续两个空行产生两个 `\par` 记号，实际上它们等同于一个 `\par` 记号，这是因为在第一个 `\par` 之后，**T<sub>E</sub>X** 进入竖直模式，而在竖直模式中的 `\par` 只会触发 **T<sub>E</sub>X** 的页面构建器以及清除段落形状参数。

在非受限水平模式中遇到 `<vertical command>`（竖直命令）时，**T<sub>E</sub>X** 也会向输入中插入一个 `\par` 记号，并对其读取和展开，然后再重新处理竖直命令（见第 6 和 17 章）。

`\end` 命令也会插入 `\par` 记号，然后结束 **T<sub>E</sub>X** 的运行；见第 28 章。

要知道 **T<sub>E</sub>X** 在遇到空白行时通常所做的事情（结束当前段落）取决于 `\par` 记号的默认定义。如果重定义 `\par`，那么空白行和竖直命令的行为可能就完全不同了，甚至可以藉此实现一些不同寻常的效果。为了能够得到与 `\par` 相同的行为，Plain **T<sub>E</sub>X** 提供了 `\par` 的“同义词”`\endgraf`。详见第 17 章。

`\par` 记号不可以出现在宏参量中，除非是使用 `\long` 定义的宏。对于非 `\long` 定义的宏，如果 `\par` 出现在参量中，**T<sub>E</sub>X** 会给出“runaway argument”的错误信息。不过，使用 `\let` 定义的指向 `\par` 记号的控制序列（例如 `\endgraf`）则可以出现。

## 2.10 空格

这一节讨论空格字符的一些表现，以及 **T<sub>E</sub>X** 初始化进程中的空格记号。至于文本排版的空格，将在第 20 章中讨论。

### 2.10.1 被忽略的空格

从对 **T<sub>E</sub>X** 输入处理器的内部状态的讨论中，容易知道有些空格是不可能被输出的；实际上它们甚至都无法通过输入处理器。例如输入行开头的空格，还有放在让 **T<sub>E</sub>X** 进入状态 *S* 的字符之后的空格。

另一方面，行结束符可以产生空格，并且可被输出。还有第三种空格：它可以通过输入处理器，甚至可在输入处理器中生成，但是依然没有机会被输出，它们便是 `<optional spaces>`（可选空格），**T<sub>E</sub>X** 语法的多个地方都允许出现这种空格。

### 2.10.2 可选空格

**T<sub>E</sub>X** 的语法中有“可选空格”与“单个可选空格”的概念：

`<one optional space>`  $\longrightarrow$  `<space token>` | `<empty>`

`<optional spaces>`  $\longrightarrow$  `<empty>` | `<space token>``<optional spaces>`

通常，`<one optional space>` 允许出现在数值以及粘连描述之后，而 `<optional spaces>` 允许出现在数值内部（比如在负号和数字之间）或者粘连描述内部（比

如在 `plus` 和 `ifil` 之间) 可以有空格的地方。另外, 在 `<equals>` 的定义中也允许 `<optional spaces>` 出现在 `=` 号前后。

下面是可选空格的一些例子:

- 数值可被 `<one optional space>` 分割。这样防止了偶然的失误 (见第 7 章) 并加速了处理过程, 因为  $\text{\TeX}$  检测 `<number>` 在何处终止更容易。不过, 要注意并非每个“数值”都是 `<number>`: 例如 `\magstep2` 中的 `2` 并非数字, 而是单个记号并且是 `\magstep` 宏的参量, 因此在其之后的空格或行结束符是有效的。另一个例子是宏参数中的数字, 例如 `#1`: 因为一个宏最多允许有 9 个参数, 只需在参数字符之后扫描一位数字即可。
- 根据  $\text{\TeX}$  的语法, 关键字 `fill` 和 `filll` 是由 `fil` 关键字以及一两个单独的 `l` 关键字构成的 (见第 303 页的更详细讨论), 因此其中允许可选空格的存在: 比如 `fil l l` 也是有效的关键字。不过这也许会导致  $\text{\TeX}$  误解你的本意, 对于大多数情形, 在这种关键字后添加一个 `\relax` 可以防止这种灾难。
- 在宏定义末尾使用原始命令 `\ignorespaces` 可能会比较方便。由于它可以吞噬可选空格, 使用它可避免把参量的右花括号后的空格无意中带入输出中。例如下面这个例子:

```
\def\item#1{\par\leavevmode
  \llap{#1\enspace}\ignorespaces}
\item{a/}one line \item{b/} another line \item{c/}
yet another
```

其中 `\ignorespaces` 吞掉了第二、三项的那些不希望被输出的空格。不过 `\ignorespaces` 之后的空行仍然会插入 `\par` 记号。

### 2.10.3 被忽略的和被保留的空格

控制词之后的空格会被忽略。不过这个不是可选空格的例子, 只是因为  $\text{\TeX}$  在控制词之后会进入状态 *S* 而已。同样, 控制词之后的行结束符也会被忽略。

数值只能被 `<one optional space>` 定界, 但是

```
a\count0=3 b 仍然给出 'ab',
```

这是因为  $\text{\TeX}$  在第一个空格记号之后会进入状态 *S*, 因此第二个空格永远也不可能变成空格记号。

当  $\text{\TeX}$  在状态 *N* 中时, 空格会被忽略。当  $\text{\TeX}$  在竖直模式中时, 空格记号 (就是那些起初未被忽略的空格) 会被忽略。例如下面第一个盒子之后 (由行结束符生成的) 的空格会被忽略:

```
\par
\hbox{a}
\hbox{b}
```

Plain  $\text{\TeX}$  和  $\text{\LaTeX}$  都定义了一个 `\obeyspaces` 宏, 这使得空格都是有效的,

比如控制词后的空格以及空格后的空格都不会被忽略。这个宏的基本实现方式为<sup>1</sup>

```
\def\space{ }
\catcode`\ =13 \def {\space}
```

在实现多行抄录环境时，还需要另一个 `\obeylines` 宏：它将每个行结束符定义为 `\par` 命令，使得下面各行都在竖直模式中开始。此时活动空格展开的空格记号在竖直模式中将会被忽略，即空白行将会被删除。为此我们可以修改上述 `\space` 宏的定义如下：

```
\def\space{\leavevmode{ } }
```

这样，活动空格将会让 **T<sub>E</sub>X** 立即切换到水平模式，从而保留了每个空格。

#### 2.10.4 空格被忽略的其他情形

还有三种情况会导致 **T<sub>E</sub>X** 忽略空格记号：

1. 在寻找非定界的宏参量时，**T<sub>E</sub>X** 会接受第一个非空格的记号（或编组）作为参量。这将在第 11 章中介绍。
2. 在数学模式中，空格记号会被忽略（见第 23 章）。
3. 在阵列制表符之后，空格记号会被忽略（见第 25 章）。

#### 2.10.5 `\space token`

空格在 **T<sub>E</sub>X** 中有些反常。例如，`\string` 操作会对所有的字符赋以类别码 12，唯独对空格例外，它还是坚持自己的类别码为 10。还有在前文中提到过的，**T<sub>E</sub>X** 的输入处理器（在状态 *M* 中）会将所有类别码为 10 的记号转化为真正的空格：它们的字符编码为 32。任何类别码为 10 的空格称为 `\space token`。那些字符编码不是 32 的空格记号被称为滑稽空格。

例子：将空格字符的类别码赋予字符 Q，并在宏定义中使用它：

```
\catcode`Q=10 \def\q{aQb}
```

那么，我们可得到

```
\show\q
macro:-> a b
```

这是因为输入处理器改变了宏定义中滑稽空格的字符编码。

字符码不为 32 的空格记号可以用 `\uppercase` 等命令生成。然而，‘由于各种不同的空格记号的表现几乎是一致的，纠缠于细节毫无意义’；见 [25] 第 377 页。

<sup>1</sup>译注：原文最后两个段落的描述有些错乱，已经稍作修订。

### 2.10.6 控制空格

‘控制空格’命令 `\_` 给出的空白的大小与 `\spacefactor` 等于 1000 时 `\space token` 给出的一样。控制空格不能当成空格记号来用，也不能像宏一样展开成为空格记号（像 **Plain TeX** 定义的 `\space` 那样）。例如，**TeX** 会忽略输入行开头的空格，但是控制空格是一个 `\horizontal command`，因此它使得 **TeX** 从竖直模式切换到水平模式（并插入缩进盒子）。见第 20 章介绍的空白因子，以及第 6 章介绍的水平和竖直模式。

### 2.10.7 可见空格

显式的空格符号 ‘`␣`’ 是计算机现代打字机字体中字符编码为 32 的字符，但仅使用 `\tt` 是无法将其显现出来的，因为空格在输入处理器中受到了特别处理。

使空格字符 `␣` 现形的一种方法是设置

```
\catcode\ =12
```

这样 **TeX** 便会将空格字符作为编码为 32 的字符排印出来，而且后续的空格也不再被忽略，同样会被排印出来：状态 *S* 只是在类别码为 10 的字符之后才会出现。类似地，控制序列之后的空格也因为类别码改变而被显现出来。

## 2.11 行结束符的更多知识

**TeX** 从输入文件中获得文本行，并从中消除行终止符。正是这一行为，使得 **TeX** 不依赖于各个操作系统特定的行终止符（CR-LF，LF，或者在块存储系统中根本不存在）。文本行末尾的空白字符也会被移除。这样处理是由于历史原因：它与 IBM 大型计算机的块存储模式有关。在 [2] 中介绍了行尾符造成的一些计算机问题。

完成上述处理后，**TeX** 会将 `\endlinechar` 所表示的字符置于行尾，除非 `\endlinechar` 的字符码为负数或者大于 255。注意这个行结束符也可以不是类别码为 5 的字符。

### 2.11.1 保持各行

有时候会期望会希望输入文本中的行结束符能够在排版输出后保持。下面的代码可以解决这一问题：

```
\catcode\^^M=13 %
\def^^M{\par}%
```

这里，`\endlinechar` 成为活动符，其含义变为 `\par`。上述代码中的注释符用于阻止 **TeX** 看到代码末尾的行终止符，以防它将其作为活动字符而展开。

然而，将上述代码嵌入宏定义时要小心，比如

```
\def\obeylines{\catcode\^^M=13 \def^^M{\par}}
```



是会被  $\text{\TeX}$  误解的： $\text{\TeX}$  将丢弃第二个  $\text{\~M}$  之后的所有字符，因为此时  $\text{\~M}$  类别码为 5，而非 13。也就是说，这一行实际上变成

```
\def\obeylines{\catcode`\~M=13 \def
```

要修正上述问题，需要为  $\text{\~M}$  营造一个可作为活动字符使用的环境：

```
{\catcode`\~M=13 %
  \gdef\obeylines{\catcode`\~M=13 \def~M{\par}}}%
}
```

不过这个定义还是有缺陷，因为输入文本中的空行会被忽略。这是因为连续两个  $\text{\par}$  记号会被当成一个。对上述定义稍作改进即可解决这个问题，如下：

```
\def~M{\par\leavevmode}
```

这样，输入文本中的每一行都会开启一个新段落，空行则开启一个空段落。

### 2.11.2 改变 $\text{\endlinechar}$

有时，你可能想改变  $\text{\endlinechar}$  或者  $\text{\~M}$  的类别码以获得一些特殊效果，例如让宏的参量用行结束符定界。参考第 127 页给出的例子。

这里有几个陷阱。首先考虑下面的写法：

```
{\catcode`\~M=12 \endlinechar=~J \catcode`\~J=5
...
... }
```

这将导致无意中输出了第 13 号 ( $\text{\~M}$ ) 与第 10 号 ( $\text{\~J}$ ) 字符，由于第一行和最后一行的行终止符。

在第一行和最后一行末尾加上注释符可以解决此问题，但还有另一种方法是将第一行拆成下面两行

```
{\endlinechar=~J \catcode`\~J=5
\catcode`\~M=12
```

当然，在多数情况下没必要将行结束符替换为另一个字符；设置

```
\endlinechar=-1
```

就等同于各行都以注释符结尾。

### 2.11.3 行结束符的更多注记

$\text{\TeX}$  和其他字符一样对待添加到行尾的字符。通常我们不会注意到它，因为它的类别码比较特殊，但是有一些方法可以特殊地处理它。

例子：把  $\text{\~}$  置于文本行的末尾（假定  $\text{\endlinechar}$  保持默认值为 13），将输出字符  $\text{\textit{M}}$ ，它是编码为  $13+64$  的 `ascii` 字符。

例子：如果已经定义了  $\text{\~M}$ ，在输入行中用反斜线结尾将执行此命令。在 *Plain* 格式中定义

```
\def~M{\ }
```



这使得‘控制换行’与控制空格等价。

## 2.12 输入处理器的更多知识

### 2.12.1 输入处理器作为独立进程

**T<sub>E</sub>X** 处理器的各个层面都是同时运行的，但是在概念上它们常被视为依次独立运行，前者的输出是后者的输入。空格的花招可以展示出这一规律。

例如定义一个宏：

```
\def\DoAssign{\count42=800}
```

然后调用它：

```
\DoAssign 0
```

输入处理器作为 **T<sub>E</sub>X** 构建记号列表的层面会忽略 0 之前的所有空格，因此上述宏的展开的结果是：

```
\count42=8000
```

不要认为 `\DoAssign` 被读取然后展开，接着寄存器被赋值为 800，因此 `\DoAssign` 之后的那个 0 会被排印出来。注意即使最后的 0 出现在下一行结果也一样。

再来看下面这个让可选空格字符在多个处理层面中出现的例子：

```
\def\c.{\relax}
a\c. b
```

它的展开结果为

```
a\relax b
```

输出结果为

```
‘a b’
```

这是因为 `\relax` 之后的空格仅仅在文本行被读取时可能会被忽略，在 `\relax` 之后不会被忽略。另一方面，下面例子：

```
\def\c.{\ignorespaces}
a\c. b
```

会被展开为

```
a\ignorespaces b
```

在执行处理器中 `\ignorespaces` 会移除它后面的空格，所以输出结果会是

```
‘ab’.
```

在上述两个例子中，`\c` 之后的西文句号是一个定界记号，用于保护控制序列之后的空格不被输入处理器吃掉。

### 2.12.2 输入处理器不作为单独进程

将  $\text{\TeX}$  对输入文本的记号化过程视为一个独立进程是比较普遍的看法，但是有时会出现反常的现象。例如

```
\catcode\^^M=13{}
```

使得行结束符变成活动符，随后  $\text{\TeX}$  便会报错“未定义的控制序列”，即对文本行中的命令的执行影响到  $\text{\TeX}$  输入处理器对该行文本的扫描过程。

与此相反，

```
\catcode\^^M=13
```

却不会出错。这是因为  $\text{\TeX}$  输入处理器是在扫描数值 13 时读到行结束符，也就是说在那时赋值还未完成，因此行结束符会被视为数值的定界符，即可选空格。

### 2.12.3 输入处理器的递归调用

前文中谈到，参数符加数字会被替换为一个参数记号，这种替换行为类似于将一些字符捆绑为控制序列记号的行为。实际上情况比这复杂得多。从文件输入和从记号列（比如宏定义）输入都会调用  $\text{\TeX}$  的记号扫描机制，但内部状态的变化只适用于前者。

但是，宏参数符在两种情况下会被以相同方式处理，否则  $\text{\TeX}$  便无法处理下面这样的宏定义

```
\def\ a{\def\ b{\def\ c####1{####1}}}
```

见第 120 页对这种嵌套定义的解释。

## 2.13 @ 约定

如果读过 Plain 或  $\text{\LaTeX}$  格式的源代码，就会注意到许多控制序列都包含 ‘at’ 符号 @，这意味着这些控制序列不可被普通用户直接使用。

在靠近格式文件的起始处有

```
\catcode\@=11
```

它将 @ 变为字母字符，从而可以用于组成控制序列。而在靠近格式文件的结尾处有

```
\catcode\@=12
```

它将 @ 恢复为其他字符。

为何我们可以调用那些由带有 @ 字符的控制序列所构成的宏，而不能直接调用带有 @ 字符的控制序列呢？原因是带有 @ 字符的控制序列在定义时已经被转换为记号，不再是字符串，而宏展开时直接将这此控制序列替换为那些记号即可，这个过程与控制序列字符的类别码无关。

## 第 3 章 字符

**T<sub>E</sub>X** 在其内部使用字符编码来表示字符。这一章讨论字符编码及相关命令。

`\char` 显式表示所要排印的字符。

`\chardef` 定义一个控制序列用以表示一个字符编码。

`\accent` 放置重音符号的命令。

`\if` 测试字符编码是否相等。

`\ifx` 测试字符编码与类别码是否都相等。

`\let` 定义一个控制序列，使之成为一个记号的别名。

`\uccode` 对于给定的字符编码，查询或设置其对应的大写字符编码。

`\lccode` 对于给定的字符编码，查询或设置其对应的小写字符编码。

`\uppercase` 将 `<general text>` 转换为大写形式。

`\lowercase` 将 `<general text>` 转换为小写形式。

`\string` 将一个记号转换为一个字符串。

`\escapechar` 在将控制序列转换为字符记号列时，用于转义符的字符编码。

**IniT<sub>E</sub>X** 默认为 92 (`\`)。

### 3.1 字符编码

表面上看，**T<sub>E</sub>X** 内部处理的是字符，但实际上 **T<sub>E</sub>X** 处理的是整型数：字符编码。

在计算机中，字符编码在各个系统中可能有差别。因而 **T<sub>E</sub>X** 不得不使用它自己的字符编码方式。从文件中读取的任何字符都会根据 **T<sub>E</sub>X** 的字符码表转换为字符编码，并赋以相应的类别码（见第 2 章）。**T<sub>E</sub>X** 的字符码表是基于 7 位的 `ascii` 码表构建的，只有 128 个字符编码（见第 38.1 节）。

利用左引号（或称为反引号）字符 ```，可以将字符记号显式地转换为对应的字符编码：在 **T<sub>E</sub>X** 要求 `<number>` 的所有地方，你都可以用左引号加一个字符记

号或一个单字符控制序列。因此 `\count`a` 和 `\count`\a` 都表示 `\count97`。另见第 7 章。

虽然上述两种写法是等价的，但有时必须使用后者的形式，例如：

```
\catcode`\%=11 或 \def\CommentSign{\char`\%}
```

此时如果去掉 `\`，就会让  $\TeX$  误解。比如

```
\catcode%=11
```

中的 `=11` 将被当成注释。单字符控制序列可以由任意类别码的字符构成。

在转换为字符编码后，字符与其外部表示的联系都已经消失了。当然，对于大多数字符，见到的输出将‘等同’于输入（即‘a’将输出‘a’）。然而即使对于常见符号也还是有例外。在计算机现代罗马字体中，没有‘小于号’和‘大于号’，而输入‘<>’得到的输出是‘*l*’。

为了使  $\TeX$  输出不依赖于系统环境，在 `dvi` 文件中也是使用字符编码：操作码  $n = 0 \dots 127$  用于表示指令“从当前字体中取第  $n$  个字符”。在 [23] 中可以找到 `dvi` 文件的操作码的完整定义。

## 3.2 用于字符的控制序列

用控制序列表示字符的方式有多种。`\char` 命令可以使用字符编码形式指定要排印的字符；`\let` 命令可以使用一个控制序列作为字符记号的别名。

### 3.2.1 表示要排印的字符：\char

字符可以用数值来表示，比如 `\char98`。这个命令告诉  $\TeX$  将当前字体中编码为 98 的字符添加到当前正在构建的水平列中。

但是通常不使用十进制而使用八进制或十六进制来表示字符编码。八进制数前面用单引号引导，比如 `\char'142`；十六进制数前面用双引号引导，比如 `\char"62`。注意 `\char'62` 是错误的；因为数值扫描操作（由执行处理器处理）发生在将两个单引号替换为一个双引号的操作（由可视化处理器处理）之前。

由于使用反引号 ``` 可以将字符转换为字符编码，所以用 `\char`b` 或 `\char` 也可以得到‘b’这个字符；前提是当前字体的字符编码是符合 `ascii` 码表。

表面上看，`\char` 有点像 `^^`（第 2 章），因为这两种机制都是采用间接的方式来表示字符。但是，`^^` 机制的工作时机要早于 `\char`，前者是在输入处理器的第一阶段运作，而后者则是在  $\TeX$  可视化处理阶段运作。

利用 `\chardef` 命令可以定义一个控制序列来代替某个字符编码。这个命令的语法如下：

```
\chardef⟨control sequence⟩⟨equals⟩⟨number⟩,
```

其中的数值可以显式给出或者用计数器值表示，还可以用反引号命令给出的字符码表示（如上所述；`⟨number⟩` 的定义在第 7 章给出）。在 Plain  $\TeX$  中，类似下面的

定义就使用后面这种写法：

```
\chardef\%=`%
```

当然，它可以用下面的等价方式定义：

```
\chardef\%=37
```

在此定义之后，控制符 `\%` 就可以作为 `\char37` 同义词使用，也就是说，这个命令将排版第 37 个字符（通常是英文百分号）。

用 `\chardef` 命令定义的控制序列也可以作为 `<number>` 使用。此事实类似 `\newbox` 的寄存器分配命令中用到（见第 7 和 31 章）。用 `\mathchardef` 定义的记号同样可以这样使用。

### 3.2.2 隐式字符记号：\let

再一种利用控制序列来表示字符的方式是使用 `\let` 命令：

```
\let<control sequence><equals><token>
```

如果可选等号的右边是一个字符记号，得到的控制序列称为隐式字符记号（见第 122 页对 `\let` 的进一步讨论。）

比如在 Plain TeX 中用下面方式定义左右花括号的同义词：

```
\let\bggroup={ \let\egroup=}
```

这样得到的控制序列称为‘隐式花括号’（见第 10 章）。

但是 `\let` 与 `\chardef` 是有区别的，因为 `\let` 是用控制序列来表示字符码与类别码的组合结构。

例如

```
\catcode`=2 % make the bar an end of group
\let\b=| % make \b a bar character
{\def\m{...}\b \m
```

会得到“未定义的控制序列 `\m`”的错误，这是因为 `\b` 关闭了 `\m` 定义所在的编组。另一方面，

```
\let\b=| % make \b a bar character
\catcode`=2 % make the bar character end of group
{\def\m{...}\b \m
```

构造的只是一个不闭合的编组，因为这次 `\b` 无法作为组结束符来用，它只能表示一个竖线（或者是当前字体位于第 124 个位置的字符）。

本小节的第一个例子实际上说明，即使花括号已经被重新定义（比如在排版 C 语言代码的宏中将它们定义为活动符），编组开始和编组结束的功能还是可以通过 `\bggroup` 和 `\egroup` 使用。

这里还有另一个例子，说明隐式字符记号与真实字符记号难以区分。在下面的控制序列之后：

```
\catcode`=2 \let\b=|
```

这个测试

```
\if\b|
```

和这个测试

```
\ifcat\b}
```

都给出真值。

在 Plain  $\text{\TeX}$  中还有下面的定义：

```
\let\sp=^ \let\sb=_
```

它使得键盘上没有扬抑符和底线符的用户也能够数学公式中写出上下标。比如

```
x\sp2\sb{ij}
```

给出  $x_{ij}^2$

如果编写格式文件的人也无法键入这两个字符，则需要更多花招。比如下面的代码就可以完成此任务：

```
{\lccode`,=94 \lccode`. =95 \catcode`,=7 \catcode`. =8
\lowercase{\global\let\sp=, \global\let\sb=.}}
```

详见下面对 `\lowercase` 命令的解释。此时无法使用  $\text{\TeX}$  2 中的 `^^` 表示法（见第 30 页），因为这需要键入两个无法输入的字符。用  $\text{\TeX}$  3 中扩展的表示法，下面的方法也是可行的：

```
{\catcode`,=7
\global\let\sp=,,5e \global\let\sb=,,5f}
```

其中用十六进制表示 `^` 和 `_` 的字符码 94 和 95。

要查看 `\let` 定义的控制序列代表的是哪个字符，可使用 `\meaning`，例如：

```
\let\x=3 \meaning\x
```

可给出 ‘the character 3’。

### 3.3 重音

重音可以用 (horizontal command) `\accent` 给出：

```
\accent<8-bit number><optional assignments><character>
```

其中 `<character>` 或者是第 11 类或 12 类的字符，或者是 `\char<8-bit number>` 命令，或者是 `\chardef` 记号。如果 `<character>` 不是这四种类型，`\accent` 命令就被视为 `\char` 命令；从而给出‘悬在半空中’的重音。否则重音被放置在后面跟随的字符的顶部。`<optional assignments>` 可用于在重音和字符之间改变字体。

`\accent` 命令后面跟随的必须是一个 `<character>`，这意味着一个令人不悦的事实，即不可能将重音放置在连写上，或者将重音放在另一个重音上。在某些语言中，比如印地语或越南语，这种连写重音确实存在。将重音放在另一个重音是可能的，但只能用于数学模式。

字符添加重音后宽度保持不变。对于字体中高度等于 `x-height` 的字符， $\text{\TeX}$  假定按照字体文件中描述的位置可以正确放置重音；对于其他字符，它相应地升高或降低重音的位置。

在  $\text{\TeX}$  中没有真正的下重音。它们都是作为位置很低的上重音实现的。更好的解决方法是写一个宏测量后面跟随的字符，并相应地升高或降低重音。Plain  $\text{\TeX}$  中的变音符  $\backslash c$  是用这种方式实现的。然而，对于包含降部的字符，它并不会降低重音的位置。

重音的水平位置由  $\backslash fontdimen1$  (*slant per point*) 控制，其水平位移用紧排表示。注意，虽然这些紧排是被自动插入的，它们被划分为显式紧排。所以，它们会抑制在紧排前后连字。

作为重音对应的紧排的例子，下面显示了一个水平列表：

```
 $\backslash setbox0=\hbox{\it \`l}$ 
 $\backslash showbox0$ 
```

给出

```
 $\backslash hbox(9.58334+0.0)x2.55554$ 
 $\backslash kern -0.61803$  (for accent)
 $\backslash hbox(6.94444+0.0)x5.11108$ , shifted -2.6389
 $\backslash tenit \text{\`R}$ 
 $\backslash kern -4.49306$  (for accent)
 $\backslash tenit l$ 
```

注意  $\text{\TeX}$  先放置重音，这样最后一个字符的倾斜校正仍然有效。

## 3.4 字符测试

要测试字符编码是否相等，可使用  $\backslash if$  命令：

```
 $\backslash if\langle token_1 \rangle \langle token_2 \rangle$ 
```

注意  $\backslash if$  后面的记号会被展开，直到得到两个不可展开的记号，然后  $\backslash if$  如果它们是两个字符码相同的字符记号（不考虑它们的类别码），则结果为真。

$\text{\TeX}$  将不可展开的控制序列的字符码和类别码分别视为 256 和 16。（因此它只能与另一个控制序列相等），除非用  $\backslash let$  让它等同于一个非活动符。在那种情况下，该控制序列的字符码和类别码就与该字符的相同。这在之前提到过。

用于测试类别码是否相等的  $\backslash ifcat$  在第 2 章已经介绍。而要同时测试字符的字符码和类别码是否相等，可使用  $\backslash ifx$  命令：

```
 $\backslash ifx\langle token_1 \rangle \langle token_2 \rangle$ 
```

在执行此测试时  $\backslash ifx$  后面的记号不会被展开。然若它们均为宏， $\text{\TeX}$  会比较它们的展开是否相等。

使用  $\backslash chardef$  定义的量值可使用  $\backslash ifnum$  来测试：

```
 $\backslash chardef\text{\`a}=\text{\`x}$   $\backslash chardef\text{\`b}=\text{\`y}$   $\backslash ifnum\text{\`a}=\text{\`b} \% \text{ is false}$ 
```

此测试用到  $\langle chardef token \rangle$  可以作为数值使用的事实（见第 7 章）。

另外可以参考第 13.2 节。

## 3.5 大写和小写

### 3.5.1 大写码和小写码

每个字符码都对应一个大写码和一个小写码（下面列出了更多编码）。它们可以分别用

```
\uccode⟨number⟩⟨equals⟩⟨number⟩
```

以及

```
\lccode⟨number⟩⟨equals⟩⟨number⟩.
```

指定。在 **IniT<sub>E</sub>X** 中 ``a...`z` 和 ``A...`Z` 的大写码为 ``A...`Z`，小写码为 ``a...`z`。其他字符码的大写码和小写码均为零。

### 3.5.2 大写和小写命令

命令 `\uppercase{...}` 和 `\lowercase{...}` 遍历它们的参量记号列，将所有显式字符记号的字符码分别替换为它们的大写码和小写码，只要这两个编码非零，但不改动字符记号的类别码。

`\uppercase` 和 `\lowercase` 的参量是一个 `⟨general text⟩`，它的定义如下：

```
⟨general text⟩ → ⟨filler⟩{⟨balanced text⟩⟨right brace⟩}
```

其中 `⟨filler⟩` 的定义可以见第 36 章。这个定义意味着左花括号可以是隐式的，但右花括号必须是类别码为 2 的显式字符记号。**T<sub>E</sub>X** 展开表达式以找到左花括号。

大小写转换在执行处理器中执行；它们并非像 `\number` 或 `\string` 这样的‘宏展开’活动。下面的语句（试图生成 `\A`）

```
\expandafter\csname\uppercase{a}\endcsname
```

将给出一个错误（由于 `\uppercase` 是不可展开的，**T<sub>E</sub>X** 会在 `\uppercase` 前插入 `\endcsname`），然而

```
\uppercase{\csname a\endcsname}
```

就是正确的。

下面的例子正确地使用了 `\uppercase`，它是一个测试字符是否为大写的宏：

```
\def\ifIsUppercase#1{\uppercase{\if#1}#1}
```

用 `\ifnum`#1=\uccode`#1` 也可以执行相同的测试。

大写字符就是与其 `\lccode` 不同的字符。首字母大写的单词是否可以连字化取决于 `\uchyph` 参数：如果它大于零，就允许对这种单词连字化。这将在第 19 章中介绍。



### 3.5.3 关键词的大小写形式

对于  $\TeX$  关键词，比如 `pt`，其中每个字符都可以用大写或小写表示。例如 `pT`、`Pt`、`pt` 和 `PT` 都表示同一个关键词。这里  $\TeX$  并没有用 `\uccode` 和 `\lccode` 表确定小写形式，而是直接将大写字符加上 32（即两者编码之差）以转换为小写字符。这影响到在非罗马字母表上的  $\TeX$  实现；见 the  $\TeX$  book [25] 第 370 页。

### 3.5.4 妙用 `\uppercase` 和 `\lowercase`

前面已经说到 `\uppercase` 和 `\lowercase` 不改变类别码，这个事实有时可以用于制造用其他方式很难得到的（字符码，类别码）组合。比如可以参考第 13 章的 `\newif` 宏，以及另一个在第 44 页的例子。

这里给出一个稍微不同的应用。考虑 Rainer Schöpf 解决的问题：如何将寄存器 `\newcount\mycount` 的编号 `\mycount` 输出到终端？下面是一种解法：

```
\lccode`a=\mycount \chardef\terminal=16
\lowercase{\write\terminal{a}}
```

其中 `\lowercase` 命令成功地将 `\write` 命令的参量中的 ‘a’ 修改成所需要的编号。

## 3.6 字符相关编码

每个字符都带有一系列 `<codename>`。这些整数的取值范围各不相同，它们决定在各个地方如何处理该字符，或者在某些地方  $\TeX$  的活动方式如何被该字符改变。

这些编码名称如下所列：

`\catcode` `<4-bit number>` (0–15)；字符所属的类别。在第 2 章中介绍。

`\mathcode` `<15-bit number>` (0–"7FFF 或 "8000；确定在数学模式中如何处理该字符。见第 21 章。

`\delcode` `<27-bit number>` (0–"7 FFF FFF)；确定在数学模式的 `\left` 或 `\right` 命令后如何处理该字符。见第 207 页。

`\sfcode` 整数；确定该字符如何影响其后的空白。见第 20 章。

`\lccode`, `\uccode` `<8-bit number>` (0-255)；小写码和大写码。上面刚介绍过。

## 3.7 将记号转换为字符串

`\string` 命令可将其后尾随的记号转为字串。例如：

```
\tt\string\control
```

的排版结果为 `\control`。再例如：

```
\tt\string$
```

的排版结果为 \$。要注意的是，\string 是在输入处理器产生记号这个过程之后运作的，所以

```
\tt\string%
```

就没法排出 %，因为注释符在  $\TeX$  输入处理器中会被移除。因此，这个命令将会给出下一行第一个记号转换为字符串。

\string 命令是在展开处理器中运行的，因此除非显式抑制才能阻止它被展开；这是第 12 章的话题。

### 3.7.1 输出控制序列

前面例子选用了打字机字体，这是因计算机现代罗马字体不包含反斜线字符。然而， $\TeX$  显示控制序列时未必得用反斜线：它使用 \escapechar 字符码。在用 \write、\message、\errmessage、\show、\showthe 或 \meaning 输出控制序列时，也使用这个编码。如果 \escapechar 大于零或者大于 255，转义符将不会被显式；在  $\text{Ini}\TeX$  中它的默认值为 92，也就是反斜线字符的编码。

在 \write 语句中的 \string 有时可以用 \noexpand 代替，见第 141 页。

### 3.7.2 \string 的类别码

在 \string 命令给出的字符串中，各个字符的类别码都是 12，但空格字符除外，它们的类别码是 10。由于在控制序列中没有类别码，从 \string 得到的空格必定只有空格字符，即编码为 32 的字符。然而， $\TeX$  的输入处理器将任何字符码不为 32 的空格记号转换为字符码为 32 的字符记号，因此‘滑稽空格’有可能出现在控制序列中。

在所给出的类别码这方面，这些命令的表现与 \string 是一致的：\number、\romannumeral、\jobname、\fontname、\meaning 和 \the。

## 第 4 章 Fonts

In text mode  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  takes characters from a ‘current font’. This chapter describes how *fonts* are identified to  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ , and what attributes a font can have.

`\font` Declare the identifying control sequence of a font.

`\fontname` The external name of a font.

`\nullfont` Name of an empty font that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  uses in emergencies.

`\hyphenchar` Number of the hyphen character of a font.

`\defaultshyphenchar` Value of `\hyphenchar` when a font is loaded. Plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  default: ``\-`.

`\fontdimen` Access various parameters of fonts.

`\char47` Italic correction.

`\noboundary` Omit implicit boundary character.

### 4.1 Fonts

In  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  terminology a font is the set of characters that is contained in one external font file. During processing,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  decides from what font a character should be taken. This decision is taken separately for text mode and math mode.

When  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is processing ordinary text, characters are taken from the ‘current font’. External font file names are coupled to control sequences by statements such as

```
\font\MyFont=myfont10
```

which makes  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  load the file `myfont10.tfm`. Switching the current font to the font described in that file is then done by

```
\MyFont
```

The status of the current font can be queried: the sequence

`\the\font`

produces the control sequence for the current font.

Math mode completely ignores the current font. Instead it looks at the ‘current family’, which can contain three fonts: one for text style, one for script style, and one for scriptscript style. This is treated in Chapter 21.

See [42] for a consistent terminology of fonts and typefaces.

With ‘virtual fonts’ (see [24]) it is possible that what looks like one font to  $\TeX$  resides in more than one physical font file. See further page 288.

## 4.2 Font declaration

Somewhere during a run of  $\TeX$  or  $\text{Ini}\TeX$  the coupling between an internal identifying control sequence and the external file name of a font has to be made. The syntax of the command for this is

`\font<control sequence><equals><file name><at clause>`

where

`<at clause>  $\longrightarrow$  at <dimen> | scaled <number> | <optional spaces>`

Font declarations are local to a group.

By the `<at clause>` the user specifies that some magnified version of the font is wanted. The `<at clause>` comes in two forms: if the font is given scaled  $f$   $\TeX$  multiplies all its font dimensions for that font by  $f/1000$ ; if the font has a design size  $d_{\text{pt}}$  and the `<at clause>` is at  $p_{\text{pt}}$   $\TeX$  multiplies all font data by  $p/d$ . The presence of an `<at clause>` makes no difference for the external font file (the `.tfm` file) that  $\TeX$  reads for the font; it just multiplies the font dimensions by a constant.

After such a font declaration, using the defined control sequence will set the current font to the font of the control sequence.

### 4.2.1 Fonts and `tfm` files

The external file needed for the font is a `tfm` ( $\TeX$  font metrics) file, which is taken independent of any `<at clause>` in the `\font` declaration. If the `tfm` file has been loaded already (for instance by  $\text{Ini}\TeX$  when it constructed the format), an assignment of that font file can be reexecuted without needing recourse to the `tfm` file.

Font design sizes are given in the font metrics files. The `cmr10` font, for instance, has a design size of 10 point. However, there is not much in the font

that actually has a size of 10 points: the opening and closing parentheses are two examples, but capital letters are considerably smaller.

## 4.2.2 Querying the current font and font names

It was already mentioned above that the control sequence which set the current font can be retrieved by the command `\the\font`. This is a special case of

`\the<font>`

where

`<font>`  $\longrightarrow$  `\font` | `<fontdef token>` | `<family member>`

`<family member>`  $\longrightarrow$  `<font range>``<4-bit number>`

`<font range>`  $\longrightarrow$  `\textfont` | `\scriptfont` | `\scriptscriptfont`

A `<fontdef token>` is a control sequence defined by `\font`, or the predefined control sequence `\nullfont`. The concept of `<family member>` is only relevant in math mode.

Also, the external name of fonts can be retrieved:

`\fontname<font>`

gives a sequence of character tokens of category 12 (but space characters get category 10) that spells the font file name, plus an `<at clause>` if applicable.

例子: *After*

```
\font\tenroman=cmr10 \tenroman
```

*the calls `\the\font` and `\the\tenroman` both give `\tenroman`. The call `\fontname\tenroman` gives `cmr10`.*

## 4.2.3 `\nullfont`

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$  always knows a font that has no characters: the `\nullfont`. If no font has been specified, or if in math mode a family member is needed that has not been specified,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  will take its characters from the nullfont. This control sequence qualifies as a `<fontdef token>`: it acts like any other control sequence that stands for a font; it just does not have an associated `tfm` file.

## 4.3 Font information

During a run of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  the main information needed about the font consists of the dimensions of the characters.  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  finds these in the font metrics files, which usually have extension `.tfm`. Such files contain

- global information: the `\fontdimen` parameters, and some other information,
- dimensions and the italic corrections of characters, and
- ligature and kerning programs for characters.

Also, the design size of a font is specified in the `tfm` file; see above. The definition of the `tfm` format can be found in [23].

### 4.3.1 Font dimensions

Text fonts need to have at least seven `\fontdimen` parameters to describe *font dimensions* (but  $\text{\TeX}$  will take zero for unspecified parameters); math symbol and math extension fonts have more (see page 224). For text fonts the minimal set of seven comprises the following:

1. the slant per point; this dimension is used for the proper horizontal positioning of accents;
2. the interword space: this is used unless the user specifies an explicit `\spaceskip`; see Chapter 20;
3. interword stretch: the stretch component of the interword space;
4. interword shrink: the shrink component of the interword space;
5. the x-height: the value of the  $\langle\text{internal unit}\rangle_{\text{ex}}$ , which is usually about the height of the lowercase letter ‘x’;
6. the quad width: the value of the  $\langle\text{internal unit}\rangle_{\text{em}}$ , which is approximately the width of the capital letter ‘M’; and
7. the extra space: the space added to the interword space at the end of sentences (that is, when `\spacefactor`  $\geq 2000$ ) unless the user specifies an explicit `\xspaceskip`.

Parameters 1 and 5 are purely information about the font and there is no point in varying them. The values of other parameters can be changed in order to adjust spacing; see Chapter 20 for examples of changing parameters 2, 3, 4, and 7.

Font dimensions can be altered in a  $\langle\text{font assignment}\rangle$ , which is a  $\langle\text{global assignment}\rangle$  (see page 112):

```
\fontdimen<number>\font\equals<dimen>
```

See above for the definition of  $\langle\text{font}\rangle$ .

### 4.3.2 Kerning

Some combinations of characters should be moved closer together than would be the case if their bounding boxes were to be just abutted. This fine spacing is called *kerning*, and a proper kerning is as essential to a font as the design of the letter shapes.

Consider as an example

‘Vo’ versus the unkered variant ‘Vo’

Kerning in  $\text{\TeX}$  is controlled by information in the `tfm` file, and is therefore outside the influence of the user. The `tfm` file can be edited, however (see Chapter 33).

The `\kern` command has (almost) nothing to do with the phenomenon of kerning; it is explained in Chapter 8.

### 4.3.3 Italic correction

The primitive control symbol `\/` inserts the *italic correction* of the previous character or ligature. Such a correction may be necessary owing to the definition of the *bounding box* of a character. This box always has vertical sides, and the width of the character as  $\text{\TeX}$  perceives it is the distance between these sides. However, in order to achieve proper spacing for slanted or italic typefaces, characters may very well project outside their bounding boxes. The italic correction is then needed if such an overhanging character is followed by a character from a non-slanting typeface.

Compare for instance

‘ $\text{\TeX}$  has’ to ‘ $\text{\TeX}$  has’,

where the second version was typed as

```
{\italic\TeX\/} has
```

The size of the italic correction of each character is determined by font information in the font metrics file; for the Computer Modern fonts it is approximately half the ‘overhang’ of the characters; see [17]. Italic correction is not the same as `\fontdimen1`, slant per point. That font dimension is used only for positioning accents on top of characters.

An italic correction can only be inserted if the previous item processed by  $\text{\TeX}$  was a character or ligature. Thus the following solution for roman text inside an italic passage does not work:

```
{\italic Some text {\/\roman not} emphasized}
```

The italic correction has no effect here, because the previous item is glue.

#### 4.3.4 Ligatures

Replacement of character sequences by *ligatures* is controlled by information in the `tfm` file of a font. Ligatures are formed from `<character>` commands: sequences such as `fi` are replaced by ‘fi’ in some fonts.

Other ligatures traditionally in use are between `ff`, `ffi`, `fl`, and `ffl`; in some older works `ft` and `st` can be found, and similarly to the `fl` ligature `fk` and `fb` can also occur.

Ligatures in  $\text{\TeX}$  can be formed between explicit character tokens, `\char` commands, and `<chardef token>`s. For example, the sequence `\char`f\char`i` is replaced by the ‘fi’ ligature, if such a ligature is part of the font.

Unwanted ligatures can be suppressed in a number of ways: the unwanted ligature ‘half life’ can for instance be prevented by

```
half{}life, half{l}life, half\/life, or half\hbox{}life
```

but the solution using italic correction is not equivalent to the others.

#### 4.3.5 Boundary ligatures

Each word is surrounded by a left and a right boundary character ( $\text{\TeX}$ 3 only). This makes phenomena possible such as the two different sigmas in Greek: one at the end of a word, and one for every other position. This can be realized through a ligature with the boundary character. A `\noboundary` command immediately before or after a word suppresses the boundary character at that place.

In general, the ligature mechanism has become more complicated with the transition to  $\text{\TeX}$  version 3; see [20].



## 第 5 章 Boxes

The horizontal and vertical boxes of  $\text{T}_{\text{E}}\text{X}$  are containers for pieces of horizontal and vertical lists. Boxes can be stored in box registers. This chapter treats box registers and such aspects of boxes as their dimensions, and the way their components are placed relative to each other.

`\hbox` Construct a horizontal box.

`\vbox` Construct a vertical box with reference point of the last item.

`\vtop` Construct a vertical box with reference point of the first item.

`\vcenter` Construct a vertical box vertically centred on the math axis; this command can only be used in math mode.

`\vsplit` Split off the top part of a vertical box.

`\box` Use a box register, emptying it.

`\setbox` Assign a box to a box register.

`\copy` Use a box register, but retain the contents.

`\ifhbox` `\ifvbox` Test whether a box register contains a horizontal/vertical box.

`\ifvoid` Test whether a box register is empty.

`\newbox` Allocate a new box register.

`\unhbox` `\unvbox` Unpack a box register containing a horizontal/vertical box, adding the contents to the current horizontal/vertical list, and emptying the register.

`\unhcopy` `\unvcopy` The same as `\unhbox`/`\unvbox`, but do not empty the register.

`\ht` `\dp` `\wd` Height/depth/width of the box in a box register.

`\boxmaxdepth` Maximum allowed depth of boxes. Plain  $\text{T}_{\text{E}}\text{X}$  default: `\maxdimen`.

`\splitmaxdepth` Maximum allowed depth of boxes generated by `\vsplit`.

`\badness` Badness of the most recently constructed box.

`\hfuzz` `\vfuzz` Excess size that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  tolerates before it considers a horizontal/vertical box overfull.

`\hbadness` `\vbadness` Amount of tolerance before  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  reports an underfull or overfull horizontal/vertical box.

`\overfullrule` Width of the rule that is printed to indicate overfull horizontal boxes.

`\hsize` Line width used for text typesetting inside a vertical box.

`\vsize` Height of the page box.

`\lastbox` Register containing the last item added to the current list, if this was a box.

`\raise` `\lower` Adjust vertical positioning of a box in horizontal mode.

`\moveleft` `\moveright` Adjust horizontal positioning of a box in vertical mode.

`\everyhbox` `\everyvbox` Token list inserted at the start of a horizontal/vertical box.

## 5.1 Boxes

In this chapter we shall look at boxes. Boxes are containers for pieces of horizontal or vertical lists. Boxes that are needed more than once can be stored in box registers.

When  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  expects a  $\langle\text{box}\rangle$ , any of the following forms is admissible:

- `\hbox` $\langle\text{box specification}\rangle\{\langle\text{horizontal material}\rangle\}$
- `\vbox` $\langle\text{box specification}\rangle\{\langle\text{vertical material}\rangle\}$
- `\vtop` $\langle\text{box specification}\rangle\{\langle\text{vertical material}\rangle\}$
- `\box` $\langle\text{8-bit number}\rangle$
- `\copy` $\langle\text{8-bit number}\rangle$
- `\vsplit` $\langle\text{8-bit number}\rangle\text{to}\langle\text{dimen}\rangle$
- `\lastbox`

A  $\langle\text{box specification}\rangle$  is defined as

$\langle\text{box specification}\rangle \longrightarrow \langle\text{filler}\rangle$   
 | to  $\langle\text{dimen}\rangle\langle\text{filler}\rangle$  | spread  $\langle\text{dimen}\rangle\langle\text{filler}\rangle$

An  $\langle\text{8-bit number}\rangle$  is a number in the range 0–255.

The braces surrounding box material define a group; they can be explicit characters of categories 1 and 2 respectively, or control sequences `\let` to such characters; see also below.

A `\box` can in general be used in horizontal, vertical, and math mode, but see below for the `\lastbox`. The connection between boxes and modes is explored further in Chapter 6.

The box produced by `\vcenter` – a command that is allowed only in math mode – is not a `\box`. For instance, it can not be assigned with `\setbox`; see further Chapter 23.

The `\vsplit` operation is treated in Chapter 27.

## 5.2 Box registers

There are 256 box registers, numbered 0–255. Either a box register is empty (‘void’), or it contains a horizontal or vertical box. This section discusses specifically box *registers*; the sizes of boxes, and the way material is arranged inside them, is treated below.

### 5.2.1 Allocation: `\newbox`

The plain TeX `\newbox` macro allocates an unused box register:

```
\newbox\MyBox
```

after which one can say

```
\setbox\MyBox=...
```

or

```
\box\MyBox
```

and so on. Subsequent calls to this macro give subsequent box numbers; this way macro collections can allocate their own boxes without fear of collision with other macros.

The number of the box is assigned by `\chardef` (see Chapter 31). This implies that `\MyBox` is equivalent to, and can be used as, a `\number`. The control sequence `\newbox` is an `\outer` macro. Newly allocated box registers are initially empty.

### 5.2.2 Usage: `\setbox`, `\box`, `\copy`

A register is filled by assigning a `\box` to it:

```
\setbox\<number>\<equals>\<box>
```

For example, the `<box>` can be explicit

```
\setbox37=\hbox{...} or \setbox37=\vbox{...}
```

or it can be a box register:

```
\setbox37=\box38
```

Usually, box numbers will have been assigned by a `\newbox` command.

The box in a box register is appended by the commands `\box` and `\copy` to whatever list  $\TeX$  is building: the call

```
\box38
```

appends box 38. To save memory space, box registers become empty by using them:  $\TeX$  assumes that after you have inserted a box by calling `\boxnn` in some mode, you do not need the contents of that register any more and empties it. In case you *do* need the contents of a box register more than once, you can `\copy` it. Calling `\copynn` is equivalent to `\boxnn` in all respects except that the register is not cleared.

It is possible to unwrap the contents of a box register by ‘unboxing’ it using the commands `\unhbox` and `\unvbox`, and their copying versions `\unhcopy` and `\unvcopy`. Whereas a box can be used in any mode, the unboxing operations can only be used in the appropriate mode, since in effect they contribute a partial horizontal or vertical list (see also Chapter 6). See below for more information on unboxing registers.

### 5.2.3 Testing: `\ifvoid`, `\ifhbox`, `\ifvbox`

Box registers can be tested for their contents:

```
\ifvoid<number>
```

is true if the box register is empty. Note that an empty, or ‘void’, box register is not the same as a register containing an empty box. An empty box is still either a horizontal or a vertical box; a void register can be used as both.

The test

```
\ifhbox<number>
```

is true if the box register contains a horizontal box;

```
\ifvbox<number>
```

is true if the box register contains a vertical box. Both tests are false for void registers.

### 5.2.4 The `\lastbox`

When  $\TeX$  has built a partial list, the last box in this list is accessible as the `\lastbox`. This behaves like a box register, so you can remove the last box from the list by assigning the `\lastbox` to some box register. If the last item on the current list is not a box, the `\lastbox` acts like a void box register. It is not possible to get hold of the last box in the case of the main vertical list. The `\lastbox` is then always void.

As an example, the statement

```
{\setbox0=\lastbox}
```

removes the last box from the current list, assigning it to box register 0. Since this assignment occurs inside a group, the register is cleared at the end of the group. At the start of a paragraph this can be used to remove the indentation box (see Chapter 16). Another example of `\lastbox` can be found on page 72.

Because the `\lastbox` is always empty in external vertical mode, it is not possible to get hold of boxes that have been added to the page. However, it is possible to dissect the page once it is in `\box255`, for instance doing

```
\vbox{\unvbox255{\setbox0=\lastbox}}
```

inside the output routine.

If boxes in vertical mode have been shifted by `\moveright` or `\moveleft`, or if boxes in horizontal mode have been raised by `\raise` or lowered by `\lower`, any information about this displacement due to such a command is lost when the `\lastbox` is taken from the list.

## 5.3 Natural dimensions of boxes

### 5.3.1 Dimensions of created horizontal boxes

Inside an `\hbox` all constituents are lined up next to each other, with their reference points on the baseline of the box, unless they are moved explicitly in the vertical direction by `\lower` or `\raise`.

The resulting width of the box is the sum of the widths of the components. Thus the width of

```
\hbox{\hskip1cm}
```

is positive, and the width of

```
\hbox{\hskip-1cm}
```

is negative. By way of example,

```
a\hbox{\kern-1em b}--
```

gives as output

`ba-`

which shows that a horizontal box can have negative width.

The height and depth of an `\hbox` are the maximum amount that constituent boxes project above and below the baseline of the box. They are non-negative when the box is created.

The commands `\lower` and `\raise` are the only possibilities for vertical movement inside an `\hbox` (other than including a `\vbox` inside the `\hbox`, of course); a `\langle`vertical command`\rangle` – such as `\vskip` – is not allowed in a horizontal box, and `\par`, although allowed, does not do anything inside a horizontal box.

### 5.3.2 Dimensions of created vertical boxes

Inside a `\vbox` vertical material is lined up with the reference points on the vertical line through the reference point of the box, unless components are moved explicitly in the horizontal direction by `\moveleft` or `\moveright`.

The reference point of a vertical box is always located at the left boundary of the box. The width of a vertical box is then the maximal amount that any material in the box sticks to the right of the reference point. Material to the left of the reference point is not taken into account in the width. Thus the result of

```
a\vbox{\hbox{\kern-1em b}}--
```

is

`ba-`

This should be contrasted with the above example.

The calculation of height and depth is different for vertical boxes constructed by `\vbox` and `\vtop`. The ground rule is that a `\vbox` has a reference point that lies on the baseline of its last component, and a `\vtop` has its reference point on the baseline of the first component. In general, the depth (height) of a `\vbox` (`\vtop`) can be non-zero if the last (first) item is a box or rule.

The height of a `\vbox` is then the sum of the heights and depths of all components except the last, plus the height of that last component; the depth of the `\vbox` is the depth of its last component. The depth of a `\vtop` is the sum of the depth of the first component and the heights and depths of all subsequent material; its height is the height of the first component.

However, the actual rules are a bit more complicated when the first component of a `\vtop` or the last component of a `\vbox` is not a box or rule. If the last component of a `\vbox` is a kern or a glue, the depth of that box is zero; a `\vtop`'s height is zero unless its first component is a box or rule. (Note the asymmetry

in these definitions; see below for an example illustrating this.) The depth of a `\vtop`, then, is equal to the total height plus depth of all enclosed material minus the height of the `\vtop`.

There is a limit on the depth of vertical boxes: if the depth of a `\vbox` or `\vtop` calculated by the above rules would exceed `\boxmaxdepth`, the reference point of the box is moved down by the excess amount. More precisely, the excess depth is added to the natural height of the box. If the box had a `to` or `spread` specification, any glue is set anew to take the new height into account.

Ordinarily, `\boxmaxdepth` is set to the maximum dimension possible in  $\text{\TeX}$ . It is for instance reduced during some of the calculations in the plain  $\text{\TeX}$  output routine; see Chapter 28.

### 5.3.3 Examples

Horizontal boxes are relatively straightforward. Their width is the distance between the ‘beginning’ and the ‘end’ of the box, and consequently the width is not necessarily positive. With

```
\setbox0=\hbox{aa} \setbox1=\hbox{\copy0 \hskip-\wd0}
```

the `\box1` has width zero;

```
/\box1/ gives ‘aa’
```

The height and depth of a horizontal box cannot be negative: in

```
\setbox0=\hbox{\vrule height 5pt depth 5pt}
\setbox1=\hbox{\raise 10pt \box0}
```

the `\box1` has depth 0pt and height 15pt

Vertical boxes are more troublesome than horizontal boxes. Let us first treat their width. After

```
\setbox0=\hbox{\hskip 10pt}
```

the box in the `\box0` register has a width of 10pt. Defining

```
\setbox1=\vbox{\moveleft 5pt \copy0}
```

the `\box1` will have width 5pt; material to the left of the reference point is not accounted for in the width of a vertical box. With

```
\setbox2=\vbox{\moveright 5pt \copy0}
```

the `\box2` will have width 15pt.

The depth of a `\vbox` is the depth of the last item if that is a box, so

```
\vbox{\vskip 5pt \hbox{\vrule height 5pt depth 5pt}}
```

has height 10pt and depth 5pt, and

```
\vbox{\vskip -5pt \hbox{\vrule height 5pt depth 5pt}}
```

has height 0pt and depth 5pt. With a glue or kern as the last item in the box, the resulting depth is zero, so

```
\vbox{\hbox{\vrule height 5pt depth 5pt}\vskip 5pt}
```

has height 15pt and depth 0pt;

```
\vbox{\hbox{\vrule height 5pt depth 5pt}\vskip -5pt}
```

has height 5pt and depth 0pt.

The height of a `\vtop` behaves (almost) the same with respect to the first item of the box, as the depth of a `\vbox` does with respect to the last item. Repeating the above examples with a `\vtop` gives the following:

```
\vtop{\vskip 5pt \hbox{\vrule height 5pt depth 5pt}}
```

has height 0pt and depth 15pt, and

```
\vtop{\vskip -5pt \hbox{\vrule height 5pt depth 5pt}}
```

has height 0pt and depth 5pt;

```
\vtop{\hbox{\vrule height 5pt depth 5pt} \vskip 5pt}
```

has height 5pt and depth 10pt, and

```
\vtop{\hbox{\vrule height 5pt depth 5pt} \vskip -5pt}
```

has height 5pt and depth 0pt.

## 5.4 More about box dimensions

### 5.4.1 Predetermined dimensions

The size of a box can be specified in advance with a `<box specification>`; see above for the syntax. Any glue in the box is then set in order to reach the required size. Prescribing the size of the box is done by

```
\hbox to <dimen> {...}, \vbox to <dimen> {...}
```

If stretchable or shrinkable glue is present in the box, it is stretched or shrunk in order to give the box the specified size. Associated with this glue setting is a badness value (see Chapter 8). If no stretch or shrink – whichever is necessary – is present, the resulting box will be underfull or overfull respectively. Error reporting for over/underfull boxes is treated below.

Another command to let a box have a size other than the natural size is

```
\hbox spread <dimen> {...}, \vbox spread <dimen> {...}
```

which tells  $\text{\TeX}$  to set the glue in such a way that the size of the box is a specified amount more than the natural size.

Box specifications for `\vtop` vertical boxes are somewhat difficult to interpret.  $\text{\TeX}$  constructs a `\vtop` by first making a `\vbox`, including glue settings



induced by a  $\langle$ box specification $\rangle$ ; then it computes the height and depth by the above rules. Glue setting is described in Chapter 8.

### 5.4.2 Changes to box dimensions

The dimensions of a box register are accessible by the commands `\ht`, `\dp`, and `\wd`; for instance `\dp13` gives the depth of box 13. However, not only can boxes be measured this way; by assigning values to these dimensions  $\text{\TeX}$  can even be fooled into thinking that a box has a size different from its actual. However, changing the dimensions of a box does not change anything about the contents; in particular it does not change the way the glue is set.

Various formats use this in ‘smash’ macros: the macro defined by

```
\def\smash#1{\setbox0=\hbox{#1}\dp0=0pt \ht0=0pt \box0\relax}}
```

places its argument but annihilates its height and depth; that is, the output does show the whole box, but further calculations by  $\text{\TeX}$  act as if the height and depth were zero.

Box dimensions can be changed only by setting them. They are  $\langle$ box dimension $\rangle$ s, which can only be set in a  $\langle$ box size assignment $\rangle$ , and not, for instance changed with `\advance`.

Note that a  $\langle$ box size assignment $\rangle$  is a  $\langle$ global assignment $\rangle$  its effect transcends any groups in which it occurs (see Chapter 10). Thus the output of

```
\setbox0=\hbox{---} {\wd0=0pt} a\box0b
```

is ‘ $\text{ab}$ ’.

The limits that hold on the dimensions with which a box can be created (see above) do not hold for explicit changes to the size of a box: the assignment `\dp0=-2pt` for a horizontal box is perfectly admissible.

### 5.4.3 Moving boxes around

In a horizontal box all constituent elements are lined up with their reference points at the same height as the reference point of the box. Any box inside a horizontal box can be lifted or dropped using the macros `\raise` and `\lower`.

Similarly, in a vertical box all constituent elements are lined up with their reference points underneath one another, in line with the reference point of the box. Boxes can now be moved sideways by the macros `\moveleft` and `\moveright`.

Only boxes can be shifted thus; these operations cannot be applied to, for instance, characters or rules.

#### 5.4.4 Box dimensions and box placement

$\text{\TeX}$  places the components of horizontal and vertical lists by maintaining a reference line and a current position on that line. For horizontal lists the reference line is the baseline of the surrounding  $\text{\hbox}$ ; for vertical lists it is the vertical line through the reference point of the surrounding  $\text{\vbox}$ .

In horizontal mode a component is placed as follows. The current position coincides initially with the reference point of the surrounding box. After that, the following actions are carried out.

1. If the component has been shifted by  $\text{\raise}$  or  $\text{\lower}$ , shift the current position correspondingly.
2. If the component is a horizontal box, use this algorithm recursively for its contents; if it is a vertical box, go up by the height of this box, putting a new current position for the enclosed vertical list there, and place its components using the algorithm for vertical lists below.
3. Move the current position (on the reference line) to the right by the width of the component.

For the list in a vertical box  $\text{\TeX}$ 's current position is initially at the upper left corner of that box, as explained above, and the reference line is the vertical line through that point; it also runs through the reference point of the box. Enclosed components are then placed as follows.

1. If a component has been shifted using  $\text{\moveleft}$  or  $\text{\moveright}$ , shift the current position accordingly.
2. Put the component with its upper left corner at the current position.
3. If the component is a vertical box, use this algorithm recursively for its contents; if it is a horizontal box, its reference point can be found below the current position by the height of the box. Put the current position for that box there, and use the above algorithm for horizontal lists.
4. Go down by the height plus depth of the box (that is, starting at the upper left corner of the box) on the reference line, and continue processing vertically.

Note that the above processes do not describe the construction of boxes. That would (for instance) involve for vertical boxes the insertion of  $\text{\baselineskip}$  glue. Rather, it describes the way the components of a finished box are arranged in the output.

### 5.4.5 Boxes and negative glue

Sometimes it is useful to have boxes overlapping instead of line up. An easy way to do this is to use negative glue. In horizontal mode

```
{\dimen0=\wd8 \box8 \kern-\dimen0}
```

places box 8 without moving the current location.

More versatile are the macros `\llap` and `\rlap`, defined as

```
\def\llap#1{\hbox to 0pt{\hss #1}}
```

and

```
\def\rlap#1{\hbox to 0pt{#1\hss}}
```

that allow material to protrude left or right from the current location. The `\hss` glue is equivalent to `\hskip 0pt plus 1fil minus 1fil`, which absorbs any positive or negative width of the argument of `\llap` or `\rlap`.

例子: *The sequence*

```
\llap{\hbox to 10pt{a\hfil}}
```

is effectively the same as

```
\hbox{\hskip-10pt \hbox to 10pt{a\hfil}}
```

which has a total width of 0pt.

## 5.5 Overfull and underfull boxes

If a box has a size specification  $\TeX$  will stretch or shrink glue in the box. For glue with only finite stretch or shrink components the *badness* (see Chapter 19) of stretching or shrinking is computed. In  $\TeX$  version 3 the badness of the box most recently constructed is available for inspection by the user through the `\badness` parameter. Values for badness range 0–10 000, but if the box is overfull it is 1 000 000.

When  $\TeX$  considers the badness too large, it gives a diagnostic message. Let us first consider error reporting for horizontal boxes.

Horizontal boxes of which the glue has to stretch are never reported if `\hbadness`  $\geq 10\,000$ ; otherwise  $\TeX$  reports them as ‘underfull’ if their badness is more than `\hbadness`.

Glue shrinking can lead to ‘overfull’ boxes: a box is called overfull if the available shrink is less than the shrink necessary to meet the box specification. An overfull box is only reported if the difference in shrink is more than `\hfuzz`, or if `\hbadness`  $< 100$  (and it turns out that using all available shrinkability has badness 100).

例子: *Setting `\hfuzz=1pt` will let  $\TeX$  ignore boxes that can not shrink enough if they lack less than 1pt. In*

```
\hbox to 1pt{\hskip3pt minus .5pt}

\hbox to 1pt{\hskip3pt minus 1.5pt}
```

*only the first box will give an error message: it is 1.5pt too big, whereas the second lacks .5pt which is less than `\hfuzz`.*

Also, boxes that shrink but that are not overfull can be reported: if a box is ‘tight’, that is, if it uses at least half its shrinkability,  $\TeX$  reports this fact if the computed badness (which is between 13 and 100) is more than `\hbadness`.

For horizontal and vertical boxes this error reporting is almost the same, with parameters `\vbadness` and `\vfuzz`. The difference is that for horizontal overfull boxes  $\TeX$  will draw a rule to the right of the box that has the same height as the box, and width `\overfullrule`. No overfull rule ensues if the `\tabskip` glue in an `\halign` cannot be shrunk enough.

## 5.6 Opening and closing boxes

The opening and closing braces of a box can be either explicit, that is, character tokens of category 1 and 2, or implicit, a control sequence `\let` to such a character. After the opening brace the `\everyhbox` or `\everyvbox` tokens are inserted. If this box appeared in a `\setbox` assignment any `\afterassignment` token is inserted even before the ‘everybox’ tokens.

例子:

```
\everyhbox{b}
\afterassignment a
\setbox0=\hbox{c}
\showbox0
```

*gives*

```
> \box0=
\hbox(6.94444+0.0)x15.27782
.\tenrm a
.\tenrm b
.\kern0.27779
.\tenrm c
```

Implicit braces can be used to let a box be opened or closed by a macro, for example:

```

\def\openbox#1{\setbox#1=\hbox\bgroup}
\def\closebox#1{\egroup\DoSomethingWithBox#1}
\openbox0 ... \closebox0

```

This mechanism can be used to scoop up paragraphs:

```

\everypar{\setbox\parbox=
  \vbox\bgroup
    \everypar{ }
  \def\par{\egroup\UseBox\parbox}}

```

Here the `\everypar` opens the box and lets the text be set in the box: starting for instance

```
Begin a text ...
```

gives the equivalent of

```
\setbox\parbox=\vbox{Begin a text ...}
```

Inside the box `\par` has been redefined, so

```
... a text ends.\par
```

is equivalent to

```
... a text ends.}\Usebox\parbox
```

In this example, the `\UseBox` command can only treat the box as a whole; if the elements of the box should somehow be treated separately another approach is necessary. In

```

\everypar{\setbox\parbox=
  \vbox\bgroup\everypar{ }%
  \def\par{\endgraf\HandleLines
    \egroup\box\parbox}}
\def\HandleLines{ ... \lastbox ... }

```

the macro `\HandleLines` can have access to successive elements from the vertical list of the paragraph. See also the example on page [72](#).

## 5.7 Unboxing

Boxes can be unwrapped by the commands `\unhbox` and `\unvbox`, and by their copying versions `\unhcopy` and `\unvcopy`. These are horizontal and vertical commands (see Chapter 6), considering that in effect they contribute a partial horizontal or vertical list. It is not possible to `\unhbox` a register containing a `\vbox` or vice versa, but a void box register can both be `\unhboxed` and `\unvboxed`.

Unboxing takes the contents of a box in a box register and appends them to the surrounding list; any glue can then be set anew. Thus

```
\setbox0=\hbox to 1cm{\hfil} \hbox to 2cm{\unhbox0}
```

is completely equivalent to

```
\hbox to 2cm{\hfil}
```

and not to

```
\hbox to 2cm{\kern1cm}
```

The intrinsically horizontal nature of `\unhbox` is used to define

```
\def\leavevmode{\unhbox\voidb@x}
```

This command switches from vertical mode to horizontal without adding anything to the horizontal list. However, the subsequent `\indent` caused by this transition adds an indentation box. In horizontal mode the `\leavevmode` command has no effect. Note that here it is not necessary to use `\unhcopy`, because the register is empty anyhow.

Beware of the following subtlety: unboxing in vertical mode does not add interline glue between the box contents and any preceding item. Also, the value of `\prevdepth` is not changed, so glue between the box contents and any following item will occur only if there was something preceding the box; interline glue will be based on the depth of that preceding item. Similarly, unboxing in horizontal mode does not influence the `\spacefactor`.

## 5.8 Text in boxes

Both horizontal and vertical boxes can contain text. However, the way text is treated differs. In horizontal boxes the text is placed in one straight line, and the width of the box is in principle the natural width of the text (and other items) contained in it. No `<vertical command>`s are allowed inside a horizontal box, and `\par` does nothing in this case.

For vertical boxes the situation is radically different. As soon as a character, or any other `<horizontal command>` (see page 75), is encountered in a vertical box,  $\TeX$  starts building a paragraph in unrestricted horizontal mode, that is, just as if the paragraph were directly part of the page. At the occurrence of a `<vertical command>` (see page 75), or at the end of the box, the paragraph is broken into lines using the current values of parameters such as `\hsize`.

Thus

```
\hbox to 3cm{\vbox{some reasonably long text}}
```

will not give a paragraph of width 3 centimetres (it gives an overfull horizontal box if `\hsize > 3cm`). However,

```
\vbox{\hsize=3cm some reasonably long text}
```

will be 3 centimetres wide.

A paragraph of text inside a vertical box is broken into lines, which are packed in horizontal boxes. These boxes are then stacked in internal vertical mode, possibly with `\baselineskip` and `\lineskip` separating them (this is treated in Chapter 15). This process is also used for text on the page; the boxes are then stacked in outer vertical mode.

If the internal vertical list is empty, no `\parskip` glue is added at the start of a paragraph.

Because text in a horizontal box is not broken into lines, there is a further difference between text in restricted and unrestricted horizontal mode. In restricted horizontal mode no discretionary nodes and whatsit items changing the value of the current language are inserted. This may give problems if the text is subsequently unboxed to form part of a paragraph.

See Chapter 19 for an explanation of these items, and [7] for a way around this problem.

## 5.9 Assorted remarks

### 5.9.1 Forgetting the `\box`

After `\newcount\foo`, one can use `\foo` on its own to get the `\foo` counter. For boxes, however, one has to use `\box\foo` to get the `\foo` box. The reason for this is that there exists no separate `\boxdef` command, so `\chardef` is used (see Chapter 31).

例子: *Suppose `\newbox\foo` allocates box register 25; then typing `\foo` is equivalent to typing `\char25`.*

### 5.9.2 Special-purpose boxes

Some box registers have a special purpose:

- `\box255` is by used  $\TeX$  internally to give the page to the output routine.
- `\voidb@x` is the number of a box register allocated in `plain.tex`; it is supposed to be empty always. It is used in the macro `\leavevmode` and others.
- when a new `\insert` is created with the plain  $\TeX$  `\newinsert` macro, a `\count`, `\dimen`, `\skip`, and `\box` all with the same number are reserved for that insert. The numbers for these registers count down from 254.

### 5.9.3 The height of a vertical box in horizontal mode

In horizontal mode a vertical box is placed with its reference point aligned vertically with the reference point of the surrounding box.  $\TeX$  then traverses its contents starting at the left upper corner; that is, the point that lies above the reference point by a distance of the height of the box. Changing the height of the box implies then that the contents of the box are placed at a different height.

Consider as an example

```
\hbox{a\setbox0=\vbox{\hbox{b}}\box0 c}
```

which gives

abc

and

```
\hbox{a\setbox0=\vbox{\hbox{b}}\ht0=0cm \box0 c}
```

which gives

a<sub>b</sub>c

By contrast, changing the width of a box placed in vertical mode has no effect on its placement.

### 5.9.4 More subtleties with vertical boxes

Since there are two kinds of vertical boxes, the `\vbox` and the `\vtop`, using these two kinds nested may lead to confusing results. For instance,

```
\vtop{\vbox{...}}
```

is completely equivalent to just

```
\vbox{...}
```

It was stated above that the depth of a `\vbox` is zero if the last item is a kern or glue, and the height of a `\vtop` is zero unless the first item in it is a box. The above examples used a kern for that first or last item, but if, in the case of a `\vtop`, this item is not a glue or kern, one is apt to overlook the effect that it has on the surrounding box. For instance,

```
\vtop{\write16{...}...}
```

has zero height, because the write instruction is packed into a ‘whatsit’ item that is placed on the current, that is, the vertical, list. The remedy here is

```
\vtop{\leavevmode\write16{...}...}
```

which puts the whatsit in the beginning of the paragraph, instead of above it.

Placement of items in a vertical list is sometimes a bit tricky. There is for instance a difference between how vertical and horizontal boxes are treated in



a vertical list. Consider the following examples. After `\offinterlineskip` the first example

```
\vbox{\hbox{a}
      \setbox0=\vbox{\hbox{}}
      \ht0=0pt \dp0=0pt \box0
      \hbox{ b}}
```

gives

a  
b

while a slight variant

```
\vbox{\hbox{a}
      \setbox0=\hbox{ }
      \ht0=0pt \dp0=0pt \box0
      \hbox{ b}}
```

gives

a  
b

The difference is caused by the fact that horizontal boxes are placed with respect to their reference point, but vertical boxes with respect to their upper left corner.

### 5.9.5 Hanging the `\lastbox` back in the list

You can pick the last box off a vertical list that has been compiled in (internal) vertical mode. However, if you try to hang it back in the list the vertical spacing may go haywire. If you just hang it back,

```
\setbox\tmpbox=\lastbox
\usethetmpbox \box\tmpbox
```

baselineskip glue is added a second time. If you ‘unskip’ prior to hanging the box back,

```
\setbox\tmpbox=\lastbox \unskip
\usethetmpbox \box\tmpbox
```

things go wrong in a more subtle way. The `(internal dimen) \prevdepth` (which controls interline glue; see Chapter 15) will have a value based on the last box, but what you need for the proper interline glue is a depth based on one box earlier. The solution is not to unskip, but to specify `\nointerlineskip`:

```
\setbox\tmpbox=\lastbox
\usethetmpbox \nointerlineskip \box\tmpbox
```

### 5.9.6 Dissecting paragraphs with `\lastbox`

Repeatedly applying `\last...` and `\un...` macros can be used to take a paragraph apart. Here is an example of that.

In typesetting advertisement copy, a way of justifying paragraphs has become popular in recent years that is somewhere between flushright and raggedright setting. Lines that would stretch beyond certain limits are set with their glue at natural width. This single paragraph is but an example of this procedure; the macros are given next.

```
\newbox\linebox \newbox\snapbox
\def\eatlines{
  \setbox\linebox\lastbox      % check the last line
  \ifvoid\linebox
  \else                        % if it's not empty
  \unskip\unpenalty           % take whatever is
  {\eatlines}                 % above it;
                              % collapse the line
  \setbox\snapbox\hbox{\unhcopy\linebox}
                              % depending on the difference
  \ifdim\wd\snapbox<.98\wd\linebox
    \box\snapbox % take the one or the other,
  \else \box\linebox \fi
  \fi}
```

This macro can be called as

```
\vbox{ ... some text ... \par\eatlines}
```

or it can be inserted automatically with `\everypar`; see [10].

In the macro `\eatlines`, the `\lastbox` is taken from a vertical list. If the list is empty the last box will test true on `\ifvoid`. These boxes containing lines from a paragraph are actually horizontal boxes: the test `\ifhbox` applied to them would give a true result.

## 第 6 章 Horizontal and Vertical Mode

At any point in its processing  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is in some *mode*. There are six modes, divided in three categories:

1. horizontal mode and restricted horizontal mode,
2. vertical mode and internal vertical mode, and
3. math mode and display math mode.

The math modes will be treated elsewhere (see page 217). Here we shall look at the horizontal and vertical modes, the kinds of objects that can occur in the corresponding lists, and the commands that are exclusive for one mode or the other.

`\ifhmode` Test whether the current mode is (possibly restricted) horizontal mode.

`\ifvmode` Test whether the current mode is (possibly internal) vertical mode.

`\ifinner` Test whether the current mode is an internal mode.

`\vadjust` Specify vertical material for the enclosing vertical list while in horizontal mode.

`\showlists` Write to the log file the contents of the partial lists currently being built in all modes.

### 6.1 Horizontal and vertical mode

When not typesetting mathematics,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is in horizontal or vertical mode, building horizontal or vertical lists respectively. Horizontal mode is typically used to make lines of text; vertical mode is typically used to stack the lines of a paragraph on top of each other. Note that these modes are different from the internal states of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 's input processor (see page 29).

### 6.1.1 Horizontal mode

The main activity in *horizontal mode* is building lines of text. Text on the page and text in a `\vbox` or `\vtop` is built in horizontal mode (this might be called ‘paragraph mode’); if the text is in an `\hbox` there is only one line of text, and the corresponding mode is the restricted horizontal mode.

In horizontal mode all material is added to a horizontal list. If this list is built in unrestricted horizontal mode, it will later be broken into lines and added to the surrounding vertical list.

Each element of a *horizontal list* is one of the following:

- a box (a character, ligature, `\vrule`, or a `\box`),
- a discretionary break,
- a whatsit (see Chapter 30),
- vertical material enclosed in `\mark`, `\vadjust`, or `\insert`,
- glue or leaders, a kern, a penalty, or a math-on/off item.

The items in the last point are all discardable. *Discardable items* are called that, because they disappear in a break. Breaking of horizontal lists is discussed in Chapter 19.

### 6.1.2 Vertical mode

*Vertical mode* can be used to stack items on top of one another. Most of the time, these items are boxes containing the lines of paragraphs.

Stacking material can take place inside a vertical box, but the items that are stacked can also appear by themselves on the page. In the latter case  $\text{\TeX}$  is in vertical mode; in the former case, inside a vertical box,  $\text{\TeX}$  operates in internal vertical mode.

In vertical mode all material is added to a vertical list. If this list is built in external vertical mode, it will later be broken when pages are formed.

Each element of a *vertical list* is one of the following:

- a box (a horizontal or vertical box or an `\hrule`),
- a whatsit,
- a mark,
- glue or leaders, a kern, or a penalty.

The items in the last point are all discardable. Breaking of vertical lists is discussed in Chapter 27.

There are a few exceptional conditions at the beginning of a vertical list:

the value of `\prevdepth` is set to `-1000pt`. Furthermore, no `\parskip` glue is added at the top of an internal vertical list; at the top of the main vertical list (the top of the ‘current page’) no glue or other discardable items are added, and `\topskip` glue is added when the first box is placed on this list (see Chapters 26 and 27).

## 6.2 Horizontal and vertical commands

Some commands are so intrinsically horizontal or vertical in nature that they force  $\TeX$  to go into that mode, if possible. A command that forces  $\TeX$  into horizontal mode is called a *horizontal command*; similarly a command that forces  $\TeX$  into vertical mode is called a *vertical command*.

However, not all transitions are possible:  $\TeX$  can switch from both vertical modes to (unrestricted) horizontal mode and back through horizontal and vertical commands, but no transitions to or from restricted horizontal mode are possible (other than by enclosing horizontal boxes in vertical boxes or the other way around). A vertical command in restricted horizontal mode thus gives an error; the `\par` command in restricted horizontal mode has no effect.

The *horizontal commands* are the following:

- any *letter*, *otherchar*, `\char`, a control sequence defined by `\chardef`, or `\noboundary`;
- `\accent`, `\discretionary`, the discretionary hyphen `\-` and control space `\` ;
- `\unhbox` and `\unhcopy`;
- `\vrule` and the *horizontal skip* commands `\hskip`, `\hfil`, `\hfill`, `\hss`, and `\hfilneg`;
- `\valign`;
- math shift (`$`).

The *vertical commands* are the following:

- `\unvbox` and `\unvcopy`;
- `\hrule` and the *vertical skip* commands `\vskip`, `\vfil`, `\vfill`, `\vss`, and `\vfilneg`;
- `\halign`;
- `\end` and `\dump`.

Note that the vertical commands do not include `\par`; nor are `\indent` and `\noindent` horizontal commands.

The connection between boxes and modes is explored below; see Chapter 9

for more on the connection between rules and modes.

## 6.3 The internal modes

The *restricted horizontal mode* and *internal vertical mode* are those variants of horizontal mode and vertical mode that hold inside an `\hbox` and `\vbox` (or `\vtop` or `\vcenter`) respectively. However, restricted horizontal mode is rather more restricted in nature than internal vertical mode. The third internal mode is non-display math mode (see Chapter 23).

### 6.3.1 Restricted horizontal mode

The main difference between restricted horizontal mode, the mode in an `\hbox`, and unrestricted horizontal mode, the mode in which paragraphs in vertical boxes and on the page are built, is that you cannot break out of restricted horizontal mode: `\par` does nothing in this mode. Furthermore, a `<vertical command>` in restricted horizontal mode gives an error. In unrestricted horizontal mode it would cause a `\par` token to be inserted and vertical mode to be entered (see also Chapter 17).

### 6.3.2 Internal vertical mode

Internal vertical mode, the vertical mode inside a `\vbox`, is a lot like external vertical mode, the mode in which pages are built. A `<horizontal command>` in internal vertical mode, for instance, is perfectly valid:  $\TeX$  then starts building a paragraph in unrestricted horizontal mode.

One difference is that the commands `\unskip` and `\unkern` have no effect in external vertical mode, and `\lastbox` is always empty in external vertical mode. See further pages 59 and 101.

The entries of alignments (see Chapter 25) are processed in internal modes: restricted horizontal mode for the entries of an `\halign`, and internal vertical mode for the entries of a `\valign`. The material in `\vadjust` and `\insert` items is also processed in internal vertical mode; furthermore,  $\TeX$  enters this mode when processing the `\output` token list.

The commands `\end` and `\dump` (the latter exists only in  $\text{\texttt{Init}\TeX}$ ) are not allowed in internal vertical mode; furthermore, `\dump` is not allowed inside a group (see Chapter 33).

## 6.4 Boxes and modes

There are horizontal and vertical boxes, and there is horizontal and vertical mode. Not surprisingly, there is a connection between the boxes and the modes. One can ask about this connection in two ways.

### 6.4.1 What box do you use in what mode?

This is the wrong question. Both horizontal and vertical boxes can be used in both horizontal and vertical mode. Their placement is determined by the prevailing mode at that moment.

### 6.4.2 What mode holds in what box?

This is the right question. When an `\hbox` starts,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is in restricted horizontal mode. Thus everything in a horizontal box is lined up horizontally.

When a `\vbox` is started,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is in internal vertical mode. Boxes of both kinds and other items are then stacked on top of each other.

### 6.4.3 Mode-dependent behaviour of boxes

Any `\box` (see Chapter 5 for the full definition) can be used in horizontal, vertical, and math mode. Unboxing commands, however, are specific for horizontal or vertical mode. Both `\unhbox` and `\unhcopy` are `\horizontal command`s, so they can make  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  switch from vertical to horizontal mode; both `\unvbox` and `\unvcopy` are `\vertical command`s, so they can make  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  switch from horizontal to vertical mode.

In horizontal mode the `\spacefactor` is set to 1000 after a box has been placed. In vertical mode the `\prevdepth` is set to the depth of the box placed. Neither statement holds for unboxing commands: after an `\unhbox` or `\unhcopy` the `\spacefactor` is not altered, and after `\unvbox` or `\unvcopy` the `\prevdepth` remains unchanged. After all, these commands do not add a box, but a piece of a (horizontal or vertical) list.

The operations `\raise` and `\lower` can only be applied to a box in horizontal mode; similarly, `\moveleft` and `\moveright` can only be applied in vertical mode.

## 6.5 Modes and glue

Both in horizontal and vertical mode  $\TeX$  can insert glue items the size of which is determined by the preceding object in the list.

For horizontal mode the amount of glue that is inserted for a space token depends on the `\spacefactor` of the previous object in the list. This is treated in Chapter 20.

In vertical mode  $\TeX$  inserts glue to keep boxes at a certain distance from each other. This glue is influenced by the height of the current item and the depth of the previous one. The depth of items is recorded in the `\prevdepth` parameter (see Chapter 15).

The two quantities `\prevdepth` and `\spacefactor` use the same internal register of  $\TeX$ . Thus the `\prevdepth` can be used or asked only in vertical mode, and the `\spacefactor` only in horizontal mode.

## 6.6 Migrating material

The three control sequences `\insert`, `\mark`, and `\vadjust` can be given in a paragraph (the first two can also occur in vertical mode) to specify *migrating material*: material that will wind up on the surrounding vertical list rather than on the current list. Note that this need not be the main vertical list: it can be a vertical box containing a paragraph of text. In this case a `\mark` or `\insert` command will not reach the page breaking algorithm.

When several migrating items are specified in a certain line of text, their left-to-right order is preserved when they are placed on the surrounding vertical list. These items are placed directly after the horizontal box containing the line of text in which they were specified: they come before any penalty or glue items that are automatically inserted (see page 191).

### 6.6.1 `\vadjust`

The command

```
\vadjust<filler>{<vertical mode material>}
```

is only allowed in horizontal and math modes (but it is not a `<horizontal command>`). Vertical mode material specified by `\vadjust` is moved from the horizontal list in which the command is given to the surrounding vertical list, directly after the box in which it occurred.



- In the current line a `\vadjust` item was placed to put the bullet in the margin.

Any vertical material in a `\vadjust` item is processed in internal vertical mode, even though it will wind up on the main vertical list. For instance, the `\ifinner` test is true in a `\vadjust`, and at the start of the vertical material `\prevdepth=-1000pt`.

## 6.7 Testing modes

The three conditionals `\ifhmode`, `\ifvmode`, and `\ifinner` can distinguish between the four modes of  $\TeX$  that are not math modes. The `\ifinner` test is true if  $\TeX$  is in restricted horizontal mode or internal vertical mode (or in non-display math mode). Exceptional condition: during a `\write`  $\TeX$  is in a ‘no mode’ state. The tests `\ifhmode`, `\ifvmode`, and `\ifmmode` are then all false.

Inspection of all current lists, including the ‘recent contributions’ (see Chapter 27), is possible through the command `\showlists`. This command writes to the log file the contents of all lists that are being built at the moment the command is given.

Consider the example

```
a\hfil\break b\par
c\hfill\break d
\hbox{e\vbox{f\showlists
```

Here the first paragraph has been broken into two lines, and these have been added to the current page. The second paragraph has not been concluded or broken into lines.

The log file shows the following.  $\TeX$  was busy building a paragraph (starting with an indentation box 20pt wide):

```
### horizontal mode entered at line 3
\hbox(0.0+0.0)x20.0
\tenrm f
spacefactor 1000
```

This paragraph was inside a vertical box:

```
### internal vertical mode entered at line 3
prevdepth ignored
```

The vertical box was in a horizontal box,

```
### restricted horizontal mode entered at line 3
\tenrm e
spacefactor 1000
```

which was part of an as-yet unfinished paragraph:

```

### horizontal mode entered at line 2
\hbox(0.0+0.0)x20.0
\tenrm c
\glue 0.0 plus 1.0fil
\penalty -10000
\tenrm d
etc.
spacefactor 1000

```

Note how the infinite glue and the `\break` penalty are still part of the horizontal list.

Finally, the first paragraph has been broken into lines and added to the current page:

```

### vertical mode entered at line 0
### current page:
\glue(\topskip) 5.69446
\hbox(4.30554+0.0)x469.75499, glue set 444.75497fil
.\hbox(0.0+0.0)x20.0
.\tenrm a
.\glue 0.0 plus 1.0fil
.\penalty -10000
.\glue(\rightskip) 0.0
\penalty 300
\glue(\baselineskip) 5.05556
\hbox(6.94444+0.0)x469.75499, glue set 464.19943fil
.\tenrm b
.\penalty 10000
.\glue(\parfillskip) 0.0 plus 1.0fil
.\glue(\rightskip) 0.0
etc.
total height 22.0 plus 1.0
goal height 643.20255
prevdepth 0.0

```

## 第 7 章 Numbers

In this chapter integers and their denotations will be treated, the conversions that are possible either way, allocation and use of `\count` registers, and arithmetic with integers.

`\number` Convert a `<number>` to decimal representation.

`\romannumeral` Convert a positive `<number>` to lowercase roman representation.

`\ifnum` Test relations between numbers.

`\ifodd` Test whether a number is odd.

`\ifcase` Enumerated case statement.

`\count` Prefix for count registers.

`\countdef` Define a control sequence to be a synonym for a `\count` register.

`\newcount` Allocate an unused `\count` register.

`\advance` Arithmetic command to add to or subtract from a `<numeric variable>`.

`\multiply` Arithmetic command to multiply a `<numeric variable>`.

`\divide` Arithmetic command to divide a `<numeric variable>`.

### 7.1 Numbers and `<number>`s

An important part of the grammar of `TEX` is the rigorous definition of a `<number>`, the syntactic entity that `TEX` expects when semantically an *integer* is expected. This definition will take the largest part of this chapter. Towards the end, `\count` registers, arithmetic, and tests for numbers are discussed.

For clarity of discussion a distinction will be made here between integers and numbers, but note that a `<number>` can be both an ‘integer’ and a ‘number’. ‘Integer’ will be taken to denote a mathematical number: a quantity that can

be added or multiplied. ‘Number’ will be taken to refer to the printed representation of an integer: a string of digits, in other words.

## 7.2 Integers

Quite a few different sorts of objects can function as integers in  $\text{\TeX}$ . In this section they will all be treated, accompanied by the relevant lines from the grammar of  $\text{\TeX}$ .

First of all, an integer can be positive or negative:

```
⟨number⟩ → ⟨optional signs⟩⟨unsigned number⟩
⟨optional signs⟩ → ⟨optional spaces⟩
| ⟨optional signs⟩⟨plus or minus⟩⟨optional spaces⟩
```

A first possibility for an unsigned integer is a string of digits in decimal, octal, or hexadecimal notation. Together with the alphabetic constants these will be named here  $\langle\text{integer denotation}\rangle$ . Another possibility for an integer is an internal integer quantity, an  $\langle\text{internal integer}\rangle$ ; together with the denotations these form the  $\langle\text{normal integer}\rangle$ s. Lastly an integer can be a  $\langle\text{coerced integer}\rangle$ : an internal  $\langle\text{dimen}\rangle$  or  $\langle\text{glue}\rangle$  quantity that is converted to an integer value.

```
⟨unsigned number⟩ → ⟨normal integer⟩ | ⟨coerced integer⟩
⟨normal integer⟩ → ⟨integer denotation⟩ | ⟨internal integer⟩
⟨coerced integer⟩ → ⟨internal dimen⟩ | ⟨internal glue⟩
```

All of these possibilities will be treated in sequence.

### 7.2.1 Denotations: integers

Anything that looks like a number can be used as a  $\langle\text{number}\rangle$ : thus 42 is a number. However, bases other than decimal can also be used:

```
'123
```

is the octal notation for  $1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0 = 83$ , and

```
"123
```

is the hexadecimal notation for  $1 \times 16^2 + 2 \times 16^1 + 3 \times 16^0 = 291$ .

```
⟨integer denotation⟩ → ⟨integer constant⟩⟨one optional space⟩
| '⟨octal constant⟩⟨one optional space⟩
| "⟨hexadecimal constant⟩⟨one optional space⟩
```

The octal digits are 0–7; a digit 8 or 9 following an octal denotation is not part of the number: after

```
\count0='078
```

the `\count0` will have the value 7, and the digit 8 is typeset.

The hexadecimal digits are 0–9, A–F, where the A–F can have category code 11 or 12. The latter has a somewhat far-fetched justification: the characters resulting from a `\string` operation have category code 12. Lowercase a–f are not hexadecimal digits, although (in  $\text{\TeX}$ 3) they are used for hexadecimal notation in the ‘circumflex method’ for accessing all character codes (see Chapter 3).

### 7.2.2 Denotations: characters

A character token is a pair consisting of a character code, which is a number in the range 0–255, and a category code. Both of these codes are accessible, and can be used as a `<number>`.

The character code of a character token, or of a single letter control sequence, is accessible through the left quote command: both ``a` and ``\a` denote the character code of a, which can be used as an integer.

`<integer denotation>  $\longrightarrow$  `<character token><one optional space>`

In order to emphasize that accessing the character code is in a sense using a denotation, the syntax of  $\text{\TeX}$  allows an optional space after such a ‘character constant’. The left quote must have category 12.

### 7.2.3 Internal integers

The class of `<internal integers>` can be split into five parts. The `<codename>`s and `<special integer>`s will be treated separately below; furthermore, there are the following.

- The contents of `\count` registers; either explicitly used by writing for instance `\count23`, or by referring to such a register by means of a control sequence that was defined by `\countdef`: after

```
\countdef\MyCount=23
```

`\MyCount` is called a `<countdef token>`, and it is fully equivalent to `\count23`.

- All parameters of  $\text{\TeX}$  that hold integer values; this includes obvious ones such as `\linepenalty`, but also parameters such as `\hyphenchar<font>` and `\parshape` (if a paragraph shape has been defined for  $n$  lines, using `\parshape` in the context of a `<number>` will yield this value of  $n$ ).
- Tokens defined by `\chardef` or `\mathchardef`. After

```
\chardef\foo=74
```

the control sequence `\foo` can be used on its own to mean `\char74`, but in a context where a `<number>` is wanted it can be used to denote 74:

`\count\foo`

is equivalent to `\count74`. This fact is exploited in the allocation routines for registers (see Chapter 31).

A control sequence thus defined by `\chardef` is called a `<chardef token>`; if it is defined by `\mathchardef` it is called a `<mathchardef token>`.

Here is the full list:

```

<internal integer> → <integer parameter>
| <special integer> | \lastpenalty
| <countdef token> | \count<8-bit number>
| <chardef token> | <mathchardef token>
| <codename><8-bit number>
| \hyphenchar<font> | \skewchar<font> | \parshape
| \inputlineno | \badness
<integer parameter> → | \adjdemerits | \binoppenalty
| \brokenpenalty | \clubpenalty | \day
| \defaultthyphenchar | \defaultskewchar
| \delimiterfactor | \displaywidowpenalty
| \doublehyphendemerits | \endlinechar | \escapechar
| \exhyphenpenalty | \fam | \finalhyphendemerits
| \floatingpenalty | \globaldefs | \hangafter
| \hbadness | \hyphenpenalty | \interlinepenalty
| \linepenalty | \looseness | \mag
| \maxdeadcycles | \month
| \newlinechar | \outputpenalty | \pausing
| \postdisplaypenalty | \predisplaypenalty
| \pretolerance | \relpenalty | \showboxbreadth
| \showboxdepth | \time | \tolerance
| \tracingcommands | \tracinglostchars | \tracingmacros
| \tracingonline | \tracingoutput | \tracingpages
| \tracingparagraphs | \tracingrestores | \tracingstats
| \uchyph | \vbadness | \widowpenalty | \year

```

Any internal integer can function as an `<internal unit>`, which – preceded by `<optional spaces>` – can serve as a `<unit of measure>`. Examples of this are given in Chapter 8.

### 7.2.4 Internal integers: other codes of a character

The `\catcode` command (which was described in Chapter 2) is a `<codename>`, and like the other code names it can be used as an integer.

```
<codename> → \catcode | \mathcode | \uccode | \lccode
          | \sfcode | \delcode
```

A `<codename>` has to be followed by an `<8-bit number>`.

Uppercase and lowercase codes were treated in Chapter 3; the `\sfcode` is treated in Chapter 20; the `\mathcode` and `\delcode` are treated in Chapter 21.

### 7.2.5 `<special integer>`

One of the subclasses of the internal integers is that of the special integers.

```
<special integer> → \spacefactor | \prevgraf
                  | \deadcycles | \insertpenalties
```

An assignment to any of these is called an `<intimate assignment>`, and is automatically global (see Chapter 10).

### 7.2.6 Other internal quantities: coercion to integer

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$  provides a conversion between dimensions and integers: if an integer is expected, a `<dimen>` or `<glue>` used in that context is converted by taking its (natural) size in scaled points. However, only `<internal dimen>`s and `<internal glue>` can be used this way: no dimension or glue denotations can be coerced to integers.

### 7.2.7 Trailing spaces

The syntax of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  defines integer denotations (decimal, octal, and hexadecimal) and ‘back-quoted’ character tokens to be followed by `<one optional space>`. This means that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  reads the token after the number, absorbing it if it was a space token, and backing up if it was not.

Because  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ ’s input processor goes into the state ‘skipping spaces’ after it has seen one space token, this scanning behaviour implies that integer denotations can be followed by arbitrarily many space characters in the input. Also, a line end is admissible. However, only one space token is allowed.

## 7.3 Numbers

$\text{\TeX}$  can perform an implicit *number conversion* from a string of digits to an integer. Conversion from a representation in decimal, octal, or hexadecimal notation was treated above. The conversion the other way, from an  $\langle$ internal integer $\rangle$  to a printed representation, has to be performed explicitly.  $\text{\TeX}$  provides two conversion routines, `\number`, to decimal, and `\romannumeral` to *roman numerals*. The command `\number` is equivalent to `\the` when followed by an internal integer. These commands are performed in the expansion processor of  $\text{\TeX}$ , that is, they are expanded whenever expansion has not been inhibited.

Both commands yield a string of tokens with category code 12; their argument is a  $\langle$ number $\rangle$ . Thus `\romannumeral51`, `\romannumeral\year`, and `\number\linepenalty` are valid, and so is `\number13`. Applying `\number` to a denotation has some uses: it removes leading zeros and superfluous plus and minus signs.

A roman numeral is a string of lowercase ‘roman digits’, which are characters of category code 12. The sequence

```
\uppercase\expandafter{\romannumeral ...}
```

gives uppercase roman numerals. This works because  $\text{\TeX}$  expands tokens in order to find the opening brace of the argument of `\uppercase`. If `\romannumeral` is applied to a negative number, the result is simply empty.

## 7.4 Integer registers

Integers can be stored in `\count` registers:

```
\count<8-bit number>
```

is an  $\langle$ integer variable $\rangle$  and an  $\langle$ internal integer $\rangle$ . As an integer variable it can be used in a  $\langle$ variable assignment $\rangle$ :

```
<variable assignment> \rightarrow <integer variable> \langle equals \rangle <number> | ...
```

As an internal integer it can be used as a  $\langle$ number $\rangle$ :

```
<number> \rightarrow <optional signs> \langle internal integer \rangle | ...
```

Synonyms for `\count` registers can be introduced by the `\countdef` command in a  $\langle$ shorthand definition $\rangle$ :

```
\countdef \langle control sequence \rangle \langle equals \rangle <8-bit number>
```

A control sequence defined this way is called a  $\langle$ countdef token $\rangle$ , and it serves as an  $\langle$ internal integer $\rangle$ .

The plain  $\text{\TeX}$  macro `\newcount` (which is declared `\outer`) uses the `\countdef` command to allocate an unused `\count` register. Counters 0–9 are scratch reg-



isters, like all registers with numbers 0–9. However, counters 0–9 are used for page identification in the `dvi` file (see Chapter 33), so they should be used as scratch registers only inside a group. Counters 10–22 are used for plain  $\text{\TeX}$ 's bookkeeping of allocation of registers. Counter 255 is also scratch.

## 7.5 Arithmetic

The user can perform some *arithmetic* in  $\text{\TeX}$ , and  $\text{\TeX}$  also performs arithmetic internally. User arithmetic is concerned only with integers; the internal arithmetic is mostly on fixed-point quantities, and only in the case of glue setting on floating-point numbers.

### 7.5.1 Arithmetic statements

$\text{\TeX}$  allows the user to perform some arithmetic on integers. The statement

```
\advance<integer variable><optional by><number>
```

adds the value of the `<number>` – which may be negative – to the `<integer variable>`. Similarly,

```
\multiply<integer variable><optional by><number>
```

multiplies the value of the `<integer variable>`, and

```
\divide<integer variable><optional by><number>
```

divides an `<integer variable>`.

Multiplication and division are also available for any so-called `<numeric variable>`: their most general form is

```
\multiply<numeric variable><optional by><number>
```

where

```
<numeric variable> → <integer variable> | <dimen variable>
| <glue variable> | <muglue variable>
```

The result of an arithmetic operation should not exceed  $2^{30}$  in absolute value.

Division of integers yields an integer; that is, the remainder is discarded. This raises the question of how rounding is performed when either operand is negative. In such cases  $\text{\TeX}$  performs the division with the absolute values of the operands, and takes the negative of the result if exactly one operand was negative.

### 7.5.2 Floating-point arithmetic

Internally some *floating-point arithmetic* is performed, namely in the calculation of glue set ratios. However, machine-dependent aspects of rounding cannot influence the decision process of  $\text{\TeX}$ , so machine independence of  $\text{\TeX}$  is guaranteed in this respect (sufficient accuracy of rounding is enforced by the Trip test of [21]).

### 7.5.3 Fixed-point arithmetic

All fractional arithmetic in  $\text{\TeX}$  is performed in *fixed-point arithmetic* of ‘scaled integers’: multiples of  $2^{-16}$ . This ensures the machine independence of  $\text{\TeX}$ . Printed representations of scaled integers are rounded to 5 decimal digits.

In ordinary 32-bit implementations of  $\text{\TeX}$  the largest integers are  $2^{31} - 1$  in absolute size. The user is not allowed to specify dimensions larger in absolute size than  $2^{30} - 1$ : two such dimensions can be added or subtracted without overflow on a 32-bit system.

## 7.6 Number testing

The most general test for integers in  $\text{\TeX}$  is

```
\ifnum⟨number1⟩⟨relation⟩⟨number2⟩
```

where  $\langle\text{relation}\rangle$  is a  $<$ ,  $>$ , or  $=$  character, all of category 12.

Distinguishing between odd and even numbers is done by

```
\ifodd⟨number⟩
```

A numeric case statement is provided by

```
\ifcase⟨number⟩⟨case0⟩\or... \or⟨casen⟩\else⟨other cases⟩\fi
```

where the  $\backslash\text{else}$ -part is optional. The tokens for  $\langle\text{case}_i\rangle$  are processed if the number turns out to be  $i$ ; other cases are skipped, similarly to what ordinarily happens in conditionals (see Chapter 13).

## 7.7 Remarks

### 7.7.1 Character constants

In formats and macro collections numeric constants are often needed. There are several ways to implement these in  $\text{\TeX}$ .

Firstly,

```
\newcount\SomeConstant \SomeConstant=42
```

This is wasteful, as it uses up a `\count` register.

Secondly,

```
\def\SomeConstant{42}
```

Better but accident prone:  $\TeX$  has to expand to find the number – which in itself is a slight overhead – and may inadvertently expand some tokens that should have been left alone.

Thirdly,

```
\chardef\SomeConstant=42
```

This one is fine. A `\chardef` token has the same status as a `\count` register: both are `\internal integer`s. Therefore a number defined this way can be used everywhere that a `\count` register is feasible. For large numbers the `\chardef` can be replaced by `\mathchardef`, which runs to  $7FFF = 32767$ . Note that a `\mathchardef` token can usually only appear in math mode, but in the context of a number it can appear anywhere.

### 7.7.2 Expanding too far / how far

It is a common mistake to write pieces of  $\TeX$  code where  $\TeX$  will inadvertently expand something because it is trying to compose a number. For example:

```
\def\par{\endgraf\penalty200}
... \par \number\pageno
```

Here the page number will be absorbed into the value of the penalty.

Now consider

```
\newcount\midpenalty \midpenalty=200
\def\par{\endgraf\penalty\midpenalty}
... \par \number\pageno
```

Here the page number is not scooped up by mistake:  $\TeX$  is trying to locate a `\number` after the `\penalty`, and it finds a `\countdef` token. This is *not* converted to a representation in digits, so there is never any danger of the page number being touched.

It is possible to convert a `\countdef` token first to a representation in digits before assigning it:

```
\penalty\number\midpenalty
```

and this brings back again all previous problems of expansion.

## 第 8 章    Dimensions and Glue

In T<sub>E</sub>X vertical and horizontal white space can have a possibility to adjust itself through ‘stretching’ or ‘shrinking’. An adjustable white space is called *glue*. This chapter treats all technical concepts related to dimensions and glue, and it explains how the badness of stretching or shrinking a certain amount is calculated.

`\dimen` Dimension register prefix.

`\dimendef` Define a control sequence to be a synonym for a `\dimen` register.

`\newdimen` Allocate an unused `\dimen` register.

`\skip` Skip register prefix.

`\skipdef` Define a control sequence to be a synonym for a `\skip` register.

`\newskip` Allocate an unused `\skip` register.

`\ifdim` Compare two dimensions.

`\hskip` Insert in horizontal mode a glue item.

`\hfil` Equivalent to `\hskip 0cm plus 1fil`.

`\hfilneg` Equivalent to `\hskip 0cm minus 1fil`.

`\hfill` Equivalent to `\hskip 0cm plus 1fill`.

`\hss` Equivalent to `\hskip 0cm plus 1fil minus 1fil`.

`\vskip` Insert in vertical mode a glue item.

`\vfil` Equivalent to `\vskip 0cm plus 1fil`.

`\vfill` Equivalent to `\vskip 0cm plus 1fill`.

`\vfilneg` Equivalent to `\vskip 0cm minus 1fil`.

`\vss` Equivalent to `\vskip 0cm plus 1fil minus 1fil`.

`\kern` Add a kern item to the current horizontal or vertical list.

`\lastkern` If the last item on the current list was a kern, the size of it.

`\lastskip` If the last item on the current list was a glue, the size of it.

$\backslash\text{unkern}$  If the last item of the current list was a kern, remove it.

$\backslash\text{unskip}$  If the last item of the current list was a glue, remove it.

$\backslash\text{removelastskip}$  Macro to append the negative of the  $\backslash\text{lastskip}$ .

$\backslash\text{advance}$  Arithmetic command to add to or subtract from a  $\langle\text{numeric variable}\rangle$ .

$\backslash\text{multiply}$  Arithmetic command to multiply a  $\langle\text{numeric variable}\rangle$ .

$\backslash\text{divide}$  Arithmetic command to divide a  $\langle\text{numeric variable}\rangle$ .

## 8.1 Definition of $\langle\text{glue}\rangle$ and $\langle\text{dimen}\rangle$

This section gives the syntax of the quantities  $\langle\text{dimen}\rangle$  and  $\langle\text{glue}\rangle$ . In the next section the practical aspects of glue are treated.

Unfortunately the terminology for glue is slightly confusing. The syntactical quantity  $\langle\text{glue}\rangle$  is a dimension (a distance) with possibly a stretch and/or shrink component. In order to add a glob of ‘glue’ (a white space) to a list one has to let a  $\langle\text{glue}\rangle$  be preceded by commands such as  $\backslash\text{vskip}$ .

### 8.1.1 Definition of dimensions

A  $\langle\text{dimen}\rangle$  is what  $\text{T}_{\text{E}}\text{X}$  expects to see when it needs to indicate a dimension; it can be positive or negative.

$$\langle\text{dimen}\rangle \longrightarrow \langle\text{optional signs}\rangle\langle\text{unsigned dimen}\rangle$$

The unsigned part of a  $\langle\text{dimen}\rangle$  can be

$$\begin{aligned} \langle\text{unsigned dimen}\rangle &\longrightarrow \langle\text{normal dimen}\rangle \mid \langle\text{coerced dimen}\rangle \\ \langle\text{normal dimen}\rangle &\longrightarrow \langle\text{internal dimen}\rangle \mid \langle\text{factor}\rangle\langle\text{unit of measure}\rangle \\ \langle\text{coerced dimen}\rangle &\longrightarrow \langle\text{internal glue}\rangle \end{aligned}$$

That is, we have the following three cases:

- an  $\langle\text{internal dimen}\rangle$ ; this is any register or parameter of  $\text{T}_{\text{E}}\text{X}$  that has a  $\langle\text{dimen}\rangle$  value:

$$\begin{aligned} \langle\text{internal dimen}\rangle &\longrightarrow \langle\text{dimen parameter}\rangle \\ &\mid \langle\text{special dimen}\rangle \mid \backslash\text{lastkern} \\ &\mid \langle\text{dimendef token}\rangle \mid \backslash\text{dimen}\langle\text{8-bit number}\rangle \\ &\mid \backslash\text{fontdimen}\langle\text{number}\rangle\langle\text{font}\rangle \\ &\mid \langle\text{box dimension}\rangle\langle\text{8-bit number}\rangle \\ \langle\text{dimen parameter}\rangle &\longrightarrow \backslash\text{boxmaxdepth} \\ &\mid \backslash\text{delimitershortfall} \mid \backslash\text{displayindent} \end{aligned}$$

```

| \displaywidth | \hangindent
| \hfuzz | \hoffset | \hsize
| \lineskiplimit | \mathsurround
| \maxdepth | \nulldelimiterspace
| \overfullrule | \parindent
| \predisplaysize | \scriptspace
| \splitmaxdepth | \vfuzz
| \voffset | \vsize

```

- a dimension denotation, consisting of  $\langle \text{factor} \rangle \langle \text{unit of measure} \rangle$ , for example `0.7\size`; or
- an  $\langle \text{internal glue} \rangle$  (see below) coerced to a dimension by omitting the stretch and shrink components, for example `\parfillskip`.

A dimension denotation is a somewhat complicated entity:

- a  $\langle \text{factor} \rangle$  is an integer denotation, a decimal constant denotation (a number with an integral and a fractional part), or an  $\langle \text{internal integer} \rangle$

$\langle \text{factor} \rangle \longrightarrow \langle \text{normal integer} \rangle \mid \langle \text{decimal constant} \rangle$

$\langle \text{normal integer} \rangle \longrightarrow \langle \text{integer denotation} \rangle$

$\mid \langle \text{internal integer} \rangle$

$\langle \text{decimal constant} \rangle \longrightarrow ._{12} \mid ,_{12}$

$\mid \langle \text{digit} \rangle \langle \text{decimal constant} \rangle$

$\mid \langle \text{decimal constant} \rangle \langle \text{digit} \rangle$

An internal integer is a parameter that is ‘really’ an integer (for instance, `\count0`), and not coerced from a dimension or glue. See Chapter 7 for the definition of various kinds of integers.

- a  $\langle \text{unit of measure} \rangle$  can be a  $\langle \text{physical unit} \rangle$ , that is, an ordinary unit such as `cm` (possibly preceded by `true`), an internal unit such as `em`, but also an  $\langle \text{internal integer} \rangle$  (by conversion to scaled points), an  $\langle \text{internal dimen} \rangle$ , or an  $\langle \text{internal glue} \rangle$ .

$\langle \text{unit of measure} \rangle \longrightarrow \langle \text{optional spaces} \rangle \langle \text{internal unit} \rangle$

$\mid \langle \text{optional true} \rangle \langle \text{physical unit} \rangle \langle \text{one optional space} \rangle$

$\langle \text{internal unit} \rangle \longrightarrow \text{em} \langle \text{one optional space} \rangle$

$\mid \text{ex} \langle \text{one optional space} \rangle \mid \langle \text{internal integer} \rangle$

$\mid \langle \text{internal dimen} \rangle \mid \langle \text{internal glue} \rangle$

Some  $\langle \text{dimen} \rangle$ s are called  $\langle \text{special dimen} \rangle$ s:

$\langle \text{special dimen} \rangle \longrightarrow \text{\prevdepth}$

$\mid \text{\pagegoal} \mid \text{\pagetotal} \mid \text{\pagestretch}$

$\mid \text{\pagefilstretch} \mid \text{\pagefillstretch}$

|  $\backslash\text{pagefilllstretch}$  |  $\backslash\text{pageshrink}$  |  $\backslash\text{pagedepth}$

An assignment to any of these is called an  $\langle\text{intimate assignment}\rangle$ , and it is automatically global (see Chapter 10). The meaning of these dimensions is explained in Chapter 27, with the exception of  $\backslash\text{prevdepth}$  which is treated in Chapter 15.

### 8.1.2 Definition of glue

A  $\langle\text{glue}\rangle$  is either some form of glue variable, or a glue denotation with explicitly indicated stretch and shrink. Specifically,

$\langle\text{glue}\rangle \longrightarrow \langle\text{optional signs}\rangle\langle\text{internal glue}\rangle$  |  $\langle\text{dimen}\rangle\langle\text{stretch}\rangle\langle\text{shrink}\rangle$   
 $\langle\text{internal glue}\rangle \longrightarrow \langle\text{glue parameter}\rangle$  |  $\backslash\text{lastskip}$   
 |  $\langle\text{skipdef token}\rangle$  |  $\backslash\text{skip}\langle\text{8-bit number}\rangle$   
 $\langle\text{glue parameter}\rangle \longrightarrow \backslash\text{abovedisplayshortskip}$   
 |  $\backslash\text{abovedisplayskip}$  |  $\backslash\text{baselineskip}$   
 |  $\backslash\text{belowdisplayshortskip}$  |  $\backslash\text{belowdisplayskip}$   
 |  $\backslash\text{leftskip}$  |  $\backslash\text{lineskip}$  |  $\backslash\text{parfillskip}$  |  $\backslash\text{parskip}$   
 |  $\backslash\text{rightskip}$  |  $\backslash\text{spaceskip}$  |  $\backslash\text{splittopskip}$  |  $\backslash\text{tabskip}$   
 |  $\backslash\text{topskip}$  |  $\backslash\text{xspaceskip}$

The stretch and shrink components in a glue denotation are optional, but when both are specified they have to be given in sequence; they are defined as

$\langle\text{stretch}\rangle \longrightarrow \text{plus } \langle\text{dimen}\rangle$  |  $\text{plus } \langle\text{fil dimen}\rangle$  |  $\langle\text{optional spaces}\rangle$   
 $\langle\text{shrink}\rangle \longrightarrow \text{minus } \langle\text{dimen}\rangle$  |  $\text{minus } \langle\text{fil dimen}\rangle$  |  $\langle\text{optional spaces}\rangle$   
 $\langle\text{fil dimen}\rangle \longrightarrow \langle\text{optional signs}\rangle\langle\text{factor}\rangle\langle\text{fil unit}\rangle\langle\text{optional spaces}\rangle$   
 $\langle\text{fil unit}\rangle \longrightarrow$  |  $\text{fil}$  |  $\text{fill}$  |  $\text{filll}$

The actual definition of  $\langle\text{fil unit}\rangle$  is recursive (see Chapter 36), but these are the only valid possibilities.

### 8.1.3 Conversion of $\langle\text{glue}\rangle$ to $\langle\text{dimen}\rangle$

The grammar rule

$\langle\text{dimen}\rangle \longrightarrow \langle\text{factor}\rangle\langle\text{unit of measure}\rangle$

has some noteworthy consequences, caused by the fact that a  $\langle\text{unit of measure}\rangle$  need not look like a ‘unit of measure’ at all (see the list above).

For instance, from this definition we conclude that the statement

$\backslash\text{dimen0}=\backslash\text{lastpenalty}\backslash\text{lastpenalty}$

is syntactically correct because  $\backslash\text{lastpenalty}$  can function both as an integer and as  $\langle\text{unit of measure}\rangle$  by taking its value in scaled points. After  $\backslash\text{penalty8}$

the `\dimen0` thus defined will have a size of 64sp.

More importantly, consider the case where the `<unit of measure>` is an `<internal glue>`, that is, any sort of glue parameter. Prefixing such a glue with a number (the `<factor>`) makes it a valid `<dimen>` specification. Thus

```
\skip0=\skip1
```

is very different from

```
\skip0=1\skip1
```

The first statement makes `\skip0` equal to `\skip1`, the second converts the `\skip1` to a `<dimen>` before assigning it. In other words, the `\skip0` defined by the second statement has no stretch or shrink.

#### 8.1.4 Registers for `\dimen` and `\skip`

$\mathrm{T}_{\mathrm{E}}\mathrm{X}$  has registers for storing `<dimen>` and `<glue>` values: the `\dimen` and `\skip` registers respectively. These are accessible by the expressions

```
\dimen<number>
```

and

```
\skip<number>
```

As with all registers of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ , these registers are numbered 0–255.

Synonyms for registers can be made with the `\dimendef` and `\skipdef` commands. Their syntax is

```
\dimendef<control sequence><equals><8-bit number>
```

and

```
\skipdef<control sequence><equals><8-bit number>
```

For example, after `\skipdef\foo=13` using `\foo` is equivalent to using `\skip13`.

Macros `\newdimen` and `\newskip` exist in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  for allocating an unused `dimen` or `skip` register. These macros are defined to be `\outer` in the plain format.

#### 8.1.5 Arithmetic: addition

As for integer variables, arithmetic operations exist for *arithmetic on glue*: `dimen`, `glue`, and `muglue` (mathematical glue; see page 221) variables.

The expressions

```
\advance<dimen variable><optional by><dimen>
```

```
\advance<glue variable><optional by><glue>
```

```
\advance<muglue variable><optional by><muglue>
```



add to the size of a `dimen`, `glue`, or `muglue`.

Advancing a `<glue variable>` by `<glue>` is done by adding the natural sizes, and the stretch and shrink components. Because  $\text{\TeX}$  converts between `<glue>` and `<dimen>`, it is possible to write for instance

```
\advance\skip1 by \dimen1
```

or

```
\advance\dimen1 by \skip1
```

In the first case `\dimen1` is coerced to `<glue>` without stretch or shrink; in the second case the `\skip1` is coerced to a `<dimen>` by taking its natural size.

### 8.1.6 Arithmetic: multiplication and division

Multiplication and division operations exist for glue and dimensions. One may for instance write

```
\multiply\skip1 by 2
```

which multiplies the natural size, and the stretch and shrink components of `\skip1` by 2.

The second operand of a `\multiply` or `\divide` operation can only be a `<number>`, that is, an integer. Introducing the notion of `<numeric variable>`:

```
<numeric variable> → <integer variable> | <dimen variable>
| <glue variable> | <muglue variable>
```

these operations take the form

```
\multiply<numeric variable>[optional by]<number>
```

and

```
\divide<numeric variable>[optional by]<number>
```

Glue and `dimen` can be multiplied by non-integer quantities:

```
\skip1=2.5\skip2
\dimen1=.78\dimen2
```

However, in the first line the `\skip2` is first coerced to a `<dimen>` value by omitting its stretch and shrink.

## 8.2 More about dimensions

### 8.2.1 Units of measurement

In  $\text{\TeX}$  dimensions can be indicated in the following *units of measurement*:  
**centimetre** denoted `cm` or

**millimetre** denoted mm; these are SI units (*Système International d'Unités*, the international system of standard units of measurements).

**inch** in; more common in the Anglo-American world. One inch is 2.54 centimetres.

**pica** denoted pc; one pica is 12 points.

**point** denoted pt; the common system for Anglo-American printers. One inch is 72.27 points.

**didot point** denoted dd; the common system for continental European printers. Furthermore, 1157 didot points are 1238 points.

**cicero** denoted cc; one cicero is 12 didot points.

**big point** denoted bp; one inch is 72 big points.

**scaled point** denoted sp; this is the smallest unit in  $\text{\TeX}$ , and all measurements are integral multiples of one scaled point. There are 65 536 scaled points in a point.

Decimal fractions can be written using both the Anglo-American system with the decimal point (for example,  $1\text{in}=72.27\text{pt}$ ) and the continental European system with a decimal comma;  $1\text{in}=72,27\text{pt}$ .

Internally  $\text{\TeX}$  works with multiples of a smallest dimension: the scaled point. Dimensions larger (in absolute value) than  $2^{30} - 1\text{sp}$ , which is about 5.75 metres or 18.9 feet, are illegal.

Both the pica system and the didot system are of French origin: in 1737 the type founder Pierre Simon Fournier introduced typographical points based on the French foot. Although at first he introduced a system based on lines and points, he later took the point as unit: there are 72 points in an inch, which is one-twelfth of a foot. About 1770 another founder, François Ambroise Didot, introduced points based on the more common, and slightly longer, ‘pied du roi’.

## 8.2.2 Dimension testing

Dimensions and natural sizes of glue can be compared with the `\ifdim` test. This takes the form

```
\ifdim⟨dimen1⟩⟨relation⟩⟨dimen2⟩
```

where the relation can be an `>`, `<`, or `=` token, all of category 12.

## 8.2.3 Defined dimensions

```
\z@ 0pt
```

`\maxdimen 16383.99999pt`; the largest legal dimension.

These  $\langle \text{dimen} \rangle$ s are predefined in the plain format; for instance

```
\newdimen\z@ \z@=0pt
```

Using such abbreviations for commonly used dimensions has at least two advantages. First of all it saves main memory if such a dimension occurs in a macro: a control sequence is one token, whereas a string such as `0pt` takes three. Secondly, it saves time in processing, as  $\text{\TeX}$  does not need to perform conversions to arrive at the correct type of object.

Control sequences such as `\z@` are only available to a user who changes the category code of the ‘at’ sign. Ordinarily, these control sequences appear only in the macros defined in packages such as the plain format.

## 8.3 More about glue

Glue items can be added to a vertical list with one of the commands `\vskip<glue>`, `\vfil`, `\vfill`, `\vss` or `\vfildneg`; glue items can be added to a horizontal list with one of the commands `\hskip<glue>`, `\hfil`, `\hfill`, `\hss` or `\hfildneg`. We will now treat the properties of glue.

### 8.3.1 Stretch and shrink

In the syntax given above,  $\langle \text{glue} \rangle$  was defined as having

- a ‘natural size’, which is a  $\langle \text{dimen} \rangle$ , and optionally
- a *stretch* and *shrink* components *stretchshrink* built out of a  $\langle \text{fil dimen} \rangle$ .

Each list that  $\text{\TeX}$  builds has amounts of stretch and shrink (possibly zero), which are the sum of the stretch and shrink components of individual pieces of glue in the list. Stretch and shrink are used if the context in which the list appears requires it to assume a size that is different from its natural size.

There is an important difference in behaviour between stretch and shrink components when they are finite – that is, when the  $\langle \text{fil dimen} \rangle$  is not `fil(1(1))`. A finite amount of shrink is indeed the maximum shrink that  $\text{\TeX}$  will take: the amount of glue specified as

```
5pt minus 3pt
```

can shrink to 2pt, but not further. In contrast to this, a finite amount of stretch can be stretched arbitrarily far. Such arbitrary stretching has a large ‘badness’, however. Badness calculation is treated below.

例子: *The sequence with natural size 20pt*

```
\hskip 10pt plus 2pt \hskip 10pt plus 3pt
```

*has 5pt of stretch, but it has no shrink. In*

```
\hskip 10pt minus 2pt \hskip 10pt plus 3pt
```

*there is 3pt of stretch, and 2pt of shrink, so its minimal size is 18pt.*

*Positive shrink is not the same as negative stretch:*

```
\hskip 10pt plus -2pt \hskip 10pt plus 3pt
```

*looks a lot like the previous example, but it cannot be shrunk as there are no `minus⟨dimen⟩` specifications. It does have 1pt of stretch, however.*

*This is another example of negative amounts of shrink and stretch. It is not possible to stretch glue (in the informal sense) by shrinking it (in the technical sense):*

```
\hbox to 5cm{a\hskip 0cm minus -1fil}
```

*is an underfull box, because  $\text{\TeX}$  looks for a `plus⟨dimen⟩` specification when it needs to stretch the contents.*

*Finally,*

```
\hskip 10pt plus -3pt \hskip 10pt plus 3pt
```

*can neither stretch nor shrink. The fact that there is only stretch available means that the sequence cannot shrink. However, the stretch components cancel out: the total stretch is zero. Another way of looking at this is to consider that for each point that the second glue item would stretch, the first one would ‘stretch back’ one point.*

Any amount of infinite stretch or shrink overpowers all finite stretch or shrink available:

```
\hbox to 5cm{\hskip 0cm plus 16384pt
               text\hskip 0cm plus 0.0001fil}
```

has the text at the extreme left of the box. There are three orders of ‘infinity’, each one infinitely stronger than the previous one:

```
\hbox to 5cm{\hskip 0cm plus 16384fil
               text\hskip 0cm plus 0.0001fill}
```

and

```
\hbox to 5cm{\hskip 0cm plus 16384fill
               text\hskip 0cm plus 0.0001fillll}
```

both have the text at the left end of the box.

### 8.3.2 Glue setting

In the process of *glue setting*, the desired width (or height) of a box is compared with the natural dimension of its contents, which is the sum of all natural dimensions of boxes and globs of glue. If the two differ, any available stretchability or shrinkability is used to bridge the gap. To attain the desired dimension of the box only the glue of the highest available order is set: each piece of glue of that order is stretched or shrunk by the same ratio.

For example, in

```
\hbox to 6pt{\hskip 0pt plus 3pt \hskip 0pt plus 9pt}
```

the natural size of the box is 0pt, and the total stretch is 12pt. In order to obtain a box of 6pt each glue item is set with a stretch ratio of 1/2. Thus the result is equivalent to

```
\hbox {\hskip 1.5pt \hskip 4.5pt}
```

Only the highest order of stretch or shrink is used: in

```
\hbox to 6pt{\hskip 0pt plus 1fil \hskip 0pt plus 9pt}
```

the second glue will assume its natural size of 0pt, and only the first glue will be stretched.

$\text{\TeX}$  will never exceed the maximum value of a finite amount of shrink. A box that cannot be shrunk enough is called ‘overfull’. Finite stretchability can be exceeded to provide an escape in difficult situations; however,  $\text{\TeX}$  is likely to give an Underfull `\hbox` message about this (see page 65). For an example of infinite shrink see page 65.

### 8.3.3 Badness

When stretching or shrinking a list  $\text{\TeX}$  calculates *badness* badness based on the ratio between actual stretch and the amount of stretch present in the line. See Chapter 19 for the application of badness to the paragraph algorithm.

The formula for badness of a list that is stretched (shrunk) is

$$b = \min \left( 10\,000, 100 \times \left( \frac{\text{actual amount stretched (shrunk)}}{\text{possible amount of stretch (shrink)}} \right)^3 \right)$$

In reality  $\text{\TeX}$  uses a slightly different formula that is easier to calculate, but behaves the same. Since glue setting is one of the main activities of  $\text{\TeX}$ , this must be performed as efficiently as possible.

This formula lets the badness be a reasonably small number if the glue set ratio (the fraction in the above expression) is reasonably small, but will let it grow rapidly once the ratio is more than 1. Badness is infinite if the glue would have to shrink more than the allotted amount; stretching glue beyond

its maximum is possible, so this provides an escape for very difficult lines of text or pages.

In  $\text{\TeX}$ 3, the `\badness` parameter records the badness of the most recently formed box.

### 8.3.4 Glue and breaking

$\text{\TeX}$  can break lines and pages in several kinds of places. One of these places is before a glue item. The glue is then discarded. For line breaks this is treated in Chapter 19, for page breaks see Chapter 27.

There are two macros in plain  $\text{\TeX}$ , `\hglue` and `\vglue`, that give non-disappearing glue in horizontal and vertical mode respectively. For the horizontal case this is accomplished by placing:

```
\vrule width Opt \nobreak \hskip ...
```

Because  $\text{\TeX}$  breaks at the front end of glue, this glue will always stay attached to the rule, and will therefore never disappear. The actual macro definitions are somewhat more complicated, because they take care to preserve the `\spacefactor` and the `\prevdepth`.

### 8.3.5 `\kern`

The `\kern` command specifies a kern item in whatever mode  $\text{\TeX}$  is currently in. A kern item is much like a glue item without stretch or shrink. It differs from glue in that it is in general not a legal breakpoint. Thus in

```
.. text .. \hbox{a}\kernOpt\hbox{b}
```

$\text{\TeX}$  will not break lines in between the boxes; in

```
.. text .. \hbox{a}\hskipOpt\hbox{b}
```

a line can be broken in between the boxes.

However, if a kern is followed by glue,  $\text{\TeX}$  can break at the kern (provided that it is not in math mode). In horizontal mode both the kern and the glue then disappear in the break. In vertical mode they are discarded when they are moved to the (empty) current page after the material before the break has been disposed of by the output routine (see Chapter 27).

### 8.3.6 Glue and modes

All horizontal skip commands are  $\langle\text{horizontal command}\rangle$ s and all vertical skip commands are  $\langle\text{vertical commands}\rangle$ s. This means that, for instance, an

`\hskip` command makes  $\TeX$  start a paragraph if it is given in vertical mode. The `\kern` command can be given in both modes.

### 8.3.7 The last glue item in a list: backspacing

The last glue item in a list can be measured, and it can be removed in all modes but external vertical mode. The internal variables `\lastskip` and `\lastkern` can be used to measure the last glob of glue in all modes; if the last glue was not a skip or kern respectively they give 0pt. In math mode the `\lastskip` functions as  $\langle$ internal muglue $\rangle$ , but in general it classifies as  $\langle$ internal glue $\rangle$ . The `\lastskip` and `\lastkern` are also 0pt if that was the size of the last glue or kern item on the list.

The operations `\unskip` and `\unkern` remove the last item of a list, if this is a glue or kern respectively. They have no effect in external vertical mode; in that case the best substitute is `\vskip-\lastskip` and `\kern-\lastkern`.

In the process of paragraph building  $\TeX$  itself performs an important `\unskip`: a paragraph ending with a white line will have a space token inserted by  $\TeX$ 's input processor. This is removed by an `\unskip` before the `\parfillskip` glue (see Chapter 17) is inserted.

Glue is treated by  $\TeX$  as a special case of leaders, which becomes apparent when `\unskip` is applied to leaders: they are removed.

### 8.3.8 Examples of backspacing

The plain  $\TeX$  macro `\removelastskip` is defined as

```
\ifdim\lastskip=0pt \else \vskip-\lastskip \fi
```

If the last item on the list was a glue, this macro will backspace by its value, provided its natural size was not zero. In all other cases, nothing is added to the list.

Sometimes an intelligent version of commands such as `\vskip` is necessary, in the sense that two subsequent skip commands should result only in the larger of the two glue amounts. On page 174 such a macro is used:

```
\newskip\tempskipa
\def\vspace#1{\tempskipa=#1\relax
  \ifvmode \ifdim\tempskipa<\lastskip
    \else \vskip-\lastskip \vskip\tempskipa
  \fi
  \else \vskip\tempskipa \fi}
```

First of all, this tests whether the mode is vertical; if not, the argument can safely be placed. Copying the argument into a skip register is necessary because

`\vspace{2pt plus 3pt}` would lead to problems in an `\ifdim#1<\lastskip` test.

If the surrounding mode was vertical, the argument should only be placed if it is not less than what is already there. The macro would be incorrect if the test read

```
\ifdim\tempskipa>\lastskip
  \vskip-\lastskip \vskip\tempskipa
\fi
```

In this case the sequence

```
... last word.\par \vspace{0pt plus 1fil}
```

would not place any glue, because after the `\par` we are in vertical mode and `\lastskip` has a value of 0pt.

### 8.3.9 Glue in trace output

If the workings of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  are traced by setting `\tracingoutput` positive, or if  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  writes a box to the log file (because of a `\showbox` command, or because it is overfull or underfull), glue is denoted by the control sequence `\glue`. This is not a  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  command; it merely indicates the presence of glue in the current list.

The box representation that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  generated from, for instance, `\showbox` inserts a space after every explicit `\kern`, but no space is inserted after an implicit kern that was inserted by the kerning information in the font `tfm` file. Thus `\kern 2.0pt` denotes a kern that was inserted by the user or by a macro, and `\kern2.0pt` denotes an implicit kern.

Glue that is inserted automatically (`\topskip`, `\baselineskip`, et cetera) is denoted by name in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 's trace output. For example, the box

```
\vbox{\hbox{Vo}\hbox{b}}
```

looks like

```
\vbox(18.83331+0.0)x11.66669
.\hbox(6.83331+0.0)x11.66669
..\tenrm V
..\kern-0.83334
..\tenrm o
.\glue(\baselineskip) 5.05556
.\hbox(6.94444+0.0)x5.55557
..\tenrm b
```

Note the implicit kern inserted between ‘V’ and ‘o’.



## 第 9 章 Rules and Leaders

Rules and leaders are two ways of getting  $\text{\TeX}$  to draw a line. Leaders are more general than rules: they can also fill available space with copies of a certain box. This chapter explain how rules and leaders work, and how they interact with modes.

`\hrule` Rule that spreads in horizontal direction.


`\vrule` Rule that spreads in vertical direction.

`\leaders` Fill a specified amount of space with a rule or copies of box.

`\cleaders` Like `\leaders`, but with box leaders any excess space is split equally before and after the leaders.

`\xleaders` Like `\leaders`, but with box leaders any excess space is spread equally before, after, and between the boxes.

### 9.1 Rules

$\text{\TeX}$ 's rule commands give *rules*: rectangular black patches with horizontal and vertical sides. Most of the times, a rule command will give output that looks like a rule, but  can also be produced by a rule.

$\text{\TeX}$  has both horizontal and vertical rules, but the names do not necessarily imply anything about the shape. They do, however, imply something about modes: an `\hrule` command can only be used in vertical mode, and a `\vrule` only in horizontal mode. In fact, an `\hrule` is a  $\langle$ vertical command $\rangle$ , and a `\vrule` is a  $\langle$ horizontal command $\rangle$ , so  $\text{\TeX}$  may change modes when encountering these commands.

Why then is a `\vrule` called a *vertical* rule? The reason is that a `\vrule` can expand arbitrarily far in the vertical direction: if its height and depth are not specified explicitly it will take as much room as its surroundings allow.

例子:

```
\hbox{\vrule\ text \vrule}
```

*looks like*

| *text* |

*and*

```
\hbox{\vrule\ A gogo! \vrule}
```

*looks like*

| *A gogo!* |

For the `\hrule` command a similar statement is true: a horizontal rule can spread to assume the width of its surroundings. Thus

```
\vbox{\hbox{One line of text}\hrule}
```

looks like

One line of text

### 9.1.1 Rule dimensions

Horizontal and vertical rules have a default thickness:

```
\hrule is the same as \hrule height.4pt depth0pt
```

and

```
\vrule is the same as \vrule width.4pt
```

and if the remaining dimension remains unspecified, the rule extends in that direction to fill the enclosing box.

Here is the formal specification of how to indicate rule sizes:

⟨vertical rule⟩ → `\vrule`⟨rule specification⟩

⟨horizontal rule⟩ → `\hrule`⟨rule specification⟩

⟨rule specification⟩ → ⟨optional spaces⟩

| ⟨rule dimensions⟩⟨rule specification⟩

⟨rule dimension⟩ → `width`⟨dimen⟩ | `height`⟨dimen⟩ | `depth`⟨dimen⟩

If a rule dimension is specified twice, the second instance takes precedence over the first. This makes it possible to override the default dimensions. For instance, after

```
\let\xhrule\hrule \def\hrule{\xhrule height .8pt}
```

the macro `\hrule` gives a horizontal rule of double the original height, and it is still possible with

```
\hrule height 2pt
```

to specify other heights.

It is possible to specify all three dimensions; then

```
\vrule height1ex depth0pt width1ex
```

and

```
\hrule height1ex depth0pt width1ex
```

look the same. Still, each of them can be used only in the appropriate mode.

## 9.2 Leaders

Rules are intimately connected to modes, which makes it easy to obtain some effects. For instance, a typical application of a vertical rule looks like

```
\hbox{\vrule width1pt\ Important text! \vrule width 1pt}
```

which gives

| Important text! |

However, one might want to have a horizontal rule in horizontal mode for effects such as

← 5cm →  
from here\_\_\_\_\_to there

An `\hrule` can not be used in horizontal mode, and a vertical rule will not spread automatically.

However, there is a way to use an `\hrule` command in horizontal mode and a `\vrule` in vertical mode, and that is with *leaders*, so called because they lead your eye across the page. A leader command tells  $\text{\TeX}$  to fill a specified space, in whatever mode it is in, with as many copies of some box or rule specification as are needed. For instance, the above example was given as

```
\hbox to 5cm{from here\leaders\hrule\hfil to there}
```

that is, with an `\hrule` that was allowed to stretch along an `\hfil`. Note that the leader was given a horizontal skip, corresponding to the horizontal mode in which it appeared.

A general leader command looks like

```
(leaders)<box or rule><vertical/horizontal/mathematical skip>
```

where `<leaders>` is `\leaders`, `\cleaders`, or `\xleaders`, a `<box or rule>` is a `<box>`, `\vrule`, or `\hrule`, and the lists of horizontal and vertical skips appear in Chapter 6; a mathematical skip is either a horizontal skip or an `\mskip` (see page 221). Leaders can thus be used in all three modes. Of course, the appropriate kind of skip must be specified.

A horizontal (vertical) box containing leaders has at least the height and depth (width) of the  $\langle\text{box or rule}\rangle$  used in the leaders, even if, as can happen in the case of box leaders, no actual leaders are placed.

### 9.2.1 Rule leaders

*Rule leaders* fill the specified amount of space with a rule extending in the direction of the skip specified. The other dimensions of the resulting rule leader are determined by the sort of rule that is used: either dimensions can be specified explicitly, or the default values can be used.

For instance,

```
\hbox{g\leaders\hrule\hskip20pt f}
```

gives

g\_\_\_\_\_f

because a horizontal rule has a default height of .4pt. On the other hand,

```
\hbox{g\leaders\vrule\hskip20pt f}
```

gives

g■f

because the height and depth of a vertical rule by default fill the surrounding box.

Spurious rule dimensions are ignored: in horizontal mode

```
\leaders\hrule width 10pt \hskip 20pt
```

is equivalent to

```
\leaders\hrule \hskip 20pt
```

If the width or height-plus-depth of either the skip or the box is negative,  $\text{\TeX}$  uses ordinary glue instead of leaders.

### 9.2.2 Box leaders

Box leaders fill the available spaces with copies of a given box, instead of with a rule.

For all of the following examples, assume that a box register has been allocated:

```
\newbox\centerdot \setbox\centerdot=\hbox{\hskip.7em.\hskip.7em}
```

Now the output of

```
\hbox to 8cm {here\leaders\copy\centerdot\hfil there}
```

is

here . . . . . there

That is, copies of the box register fill up the available space.

Dot leaders, as in the above example, are often used for tables of contents. In such applications it is desirable that dots on subsequent lines are vertically aligned. The `\leaders` command does this automatically:

```
\hbox to 8cm {here\leaders\copy\centerdot\hfil there}
\hbox to 8cm {over here\leaders\copy\centerdot\hfil over there}
```

gives

```
here      . . . . . there
over here . . . . . over there
```

The mechanism behind this is the following:  $\text{\TeX}$  acts as if an infinite row of boxes starts (invisibly) at the left edge of the surrounding box, and the row of copies actually placed is merely the part of this row that is not obscured by the other contents of the box.

Stated differently, box leaders are a window on an infinite row of boxes, and the row starts at the left edge of the surrounding box. Consider the following example:

```
\hbox to 8cm {\leaders\copy\centerdot\hfil}
\hbox to 8cm {word\leaders\copy\centerdot\hfil}
```

which gives

```
. . . . .
word . . . . .
```

The row of leaders boxes becomes visible as soon as it does not coincide with other material.

The above discussion only talked about leaders in horizontal mode. Leaders can equally well be placed in vertical mode; for box leaders the ‘infinite row’ then starts at the top of the surrounding box.

### 9.2.3 Evenly spaced leaders

Aligning subsequent box leaders in the way described above means that the white space before and after the leaders will in general be different. If vertical alignment is not an issue it may be aesthetically more pleasing to have the leaders evenly spaced. The `\cleaders` command is like `\leaders`, except that it splits excess space before and after the leaders into two equal parts, centring the row of boxes in the available space.

例子:

```
\hbox to 7.8cm {here\cleaders\copy\centerdot\hfil there}
\hbox to 7.8cm {here is\cleaders\copy\centerdot\hfil there}
```

*gives*

*here . . . . . there*

*here is . . . . . there*

*The ‘expanding leaders’ \xleaders spread excess space evenly between the boxes, with equal globs of glue before, after, and in between leader boxes.*

例子:

```
\hbox to 7.8cm{here\hskip.7em
\xleaders\copy\centerdot\hfil \hskip.7em there}
```

*gives*

*here . . . . . there*

*Note that the glue in the leader box is balanced here with explicit glue before and after the leaders; leaving out these glue items, as in*

```
\hbox to 7.8cm {here\xleaders\copy\centerdot\hfil there}
```

*gives*

*here . . . . . there*

*which is clearly not what was intended.*

## 9.3 Assorted remarks

### 9.3.1 Rules and modes

Above it was explained how rules can only occur in the appropriate modes. Rules also influence mode-specific quantities: no `baselineskip` is added before rules in vertical mode. In order to prevent glue after rules, `TeX` sets `\prevdepth` to `-1000pt` (see Chapter 15). Similarly the `\spacefactor` is set to 1000 after a `\vrule` in horizontal mode (see Chapter 19).

### 9.3.2 Ending a paragraph with leaders

An attempt to simulate an `\hrule` at the end of a paragraph by

```
\nobreak\leaders\hrule\hfill\par
```

does not work. The reason for this is that `TeX` performs an `\unskip` at the end of a paragraph, which removes the leaders. Normally this `\unskip` removes any space token inserted by the input processor after the last line. Remedy: stick an `\hbox{}` at the end of the leaders.

### 9.3.3 Leaders and box registers

In the above examples the leader box was inserted with `\copy`. The output of

```
\hbox to 8cm {here\leaders\box\centerdot\hfil there}
\hbox to 8cm {over here\leaders\box\centerdot\hfil
               over there}
```

is

```
here      . . . . . there
over here                                over there
```

The box register is emptied after the first leader command, but more than one copy is placed in that first command.

### 9.3.4 Output in leader boxes

Any `\write`, `\openout`, or `\closeout` operation appearing in leader boxes is ignored. Otherwise such an operation would be executed once for every copy of the box that would be shipped out.

### 9.3.5 Box leaders in trace output

The dumped box representation obtained from, for instance, `\tracingoutput` does not write out box leaders in full: only the total size and one copy of the box used are dumped. In particular, the surrounding white space before and after the leaders is not indicated.

### 9.3.6 Leaders and shifted margins

If margins have been shifted, leaders may look different depending on how the shift has been realized. For an illustration of how `\hangindent` and `\leftskip` influence the look of leaders, consider the following examples, where

```
\setbox0=\hbox{K o }
```

The horizontal boxes above the leaders serve to indicate the starting point of the row of leaders.

First

```
\hbox{\leaders\copy0\hskip5cm}
\noindent\advance\leftskip 1em
\leaders\copy0\hskip5cm\hbox{ }\par
```

gives

```
K o K o K o K o K o K o K o
K o K o K o K o K o K o K o
```

Then

```
\hbox{\kern1em\hbox{\leaders\copy0\hskip5cm}}
\hangindent=1em \hangafter=-1 \noindent
\leaders\copy0\hskip5cm\hbox{}\par
```

gives (note the shift with respect to the previous example)

K o K o K o K o K o K o K o

K o K o K o K o K o K o K o

In the first paragraph the `\leftskip` glue only obscures the first leader box; in the second paragraph the hanging indentation actually shifts the orientation point for the row of leaders. Hanging indentation is performed in  $\text{\TeX}$  by a `\moveright` of the boxes containing the lines of the paragraph.



## 第 10 章 编组

**T<sub>E</sub>X** 提供的编组机制可以将大多数改变限制在局部范围内。本章解释哪些操作是局部的，以及编组是如何形成的。

`\bgroup` 隐式的编组开始符。

`\egroup` 隐式的编组结束符。

`\begingroup` 开始一个必须用 `\endgroup` 结束的编组。

`\endgroup` 结束一个用 `\begingroup` 开始的编组。

`\aftergroup` 保存下一个记号，并在当前编组结束后插入它。

`\global` 使得赋值，宏定义和算术运算是全局的。

`\globaldefs` 用于覆盖 `\global` 前缀的参数。**IniT<sub>E</sub>X** 默认值：0。

### 10.1 编组机制

一个编组是一串记号，以‘组开始’记号开头，以‘组结束’记号结尾，而且这两种记号在其中是配对的。

**T<sub>E</sub>X** 的编组机制与普通编程语言的块作用域不同。大多数语言的块结构中只能有局部定义。**T<sub>E</sub>X** 的编组机制更加强大：编组中的大多数赋值是局部的，在离开该编组后旧的值将被还原，但可以显式指明全局赋值。

下面局部定义的例子

```
{\def\ a{b}}\ a
```

将给出‘undefined control sequence’的错误，因为 `\ a` 是在编组内部定义的，类似地，下列代码

```
\count0=1 {\count0=2 } \showthe\count0
```

显示的值为 1；编组内的赋值在该编组结束后将被撤销。

对编组结束时要还原的值的记录是通过保存堆栈的结构来实现的。保存堆栈的溢出问题在第 35 章中介绍。这个保存堆栈还有另外几个用途：比如在调用 `\hbox to 100pt{...}` 时，在开始新层级的编组之前，所指定的 `to 100pt` 将被放入保存堆栈中。

为避免给保存堆栈造成各种麻烦，**IniTeX** 不允许在编组内用 `\dump` 命令转储格式文件。`\end` 命令可以出现在编组内，但 **TeX** 将对此给出一个诊断信息。

控制系列 `\aftergroup` 保存一个记号，并在当前编组结束后插入它。此命令可以放置多个记号，这些记号将按出现的先后顺序被插入。此命令将在第 12 章中讨论。

## 10.2 局部和全局赋值

给赋值或宏定义加上 `\global` 前缀，通常可让它们成为全局的，但 `\integer parameter` `\globaldefs` 的非零值将覆盖 `\global` 指定：如果 `\globaldefs` 为正数，每个赋值都被隐式地添加 `\global` 前缀，而如果 `\globaldefs` 为负数，`\global` 将会被忽略。通常这个参数的取值为零。

有些赋值总是全局的：所有的 `\global assignment` 如下：

**<font assignment>** `\fontdimen`、`\hyphenchar` 和 `\skewchar` 的赋值。

**<hyphenation assignment>** `\hyphenation` 和 `\patterns` 命令（见第 19 章）。

**<box size assignment>** 用 `\ht`、`\dp` 和 `\wd` 修改盒子尺寸（见第 5 章）。

**<interaction mode assignment>** **TeX** 任务的运行模式（见第 32 章）。

**<intimate assignment>** 对 `\special integer` 或 `\special dimen` 的赋值；见第 85 和 92 页。

## 10.3 编组定界符

编组定界符可以由类别码为 1 的‘组开始符’和类别码为 2 的‘组结束符’组成（显式花括号），或者用 `\let` 等价到这两类字符的控制序列组成（隐式花括号），正如 **plain TeX** 中的 `\bgroup` 和 `\egroup`。隐式和显式花括号可以配对以定界一个编组；比如见第 12.3.4 节的例子。

编组也可以用 `\begingroup` 和 `\endgroup` 定界。这两个控制序列必须一起使用，它们不能与隐式或显式花括号配对，也不能像花括号那样用于围住盒内素材等。

用 `\begingroup` 和 `\endgroup` 定界可以提供一种有限的运行时错误检测。在这两个编组定界符之间一个多余的开或闭花括号将有

```
\begingroup ... } ... \endgroup
```

或者

```
\begingroup ... { ... \endgroup
```

的形式。对这两种情形 **TeX** 都将给出括号不配对的错误信息。这里若改用 `\bgroup` 和 `\egroup` 将使得错误更加难以发现，因为出现了不正确的配对。这个想法在多种环境宏中用到。

$\text{\TeX}$  并没有硬性规定花括号作为组开始和结束符。在 `plain` 格式中，它是类似这样编写的：

```
\catcode\{=1 % left brace is begin-group character
\catcode\}=2 % right brace is end-group character
```

隐式花括号也在 `plain` 格式中定义：

```
\let\bgroup={ \let\egroup=}
```

下面列出一些特殊情形：

- 宏的替换文本必须用显式组开始符和组结束符围住。
- 盒子、`\vadjust` 和 `\insert` 的开始和结束花括号可以是隐式的。这样就可能像下面这样定义

```
\def\openbox#1{\setbox#1=\hbox\bgroup}
\def\closebox#1{\egroup\box#1}
\openbox{15}Foo bar\closebox{15}
```

- 记号列赋值的右边，以及命令 `\write`、`\message`、`\errmessage`、`\uppercase`、`\lowercase`、`\special` 和 `\mark` 的参量是  $\langle\text{general text}\rangle$ ，定义为

$\langle\text{general text}\rangle \longrightarrow \langle\text{filler}\rangle\{\langle\text{balanced text}\rangle\langle\text{right brace}\rangle$

这意味着左花括号可以是隐式的，但右花括号必须是类别码 2 的显式字符记号。

在可以使用隐式花括号的情形中，如果展开没被显式禁止， $\text{\TeX}$  将展开记号直到遇到一个左花括号。这是 `\uppercase\expandafter{\romannumeral80}` 这种写法的原理，在未展开之前这样的写法并不符合语法。如果第一个不可展开的记号不是一个左花括号， $\text{\TeX}$  将给出一个错误信息。

在  $\text{\TeX}$  的文法（见第 36 章）中，用  $\langle\text{left brace}\rangle$  和  $\langle\text{right brace}\rangle$  表示显式字符，即字符记号；而用 `{` 和 `}` 表示可能为隐式的字符，即已经用 `\let` 等价到这些显式字符的控制序列。

## 10.4 花括号进阶

### 10.4.1 花括号计数器

$\text{\TeX}$  有两个计数器用于记录编组的层级：主计数器和平衡计数器。这两个计数器都是语法计数器：它们计算显式花括号记号的个数，但不受语义上与显式花括号等价的隐式花括号（比如 `\bgroup`）的影响。

平衡计数器处理除了阵列之外的所有情形。其工作原理直观而清晰：对每个左花括号增加一，对每个右花括号减少一，只要该花括号不被跳过。因此对于下面的语句

```
\iffalse{\fi
```

在它仅仅被扫描时（比如它出现在宏定义文本中）将增加平衡计数器的值；而在它被执行时花括号将被跳过，从而不会改变平衡计数器的值。

主计数器更加复杂；在阵列中只使用主计数器，不使用平衡计数器。这个计数器记录所有花括号，即使在被跳过时，像 `\iffalse{\fi}` 这种。对这个计数器，未计数的已跳过花括号还是可能的：当字母表常数 ``{` 和 ``}` 被执行处理器作为 `<number>` 使用时，不影响这个计数器；当它们被输入处理器查看（仅查看字符，不含上下文）时，确实会影响这个计数器。

### 10.4.2 花括号作为记号

显式花括号是字符记号，而作为字符记号它们是不可展开的。这表示它们直到 **T<sub>E</sub>X** 处理的最后一个阶段还是存在的。例如，

```
\count255=1{2}
```

将给 `\count255` 赋值 1，并排印 ‘2’，因为左花括号充当了数字 1 的定界符。类似地，

```
f{f}
```

将阻止 **T<sub>E</sub>X** 形成 ‘ff’ 连写。

从花括号不可展开的事实，可以得知它们的嵌套与条件语句的嵌套相互独立。比如

```
\iftrue{\else}\fi
```

将给出左花括号，因为条件语句是在展开时处理的。右花括号作为 `<false text>` 的一部分被直接跳过了；这对编组的影响在 **T<sub>E</sub>X** 处理的后续阶段才显现出来。

非定界的宏参量或者是单个记号，或者是一个用显式花括号围住的编组。因此，显式的左或右花括号不能作为宏参量。然而，花括号可以用于 `\let` 赋值，就像这样

```
\let\bgroup={
```

这在 **plain T<sub>E</sub>X** 的 `\footnote` 宏（见第 135 页）中用到。

### 10.4.3 花括号控制符号

控制序列 `\{` 和 `\}` 并不属于这一章，也与编组无关。它们被用 `\let` 分别定义为 `\lbrace` 和 `\rbrace` 的别名，而这两个控制序列为 `\delimiter` 指令（见第 21 章）。

计算机现代罗马字体中不包含花括号，但打字机字体中包含；而对于数学公式，在扩展字体中有不同尺寸的 – 且可伸展的 – 花括号。

# 第 11 章 宏定义

在  $\text{T}_{\text{E}}\text{X}$  中，宏是对一串需要多次用到的命令的缩写机制，它有点像普通编程语言的过程。然而  $\text{T}_{\text{E}}\text{X}$  的参数机制是与众不同的。这一章解释  $\text{T}_{\text{E}}\text{X}$  的宏如何运作，并介绍命令 `\let` 和 `\futurelet`。

`\def` 开始一个宏定义。

`\gdef` 等同于 `\global\def`。

`\edef` 开始一个宏定义；在定义时替换文本被展开。此命令在下一章也会介绍。

`\xdef` 等同于 `\global\edef`。

`\csname` 开始生成一个控制序列的名称。

`\endcsname` 结束生成一个控制序列的名称。

`\global` 使得后面的定义、算术语句或赋值是全局的。

`\outer` 此前缀表示正在定义的宏只能用在‘外部’层级中。

`\long` 此前缀表示正在定义的宏的参量可以包含 `\par` 记号。

`\let` 将一个控制序列定义为等价于下一个记号。

`\futurelet` 将一个控制序列定义为等价于下一个记号之后的记号。

## 11.1 介绍

基本上宏是缩写为一个控制序列的一串记号。以 `\def` 等开始的语句称为宏定义，而语句

```
\def\abc{\de f\g}
```

就定义了以 `\de f\g` 为替换文本的宏 `\abc`。以这种方式，宏可以用于缩写需要多次用到的一段文本或一串命令。在任何时候，只要  $\text{T}_{\text{E}}\text{X}$  的展开处理器遇到控制序列 `\abc`，它就将这个宏用其替换文本代替。

如果想让宏依赖它被用到时的上下文，就可以将它定义为带参数的宏。像下面这样定义的 `\PickTwo`

```
\def\PickTwo#1#2{(#1,#2)}
```

有两个参数。这个宏被用到时将取出两段文本，即对应的参量，并将它们排印在圆括号中。例如：

	macro	argument1	argument2	expansion
definition	<code>\def\PickTwo</code>	<code>#1</code>	<code>#2</code>	<code>{ (#1,#2) }</code>
use	<code>\PickTwo</code>	<code>1</code>	<code>2</code>	<code>(1,2)</code>
use	<code>\PickTwo</code>	<code>{ab}</code>	<code>{cd}</code>	<code>(ab,cd)</code>

将宏及其参量用替换文本代替的活动称为宏展开。

11.2 宏定义的结构

一个宏定义按照顺序包含下列各部分：

- 1. 任意多个 `\global`、`\long` 和 `\outer` 前缀，
- 2. 一个 `(def)` 控制序列，或者任何用 `\let` 等价到此种控制序列的东西，
- 3. 一个将要定义的控制序列或活动字符，
- 4. 一个可能存在的 `(parameter text)` 用于指定宏参数的个数及其他东西，以及
- 5. 一个用类别码为 1 和 2 的显式字符记号包围的替换文本，在 `plain TEX` 中这两类字符默认为 `{` 和 `}`。

这些元素将在接下来各节中解释。

在宏定义完成之后，任何已保存的 `\afterassignment` 记号（见第 12.3.3 节）将被插入进来。

‘展开的’定义 `\edef` 和 `\xdef` 将在第 12 章讨论。

11.3 前缀

有三种前缀用于改变宏定义的状态：

`\global` 如果定义出现在编组内，此前缀将使得该定义成为全局的。除了宏定义之外，此前缀还能用于赋值；实际上，对于宏定义，可以用缩写形式而不用 `\global`：

`\gdef\foo...` 等价于 `\global\def\foo...`

而

`\xdef\foo...` 等价于 `\global\edef\foo...`

如果参数 `\globaldefs` 为正数，所有赋值都被视为全局的；而如果 `\globaldefs` 为负数，所有 `\global` 前缀都被忽略，而且 `\gdef` 和 `\xdef` 也生成局部的定义（见第 10 章）。

`\outer` 外部宏的定义机制用于帮助定位未配对花括号（及其他错误）：`\outer` 宏只能出现在非嵌入语境中。更准确地说，它不可以出现

- 在宏的替换文本中（但如果将它放在 `\noexpand` 或 `\meaning` 之后，就可以出现在 `\edef` 的替换文本中），
- 在参数文本中，
- 在跳过的条件文本中，
- 在阵列的导言中，以及
- 在 `\message`、`\write` 等的 `(balanced text)` 中。

然而，在特定应用中某些 **plain** 宏是外部的很不方便，特别是像 `\newskip` 这样的宏。有个补救方法是将它们重新定义为非外部的宏，这正是 **L<sup>A</sup>T<sub>E</sub>X** 所做的，但还有更巧妙的做法。

`\long` 宏的参量通常不允许包含 `\par` 记号。这个限制在定位遗漏的右括号时很有用（比 `\outer` 定义有用得多）。例如，对于下面的输入 **T<sub>E</sub>X** 将抱怨 ‘runaway argument’：

```
\def\aa#1{ ... #1 ... }
\aa {This sentence should be in braces.
```

```
And this is not supposed to be part of the argument
```

其中的空行生成一个 `\par`，这在多数时候意味着有个右花括号漏掉了。

如果某个宏允许其参量包含 `\par` 记号，则这个宏应当定义为 `\long` 宏。

在测试两个记号是否相等时（见第 13 章），`\ifx` 也会将前缀考虑在内。

## 11.4 定义的类型

在 **T<sub>E</sub>X** 中有四种 `(def)` 控制序列：`\def`、`\gdef`、`\edef` 和 `\xdef`。其中 `\gdef` 是 `\global\def` 的同义词，而 `\xdef` 是 `\global\edef` 的同义词。而 ‘展开的定义’ `\edef` 在第 12 章中介绍。

各种宏定义仅在定义时有区别。在宏被调用时是无法知道它们是如何定义的。

## 11.5 参数文本

在所定义的控制序列或活动字符以及替换文本的左花括号之间，可以有 `(parameter text)`，参数文本有点像普通编程语言的参量。它指定这个宏是否有参数，有多少个参数，以及各参数之间如何定界。`(parameter text)` 不能包含显式花括号。

一个宏最多可以有九个参数。参数用参数记号表示，参数记号由宏参数字符（即类别码为 6 的字符，在 **plain T<sub>E</sub>X** 中为 #）后跟一个 1–9 的数字组成。举个例子，`#6` 表示宏的第六个参数。参数记号不能出现在宏定义之外的其他地方。

在参数文本中，参数必须从 1 开始顺序编号。参数记号后的空格是有意义的，不管是在参数文本还是在替换文本中。

参数分为定界参数和非定界参数；它决定宏参量的范围。如果在 `<parameter text>` 中一个参数后面紧跟着另一个参数，就像 `\def\foo#1#2` 这样，则前面的参数就是非定界的。如果一个参数后面紧跟着替换文本的左花括号，就像 `\def\foo#1{...}` 这样，则它也是非定界的。一个参数称为定界的，如果它后面紧跟着其他记号；比如 `\def\foo#1!#2{...}` 的第一个参数就是被感叹号定界的参数。

在宏展开（或‘调用’）时，用于替换参数的（零个或多个）记号称为该参数对应的‘参量’。

### 11.5.1 非定界参数

在带有非定界参数的宏，比如单参数宏 `\foo`

```
\def\foo#1{ ... #1 ...}
```

被展开时，**T<sub>E</sub>X** 往前扫描（但不展开）直到遇到一个非空格记号。如果这个记号不是显式 `<left brace>`，它就被取为对应于该参数的参量。否则，通过扫描直到找到匹配的显式 `<right brace>`，**T<sub>E</sub>X** 得到一个 `<balanced text>`。这个平衡文本就要找的参量。

这里是个带有三个非定界参数的宏的例子：对于

```
\def\foo#1#2#3{#1(#2)#3}
```

宏调用 `\foo123` 给出 `‘1(2)3’`；而 `\foo 1 2 3` 给出同样的结果。在调用

```
\foo 1 2 3
```

中，第一个空格被 **T<sub>E</sub>X** 的输入处理器跳过。从而对应于第一个参数的参量是 1。为找到第二个参量，**T<sub>E</sub>X** 跳过所有空格（在此例子中跳过一个空格），最后找到的第二个参量是 2。类似的第三个参量是 3。

为了将多个记号作为一个非定界参量，你可以使用花括号。对于上述的 `\foo` 定义，调用 `\foo a{bc}d` 将给出 `‘a(bc)d’`。当宏的参量是平衡文本而非单个记号时，在将参量插入替换文本时定界花括号会被去掉。例如：

```
\def\foo#1{\count0=1#1\relax}
\foo{23}
```

将展开为 `\count0=123\relax`，这将对计数器赋值 123。另一方面，下面语句

```
\count0=1{23}
```

将赋值 1 并排印 23。

### 11.5.2 定界参数

除了将它们括在花括号里面，还有另一种方式可将一串记号作为宏的一个参量，即使用定界的参数。



在  $\langle$ parameter text $\rangle$  中，出现在宏参数之后（即在紧跟参数字符的参数编号之后）的非参数记号被当作该参数的定界子。定界子可以包含空格记号：参数编号之后的空格是有意义的。定界记号同样出现在所定义的控制序列和它的第一个参数记号 #1 之间。

对于在参数文本中充当定界子的字符记号，它们的字符码和类别码都被存储下来；实际参量的定界字符记号必须和两者都匹配。这些字符可以拥有一些通常只出现在特殊环境中的类别码；比如，在定义

```
\def\foo#1_#2^{...}
```

之后，宏 `\foo` 可以在数学模式之外使用。

在寻找对应于一个定界参数的参量时，**T<sub>E</sub>X** 吸收所有记号而不展开它们（但保持花括号配对），直到遇到（完全一致的）定界记号串。定界记号串不是参量的一部分；在宏调用时它们被从输入流中移除。

### 11.5.3 定界参量举例

作为一个简单例子，我们定义一个宏

```
\def\DoASentence#1#2.{\bf#1#2.}}
```

它的第一个参数是非定界的，而第二个参数是用句号定界的。像下面这样调用时

```
\DoASentence \bf This sentence is the argument.
```

它的两个参量分别是：

```
#1<-\bf
#2<-This sentence is the argument
```

注意结尾的句号不在参量中，但它已经被吸收了，不会再出现在输入流中。

常用的定界子是 `\par`，例如：

```
\def\section#1. #2\par{\medskip\noindent {\bf#1. #2\par}}
```

这个宏第一个参数用 ‘.’ 定界，而第二个参数用 `\par` 定界。像下面调用此宏

```
\section 2.5. Some title
```

```
The text of the section...
```

将给出

```
#1<-2.5
#2<-Some title
```

注意在第二个由行尾符生成的参量的末尾有个空格。如果这个空格是多余的，你可以定义

```
\def\section#1. #2 \par{...}
```

从而用 `\par` 定界第二个参量。然而这种方法导致用户不可以显式写上 `\par`：

```
\section 2.5 Some title\par
```

解决这种两难选择的其中一种方法是，在需要去掉结尾空格时，在定义文本中写上 `#2\unskip`。

充当定界子的控制序列不需要已经定义，因为它们只被吸收不会被展开。因此

```
\def\control#1\sequence{...}
```

是有效的定义，即使在 `\sequence` 未定义时。

下面的例子展示了定界参量的类别码是至关重要的：

```
\def\a#1 #2.{ ... }
\catcode\ =12
\ a b c
d.
```

将给出

```
\ a #1 #2.-> ...
#1<- b c
#2<-d
```

解释：参数 1 和参数 2 之间的定界子是一个第 10 类的空格。在 `a` 和 `b` 之间有一个第 12 类的空格；第一个第 10 类的空格是由行尾生成的空格。

在第 13 章对 `\newif` 的解释，以及在第 33 页的例子中，有类别码匹配的实际例子。

#### 11.5.4 空参量

在  $\TeX$  需要一个宏参量时，如果用户指定一个围在花括号中的 (balanced text)，则该文本就被用作参量。因此，指定 `{ }` 给出的参量是一个空记号列；它称为‘空参量’。

在使用定界参数时也可能得到空参量。例如，在下述定义

```
\def\mac#1\ro{ ... }
```

之后，这样调用

```
\mac\ro
```

将给出一个空参量。

#### 11.5.5 宏参数字符

在  $\TeX$  的输入处理器扫描宏定义文本时，它对后面跟着数字的每个宏参数字符插入一个参数记号。实际上，在替换文本中，参数记号表示‘在这里插入某某编号的参量’。连续两个参数字符被替换为一个。

后一个的事实可以用于嵌套的宏定义。下面的定义

```
\def\a{\def\b#1{...}}
```

给出一个错误信息，因为 `\a` 被定义为不带参数的宏，而在它的替换文本中有一个参数记号。

下面的语句

```
\def\a#1{\def\b#1{...}}
```

定义了宏 `\a`，这个宏又定义了另一个宏 `\b`。然而，`\b` 任然不带任何参数：这样调用

```
\a z
```

就定义了一个不带参数的宏 `\b`，其后必须跟着 `z`。注意这并不是在定义宏 `\bz`，因为 `TEX` 的输入处理器读取输入行时就已经形成了控制序列 `\b`。

最后，

```
\def\a{\def\b##1{...}}
```

定义了带一个参数的宏 `\b`。

我们来仔细检查一下参数字符的处理过程。考虑下面定义

```
\def\a#1{ .. #1 .. \def\b##1{ ... } }
```

当这个语句作为输入被读取时，输入处理器

- 将字符串 `#1` 替换为 `(parameter token1)`，并且
- 将字符串 `##` 替换为 `#`

在调用宏 `\a` 时，输入处理器将扫描

```
\def\b#1{ ... }
```

并将其中两个字符的 `#1` 替换为一个参数记号。

### 11.5.6 花括号定界

在定义的 `(parameter text)` 中通常是不可能左或右花括号的。然而，有一种特殊规定可以让宏的最后一个参数看似用左花括号定界的。

如果最后一个参数记号后面是一个参数字符 (`#`)，接着是替换文本的左花括号，`TEX` 将让最后一个参数以组开始符定界。此外，与参数文本的其他定界记号不同，这个左花括号不会被从输入流中移除。

考虑一个例子。假设你想有个宏 `\every` 宏用于像下面这样填充记号列：

```
\every par{abc} \every display{def}
```

这个宏可以定义为

```
\def\every#1#{\csname every#1\endcsname}
```

在上面的第一个调用中，对应于参数的参量是 `abc`，因此该调用展开为

```
\csname everypar\endcsname{abc}
```

这就给出所要的结果。

## 11.6 构造控制序列

命令 `\csname` 和 `\endcsname` 可用于构造控制序列。例如

```
\csname hskip\endcsname 5pt
```

等价于 `\hskip5pt`。

在构造的过程中，介于 `\csname` 和 `\endcsname` 之间的所有宏和其他可展开控制序列都如常展开，直到仅留下不可展开的字符记号。若将上面例子改写为

```
\csname \ifhmode h\else v\fi skip\endcsname 5pt
```

则它将依据当前模式执行 `\hskip` 或 `\vskip`。展开的最后结果必须只包含字符记号，但对它们的类别码却无限制。在碰到不可展开的控制序列时， $\text{\TeX}$  将抛出一个错误，并在错误恢复时在该处之前插入一个 `\endcsname`。

利用 `\csname`，我们可以构造通常无法写出的，包含类别码不为 11（字母）的字符的控制序列。这个原理可用于对用户隐藏宏包内部的控制序列。

例子：

```
\def\newcounter#1{\expandafter\newcount
  \csname #1:counter\endcsname}
\def\stepcounter#1{\expandafter\advance
  \csname #1:counter\endcsname 1\relax}
```

第二个定义中的 `\expandafter` 是多余的，但它没有坏处，且可以让代码更清楚。

`\newcounter` 创建的计数器的名称中包含一个冒号，因此写起来有点麻烦。好处是现在这个计数器对用户是隐藏的，它只能通过类似 `\stepcounter` 的控制序列来访问。顺便说一下，在 `plain` 格式中 `\newcount` 被定义为 `\outer` 宏，因此在重新定义 `\newcount` 后才能写出上述定义。

如果用 `\csname...\endcsname` 生成的控制序列之前尚未定义，它就被定义为 `\relax`。因此如果 `\xx` 是一个未定义的控制序列，命令

```
\csname xx\endcsname
```

将不会导致错误信息，因为它等价于 `\relax`。此外，在执行 `\csname...\endcsname` 语句后，控制序列 `\xx` 本身等价于 `\relax`，因此它也不再导致 ‘undefined control sequence’ 的错误（另见第 137 页）。

## 11.7 用 `\let` 和 `\futurelet` 给出记号赋值

在  $\text{\TeX}$  中有两个 (let assignment)。它们的语法为

```
\let<control sequence><equals><one optional space><token>
\futurelet<control sequence><token><token>
```

在 `\futurelet` 赋值的语法中，不可以出现可选的等号。

### 11.7.1 `\let`

原始命令 `\let` 将某个记号的当前含义赋予一个控制序列或活动字符。

例如，在 `plain` 格式中，`\endgraf` 定义为

```
\let\endgraf=\par
```

这使得宏的编写者可以重新定义 `\par`，而原始的 `\par` 命令的功能仍然可以使用。例如，

```
\everypar={\bgroup\it\def\par{\endgraf\egroup}}
```

在第 3 章中已经讨论了  $\langle \text{token} \rangle$  不是控制序列而是字符记号的情形。

### 11.7.2 `\futurelet`

如上所述，`\let` 系列

```
\let<control sequence><token1><token2><token3><token...>
```

将  $\langle \text{token}_1 \rangle$ （的含义）赋予该控制序列，而剩下的输入流如下

```
<token2><token3><token...>
```

即  $\langle \text{token}_1 \rangle$  从输入流中消失了。

命令 `\futurelet` 的运作稍有不同：对于下面的输入流

```
\futurelet<control sequence><token1><token2><token3><token...>
```

它将  $\langle \text{token}_2 \rangle$ （的含义）赋予该控制序列，而剩下的输入流如下

```
<token1><token2><token3><token...>
```

也就是说， $\langle \text{token}_1 \rangle$  和  $\langle \text{token}_2 \rangle$  都不会被从输入流中去掉。然而，现在  $\langle \text{token}_1 \rangle$  无需将  $\langle \text{token}_2 \rangle$  取为宏参数就‘知道’它是什么。见后面给出的例子。

在字符记号已经被 `\futurelet` 到一个控制序列时，它的类别码就已经确定了。随后的  $\langle \text{token}_1 \rangle$  无法再改变它。

## 11.8 杂项注记

### 11.8.1 活动字符

类别码 13 的字符记号称为活动字符，我们可以像控制序列那样定义它。如果这种字符的定义出现在宏里面，在宏定义时这个字符必须是活动的。

以下面的定义为例（取自第 2 章）：

```
{\catcode\^^M=13 %
\gdef\obeylines{\catcode\^^M=13 \def^^M{\par}}%
}
```

在定义 `\obeylines` 前必须先设定好  $\^^M$  字符的类别码，否则  $\text{\TeX}$  会认为该行在 `\def` 之后结束。

### 11.8.2 宏与原始命令

相比其他编程语言，在  $\text{\TeX}$  中原始命令与用户宏的区分并不大重要。

- 用户可以用别的名称使用原始命令：

```
\let\StopThisParagraph=\par
```

- 原始命令的名称可以给用户宏使用：

```
\def\par{\hfill$\bullet$\endgraf}
```

- 很多原始命令和用户宏一样都可以展开，比如所有条件句，以及类似 `\number` 和 `\jobname` 的命令。

### 11.8.3 尾递归

**T<sub>E</sub>X**中的宏，与大多数现代编程语言一样，可以是递归的：即在宏定义中可以调用这个宏本身，或者另一个将调用这个宏的宏。如果同一个递归宏的‘化身’在同一时间出现太多次，它容易弄乱 **T<sub>E</sub>X** 的内存。然而，对于尾递归这种常见的递归情形，**T<sub>E</sub>X** 能够避免出现混乱。

要理解这里发生的事情，需要一些背景知识。在开始执行一个宏时，**T<sub>E</sub>X** 抓取宏的参量，然后在输入栈上放置一个指向替换文本的项目，这样扫描器接着将转到替换文本中。一旦它处理完毕，输入栈中的这个项目将被移除。然而，如果宏的定义文本中包含另外的宏，这个过程将对它们重复进行：新的项目将被放在输入栈上，引导扫描器到其他宏中，甚至在第一个宏还未完成时，

一般来说，这种‘栈构建’不是很好但却是无可避免的；然而如果嵌套宏调用出现在宏的替换文本末尾的记号处，这是可以避免的。在末尾的记号后没有其他记号需要处理，因此我们可以在新的宏放进来之前清除输入栈顶部的项目。这正是 **T<sub>E</sub>X** 所做的事情。

Plain **T<sub>E</sub>X** 的 `\loop` 对这个原理给出很好的说明。它的定义是

```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body \let\next=\iterate
\else \let\next=\relax\fi \next}
```

而这个宏可以用类似下面的例子来调用：

```
\loop \message{\number\MyCount}
\advance\MyCount by 1
\ifnum\MyCount<100 \repeat
```

宏 `\iterate` 可以调用它自己，而且当它这样做时，递归调用出现在列表的末尾记号。也许可以将 `\iterate` 定义为

```
\def\iterate{\body \iterate\fi}
```

但这样 **T<sub>E</sub>X** 将无法消除这个递归，因为对 `\iterate` 的调用并不出现在 `\iterate` 的替换文本的末尾记号。这里的赋值 `\let\next=\iterate` 就是一种让递归调用出现在列表的末尾记号的方法。

另一种消除尾递归的方法是使用 `\expandafter`（见第 155 页）：在下面

```
\def\iterate{\body \expandafter\iterate\fi}
```

这种写法中，它去掉了 `\fi` 记号。如果列表的末尾记号是递归宏的参量，尾递归也将被消除。

顺便说一句：如果将 `\iterate` 定义为

```
\def\iterate{\let\next\relax
  \body \let\next\iterate \fi \next}
```

就可以这样写

```
\loop ... \if... ... \else ... \repeat
```

## 11.9 宏的技巧

### 11.9.1 不确定个数的参量

在某些应用中，我们希望宏的参量个数不用先规定好。

考虑在国际象棋棋盘上转换位置的问题（全部宏和字体可以在 [37] 和 [47] 中找到），比如要将棋谱记法

```
\White(Ke1,Qd1,Na1,e2,f4)
```

转换为一串排版好的走法

```
\WhitePiece{K}{e1} \WhitePiece{Q}{d1} \WhitePiece{N}{a1}
\WhitePiece{P}{e2} \WhitePiece{P}{f4}
```

注意在位置列表中兵前面的 ‘P’ 被省略掉了。

首要问题在于棋子列表是变长的，因此我们添加一个终止棋子：

```
\def\White(#1){\xWhite#1,xxx,}
\def\endpiece{xxx}
```

这样我们就可以检测到。接下来，`\xWhite` 从列表中读取一个位置，检测它是否为终止棋子；如果不是，依据它是否为一个兵分别处理。

```
\def\xWhite#1,{\def\temp{#1}%
  \ifx\temp\endpiece
  \else \WhitePieceOrPawn#1XY%
    \expandafter\xWhite
  \fi}
```

其中必须用 `\expandafter` 命令去掉 `\fi`（见第 155 页），使得 `\xWhite` 以下一个位置而不是 `\fi` 为参量。

位置的长度为两个或三个字符长。在调用四参数宏 `\WhitePieceOrPawn` 时添加了一个终止字符串 XY。因此，对棋子为兵的情形，第 3 个参量为字符 X 而第 4 个参量为空；对其他棋子的情形，第 1 个参量是该棋子，而第 2 和 3 个参量为具体的位置，第 4 个参量为 X。

```
\def\WhitePieceOrPawn#1#2#3#4Y{
  \if#3X \WhitePiece{P}{#1#2}%
  \else \WhitePiece{#1}{#2#3}\fi}
```

### 11.9.2 检查参量

在有些时候。我们需要检查宏参量中是否包含某个元素。考虑下面的实际例子（另见第 231 页的 `\DisplayEquation` 的例子）。

假设文章的标题和作者为

```
\title{An angle trisector}
\author{A.B. Cee\footnote*{Research supported by the
Very Big Company of America}}
```

而有多个作者时为

```
\author{A.B. Cee\footnote*{Supported by NSF grant 1}
\and
X.Y. Zee\footnote**}{Supported by NATO grant 2}}
```

另外假设 `\title` 和 `\author` 宏定义为

```
\def\title#1{\def\TheTitle{#1}} \def\author#1{\def\TheAuthor{#1}}
```

这将会被这样使用

```
\def\ArticleHeading{ ... \TheTitle ... \TheAuthor ... }
```

有些期刊要求文章的作者和标题全部大写。这种要求的实现方式可能是

```
\def\ArticleCapitalHeading
{ ...
\uppercase\expandafter{\TheTitle}
...
\uppercase\expandafter{\TheAuthor}
...
}
```

现在 `\expandafter` 命令将展开标题和作者为实际文本，接着 `\uppercase` 命令将把它们变为大写。然而，对于这样写对作者名来说是错误的，因为 `\uppercase` 命令将脚注文本也大写了。现在的问题在于如何仅将在脚注之间的文本变为大写。

作为第一次尝试，让我们先考虑只有一个作者的情形，设基本调用为

```
\expandafter\UCnoFootnote\TheAuthor
```

这将展开为

```
\UCnoFootnote A.B. Cee\footnote*{Supported ... }
```

这个宏

```
\def\UCnoFootnote#1\footnote#2#3{\uppercase{#1}\footnote{#2}{#3}}
```

将正确地分析它：

```
#1<-A.B. Cee
#2<-*
#3<-Supported ...
```

然而，如果脚注不存在时，这个宏将是完全错误的。

作为首个改进，我们自己添加脚注，以让它始终存在：

```
\expandafter\UCnoFootnote\TheAuthor\footnote 00
```



现在我们得检测所找到的脚注的类型：

```
\def\stopper{0}
\def\UCnoFootnote#1\footnote#2#3{\uppercase{#1}\def\tester{#2}%
\ifx\stopper\tester
\else\footnote{#2}{#3}\fi}
```

我们用 `\ifx` 区分出定界脚注号和实际脚注号。注意如果改用

```
\ifx0#2
```

将是错误的，因为脚注号可能包含多个记号，比如 `{**}`。

到目前为止，如果没有脚注这个宏是正确的，但如果有个脚注它是错误的：这几个终结记号还有待我们处理掉。在下面这个版本中它们会被小心处理好：

```
\def\stopper{0}
\def\UCnoFootnote#1\footnote#2#3{\uppercase{#1}\def\tester{#2}%
\ifx\stopper\tester
\else\footnote{#2}{#3}\expandafter\UCnoFootnote
\fi}
```

重复调用 `\UCnoFootnote` 将去掉定界记号 (`\expandafter` 先去掉了 `\fi`)，而且作为额外收获，这个宏对于多个作者的情形也是正确的。

### 11.9.3 用 `\futurelet` 实现可选的宏参数

`\futurelet` 的其中一个标准用法就是实现宏的可选参数。一般做法如下：

```
\def\Com{\futurelet\testchar\MaybeOptArgCom}
\def\MaybeOptArgCom{\ifx[\testchar \let\next\OptArgCom
\else \let\next\NoOptArgCom \fi \next}
\def\OptArgCom[#1]#2{ ... }\def\NoOptArgCom#1{ ... }
```

注意即便它测试的是个字符还是使用了 `\ifx`。原因当然是，如果可选参量不存在，在 `\Com` 之后可能会是一个可展开的控制序列。

现在宏 `\Com` 有一个可选参量以及一个常规参量；我们可以这样

```
\Com{argument}
```

或者这样

```
\Com[optional]{argument}
```

调用它。通常不带可选参量的调用将插入默认值：

```
\def\NoOptArgCom#1{\OptArgCom[default]{#1}}
```

这种机制在类似  $\text{\LaTeX}$  和  $\text{\LaTeX}$  的格式中广泛用到；另外可以参考 [49]。

### 11.9.4 两步宏

用户经常会觉得某个宏事实上是一个两步的过程，其中第一个宏设定好条件，而第二个宏将执行操作。

作为例子，这里有个 `\PickToEol` 宏，它的参量以行尾符定界。首先我们写出第一个不带参量的宏，它改变行尾符的类别码，然后调用第二个宏。

```
\def\PickToEol{\begingroup\catcode\^^M=12 \xPickToEol}
```

现在第二个宏将以直到行尾的所有内容作为它的参量：

```
\def\xPickToEol#1^^M{ ... #1 ... \endgroup}
```

这个定义有个问题：^^M 符的类别码必须为 12。我们改进一下：

```
\def\PickToEol{\begingroup\catcode\^^M=12 \xPickToEol}
{\catcode\^^M=12 %
  \gdef\xPickToEol#1^^M{ ... #1 ... \endgroup}%
}
```

其中为 \xPickToEol 的定义改变了 ^^M 的类别码。注意 \PickToEol 中的 ^^M 作为控制符号出现，此时它的类别码是无关紧要的。因此那个定义可以出现在重新定义 ^^M 的类别码的编组的外边。

### 11.9.5 注释环境

作为上述两步宏的想法的应用，也为了演示尾递归，这里是一个‘注释’环境的宏。

我们经常需要临时删除部分  $\TeX$  输入。为此我们想这样写

```
\comment
...
\endcomment
```

最简单的实现方式，

```
\def\comment#1\endcomment{}
```

有许多不足。比如，它无法妥善处理外部宏或者花括号不匹配的输入。然而，最大的不足是它将整个注释文本作为宏参量读取。这使得注释文本的长度不能超过  $\TeX$  的输入缓冲区的大小。

将注释文本逐行取出会更好些。为此我们需要编写一个递归宏

```
\def\comment#1^^M{ ... \comment }
```

为使这个定义能写出来，行尾符的类别码必须改变。和上面一样我们将有

```
\def\comment{\begingroup \catcode\^^M=12 \xcomment}
{\catcode\^^M=12 \endlinechar=-1 %
  \gdef\xcomment#1^^M{ ... \xcomment}
}
```

改变 \endlinechar 只不过是为了避免在这个定义的每行末尾加上注释符。

当然，这个过程必须在某个地方停止。为此我们考察已被取为宏参量的文本行：

```
{\catcode\^^M=12 \endlinechar=-1 %
  \gdef\xcomment#1^^M{\def\test{#1}
    \ifx\test\endcomment \let\next=\endgroup
    \else \let\next=\xcomment \fi
    \next}
}
```

而我们必须定义 `\endcomment` 如下：

```
\def\endcomment{\endcomment}
```

这个命令将永远不会被执行；它只是用于测试是否已经到达环境的结尾处。

我们也许想注释掉语法不正确的文本。因此在注释时我们切换到抄录模式。下面的宏是 plain  $\text{\TeX}$  给出的：

```
\def\dospecials{\do\ \do\\\do\{\do\}\do\$\do\&\%
\do\#\do\~\do\^~K\do\_ \do\^~A\do\%\do\~}
```

我们把它用到 `\comment` 的定义中：

```
\def\makeinnocent#1{\catcode`#1=12 }
\def\comment{\begingroup
\let\do=\makeinnocent \dospecials
\endlinechar`^^M \catcode`^^M=12 \xcomment}
```

这种方法除了能注释掉语法错误比如花括号不匹配的文本外，还能处理外部宏。之前的 `\xcomment` 实现在注释文本包含外部宏时将会产生  $\text{\TeX}$  错误。

然而，使用抄录模式给结束环境造成了问题。注释的最后一行现在不是控制序列 `\endcomment`，而是由一串字符组成。这样我们就必须测试这个字符串：

```
{\escapechar=-1
\xdef\endcomment{\string\endcomment}
}
```

代码 `\string\` 给出一个反斜线。我们不能这样写

```
\edef\endcomment{\string\endcomment}
```

因为这样写单词 `endcomment` 各字母的类别码将为 12，而与它们在注释最后一行中的类别码 11 不相等。

## 第 12 章 Expansion

*Expansion* in  $\text{T}_{\text{E}}\text{X}$  is rather different from procedure calls in most programming languages. This chapter treats the commands connected with expansion, and gives a number of (non-trivial) examples.

`\relax` Do nothing.

`\expandafter` Take the next two tokens and place the expansion of the second after the first.

`\noexpand` Do not expand the next token.

`\edef` Start a macro definition; the replacement text is expanded at definition time.

`\aftergroup` Save the next token for insertion after the current group.

`\afterassignment` Save the next token for execution after the next assignment or macro definition.

`\the` Expand the value of various quantities in  $\text{T}_{\text{E}}\text{X}$  into a string of character tokens.

### 12.1 Introduction

$\text{T}_{\text{E}}\text{X}$ 's expansion processor accepts a stream of tokens coming out of the input processor, and its result is again a stream of tokens, which it feeds to the execution processor. For the input processor there are two kinds of tokens: expandable and unexpandable ones. The latter category is passed untouched, and it contains largely assignments and typesettable material; the former category is expanded, and the result of that expansion is examined anew.

## 12.2 Ordinary expansion

The following list gives those constructs that are expanded, unless expansion is inhibited:

- macros
- conditionals
- `\number`, `\romannumeral`
- `\string`, `\fontname`, `\jobname`, `\meaning`, `\the`
- `\csname` ... `\endcsname`
- `\expandafter`, `\noexpand`
- `\topmark`, `\botmark`, `\firstmark`, `\splitfirstmark`, `\splitbotmark`
- `\input`, `\endinput`

This is the list of all instances where expansion is inhibited:

- when  $\TeX$  is reading a token to be defined by
  - a `\let` assignment, that is, by `\let` or `\futurelet`,
  - a `\shorthand` definition, that is, by `\chardef` or `\mathchardef`, or a `\register` def, that is, `\countdef`, `\dimendef`, `\skipdef`, `\muskipdef`, or `\toksdef`,
  - a `\def` definition, that is a macro definition with `\def`, `\gdef`, `\edef`, or `\xdef`,
  - the `\read` and `\font` simple assignment/s
- when a `\parameter` text or macro arguments are being read; also when the replacement text of a control sequence being defined by `\def`, `\gdef`, or `\read` is being read;
- when the token list for a `\token` variable or `\uppercase`, `\lowercase`, or `\write` is being read; however, the token list for `\write` will be expanded later when it is shipped out;
- when tokens are being deleted during error recovery;
- when part of a conditional is being skipped;
- in two instances when  $\TeX$  has to know what follows
  - after a left quote in a context where that is used to denote an integer (thus in `\catcode`a` the `a` is not expanded), or
  - after a math shift character that begins math mode to see whether another math shift character follows (in which case a display opens);
- when an alignment preamble is being scanned; however, in this case a to-

ken preceded by `\span` and the tokens in a `\tabskip` assignment are still expanded.

## 12.3 Reversing expansion order

Every once in a while you need to change the normal order of expansion of tokens.  $\text{\TeX}$  provides several mechanisms for this. Some of the control sequences in this section are not strictly concerned with expansion.

### 12.3.1 One step expansion: `\expandafter`

The most obvious tool for reversed expansion order is `\expandafter`. The sequence

```
\expandafter⟨token1⟩⟨token2⟩
```

expands to

```
⟨token1⟩⟨the expansion of token2⟩
```

Note the following.

- If `⟨token2⟩` is a macro, it is replaced by its replacement text, not by its final expansion. Thus, if

```
\def\tokentwo{\ifsomecondition this \else that \fi}
\def\tokenone#1{ ... }
```

the call

```
\expandafter\tokenone\tokentwo
```

will give `\ifsomecondition` as the parameter to `\tokenone`:

```
\tokenone #1-> ...
#1<-\ifsomecondition
```

- If the `\tokentwo` is a macro with one or more parameters, sufficiently many subsequent tokens will be absorbed to form the replacement text.

### 12.3.2 Total expansion: `\edef`

Macros are usually defined by `\def`, but for the cases where one wants the replacement text to reflect current conditions (as opposed to conditions at the time of the call), there is an ‘expanding define’, `\edef`, which expands everything in the replacement text, before assigning it to the control sequence.

例子:

```
\edef\modedef{This macro was defined in
  \ifvmode vertical\else \ifmmode math
    \else horizontal\fi\fi' mode}
```

*The mode tests will be executed at definition time, so the replacement text will be a single string.*

*As a more useful example, suppose that in a file that will be `\input` the category code of the `@` will be changed. One could then write*

```
\edef\restorecat{\catcode`@=\the\catcode`@}
```

*at the start, and*

```
\restorecat
```

*at the end. See page 144 for a fully worked-out version of this.*

Contrary to the ‘one step expansion’ of `\expandafter`, the expansion inside an `\edef` is complete: it goes on until only unexpandable character and control sequence tokens remain. There are two exceptions to this total expansion:

- any control sequence preceded by `\noexpand` is not expanded, and,
- if `\sometokenlist` is a token list, the expression

```
\the\sometokenlist
```

is expanded to the contents of the list, but the contents are not expanded any further (see Chapter 14 for examples).

On certain occasions the `\edef` can conveniently be abused, in the sense that one is not interested in defining a control sequence, but only in the result of the expansion. For example, with the definitions

```
\def\othermacro{\ifnum1>0 {this}\else {that}\fi}
\def\somemacro#1{ ... }
```

the call

```
\expandafter\somemacro\othermacro
```

gives the parameter assignment

```
#1<-\ifnum
```

This can be repaired by calling

```
\edef\next{\noexpand\somemacro\othermacro}\next
```

Conditionals are completely expanded inside an `\edef`, so the replacement text of `\next` will consist of the sequence

```
\somemacro{this}
```

and a subsequent call to `\next` executes this statement.

### 12.3.3 `\afterassignment`

The `\afterassignment` command takes one token and sets it aside for insertion in the token stream after the next assignment or macro definition. If the first assignment is of a box to a box register, the token will be inserted right after the opening brace of the box (see page 66).

Only one token can be saved this way; a subsequent token saved by `\afterassignment` will override the first.

Let us consider an example of the use of `\afterassignment`. It is often desirable to have a macro that will

- assign the argument to some variable, and then
- do a little calculation, based on the new value of the variable.

The following example illustrates the straightforward approach:

```
\def\setfontsize#1{\thefontsize=#1pt\relax
  \baselineskip=1.2\thefontsize\relax}
\setfontsize{10}
```

A more elegant solution is possible using `\afterassignment`:

```
\def\setbaselineskip
  {\baselineskip=1.2\thefontsize\relax}
\def\fontsize{\afterassignment\setbaselineskip
  \thefontsize}
\fontsize=10pt
```

Now the macro looks like an assignment: the equals sign is even optional. In reality its expansion ends with a variable to be assigned to. The control sequence `\setbaselineskip` is saved for execution after the assignment to `\thefontsize`.

Examples of `\afterassignment` in plain  $\TeX$  are the `\magnification` and `\hglue` macros. See [31] for another creative application of this command.

### 12.3.4 `\aftergroup`

Several tokens can be saved for insertion after the current group with an `\aftergroup(token)`

command. The tokens are inserted after the group in the sequence the `\aftergroup` commands were given in. The group can be delimited either by implicit or explicit braces, or by `\begingroup` and `\endgroup`.

例子:

```
{\aftergroup\ a \aftergroup\ b}
```

*is equivalent to*

```
\ a \ b
```



This command has many applications. One can be found in the `\textvcenter` macro on page 146; another one is provided by the footnote mechanism of plain  $\text{T}_{\text{E}}\text{X}$ .

The footnote command of plain  $\text{T}_{\text{E}}\text{X}$  has the layout

```
\footnote{footnote symbol}{footnote text}
```

which looks like a macro with two arguments. However, it is undesirable to scoop up the footnote text, since this precludes for instance category code changes in the footnote.

What happens in the plain footnote macro is (globally) the following.

- The `\footnote` command opens an insert,
 

```
\def\footnote#1{ ...#1... %treat the footnote sign
  \insert\footins\bgroup
```
- In the insert box a group is opened, and an `\aftergroup` command is given to close off the insert properly:

```
\bgroup\aftergroup\@foot
```

This command is meant to wind up after the closing brace of the text that the user typed to end the footnote text; the opening brace of the user's footnote text must be removed by

```
\let\next=}%end of definition \footnote
```

which assigns the next token, the brace, to `\next`.

- The footnote text is set as ordinary text in this insert box.
- After the footnote the command `\@foot` defined by

```
\def\@foot{\strut\egroup}
```

will be executed.

## 12.4 Preventing expansion

Sometimes it is necessary to prevent expansion in a place where it normally occurs. For this purpose the control sequences `\string` and `\noexpand` are available.

The use of `\string` is rather limited, since it converts a control sequence token into a string of characters, with the value of `\escapechar` used for the character of category code 0. It is eminently suitable for use in a `\write`, in order to output a control sequence name (see also Chapter 30); for another application see the explanation of `\newif` in Chapter 13.

All characters resulting from `\string` have category code 12, 'other', except

for space characters; they receive code 10. See also Chapter 3.

### 12.4.1 `\noexpand`

The `\noexpand` command is expandable, and its expansion is the following token. The meaning of that token is made temporarily equal to `\relax`, so that it cannot be expanded further.

For `\noexpand` the most important application is probably in `\edef` commands (but in write statements it can often replace `\string`). Consider as an example

```
\edef\one{\def\noexpand\two{\the\prevdepth}}
```

Without the `\noexpand`,  $\TeX$  would try to expand `\two`, thus giving an ‘undefined control sequence’ error.

A (rather pointless) illustration of the fact that `\noexpand` makes the following token effectively into a `\relax` is

```
\def\a{b}
\noexpand\a
```

This will not produce any output, because the effect of the `\noexpand` is to make the control sequence `\a` temporarily equal to `\relax`.

### 12.4.2 `\noexpand` and active characters

The combination `\noexpand⟨token⟩` is equivalent to `\relax`, even if the token is an active character. Thus,

```
\csname\noexpand~\endcsname
```

will not be the same as `\char`\~`. Instead it will give an error message, because unexpandable commands – such as `\relax` – are not allowed to appear in between `\csname` and `\endcsname`. The solution is to use `\string` instead; see page 144 for an example.

In another context, however, the sequence `\noexpand⟨active character⟩` is equivalent to the character, but in unexpandable form. This is when the conditionals `\if` and `\ifcat` are used (for an explanation of these, see Chapter 13). Compare

```
\if\noexpand~\relax % is false
```

where the character code of the tilde is tested, with

```
\def\a{ ... } \if\noexpand\a\relax % is true
```

where two control sequences are tested.

## 12.5 \relax

The control sequence `\relax` cannot be expanded, but when it is executed nothing happens.

This statement sounds a bit paradoxical, so consider an example. Let counters

```
\newcount\MyCount
\newcount\MyOtherCount \MyOtherCount=2
```

be given. In the assignment

```
\MyCount=1\number\MyOtherCount3\relax4
```

the command `\number` is expandable, and `\relax` is not. When  $\text{\TeX}$  constructs the number that is to be assigned it will expand all commands, either until a non-digit is found, or until an unexpandable command is encountered. Thus it reads the 1; it expands the sequence `\number\MyOtherCount`, which gives 2; it reads the 3; it sees the `\relax`, and as this is unexpandable it halts. The number to be assigned is then 123, and the whole call has been expanded into

```
\MyCount=123\relax4
```

Since the `\relax` token has no effect when it is executed, the result of this line is that 123 is assigned to `\MyCount`, and the digit 4 is printed.

Another example of how `\relax` can be used to indicate the end of a command is

```
\everypar{\hspace 0cm plus 1fil }
\indent Later that day, ...
```

This will be misunderstood:  $\text{\TeX}$  will see

```
\hspace 0cm plus 1fil L
```

and `fil L` is a valid, if bizarre, way of writing `fill` (see Chapter 36). One remedy is to write

```
\everypar{\hspace 0cm plus 1fil\relax}
```

### 12.5.1 \relax and \csname

If a `\csname ... \endcsname` command forms the name of a previously undefined control sequence, that control sequence is made equal to `\relax`, and the whole statement is also equivalent to `\relax` (see also page 121).

However, this assignment of `\relax` is only local:

```
{\xdef\test{\expandafter\noexpand\csname xx\endcsname}}
\test
```

gives an error message for an undefined control sequence `\xx`.

Consider as an example the  $\text{\LaTeX}$  environments, which are delimited by

```
\begin{...} ... \end{...}
```

The begin and end commands are (in essence) defined as follows:

```
\def
\begin#1{\begingroup\csname#1\endcsname}
\def\end#1{\csname end#1\endcsname \endgroup}
```

Thus, for the list environment the commands `\list` and `\endlist` are defined, but any command can be used as an environment name, even if no corresponding `\end...` has been defined. For instance,

```
\begin{it} ... \end{it}
```

is equivalent to

```
\begingroup\it ... \relax\endgroup
```

See page 112 for the rationale behind using `\begingroup` and `\endgroup` instead of `\bgroup` and `\egroup`.

### 12.5.2 Preventing expansion with `\relax`

Because `\relax` cannot be expanded, a control sequence can be prevented from being expanded (for instance in an `\edef` or a `\write`) by making it temporarily equal to `\relax`:

```
{\let\somemacro=\relax \write\outfile{\somemacro}}
```

will write the string ‘`\somemacro`’ to an output file. It would write the expansion of the macro `\somemacro` (or give an error message if the macro is undefined) if the `\let` statement had been omitted.

### 12.5.3 $\TeX$ inserts a `\relax`

$\TeX$  itself inserts `\relax` on some occasions. For instance, `\relax` is inserted if  $\TeX$  encounters an `\or`, `\else`, or `\fi` while still determining the extent of the test.

例子:

```
\ifvoid1\else ... \fi
```

*is changed into*

```
\ifvoid1\relax \else ... \fi
```

*internally.*

Similarly, if one of the tests `\if`, `\ifcat` is given only one comparand, as in `\if1\else ...`

a `\relax` token is inserted. Thus this test is equivalent to

```
\if1\relax\else ...
```

Another place where `\relax` is used is the following. While a control sequence is being defined in a `<shorthand definition>` – that is, a `<registerdef>` or `\chardef` or `\mathchardef` – its meaning is temporarily made equal to `\relax`. This makes it possible to write `\chardef\foo=123\foo`.

#### 12.5.4 The value of non-macros; `\the`

Expansion is a precisely defined activity in  $\TeX$ . The full list of tokens that can be expanded was given above. Other tokens than those in the above list may have an ‘expansion’ in an informal sense. For instance one may wish to ‘expand’ the `\parindent` into its value, say 20pt.

Converting the value of (among others) an `<integer parameter>`, a `<glue parameter>`, `<dimen parameter>` or a `<token parameter>` into a string of character tokens is done by the expansion processor. The command `\the` is expanded whenever expansion is not inhibited, and it takes the value of various sorts of parameters. Its result (in most cases) is a string of tokens of category 12, except that spaces have category code 10.

Here is the list of everything that can be prefixed with `\the`.

**<parameter> or <register>** If the parameter or register is of type integer, glue, dimen or muglue, its value is given as a string of character tokens; if it is of type token list (for instance `\everypar` or `\toks5`), the result is a string of tokens. Box registers are excluded here.

**<codename><8-bit number>** See page 47.

**<special register>** The integer registers `\prevgraf`, `\deadcycles`, `\insertpenalties`, `\inputlineno`, `\badness`, `\parshape`, `\spacefactor` (only in horizontal mode), or `\prevdepth` (only in vertical mode). The dimension registers `\pagetotal`, `\pagegoal`, `\pagestretch`, `\pagefilstretch`, `\pagefillstretch`, `\pagefilllstretch`, `\pageshrink`, or `\pagedepth`.

**Font properties:** `\fontdimen<parameter number><font>`, `\skewchar<font>`, `\hyphenchar<font>`.

**Last quantities:** `\lastpenalty`, `\lastkern`, `\lastskip`.

**<defined character>** Any control sequence defined by `\chardef` or `\mathchardef`; the result is the decimal value.

In some cases `\the` can give a control sequence token or list of such tokens.

**<font>** The result is the control sequence that stands for the font.

**<token variable>** Token list registers and `<token parameter>`s can be prefixed

with `\the`; the result is their contents.

Let us consider an example of the use of `\the`. If in a file that is to be `\input` the category code of a character, say the at sign, is changed, one could write

```
\edef\restorecat{\catcode`@=\the\catcode`@}
```

and call `\restorecat` at the end of the file. If the category code was 11, `\restorecat` is defined equivalent to

```
\catcode`@=11
```

See page 144 for more elaborate macros for saving and restoring catcodes.

## 12.6 Examples

### 12.6.1 Expanding after

The most obvious use of `\expandafter` is to reach over a control sequence:

```
\def\stepcounter
  #1{\expandafter\advance\csname
      #1:counter\endcsname 1\relax}
\stepcounter{foo}
```

Here the `\expandafter` lets the `\csname` command form the control sequence `\foo:counter`; after `\expandafter` is finished the statement has reduced to

```
\advance\foo:counter 1\relax
```

It is possible to reach over tokens other than control sequences: in

```
\uppercase\expandafter{\romannumeral \year}
```

it expands `\romannumeral` on the other side of the opening brace.

You can expand after two control sequences:

```
\def\globalstepcounter
  #1{\expandafter\global\expandafter\advance
      \csname #1:counter\endcsname 1\relax}
```

If you think of `\expandafter` as reversing the evaluation order of two control sequences, you can reverse three by

```
\expandafter\expandafter\expandafter\ a\expandafter\ b\ c
```

which reaches across the three control sequences

```
\expandafter          \ a          \ b
```

to expand `\ c` first.

There is even an unexpected use for `\expandafter` in conditionals; with

```
\def\bold#1{\bf #1}
```

the sequence

```
\ifnum1>0 \bold \fi {word}
```

will not give a boldface ‘word’, but

```
\ifnum1>0 \expandafter\bold \fi {word}
```

will. The `\expandafter` lets  $\TeX$  see the `\fi` and remove it before it tackles the macro `\bold` (see also page 155).

### 12.6.2 Defining inside an `\edef`

There is one  $\TeX$  command that is executed instead of expanded that is worth pointing out explicitly: the primitive command `\def` (and all other `<def>` commands) is not expanded.

Thus the call

```
\edef\next{\def\thing{text}}
```

will give an ‘undefined control sequence’ for `\thing`, even though after `\def` expansion is ordinarily inhibited (see page 131). After

```
\edef\next{\def\noexpand\thing{text}}
```

the ‘meaning’ of `\next` will be

```
macro: \def \thing {text}
```

The definition

```
\edef\next{\def\noexpand\thing{text}\thing}
```

will again give an ‘undefined control sequence’ for `\thing` (this time on its second occurrence), as it will only be defined when `\next` is called, not when `\next` is defined.

### 12.6.3 Expansion and `\write`

The argument token list of `\write` is treated in much the same way as the replacement text of an `\edef`; that is, expandable control sequences and active characters are completely expanded. Unexpandable control sequences are treated by `\write` as if they are prefixed by `\string`.

Because of the expansion performed by `\write`, some care has to be taken when outputting control sequences with `\write`. Even more complications arise from the fact that the expansion of the argument of `\write` is only performed when it is shipped out. Here follows a worked-out example.

Suppose `\somecs` is a macro, and you want to write the string

```
\def\othercs{the expansion of \somecs}
```

to a file.

The first attempt is

```
\write\myfile{\def\othercs{\somecs}}
```

This gives an error ‘undefined control sequence’ for `\othercs`, because the `\write` will try to expand that token. Note that the `\somecs` is also expanded, so that part is right.

The next attempt is

```
\write\myfile{\def\noexpand\othercs{\somecs}}
```

This is almost right, but not quite. The statement written is

```
\def\othercs{expansion of \somecs}
```

which looks right.

However, `writes` – and the expansion of their argument – are not executed on the spot, but saved until the part of the page on which they occur is shipped out (see Chapter 30). So, in the meantime, the value of `\somecs` may have changed. In other words, the value written may not be the value at the time the `\write` command was given. Somehow, therefore, the current expansion must be inserted in the write command.

The following is an attempt at repair:

```
\edef\act{\write\myfile{\def\noexpand\othercs{\somecs}}}  
\act
```

Now the write command will be

```
\write\myfile{\def\othercs{value of \somecs}}
```

The `\noexpand` prevented the `\edef` from expanding the `\othercs`, but after the definition it has disappeared, so that execution of the write will again give an undefined control sequence. The final solution is

```
\edef\act{\write\myfile  
          {\def \noexpand\noexpand \noexpand\othercs{\somecs}}}  
\act
```

In this case the write command caused by the expansion of `\act` will be

```
\write\myfile{\def\noexpand\othercs{current value of \somecs}}
```

and the string actually written is

```
\def\othercs{current value of \somecs}
```

This mechanism is the basis for cross-referencing macros in several macro packages.

#### 12.6.4 Controlled expansion inside an `\edef`

Sometimes you may need an `\edef` to evaluate current conditions, but you want to expand something in the replacement text only to a certain level. Suppose that

```
\def\a{\b} \def\b{c} \def\d{e} \def{e{f}
```



is given, and you want to define `\g` as `\a` expanded one step, followed by `\d` fully expanded. The following works:

```
\edef\g{\expandafter\noexpand\a \d}
```

Explanation: the `\expandafter` reaches over the `\noexpand` to expand `\a` one step, after which the sequence `\noexpand\b` is left.

This trick comes in handy when you need to construct a control sequence with `\csname` inside an `\edef`. The following sequence inside an `\edef`

```
\expandafter\noexpand\csname name\endcsname
```

will expand exactly to `\name`, but not further. As an example, suppose

```
\def\condition{true}
```

has been given, then

```
\edef\setmycondition{\expandafter\noexpand
\csname mytest\condition\endcsname}
```

will let `\setmycondition` expand to `\mytesttrue`.

### 12.6.5 Multiple prevention of expansion

As was pointed out above, prefixing a command with `\noexpand` prevents its expansion in commands such as `\edef` and `\write`. However, if a sequence of tokens passes through more than one expanding command stronger measures are needed.

The following trick can be used: in order to protect a command against expansion it can be prefixed with `\protect`. During the stages of processing where expansion is not desired the definition of `\protect` is

```
\def\protect{\noexpand\protect\noexpand}
```

Later on, when the command is actually needed, `\protect` is defined as

```
\def\protect{}
```

Why does this work? The expansion of

```
\protect\somecs
```

is at first

```
\noexpand\protect\noexpand\somecs
```

Inside an `\edef` this sequence is expanded further, and the subsequent expansion is

```
\protect\somecs
```

That is, the expansion is equal to the original sequence.

### 12.6.6 More examples with `\relax`

Above, a first example was given in which `\relax` served to prevent  $\text{\TeX}$  from scanning too far. Here are some more examples, using `\relax` to bound numbers.

After

```
\countdef\pageno=0 \pageno=1
\def\Par{\par\penalty200}
```

the sequence

```
\Par\number\pageno
```

is misunderstood as

```
\par\penalty2001
```

In this case it is sufficient to define

```
\def\Par{\par\penalty200 }
```

as an `<optional space>` is allowed to follow a number.

Sometimes, however, such a simple escape is not possible. Consider the definition

```
\def\ifequal#1#2{\ifnum#1=#2 1\else 0\fi}
```

The question is whether the space after `#2` is necessary, superfluous, or simply wrong. Calls such as `\ifequal{27}{28}` that compare two numbers (denotations) will correctly give 1 or 0, and the space is necessary to prevent misinterpretation.

However, `\ifequal\somecounter\othercounter` will give 1 if the counters are equal; in this case the space could have been dispensed with. The solution that works in both cases is

```
\def\ifequal#1#2{\ifnum#1=#2\relax 1\else 0\fi}
```

Note that `\relax` is not expanded, so

```
\edef\foo{1\ifequal\counta\countb}
```

will define `\foo` as either `1\relax1` or `10`.

### 12.6.7 Example: category code saving and restoring

In many applications it is necessary to change the category code of a certain character during the execution of some piece of code. If the writer of that code is also the writer of the surrounding code, s/he can simply change the category code back and forth. However, if the surrounding code is by another author, the value of the category code will have to be stored and restored.

Thus one would like to write

```

\storecat@
... some code ...
\restorecat@

```

or maybe

```

\storecat\%

```

for characters that are possibly a comment character (or ignored or invalid).

The basic idea is to define

```

\def\storecat#1{%
  \expandafter\edef\csname restorecat#1\endcsname
    {\catcode`#1=\the\catcode`#1}}

```

so that, for instance, `\storecat$` will define the single control sequence ‘`\restorecat$`’ (one control sequence) as

```

\catcode`$=3

```

The macro `\restorecat` can then be implemented as

```

\def\restorecat#1{%
  \csname restorecat#1\endcsname}

```

Unfortunately, things are not so simple.

The problems occur with active characters, because these are expanded inside the `\csname ... \endcsname` pairs. One might be tempted to write `\noexpand#1` everywhere, but this is wrong. As was explained above, this is essentially equal to `\relax`, which is unexpandable, and will therefore lead to an error message when it appears between `\csname` and `\endcsname`. The proper solution is then to use `\string#1`. For the case where the argument was given as a control symbol (for example `\%`), the escape character has to be switched off for a while.

Here are the complete macros. The `\storecat` macro gives its argument a default category code of 12.

```

\newcount\tempcounta % just a temporary
\def\csarg#1#2{\expandafter#1\csname#2\endcsname}
\def\storecat#1%
  {\tempcounta\escapechar \escapechar=-1
   \csarg\edef{restorecat\string#1}%
     {\catcode`\string#1=
      \the\catcode\expandafter`\string#1}%
   \catcode\expandafter`\string#1=12\relax
   \escapechar\tempcounta}
\def\restorecat#1%
  {\tempcounta\escapechar \escapechar=-1
   \csname restorecat\string#1\endcsname
   \escapechar\tempcounta}

```

### 12.6.8 Combining `\aftergroup` and boxes

At times, one wants to construct a box and immediately after it has been constructed to do something with it. The `\aftergroup` command can be used to put both the commands creating the box, and the ones handling it, in one macro.

As an example, here is a macro `\textvcenter` which defines a variant of the `\vcenter` box (see page 221) that can be used outside math mode.

```
\def\textvcenter
  {\hbox \bgroup$\everyvbox{\everyvbox{}}%
   \aftergroup$\aftergroup\egroup}\vcenter}
```

The idea is that the macro inserts `\hbox {` \$, and that the matching `}` gets inserted by the `\aftergroup` commands. In order to get the `\aftergroup` commands inside the box, an `\everyvbox` command is used.

This macro can even be used with a `\box` specification (see page 56), for example

```
\textvcenter spread 8pt{\hbox{a}\vfil\hbox{b}}
```

and because it is really just an `\hbox`, it can also be used in a `\setbox` assignment.

### 12.6.9 More expansion

There is a particular charm to macros that work purely by expansion. See the articles by [11], [16], and [32].

## 第 13 章 Conditionals

*Conditionals* are an indispensable tool for powerful macros. T<sub>E</sub>X has a large repertoire of conditionals for querying such things as category codes or processing modes. This chapter gives an inventory of the various conditionals, and it treats the evaluation of conditionals in detail.

`\if` Test equality of character codes.  
`\ifcat` Test equality of category codes.  
`\ifx` Test equality of macro expansion, or equality of character code and category code.  
`\ifcase` Enumerated case statement.  
`\ifnum` Test relations between numbers.  
`\ifodd` Test whether a number is odd.  
`\ifhmode` Test whether the current mode is (possibly restricted) horizontal mode.  
`\ifvmode` Test whether the current mode is (possibly internal) vertical mode.  
`\ifmmode` Test whether the current mode is (possibly display) math mode.  
`\ifinner` Test whether the current mode is an internal mode.  
`\ifdim` Compare two dimensions.  
`\ifvoid` Test whether a box register is empty.  
`\ifhbox` Test whether a box register contains a horizontal box.  
`\ifvbox` Test whether a box register contains a vertical box.  
`\ifeof` Test for end of input stream or non-existence of file.  
`\iftrue` A test that is always true.  
`\iffalse` A test that is always false.  
`\fi` Closing delimiter for all conditionals.  
`\else` Select `\false text` of a conditional or default case of `\ifcase`.

`\or` Separator for entries of an `\ifcase`.

`\newif` Create a new test.

## 13.1 The shape of conditionals

Conditionals in  $\text{\TeX}$  have one of the following two forms

```
\if...<test tokens><true text>\fi
```

```
\if...<test tokens><true text>\else<false text>\fi
```

where the `<test tokens>` are zero or more tokens, depending on the particular conditional; the `<true text>` is a series of tokens to be processed if the test turns out true, and the `<false text>` is a series of tokens to be processed if the test turns out false. Both the `<true text>` and the `<false text>` can be empty.

The exact process of how  $\text{\TeX}$  expands conditionals is treated below.

## 13.2 Character and control sequence tests

Three tests exist for testing character tokens and control sequence tokens.

### 13.2.1 `\if`

Equality of character codes can be tested by

```
\if<token1><token2>
```

In order to allow the tokens to be control sequences,  $\text{\TeX}$  assigns character code 256 to control sequences, the lowest positive number that is not the character code of a character token (remember that the legal character codes are 0–255).

Thus all control sequences are equal as far as `\if` is concerned, and they are unequal to all character tokens. As an example, this fact can be used to define

```
\def\ifIsControlSequence#1{\if\noexpand#1\relax}
```

which tests whether a token is a control sequence token instead of a character token (its result is unpredictable if the argument is a `{...}` group).

After `\if`  $\text{\TeX}$  will expand until two unexpandable tokens are obtained, so it is necessary to prefix expandable control sequences and active characters with `\noexpand` when testing them with `\if`.

例子: *After*

```
\catcode`\b=13 \catcode`\c=13 \def b{a} \def c{a} \let\d=a
```

*we find that*

`\if bc` is true, because both `b` and `c` expand to `a`,  
`\if\noexpand b\noexpand c` is false, and  
`\if b\d` is true because `b` expands to the character `a`, and `\d` is an implicit character token `a`.

### 13.2.2 `\ifcat`

The `\if` test ignores category codes; these can be tested by

`\ifcat⟨token1⟩⟨token2⟩`

This test is a lot like `\if`:  $\TeX$  expands after it until unexpandable tokens remain. For this test control sequences are considered to have category code 16 (ordinarily, category codes are in the range 0–15), which makes them all equal to each other, and different from all character tokens.

### 13.2.3 `\ifx`

Equality of tokens is tested in a stronger sense than the above by

`\ifx⟨token1⟩⟨token2⟩`

- Character tokens are equal for `\ifx` if they have the same character code and category code.
- Control sequence tokens are equal if they represent the same  $\TeX$  primitive, or have been similarly defined by `\font`, `\countdef`, or some such. For example,

```
\let\boxhor=\hbox \ifx\boxhor\hbox %is true
\font\A=cmr10 \font\B=cmr10 \ifx\A\B %is true
```

- Control sequences are also equal if they are macros with the same parameter text and replacement text, and the same status with respect to `\outer` and `\long`. For example,

```
\def\A{z} \def\B{z} \def\c1{z} \def\d{\A}
\ifx\A\B %is true
\ifx\A\c %is false
\ifx\A\d %is false
```

Tokens following this test are not expanded.

By way of example of the use of `\ifx` consider string testing. A simple implementation of string testing in  $\TeX$  is as follows:

```
\def\ifEqString#1#2{\def\testa{#1}\def\testb{#2}%
\ifx\testa\testb}
```

The two strings are used as the replacement text of two macros, and equality of these macros is tested. This is about as efficient as string testing can get:  $\TeX$

will traverse the definition texts of the macros `\testa` and `\testb`, which has precisely the right effect.

As another example, one can test whether a control sequence is defined by

```
\def\ifUndefinedCs#1{\expandafter
  \ifx\csname#1\endcsname\relax}
\ifUndefinedCs{parindent} %is not true
\ifUndefinedCs{undefined} %is (one hopes) true
```

This uses the fact that a `\csname... \endcsname` command is equivalent to `\relax` if the control sequence has not been defined before. Unfortunately, this test also turns out true if a control sequence has been `\let` to `\relax`.

### 13.3 Mode tests

In order to determine in which of the six modes (see Chapter 6)  $\TeX$  is currently operating, the tests `\ifhmode`, `\ifvmode`, `\ifmmode`, and `\ifinner` are available.

- `\ifhmode` is true if  $\TeX$  is in horizontal mode or restricted horizontal mode.
- `\ifvmode` is true if  $\TeX$  is in vertical mode or internal vertical mode.
- `\ifmmode` is true if  $\TeX$  is in math mode or display math mode.

The `\ifinner` test is true if  $\TeX$  is in any of the three internal modes: restricted horizontal mode, internal vertical mode, and non-display math mode.

See also chapter 6.

### 13.4 Numerical tests

Numerical relations between  $\langle \text{number} \rangle$ s can be tested with

```
\ifnum $\langle \text{number}_1 \rangle$  $\langle \text{relation} \rangle$  $\langle \text{number}_2 \rangle$ 
```

where the relation is a character `<`, `=`, or `>`, of category 12.

Quantities such as glue can be used as a number here through the conversion to scaled points, and  $\TeX$  will expand in order to arrive at the two  $\langle \text{number} \rangle$ s.

Testing for odd or even numbers can be done with `\ifodd`: the test

```
\ifodd $\langle \text{number} \rangle$ 
```

is true if the  $\langle \text{number} \rangle$  is odd.



## 13.5 Other tests

### 13.5.1 Dimension testing

Relations between  $\langle\text{dimen}\rangle$  values (Chapter 8) can be tested with `\ifdim` using the same three relations as in `\ifnum`.

### 13.5.2 Box tests

Contents of box registers (Chapter 5) can be tested with

`\ifvoid $\langle$ 8-bit number $\rangle$`

which is true if the register contains no box,

`\ifhbox $\langle$ 8-bit number $\rangle$`

which is true if the register contains a horizontal box, and

`\ifvbox $\langle$ 8-bit number $\rangle$`

which is true if the register contains a vertical box.

### 13.5.3 I/O tests

The status of input streams (Chapter 30) can be tested with the end-of-file test `\ifeof $\langle$ number $\rangle$` , which is only false if the number is in the range 0–15, and the corresponding stream is open and not fully read. In particular, this test is true if the file name connected to this stream (through `\openin`) does not correspond to an existing file. See the example on page 271.

### 13.5.4 Case statement

The  $\text{\TeX}$  case statement is called `\ifcase`; its syntax is

`\ifcase $\langle$ number $\rangle$  $\langle$ case $_0$  $\rangle$ \or... \or $\langle$ case $_n$  $\rangle$ \else $\langle$ other cases $\rangle$ \fi`

where for  $n$  cases there are  $n - 1$  `\or` control sequences. Each of the  $\langle\text{case}_i\rangle$  parts can be empty, and the `\else $\langle$ other cases $\rangle$`  part is optional.

### 13.5.5 Special tests

The tests `\iftrue` and `\iffalse` are always true and false respectively. They are mainly useful as tools in macros.

For instance, the sequences

`\iftrue{\else}\fi`

and

```
\iffalse{\else}\fi
```

yield a left and right brace respectively, but they have balanced braces, so they can be used inside a macro replacement text.

The `\newif` macro, treated below, provides another use of `\iftrue` and `\iffalse`. On page 260 of the  $\TeX$  book these control sequences are also used in an interesting manner.

## 13.6 The `\newif` macro

The plain format defines an (outer) macro `\newif` by which the user can define new conditionals. If the user defines

```
\newif\iffoo
```

$\TeX$  defines three new control sequences, `\footrue` and `\foofalse` with which the user can set the condition, and `\iffoo` which tests the ‘foo’ condition.

The macro call `\newif\iffoo` expands to

```
\def\footrue{\let\iffoo=\iftrue} \def\foofalse{\let\iffoo=\iffalse}
\foofalse
```

The actual definition, especially the part that ensures that the `\iffoo` indeed starts with `\if`, is a pretty hack. An explanation follows here. This uses concepts from Chapters 11 and 12.

The macro `\newif` starts as follows:

```
\outer\def\newif#1{\count@\escapechar \escapechar\m@ne
```

This saves the current escape character in `\count@`, and sets the value of `\escapechar` to -1. The latter action has the effect that no escape character is used in the output of `\string⟨control sequence⟩`.

An auxiliary macro `\if@` is defined by

```
{\uccode`1=`i \uccode`2=`f \uppercase{\gdef\if@12{}}}
```

Since the uppercase command changes only character codes, and not category codes, the macro `\if@` now has to be followed by the characters `if` of category 12. Ordinarily, these characters have category code 11. In effect this macro then eats these two characters, and  $\TeX$  complains if they are not present.

Next there is a macro `\@if` defined by

```
\def\@if#1#2{\csname\expandafter\if@\string#1#2\endcsname}
```

which will be called like `\@if\iffoo{true}` and `\@if\iffoo{false}`.

Let us examine the call `\@if\iffoo{true}`.

- The `\expandafter` reaches over the `\if@` to expand `\string` first. The part `\string\iffoo` expands to `iffoo` because the escape character is not printed, and all characters have category 12.

- The `\if@` eats the first two characters `i12f12` of this.
- As a result, the final expansion of `\@if\iffoo{true}` is then

```
\csname footrue\endcsname
```

Now we can treat the relevant parts of `\newif` itself:

```
\expandafter\expandafter\expandafter
\edef\@if#1{true}{\let\noexpand#1=\noexpand\iftrue}%
```

The three `\expandafter` commands may look intimidating, so let us take one step at a time.

- One `\expandafter` is necessary to reach over the `\edef`, such that `\@if` will expand:

```
\expandafter\edef\@if\iffoo{true}
```

gives

```
\edef\csname footrue\endcsname
```

- Then another `\expandafter` is necessary to activate the `\csname`:

```
\expandafter \expandafter \expandafter \edef \@if ...
%   new           old           new
```

- This makes the final expansion

```
\edef\footrue{\let\noexpand\iffoo=\noexpand\iftrue}
```

After this follows a similar statement for the false case:

```
\expandafter\expandafter\expandafter
\edef\@if#1{false}{\let\noexpand#1=\noexpand\iffalse}%
```

The conditional starts out false, and the escape character has to be reset:

```
\@if#1{false}\escapechar\count@}
```

## 13.7 Evaluation of conditionals

$\text{\TeX}$ 's conditionals behave differently from those in ordinary programming languages. In many instances one may not notice the difference, but in certain contexts it is important to know precisely the *evaluation of conditionals* proceeds.

When  $\text{\TeX}$  evaluates a conditional, it first determines what is to be tested. This in itself may involve some expansion; as we saw in the previous chapter, only after an `\ifx` test does  $\text{\TeX}$  not expand. After all other tests  $\text{\TeX}$  will expand tokens until the extent of the test and the tokens to be tested have been determined. On the basis of the outcome of this test the  $\langle$ true text $\rangle$  and the  $\langle$ false text $\rangle$  are either expanded or skipped.

For the processing of the parts of the conditional let us consider some cases separately.

- `\if... \fi` and the result of the test is false. After the test  $\TeX$  will start skipping material without expansion, without counting braces, but balancing nested conditionals, until a `\fi` token is encountered. If the `\fi` is not found an error message results at the end of the file:

Incomplete `\if...`; all text was ignored after line ...

where the line number indicated is that of the line where  $\TeX$  started skipping, that is, where the conditional occurred.

- `\if... \else ... \fi` and the result of the test is false. Any material in between the condition and the `\else` is skipped without expansion, without counting braces, but balancing nested conditionals.

The `\fi` token can be the result of expansion; if it never turns up  $\TeX$  will give a diagnostic message

`\end occurred when \if... on line ... was incomplete`

This sort of error is not visible in the output.

This point plus the previous may jointly be described as follows: after a false condition  $\TeX$  skips until an `\else` or `\fi` is found; any material in between `\else` and `\fi` is processed.

- `\if... \fi` and the result of the test is true.  $\TeX$  will start processing the material following the condition. As above, the `\fi` token may be inserted by expansion of a macro.
- `\if... \else ... \fi` and the result of the test is true. Any material following the condition is processed until the `\else` is found; then  $\TeX$  skips everything until the matching `\fi` is found.

This point plus the previous may be described as follows: after a true test  $\TeX$  starts processing material until an `\else` or `\fi` is found; if an `\else` is found  $\TeX$  skips until it finds the matching `\fi`.

## 13.8 Assorted remarks

### 13.8.1 The test gobbles up tokens

A common mistake is to write the following:

```
\ifnum<x>0\someaction \else\anotheraction \fi
```

which has the effect that the `\someaction` is expanded, regardless of whether the test succeeds or not. The reason for this is that  $\TeX$  evaluates the input stream until it is certain that it has found the arguments to be tested. In this case it is perfectly possible for the `\someaction` to yield a digit, so it is expanded. The remedy is to insert a space or a `\relax` control sequence after the last digit of the number to be tested.

### 13.8.2 The test wants to gobble up the `\else` or `\fi`

The same mechanism that underlies the phenomenon in the previous point can lead to even more surprising effects if  $\TeX$  bumps into an `\else`, `\or`, or `\fi` while still busy determining the extent of the test itself.

Recall that `\pageno` is a synonym for `\count0`, and consider the following examples:

```
\newcount\nct \nct=1\ifodd\pageno\else 2\fi 1
```

and

```
\newcount\nct \nct=1\ifodd\count0\else 2\fi 1
```

The first example will assign either 11 or 121 to `\nct`, but the second one will assign 1 or 121. The explanation is that in cases like the second, where an `\else` is encountered while the test still has not been delimited, a `\relax` is inserted. In the case that `\count0` is odd the result will thus be `\relax`, and the example will yield

```
\nct=1\relax2
```

which will assign 1 to `\nct`, and print 2.

### 13.8.3 Macros and conditionals; the use of `\expandafter`

Consider the following example:

```
\def\bold#1{\bf #1} \def\slant#1{\sl #1}
\ifnum1>0 \bold \else \slant \fi {some text} ...
```

This will make not only ‘some text’, but all subsequent text bold. Also, at the end of the job there will be a notice that ‘end occurred inside a group at level 1’. Switching on `\tracingmacros` reveals that the argument of `\bold` was `\else`. This means that, after expansion of `\bold`, the input stream looked like

```
\ifnum1>0 {\bf \else }\fi {some text} rest of the text
```

so the closing brace was skipped as part of the `\false text`. Effectively, then, the resulting stream is

```
{\bf {some text} rest of the text
```

which is unbalanced.

One solution to this sort of problem would be to write

```
\ifnum1>0 \let\next=\bold \else \let\next=\slant \fi \next
```

but a solution using `\expandafter` is also possible:

```
\ifnum1>0 \expandafter \bold \else \expandafter \slant \fi
```

This works, because the `\expandafter` commands let  $\text{\TeX}$  determine the boundaries of the  $\langle$ true text $\rangle$  and the  $\langle$ false text $\rangle$ .

In fact, the second solution may be preferred over the first, since conditionals are handled by the expansion processor, and the `\let` statements are tackled only by the execution processor; that is, they are not expandable. Thus the second solution will (and the first will not) work, for instance, inside an `\edef`.

Another example with `\expandafter` is the sequence

```
\def\get#1\get{ ... }
\expandafter \get \ifodd1 \ifodd3 5\fi \fi \get
```

This gives

```
#1<- \ifodd3 5\fi \fi
```

and

```
\expandafter \get \ifodd2 \ifodd3 5\fi\fi \get
```

gives

```
#1<-
```

This illustrates again that the result of evaluating a conditional is not the final expansion, but the start of the expansion of the  $\langle$ true text $\rangle$  or  $\langle$ false text $\rangle$ , depending on the outcome of the test.

A detail should be noted: with `\expandafter` it is possible that the `\else` is encountered before the  $\langle$ true text $\rangle$  has been expanded completely. This raises the question as to the exact timing of expansion and skipping. In the example

```
\def\hello{\message{Hello!}}
\ifnum1>0 \expandafter \hello \else \message{goodbye} \bye
```

the error message caused by the missing `\fi` is given without `\hello` ever having been expanded. The conclusion must be that the  $\langle$ false text $\rangle$  is skipped as soon as it has been located, even if this is at a time when the  $\langle$ true text $\rangle$  has not been expanded completely.

#### 13.8.4 Incorrect matching

$\text{\TeX}$ 's matching of `\if`, `\else`, and `\fi` is easily upset. For instance, the  $\text{\TeX}$  book warns you that you should not say

```
\let\ifabc=\iftrue
```

inside a conditional, because if this text is skipped  $\text{\TeX}$  sees at least one `\if` to be matched.

The reason for this is that when  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is skipping it recognizes all `\if...`, `\or`, `\else`, and `\fi` tokens, and everything that has been declared a synonym of such a token by `\let`. In `\let\ifabc=\iftrue`  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  will therefore at least see the `\iftrue` as the opening of a conditional, and, if the current meaning of `\ifabc` was for instance `\iffalse`, it will also be considered as the opening of a conditional statement.

As another example, if

```
\csname if\sometest\endcsname \someaction \fi
```

is skipped as part of conditional text, the `\fi` will unintentionally close the outer conditional.

It does not help to enclose such potentially dangerous constructs inside a group, because grouping is independent of conditional structure. Burying such commands inside macros is the safest approach.

Sometimes another solution is possible, however. The `\loop` macro of plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  (see page 124) is used as

```
\loop ... \if ... \repeat
```

where the `\repeat` is not an actually executable command, but is merely a delimiter:

```
\def\loop#1\repeat{ ... }
```

Therefore, by declaring

```
\let\repeat\fi
```

the `\repeat` balances the `\if...` that terminates the loop, and it becomes possible to have loops in skipped conditional text.

### 13.8.5 Conditionals and grouping

It has already been mentioned above that group nesting in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is independent of conditional nesting. The reason for this is that conditionals are handled by the expansion part of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ ; in that stage braces are just unexpandable tokens that require no special treatment. Grouping is only performed in the later stage of execution processing.

An example of this independence is now given. One may write a macro that yields part of a conditional:

```
\def\elsepart{\else \dosomething \fi}
```

The other way around, the following macros yield a left brace and a right brace respectively:

```
\def\leftbrace{\iftrue{\else}\fi}
\def\rightbrace{\iffalse{\else}\fi}
```

Note that braces in these definitions are properly nested.

### 13.8.6 A trick

In some contexts it may be hard to get rid of `\else` or `\fi` tokens in a proper manner. The above approach with `\expandafter` works only if there is a limited number of tokens involved. In other cases the following trick may provide a way out:

```
\def\hop#1\fi{\fi #1}
```

Using this as

```
\if... \hop <lots of tokens>\fi
```

will place the tokens outside the conditional. This is for instance used in [11].

As a further example of this sort of trick, consider the problem (suggested to me and solved by Alan Jeffrey) of implementing a conditional `\ifLessThan#1#2#3#4` such that the arguments corresponding to #3 or #4 result, depending on whether #1 is less than #2 or not.

The problem here is how to get rid of the `\else` and the `\fi`. The – or at least, one – solution is to scoop them up as delimiters for macros:

```
\def\ifLessThan#1#2{\ifnum#1<#2\relax\takettrue \else \takefalse \fi}
\def\takefalse\fi#1#2{\fi#2}
\def\takettrue\else\takefalse\fi#1#2{\fi#1}
```

Note that `\ifLessThan` has only two parameters (the things to be tested); however, its result is a macro that chooses between the next two arguments.

### 13.8.7 More examples of expansion in conditionals

Above, the macro `\ifEqString` was given that compares two strings:

```
\def\ifEqString#1#2%
  {\def\csa{#1}\def\csb{#2}\ifx\csa\csb }
```

However, this macro relies on `\def`, which is not an expandable command. If we need a string tester that will work, for instance, inside an `\edef`, we need some more ingenuity (this solution was taken from [11]). The basic principle of this solution is to compare the strings one character at a time. Macro delimiting by `\fi` is used; this was explained above.

First of all, the `\ifEqString` call is replaced by a sequence `\ifAllChars ... \Are ... \TheSame` and both strings are delimited by a dollar sign, which is not supposed to appear in the strings themselves.

```
\def\ifEqString
  #1#2{\ifAllChars#1$\Are#2$\TheSame}
```

The test for equality of characters first determines whether either string has ended. If both have ended, the original strings were equal; if only one has ended,



they were of unequal length, hence unequal. If neither string has ended, we test whether the first characters are equal, and if so, we make a recursive call to test the remainder of the string.

```
\def\ifAllChars#1#2\Are#3#4\TheSame
  {\if#1$\if#3$\say{true}%
    \else \say{false}\fi
   \else \if#1#3\ifRest#2\TheSame#4\else
    \say{false}\fi\fi}
\def\ifRest#1\TheSame#2\else#3\fi\fi
  {\fi\fi \ifAllChars#1\Are#2\TheSame}
```

The `\say` macro is supposed to give `\iftrue` for `\say{true}` and `\iffalse` for `\say{false}`. Observing that all calls to this macro occur two conditionals deep, we use the ‘hop’ trick explained above as follows.

```
\def\say#1#2\fi\fi
  {\fi\fi\csname if#1\endcsname}
```

Similar to the above example, let us write a macro that will test lexicographic (‘dictionary’) precedence of two strings:

```
\let\ex=\expandafter
\def\ifbefore
  #1#2{\ifallchars#1$\are#2$\before}
\def\ifallchars#1#2\are#3#4\before
  {\if#1$\say{true\ex}\else
   \if#3$\say{false\ex\ex\ex}\else
   \ifnum`#1>`#3 \say{false%
    \ex\ex\ex\ex\ex\ex\ex}\else
   \ifnum`#1<`#3 \say{true%
    \ex\ex\ex\ex\ex\ex\ex}
   \ex\ex\ex\ex\ex\ex\ex\ex}\else
   \ifrest#2\before#4\fi\fi\fi\fi}
\def\ifrest#1\before#2\fi\fi\fi\fi
  {\fi\fi\fi\fi
   \ifallchars#1\are#2\before}
\def\say#1{\csname if#1\endcsname}
```

In this macro a slightly different implementation of `\say` is used.

Simplified, a call to `\ifbefore` will eventually lead to a situation that looks (in the ‘true’ case) like

```
\ifbefore{...}{...}
  \if... %% some comparison that turns out true
    \csname iftrue\expandafter\endcsname
  \else .... \fi
... %% commands for the ‘before’ case
\else
... %% commands for the ‘not-before’ case
\fi
```

When the comparison has turned out true,  $\TeX$  will start processing the  $\langle \text{true text} \rangle$ , and make a mental note to remove any `\else ... \fi` part once an `\else` token is seen. Thus, the sequence

```
\csname iftrue\expandafter\endcsname \else ... \fi
```

is replaced by

```
\csname iftrue\endcsname
```

as the `\else` is seen while  $\TeX$  is still processing `\csname ... \endcsname`.

Calls to `\say` occur inside nested conditionals, so the number of `\expandafter` commands necessary may be larger than 1: for level two it is 3, for level three it is 7, and for level 4 it is 15. Slightly more compact implementations of this macro do exist.

## 第 14 章 Token Lists

$\text{\TeX}$  has only one type of data structure: the *token list*. There are token list registers that are available to the user, and  $\text{\TeX}$  has some special token lists: the `\every...` variables, `\errhelp`, and `\output`.

`\toks` Prefix for a token list register.

`\toksdef` Define a control sequence to be a synonym for a `\toks` register.

`\newtoks` Macro that allocates a token list register.

### 14.1 Token lists

Token lists are the only type of data structure that  $\text{\TeX}$  knows. They can contain character tokens and control sequence tokens. Spaces in a token list are significant. The only operations on token lists are assignment and unpacking.

$\text{\TeX}$  has 256 token list registers `\toks $nnn$`  that can be allocated using the macro `\newtoks`, or explicitly assigned by `\toksdef`; see below.

### 14.2 Use of token lists

Token lists are assigned by a  $\langle$ variable assignment $\rangle$ , which is in this case takes one of the forms

$\langle$ token variable $\rangle$ `\equals` $\langle$ general text $\rangle$

$\langle$ token variable $\rangle$ `\equals` $\langle$ filler $\rangle$  $\langle$ token variable $\rangle$

Here a  $\langle$ token variable $\rangle$  is an explicit `\toks $nnn$`  register, something that has been defined to such a register by `\toksdef` (probably hidden in `\newtoks`), or one of the special  $\langle$ token parameter $\rangle$  lists below. A  $\langle$ general text $\rangle$  has an explicit closing brace, but the open brace can be implicit.

Examples of token lists are (the first two lines are equivalent):

```
\toks0=\bgroup \a \b cd}
\toks0={\a \b cd}
\toks1=\toks2
```

Unpacking a token list is done by the command `\the`: the expansion of `\the⟨token variable⟩` is the sequence of tokens that was in the token list.

Token lists have a special behaviour in `\edef`: when prefixed by `\the` they are unpacked, but the resulting tokens are not evaluated further. Thus

```
\toks0={\a \b} \edef\SomeCs{\the\toks0}
```

gives

```
\SomeCs: macro:-> \a \b
```

This is in contrast to what happens ordinarily in an `\edef`; see page 132.

### 14.3 ⟨token parameter⟩

There are in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  a number of token lists that are automatically inserted at certain points. These ⟨token parameter⟩s are the following:

`\output` this token list is inserted whenever  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  decides it has sufficient material for a page, or when the user forces activation by a penalty  $\leq -10\,000$  in vertical mode (see Chapter 28);

`\everypar` is inserted when  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  switches from external or internal vertical mode to unrestricted horizontal mode (see Chapter 16);

`\everymath` is inserted after a single math-shift character that starts a formula;

`\everydisplay` is inserted after a double math-shift character that starts a display formula;

`\everyhbox` is inserted when an `\hbox` begins (see Chapter 5);

`\everyvbox` is inserted when a vertical box begins (see Chapter 5);

`\everyjob` is inserted when a job begins (see Chapter 32);

`\everycr` is inserted in alignments after `\cr` or a non-redundant `\crr` (see Chapter 25);

`\errhelp` contains tokens to supplement an `\errmessage` (see Chapter 35).

A ⟨token parameter⟩ behaves the same as an explicit `\toks $nnn$`  list, or a quantity defined by `\toksdef`.

### 14.4 Token list registers

Token lists can be stored in `\toks` registers:

```
\toks<8-bit number>
```

which is a  $\langle$ token variable $\rangle$ . Synonyms for token list registers can be made by the  $\langle$ registerdef $\rangle$  command `\toksdef` in a  $\langle$ shorthand definition $\rangle$ :

```
\toksdef<control sequence>\equals<8-bit number>
```

A control sequence defined this way is called a  $\langle$ toksdef token $\rangle$ , and this is also a token variable (the remaining third kind of token variable is the  $\langle$ token parameter $\rangle$ ).

The plain  $\text{\TeX}$  macro `\newtoks` uses `\toksdef` to allocate unused token list registers. This macro is `\outer`.

## 14.5 Examples

Token lists are probably among the least obvious components of  $\text{\TeX}$ : most  $\text{\TeX}$  users will never find occasion for their use, but format designers and other macro writers can find interesting applications. Following are some examples of the sorts of things that can be done with token lists.

### 14.5.1 Operations on token lists: stack macros

The number of primitive operations available for token lists is rather limited: assignment and unpacking. However, these are sufficient to implement other operations such as appending.

Let us say we have allocated a token register

```
\newtoks\list \list={\c}
```

and we want to add tokens to it, using the syntax

```
\Prepend \a \b (to:)\list
```

such that

```
\showthe\list
```

gives

```
> \a \b \c .
```

For this the original list has to be unpacked, and the new tokens followed by the old contents have to be assigned again to the register. Unpacking can be done with `\the` inside an `\edef`, so we arrive at the following macro:

```
\def\Prepend#1(to:)#2{\toks0={#1}%
  \edef\act{\noexpand#2={\the\toks0 \the#2}}%
  \act}
```

Note that the tokens that are to be added are first packed into a temporary token list, which is then again unpacked inside the `\edef`. Including them directly

would have led to their expansion.

Next we want to use token lists as a sort of stack: we want a ‘pop’ operation that removes the first element from the list. Specifically,

```
\Pop\list(into:)\first
\show\first \showthe\list
```

should give

```
> \first=macro:
->\a .
```

and for the remaining list

```
> \b \c .
```

Here we make creative use of delimited and undelimited parameters. With an `\edef` we unpack the list, and the auxiliary macro `\SplitOff` scoops up the elements as one undelimited argument, the first element, and one delimited argument, the rest of the elements.

```
\def\Pop#1(into:)#2{%
  \edef\act{\noexpand\SplitOff\the#1%
            (head:)\noexpand#2(tail:)\noexpand#1}%
  \act}
\def\SplitOff#1#2(head:)#3(tail:)#4{\def#3{#1}#4={#2}}
```

### 14.5.2 Executing token lists

The `\the` operation for unpacking token lists was used above only inside an `\edef`. Used on its own it has the effect of feeding the tokens of the list to  $\TeX$ ’s expansion mechanism. If the tokens have been added to the list in a uniform syntax, this gives rise to some interesting possibilities.

Imagine that we are implementing the bookkeeping of external files for a format. Such external files can be used for table of contents, list of figures, et cetera. If the presence of such objects is under the control of the user, we need some general routines for opening and closing files, and keeping track of what files we have opened at the user’s request.

Here only some routines for bookkeeping will be described. Let us say there is a list of auxiliary files, and an auxiliary counter:

```
\newtoks\auxlist \newcount\auxcount
```

First of all there must be an operation to add auxiliary files:

```
\def\NewAuxFile#1{\AddToAuxList{#1}%
  % plus other actions
}
\def\AddToAuxList#1{\let\=\relax
```

```
\edef\act{\noexpand\auxlist={\the\auxlist \\\{#1}}}%
\act}
```

This adds the name to the list in a uniform format:

```
\NewAuxFile{toc} \NewAuxFile{lof}
\showthe\auxlist
> \\\{toc}\\\{lof}.
```

using the control sequence `\\` which is left undefined.

Now this control sequence can be used for instance to count the number of elements in the list:

```
\def\ComputeLengthOfAuxList{\auxcount=0
\def\\#1{\advance\auxcount1\relax}%
\the\auxlist}
\ComputeLengthOfAuxList \showthe\auxcount
> 2.
```

Another use of this structure is the following: at the end of the job we can now close all auxiliary files at once, by

```
\def\CloseAuxFiles{\def\\#1{\CloseAuxFile{##1}}%
\the\auxlist}
\def\CloseAuxFile#1{\message{closing file: #1. }}%
% plus other actions
}
\CloseAuxFiles
```

which gives the output

```
closing file: toc. closing file: lof.
```

## 第 15 章 基线距离

文本行的高度和深度多半不会是相等的。因此  $\text{T}_{\text{E}}\text{X}$  添加行间粘连，以保证各行基线有一致的距离。这一章处理行间粘连的计算。

`\baselineskip` 竖直列中相邻盒子基线距离的‘理想值’。Plain  $\text{T}_{\text{E}}\text{X}$  默认为 12pt。

`\lineskiplimit` 竖直列中相邻盒子的底部和顶部需要保持的距离。Plain  $\text{T}_{\text{E}}\text{X}$  默认为 0pt。

`\lineskip` 在相邻盒子的底部和顶部的距离小于 `\lineskiplimit` 时添加的粘连。Plain  $\text{T}_{\text{E}}\text{X}$  默认为 1pt。

`\prevdepth` 上一个被  $\text{T}_{\text{E}}\text{X}$  添加到竖直列的盒子的深度。

`\nointerlineskip` 用于阻止一次行间粘连插入的宏。

`\offinterlineskip` 用于全局地阻止此后所有行间粘连的宏。

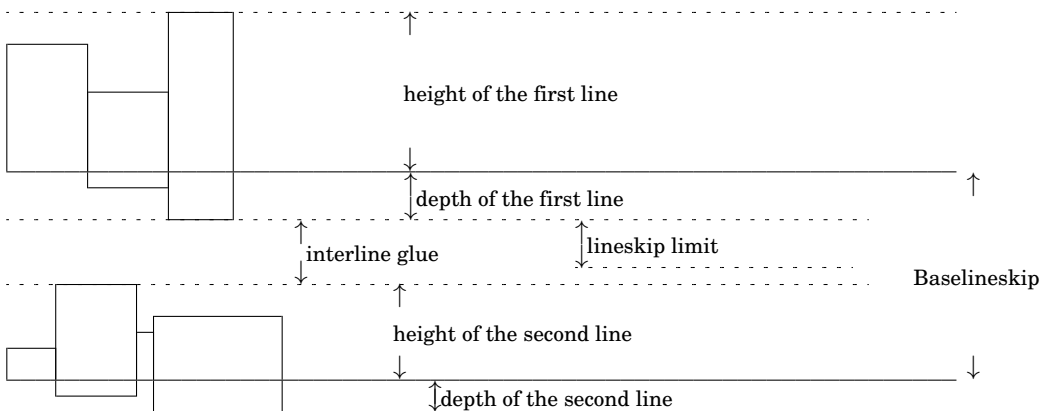
`\openup` 给 `\baselineskip`、`\lineskip` 和 `\lineskiplimit` 增加指定的量。

### 15.1 行间粘连

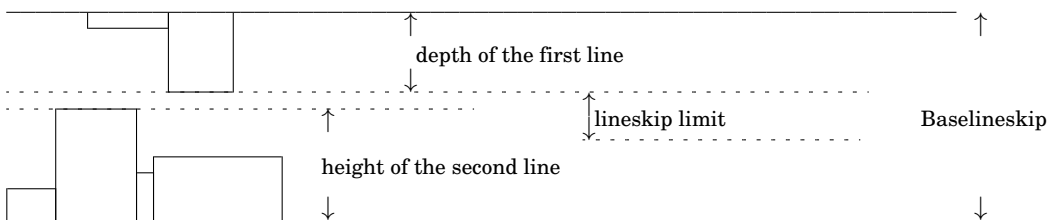
通过插入行间粘连， $\text{T}_{\text{E}}\text{X}$  试图让竖直列的盒子的基准点保持一定的距离。特别地，它试图让普通文本的各行保持恒定的基线距离。实际上，`\baselineskip` 是一个 (glue)，因此行距是可以伸缩的。然而，它的自然尺寸以及伸长量和收缩量，在各行之间是一致的。

当添加盒子——不管是段落文本行还是显式盒子——到竖直列时， $\text{T}_{\text{E}}\text{X}$  通常会添加粘连，使得它与前面盒子深度和当前盒子高度之和等于 `\baselineskip`。这样能让各行的基准点保持相同的距离。

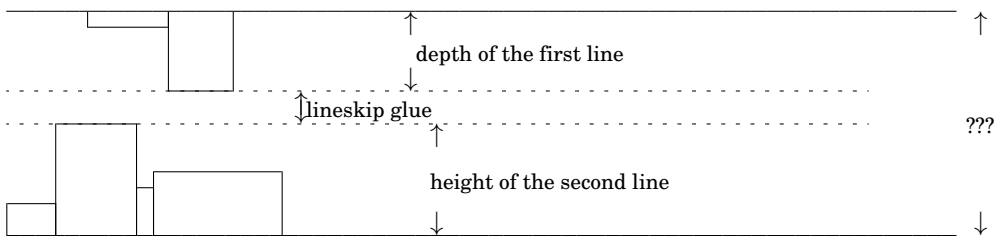




然而，这样做可能导致相邻盒子的底部和顶部的距离小于 `\lineskiplimit`：



碰到这种情形时，**T<sub>E</sub>X** 改为添加 `\lineskip` 粘连：



注意这通常会导致盒子的基线距离超过 `\baselineskip`。

确切的处理过程如下：

- 如果 `\prevdepth` 小于或等于 `-1000pt`，不会添加任何粘连，否则
- **T<sub>E</sub>X** 计算出上一个盒子底部和当前盒子顶部的距离：它用 `\baselineskip` 减去 `\prevdepth`（上一个盒子的深度）再减去当前盒子的高度；
- 如果这个距离小于 `\lineskiplimit`，**T<sub>E</sub>X** 添加一个粘连，这个粘连的自然尺寸等于该距离，伸长量和收缩量等于 `\baselineskip` 的伸长量和收缩量。
- 否则，**T<sub>E</sub>X** 改为添加 `\lineskip` 粘连。
- **T<sub>E</sub>X** 将 `\prevdepth` 设为当前项目的深度。

这里有两个例外情形：在标尺前后不添加行间粘连，而 `\unvbox` 或 `\unvcopy` 命令不改变 `\prevdepth` 值。在标线之后行间粘连是由于 `\prevdepth` 等于 `-1000pt` 而被阻止。

上面的处理过程不考虑盒子间另外插入的粘连。因此竖直模式中添加的盒子间距不会影响从基线距离算出的结果，从而不影响行间粘连的大小。这个规则也适用于在段落中用 `\vadjust` 添加的粘连。

例子：

```
\baselineskip=10pt \lineskiplimit=2pt \lineskip=2pt
\setbox0=\vbox{\hbox{\vrule depth4pt}
                \hbox{\vrule height 3pt}}
\showbox0
```

将给出

```
\box0=
\vbox(10.0+0.0)x0.4
.\hbox(0.0+4.0)x0.4
..\rule(*+4.0)x0.4
.\glue(\baselineskip) 3.0
.\hbox(3.0+0.0)x0.4
..\rule(3.0+*)x0
```

将盒子的距离减少到 `\lineskiplimit` 以内，即

```
\setbox0\vbox{\hbox{\vrule depth4pt}
                \hbox{\vrule height 5pt}}
\showbox0
```

将给出

```
\box0=
\vbox(11.0+0.0)x0.4
.\hbox(0.0+4.0)x0.4
..\rule(*+4.0)x0.4
.\glue(\lineskip) 2.0
.\hbox(5.0+0.0)x0.4
..\rule(5.0+*)x0.4
```

其中插入的是 `\lineskip` 粘连，而不是通常的 `\baselineskip` 粘连。

Plain TeX 中的默认值为

```
\lineskiplimit=0pt \lineskip=1pt
```

因此，当两个盒子开始碰到时，它们被拉开一个点的距离。

## 15.2 盒子深度

在确定行间粘连时，`\prevdepth` 代表竖列中前一个盒子的深度。`\prevdepth` 参数只能在竖直模式中使用。

`\prevdepth` 被设定为添加到竖列的盒子的深度，但它不受 `\unvbox` 或 `\unvcopy` 影响。在 `\hrule` 之后它被设为 `-1000pt` 以阻止在下个盒子前生成行间

粘连。

在竖直列的开头，`\prevdepth` 被设为 `-1000pt`，除非在 `\halign` 和 `\noalign` 中，此时使用的是外层列表的 `\prevdepth` 值。在阵列的结尾，阵列末尾行设定的 `\prevdepth` 的值被送到外层列表中。

为阻止一次行间粘连，只需要修改 `\prevdepth`。

```
\def\nointerlineskip{\prevdepth=-1000pt}
```

`\offinterlineskip` 宏要彻底得多：它使得所有行间粘连都为零，从调用它的时刻算起，或者，倘若在段落中间使用它，从段落开头处算起。它的定义为

```
\baselineskip=-1000pt \lineskip=0pt
\lineskiplimit\maxdimen
```

其中第二行是关键之处：它使得  $\text{\TeX}$  添加的 `\lineskip` 粘连始终等于零。这样 `\baselineskip` 的设置将不再重要。

`\offinterlineskip` 宏在阵列中能派上用场（见第 25 章）。

通过设置

```
\lineskiplimit=-\maxdimen
```

你可以强制  $\text{\TeX}$  始终使用 `\baselineskip`，而不管是否将导致盒子靠得太近，或者导致它们重叠。

## 15.3 术语

在铅字排版时代，一个字体的所有字母都印在相同大小的‘字身’上。因此排版的每行都有相同的高度和深度，从而基线的距离将有合适的大小。若由于某原因需要增加此距离（见 [52] 对此问题的讨论），则需要插入一个铅条（**strips of lead**）。这个额外的距离就称为‘**leading**’（发音同‘**led**ding’）。

在照相排版时代，基线距离有时候被称为‘**film transport**’，而‘**leading**’这个术语变得模糊，有时也被用于表示基线距离。这种混乱同样出现在  $\text{\TeX}$  中：参数 `\baselineskip` 指定基线距离，但是在追踪输出中（见上面的例子），为保证基线距离等于 `\baselineskip` 而插入的粘连也被称为 `\baselineskip`。

## 15.4 补充说明

一般地，对于超过一页的文档，自始至终保持相同的基线距离比较合适。然而，对于单页文档，你可以添加伸长度到 `\baselineskip`，这样可以让文本在底部对齐。

若仅仅想增加某两行的距离，你可以使用 `\adjust` 命令。这个命令的参量是竖直素材，它们将被插入到竖直列并放在这个命令所在的行之后。比如这个段落的第二行就包含命令 `\adjust{\kern2pt}`。

行间距的大小可以在段落中间修改，这是因为在分段为行并添加到主竖直列时才会用到 `\baselineskip` 的当前值。`\lineskip` 和 `\lineskiplimit` 类似。

**Plain TeX** 宏 `\openup` 给 `\baselineskip`、`\lineskip` 和 `\lineskiplimit` 增加宏参量所指定的大小。其效果是给行间距增加所指定的大小，无论它是由 `\baselineskip` 还是由 `\lineskip` 给出。

## 第 16 章 Paragraph Start

At the start of a paragraph  $\text{\TeX}$  inserts a vertical skip as a separation from the preceding paragraph, and a horizontal skip as an indentation for the current paragraph. This chapter explains the exact sequence of actions, and it discusses how  $\text{\TeX}$ 's decisions can be altered.

`\indent` Switch to horizontal mode and insert a box of width `\parindent`.

`\noindent` Switch to horizontal mode with an empty horizontal list.

`\parskip` Amount of glue added to the surrounding vertical list when a paragraph starts. Plain  $\text{\TeX}$  default: 0pt plus 1pt.

`\parindent` Size of the indentation box added in front of a paragraph. Plain  $\text{\TeX}$  default: 20pt.

`\everypar` Token list inserted in front of paragraph text;

`\leavevmode` Macro to switch to horizontal mode if necessary.

### 16.1 When does a paragraph start

$\text{\TeX}$  starts a paragraph whenever it switches from vertical mode to (unrestricted) horizontal mode. This switch can be effected by one of the commands `\indent` and `\noindent`, for example

```
{\bf And now~\dots}  
\vskip3pt  
\noindent It's~\dots
```

or by any  $\langle$ horizontal command $\rangle$ . Horizontal commands include characters, inline formulas, and horizontal skips, but not boxes. Consider the following examples. The character ‘T’ is a horizontal command:

```
\vskip3pt  
It's~\dots
```

A single \$ is a horizontal command:

`$x$ is supposed~\dots`

The control sequence `\hskip` is a horizontal command:

`\hskip .5\hsize Long indentation~\dots`

The full list of horizontal commands is given on page 75.

Upon recognizing a horizontal command in vertical mode,  $\TeX$  will perform an `\indent` command (and all the actions associated with it; see below), and after that it will reexamine the horizontal command, this time executing it.

## 16.2 What happens when a paragraph starts

The `\indent` and `\noindent` commands cause a paragraph to be started. An `\indent` command can either be placed explicitly by the user or a macro, or it can be inserted by  $\TeX$  when a  $\langle$ horizontal command $\rangle$  occurs in vertical mode; a `\noindent` command can only be placed explicitly.

After either command is encountered, `\parskip` glue is appended to the surrounding vertical list unless  $\TeX$  is in internal vertical mode and that list is empty (for example, at the start of a `\vbox` or `\vtop`).  $\TeX$  then switches to unrestricted horizontal mode with an empty horizontal list. In the case of `\indent` (which may be inserted implicitly) an empty `\hbox` of width `\parindent` is placed at the start of the horizontal list; after `\noindent` no indentation box is inserted.

The contents of the `\everypar`  $\langle$ token parameter $\rangle$  are then inserted into the input (see some applications below). After that, the page builder is exercised (see Chapter 27). Note that this happens in horizontal mode: this is to move the `\parskip` glue to the current page.

If an `\indent` command is given while  $\TeX$  is already in horizontal mode, the indentation box is inserted just the same. This is not very useful.

## 16.3 Assorted remarks

### 16.3.1 Starting a paragraph with a box

An `\hbox` does not imply horizontal mode, so an attempt to start a paragraph with a box, for instance

`\hbox to 0cm{\hss$\bullet$\hskip1em}Text ....`

will make the text following the box wind up one line below the box. It is necessary to switch to horizontal mode explicitly, using for instance `\noindent` or

`\leavevmode`. The latter is defined using `\unhbox`, which is a horizontal command.

### 16.3.2 Starting a paragraph with a group

If the first  $\langle$ horizontal command $\rangle$  of a paragraph is enclosed in braces, the `\everypar` is evaluated inside the group. This may give unexpected results. Consider this example:

```
\everypar={\setbox0=\vbox\bgroup\def\par{\egroup}}
{\bf Start} a paragraph ... \par
```

The  $\langle$ horizontal command $\rangle$  starting the paragraph is the character ‘S’, so when `\everypar` has been inserted the input is essentially

```
{\bf \indent\setbox0=\vbox\bgroup
\def\par{\egroup}Start} a paragraph ... \par
```

which is equivalent to

```
{\bf \setbox0=\vbox{Start} a paragraph ... \par
```

The effect of this is rather different from what was intended. Also,  $\TeX$  will probably end the job inside a group.

## 16.4 Examples

### 16.4.1 Stretchable indentation

Considering that `\parindent` is a  $\langle$ dimen $\rangle$ , not a  $\langle$ glue $\rangle$ , it is not possible to declare

```
\parindent=1cm plus 1fil
```

in order to get a variable indentation at the start of a paragraph. This problem may be solved by putting

```
\everypar={\nobreak\hskip 1cm plus 1fil\relax}
```

The `\nobreak` serves to prevent (in rare cases) a line break at the stretchable glue.

### 16.4.2 Suppressing indentation

Inserting `{\setbox0=\lastbox}` in the horizontal list at the beginning of the paragraph removes the indentation: indentation consists of a box, which is available through `\lastbox`. Assigning it effectively removes it from the list.

However, this command sequence has to be inserted at a moment when  $\TeX$  has already switched to horizontal mode, so explicit insertion of these com-

mands in front of the first `<horizontal command>` of the paragraph does not work. The moment of insertion of the `\everypar` tokens is a better candidate: specifying

```
\everypar={{\setbox0=\lastbox}}
```

leads to unindented paragraphs, even if `\parindent` is not zero.

### 16.4.3 An indentation scheme

The above idea of letting the indentation box be removed by `\everypar` can be put to use in a systematic approach to indentation, where two conditionals

```
\newif\ifNeedIndent %as a rule
\newif\ifneedindent %special cases
```

control whether paragraphs should indent as a rule, and whether in special cases indentation is needed. This section is taken from [8].

We take a fixed `\everypar`:

```
\everypar={\ControlledIndentation}
```

which executes in some cases the macro `\RemoveIndentation`

```
\def\RemoveIndentation{{\setbox0=\lastbox}}
```

The implementation of `\ControlledIndentation` is:

```
\def\ControlledIndentation
  {\ifNeedIndent \ifneedindent
    \else \RemoveIndentation\needindenttrue \fi
  \else \ifneedindent \needindentfalse
    \else \RemoveIndentation
  \fi \fi}
```

In order to regulate indentation for a whole document, the user now once specifies, for instance,

```
\NeedIndenttrue
```

to indicate that, in principle, all paragraphs should indent. Macros such as `\section` can then prevent indentation in individual cases:

```
\def\section#1{ ... \needindentfalse}
```

### 16.4.4 A paragraph skip scheme

The use of `\everypar` to control indentation, as was sketched above, can be extended to the paragraph skip.

A visible white space between paragraphs can be created by the `\parskip` parameter, but, once this parameter has been set to some value, it is difficult to prevent paragraph skip in certain places elegantly. Usually, white space



above and below environments and section headings should be specifiable independently of the paragraph skip. This section sketches an approach where `\parskip` is set to zero directly above and below certain constructs, while the `\everypar` is used to restore former values. This section is taken from [9].

First of all, here are two tools. The control sequence `\csarg` will be used only inside other macros; a typical call will look like

```
\csarg\vskip{#1\parskip}
```

Here is the definition:

```
\def\csarg#1#2{\expandafter#1\csname#2\endcsname}
```

Next follows a generalization of `\vskip`: the macro `\vspace` will not place its argument if the previous glue item is larger; otherwise it will eliminate the preceding glue, and place its argument.

```
\newskip\tempskipa
\def\vspace#1{\tempskipa=#1\relax
  \ifvmode \ifdim\tempskipa<\lastskip
    \else \vskip-\lastskip \vskip\tempskipa \fi
  \else \vskip\tempskipa \fi}
```

Now assume that any construct `foo` with surrounding white space starts and ends with macro calls `\StartEnvironment{foo}` and `\EndEnvironment{foo}` respectively. Furthermore, assume that to this environment there correspond three glue registers: the `\fooStartskip` (glue above the environment), `\fooParskip` (the paragraph skip inside the environment), and the `\fooEndskip` (glue below the environment).

For restoring the value of the paragraph skip a conditional and a glue register are needed:

```
\newskip\TempParskip \newif\ifParskipNeedsRestoring
```

The basic sequence for the starting and ending macros for the environments is then

```
\TempParskip=\parskip\parskip=0cm\relax
\ParskipNeedsRestoringtrue
```

The implementations can now be given as:

```
\def\StartEnvironment#1{\csarg\vspace{#1Startskip}

\begingroup % make changes local
  \csarg\TempParskip{#1Parskip} \parskip=0cm\relax
  \ParskipNeedsRestoringtrue}
\def\EndEnvironment#1{\csarg\vspace{#1Endskip}
  \endgroup % restore global values
  \ifParskipNeedsRestoring
  \else \TempParskip=\parskip \parskip=0cm\relax
  \ParskipNeedsRestoringtrue}
```

```
\fi}
```

The `\EndEnvironment` macro needs a little comment: if an environment is used inside another one, and it occurs before the first paragraph in that environment, the value of the paragraph skip for the outer environment has already been saved. Therefore no further actions are required in that case.

Note that both macros start with a vertical skip. This prevents the `\begingroup` and `\endgroup` statements from occurring in a paragraph.

We now come to the main point: if necessary, the `\everypar` will restore the value of the paragraph skip.

```
\everypar={\ControlledIndentation\ControlledParskip}  
\def\ControlledParskip  
  {\ifParskipNeedsRestoring  
    \parskip=\TempParskip \ParskipNeedsRestoringfalse  
  \fi}
```

## 第 17 章 Paragraph End

T<sub>E</sub>X's mechanism for ending a paragraph is ingenious and effective. This chapter explains the mechanism, the role of `\par` in it, and it gives a number of practical remarks.

`\par` Finish off a paragraph and go into vertical mode.

`\endgraf` Synonym for `\par`: `\let\endgraf=\par`

`\parfillskip` Glue that is placed between the last element of the paragraph and the line end. Plain T<sub>E</sub>X default: 0pt plus 1fil.

### 17.1 The way paragraphs end

A paragraph is terminated by the primitive `\par` command, which can be explicitly typed by the user (or inserted by a macro expansion):

```
... last words.\par
A new paragraph ...
```

It can be implicitly generated in the input processor of T<sub>E</sub>X by an empty line (see Chapter 2):

```
... last words.

A new paragraph ...
```

The `\par` can be inserted because a `<vertical command>` occurred in unrestricted horizontal mode:

```
... last words.\vskip6pt
A new paragraph ...
```

Also, a paragraph ends if a closing brace is found in horizontal mode inside `\vbox`, `\insert`, or `\output`.

After the `\par` command T<sub>E</sub>X goes into vertical mode and exercises the page builder (see page 253). If the `\par` was inserted because a vertical command occurred in horizontal mode, the vertical command is then examined anew. The

`\par` does not insert any vertical glue or penalties itself. A `\par` command also clears the paragraph shape parameters (see Chapter 18).

### 17.1.1 The `\par` command and the `\par` token

It is important to distinguish between the `\par` token and the primitive `\par` command that is the initial meaning of that token. The `\par` token is inserted when the input processor sees an empty line, or when the execution processor finds a `<vertical command>` in horizontal mode; the `\par` command is what actually closes off a paragraph. Decoupling the token and the command is an important tool for special effects in paragraphs (see some examples in Chapters 5 and 9).

### 17.1.2 Paragraph filling: `\parfillskip`

After the last element of the paragraph  $\mathrm{T\!E\!X}$  implicitly inserts the equivalent of

```
\unskip \penalty10000 \hskip\parfillskip
```

The `\unskip` serves to remove any spurious glue at the paragraph end, such as the space generated by the line end if the `\par` was inserted by the input processor. For example:

```
end.
```

```
\noindent Begin
```

results in the tokens

```
end. \par Begin
```

With the sequence inserted by the `\par` this becomes

```
end. \unskip\penalty10000\hskip ...
```

which in turn gives

```
end.\penalty ...
```

The `\parfillskip` is in plain  $\mathrm{T\!E\!X}$  first-order infinite (`0pt plus 1fil`), so ending a paragraph with `\hfil$\bullet$\par` will give a bullet halfway between the last word and the line end; with `\hfill$\bullet$\par` it will be flush right.

## 17.2 Assorted remarks

### 17.2.1 Ending a paragraph and a group at the same time

If a paragraph is set in a group, it may be necessary to ensure that the `\par` ending the paragraph occurs inside the group. The parameters influencing the typesetting of the paragraph, such as the `\leftskip` and the `\baselineskip`, are only looked at when the paragraph is finished. Thus finishing off a paragraph with

```
... last words.}\par
```

causes the values to be used that prevail outside the group, instead of those inside.

Better ways to end the paragraph are

```
... last words.\par}
```

or

```
... last words.\medskip}
```

In the second example the vertical command `\medskip` causes the `\par` token to be inserted.

### 17.2.2 Ending a paragraph with `\hfill\break`

The sequence `\hfill\break` is a way to force a ‘newline’ inside a paragraph. If you end a paragraph with this, however, you will probably get an `Underfull \hbox` error. Surprisingly, the underfull box is not the broken line – after all, that one was filled – but a completely empty box following it (actually, it does contain the `\leftskip` and `\rightskip`).

What happens? The paragraph ends with

```
\hfill\break\par
```

which turns into

```
\hfill\break\unskip\nobreak\hskip\parfillskip
```

The `\unskip` finds no preceding glue, so the `\break` is followed by a penalty item and a glue item, both of which disappear after the line break has been chosen at the `\break`. However,  $\TeX$  has already decided that there should be an extra line, that is, an `\hbox` to `\hsize`. And there is nothing to fill it with, so an underfull box results.

### 17.2.3 Ending a paragraph with a rule

See page 108 for paragraphs ending with rule leaders instead of the default `\parfillskip` white space.

### 17.2.4 No page breaks in between paragraphs

The `\par` command does not insert any glue in the vertical list, so in the sequence

```
... last words.\par \nobreak \medskip
\noindent First words ...
```

no page breaks will occur between the paragraphs. The vertical list generated is

```
\hbox(6.94444+0.0)x ...      % last line of paragraph
\penalty 10000                % \nobreak
\glue 6.0 plus 2.0 minus 2.0 % \medskip
\glue(\parskip) 0.0 plus 1.0 % \parskip
\glue(\baselineskip) 5.05556 % interline glue
\hbox(6.94444+0.0)x ...      % first line of paragraph
```

$\text{\TeX}$  will not break this vertical list above the `\medskip`, because the penalty value prohibits it; it will not break at any other place, because it can only break at glue if that glue is preceded by a non-discardable item.

### 17.2.5 Finite `\parfillskip`

In plain  $\text{\TeX}$ , `\parfillskip` has a (first-order) infinite stretch component. All other glue in the last line of a paragraph will then be set at natural width. If the `\parfillskip` has only finite (or possibly zero) stretch, other glue will be stretched or shrunk. A display formula in a paragraph with such a last line will be surrounded by `\abovedisplayskip` and `\belowdisplayskip`, even if `\abovedisplayshortskip` glue would be in order.

The reason for this is that glue setting is slightly machine-dependent, and any such processes should be kept out of  $\text{\TeX}$ 's global decisions.

### 17.2.6 A precaution for paragraphs that do not indent

If you are setting a text with both the paragraph indentation and the white space between paragraphs zero, you run the risk that the start of a new paragraph may be indiscernible when the last line of the previous paragraph ends almost or completely flush right. A sensible precaution for this is to set the `\parfillskip` to, for instance

```
\parfillskip=1cm plus 1fil
```

instead of the usual `0cm plus 1fil`.

On the other hand, you may let yourself be convinced by [46] that paragraphs should always indent.

## 第 18 章 段落形状

本章讨论影响段落形状的参数和命令。

`\parindent` 在段落开头添加的缩进盒子的宽度。**Plain T<sub>E</sub>X** 默认为 20pt。

`\hsize` 段落排版所用的行宽。**Plain T<sub>E</sub>X** 默认为 6.5in。

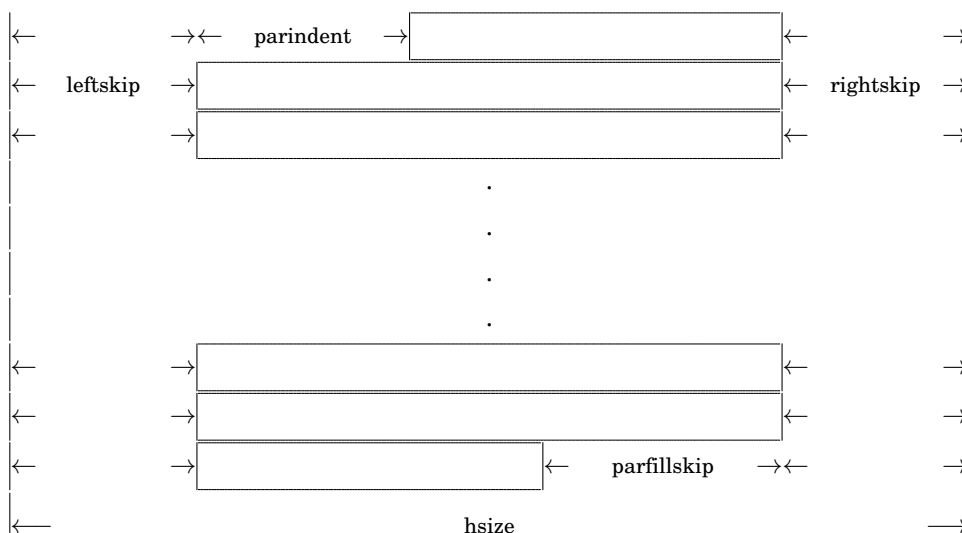
`\leftskip` 放在段落各行左侧的粘连。

`\rightskip` 放在段落各行右侧的粘连。

`\hangindent` 若大于零，表示左侧的缩进量；若小于零，表示右侧缩进量的负值。

`\hangafter` 若大于零，表示仅不缩进段落的前面多少行；若小于零，其绝对值表示仅缩进段落的前面多少行。默认值为 1。

`\parshape` 用于设定段落形状的一般命令。



## 18.1 文本行的宽度

在吸收完一个段落后，**T<sub>E</sub>X** 形成了一个以缩进盒子开头，以 `\parfillskip` 粘连结尾的水平列，然后将此列表分为长度相等的多行。段落各行的宽度等于 `\hsize`，其中包括两侧填充的 `\leftskip` 和 `\rightskip` 粘连。

**T<sub>E</sub>X** 的断行算法将 `\leftskip` 和 `\rightskip` 的值考虑在内。而 plain **T<sub>E</sub>X** 的 `\raggedright` 宏与 **L<sup>A</sup>T<sub>E</sub>X** 的 ‘flushleft’ 环境的要点就在于，将 `\rightskip` 设定为零加上某些伸长量。

命令 `\parshape` 和 `\hangindent` 也影响行宽。它们通过改变 `\hsize` 然后移动文本行所在盒子来实现。

## 18.2 段落形状参数

### 18.2.1 悬挂缩进

有种简单而常见的段落形状是开头或结尾若干行缩进。**T<sub>E</sub>X** 用 `\hangafter` 和 `\hangindent` 这两个参数实现这种形状。这两者都可以设定为正值或负值。

`\hangindent` 控制缩进量的大小：

- `\hangindent > 0`：段落根据它的大小在左侧缩进。
- `\hangindent < 0`：段落根据它的绝对值的大小在右侧缩进。

举个例子（假设 `\parindent=0pt`），

		a a a a a
		a a a a a
a a a a a a a a a a a a ...		a a ...
<code>\hangindent=10pt</code>		a a a a a
a a a a a a a a a a a a ...	给出	a a a a
		a a a ...
<code>\hangindent=-10pt</code>		a a a a a
a a a a a a a a a a a a ...		a a a a
		a a a ...

`\hangindent` 的默认值为 `0pt`。

`\hangafter` 参数确定需要缩进的行的数目：

- `\hangafter ≥ 0`：这么多行除外的其他行都被缩进；也就是说，段落开头的这么多行将不会被缩进。
- `\hangafter < 0`：它的绝对值就是段落开头将被缩进的行的数目。

举个例子，



		a a a a a
		a a a a a
a a a a a a a a a a a ...		a a ...
<code>\hangindent=10pt \hangafter=2</code>	给出	a a a a a
a a a a a a a a a a a ...		a a a a a
		a a ...
<code>\hangindent=10pt \hangafter=-2</code>		a a a a
a a a a a a a a a a a ...		a a a a
		a a a a ...

`\hangafter` 的默认值为 1。

这两个参数都可正可负，总共有四种悬挂缩进结果。下面还将介绍探出边界的悬挂缩进（‘outdent’）。

悬挂缩进是这样实现的：先从缩进行的 `\hsize` 中减去缩进量，接着将段落分段为行，然后往右移动那些左侧缩进行。

正常的 `\parindent` 缩进不受悬挂缩进影响。如果不要这个多余缩进，你应该在悬挂缩进段落开头使用 `\noindent`。

在每个 `\par` 命令之后 `\hangindent` 和 `\hangafter` 被还原为它们的默认值。

## 18.2.2 一般形状

用 `\parshape` 可以实现很一般的段落形状。这个命令可以指定段落前  $n$  行的缩进量和行宽，因此它有  $2n + 1$  个参数：先是行数  $n$ ，接着是  $n$  对缩进量和行宽。

`\parshape<equals> n i1 ℓ1 ... in ℓn`

如果段落超过  $n$  行，剩下的各行将重复第  $n$  行的指定。如果段落少于  $n$  行，多余的指定将被忽略。默认值（当然）是 `\parshape = 0`。

如果两者同时给出，那么 **TeX** 执行 `\parshape` 命令而忽略掉 `\hangindent`。在 `\parshape` 生效时，原本的 `\parindent`、`\leftskip` 和 `\rightskip` 仍然有效。

如同 `\hangindent`、`\hangafter` 和 `\looseness` 命令，`\parshape` 参数也会在 `\par` 命令后被重置（见第 19 章）。由于每个空行生成一个 `\par` 记号，在段落形状（或悬挂缩进）定义和段落内容之间你不应该留下空行。

控制序列 `\parshape` 是一个 `<internal integer>`：它的值等于所设定的行数  $n$ 。

## 18.3 杂项注记

### 18.3.1 末尾行居中

给 `\leftskip` 和 `\rightskip` 设定相等的伸长量和收缩量将给出居中文本，在各行都居中的意义上。要让段落首行和尾行真正地居中，必须将 `\parindent` 和 `\parfillskip` 都设定为零。然而，两侧边界是不对齐的。

出人意料地, `\leftskip` 和 `\rightskip` 可以让段落两侧对齐而末尾行居中:

```
\leftskip=0cm plus 0.5fil \rightskip=0cm plus -0.5fil
\parfillskip=0cm plus 1fil
```

在段落的非末尾行, 伸长量加起来等于零, 所以插入的 `\leftskip` 和 `\rightskip` 等于零。而在末尾行, `\parfillskip` 添加了 `plus 1fil` 的伸长量; 因此左右两边的总伸长量都是 `plus 0.5fil`。

如果这样写将会是错误的:

```
\leftskip=0cm plus 0.5fil \rightskip=0cm minus 0.5fil
```

对此  $\TeX$  将给出一个错误: 它抱怨 ‘infinite shrinkage’。

要让首行和尾行一样居中, 可以用下面的参数设定:

```
\parindent=0pt \everypar{\hskip 0pt plus -1fil}
\leftskip=0pt plus .5fil
\rightskip=0pt plus -.5fil
```

此时用 `\everypar` 插入的水平间距和 `\leftskip` 间距合起来将使得段落首行两侧的伸展能力相同。

### 18.3.2 凸出边界

假设你想要的悬挂缩进是让段落第二行以后各行凸出左边界 1cm。设定 `\hangindent=-1cm` 将得到在右边界缩进 1cm 的悬挂缩进, 所以需要另想办法。下面的方法可以完成此任务:

```
\leftskip=-1cm \hangindent=1cm \hangafter=-2
```

此方法的唯一问题在于段落结束后需要重设 `\leftskip`。我们可以通过重定义 `\par` 解决此问题:

```
\def\hangintomargin{\bgroup
  \leftskip=-1cm \hangindent=1cm \hangafter=-2
  \def\par{\endgraf\egroup}}
```

这里对 `\par` 的重定义是局部的, 只对需要凸出的段落生效。

还有一种更简洁的做法是利用 `\parshape`:

```
\dimen0=\hsize \advance\dimen0 by 1cm
\parshape=3          % three lines:
  0cm\hsize          % first line specification
  0cm\hsize          % second line specification
 -1cm\dimen0         % third line specification
```

### 18.3.3 悬挂在对象上

$\LaTeX$  格式有一个 `\@hangfrom` 宏, 用于将整个段落的文本悬挂在某个对象上, 此对象通常是盒子或短文本。

悬挂例子 这个段落是下面定义的 `\hangfrom` 宏的一个例子。在  $\text{\LaTeX}$  文档类中，有一个 `\@hangfrom` 宏（与这里定义的类似）用于排版多行节标题。

现在来看 `\hangfrom` 宏的定义：

```
\def\hangfrom#1{\def\hangobject{#1}\setbox0=\hbox{\hangobject}%
\hangindent \wd0 \noindent \hangobject \ignorespaces}
```

由于默认情形 `\hangafter=1`，这将使得段落首行的宽度为 `\hsize`，而后面各行按照 `\hangobject` 的宽度左缩进。

### 18.3.4 另一种悬挂缩进方法

将移动边界和凸出边界结合起来也可以得到悬挂缩进。比如逐项列表就可以用这种方式实现：

```
\newdimen\listindent
\def\itemize{\begingroup
\advance\leftskip by \listindent
\parindent=-\listindent}
\def\stopitemize{\par\endgroup}
\def\item#1{\par\leavevmode
\hbox to \listindent{#1\hfil}\ignorespaces
}
```

若想让列表项包含多个段落，可以这样实现：

```
\newdimen\listindent \newdimen\listparindent
\def\itemize{\begingroup
\advance\leftskip by \listindent
\parindent=\listparindent}
\def\stopitemize{\par\endgroup}
\def\item#1{\par\noindent
\hbox to 0cm{\kern-\listindent #1\hfil}\ignorespaces
}
```

例子：

```
\itemize\item{1.}First item\par
Is two paragraphs long.
\item{2.}Second item.\stopitemize
```

给出

#### 1. *First item*

*Is two paragraphs long.*

#### 2. *Second item.*

### 18.3.5 悬挂缩进对比边界移动

上面的例子似乎表明悬挂缩进与修改 `\leftskip` 和 `\rightskip` 是可互换的。它们确实可以，但只在一定程度上。

将段落的 `\leftskip` 设为某个正值意味着 `\hsize` 保持不变，但各行都以一个粘连项开始。另一方面，悬挂缩进是通过减少缩进行的 `\hsize` 值，然后在竖直列中移动水平盒子实现的。

这两种方法的区别可以从这个事实中看到：改变 `\leftskip` 并不会移动陈列公式。见第 9 章中展示的指引线如何受边界移动影响的例子。

### 18.3.6 更多例子

段落形状（用各种方式修改）的更多例子可以在 [10] 中找到。第 72 页的一个例子就取自该文章。

## 第 19 章 Line Breaking

This chapter treats line breaking and the concept of ‘badness’ that  $\text{\TeX}$  uses to decide how to break a paragraph into lines, or where to break a page. The various penalties contributing to the cost of line breaking are treated here, as is hyphenation. Page breaking is treated in Chapter [27](#).

`\penalty` Specify desirability of not breaking at this point.

`\linepenalty` Penalty value associated with each line break. Plain  $\text{\TeX}$  default: 10.

`\hyphenpenalty` Penalty associated with break at a discretionary item in the general case. Plain  $\text{\TeX}$  default: 50.

`\exhyphenpenalty` Penalty for breaking a horizontal line at a discretionary item in the special case where the prebreak text is empty. Plain  $\text{\TeX}$  default: 50.

`\adjdemerits` Penalty for adjacent visually incompatible lines. Plain  $\text{\TeX}$  default: 10 000.

`\doublehyphendemerits` Penalty for consecutive lines ending with a hyphen. Plain  $\text{\TeX}$  default: 10 000.

`\finalhyphendemerits` Penalty added when the penultimate line of a paragraph ends with a hyphen. Plain  $\text{\TeX}$  default: 5000.

`\allowbreak` Macro for creating a breakpoint by inserting a `\penalty0`.

`\pretolerance` Tolerance value for a paragraph without hyphenation. Plain  $\text{\TeX}$  default: 100.

`\tolerance` Tolerance value for lines in a paragraph with hyphenation. Plain  $\text{\TeX}$  default: 200.

`\emergencystretch` ( $\text{\TeX3}$  only) Assumed extra stretchability in lines of a paragraph.

`\looseness` Number of lines by which this paragraph has to be made longer

than it would be ideally.

`\prevgraf` The number of lines in the paragraph last added to the vertical list.

`\discretionary` Specify the way a character sequence is split up at a line break.

`\-` Discretionary hyphen; this is equivalent to `\discretionary{-}{-}{-}`.

`\hyphenchar` Number of the hyphen character of a font.

`\defaultshyphenchar` Value of `\hyphenchar` when a font is loaded. Plain  $\text{\TeX}$  default: ``\-`.

`\uchyph` Positive to allow hyphenation of words starting with a capital letter. Plain  $\text{\TeX}$  default: 1.

`\lefthyphenmin` ( $\text{\TeX}$ 3 only) Minimal number of characters before a hyphenation. Plain  $\text{\TeX}$  default: 2.

`\righthyphenmin` ( $\text{\TeX}$ 3 only) Minimum number of characters after a hyphenation. Plain  $\text{\TeX}$  default: 3.

`\patterns` Define a list of hyphenation patterns for the current value of `\language`; allowed only in  $\text{Init}\text{\TeX}$ .

`\hyphenation` Define hyphenation exceptions for the current value of `\language`.

`\language` Choose a set of hyphenation patterns and exceptions.

`\setlanguage` Reset the current language.

## 19.1 Paragraph break cost calculation

A paragraph is broken such that the amount  $d$  of *demerits* associated with breaking it is minimized. The total amount of demerits for a paragraph is the sum of those for the individual lines, plus possibly some extra penalties. Considering a paragraph as a whole instead of breaking it on a line-by-line basis can lead to better line breaking:  $\text{\TeX}$  can choose to take a slightly less beautiful line in the beginning of the paragraph in order to avoid bigger trouble later on.

For each line demerits are calculated from the *badness*  $b$  of stretching or shrinking the line to the break, and the *penalty*  $p$  associated with the break. The badness is not allowed to exceed a certain prescribed tolerance.

In addition to the demerits for breaking individual lines,  $\text{\TeX}$  assigns demerits for the way lines combine; see below.

The implementation of  $\text{\TeX}$ 's paragraphbreaking algorithm is explained in [27].

### 19.1.1 Badness

From the ratio between the stretch or shrink present in a line, and the actual stretch or shrink taken, the *badness* of breaking a line at a certain point is calculated. This badness is an important factor in the process of line breaking. See page 99 for the formula for badness.

In this chapter badness will only be discussed in the context of line breaking. Badness is also computed when a vertical list is stretched or shrunk (see Chapter 27).

The following terminology is used to describe badness:

**tight (3)** is any line that has shrunk with a badness  $b \geq 13$ , that is, by using at least one-half of its amount of shrink (see page 99 for the computation).

**decent (2)** is any line with a badness  $b \leq 12$ .

**loose (1)** is any line that has stretched with a badness  $b \geq 13$ , that is, by using at least one-half of its amount of stretch.

**very loose (0)** is any line that has stretched with a badness  $b \geq 100$ , that is, by using its full amount of stretch or more. Recall that glue can stretch, but not shrink more than its allowed amount.

The numbering is used in trace output (Chapter 34), and it is also used in the following definition: if the classifications of two adjacent lines differ by more than 1, the lines are said to be *visually incompatible*. See below for the `\adjdemerits` parameter associated with this.

Overfull horizontal and vertical boxes are passed unnoticed if their excess width or height is less than `\hfuzz` or `\vfuzz` respectively; they are not reported if the badness is less than `\hbadness` or `\vbadness` (see Chapter 5).

### 19.1.2 Penalties and other break locations

Line breaks can occur at the following *breakpoints* in horizontal lists:

1. At a penalty. The penalty value is the ‘aesthetic cost’ of breaking the line at that place. Negative penalties are considered as bonuses. A penalty of 10 000 or more inhibits, and a penalty of  $-10\,000$  or less forces, a break. Putting more than one penalty in a row is equivalent to putting just the one with the minimal value, because that one is the best candidate for line breaking. Penalties in horizontal mode are inserted by the user (or a user macro). The only exception is the `\nobreak` inserted before the `\parfillskip` glue.
2. At a glue, if it is not part of a math formula, and if it is preceded by a

non-discardable item (see Chapter 6). There is no penalty associated with breaking at glue.

The condition about the non-discardable precursor is necessary, because otherwise breaking in between two pieces of glue would be possible, which would cause ragged edges to the paragraph.

3. At a kern, if it is not part of a math formula and if it is followed by glue. There is no penalty associated with breaking at a kern.
4. At a math-off, if that is followed by glue. Since math-off (and math-on) act as kerns (see Chapter 23), this is very much like the previous case. There is no penalty associated with breaking at a math-off.
5. At a discretionary break. The penalty is the `\hyphenpenalty` or the `\exhyphenpenalty`. This is treated below.

Any discardable material following the break – glue, kerns, math-on/off and penalties – is discarded. If one considers a line break at glue (kern, math-on/off) to occur at the front end of the glue item, this implies that that piece of glue disappears in the break.

### 19.1.3 Demerits

From the badness of a line and the penalty, if any, the demerits of the line are calculated. Let  $l$  be the value of `\linepenalty`,  $b$  the badness of the line,  $p$  the penalty at the break; then the *demerits*  $d$  are given by

$$d = \begin{cases} (l + b)^2 + p^2 & \text{if } 0 \leq p < 10\,000 \\ (l + b)^2 - p^2 & \text{if } -10\,000 < p < 0 \\ (l + b)^2 & \text{if } p \leq -10\,000 \end{cases}$$

Both this formula and the one for the badness are described in [27] as ‘quite arbitrary’, but they have been shown to lead to good results in practice.

The demerits for a paragraph are the sum of the demerits for the lines, plus

- the `\adjdemerits` for any two adjacent lines that are not visually compatible (see above),
- `\doublehyphendemerits` for any two consecutive lines ending with a hyphen, and the
- `\finalhyphendemerits` if the penultimate line of a paragraph ends with a hyphen.

At the start of a paragraph  $\text{\TeX}$  acts as if there was a preceding line which was ‘decent’. Therefore `\adjdemerits` will be added if the first line is ‘very loose’. Also, the last line of a paragraph is ordinarily also ‘decent’ – all spaces



are set at natural width owing to the infinite stretch in the `\parfillskip` – so `\adjdemerits` are added if the preceding line is ‘very loose’.

Note that the penalties at which a line break is chosen weigh about as heavily as the badness of the line, so they can be relatively small. However, the three extra demerit parameters have to be of the order of the square of penalties and badnesses to weigh equally heavily.

#### 19.1.4 The number of lines of a paragraph

After a paragraph has been completed (or partially completed prior to a display), the variable `\prevgraf` records the number of lines in the paragraph. By assigning to this variable – and because this is a `<special integer>` such an assignment is automatically global –  $\mathrm{T\!E\!X}$ ’s decision processes can be influenced. This may be useful in combination with hanging indentation or `\parshape` specifications (see Chapter 18).

Some direct influence of the linebreaking process on the resulting number of lines exists. One factor is the `\linepenalty` which is included in the demerits of each line. By increasing the line penalty  $\mathrm{T\!E\!X}$  can be made to minimize the number of lines in a paragraph.

Deviations from the optimal number of lines, that is, the number of lines stemming from the optimal way of breaking a paragraph into lines, can be forced by the user by means of the `\looseness` parameter. This parameter, which is reset every time the shape parameters are cleared (see Chapter 18), indicates by how many lines the current paragraph should be made longer than is optimal. A negative value of `\looseness` will attempt to make the paragraph shorter by a number of lines that is the absolute value of the parameter.

$\mathrm{T\!E\!X}$  will still observe the values of `\pretolerance` and `\tolerance` (see below) when lengthening or shortening a paragraph under influence of `\looseness`. Therefore,  $\mathrm{T\!E\!X}$  will only lengthen or shorten a paragraph for as far as is possible without exceeding these parameters.

#### 19.1.5 Between the lines

$\mathrm{T\!E\!X}$ ’s paragraph mechanism packages lines into horizontal boxes that are appended to the surrounding vertical list. The resulting sequence of vertical items is then a repeating sequence of

- a box containing a line of text,
- possibly migrated vertical material (see page 78),

- a penalty item reflecting the cost of a page break at that point, which is normally the `\interlinepenalty` (see Chapter 27), and
- interline glue, which is calculated automatically on basis of the `\prevdepth` (see Chapter 15).

## 19.2 The process of breaking

$\text{\TeX}$  tries to break paragraphs in such a way that the badness of each line does not exceed a certain tolerance. If there exists more than one solution to this, the one with the fewest demerits is taken.

By setting `\tracingparagraphs` to a positive value,  $\text{\TeX}$  can be made to report the calculations of the paragraph mechanism in the log file. Some implementations of  $\text{\TeX}$  may have this option disabled to make  $\text{\TeX}$  run faster.

### 19.2.1 Three passes

First an attempt is made to split the paragraph into lines without hyphenating, that is, without inserting discretionary hyphens. This attempt succeeds if none of the lines has a badness exceeding `\pretolerance`.

Otherwise, a second pass is made, inserting discretionaries and using `\tolerance`. If `\pretolerance` is negative, the first pass is omitted.

$\text{\TeX}$  can be made to make a third pass if the first and second pass fail. If `\emergencystretch` is a positive dimension,  $\text{\TeX}$  will assume this much extra stretchability in each line when badness and demerits are calculated. Thus solutions that only slightly exceeded the given tolerances will now become feasible. However, no glue of size `\emergencystretch` is actually present, so underfull box messages may still occur.

### 19.2.2 Tolerance values

How much trouble  $\text{\TeX}$  will have typesetting a piece of text depends partly on the tolerance value. Therefore it is sensible to have some idea of what badness values mean in visual terms.

For lines that are stretched, the badness is 100 times the cube of the stretch ratio. A badness of 800 thus means that the stretch ratio is 2. If the space is, as in the ten-point Computer Modern Font,

`3.33pt plus 1.67pt minus 1.11pt`

a badness of 800 means that spaces have been stretched to

$$3.33\text{pt} + 2 \times 1.67\text{pt} = 6.66\text{pt}$$

that is, to exactly double their natural size. It is up to you to decide whether this is too large.

## 19.3 Discretionaries

A *discretionary item* `\discretionary{...}{...}{...}` marks a place where a word can be broken. Each of the three arguments is a `<general text>` (see Chapter 36): they are, in sequence,

- the *pre-break* text, which is appended to the part of the word before the break,
- the *post-break* text, which is prepended to the part of the word after the break, and
- the *no-break* text, which is used if the word is not broken at the discretionary item.

For example: `ab\discretionary{g}{h}{cd}ef` is the word `abcdef`, but it can be hyphenated with `abg` before the break and `hef` after. Note that there is no automatic hyphen character.

All three texts may contain any sorts of tokens, but any primitive commands and macros should expand to boxes, kerns, and characters.

### 19.3.1 Hyphens and discretionaries

Internally, `TEX` inserts the equivalent of

```
\discretionary{\char\hyphenchar\font}{-}{-}
```

at every place where a word can be broken. This causes a *hyphen character* to be placed before any break. No such discretionary is inserted if `\hyphenchar\font` is not in the range 0–255, or if its position in the font is not filled. When a font is loaded, its `\hyphenchar` value is set to `\defaultthyphenchar`. The `\hyphenchar` value can be changed after this.

In plain `TEX` the `\defaultthyphenchar` has the value ``-`, so for all fonts character 45 (the `ascii` hyphen character) is the hyphen sign, unless it is specified otherwise.

The primitive command `\-` (called a ‘discretionary hyphen’) `\-discretionary hyphen` is equivalent to the above

`\discretionary{\char\hyphenchar\font}{-}{-}`. Breaking at such a discretionary, whether inserted implicitly by `TEX` or explicitly by the user, has a cost of `\hyphenpenalty`.

In unrestricted horizontal mode an empty discretionary `\discretionary{}{}{}` is automatically inserted after characters whose character code is the `\hyphenchar` value of the font, thus enabling hyphenation at that point. The penalty for breaking a line at such a discretionary with an empty pre-break text is `\exhyphenpenalty`, that is, the ‘explicit hyphen’ penalty.

If a word contains discretionary breaks, for instance because of explicit hyphen characters,  $\text{\TeX}$  will not consider it for further hyphenation. People have solved the ensuing problems by tricks such as

```
\def\={\penalty10000 \hskip0pt -\penalty0 \hskip0pt\relax}
... integro\=differential equations...
```

The skips before and after the hyphen lead  $\text{\TeX}$  into treating the first and second half of the compound expression as separate words; the penalty before the first skip inhibits breaking before the hyphen.

### 19.3.2 Examples of discretionaries

*Languages* such as German or Dutch have words that change spelling when hyphenated (German: ‘backen’ becomes ‘bak-ken’; Dutch: ‘autootje’ becomes ‘auto-tje’). This problem can be solved with  $\text{\TeX}$ ’s discretionaries.

For instance, for German (this is inspired by [36]):

```
\catcode`\="=\active
\def"#1{\ifx#1k\discretionary{k-}{k}{ck}\fi}
```

which enables the user to write `ba"ken`.

In Dutch there is a further problem which allows a nice systematic solution. Umlaut characters (‘trema’ is the Dutch term) should often disappear in a break, for instance ‘na"apen’ hyphenates as ‘na-apen’, and ‘onbe"invloedbaar’ hyphenates as ‘onbe-invloedbaar’. A solution (inspired by [5]) is

```
\catcode`\="=\active
\def"#1{\ifx#1i\discretionary{-}{i}{\\"i}%
\else \discretionary{-}{#1}{\\"#1}\fi}
```

which enables the user to type `na"apen` and `onbe"invloedbaar`.

## 19.4 Hyphenation

$\text{\TeX}$ ’s *hyphenation* algorithm uses a list of patterns to determine at what places a word that is a candidate for hyphenation can be broken. Those aspects of hyphenation connected with these patterns are treated in appendix H of the  $\text{\TeX}$  book; the method of generating hyphenation patterns automatically is described in [30]. People have been known to generate lists of patterns by hand;

see for instance [28]. Such hand-generated lists may be superior to automatically generated lists.

Here it will mainly be described how  $\TeX$  declares a word to be a candidate for hyphenation. The problem here is how to cope with punctuation and things such as quotation marks that can be attached to a word. Also, *implicit kerns*, that is, kerns inserted because of font information, must be handled properly.

### 19.4.1 Start of a word

$\TeX$  starts at glue items (if they are not in math mode) looking for a *starting letter* of a word: a character with non-zero `\lccode`, or a ligature starting with such a character (upper/lowercase codes are explained on page 46). Looking for this starting letter,  $\TeX$  bypasses any implicit kerns, and characters with zero `\lccode` (this includes, for instance, punctuation and quotation marks), or ligatures starting with such a character.

If no suitable starting letter turns up, that is, if something is found that is not a character or ligature,  $\TeX$  skips to the next glue, and starts this algorithm anew. Otherwise a trial word is collected consisting of all following characters with non-zero `\lccode` from the same font as the starting letter, or ligatures consisting completely of such characters. Implicit kerns are allowed between the characters and ligatures.

If the starting letter is from a font for which the value of `\hyphenchar` is invalid, or for which this character does not exist, hyphenation is abandoned for this word. If the starting letter is an uppercase letter (that is, it is not equal to its own `\lccode`),  $\TeX$  will abandon hyphenation unless `\uchyph` is positive. The default value for this parameter is 1 in plain  $\TeX$ , implying that capitalized words are subject to hyphenation.

### 19.4.2 End of a word

Following the trial word can be characters (from another font, or with zero `\lccode`), ligatures or implicit kerns. After these items, if any, must follow

- glue or an explicit kern,
- a penalty,
- a *whatsit*, or
- a `\mark`, `\insert`, or `\vadjust` item.

In particular, the word will not be hyphenated if it is followed by a

- `box`,

- rule,
- math formula, or
- discretionary item.

Since discretionaries are inserted after the `\hyphenchar` of the font, occurrence of this character inhibits further hyphenation. Also, placement of accents is implemented using explicit kerns (see Chapter 3), so any `\accent` command is considered to be the end of a word, and inhibits hyphenation of the word.

### 19.4.3 $\text{\TeX}$ 2 versus $\text{\TeX}$ 3

There is a noticeable difference in the treatment of hyphenated fragments between  $\text{\TeX}$ 2 and  $\text{\TeX}$ 3.  $\text{\TeX}$ 2 insists that the part before the break should be at least two characters, and the part after the break three characters, long. Typographically this is a sound decision: this way there are no two-character pieces of a word stranded at the end or beginning of the line. Both before and after the break there are at least three characters.

In  $\text{\TeX}$ 3 two integer parameters have been introduced to control the length of these fragments: `\lefthyphenmin` and `\righthyphenmin`. These are set to 2 and 3 respectively in the plain format for  $\text{\TeX}$ 3. If the sum of these two is 63 or more, all hyphenation is suppressed.

Another addition in  $\text{\TeX}$ 3, the possibility to have several sets of hyphenation patterns, is treated below.

### 19.4.4 Patterns and exceptions

The statements

```
\patterns⟨general text⟩
\hyphenation⟨general text⟩
```

are `⟨hyphenation assignment⟩`s, which are `⟨global assignment⟩`s. The `\patterns` command, which specifies a list of hyphenation patterns, is allowed only in  $\text{Init}\text{\TeX}$  (see Chapter 33), and all patterns must be specified before the first paragraph is typeset.

Hyphenation exceptions can be specified at any time with statements such as

```
\hyphenation{oxy-mo-ron gar-goyle}
```

which specify locations where a word may be hyphenated. Subsequent `\hyphenation` statements are cumulative.

In  $\text{\TeX}$ 3 these statements are taken to hold for the language that is the current value of the `\language` parameter.

## 19.5 Switching hyphenation patterns

When typesetting paragraphs,  $\text{\TeX}$  (version 3) can use several sets of patterns and hyphenation exceptions, for at most 256 *languages*.

If a `\patterns` or `\hyphenation` command is given (see above),  $\text{\TeX}$  stores the patterns or exceptions under the current value of the `\language` parameter. The `\patterns` command is only allowed in  $\text{Init}\text{\TeX}$ , and patterns must be specified before any typesetting is done. Hyphenation exceptions, however, can be specified cumulatively, and not only in  $\text{Init}\text{\TeX}$ .

In addition to the `\language` parameter, which can be set by the user,  $\text{\TeX}$  has internally a *current language*. This is set to zero at the start of every paragraph. For every character that is added to a paragraph the current language is compared with the value of `\language`, and if they differ a whatsit element is added to the horizontal list, resetting the current language to the value of `\language`.

At the start of a paragraph, this whatsit is inserted after the `\everypar` tokens, but `\lastbox` can still access the indentation box.

As an example, suppose that a format has been created such that language 0 is English, and language 1 is Dutch. English hyphenations will then be used if the user does not specify otherwise; if a job starts with

```
\language=1
```

the whole document will be set using Dutch hyphenations, because  $\text{\TeX}$  will insert a command changing the current language at the start of every paragraph. For example:

```
\language=1
T...
```

gives

```
.\hbox(0.0+0.0)x20.0      % indentation
.\setlanguage1 (hyphenmin 2,3) % language whatsit
.\tenrm T                 % start of text
```

The whatsit can be inserted explicitly, without changing the value of `\language`, by specifying

```
\setlanguage(number)
```

However, this will hardly ever be needed. One case where it may be necessary is when the contents of a horizontal box are unboxed to a paragraph: inside the

box no whatsits are added automatically, since inside such a box no hyphenation can take place. See page [69](#) for another problem with text in horizontal boxes.



## 第 20 章 Spacing

The usual interword space in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is specified in the font information, but the user can override this. This chapter explains the rules by which  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  calculates interword space.

- `\` Control space. Insert the same amount of space as a space token would if `\spacefactor = 1000`.
- `\spaceskip` Interword glue if non-zero.
- `\xspaceskip` Interword glue if non-zero and `\spacefactor ≥ 2000`.
- `\spacefactor` 1000 times the ratio by which the stretch (shrink) component of the interword glue should be multiplied (divided).
- `\sfcode` Value for `\spacefactor` associated with a character.
- `\frenchspacing` Macro to switch off extra space after punctuation.
- `\nonfrenchspacing` Macro to switch on extra space after punctuation.

### 20.1 Introduction

In between words in a text,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  inserts space. This space has a natural component, plus stretch and shrink to make justified (right-aligned) text possible. Now, in certain styles of typesetting, there is more space after punctuation. This chapter discusses the mechanism that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  uses to realize such effect.

Here is the general idea:

- After every character token, the `\spacefactor` quantity is updated with the space factor code of that character.
- When space is inserted, its natural size can be augmented (if `\spacefactor ≥ 2000`), and in general its stretch is multiplied, and its shrink divided, by `\spacefactor/1000`.
- There are further rules, for instance so that in `...word.)` And... the space

is modified according to the period, not the closing parenthesis.

## 20.2 Automatic interword space

For every space token in horizontal mode the interword glue of the current font is inserted, with stretch and shrink components, all determined by `\fontdimen` parameters. To be specific, font dimension 2 is the normal interword space, dimension 3 is the amount of stretch of the interword space, and 4 is the amount of shrink. Font dimension 7 is called the ‘extra space’; see below (the list of all the font dimensions appears on page 52).

Ordinarily all spaces between words (in one font) would be treated the same. To allow for differently sized spaces – for instance a typeset equivalent of the double spacing after punctuation in typewritten documents –  $\text{\TeX}$  associates with each character a so-called *space factor*.

When a character is added to the current horizontal list, the space factor code (`\sfcode`) of that character is assigned to the space factor `\spacefactor`. There are two exceptions to this rule:

- When the space factor code is zero, the `\spacefactor` does not change. This mechanism allows space factors to persist through parentheses and such; see section 20.5.3.
- When the space factor code of the last character is  $>1000$  and the current space factor is  $<1000$ , the space factor becomes 1000. This mechanism prevents elongated spaces after initials; see section 20.5.2.

The maximum space factor is 32 767.

The stretch component of the interword space is multiplied by the space factor divided by 1000; the shrink component is divided by this factor. The extra space (font dimension 7) is added to the natural component of the interword space when the space factor is  $\geq 2000$ .

## 20.3 User interword space

The user can override the interword space contained in the `\fontdimen` parameters by setting the `\spaceskip` and the `\xspaceskip` to non-zero values. If `\spaceskip` is non-zero, it is taken instead of the normal interword space (`\fontdimen2` plus `\fontdimen3` minus `\fontdimen4`), but a non-zero `\xspaceskip` is used as interword space if the space factor is  $\geq 2000$ .

If the `\spaceskip` is used, its stretch and shrink components are multiplied and divided respectively by `\spacefactor/1000`.

Note that, if `\spaceskip` and `\xspaceskip` are defined in terms of em, they change with the font.

例子: *Let the following macros be given:*

```
\def\ a.{\vrule height10pt width4pt\spacefactor=1000\relax}
\def\ b.{\vrule height10pt width4pt\spacefactor=3000\relax}
\def\ c.{\vrule height10pt width4pt\relax}
```

*then*

```
\vbox{
\fontdimen2\font=4pt % normal space
\fontdimen7\font=3pt % extra space           |||
\ a. \ b. \ c\par
% zero extra space
\fontdimen7\font=0pt
\ a. \ b. \ c\par                             |||
% set \spaceskip for normal space             gives
\spaceskip=2\fontdimen2\font
\ a. \ b. \ c\par                             |||
% set \xspaceskip
\xspaceskip=2pt
\ a. \ b. \ c\par
}
```

*In all of these lines the glue is set at natural width. In the first line the high space factor value after \ b causes the extra space \fontdimen7 to be added. If this is zero (second line), the only difference between space factor values is the stretch/shrink ratio. In the third line the \spaceskip is taken for all space factor values. If the \xspaceskip is nonzero, it is taken (fourth line) instead of the \spaceskip for the high value of the space factor.*

## 20.4 Control space and tie

The control character `\`, *control space* is a horizontal command which inserts a space, `\` acting as if the current space factor is 1000. However, it does not affect the value of `\spacefactor`.

Control space has two main uses. First, it is convenient to use after a control sequence: `\TeX\ is fun!` Secondly, it can be used after abbreviations when

`\nonfrenchspacing` (see below) is in effect. For example:

```
\hbox spread 9pt{\nonfrenchspacing
  The Reverend Dr. Drofnats}
```

gives

The Reverend Dr. Drofnats

while

```
\hbox spread 9pt{\nonfrenchspacing
  The Reverend Dr.\ Drofnats}
```

gives

The Reverend Dr. Drofnats

(The `spread 9pt` is used to make the effect more visible.)

The active character (in the plain format) tilde or *tie*, `~`, uses control space: it is defined as

```
\catcode`\~=\active
\def~{\penalty10000\ }
```

Such an active tilde is called a ‘tie’; it inserts an ordinary amount of space, and prohibits breaking at this space.

## 20.5 More on the space factor

### 20.5.1 Space factor assignments

The space factor of a particular character is contained in its *spacefactor code* and can be assigned as

```
\sfcode⟨8-bit number⟩⟨equals⟩⟨number⟩
```

Ini $\TeX$  assigns a space factor code of 1000 to all characters except uppercase characters; they get a space factor code of 999. The plain format then assigns space factor codes greater than 1000 to various punctuation symbols, for instance `\sfcode`\.=3000`, which triples the stretch and shrink after a full stop. Also, for all space factor values  $\geq 2000$  the extra space is added; see above.

### 20.5.2 Punctuation

Because the space factor cannot jump from a value below 1000 to one above, a punctuation symbol after an uppercase character will not have the effect on the interword space that punctuation after a lowercase character has.

例子:

```
a% \sfcode`a=1000, space factor becomes 1000
.% \sfcode`=3000, spacefactor becomes 3000
% subsequent spaces will be increased.

A% \sfcode`A=999, space factor becomes 999
.% \sfcode`=3000, space factor becomes 1000
% subsequent spaces will not be increased.
```

Thus, initials are not mistaken for sentence ends. If an uppercase character does end a sentence, for instance

```
... and NASA.
```

there are several solutions:

```
... NASA\spacefactor=1000.
```

or

```
... NASA\hbox{ }.
```

which abuses the fact that after a box the space factor is set to 1000. The  $\LaTeX$  macro `\@` is equivalent to the first possibility.

In the plain format two macros are defined that switch between uniform interword spacing, *frenchspacing*, and extra space after punctuation, which is more an American custom. The macro `\frenchspacing` sets the space factor code of all punctuation to 1000; the macro `\nonfrenchspacing` sets it to values greater than 1000.

Here are the actual definitions from `plain.tex`:

```
\def\frenchspacing{\sfcode`.\@m \sfcode`?\@m
 \sfcode`!\@m \sfcode`:\@m
 \sfcode`; \@m \sfcode`, \@m}
\def\nonfrenchspacing{\sfcode`\.3000 \sfcode`?3000
 \sfcode`!3000 \sfcode`:2000
 \sfcode`;1500 \sfcode`,1250 }
```

where

```
\mathchardef\@m=1000
```

is given in the plain format.

French spacing is a somewhat controversial issue: the  $\TeX$  book acts as if non-French spacing is standard practice in printing, but for instance in [14] one finds ‘The space of the line should be used after all points in normal text’. Extra space after punctuation may be considered a ‘typewriter habit’, but this is not entirely true. It used to be a lot more common than it is nowadays, and there are rational arguments against it: the full stop (point, period) at the end of a sentence, where extra punctuation is most visible, is rather small, so it carries some extra visual space of its own above it. This book does not use extra space after punctuation.

### 20.5.3 Other non-letters

The zero value of the space factor code makes characters that are not a letter and not punctuation ‘transparent’ for the space factor.

例子:

```
a% \sfcode`a=1000, space factor becomes 1000
.% \sfcode`.=3000, spacefactor becomes 3000
% subsequent spaces will be increased.
```

```
a% \sfcode`a=1000, space factor becomes 1000
.% \sfcode`.=3000, space factor becomes 3000
)% \sfcode`=0,    space factor stays 3000
% subsequent spaces will be increased.
```

### 20.5.4 Other influences on the space factor

The space factor is 1000 when  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  starts forming a horizontal list, in particular after `\indent`, `\noindent`, and directly after a display. It is also 1000 after a `\vrule`, an accent, or a  $\langle\mathrm{box}\rangle$  (in horizontal mode), but it is not influenced by `\unhbox` or `\unhcopy` commands.

In the first column of a `\valign` the space factor of the surrounding horizontal list is carried over; similarly, after a vertical alignment the space factor is set to the value reached in the last column.

## 第 21 章 Characters in Math Mode

In math mode every character specifies by its `\mathcode` what position of a font to access, among other things. For delimiters this story is a bit more complicated. This chapter explains the concept of math codes, and shows how  $\TeX$  implements variable size delimiters.

`\mathcode` Code of a character determining its treatment in math mode.

`\mathchar` Explicit denotation of a mathematical character.

`\mathchardef` Define a control sequence to be a synonym for a math character code.

`\delcode` Code specifying how a character should be used as delimiter.

`\delimiter` Explicit denotation of a delimiter.

`\delimiterfactor` 1000 times the fraction of a delimited formula that should be covered by a delimiter. Plain  $\TeX$  default: 901

`\delimitershortfall` Size of the part of a delimited formula that is allowed to go uncovered by a delimiter. Plain  $\TeX$  default: 5pt

`\nulldelimiterspace` Width taken for empty delimiters. Plain  $\TeX$  default: 1.2pt

`\left` Use the following character as an open delimiter.

`\right` Use the following character as a closing delimiter.

`\big` One line high delimiter.

`\Big` One and a half line high delimiter.

`\bigg` Two lines high delimiter.

`\Bigg` Two and a half lines high delimiter.

`\bigl` etc. Left delimiters.

`\bigm` etc. Delimiters used as binary relations.

`\bigr` etc. Right delimiters.

`\radical` Command for setting things such as root signs.

`\mathaccent` Place an accent in math mode.

`\skewchar` Font position of an after-placed accent.

`\defaultskewchar` Value of `\skewchar` when a font is loaded.

`\skew` Macro to shift accents on top of characters explicitly.

`\widehat` Hat accent that can accommodate wide expressions.

`\widetilde` Tilde accent that can accommodate wide expressions.

## 21.1 Mathematical characters

Each of the 256 permissible character codes has an associated `\mathcode`, which can be assigned by

```
\mathcode⟨8-bit number⟩⟨equals⟩⟨15-bit number⟩
```

When processing in math mode,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  replaces all characters of categories 11 and 12, and `\char` and `\chardef` characters, by their associated `mathcode`.

The 15-bit math code is most conveniently denoted hexadecimally as "xyzz, where

$x \leq 7$  is the class (see page 219),  
 $y$  is the font family number (see Chapter 22), and  
 $zz$  is the position of the character in the font.

Math codes can also be specified directly by a `⟨math character⟩`, which can be

- `\mathchar⟨15-bit number⟩;`
  - `⟨mathchardef token⟩`, a control sequence that was defined by  
`\mathchardef⟨control sequence⟩⟨equals⟩⟨15-bit number⟩`
- or
- a delimiter command  
`\delimiter⟨27-bit number⟩`

where the last 12 bits are discarded.

The commands `\mathchar` and `\mathchardef` are analogous to `\char` and `\chardef` in text mode. Delimiters are treated below. A `⟨mathchardef token⟩` can be used as a `⟨number⟩`, even outside math mode.

In  $\mathrm{I}_{\mathrm{n}}\mathrm{T}_{\mathrm{E}}\mathrm{X}$  all letters receive `\mathcode "71zz` and all digits receive `"70zz`, where "zz is the hexadecimal position of the character in the font. Thus, letters



are initially from family 1 (math italic in plain  $\text{T}_{\text{E}}\text{X}$ ), and digits are from family 0 (roman). For all other characters,  $\text{Init}_{\text{E}}\text{X}$  assigns

```
\mathcode x = x,
```

thereby placing them also in family 0.

If the mathcode is "8000, the smallest integer that is not a ⟨15-bit number⟩, the character is treated as an active character with the original character code. Plain  $\text{T}_{\text{E}}\text{X}$  assigns a \mathcode of "8000 to the space, underscore and prime.

## 21.2 Delimiters

After `\left` and `\right` commands  $\text{T}_{\text{E}}\text{X}$  looks for a delimiter. A delimiter is either an explicit `\delimiter` command (or a macro abbreviation for it), or a character with a non-zero delimiter code.

The `\left` and `\right` commands implicitly delimit a group, which is considered as a subformula. Since the enclosed formula can be arbitrarily large, the quest for the proper delimiter is a complicated story of looking at variants in two different fonts, linked chains of variants in a font, and building extendable delimiters from repeatable pieces.

The fact that a group enclosed in `\left... \right` is treated as an independent subformula implies that a sub- or superscript at the start of this formula is not considered to belong to the delimiter. For example,  $\text{T}_{\text{E}}\text{X}$  acts as if `\left(_2` is equivalent to `\left(\{ \}_2`. (A subscript after a `\right` delimiter is positioned with respect to that delimiter.)

### 21.2.1 Delimiter codes

To each character code there corresponds a *delimiter code*, assigned by

```
\delcode⟨8-bit number⟩⟨equals⟩⟨24-bit number⟩
```

A delimiter code thus consists of six hexadecimal digits "uvvxyy, where

uvv is the small variant of the delimiter, and

xyy is the large variant;

u, x are the font families of the variants, and

vv, yy are the locations in those fonts.

Delimiter codes are used after `\left` and `\right` commands.  $\text{Init}_{\text{E}}\text{X}$  sets all delimiter codes to  $-1$ , except `\delcode`.=0`, which makes the period an empty delimiter. In plain  $\text{T}_{\text{E}}\text{X}$  delimiters have typically  $u = 2$  and  $x = 3$ , that is, first family 2 is tried, and if no big enough delimiter turns up family 3 is tried.

### 21.2.2 Explicit `\delimiter` commands

Delimiters can also be denoted explicitly by a  $\langle 27\text{-bit number} \rangle$ ,

```
\delimiter"tuvvxy
```

where `uvvxy` are the small and large variant of the delimiter as above; the extra digit `t` (which is  $< 8$ ) denotes the class (see page 219). For instance, the `\langle` macro is defined as

```
\def\langle{\delimiter "426830A }
```

which means it belongs to class 4, opening. Similarly, `\rangle` is of class 5, closing; and `\uparrow` is of class 3, relation.

After `\left` and `\right` – that is, when  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is looking for a delimiter – the class digit is ignored; otherwise – when  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is not looking for a delimiter – the rightmost three digits are ignored, and the four remaining digits are treated as a `\mathchar`; see above.

### 21.2.3 Finding a delimiter; successors

Typesetting a delimiter is a somewhat involved affair. First  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  determines the size  $y$  of the formula to be covered, which is twice the maximum of the height and depth of the formula. Thus the formula may not look optimal if it is not centred itself.

The size of the delimiter should be at least `\delimiterfactor`  $\times y/1000$  and at least  $y - \text{\code{\delimiterfactor}}$ .  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  then tries first the small variant, and if that one is not satisfactory (or if the `uvv` part of the delimiter is 000) it tries the large variant. If trying the large variant does not meet with success,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  takes the largest delimiter encountered in this search; if no delimiter at all was found (which can happen if the `xyy` part is also 000), an empty box of width `\nulldelimiterspace` is taken.

Investigating a variant means, in sequence,

- if the current style (see page 217) is `scriptscriptstyle`, the `\scriptscriptfont` of the family is tried;
- if the current style is `scriptstyle` or smaller, the `\scriptfont` of the family is tried;
- otherwise the `\textfont` of the family is tried.

The plain format puts the `cmex10` font in all three styles of family 3.

Looking for a delimiter at a certain position in a certain font means

- if the character is large enough, accept it;
- if the character is *extendable*, accept it;

- otherwise, if the character has a *successor*, that is, it is part of a chain of increasingly bigger delimiters in the same font, try the successor.

Information about successors and extensibility of a delimiter is coded in the font metric file of the font. An extendable character has a top, a bottom, possibly a mid piece, and a piece which is repeated directly below the top piece, and directly above the bottom piece if there is a mid piece.

#### 21.2.4 `\big`, `\Big`, `\bigg`, and `\Bigg` delimiter macros

In order to be able to use a delimiter outside the `\left...\right` context, or to specify a delimiter of a different size than  $\TeX$  would have chosen, four macros for ‘big’ delimiters exist: `\big`, `\Big`, `\bigg`, and `\Bigg`. These can be used with anything that can follow `\left` or `\right`.

Twelve further macros (for instance `\bigl`, `\bigm`, and `\bigr`) force such delimiters in the context of an opening symbol, a binary relation, and a closing symbol respectively:

```
\def\bigl{\mathopen\big}
\def\bigm{\mathrel\big} \def\bigr{\mathclose\big}
```

The ‘big’ macros themselves put the requested delimiter and a null delimiter around an empty vertical box:

```
\def\big#1{\nulldelimiterspace=0pt \mathsurround=0pt
\hbox{${\left#1\right#1\to 8.5pt}\right.$}}
```

As an approximate measure, the `\Big` delimiters are one and a half times as large (11.5pt) as `\big` delimiters; `\bigg` ones are twice (14.5pt), and `\Bigg` ones are two and a half times as large (17.5pt).

## 21.3 Radicals

A *radical* is a compound of a left delimiter and an overlined math expression. The overlined expression is set in the cramped version of the surrounding style (see page 217).

In the plain format and the Computer Modern math fonts there is only one radical: the square root construct

```
\def\sqrt{\radical"270370 }
```

The control sequence `\radical` is followed by a ⟨24-bit number⟩ which specifies a small and a large variant of the left delimiter as was explained above. Joining the delimiter and the rule is done by letting the delimiter have a large depth, and a height which is equal to the desired rule thickness. The rule can then be placed on the current baseline. After the delimiter and the ruled expression

have been joined the whole is shifted vertically to achieve the usual vertical centring (see Chapter 23).

## 21.4 Math accents

Accents in math mode are specified by

```
\mathaccent⟨15-bit number⟩⟨math field⟩
```

Representing the 15-bit number as "xyzz, only the family y and the character position zz are used: an accented expression acts as `\mathord` expression (see Chapter 23).

In math mode whole expressions can be accented, whereas in text mode only characters can be accented. Thus in math mode accents can be stacked. However, the top accent may (or, more likely, will) not be properly positioned horizontally. Therefore the plain format has a macro `\skew` that effectively shifts the top accent. Its definition is

```
\def\skew#1#2#3{{#2{#3\mkern#1mu}\mkern-#1mu}{}}
```

and it is used for instance like

```
$_\skew4\hat{\hat x}$
```

which gives  $\hat{\hat x}$ .

For the correct positioning of accents over single characters the symbol and extension font have a `\skewchar`: this is the largest accent that adds to the width of an accented character. Positioning of any accent is based on the width of the character to be accented, followed by the skew character.

The skew characters of the Computer Modern math italic and symbol fonts are character "7F, 'Ä', and "30, '0', respectively. The `\defaultskewchar` value is assigned to the `\skewchar` when a font is loaded. In plain  $\TeX$  this is -1, so fonts ordinarily have no `\skewchar`.

Math accents can adapt themselves to the size of the accented expression:  $\TeX$  will look for a successor of an accent in the same way that it looks for a successor of a delimiter. In the Computer Modern math fonts this mechanism is used in the `\widehat` and `\widetilde` macros. For example,

```
\widehat x, \widehat{xy}, \widehat{xyz}
```

give

$$\widehat{x}, \widehat{xy}, \widehat{xyz}$$

respectively.

## 第 22 章    Fonts in Formulas

For math typesetting a single current font is not sufficient, as it is for text typesetting. Instead  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  uses several font families, and each family can contain three fonts. This chapter explains how font families are organized, and how  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  determines from what families characters should be taken.

`\fam`    The number of the current font family.

`\newfam`    Allocate a new math font family.

`\textfont`    Access the textstyle font of a family.

`\scriptfont`    Access the scriptstyle font of a family.

`\scriptscriptfont`    Access the scriptscriptstyle font of a family.

### 22.1    Determining the font of a character in math mode

The characters in math formulas can be taken from several different fonts (or better, *font families*) without any user commands. For instance, in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  math formulas use the roman font, the math italic font, the symbol font and the math extension font.

In order to determine from which font a character is to be taken,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  considers for each character in a formula its `\mathcode` (this is treated in Chapter 21). A `\mathcode` is a 15-bit number of the form "xyzz, where the hex digits have the following meaning:

x: class,

y: family,

zz: position in font.

In general only the family determines from what font a character is to be taken. The class of a math character is mostly used to control spacing and other aspects of typesetting. Typical classes include ‘relation’, ‘operator’, ‘delimiter’; see section 23.3 for details.

Class 7 is special in this respect: it is called ‘variable family’. If a character has a `\mathcode` of the form `"7yzz` it is taken from family `y`, unless the parameter `\fam` has a value in the range 0–15; then it is taken from family `\fam`.

## 22.2 Initial family settings

Both lowercase and uppercase letters are defined by `IniTEX` to have math codes `"71zz`, which means that they are of variable family, initially from family 1. As `TEX` sets `fam=-1`, that is, an invalid value, when a formula starts, characters are indeed taken from family 1, which in plain `TEX` is math italic.

Digits have math code `"70zz` so they are initially from family 0, in plain `TEX` the roman font. All other character codes have a mathcode assigned by `IniTEX` as

```
\mathcode x = x
```

which puts them in class 0, ordinary, and family 0, roman in plain `TEX`.

In plain `TEX`, commands such as `\sl` then set both a font and a family:

```
\def\sl{\fam\slfam\tensl}
```

so putting `\sl` in a formula will cause all letters, digits, and uppercase Greek characters, to change to slanted style.

In most cases, any font can be assigned to any family, but two families in `TEX` have a special meaning: these are families 2 and 3. For instance, their number of `\fontdimen` parameters is different from the usual 7. Family 2 needs 22 parameters, and family 3 needs 13. These parameters have all a very specialized meaning for positioning in math typesetting. Their meaning is explained below, but for the full story the reader is referred to appendix G of the `TEX` book.

## 22.3 Family definition

`TEX` can access 16 families of fonts in math mode; font families have numbers 0–15. The number of the current family is recorded in the parameter `\fam`.

The macro `\newfam` gives the number of an unused family. This number is assigned using `\chardef` to the control sequence.

Each font family can have a font meant for text style, script style, and scriptscript style. Below it is explained how `TEX` determines in what style a (sub-) formula is to be typeset.

Fonts are assigned to a family as follows:

```

\newfam\MyFam
\textfont\MyFam=\tfont \scriptfont\MyFam=\sfont
\scriptscriptfont\MyFam=\ssfont

```

for the text, script, and scriptscript fonts of a family. In general it is not necessary to fill all three members of a family (but it is for family 3). If  $\TeX$  needs a character from a family member that has not been filled, it uses the `\nullfont` instead, a primitive font that has no characters (nor a `.tfm` file).

## 22.4 Some specific font changes

### 22.4.1 Change the font of ordinary characters and uppercase Greek

All letters and the uppercase Greek characters are by default in plain  $\TeX$  of class 7, variable family, so changing `\fam` will change the font from which they are taken. For example

```
{\fam=9 x}
```

gives an `x` from family 9.

Uppercase Greek characters are defined by `\mathchardef` statements in the plain format as "70zz, that is, variable family, initially roman. Therefore, uppercase Greek character also change with the family.

### 22.4.2 Change uppercase Greek independent of text font

In the Computer Modern font layout, uppercase Greek letters are part of the roman font; see page 330. Therefore, introducing another text font (with another layout) will change the uppercase Greek characters (or even make them disappear). One way of remedying this is by introducing a new family in which the `cmr` font, which contains the uppercase Greek, resides. The control sequences accessing these characters then have to be redefined:

```

\newfam\Kgreek
\textfont\Kgreek=cmr10 ...
\def\hex#1{\ifcase#1\or 1\or 2\or 3\or 4\or 5\or 6\or
  7\or 8\or 9\or A\or B\or C\or D\or E\or F\fi}
\mathchardef\Gamma="0\hex\Kgreek00 % was: "0100
\mathchardef\Beta ="0\hex\Kgreek01 % was: "0101
\mathchardef\Gamma ...

```

Note, by the way, the absence of a either a space or a `\relax` token after `#1` in the definition of `\hex`. This implies that this macro can only be called with an argument that is a control sequence.

### 22.4.3 Change the font of lowercase Greek and mathematical symbols

Lowercase Greek characters have math code "01zz, meaning they are always from the math italic family. In order to change this one might redefine them, for instance `\mathchardef\alpha=10B`, to make them variable family. This is not done in plain  $\TeX$ , because the Computer Modern roman font does not have Greek lowercase, although it does have the uppercase characters.

Another way is to redefine them like `\mathchardef\alpha="0n0B` where *n* is the (hexadecimal) number of a family compatible with math italic, containing for instance a bold math italic font.

## 22.5 Assorted remarks

### 22.5.1 New fonts in formulas

There are two ways to access a font inside mathematics. After `\font\newfont=...` it is not possible to get the ‘a’ of the new font by `$...{\newfont a}...$` because  $\TeX$  does not look at the current font in math mode. What does work is

```
$ ... \hbox{\newfont a} ... $
```

but this precludes the use of the new font in script and scriptscript styles.

The proper solution takes a bit more work:

```
\font\newtextfont=...
\font\newscriptfont=... \font\newsscriptfont=...
\newfam\newfontfam
\textfont\newfontfam=\newtextfont
\scriptfont\newfontfam=\newscriptfont
\scriptscriptfont\newfontfam=\newsscriptfont
\def\newfont{\newtextfont \fam=\newfontfam}
```

after which the font can be used as

```
$... {\newfont a_{b_c}} ... $
```

in all three styles.

### 22.5.2 Evaluating the families

$\TeX$  will only look at what is actually in the `\textfont` et cetera of the various families at the end of the whole formula. Switching fonts in the families is thus not possible inside a single formula. The number of 16 families may therefore turn out to be restrictive for some applications.



## 第 23 章 Mathematics Typesetting

$\TeX$  has two math modes, display and non-display, and four styles, display, text, script, and scriptscript style, and every object in math mode belongs to one of eight classes. This chapter treats these concepts.

`\everymath` Token list inserted at the start of a non-display formula.

`\everydisplay` Token list inserted at the start of a display formula.

`\displaystyle` Select the display style of mathematics typesetting.

`\textstyle` Select the text style of mathematics typesetting.

`\scriptstyle` Select the script style of mathematics typesetting.

`\scriptscriptstyle` Select the scriptscript style of mathematics typesetting.

`\mathchoice` Give four variants of a formula for the four styles of mathematics typesetting.

`\mathord` Let the following character or subformula function as an ordinary object.

`\mathop` Let the following character or subformula function as a large operator.

`\mathbin` Let the following character or subformula function as a binary operation.

`\mathrel` Let the following character or subformula function as a relation.

`\mathopen` Let the following character or subformula function as an opening symbol.

`\mathclose` Let the following character or subformula function as a closing symbol.

`\mathpunct` Let the following character or subformula function as a punctuation symbol.

`\mathinner` Let the following character or subformula function as an inner formula.

`\mathaccent` Place an accent in math mode.

`\vcenter` Construct a vertical box, vertically centred on the math axis.

`\limits` Place limits over and under a large operator.

`\nolimits` Place limits of a large operator as subscript and superscript expressions.

`\displaylimits` Restore default placement for limits.

`\scriptspace` Extra space after subscripts and superscripts. Plain  $\TeX$  default: 0.5pt

`\nonscript` Cancel the next glue item if it occurs in scriptstyle or scriptscriptstyle.

`\mkern` Insert a kern measured in mu units.

`\mskip` Insert glue measured in mu units.

`\muskip` Prefix for skips measured in mu units.

`\muskipdef` Define a control sequence to be a synonym for a `\muskip` register.

`\newmuskip` Allocate a new muskip register.

`\thinmuskip` Small amount of mu glue.

`\medmuskip` Medium amount of mu glue.

`\thickmuskip` Large amount of mu glue.

`\mathsurround` Kern amount placed before and after in-line formulas.

`\over` Fraction.

`\atop` Place objects over one another.

`\above` Fraction with specified bar width.

`\overwithdelims` Fraction with delimiters.

`\atopwithdelims` Place objects over one another with delimiters.

`\abovewithdelims` Generalized fraction with delimiters.

`\underline` Underline the following  $\langle$ math symbol $\rangle$  or group.

`\overline` Overline the following  $\langle$ math symbol $\rangle$  or group.

`\relpenalty` Penalty for breaking after a binary relation not enclosed in a subformula. Plain  $\TeX$  default: 500

`\binoppenalty` Penalty for breaking after a binary operator not enclosed in a subformula. Plain  $\TeX$  default: 700

`\allowbreak` Macro for creating a breakpoint.

## 23.1 Math modes

$\TeX$  changes to *math mode* when it encounters a *math shift character*, category 3, in the input. After such an opening math shift it investigates (without expansion) the next token to see whether this is another math shift. In the latter case  $\TeX$  starts processing in *display math mode* until a closing double math shift is encountered:

..  $\$$  *displayed formula*  $\$$  ..

Otherwise it starts processing an in-line formula in *non-display math mode*:

..  $\$$  *in-line formula*  $\$$  ..

The single math shift character is a  $\langle$ horizontal command $\rangle$ .

Exception: displays are not possible in restricted horizontal mode, so inside an `\hbox` the sequence  $\$$  is an empty math formula and not the start of a displayed formula.

Associated with the two math modes are two  $\langle$ token parameter $\rangle$  registers (see also Chapter 14): at the start of an in-line formula the `\everymath` tokens are inserted; at the start of a displayed formula the `\everydisplay` tokens are inserted. Display math is treated further in the next chapter.

Math modes can be tested for: `\ifmmode` is true in display and non-display math mode, and `\ifinner` is true in non-display mode, but not in display mode.

## 23.2 Styles in math mode

Math formulas are set in any of eight *math styles*:

**D** display style,

**T** text style,

**S** script style,

**SS** scriptscript style,

and the four *cramped* styles variants  $D'$ ,  $T'$ ,  $S'$ ,  $SS'$  of these. The cramped styles differ mainly in the fact that superscripts are not raised as far as in the original styles.

### 23.2.1 Superscripts and subscripts

$\TeX$  can typeset a symbol or group as a superscript (or subscript) to the preceding symbol or group, if that preceding item does not already have a superscript (subscript). Superscripts (subscripts) are specified by the syntax

$\langle\text{superscript}\rangle\langle\text{math field}\rangle$

or

$\langle\text{subscript}\rangle\langle\text{math field}\rangle$

where a  $\langle\text{superscript}\rangle$  ( $\langle\text{subscript}\rangle$ ) is either a character of category 7 (8), or a control sequence `\let` to such a character. The plain format has the control sequences

```
\let\sp=^ \let\sb=_
```

as implicit superscript and subscript characters.

Specifying a superscript (subscript) expression as the first item in an empty math list is equivalent to specifying it as the superscript (subscript) of an empty expression. For instance,

$\mathcal{E}^{\{...\}}$  is equivalent to  $\mathcal{E}^{\{...\}}$

For  $\text{T}_{\text{E}}\text{X}$ 's internal calculations, superscript and subscript expressions are made wider by `\scriptspace`; the value of this in plain  $\text{T}_{\text{E}}\text{X}$  is 0.5pt.

### 23.2.2 Choice of styles

Ordering the four styles  $D$ ,  $T$ ,  $S$ , and  $SS$ , and considering the other four as mere variants, the style rules for math mode are as follows:

- In any style superscripts and subscripts are taken from the next smaller style. Exception: in display style they are taken in script style.
- Subscripts are always in the cramped variant of the style; superscripts are only cramped if the original style was cramped.
- In an  $\{...\over...\}$  formula in any style the numerator and denominator are taken from the next smaller style.
- The denominator is always in cramped style; the numerator is only in cramped style if the original style was cramped.
- Formulas under a `\sqrt` or `\overline` are in cramped style.

Styles can be forced by the explicit commands `\displaystyle`, `\textstyle`, `\scriptstyle`, and `\scriptscriptstyle`.

In display style and text style the `\textfont` of the current family is used, in scriptstyle the `\scriptfont` is used, and in scriptscriptstyle the `\scriptscriptfont` is used.

The primitive command

```
\mathchoice{D}{T}{S}{SS}
```

lets the user specify four variants of a formula for the four styles.  $\text{T}_{\text{E}}\text{X}$  constructs all four and inserts the appropriate one.

## 23.3 Classes of mathematical objects

Objects in math mode belong to one of eight *math classes*. Depending on the class the object may be surrounded by some amount of white space, or treated specially in some way. Commands exist to force symbols, or sequences of symbols, to act as belonging to a certain class. In the hexadecimal representation "xyzz the class is the  $\langle 3\text{-bit number} \rangle$  x.

This is the list of classes and commands that force those classes. The examples are from the plain format (see the tables starting at page 335).

1. *ordinary*: lowercase Greek characters and those symbols that are ‘just symbols’; the command `\mathord` forces this class.
2. *large operator*: integral and sum signs, and ‘big’ objects such as `\bigcap` or `\bigotimes`; the command `\mathop` forces this class. Characters that are large operators are centred vertically, and they may behave differently in display style from in the other styles; see below.
3. *binary operation*: plus and minus, and things such as `\cap` or `\otimes`; the command `\mathbin` forces this class.
4. *relation* (also called *binary relation*): equals, less than, and greater than signs, subset and superset, perpendicular, parallel; the command `\mathrel` forces this class.
5. *opening symbol*: opening brace, bracket, parenthesis, angle, floor, ceiling; the command `\mathopen` forces this class.
6. *closing symbol*: closing brace, bracket, parenthesis, angle, floor, ceiling; the command `\mathclose` forces this class.
7. *punctuation*: most punctuation marks, but `:` is a relation, the `\colon` is a punctuation colon; the command `\mathpunct` forces this class.
8. *variable family*: symbols in this class change font with the `\fam` parameter; in plain T<sub>E</sub>X uppercase Greek letters and ordinary letters and digits are in this class.

There is one further class: the *inner* subformulas. No characters can be assigned to this class, but characters and subformulas can be forced into it by `\mathinner`. The  $\langle \text{generalized fraction} \rangle$ s and `\left... \right` groups are inner formulas. Inner formulas are surrounded by some white space; see the table below.

Other subformulas than those that are inner are treated as ordinary symbols. In particular, subformulas enclosed in braces are ordinary: `$a+b$` looks like ‘ $a + b$ ’, but `$a{+}b$` looks like ‘ $a + b$ ’. Note, however, that in `${a+b}$` the

whole subformula is treated as an ordinary symbol, not its components; therefore the result is ‘ $a + b$ ’.

## 23.4 Large operators and their limits

The large operators in the Computer Modern fonts come in two sizes: one for text style and one for display style. Control sequences such as `\sum` are simply defined by `\mathchardef` to correspond to a position in a font:

```
\mathchardef\sum="1350
```

but if the current style is display style,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  looks to see whether that character has a successor in the font.

Large operators in text style behave as if they are followed by `\nolimits`, which places the limits as sub/superscript expressions after the operator:

$$\sum_{k=1}^{\infty}$$

In display style they behave as if they are followed by `\limits`, which places the limits over and under the operator:

$$\sum_{k=1}^{\infty}$$

The successor mechanism (see page 208) lets  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  take a larger variant of the delimiter here.

The integral sign has been defined in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  as

```
\mathchardef\intop="1352 \def\int{\intop\nolimits}
```

which places the limits after the operator, even in display style:

$$\int_0^{\infty} e^{-x^2} dx = \sqrt{\pi}/2$$

With `\limits\nolimits` or `\nolimits\limits` the last specification has precedence; the default placement can be restored by `\displaylimits`. For instance,

```
$ ... \sum\limits\displaylimits ... $
```

is equivalent to

```
$ ... \sum ... $
```

and

```
$$ ... \sum\nolimits\displaylimits ... $$
```

is equivalent to

```
$$$ ... \sum ... $$$
```

## 23.5 Vertical centring: `\vcenter`

Each formula has an *axis*, which is for an in-line formula about half the x-height of the surrounding text; the exact value is the `\fontdimen22` of the font in family 2, the symbol font, in the current style.

The bar line in fractions is placed on the axis; large operators, delimiters and `\vcenter` boxes are centred on it.

A `\vcenter` box is a vertical box that is arranged so that it is centred on the math axis. It is possible to give a spread or to specification with a `\vcenter` box.

The `\vcenter` box is allowed only in math mode, and it does not behave like other boxes; for instance, it can not be stored in a box register. It does not qualify as a `\box`. See page 146 for a macro that repairs this.

## 23.6 Mathematical spacing: `\mu` glue

Spacing around mathematical objects is measured in *math units*: multiples of a `\mu`. A `\mu` is 1/18th part of `\fontdimen6` of the font in family 2 in the current style, the *quad* value of the symbol font.

### 23.6.1 Classification of `\mu` glue

The user can specify `\mu` spacing by `\mkern` or `\mskip`, but most *mu glue* is inserted automatically by  $\TeX$ , based on the classes to which objects belong (see above). First, here are some rules of thumb describing the global behaviour.

- A `\thickmuskip` (default value in plain  $\TeX$ : 5`\mu` plus 5`\mu`) is inserted around (binary) relations, except where these are preceded or followed by other relations or punctuation, and except if they follow an open, or precede a close symbol.
- A `\medmuskip` (default value in plain  $\TeX$ : 4`\mu` plus 2`\mu` minus 4`\mu`) is put around binary operators.
- A `\thinmuskip` (default value in plain  $\TeX$ : 3`\mu`) follows after punctuation, and is put around inner objects, except where these are followed by a close or preceded by an open symbol, and except if the other object is a large operator or a binary relation.
- No `\mu` glue is inserted after an open or before a close symbol except where the latter is preceded by punctuation; no `\mu` glue is inserted also before

punctuation, except where the preceding object is punctuation or an inner object.

The following table gives the complete definition of mu glue between math objects.

	0:	1:	2:	3:	4:	5:	6:	
	Ord	Op	Bin	Rel	Open	Close	Punct	Inner
0: Ord	0	1	(2)	(3)	0	0	0	(1)
1: Op	1	1	*	(3)	0	0	0	(1)
2: Bin	(2)	(2)	*	*	(2)	*	*	(2)
3: Rel	(3)	(3)	*	0	(2)	*	*	(2)
4: Open	0	0	*	0	0	0	0	0
5: Close	0	1	(2)	(3)	0	0	0	(1)
6: Punct	(1)	(1)	*	(1)	(1)	(1)	(1)	(1)
Inner	(1)	1	(2)	(3)	(1)	0	(1)	(1)

where the symbols have the following meanings:

- 0, no space; 1, thin space; 2, medium space; 3, thick space;
- (·), insert only in text and display mode, not in script or scriptscript mode;
- cases \* cannot occur, because a Bin object is converted to Ord if it is the first in the list, preceded by Bin, Op, Open, Punct, Rel, or followed by Close, Punct, and Rel; also, a Rel is converted to Ord when it is followed by Close or Punct.

Stretchable mu glue is set according to the same rules that govern ordinary glue. However, only mu glue on the outer level can be stretched or shrunk; any mu glue enclosed in a group is set at natural width.

23.6.2 Muskip registers

Like ordinary glue, mu glue can be stored in registers, the \muskip registers, of which there are 256 in TeX. The registers are denoted by

`\muskip<8-bit number>`

and they can be assigned to a control sequence by

`\muskipdef<control sequence>⟨equals⟩<8-bit number>`

and there is a macro that allocates unused registers:

`\newmuskip<control sequence>`

Arithmetic for mu glue exists as for glue; see Chapter 8.



### 23.6.3 Other spaces in math mode

In math mode space tokens are ignored; however, the math code of the space character is "8000 in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ , so if its category is made ‘letter’ or ‘other character’, it will behave like an active character in math mode. See also page 207.

Admissible glue in math mode is of type ⟨mathematical skip⟩, which is either a ⟨horizontal skip⟩ (see Chapter 6) or `\mskip⟨muglue⟩`. Leaders in math mode can be specified with a ⟨mathematical skip⟩.

A glue item preceded by `\nonscript` is cancelled if it occurs in `scriptstyle` or `scriptscriptstyle`.

Control space functions in math mode as it does in horizontal mode.

In-line formulas are surrounded by kerns of size `\mathsurround`, the so-called ‘math-on’ and ‘math-off’ items. Line breaking can occur at the front of the math-off kern if it is followed by glue.

## 23.7 Generalized fractions

Fraction-like objects can be set with six primitive commands of type ⟨generalized fraction⟩. Each of these *generalized fractions* takes the preceding and the following subformulas and puts them over one another, if necessary with a fraction bar and with delimiters.

`\over` is the ordinary fraction; the bar thickness is `\fontdimen8` of the extension font:

`$\pi\over2$` gives  $\frac{\pi}{2}$

`\atop` is equivalent to a fraction with zero bar thickness:

`$\pi\atop2$` gives  $\frac{\pi}{2}$

`\above⟨dimen⟩` specifies the thickness of the bar line explicitly:

`$\pi\above 1pt 2$` gives  $\frac{\pi}{2}$

To each of these three there corresponds a `\dots withdelims` variant that lets the user specify delimiters for the expression. For example, the most general command, in terms of which all five others could have been defined, is

`\abovewithdelims⟨delim1⟩⟨delim2⟩⟨dimen⟩`.

Delimiters in these generalized fractions do not grow with the enclosed expression: in display mode a delimiter is taken which is at least `\fontdimen20` high, otherwise it has to be at least `\fontdimen21` high. These dimensions are taken from the font in family 2, the symbol font, in the current style.

The control sequences `\over`, `\atop`, and `\above` are primitives, although they could have been defined as `\...withdelims...`, that is, with two null delimiters. Because of these implied surrounding null delimiters, there is a kern of size `\nulldelimiterspace` before and after these simple generalized fractions.

## 23.8 Underlining, overlining

The primitive commands `\underline` and `\overline` take a  $\langle\text{math field}\rangle$  argument, that is, a  $\langle\text{math symbol}\rangle$  or a group, and draw a line under or over it. The result is an ‘Under’ or ‘Over’ atom, which is appended to the current math list. The line thickness is font dimension 8 of the extension font, which also determines the clearance between the line and the  $\langle\text{math field}\rangle$ .

Various other `\over...` and `\under...` commands exist in plain  $\text{\TeX}$ ; these are all macros that use the  $\text{\TeX}$  `\halign` command.

## 23.9 Line breaking in math formulas

In-line formulas can be broken after relations and binary operators. The respective *penalties* are the `\relpenalty` and the `\binoppenalty`. However,  $\text{\TeX}$  will only break after such symbols if they are not enclosed in braces. Other *breakpoints* can be created with `\allowbreak`, which is an abbreviation for `\penalty0`.

Unlike in horizontal or vertical mode where putting two penalties in a row is equivalent to just placing the smallest one, in math mode a penalty placed at a break point – that is, after a relation or binary operator – will effectively replace the old penalty by the new one.

## 23.10 Font dimensions of families 2 and 3

If a font is used in text mode,  $\text{\TeX}$  will look at its first 7 `\fontdimen` parameters (see page 52), for instance to control spacing. In math, however, more font dimensions are needed.  $\text{\TeX}$  will look at the first 22 parameters of the fonts in family 2, and the first 13 of the fonts in family 3, to control various aspects of math typesetting. The next two subsections have been quoted loosely from [3].

### 23.10.1 Symbol font attributes

Attributes of the font in family 2, the *symbol font*, mainly specify the initial vertical positioning of parts of fractions, subscripts, superscripts, et cetera.

The position determined by applying these attributes may be further modified because of other conditions, for example the presence of a fraction bar.

One text font dimension, number 6, the quad, determines the size of mu glue; see above.

Fraction numerator attributes: minimum shift up, from the main baseline, of the baseline of the numerator of a generalized fraction,

8. num1: for display style,
9. num2: for text style or smaller if a fraction bar is present,
10. num3: for text style or smaller if no fraction bar is present.

Fraction denominator attributes: minimum shift down, from the main baseline, of the baseline of the denominator of a generalized fraction,

11. denom1: for display style,
12. denom2: for text style or smaller.

Superscript attributes: minimum shift up, from the main baseline, of the baseline of a superscript,

13. sup1: for display style,
14. sup2: for text style or smaller, non-cramped,
15. sup3: for text style or smaller, cramped.

Subscript attributes: minimum shift down, from the main baseline, of the baseline of a subscript,

16. sub1: when no superscript is present,
17. sub2: when a superscript is present.

Script adjustment attributes: for use only with non-glyph, that is, composite, objects.

18. sup\_drop: maximum distance of superscript baseline below top of nucleus
19. sub\_drop: minimum distance of subscript baseline below bottom of nucleus.

Delimiter span attributes: height plus depth of delimiter enclosing a generalized fraction,

20. delim1: in display style,
21. delim2: in text style or smaller.

A parameter with many uses, the height of the math axis,

22. axis\_height: the height above the baseline of the fraction bar, and the centre of large delimiters and most operators and relations. This position is used in vertical centring operations.

### 23.10.2 Extension font attributes

Attributes of the font in family 3, the *extension font*, mostly specify the way the limits of large operators are set.

The first parameter, number 8, `default_rule_thickness`, serves many purposes. It is the thickness of the rule used for overlines, underlines, radical extenders (square root), and fraction bars. Various clearances are also specified in terms of this dimension: between the fraction bar and the numerator and denominator, between an object and the rule drawn by an underline, overline, or radical, and between the bottom of superscripts and top of subscripts.

Minimum clearances around large operators are as follows:

9. `big_op_spacing1`: minimum clearance between baseline of upper limit and top of large operator; see below.
10. `big_op_spacing2`: minimum clearance between bottom of large operator and top of lower limit.
11. `big_op_spacing3`: minimum clearance between baseline of upper limit and top of large operator, taking into account depth of upper limit; see below.
12. `big_op_spacing4`: minimum clearance between bottom of large operator and top of lower limit, taking into account height of lower limit; see below.
13. `big_op_spacing5`: clearance above upper limit or below lower limit of a large operator.

The resulting clearance above an operator is the maximum of parameter 7, and parameter 11 minus the depth of the upper limit. The resulting clearance below an operator is the maximum of parameter 10, and parameter 12 minus the height of the lower limit.

### 23.10.3 Example: subscript lowering

The location of a subscript depends on whether there is a superscript; for instance

$$X_1 + Y_1^2 = 1$$

If you would rather have that look like

$$X_1 + Y_1^2 = 1,$$

it suffices to specify

```
\fontdimen16\textfont2=3pt \fontdimen17\textfont2=3pt
```

which makes the subscript drop equal in both cases. Since font dimension assignments are global, you have to specify this only once in your document.

## 第 24 章 Display Math

Displayed formulas are set on a line of their own, usually somewhere in a paragraph. This chapter explains how surrounding white space (both above/below and to the left/right) is calculated.

`\abovedisplayskip` `\belowdisplayskip` Glue above/below a display. Plain

`TEX` default: 12pt plus 3pt minus 9pt

`\abovedisplayshortskip` `\belowdisplayshortskip` Glue above/below a display if the line preceding the display was short. Plain `TEX`

defaults: 0pt plus 3pt and 7pt plus 3pt minus 4pt respectively.

`\predisplaypenalty` `\postdisplaypenalty` Penalty placed in the vertical list above/below a display. Plain `TEX` defaults: 10 000 and 0 respectively.

`\displayindent` Distance by which the box, in which the display is centred, is indented owing to hanging indentation.

`\displaywidth` Width of the box in which the display is centred.

`\predisplaysize` Effective width of the line preceding the display.

`\everydisplay` Token list inserted at the start of a display.

`\eqno` Place a right equation number in a display formula.

`\leqno` Place a left equation number in a display formula.

### 24.1 Displays

`TEX` starts building a *display math* formula when it encounters two math shift characters (characters of category 3, \$ in plain `TEX`) in a row. Another such pair (possibly followed by one optional space) indicates the end of the display.

Math shift is a `\math` (horizontal command), but displays are only allowed in unrestricted horizontal mode (`$$` is an empty math formula in restricted horizontal mode). Displays themselves, however, are started in the surrounding (possibly

internal) vertical mode in order to calculate quantities such as `\prevgraf`; the result of the display is appended to the vertical list.

The part of the paragraph above the display is broken into lines as an independent paragraph (but `\prevgraf` is carried over; see below), and the remainder of the paragraph is set, starting with an empty list and `\spacefactor` equal to 1000. The `\everypar` tokens are not inserted for the part of the paragraph after the display, nor is `\parskip` glue inserted.

Right at the beginning of the display the `\everydisplay` token list is inserted (but after the calculation of `\displayindent`, `\displaywidth`, and `\predisplaysize`). See page 231 for an example of the use of `\everydisplay`.

The page builder is exercised before the display (but after the `\everydisplay` tokens have been inserted), and after the display finishes.

The ‘display style’ of math typesetting was treated in Chapter 22.

## 24.2 Displays in paragraphs

Positioning of a display in a paragraph may be influenced by hanging indentation or a `\parshape` specification. For this,  $\TeX$  uses the `\prevgraf` parameter (see Chapter 18), and acts as if the display is three lines deep.

If  $n$  is the value of `\prevgraf` when the display starts – so there are  $n$  lines of text above the display – `\prevgraf` is set to  $n + 3$  when the paragraph resumes. The display occupies, as it were, lines  $n + 1$ ,  $n + 2$ , and  $n + 3$ . The shift and line width for the display are those that would hold for line  $n + 2$ .

The shift for the display is recorded in `\displayindent`; the line width is recorded in

`\displaywidth`. These parameters (and the `\predisplaysize` explained below) are set immediately after the `$$` has been scanned. Usually they are equal to zero and `\hsize` respectively. The user can change the values of these parameters;  $\TeX$  will use the values that hold after the math list of the display has been processed.

A display is vertical material, and therefore not influenced by settings of `\leftskip` and `\rightskip`.

## 24.3 Vertical material around displays

A display is preceded in the vertical list by

- a penalty of size `\predisplaypenalty` (plain  $\TeX$  default 10 000), and

- glue of size `\abovedisplayskip` or `\abovedisplayshortskip`; this glue is omitted in cases where a `\leqno` equation number is set on a line of its own (see below).

A display is followed by

- a penalty of size `\postdisplaypenalty` (default 0), and possibly
- glue of size `\belowdisplayskip` or `\belowdisplayshortskip`; this glue is omitted in cases where an `\eqno` equation number is set on a line of its own (see below).

The ‘short’ variants of the glue are taken if there is no `\leqno` left equation number, and if the last line of the paragraph above the display is short enough for the display to be raised a bit without coming too close to that line. In order to decide this, the effective width of the preceding line is saved in `\predisplaysize`. This value is calculated immediately after the opening `$$` of the display has been scanned, together with the `\displaywidth` and `\displayindent` explained above.

Remembering that the part of the paragraph above the display has already been broken into lines, the following method for finding the effective width of the last line ensues.  $\text{\TeX}$  takes the last box of the list, which is a horizontal box containing the last line, and locates the right edge of the last box in it. The `\predisplaysize` is then the place of that rightmost edge, plus any amount by which the last line was shifted, plus two ems in the current font.

There are two exceptions to this. The `\predisplaysize` is taken to be `-\maxdimen` if there was no previous line, that is, the display started the paragraph, or it followed another display; `\predisplaysize` is taken to be `\maxdimen` if the glue in the last line was not set at its natural width, which may happen if the `\parfillskip` contained only finite stretch. The reason for the last clause is that glue setting is slightly *machinedependent*, and such dependences should be kept out of  $\text{\TeX}$ ’s global decision processes.

## 24.4 Glue setting of the display math list

The display has to fit in `\displaywidth`, but in addition to the formula there may be an equation number. The minimum separation between the formula and the equation number should be one em in the symbol font, that is, `\fontdimen6\textfont2`.

If the formula plus any equation number and separation fit into `\displaywidth`, the glue in the formula is set at its natural width. If it does not fit, but the formula contains enough shrink, it is shrunk. Otherwise  $\text{\TeX}$  puts any equa-

tion number on a line of its own, and the glue in the formula is set to fit it in `\displaywidth`. With the equation number on a separate line the formula may now very well fit in the display width; however, if it was a very long formula the box in which it is set may still be overfull.  $\text{\TeX}$  never breaks a displayed formula.

## 24.5 Centring the display formula: displacement

Based on the width of the box containing the formula – which may not really ‘contain’ it; it may be overfull –  $\text{\TeX}$  tries to centre the formula in the `\displaywidth`, that is, without taking the equation number into account. Initially, a displacement is calculated that is half the difference between `\displaywidth` and the width of the formula box.

However, if there is an equation number that will not be put on a separate line and the displacement is less than twice the width of the equation number, a new displacement is calculated. This new displacement is zero if the formula started with glue; otherwise it is such that the formula box is centred in the space left by the equation number.

If there was no equation number, or if the equation number will be put on a separate line, the formula box is now placed, shifted right by `\displayindent` plus the displacement calculated above.

## 24.6 Equation numbers

The user can specify a equation number for a display by ending it with

`\eqno<math mode material>$$`

for an equation number placed on the right, or

`\leqno<math mode material>$$`

for an equation number placed on the left.

### 24.6.1 Ordinary equation numbers

Above it was described how  $\text{\TeX}$  calculates a displacement from the display formula and the equation number, if this is to be put on the same line as the formula.

If the equation number was a `\leqno` number,  $\text{\TeX}$  places a box containing

- the equation number,



- a kern with the size of the displacement calculated, and
- the formula.

This box is shifted right by `\displayindent`.

If the equation number was an `\eqno` number,  $\TeX$  places a box containing

- the formula,
- a kern with the size of the displacement calculated, and
- the equation number.

This box is shifted right by `\displayindent` plus the displacement calculated.

### 24.6.2 The equation number on a separate line

Since displayed formulas may become rather big,  $\TeX$  can decide (as was described above) that any equation number should be placed on a line of its own. A left-placed equation number is then to be placed above the display, in a box that is shifted right by `\displayindent`; a right-placed equation number will be placed below the display, in a box that is shifted to the right by `\displayindent` plus `\displaywidth` minus the width of the equation number box.

In both cases a penalty of 10 000 is placed between the equation number box and the formula.

$\TeX$  does not put extra glue above a left-placed equation number or below a right-placed equation number;  $\TeX$  here relies on the `baselineskip` mechanism.

## 24.7 Non-centred displays

As a default,  $\TeX$  will center displays. In order to get *non-centred displays* some macro trickery is needed.

One approach would be to write a macro `\DisplayEquation` that would basically look like

```
\def\DisplayEquation#1{%
  \par \vskip\abovedisplayskip
  \hbox{\kern\parindent$\displaystyle#1$}
  \vskip\belowdisplayskip \noindent}
```

but it would be nicer if one could just write

```
$$ ... \eqno ... $$
```

and having this come out as a leftaligning display.

Using the `\everydisplay` token list, the above idea can be realized. The basic idea is to write

```
\everydisplay{\IndentedDisplay}
\def\IndentedDisplay#1${ ...
```

so that the macro `\IndentedDisplay` will receive the formula, including any equation number. The first step is now to extract an equation number if it is present. This makes creative use of delimited macro parameters.

```
\def\ExtractEqNo#1\eqno#2\eqno#3\relax
  {\def\Equation{#1}\def\EqNo{#2}}
\def\IndentedDisplay#1${%
  \ExtractEqNo#1\eqno\eqno\relax
```

Next the equation should be set in the available space `\displaywidth`:

```
\hbox to \displaywidth
  {\kern\parindent
   $\displaystyle\Equation$\hfil$\EqNo$}$$
}
```

Note that the macro ends in the closing `$$` to balance the opening dollars that caused insertion of the `\everydisplay` tokens. This also means that the box containing the displayed material will automatically be surrounded by `\abovedisplayskip` and `\belowdisplayskip` glue. There is no need to use `\displayindent` anywhere in this macro, because  $\TeX$  itself will shift the display appropriately.

## 第 25 章 Alignment

$\text{\TeX}$  provides a general alignment mechanism for making *tables*.

$\backslash\text{halign}$  Horizontal alignment.

$\backslash\text{valign}$  Vertical alignment.

$\backslash\text{omit}$  Omit the template for one alignment entry.

$\backslash\text{span}$  Join two adjacent alignment entries.

$\backslash\text{multispan}$  Macro to join a number of adjacent alignment entries.

$\backslash\text{tabskip}$  Amount of glue in between columns (rows) of an  $\backslash\text{halign}$  ( $\backslash\text{valign}$ ).

$\backslash\text{noalign}$  Specify vertical (horizontal) material to be placed in between rows (columns) of an  $\backslash\text{halign}$  ( $\backslash\text{valign}$ ).

$\backslash\text{cr}$  Terminate an alignment line.

$\backslash\text{crrc}$  Terminate an alignment line if it has not already been terminated by  $\backslash\text{cr}$ .

$\backslash\text{everycr}$  Token list inserted after every  $\backslash\text{cr}$  or non-redundant  $\backslash\text{crrc}$ .

$\backslash\text{centering}$  Glue register in plain  $\text{\TeX}$  for centring  $\backslash\text{eqalign}$  and  $\backslash\text{eqalignno}$ .  
Value: 0pt plus 1000pt minus 1000pt

$\backslash\text{hideskip}$  Glue register in plain  $\text{\TeX}$  to make alignment entries invisible.  
Value: -1000pt plus 1fill

$\backslash\text{hidewidth}$  Macro to make preceding or following entry invisible.

### 25.1 Introduction

$\text{\TeX}$  has a sophisticated alignment mechanism, based on templates, with one template entry per column or row. The templates may contain any common elements of the table entries, and in general they contain instructions for typesetting the entries.  $\text{\TeX}$  first calculates widths (for  $\backslash\text{halign}$ ) or heights (for

`\valign`) of all entries; then it typesets the whole alignment using in each column (row) the maximum width (height) of entries in that column (row).

## 25.2 Horizontal and vertical alignment

The two alignment commands in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  are

`\halign<box specification>{<alignment material>}`

for horizontal alignment of columns, and

`\valign<box specification>{<alignment material>}`

for vertical alignment of rows. `\halign` is a  $\langle$ vertical command $\rangle$ , and `\valign` is a  $\langle$ horizontal command $\rangle$ .

The braces induce a new level of grouping; they can be implicit.

The discussion below will mostly focus on horizontal alignments, but, replacing ‘column’ by ‘row’ and vice versa, it applies to vertical alignments too.

### 25.2.1 Horizontal alignments: `\halign`

A *horizontal alignment* yields a list of horizontal boxes, the rows, which are placed on the surrounding vertical list. The page builder is exercised after the alignment rows have been added to the vertical list. The value of `\prevdepth` that holds before the alignment is used for the baselineskip of the first row, and after the alignment `\prevdepth` is set to a value based on the last row.

Each entry is processed in a group of its own, in restricted horizontal mode.

A special type of horizontal alignment exists: the *display alignment*, specified as

`$$<assignments>\halign<box specification>{...}<assignments>$$`

Such an alignment is shifted by `\displayindent` (see Chapter 24) and surrounded by

`\abovedisplayskip` and `\belowdisplayskip` glue.

### 25.2.2 Vertical alignments: `\valign`

A *vertical alignment* can be considered as a ‘rotated’ horizontal alignments: they are placed on the surrounding horizontal lists, and yield a row of columns. The `\spacefactor` value is treated the same way as the `\prevdepth` for horizontal alignments: the value current before the alignment is used for the first column, and the value reached after the last column is used after the alignment. In between columns the `\spacefactor` value is 1000.

Each entry is in a group of its own, and it is processed in internal vertical mode.

### 25.2.3 Material between the lines: `\noalign`

Material that has to be contained in the alignment, but should not be treated as an entry or series of entries, can be given by

```
\noalign<filler>{<vertical mode material>}
```

for horizontal alignments, and

```
\noalign<filler>{<horizontal mode material>}
```

for vertical alignments.

Examples are

```
\noalign{\hrule}
```

for drawing a horizontal rule between two lines of an `\halign`, and

```
\noalign{\penalty100}
```

for discouraging a page break (or line break) in between two rows (columns) of an `\halign` (`\valign`).

### 25.2.4 Size of the alignment

The `<box specification>` can be used to give the alignment a predetermined size: for instance

```
\halign to \hsize{ ... }
```

Glue contained in the entries of the alignment has no role in this; any stretch or shrink required is taken from the `\tabskip` glue. This is explained below.

## 25.3 The preamble

Each line in an alignment is terminated by `\cr`; the first line is called the *template line*. It is of the form

$$u_1\#v_1\&\dots\&u_n\#v_n\backslash\mathrm{cr}$$

where each  $u_i$ ,  $v_i$  is a (possibly empty) arbitrary sequence of tokens, and the template entries are separated by the *alignment tab* character (`&` in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ ), that is, any character of category 4.

A  $u_i\#v_i$  sequence is the template that will be used for the  $i$ th column: whatever sequence  $\alpha_i$  the user specifies as the entry for that column will be inserted at the parameter character. The sequence  $u_i\alpha_i v_i$  is then processed to obtain the actual entry for the  $i$ th column on the current line. See below for more details.

The length  $n$  of the template line need not be equal to the actual number of columns in the alignment: the template is used only for as many items as are specified on a line. Consider as an example

```
\halign{a#&b#&c#\cr 1&2\cr 1\cr}
```

which has a three-item template, but the rows have only one or two items. The output of this is

```
a1b2
a1
```

### 25.3.1 Infinite preambles

For the case where the number of columns is not known in advance, for instance if the alignment is to be used in a macro where the user will specify the columns, it is possible to specify that a trailing piece of the preamble can be repeated arbitrarily many times. By preceding it with `&`, an entry can be marked as the start of this repeatable part of the preamble. See the example of `\matrix` below.

When the whole preamble is to be repeated, there will be an alignment tab character at the start of the first entry:

```
\halign{& ... & ... \cr ... }
```

If a starting portion of the preamble is to be exempted from repetition, a double alignment tab will occur:

```
\halign{ ... & ... & ... && ... & ... \cr ... }
```

The repeatable part need not be used an integral number of times. The alignment rows can end at any time; the rest of the preamble is then not used.

### 25.3.2 Brace counting in preambles

Alignments may appear inside alignments, so  $\text{\TeX}$  uses the following rule to determine to which alignment an `&` or `\cr` control sequence belongs:

All tab characters and `\cr` tokens of an alignment should be on the same level of grouping.

From this it follows that tab characters and `\cr` tokens can appear inside an entry if they are nested in braces. This makes it possible to have nested alignments.

### 25.3.3 Expansion in the preamble

All tokens in the preamble – apart from the tab characters – are stored for insertion in the entries of the alignment, but a token preceded by `\span` is expanded while the preamble is scanned. See below for the function of `\span` in the rest of the alignment.

### 25.3.4 `\tabskip`

Entries in an alignment are set to take the width of the largest element in their column. Glue for separating columns can be specified by assigning to `\tabskip`.  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  inserts this glue in between each pair of columns, and before the first and after the last column.

The value of `\tabskip` that holds outside the alignment is used before the first column, and after all subsequent columns, unless the preamble contains assignments to `\tabskip`. Any assignment to `\tabskip` is executed while  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is scanning the preamble; the value that holds when a tab character is reached will be used at that place in each row, and after all subsequent columns, unless further assignments occur. The value of `\tabskip` that holds when `\cr` is reached is used after the last column.

Assignments to `\tabskip` in the preamble are local to the alignment, but not to the entry where they are given. These assignments are ordinary glue assignments: they remove any optional trailing space.

As an example, in the following table there is no `\tabskip` glue before the first and after the last column; in between all columns there is stretchable `\tabskip`.

```
\tabskip=Opt \halign to \hsize{
  \vrule#\tabskip=Opt plus 1fil\strut&
  \hfil#\hfil& \vrule#& \hfil#\hfil& \vrule#& \hfil#\hfil&
  \tabskip=Opt\vrule#\cr
\noalign{\hrule}
&\multispan5\hfil Just a table\hfil&\cr
\noalign{\hrule}
&one&&two&&three&\cr &a&&b&&c&\cr
\noalign{\hrule}
}
```

The result of this is

Just a table				
one		two		three
a		b		c

All of the vertical rules of the table are in a separate column. This is the only

way to get the space around the items to stretch.

## 25.4 The alignment

After the template line any number of lines terminated by `\cr` can follow.  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  reads all of these lines, processing the entries in order to find the maximal width (height) in each column (row). Because all entries are kept in memory, long tables can overflow  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 's main memory. For such tables it is better to write a special-purpose macro.

### 25.4.1 Reading an entry

Entries in an alignment are composed of the constant  $u$  and  $v$  parts of the template, and the variable  $\alpha$  part. Basically  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  forms the sequence of tokens  $u\alpha v$  and processes this. However, there are two special cases where  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  has to expand before it forms this sequence.

Above, the `\noalign` command was described. Since this requires a different treatment from other alignment entries,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  expands, after it has read a `\cr`, the first token of the first  $\alpha$  string of the next line to see whether that is or expands to `\noalign`. Similarly, for all entries in a line the first token is expanded to see whether it is or expands to `\omit`. This control sequence will be described below.

Entries starting with an `\if...` conditional, or a macro expanding to one, may be misinterpreted owing to this premature expansion. For example,

```
\halign{##$\cr \ifmmode a\else b\fi\cr}
```

will give

$b$

because the conditional is evaluated before math mode has been set up. The solution is, as in many other cases, to insert a `\relax` control sequence to stop the expansion. Here the `\relax` has to be inserted at the start of the alignment entry.

If neither `\noalign` nor `\omit` (see below) is found,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  will process an input stream composed of the  $u$  part, the  $\alpha$  tokens (which are delimited by either `&` or `\span`, see below), and the  $v$  part.

Entries are delimited by `&`, `\span`, or `\cr`, but only if such a token occurs on the same level of grouping. This makes it possible to have an alignment as an entry of another alignment.



### 25.4.2 Alternate specifications: `\omit`

The template line will rarely be sufficient to describe all lines of the alignment. For lines where items should be set differently the command `\omit` exists: if the first token in an entry is (or expands to) `\omit` the trivial template `#` is used instead of what the template line specifies.

例子: *The following alignment uses the same template for all columns, but in the second column an `\omit` command is given.*

```
\tabskip=1em
\halign{&${<#>}\cr a&\omit (b)&c \cr}
```

*The output of this is*

$< a > \quad (b) \quad < c >$

### 25.4.3 Spanning across multiple columns: `\span`

Sometimes it is desirable to have material spanning several columns. The most obvious example is that of a heading above a table. For this  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  provides the `\span` command.

Entries are delimited either by `&`, by `\cr`, or by `\span`. In the last case  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  will omit the `\tabskip` glue that would normally follow the entry thus delimited, and it will typeset the material just read plus the following entry in the joint space available.

As an example,

```
\tabskip=1em
\halign{&#\cr a&b&c&d\cr a&\hrulefill\span\hrulefill&d\cr}
```

gives

a   b   c   d  
a   ———   d

Note that there is no `\tabskip` glue in between the two spanned columns, but there is `\tabskip` glue before the first column and after the last.

Using the `\omit` command this same alignment could have been generated as

```
\halign{&#\cr a&b&c&d\cr a&\hrulefill\span\omit&d\cr}
```

The `\span\omit` combination is used in the plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  macro `\multispan`: for instance

```
\multispan4 gives \omit\span\omit\span\omit\span\omit
```

which spans across three tabs, and removes the templates of four entries. Repeating the above example once again:

```
\halign{&#\cr a&b&c&d\cr a&\multispan2\hrulefill&d\cr}
```

The argument of `\multispan` is a single token, not a number, so in order to span more than 9 columns the argument should be enclosed in braces, for instance `\multispan{12}`. Furthermore, a space after a single-digit argument will wind up in the output.

For a ‘low budget’ solution to spanning columns plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  has the macro `\hidewidth`, defined by

```
\newskip\hideskip \hideskip=-1000pt plus 1fill
\def\hidewidth{\hskip\hideskip}
```

Putting `\hidewidth` at the beginning or end of an alignment entry will make its width zero, with the material in the entry sticking out to the left or right respectively.

#### 25.4.4 Rules in alignments

Horizontal rules inside a horizontal alignment will mostly be across the width of the alignment. The easiest way to attain this is to use

```
\noalign{\hrule}
```

lines inside the alignment. If the alignment is contained in a vertical box, lines above and below the alignment can be specified with

```
\vbox{\hrule \halign{...} \hrule}
```

The most general way to get horizontal lines in an alignment is to use

```
\multispan n\hrulefill
```

which can be used to underline arbitrary adjacent columns.

Vertical rules in alignments take some more care. Since a horizontal alignment breaks up into horizontal boxes that will be placed on a vertical list,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  will insert `baselineskip` glue in between the rows of the alignment. If vertical rules in adjacent rows are to abut, it is necessary to prevent `baselineskip` glue, for instance by the `\offinterlineskip` macro.

In order to ensure that rows will still be properly spaced it is then necessary to place a *strut* somewhere in the preamble. A strut is an invisible object with a certain height and depth. Putting that in the preamble guarantees that every line will have at least that height and depth. In the plain format `\strut` is defined statically as

```
\vrule height8.5pt depth3.5pt width0pt
```

so this must be changed when other fonts or sizes are used.

It is a good idea to use a whole column for a vertical rule, that is, to write

```
\vrule##
```

in the preamble and to leave the corresponding entry in the alignment empty. Omitting the vertical rule can then be done by specifying `\omit`, and the size of the rule can be specified explicitly by putting, for instance, `height 15pt` in the entry instead of leaving it empty. Of course, `tabskip glue` will now be specified to the left and right of the rule, so some extra `tabskip` assignments may be needed in the preamble.

### 25.4.5 End of a line: `\cr` and `\crcr`

All lines in an alignment are terminated by the `\cr` control sequence, including the last line.  $\TeX$  is not able to infer from a closing brace in the  $\alpha$  part that the alignment has ended, because an unmatched closing brace is perfectly valid in an alignment entry; it may match an opening brace in the  $u$  part of the corresponding preamble entry.

$\TeX$  has a primitive command `\crcr` that is equivalent to `\cr`, but it has no effect if it immediately follows a `\cr`. Consider as an example the definition in plain  $\TeX$  of `\cases`:

```
\def\cases#1{%
  \left\{\,\,\vcenter{\normalbaselines\m@th
    \ialign{\ $##\hfil$& \quad##\hfil \crcr #1\crcr}}\,%
  \right.}
```

Because of the `\crcr` after the user argument `#1`, the following two applications of this macro

```
\cases{1&2\cr 3&4} and \cases{1&2\cr 3&4\cr}
```

both work. In the first case the `\crcr` in the macro definition ends the last line; in the second case the user's `\cr` ends the line, and the `\crcr` is redundant.

After `\cr` and after a non-redundant `\crcr` the  $\langle$ token parameter $\rangle$  `\everycr` is inserted. This includes the `\cr` terminating the template line.

## 25.5 Example: math alignments

The plain format has several alignment macros that function in math mode. One example is `\matrix`, defined by

```
\def\matrix#1{\null\,\vcenter{\normalbaselines\m@th
  \ialign{\hfil$##$\hfil && \quad\hfil$##$\hfil\crcr
    \mathstrut\crcr
    \noalign{\kern-\baselineskip}
    #1\crcr
    \mathstrut\crcr
    \noalign{\kern-\baselineskip}}}\,,}
```

This uses a repeating (starting with `&&`) second preamble entry; each entry is centred by an `\hfil` before and after it, and there is a `\quad` of space in between columns. `Tabskip` glue was not used for this, because there should not be any glue preceding or following the matrix.

The combination of a `\mathstrut` and `\kern-\baselineskip` above and below the matrix increases the vertical size such that two matrices with the same number of rows will have the same height and depth, which would not otherwise be the case if one of them had subscripts in the last row, but the other not. The `\mathstrut` causes interline glue to be inserted and, because it has a size equal to `\baselineskip`, the negative kern will effectively leave only the interline glue, thereby buffering any differences in the first and last line. Only to a certain point, of course: objects bigger than the opening brace will still result in a different height or depth of the matrix.

Another, more complicated, example of an alignment for math mode is `\eq-alignno`.

```
\def\eqalignno#1{\begin{disp}l@y \tabskip\centering
  \halign to\displaywidth{
    \hfil$\@lign\displaystyle{##}$%    -- first column
      \tabskip\z@skip
    & $\@lign\displaystyle{##}$\hfil% -- second column
      \tabskip\centering
    & \llap{$\@lign##$}%                -- third column
      \tabskip\z@skip\crcr    %    end of the preamble
  #1\crcr}}
```

Firstly, the `tabskip` is set to zero after the equation number, so this number is set flush with the right margin. Since it is placed by `\llap`, its effective width is zero. Secondly, the `tabskip` between the first and second columns is also zero, and the `tabskip` before the first column and after the second is `\centering`, which is `0pt` plus `1000pt` minus `1000pt`, so the first column and second are jointly centred in the `\hsize`. Note that, because of the minus `1000pt`, these two columns will happily go outside the left and right margins, overwriting any equation numbers.

## 第 26 章 Page Shape

This chapter treats some of the parameters that determine the size of the page and how it appears on paper.

`\topskip` Minimum distance between the top of the page box and the baseline of the first box on the page. Plain  $\TeX$  default: 10pt

`\hoffset` `\voffset` Distance by which the page is shifted right/down with respect to the reference point.

`\vsize` Height of the page box. Plain  $\TeX$  default: 8.9in

`\maxdepth` Maximum depth of the page box. Plain  $\TeX$  default: 4pt

`\splitmaxdepth` Maximum depth of a box split off by a `\vsplit` operation. Plain  $\TeX$  default: `\maxdimen`

### 26.1 The reference point for global positioning

The *page positioning* on the paper is governed by a  $\TeX$  convention, to which output device drivers must adhere, that the top left point of the page is one inch from the page edges. Unfortunately this may lead to lots of trouble, for instance if a printer (or the page description language it uses) takes, say, the *lower* left corner as the reference point, and is factory set to US paper sizes, but is used with European standard A4 paper.

The page is shifted on the paper if one assigns non-zero values to `\hoffset` or `\voffset`: positive values shift to the right and down respectively.

### 26.2 `\topskip`

The `\topskip` ensures to a certain point that the first baseline of a page will be at the same location from page to page, even if font sizes are switched between pages or if the first line has no ascenders.

Before the first box on each page some glue is inserted. This glue has the same stretch and shrink as `\topskip`, but the natural size is the natural size of `\topskip` minus the height of the first box, or zero if this would be negative.

Plain  $\text{\TeX}$  sets `\topskip` to 10pt. Thus the top lines of pages will have their baselines at the same place if the top portion of the characters is ten point or less. For the Computer Modern fonts this condition is satisfied if the font size is less than (about) 13 points; for larger fonts the baseline of the top line will drop.

The height of the page box for a page containing only text (and assuming a zero `\parskip`) will be the `\topskip` plus a number of times the `\baselineskip`. Thus one can define a macro to compute the `\vsize` from the number of lines on a page:

```
\def\HeightInLines#1{\count@=#1\relax
  \advance\count@ by -1\relax
  \vsize=\baselineskip
  \multiply\vsize by \count@
  \advance\vsize by \topskip}
```

Calculating the `\vsize` this way will prevent underfull boxes for text-only pages.

In cases where the page does not start with a line of text (for instance a rule), the `\topskip` may give unwanted effects. To prevent these, start the page with

```
\hbox{}\kern-\topskip
```

followed by what you wanted on top.

Analogous to the `\topskip`, there is a `\splittopskip` for pages generated by a `\vsplit` operation; see the next chapter.

## 26.3 Page height and depth

$\text{\TeX}$  tries to build pages as a `\vbox` of height `\vsize`; see also `\pagegoal` in the next chapter.

If the last item on a page has an excessive depth, that page would be noticeably longer than other pages. To prevent this phenomenon  $\text{\TeX}$  uses `\maxdepth` as the maximum depth of the page box. If adding an item to the page would make the depth exceed this quantity, then the reference point of the page is moved down to make the depth exactly `\maxdepth`.

The ‘raggedbottom’ effect is obtained in plain  $\text{\TeX}$  by giving the `\topskip` some finite stretchability: 10pt plus 60pt. Thus the natural height of box 255 can vary when it reaches the output routine. Pages are then shipped out (more or less) as

```
\dimen0=\dp255 \unvbox255  
\ifraggedbottom \kern-\dimen0 \vfil \fi
```

The `\vfil` causes the topskip to be set at natural width, so the effect is one of a fixed top line and a variable bottom line of the page.

Before `\box255` is unboxed in the plain  $\text{\TeX}$  output routine, `\boxmaxdepth` is set to `\maxdepth` so that this box will be made under the same assumptions that the page builder used when putting together `\box255`.

The depth of box split off by a `\vsplit` operation is controlled by the `\splitmaxdepth` parameter.

## 第 27 章 分页

本章讨论‘页面构建器’：**T<sub>E</sub>X** 确定在何处将主竖直列分为多个页面的模块。页面构建器在输出例程之前运行，它将所得结果放入 `\box255` 以送给输出例程。

`\vsplit` 分割出一个盒子的顶端部分。它可与分页作对比。

`\splittopskip` 在 `\vsplit` 分割出的盒子中，第一个项目到盒子顶部的最小距离。

**Plain T<sub>E</sub>X** 默认为 10pt。

`\pagegoal` 页面盒子的目标高度。它刚开始等于 `\vsize`，并且根据插入项的高度逐步减少。

`\pagetotal` 当前页面积累的自然高度。

`\pagedepth` 当前页面的深度。

`\pagestretch` 当前页面积累的零阶可伸长度。

`\pagefilstretch` 当前页面积累的一阶可伸长度。

`\pagefillstretch` 当前页面积累的二阶可伸长度。

`\pagefilllstretch` 当前页面积累的三阶可伸长度。

`\pageshrink` 当前页面积累的可收缩量。

`\outputpenalty` 在当前分页点的惩罚值，若不在惩罚处分页，它等于 10 000。

`\interlinepenalty` 在段落内部两行间分页的惩罚值。**Plain T<sub>E</sub>X** 默认为 0。

`\clubpenalty` 在段落首行之后分页的额外惩罚值。**Plain T<sub>E</sub>X** 默认为 150。

`\widowpenalty` 在段落尾行之前分页的额外惩罚值。**Plain T<sub>E</sub>X** 默认为 150。

`\displaywidowpenalty` 在陈列公式上面的尾行之前分页的额外惩罚值。**Plain T<sub>E</sub>X** 默认为 50。

`\brokenpenalty` 在连字行之后分页的额外惩罚值。**Plain T<sub>E</sub>X** 默认为 100。

`\penalty` 在当前列表上添加一个惩罚项。

`\lastpenalty` 如果当前列表的上一项为惩罚项，它表示该惩罚项的值。

`\unpenalty` 如果当前列表的上一项为惩罚项，删除该项目。



## 27.1 当前页面与备选内容

$\text{\TeX}$  的主竖直列被分为两部分：当前页面和备选内容列。任何添加到主竖直列的素材被迫加到备选内容中；将素材从备选内容移动到当前页面的操作称为执行页面构建器。

每次有内容移动到当前页面时， $\text{\TeX}$  计算在该处分页的代价。如果  $\text{\TeX}$  判断该处已经越过最佳分页点，当前页面在最佳分页点之前的所有内容就被放入  $\text{\box255}$ ，而剩下的内容被放回备选内容顶部。如果页面在惩罚项处分页，该惩罚的值就被记录在  $\text{\outputpenalty}$  中，而且值为 10 000 的惩罚项被放在备选内容的顶部；否则， $\text{\outputpenalty}$  就被设为 10 000。

如果当前页面是空的，从备选内容移动过来的可弃项目将被丢弃。此机制会让分页点之后以及第一页顶部的粘连消失。在第一个非可弃项目移动到当前页面时， $\text{\topskip}$  粘连将会被插入；详见上一章。

要看到页面构建器的工作过程，可以将  $\text{\tracingpages}$  设定为某个正值（见第 34 章）。

## 27.2 激活页面构建器

页面构建器将在下列位置起作用：

- 在段落前后：在  $\text{\everypar}$  记号列被插入后，以及段落被加入到竖直列后。见本章结尾处的例子。
- 在陈列公式前后：在  $\text{\everydisplay}$  记号列被插入后，以及陈列公式被加入到竖直列后。
- 在竖直模式的  $\text{\par}$  命令、盒子、插入项和显式惩罚项之后。
- 在输出例程结束后。

页面构建器在这些位置将备选内容移动到当前页面。注意页面构建器运行时  $\text{\TeX}$  无需处于竖直模式中。在水平模式，激活页面构建器是为了将前面的竖直粘连（比如  $\text{\parskip}$  和  $\text{\abovedisplayskip}$ ）移动到当前页面。

$\text{\end}$  命令 – 它只能用于外部竖直模式中 – 在主竖直列为空且  $\text{\deadcycles} = 0$  时将终止  $\text{\TeX}$  任务。否则，下列这串记号

```
 $\text{\hbox{}}\text{\vfill}\text{\penalty-2}^{30}$ 
```

将被插入进来，这促使输出例程开始运行【译注：然后  $\text{\end}$  命令将被重新读入】。

## 27.3 页面长度的记录

如上一章所述，到达输出例程的页面盒子的高度和深度由  $\text{\vsize}$ 、 $\text{\topskip}$  和  $\text{\maxdepth}$  确定。在第一个盒子出现当前页面时， $\text{\TeX}$  放入  $\text{\topskip}$  粘连；而

`\vsize` 和 `\maxdepth` 在第一个盒子或插入项出现在页面时读取。此后对这些参数的修改将不会被注意到，直到下一页或者，更严格地说，直到输出例程调用完毕之后。

在第一个盒子、标线或插入项出现在当前页面之后，`\vsize` 被记录在 `\pagegoal` 中，并且在 `\output` 调用完毕之前不再被查看。改变 `\pagegoal` 对当前页面不会有任何影响。当页面为空时，`\pagegoal` 等于 `\maxdimen`，而 `\pagetotal` 等于零。

积累的尺寸和可伸缩量记录在 `\pagetotal`、`\pagedepth`、`\pagestretch`、`\pagefilstretch`、`\pagefillstretch`、`\pagefilllstretch` 以及 `\pageshrink` 这些参数中。它们由页面构建器设定。在添加每个粘连到页面时，可伸长量和可收缩量参数都会被更新。如果最后一个项目是紧排或者粘连，`\pagedepth` 参数等于零。

这些参数都是 `<special dimen>`；对它们中任何一个的赋值都是 `<intimate assignment>`，而且自动就是全局的。

## 27.4 分页点

### 27.4.1 可能的分页点

分页点可以出现在与断行点相同类型的位置。竖直模式的断点如下：

- 在粘连处，只要它前面是非可弃项目；
- 在紧排处，只要它后面是一个粘连；
- 在惩罚处。

在添加段落行到竖直列时，**T<sub>E</sub>X** 会插入行间粘连和各种行间惩罚，这使得页面上通常有足够的分页点。

### 27.4.2 分页点的惩罚

如果 **T<sub>E</sub>X** 决定在惩罚项处分页，这个惩罚项，在多数情况下，是之前自动插入到段落各行间的。

如果列表（未必得是竖直列）的最后一项是个惩罚项，它的值会在记录在 `\lastpenalty` 参数中。如果这个项目不是一个惩罚项，这个参数的值为零。列表的最后一个惩罚项可以用命令 `\unpenalty` 删除。见第 5.9.6 节的例子。

下面列出了各种竖直模式的惩罚：

`\interlinepenalty` 在段落内部两行间分页的惩罚值。在 **plain T<sub>E</sub>X** 中它默认为零，因此两行间不会添加惩罚项。这样 **T<sub>E</sub>X** 可以在 `\baselineskip` 粘连处寻找有效分页点。

`\clubpenalty` 在段落首行之后分页的额外惩罚值。**Plain TeX** 默认为 150。这个数值将被加到 `\interlinepenalty` 上，依照所得结果插入一个惩罚项到包含段落第一行的 `\hbox` 之后，以替代 `\interlinepenalty`。后面几个惩罚的情形类似。

`\widowpenalty` 在段落尾行之前分页的额外惩罚值。**Plain TeX** 默认为 150。

`\displaywidowpenalty` 在陈列公式上面的尾行之前分页的额外惩罚值。在 **plain TeX** 中默认为 50。

`\brokenpenalty` 在连字行之后分页的额外惩罚值。在 **plain TeX** 中默认为 100。

如果惩罚值等于零，惩罚项将不会被插入。

用户也可以插入惩罚项。比如 **plain** 格式有用于强制、阻止或鼓励分页的宏：

```
\def\break{\penalty-10000 }           % force break
\def\nobreak{\penalty10000 }          % prohibit break
\def\goodbreak{\par\penalty-500 }     % encourage page break
```

另外，`\vadjust{\penalty ... }` 是另一种在竖直列中添加惩罚项的方法。它可以用于阻碍或鼓励在段落某行之后分页。

27.4.3 分页点的计算

每当有项目移动到当前页面，**TeX** 计算在该位置分页时的惩罚值  $p$  和丑度  $b$ 。从惩罚值和丑度再计算出分页的代价  $c$ 。

代价最小的位置被记录下来，而当代价为无穷时，即页面过满或惩罚值为  $p \leq -10\,000$  时，当前页面就在（上回记录的）代价最小位置分开。分出来的部分就被放入 `\box255`，然后输出例程记号列被插入。第 255 号盒子的高度总是被设为 `\vsize`，不管它包含多少素材。

丑度计算基于要将页面放入高度为 `\vsize` 深度不超过 `\maxdepth` 的盒子所需的伸长或收缩量。计算方法和断行时一样（见第 8 章）。丑度满足  $0 \leq b \leq 10\,000$ ，除了在页面过满时  $b = \infty$ 。

有些惩罚是由 **TeX** 隐式插入的，比如 `\interlinepenalty` 就是在段落任何两行间自动插入的。其他惩罚可以由用户或用户宏隐式插入。惩罚值  $p \geq 10\,000$  将阻止分页；而惩罚值  $p \leq -10\,000$ （在外部竖直模式中）将强制分页，并立即激活输出例程。

分页代价依照下面步骤计算：

- 1. 若惩罚值小到导致强制分页并立即激活输出例程，而且页面不是过满的，即

未满的页面 $b = 10\,000$
可行的断点 $b < 10\,000$
过满的页面 $b = \infty$ . . .

$b < \infty$  且  $p \leq -10\,000$

则代价等于惩罚值:  $c = p$ 。

2. 若惩罚值不会导致强制或禁止分页, 而且页面不是过满的, 即

$b < \infty$  且  $|p| < 10\,000$

则代价等于  $c = b + p$ 。

3. 若页面非常糟糕, 即

$b = 10\,000$  且  $|p| < 10\,000$

则代价等于  $c = 10\,000$ 。

4. 若页面过满, 即

$b = \infty$  且  $p < 10\,000$

则代价为无穷大:  $c = \infty$ 。此时  $\text{T}\text{E}\text{X}$  判定最佳分页点应该出现在前面, 并且调用输出例程。超过 10 000 的 `\insertpenalties` 值 (见第 29 章) 同样给出无穷大的代价。

惩罚值  $p \leq -10\,000$  将激活输出例程这个事实在  $\text{L}\text{A}\text{T}\text{E}\text{X}$  输出例程中多次用到: 差值码  $|p| - 10\,000$  用于表示调用输出例程的原因【译注: 见 source2e 第 317 页】; 另外可以见下一章的第二个例子。

## 27.5 分割竖直列

用户可以通过 `\vsplit` 命令实现分页操作。

例子:

```
\setbox1 = \vsplit2 to \dimen3
```

将 2 号盒子顶部 `\dimen3` 大小的部分赋予 1 号盒子。这部分素材实际上是从 2 号盒子移除出来的。可以将它与从当前页面分割出 `\vsize` 大小的一块内容作对比。

分割命令

```
\vsplit<8-bit number>to<dimen>
```

提取出的结果是一个满足下列属性的盒子:

- 高度等于指定的 `<dimen>`;  $\text{T}\text{E}\text{X}$  将遍历原始的盒子寄存器 (它必须容纳一个竖直盒子) 以找到最佳断点。最后可能得到一个未满足盒子。
- 深度不超过 `\splitmaxdepth`; 与它类似的是页面盒子的 `\maxdepth`, 而不是对任何盒子都适用的 `\boxmaxdepth`。
- 第一个和最后一个标记放在 `\splitfirstmark` 和 `\splitbotmark` 寄存器中。

而经过 `\vsplit` 操作后盒子的剩余部分是这样的:

- 顶部的所有可弃项都被删除；
- 大小为 `\splittopskip` 的粘连被插入到顶部；如果所分割的是第 255 号盒子，它的顶部已经有了 `\topskip` 粘连；
- 其深度不得超过 `\splitmaxdepth`。

原始盒子的底部始终是 `\vsplit` 操作的合适断点。如果选择它为断点，剩余的盒子寄存器就是空的。提取出的盒子也可以是空的；它是空的当且仅当原始盒子是空的或者并非竖直盒子。

通常，`\vsplit` 被用于从 `\box255` 分割出一部分。通过设定 `\splitmaxdepth` 等于 `\boxmaxdepth`，得到的结果就和  $\TeX$  的页面构建器的一样。在修剪 `\box255` 顶部之后，标记寄存器 `\firstmark` 和 `\botmark` 就包含 `\box255` 剩余部分的第一个和最后一个标记。见下一章关于标记的更多信息。

## 27.6 分页的例子

### 27.6.1 填满页面

假设某个竖直盒子太大以致在页面剩余部分放不下。要将页面填满并将该盒子放到下一页，使用

```
\vfil\vbox{ ... }
```

是错误的。这样  $\TeX$  只会在 `\vfil` 粘连开始处分页，而分页后 `\vfil` 将会被丢弃：结果将得到一个未满足页面，或者至少是一个有伸长太多的糟糕页面。另一方面，

```
\vfil\penalty0 % or any other value
\vbox{ ... }
```

就是正确的做法： $\TeX$  将在惩罚处分页，而该页面将被填满。

### 27.6.2 确定断点

在接下来的例子中我们使用与分页机制相同的 `\vsplit` 操作。

假设已经给出下面的宏和参数设定：

```
\offinterlineskip \showboxdepth=1
\def\High{\hbox{\vrule height5pt}}
\def\HighAndDeep{\hbox{\vrule height2.5pt depth2.5pt}}
```

首先我们考虑伸长竖直列以到达断点的例子。

```
\splitmaxdepth=4pt
\setbox1=\vbox{\High \vfil \HighAndDeep}
\setbox2=\vsplit1 to 9pt
```

给出

```
> \box2=
\vbox(9.0+2.5)x0.4, glue set 1.5fil
```

```

.\hbox(5.0+0.0)x0.4 []
.\glue 0.0 plus 1.0fil
.\glue(\lineskip) 0.0
.\hbox(2.5+2.5)x0.4 []

```

这两个盒子合起来的高度为 7.5pt，因此粘连需要伸长 1.5pt。

其次，我们减少结果列表所允许的深度值。

```

\splitmaxdepth=2pt
\setbox1=\vbox{\High \vfil \HighAndDeep}
\setbox2=\vsplit1 to 9pt

```

给出

```

> \box2=
\vbox(9.0+2.0)x0.4, glue set 1.0fil
.\hbox(5.0+0.0)x0.4 []
.\glue 0.0 plus 1.0fil
.\glue(\lineskip) 0.0
.\hbox(2.5+2.5)x0.4 []

```

此时基准点被下移 0.5pt，而伸长量也相应减少，但是结果列表的尺寸不会比所指定的大。

作为这个事实的例子，再考虑下面的例子

```

\splitmaxdepth=3pt
\setbox1=\vbox{\High \kern1.5pt \HighAndDeep}
\setbox2=\vsplit1 to 9pt

```

这将会给出一个恰好 9pt 高 2.5pt 深的盒子：

```

> \box2=
\vbox(9.0+2.5)x0.4
.\hbox(5.0+0.0)x0.4 []
.\kern 1.5
.\glue(\lineskip) 0.0
.\hbox(2.5+2.5)x0.4 []

```

若改为 `\splitmaxdepth=2pt`，将不会让高度增加 0.5pt；反而由于断点提前而得到一个未满足盒子：

```

> \box2=
\vbox(9.0+0.0)x0.4
.\hbox(5.0+0.0)x0.4 []

```

有时候分割的时机也是重要的。**TeX** 首先寻找能得到所需高度的断点，然后看依照 `\maxdepth` 或 `\splitmaxdepth` 调节深度是否会超出规定高度。

考虑一个与此时机有关的例子：这段代码

```

\splitmaxdepth=4pt
\setbox1=\vbox{\High \vfil \HighAndDeep}
\setbox2=\vsplit1 to 7pt

```

所得结果并非一个 7pt 高 3pt 深的盒子，而是一个未满足盒子：

```
> \box2=
\ vbox(7.0+0.0)x0.4
.\ hbox(5.0+0.0)x0.4 []
```

### 27.6.3 段落后的页面构建

在段落结束后，页面构建器将素材移动到当前页面，但它并不决定断点是否已经找到。

例子：

```
\output{\interrupt \plainoutput}% show when you're active
\def\nl{\hfil\break}\vsize=22pt % make pages of two lines
a\nl b\nl c\par \showlists      % make a 3-line paragraph
```

将报告

```
### current page:
[...]
total height 34.0
goal height 22.0
prevdepth 0.0, prevgraf 3 lines
```

纵使已经收集到过多的素材，`\output` 还是仅会在下个段落开始时才执行：当 `\output` 被插入到 `\everypar` 后，键入 `d` 将给出

```
! Undefined control sequence.
<output> {\interrupt
               \plainoutput }

<to be read again>
d
```

## 第 28 章 Output Routines

The final stages of page processing are performed by the output routine. The page builder cuts off a certain portion of the main vertical list and hands it to the output routine in `\box255`. This chapter treats the commands and parameters that pertain to the output routine, and it explains how output routines can receive information through marks.

`\output` Token list with instructions for shipping out pages.

`\shipout` Ship a box to the dvi file.

`\mark` Specify a mark text.

`\topmark` The last mark on the previous page.

`\botmark` The last mark on the current page.

`\firstmark` The first mark on the current page.

`\splitbotmark` The last mark on a split-off page.

`\splitfirstmark` The first mark on a split-off page.

`\deadcycles` Counter that keeps track of how many times the output routine has been called without a `\shipout` taking place.

`\maxdeadcycles` The maximum number of times that the output routine is allowed to be called without a `\shipout` occurring.

`\outputpenalty` Value of the penalty at the current page break, or 10 000 if the break was not at a penalty.

### 28.1 The `\output` token list

Common parlance has it that ‘the output routine is called’ when  $\mathrm{T\!E\!X}$  has found a place to break the main vertical list. Actually, `\output` is not a macro but a token list that is inserted into  $\mathrm{T\!E\!X}$ ’s command stream.

Insertion of the `\output` token list happens inside a group that is implicitly



opened. Also,  $\TeX$  enters internal vertical mode. Because of the group, non-local assignments (to the page number, for instance) have to be prefixed with `\global`. The vertical mode implies that during the workings of the output routine spaces are mostly harmless.

The `\output` token list belongs to the class of the  $\langle$ token parameter $\rangle$ s. These behave the same as `\toks $\langle$ nnn $\rangle$`  token lists; see Chapter 14. Assigning an output routine can therefore take the following forms:

```
\output $\langle$ equals $\rangle$  $\langle$ general text $\rangle$  or \output $\langle$ equals $\rangle$  $\langle$ filler $\rangle$  $\langle$ token variable $\rangle$ 
```

## 28.2 Output and `\box255`

$\TeX$ 's page builder breaks the current page at the optimal point, and stores everything above that in `\box255`; then, the `\output` tokens are inserted into the input stream. Any remaining material on the main vertical list is pushed back to the recent contributions. If the page is broken at a penalty, that value is recorded in `\outputpenalty`, and a penalty of size 10 000 is placed on top of the recent contributions; otherwise, `\outputpenalty` is set to 10 000. When the output routine is finished, `\box255` is supposed to be empty. If it is not,  $\TeX$  gives an error message.

Usually, the output routine will take the pagebox, append a headline and/or footline, maybe merge in some insertions such as footnotes, and ship the page to the dvi file:

```
\output={\setbox255=\vbox
        {\someheadline
         \vbox to \vsize{\unvbox255 \unvbox\footins}
         \somefootline}
        \shipout\box255}
```

When box 255 reaches the output routine, its height has been set to `\vsize`. However, the material in it can have considerably smaller height. Thus, the above output routine may lead to underfull boxes. This can be remedied with a `\vfil`.

The output routine is under no obligation to do anything useful with `\box255`; it can empty it, or unbox it to let  $\TeX$  have another go at finding a page break. The number of times that the output routing postpones the `\shipout` is recorded in `\deadcycles`: this parameter is set to 0 by `\shipout`, and increased by 1 just before every `\output`.

When the number of dead cycles reaches `\maxdeadcycles`,  $\TeX$  gives an error message, and performs the default output routine

`\shipout\box255`

instead of the routine it was about to start. The  $\text{\LaTeX}$  format has a much higher value for `\maxdeadcycles` than plain  $\text{\TeX}$ , because the output routine in  $\text{\LaTeX}$  is often called for intermediate handling of floats and marginal notes.

The `\shipout` command can send any  $\langle\text{box}\rangle$  to the `dvi` file; this need not be box 255, or even a box containing the current page. It does not have to be called inside the output routine, either.

If the output routine produces any material, for instance by calling

`\unvbox255`

this is put on top of the recent contributions.

After the output routine finishes, the page builder is activated. In particular, because the current page has been emptied, the `\size` is read again. Changes made to this parameter inside the output routine (using `\global`) will therefore take effect.

## 28.3 Marks

Information can be passed to the output routine through the mechanism of *marks*. The user can specify a token list with

`\mark{\langle\text{mark text}\rangle}`

which is put in a mark item on the current vertical list. The mark text is subject to expansion as in `\edef`.

If the mark is given in horizontal mode it migrates to the surrounding vertical lists like an insertion item (see page 78); however, if this is not the external vertical list, the output routine will not find the mark.

Marks are the main mechanism through which the output routine can obtain information about the contents of the currently broken-off page, in particular its top and bottom.  $\text{\TeX}$  sets three variables:

`\botmark` the last mark occurring on the current page;

`\firstmark` the first mark occurring on the current page;

`\topmark` the last mark of the previous page, that is, the value of `\botmark` on the previous page.

If no marks have occurred yet, all three are empty; if no marks occurred on the current page, all three mark variables are equal to the `\botmark` of the previous page.

For boxes generated by a `\vsplit` command (see previous chapter), the `\splitbotmark` and `\splitfirstmark` contain the marks of the split-off part;

`\firstmark` and `\botmark` reflect the state of what remains in the register.

例子: *Marks can be used to get a section heading into the headline or footline of the page.*

```
\def\section#1{ ... \mark{#1} ... }
\def\righthheadline{\hbox to \hsize
  {\headlinefont \botmark\hfil\pagenumber}}
\def\leftheadline{\hbox to \hsize
  {\headlinefont \pagenumber\hfil\firstmark}}
```

*This places the title of the first section that starts on a left page in the left headline, and the title of the last section that starts on the right page in the right headline. Placing the headlines on the page is the job of the output routine; see below.*

*It is important that no page breaks can occur in between the mark and the box that places the title:*

```
\def\section#1{ ...
  \penalty\beforesectionpenalty
  \mark{#1}
  \hbox{ ... #1 ...}
  \nobreak
  \vskip\aftersectionskip
  \noindent}
```

Let us consider another example with headlines: often a page looks better if the headline is omitted on pages where a chapter starts. This can be implemented as follows:

```
\def\endofchapter
\chapter#1{ ... \def\chtitle{#1}\mark{1}\mark{0} ... }
\def\theheadline{\expandafter\ifx\firstmark1
  \else \chapheadline \fi}
```

Only on the page where a chapter starts will the mark be 1, and on all other pages a headline is placed.

## 28.4 Assorted remarks

### 28.4.1 Hazards in non-trivial output routines

If the final call to the output routine does not perform a `\shipout`,  $\text{\TeX}$  will call the output routine endlessly, since a run will only stop if both the vertical list is empty, and `\deadcycles` is zero. The output routine can set `\deadcycles` to zero to prevent this.

### 28.4.2 Page numbering

The page number is not an intrinsic property of the output routine; in plain  $\TeX$  it is the value of `\count0`. The output routine is responsible for increasing the page number when a shipout of a page occurs.

Apart from `\count0`, counter registers 1–9 are also used for page identification: at shipout  $\TeX$  writes the values of these ten counters to the dvi file (see Chapter 33). Terminal and log file output display only the non-zero counters, and the zero counters for which a non-zero counter with a higher number exists, that is, if `\count0 = 1` and `\count3 = 5` are the only non-zero counters, the displayed list of counters is `[1.0.0.5]`.

### 28.4.3 Headlines and footlines in plain $\TeX$

Plain  $\TeX$  has token lists `\headline` and `\footline`; these are used in the macros

`\makeheadline` and `\makefootline`. The page is shipped out as (more or less)

```
\vbox{\makeheadline\pagebody\makefootline}
```

Both headline and footline are inserted inside a `\line`. For non-standard headers and footers it is easier to redefine the macros `\makeheadline` and `\makefootline` than to tinker with the token lists.

### 28.4.4 Example: no widow lines

Suppose that one does not want to allow widow lines, but pages have in general no stretch or shrink, for instance because they only contain plain text. A solution would be to increase the page length by one line if a page turns out to be broken at a widow line.

$\TeX$ 's output routine can perform this sort of trick: if the `\widowpenalty` is set to some recognizable value, the output routine can see by the `\outputpenalty` if a widow line occurred. In that case, the output routine can temporarily increase the `\vsize`, and let the page builder have another go at finding a break point.

Here is the skeleton of such an output routine. No headers or footers are provided for.

```
\newif\ifLargePage \widowpenalty=147
\newdimen\oldvsize \oldvsize=\vsize
\output={
  \ifLargePage \shipout\box255
    \global\LargePagefalse
    \global\vsize=\oldvsize
```

```

\else \ifnum \outputpenalty=\widowpenalty
\global\LargePage>true
\global\advance\ysize\baselineskip
\unvbox255 \penalty\outputpenalty
\else \shipout\box255
\fi \fi}

```

The test `\ifLargePage` is set to true by the output routine if the `\outputpenalty` equals the `\widowpenalty`. The page box is then `\unvboxed`, so that the page builder will tackle the same material once more.

### 28.4.5 Example: no indentation top of page

Some output routines can be classified as abuse of the output routine mechanism. The output routine in this section is a good example of this.

It is imaginable that one wishes paragraphs not to indent if they start at the top of a page. (There are plenty of objections to this layout, but occasionally it is used.) This problem can be solved using the output routine to investigate whether the page is still empty and, if so, to give a signal that a paragraph should not indent.

Note that we cannot use the fact here that the page builder comes into play after the insertion of `\everypar`: even if we could force the output routine to be activated here, there is no way for it to remove the indentation box.

The solution given here lets the `\everypar` terminate the paragraph immediately with

```
\par\penalty-\specialpenalty
```

which activates the output routine. Seeing whether the pagebox is empty (after removing the empty line and any `\parskip` glue), the output routine then can set a switch signalling whether the retry of the paragraph should indent.

There are some minor matters in the following routines, the sense of which is left for the reader to ponder.

```

\mathchardef\specialpenalty=10001
\newif\ifPreventSwitch
\newbox\testbox
\topskip=10pt

\everypar{\begingroup \par
\penalty-\specialpenalty
\everypar{\endgroup}\parskip0pt
\ifPreventSwitch \noindent \else \indent \fi
\global\PreventSwitchfalse
}
\output{

```

```
\ifnum\outputpenalty=-\specialpenalty
  \setbox\testbox\ vbox{\unvbox255
    {\setbox0=\lastbox}\unskip}
  \ifdim\ht\testbox=0pt \global\PreventSwitchtrue
  \else \topskip=0pt \unvbox\testbox \fi
\else \shipout\box255 \global\advance\pageno1 \fi}
```

#### 28.4.6 More examples of output routines

A large number of examples of output routines can be found in [38] and [39].

## 第 29 章 Insertions

Insertions are T<sub>E</sub>X's way of handling floating information. T<sub>E</sub>X's page builder calculates what insertions and how many of them will fit on the page; these insertion items are then placed in insertion boxes which are to be handled by the output routine.

`\insert` Start an insertion item.

`\newinsert` Allocate a new insertion class.

`\insertpenalties` Total of penalties for split insertions. Inside the output routine, the number of held-over insertions.

`\floatingpenalty` Penalty added when an insertion is split.

`\holdinginserts` (T<sub>E</sub>X3 only) If this is positive, insertions are not placed in their boxes at output time.

`\footins` Number of the footnote insertion class in plain T<sub>E</sub>X.

`\topins` Number of the top insertion class.

`\topinsert` Plain T<sub>E</sub>X macro to start a top insert.

`\pageinsert` Plain T<sub>E</sub>X macro to start an insert that will take up a whole page.

`\midinsert` Plain T<sub>E</sub>X macro that places its argument if there is space, and converts it into a top insert otherwise.

`\endinsert` Plain T<sub>E</sub>X macro to wind up an insertion item that started with `\topinsert`, `\midinsert`, or `\pageinsert`.

### 29.1 Insertion items

Floating information, items that are not bound to a fixed place in the vertical list, such as figures or footnotes, are handled by *insertions*. The treatment of insertions is a strange interplay between the user, T<sub>E</sub>X's internal workings,

and the output routine. First the user specifies an insertion, which is a certain amount of vertical material; then  $\TeX$ 's page builder decides what insertions should go on the current page and puts these insertions in insertion boxes; finally, the output routine has to do something with these boxes.

An insertion item looks like

```
\insert<8-bit number>{<vertical mode material>}
```

where the 8-bit number should not be 255, because `\box255` is used by  $\TeX$  for passing the page to the output routine.

The braces around the vertical mode material in an insertion item can be implicit; they imply a new level of grouping. The vertical mode material is processed in internal vertical mode.

Values of `\splittopskip`, `\splitmaxdepth`, and `\floatingpenalty` are relevant for split insertions (see below); the values that are current just before the end of the group are used.

Insertion items can appear in vertical mode, horizontal mode, and math mode. For the latter two modes they have to migrate to the surrounding vertical list (see page 78). After an insertion item is put on the vertical list the page builder is exercised.

## 29.2 Insertion class declaration

In the plain format the number for a new insertion class is allocated by `\newinsert`:

```
\newinsert\myinsert % new insertion class
```

which uses `\chardef` to assign a number to the control sequence.

Insertion classes are allocated numbering from 254 downward. As `\box 255` is used for output, this allocation scheme leaves `\skip255`, `\dimen255`, and `\count255` free for scratch use.

## 29.3 Insertion parameters

For each insertion class  $n$  four registers are allocated:

- `\box n` When the output routine is active this box contains the insertion items of class  $n$  that should be placed on the current page.
- `\dimen n` This is the maximum space allotted for insertions of class  $n$  per page. If this amount would be exceeded  $\TeX$  will split insertions.



- `\skip n` Glue of this size is added the first time an insertion item of class  $n$  is added to the current page. This is useful for such phenomena as a rule separating the footnotes from the text of the page.
- `\count n` Each insertion item is a vertical list, so it has a certain height. However, the effective height, the amount of influence it has on the text height of the page, may differ from this real height. The value of `\count n` is then 1000 times the factor by which the height should be multiplied to obtain the effective height.

Consider the following examples:

- Marginal notes do not affect the text height, so the factor should be 0.
- Footnotes set in double column mode affect the page by half of their height: the count value should be 500.
- Conversely, footnotes set at page width underneath a page in double column mode affect both columns, so – provided that the double column mode is implemented by applying `\vsplit` to a double-height column – the count value should be 2000.

## 29.4 Moving insertion items from the contributions list

The most complicated issue with insertions is the algorithm that adds insertion items to the main vertical list, and calculates breakpoints if necessary.

$\TeX$  never changes the `\vsize`, but it diminishes the `\pagegoal` by the (effective) heights of the insertion items that will appear before a page break. Thus the output routine will receive a `\box255` that has height `\pagegoal`, not necessarily `\vsize`.

1. When the first insertion of a certain class  $n$  occurs on the current page  $\TeX$  has to account for the quantity `\skip n`. This step is executed only if no earlier insertion item of this class occurs on the vertical list – this includes insertions that were split – but `\box n` need not be empty at this time. If `\box n` is not empty, its height plus depth is multiplied by `\count n/1000` and the result is subtracted from `\pagegoal`. Then the `\pagegoal` is diminished by the natural component of `\skip n`. Any stretch and shrink of `\skip n` are incorporated in `\pagestretch` and `\pageshrink` respectively.
2. If there is a split insertion of class  $n$  on the page – this case and the previous step in the algorithm are mutually exclusive – the `\floatingpenalty` is added to `\insertpenalties`. A split insertion is an insertion item for which

a breakpoint has been calculated as it will not fit on the current page in its entirety. Thus the insertion currently under consideration will certainly not wind up on the current page.

3. After the preliminary action of the two previous points  $\text{\TeX}$  will place the actual insertion item on the main vertical list, at the end of the current contributions. First it will check whether the item will fit without being split.

There are two conditions to be checked:

- adding the insertion item (plus all previous insertions of that class) to  $\text{\box } n$  should not let the height plus depth of that box exceed  $\text{\dimen } n$ , and
- either the effective height of the insertion is negative, or  $\text{\pagetotal}$  plus  $\text{\pagedepth}$  minus  $\text{\pageshrink}$  plus the effective size of the insertion should be less than  $\text{\pagegoal}$ .

If these conditions are satisfied,  $\text{\pagegoal}$  is diminished by the effective size of the insertion item, that is, by the height plus depth, multiplied by  $\text{\count } n/1000$ .

4. Insertions that fail on one of the two conditions in the previous step of the algorithm will be considered for splitting.  $\text{\TeX}$  will calculate the size of the maximal portion to be split off the insertion item, such that
  - (a) adding this portion together with earlier insertions of this class to  $\text{\box } n$  will not let the size of the box exceed  $\text{\dimen } n$ , and
  - (b) the effective size of this portion, added to  $\text{\pagetotal}$  plus  $\text{\pagedepth}$ , will not exceed  $\text{\pagegoal}$ . Note that  $\text{\pageshrink}$  is not taken into account this time, as it was in the previous step.

Once this maximal size to be split off has been determined,  $\text{\TeX}$  locates the least-cost breakpoint in the current insertion item that will result in a box with a height that is equal to this maximal size. The penalty associated with this breakpoint is added to  $\text{\insertpenalties}$ , and  $\text{\pagegoal}$  is diminished by the effective height plus depth of the box to be split off the insertion item.

## 29.5 Insertions in the output routine

When the output routine comes into action – more precisely: when  $\text{\TeX}$  starts processing the tokens in the  $\text{\output}$  token list – all insertions that should

be placed on the current page have been put in their boxes, and it is the responsibility of the output routine to put them somewhere in the box that is going to be shipped out.

例子: *The plain T<sub>E</sub>X output routine handles top inserts and footnotes by packaging the following sequence:*

```
\ifvoid\topins \else \unvbox\topins \fi
\pagebody
\ifvoid\footins \else \unvbox\footins \fi
```

*Unboxing the insertion boxes makes the glue on various parts of the page stretch or shrink in a uniform manner.*

With T<sub>E</sub>X3 the insertion mechanism has been extended slightly: the parameter `\holdinginserts` can be used to specify that insertions should not yet be placed in their boxes. This is very useful if the output routine wants to recalculate the `\vsize`, or if the output routine is called to do other intermediate calculations instead of ejecting a page.

During the output routine the parameter `\insertpenalties` holds the number of insertion items that are being held over for the next page. In the plain T<sub>E</sub>X output routine this is used after the last page:

```
\def\dosupereject{\ifnum\insertpenalties>0
% something is being held over
\line{}\kern-\topskip\nobreak\vfill\supereject\fi}
```

## 29.6 Plain T<sub>E</sub>X insertions

The plain T<sub>E</sub>X format has only two insertion classes: the footnotes and the top inserts. The macro `\pageinsert` generates top inserts that are stretched to be exactly `\vsize` high. The `\midinsert` macro tests whether the vertical material specified by the user fits on the page; if so, it is placed there; if not, it is converted to a top insert.

Footnotes are allowed to be split, but once one has been split no further footnotes should appear on the current page. This effect is attained by setting

```
\floatingpenalty=20000
```

The `\floatingpenalty` is added to `\insertpenalties` if an insertion follows a split insertion of the same class. However, `\floatingpenalty > 10 000` has infinite cost, so T<sub>E</sub>X will take an earlier breakpoint for splitting off the page from the vertical list.

Top inserts essentially contain only a vertical box which holds whatever the user specified. Thus such an insert cannot be split. However, the `\endinsert`

macro puts a `\penalty100` on top of the box, so the insertion can be split with an empty part before the split. The effect is that the whole insertion is carried over to the next page. As the `\floatingpenalty` for top inserts is zero, arbitrarily many of these inserts can be moved forward until there is a page with sufficient space.

Further examples of insertion macros can be found in [\[40\]](#).

## 第 30 章 File Input and Output

This chapter discusses the various ways in which T<sub>E</sub>X can read from and write to external *files*.

`\input` Read a specified file as T<sub>E</sub>X input.

`\endinput` Terminate inputting the current file after the current line.

`\pausing` Specify that T<sub>E</sub>X should pause after each line that is read from a file.

`\inputlineno` Number of the current input line.

`\message` Write a message to the terminal.

`\write` Write a `<general text>` to the terminal or to a file.

`\read` Read a line from a stream into a control sequence.

`\newread` `\newwrite` Macro for allocating a new input/output stream.

`\openin` `\closein` Open/close an input stream.

`\openout` `\closeout` Open/close an output stream.

`\ifeof` Test whether a file has been fully read, or does not exist.

`\immediate` Prefix to have output operations executed right away.

`\escapechar` Number of the character that is used when control sequences are being converted into character tokens. In T<sub>E</sub>X default: 92.

`\newlinechar` Number of the character that triggers a new line in `\write` and `\message` statements.

### 30.1 Including files: `\input` and `\endinput`

Large documents can be segmented in T<sub>E</sub>X by putting parts in separate *input files*, and loading these with `\input` into the master file. The exact syntax for file names is implementation dependent; most of the time a `.tex` file extension is assumed if no explicit extension is given. File names can be delimited

with a space or with `\relax`. The `\input` command is expandable.

If  $\text{\TeX}$  encounters in an input file the `\endinput` statement, it acts as if the file ends after the line on which the statement occurs. Any statements on the same line as `\endinput` are still executed. The `\endinput` statement is expandable.

## 30.2 File I/O

$\text{\TeX}$  supports input and output streams for reading and writing files one line at a time.

### 30.2.1 Opening and closing streams

$\text{\TeX}$  supports up to 16 simultaneous input and 16 output *streams*. The plain  $\text{\TeX}$  macros `\newread` and `\newwrite` give the number of an unused stream. This number is assigned by a `\chardef` command. Input streams are completely independent of output streams.

Input streams are opened by

```
\openin<4-bit number>\equals<filename>
```

and closed by

```
\closein<4-bit number>
```

Output streams are opened by

```
\openout<4-bit number>\equals<filename>
```

and closed by

```
\closeout<4-bit number>
```

If an output file does not yet exist, it is created by `\openout`; if it did exist, an `\openout` will cause it to be overwritten.

The output operations `\openout`, `\closeout`, and `\write` can all three be prefixed by `\immediate`; see below.

### 30.2.2 Input with `\read`

In addition to the `\input` command, which reads a whole file,  $\text{\TeX}$  has the `\read` operation, which reads one line from a file (or from the user terminal). The syntax of the read command is

```
\read<number>to<control sequence>
```

The effect of this statement is that one input line is read from the designated stream, and the control sequence is defined as a macro without parameters, having that line as replacement text.

If the input line is not balanced with respect to braces,  $\text{\TeX}$  will read more than one line, continuing for as long as is necessary to get a balanced token list.  $\text{\TeX}$  implicitly appends an empty line to each input stream, so the last `\read` operation on a stream will always yield a single `\par` token.

Read operations from any stream outside the range 0–15 – or streams not associated with an open file, or on which the file end has been reached – read from the terminal. If the stream number is positive the user is prompted with the name of the control sequence being defined by the `\read` statement.

例子:

```
\read16 to \data
```

*displays a prompt*

```
\data=
```

*and typing ‘my name’ in response makes the read statement equivalent to*

```
\def\data{my name }
```

*The space at the end of the input derives from the line end; to prevent this one could write*

```
{\endlinechar=-1 \global\read16 to \data}
```

### 30.2.3 Output with `\write`

$\text{\TeX}$ ’s `\write` command

```
\write⟨number⟩⟨general text⟩
```

writes a balanced token list to a file which has been opened by `\openout`, to the log file, or to the terminal.

Write operations to a stream outside 0–15 – or to a stream that is not associated with an open file – go to the log file; if the stream number is positive they go to the terminal as well as to the log file.

The token list argument of `\write`, defined as

```
⟨general text⟩ → ⟨filler⟩{⟨balanced text⟩⟨right brace⟩}
```

can have an implicit opening brace. This argument is expanded as if it were the replacement text of an `\edef`, so, for instance, any macros and conditionals appearing are expanded. No commands are executed, however. This expansion

occurs at the time of shipping out; see below. Until that time the argument token list is stored in a `whatsit` item on the current list. See further Chapter 12 for a discussion of expansion during writing.

A control sequence output by `\write` (or `\message`) is represented with a trailing space, and using character number `\escapechar` for the escape character. The `InitTeX` default for this is 92, the code for the backslash. The trailing space can be prevented by prefixing the control sequence with `\string`.

### 30.3 Whatsits

There is an essential difference in execution between input and output: operations concerning output (`\openout`, `\closeout`, `\write`) are executed *asynchronously*. That is, instead of being done immediately they are saved until the box in which they appear is shipped out to the `dvi` file.

Writes and the other two output operations are placed in ‘`whatsit`’ items on whichever list is currently being built. The actual operation occurs when the part of the page that has the item is shipped out to the `dvi` file. This delayed output is made necessary by `TeX`’s asynchronous output routine behaviour. See a worked-out example on page 141.

An `\immediate\write` – or any other `\immediate` output operation – is executed on the spot, and does not place a `whatsit` item on the current list.

The argument of a `\special` command (see page 286) is also placed in a `whatsit`.

Whatsit items in leader boxes are ignored.

### 30.4 Assorted remarks

#### 30.4.1 Inspecting input

`TeX` records the current line number in the current input file in the `(internal integer)` parameter `\inputlineno` (in `TeX3`).

If the parameter `\pausing` is positive, `TeX` shows every line that is input on the terminal *screen*, and gives the user the opportunity to insert commands. These can for instance be `\show` commands. Inserted commands are treated as if they were directly in the source file: it is for instance not necessary to prefix them with ‘`i`’, as would be necessary when `TeX` pauses for an error.



### 30.4.2 Testing for existence of files

$\TeX$  is not the friendliest of systems when you ask it to input a non-existing file. Therefore the following sequence of commands can be used to prevent trouble:

```
\newread\instream \openin\instream= fname.tex
\ifeof\instream \message{File 'fname' does not exist!}
\else \closein\instream \input fname.tex
\fi
```

Here an input stream is opened with the given file name. The end-of-file test is also true if an input stream does not correspond to a physical file, so if this conditional is not true, the file exists and an `\input` command can safely be given.

### 30.4.3 Timing problems

The synchronization between write operations on the one hand, and opening/closing operations of files on the other hand, can be a crucial point. Auxiliary files, such as are used by various formats to implement cross-references, are a good illustration of this.

Suppose that during a run of  $\TeX$  the auxiliary file is written, and at the end of the run it has to be input again for a variety of purposes (such as seeing whether references have changed). An `\input` command is executed right away, so the file must have been closed with an `\immediate\closeout`. However, now it becomes possible that the file is closed before all writes to it have been performed. The following sequence remedies this:

```
\par\vfil\penalty -10000 \immediate\closeout\auxfile
```

The first three commands activate the output routine in order to close off the last page, so all writes will indeed have been performed before the file is closed.

### 30.4.4 `\message` versus `\immediate\write`

Messages to the user can be given using `\message<general text>`, which writes to the terminal. Messages are appended to one another; the line is wrapped when the line length (a  $\TeX$  compile-time constant) has been reached. In  $\TeX$  version 2, a maximum of 1000 characters is written per message; this is not a compile-time constant, but is hard-wired into the  $\TeX$  program.

Each message given with `\immediate\write` starts on a new line; the user can force a new line in the message by including the character with number `\newlinechar`.

This parameter also works in `\message`. The plain  $\text{\TeX}$  default for `\newlinechar` is `-1`; the  $\text{\LaTeX}$  default of `10` allows you to write `\message{two^^Jlines}`.

### 30.4.5 Write inside a vertical box

Since a write operation winds up on the vertical list in a whatsit, issuing one at the start of a `\vtop` will probably influence the height of that box (see Chapter 5). As an example,

```
have the \vtop{\write\terminal{Hello!}\hbox{more text}}
dangling from
```

will have the `more text` dangling from the baseline (and when this book is  $\text{\TeX}$ ed the message ‘Hello!’ appears on the screen).

### 30.4.6 Expansion and spaces in `\write` and `\message`

Both `\write` and `\message` expand their argument as if it were the replacement text of an `\edef`. Therefore

```
\def\a{b}\message{\a}
```

will write out ‘b’.

Unexpandable control sequences are displayed with a trailing space (and prefixed with the `\escapechar`):

```
\message{\hbox\vbox!}
```

will write out ‘`\hbox \vbox !`’. Undefined control sequences give an error here.

Expandable control sequences can be written out with some care:

```
\message{\noexpand\ifx}
\message{\string\ifx}
{\let\ifx\relax \message{\ifx}}
```

all write out ‘`\ifx`’.

Note, however, that spaces after expandable control sequences are removed in the input processor, which goes into state *S* after a control sequence. Therefore

```
\def\a{b}\def\c{d}
\message{\a \c}
```

writes out ‘bd’. Inserting a space can be done as follows:

```
\def\space{ } % in plain TeX
\message{\a\space\c}
```

displays ‘b d’. Note that

```
\message{\a{ } \c}
```

does not work: it displays ‘b{ }d’ since braces are unexpandable character tokens.

## 第 31 章 Allocation

$\text{\TeX}$  has registers of a number of types. For some of these, explicit commands exist to define a synonym for a certain register; for all of them macros exist in the plain format to allocate an unused register. This chapter treats the synonym and allocation commands, and discusses some guidelines for macro writers regarding allocation.

`\countdef` Define a synonym for a `\count` register.  
`\dimendef` Define a synonym for a `\dimen` register.  
`\muskipdef` Define a synonym for a `\muskip` register.  
`\skipdef` Define a synonym for a `\skip` register.  
`\toksdef` Define a synonym for a `\toks` register.  
`\newbox` Allocate an unused `\box` register.  
`\newcount` Allocate an unused `\count` register.  
`\newdimen` Allocate an unused `\dimen` register.  
`\newfam` Allocate an unused math family.  
`\newinsert` Allocate an unused insertion class.  
`\newlanguage` ( $\text{\TeX}$ 3 only) Allocate a new language number.  
`\newmuskip` Allocate an unused `\muskip` register.  
`\newskip` Allocate an unused `\skip` register.  
`\newtoks` Allocate an unused `\toks` register.  
`\newread` Allocate an unused input stream.  
`\newwrite` Allocate an unused output stream.

## 31.1 Allocation commands

In plain  $\text{\TeX}$ ,  $\text{\new} \dots$  macros are defined for allocation of registers. The registers of  $\text{\TeX}$  fall into two classes that are allocated in different ways. This is treated below.

The  $\text{\newlanguage}$  macro of plain  $\text{\TeX}$  does not allocate any register. Instead it merely assigns a number, starting from 0.  $\text{\TeX}$  (version 3) can have at most 256 different sets of hyphenation patterns.

The  $\text{\new} \dots$  macros of plain  $\text{\TeX}$  are defined to be  $\text{\outer}$  (see Chapter 11 for a precise explanation), which precludes use of the allocation macros in other macros. Therefore the  $\text{\LaTeX}$  format redefines these macros without the  $\text{\outer}$  prefix.

### 31.1.1 $\text{\count}$ , $\text{\dimen}$ , $\text{\skip}$ , $\text{\muskip}$ , $\text{\toks}$

For these registers there exists a  $\langle\text{registerdef}\rangle$  command, for instance  $\text{\countdef}$ , to couple a specific register to a control sequence:

$\langle\text{registerdef}\rangle\langle\text{control sequence}\rangle\langle\text{equals}\rangle\langle\text{8-bit number}\rangle$

After the definition

$\text{\countdef}\text{\MyCount}=42$

the allocated register can be used as

$\text{\MyCount}=314$

or

$\text{\vskip}\text{\MyCount}\text{\baselineskip}$

The  $\langle\text{registerdef}\rangle$  commands are used in plain  $\text{\TeX}$  macros  $\text{\newcount}$  et cetera that allocate an unused register; after

$\text{\newcount}\text{\MyCount}$

$\text{\MyCount}$  can be used exactly as in the above two examples.

### 31.1.2 $\text{\box}$ , $\text{\fam}$ , $\text{\write}$ , $\text{\read}$ , $\text{\insert}$

For these registers there exists no  $\langle\text{registerdef}\rangle$  command in  $\text{\TeX}$ , so  $\text{\chardef}$  is used to allocate box registers in the corresponding plain  $\text{\TeX}$  macros  $\text{\newbox}$ , for instance.

The fact that  $\text{\chardef}$  is used implies that the defined control sequence does not stand for the register itself, but only for its number. Thus after

$\text{\newbox}\text{\MyBox}$

it is necessary to write

$\text{\box}\text{\MyBox}$

Leaving out the `\box` means that the character in the current font with number `\MyBox` is typeset. The `\chardef` command is treated further in Chapter 3.

## 31.2 Ground rules for macro writers

The `\new...` macros of plain  $\TeX$  have been designed to form a foundation for macro packages, such that several of such packages can operate without collisions in the same run of  $\TeX$ . In appendix B of the  $\TeX$  book Knuth formulates some ground rules that macro writers should adhere to.

1. The `\new...` macros do not allocate registers with numbers 0–9. These can therefore be used as ‘scratch’ registers. However, as any macro family can use them, no assumption can be made about the permanency of their contents. Results that are to be passed from one call to another should reside in specifically allocated registers.

Note that count registers 0–9 are used for page identification in the `dvi` file (see Chapter 33), so no global assignments to these should be made.

2. `\count255`, `\dimen255`, and `\skip255` are also available. This is because inserts are allocated from 254 downward and, together with an insertion box, a count, dimen, and skip register, all with the same number, are allocated. Since `\box255` is used by the output routine (see Chapter 28), the count, dimen, and skip with number 255 are freely available.
3. Assignments to scratch registers 0, 2, 4, 6, 8, and 255 should be local; assignments to registers 1, 3, 5, 7, 9 should be `\global` (with the exception of the `\count` registers). This guideline prevents ‘save stack build-up’ (see Chapter 35).
4. Any register can be used inside a group, as  $\TeX$ ’s grouping mechanism will restore its value outside the group. There are two conditions on this use of a register: no global assignments should be made to it, and it must not be possible that other macros may be activated in that group that perform global assignments to that register.
5. Registers that are used over longer periods of time, or that have to survive in between calls of different macros, should be allocated by `\new...`

## 第 32 章 Running T<sub>E</sub>X

This chapter treats the run modes of T<sub>E</sub>X, and some other commands associated with the job being processed.

`\everyjob` Token list that is inserted at the start of each new job.

`\jobname` Name of the main T<sub>E</sub>X file being processed.

`\end` Command to finish off a run of T<sub>E</sub>X.

`\bye` Plain T<sub>E</sub>X macro to force the final output.

`\pausing` Specify that T<sub>E</sub>X should pause after each line that is read from a file.

`\errorstopmode` T<sub>E</sub>X will ask for user input on the occurrence of an error.

`\scrollmode` T<sub>E</sub>X fixes errors itself, but will ask the user for missing files.

`\nonstopmode` T<sub>E</sub>X fixes errors itself, and performs an emergency stop on serious errors such as missing input files.

`\batchmode` T<sub>E</sub>X fixes errors itself and performs an emergency stop on serious errors such as missing input files, but no terminal output is generated.

### 32.1 Jobs

T<sub>E</sub>X associates with each run a name for the file being processed: the `\jobname`. If T<sub>E</sub>X is run interactively – meaning that it has been invoked without a file argument, and the user types commands – the `\jobname` is `texput`.

The `\jobname` can be used to generate the names of auxiliary files to be read or written during the run. For instance, for a file `story.tex` the `\jobname` is `story`, and writing

```
\openout\Auxiliary=\jobname.aux
\openout\TableOfContents=\jobname.toc
```

will create the files `story.aux` and `story.toc`.

### 32.1.1 Start of the job

T<sub>E</sub>X starts each job by inserting the `\everyjob` token list into the command stream. Setting this variable during a run of T<sub>E</sub>X has no use, but a format can use it to identify itself to the user. If a format fills the token list, the commands therein are automatically executed when T<sub>E</sub>X is run using that format.

### 32.1.2 End of the job

A T<sub>E</sub>X job is terminated by the `\end` command. This may involve first forcing the output routine to process any remaining material (see Chapter 27). If the end of job occurs inside a group T<sub>E</sub>X will give a diagnostic message. The `\end` command is not allowed in internal vertical mode, because this would be inside a vertical box.

Usually some sugar coating of the `\end` command is necessary. For instance the plain T<sub>E</sub>X macro `\bye` is defined as

```
\def\bye{\par\vfill\supereject\end}
```

where the `\supereject` takes care of any leftover insertions.

### 32.1.3 The log file

For each run T<sub>E</sub>X creates a *log file*. Usually this will be a file with as name the value of `\jobname`, and the extension `.log`. Other extensions such as `.lis` are used by some implementations. This log file contains all information that is displayed on the screen during the run of T<sub>E</sub>X, but it will display some information more elaborately, and it can contain statistics that are usually not displayed on the screen. If the parameter `\tracingonline` has a positive value, all the log file information will be shown on the screen.

Overfull and underfull boxes are reported on the terminal screen, and they are dumped using the parameters `\showboxdepth` and `\showboxbreadth` in the log file (see Chapter 34). These parameters are also used for box dumps caused by the `\showbox` command, and for the dump of boxes written by `\shipout` if `\tracingoutput` is set to a positive value.

Statistics generated by commands such as `\tracingparagraphs` will be written to the log file; if `\tracingonline` is positive they will also be shown on the screen.

Output operations to a stream that is not open, or to a stream with a number that is not in the range 0–15, go to the log file. If the stream number is positive, they also go to the terminal.



## 32.2 Run modes

By default, `TEX` goes into `\errorstopmode` if an error occurs: it stops and asks for input from the user. Some implementations have a way of forcing `TEX` into `errorstopmode` when the user interrupts `TEX`, so that the internal state of `TEX` can be inspected (and altered). See page 297 for ways to switch the run mode when `TEX` has been interrupted.

Often, `TEX` can fix an error itself if the user asks `TEX` just to continue (usually by hitting the return key), but sometimes (for instance in alignments) it may take a while before `TEX` is on the right track again (and sometimes it never is). In such cases the user may want to turn on `\scrollmode`, which instructs `TEX` to fix as best it can any occurring error without confirmation from the user. This is usually done by typing ‘s’ when `TEX` asks for input.

In `\scrollmode`, `TEX` also does not ask for input after `\show...` commands. However, some errors, such as a file that could not be found for `\input`, are not so easily remedied, so the user will still be asked for input.

With `\nonstopmode` `TEX` will scroll through errors and, in the case of the kind of error that cannot be recovered from, it will make an emergency stop, aborting the run. Also `TEX` will abort the run if a `\read` is attempted from the terminal. The `\batchmode` differs only from `nonstopmode` in that it gives messages only to the log file, not to the terminal.

## 第 33 章 $\text{\TeX}$ and the Outside World

This chapter treats those commands that bear relevance to `dvi` files and formats. It gives some global information about `Ini\TeX`, font and format files, Computer Modern typefaces, and `web`.

`\dump` Dump a format file; possible only in `Ini\TeX`, not allowed inside a group.

`\special` Write a `\langle`balanced text`\rangle` to the `dvi` file.

`\mag` 1000 times the magnification of the document.

`\year` The year of the current job.

`\month` The month of the current job.

`\day` The day of the current job.

`\time` Number of minutes after midnight that the current job started.

`\fmtname` Macro containing the name of the format dumped.

`\fmtversion` Macro containing the version of the format dumped.

### 33.1 $\text{\TeX}$ , `Ini\TeX`, `Vir\TeX`

In the terminology established in  *$\text{\TeX}$ : the Program*, [23],  $\text{\TeX}$  programs come in three flavours. `Ini\TeX` is a version of  $\text{\TeX}$  that can generate formats; `Vir\TeX` is a production version without preloaded format, and  $\text{\TeX}$  is a production version with preloaded (plain) format. Unfortunately, this terminology is not adhered to in general. A lot of systems do not use preloaded formats (the procedure for making them may be impossible on some operating systems), and call the ‘virgin  $\text{\TeX}$ ’ simply  $\text{\TeX}$ . This manual also follows that convention.

#### 33.1.1 Formats: loading

A *format file* (usually with extension `.fmt`) is a compact dump of  $\text{\TeX}$ ’s internal structures. Loading a format file takes a considerably shorter time

than would be needed for loading the font information and the macros that constitute the format.

Both T<sub>E</sub>X and IniT<sub>E</sub>X can load a format; the user specifies this by putting the name on the command line

```
% tex &plain
```

or at the \*\* prompt

```
% tex
This is TeX. Version ....
** &plain
```

preceded by an ampersand (for UNIX, this should be \& on the command line). An input file name can follow the format name in both places.

IniT<sub>E</sub>X does not need a format, but if no format is specified for (Vir)T<sub>E</sub>X, it will try to load the plain format, and halt if that cannot be found.

### 33.1.2 Formats: dumping

IniT<sub>E</sub>X is the only version of T<sub>E</sub>X that can dump a format, since it is the only version of T<sub>E</sub>X that has the command \dump, which causes the internal structures to be dumped as a format. It is also the only version of T<sub>E</sub>X that has the command \patterns, which is needed to specify a list of hyphenation patterns.

Dumping is not allowed inside a group, that is

```
{ ... \dump }
```

is not allowed. This restriction prevents difficulties with T<sub>E</sub>X's save stack. After the \dump command T<sub>E</sub>X gives an elaborate listing of its internal state, and of the font names associated with fonts that have been loaded and ends the job.

An interesting possibility arises from the fact that IniT<sub>E</sub>X can both load and dump a format. Suppose you have written a set of macros that build on top of plain T<sub>E</sub>X, superplain.tex. You could then call

```
% initex &plain superplain
*\dump
```

and get a format file superplain.fmt that has all of plain, and all of your macros.

### 33.1.3 Formats: preloading

On some systems it is possible to interrupt a running program, and save its 'core image' such that this can be started as an independent program. The executable made from the core image of a T<sub>E</sub>X program interrupted after it has loaded a format is called a T<sub>E</sub>X program with preloaded format. The idea

behind preloaded formats is that interrupting T<sub>E</sub>X after it has loaded a format, and making this program available to the user, saves in each run the time for loading the format. In the good old days when computers were quite a bit slower this procedure made sense. Nowadays, it does not seem so necessary. Besides, dumping a core image may not always be possible.

### 33.1.4 The knowledge of IniT<sub>E</sub>X

If no format has been loaded, IniT<sub>E</sub>X knows very little. For instance, it has no open/close group characters. However, it can not be completely devoid of knowledge lest there be no way to define anything.

Here is the extent of its knowledge.

- `\catcode`\=0, \escapechar=``` (see page 27).
- `\catcode``^M=5, \endlinechar=``^M` (see page 27).
- `\catcode`\ =10` (see page 28).
- `\catcode`\%=14` (see page 28).
- `\catcode``^?=15` (see page 28).
- `\catcode x=11` for  $x = \texttt{`a}..\texttt{`z}, \texttt{`A}..\texttt{`Z}$  (see page 28).
- `\catcode x=12` for all other character codes (see page 28).
- `\sfcode x=999` for  $x = \texttt{`A}..\texttt{`Z}$ , `\sfcode x=1000` for all other characters (see page 202).
- `\lccode`a..`z,`A..`Z=`a..`z, \uccode`a..`z,`A..`Z=`A..`Z, \lccode x=0, \uccode x=0` for all other characters (see page 46).
- `\delcode`. =0, \delcode x=-1` for all other characters (see page 207).
- `\mathcode x="!7100+x` for all lowercase and uppercase letters, `\mathcode x="!7000+x` for all digits, `\mathcode x=x` for all other characters (see page 212).
- `\tolerance=10000, \mag=1000, \maxdeadcycles=25`.

### 33.1.5 Memory sizes of T<sub>E</sub>X and IniT<sub>E</sub>X

The main memory size of T<sub>E</sub>X and IniT<sub>E</sub>X is controlled by four constants in the source code: `mem_bot`, `mem_top`, `mem_min`, and `mem_max`. For IniT<sub>E</sub>X's memory `mem_bot = mem_min` and `mem_top = mem_max`; for T<sub>E</sub>X `mem_bot` and `mem_top` record the main memory size of the IniT<sub>E</sub>X used to dump the format. Thus versions of T<sub>E</sub>X and IniT<sub>E</sub>X have to be adapted to each other in this respect.

$\text{T}_{\text{E}}\text{X}$ 's own main memory can be bigger than that of the corresponding  $\text{IniT}_{\text{E}}\text{X}$ : in general  $\text{mem\_min} \leq \text{mem\_bot}$  and  $\text{mem\_top} \leq \text{mem\_max}$ .

For  $\text{IniT}_{\text{E}}\text{X}$  a smaller main memory can suffice, as this program is typically not meant to do real typesetting. There may even be a real need for the main memory to be smaller, because  $\text{IniT}_{\text{E}}\text{X}$  needs a lot of auxiliary storage for initialization and for building the hyphenation table.

## 33.2 More about formats

### 33.2.1 Compatibility

$\text{T}_{\text{E}}\text{X}$  has a curious error message: ‘Fatal format error: I’m stymied’, which is given if  $\text{T}_{\text{E}}\text{X}$  tries to load a format that was made with an incompatible version of  $\text{IniT}_{\text{E}}\text{X}$ . See the point above about memory sizes, and Chapter 35 for the hash size (parameters `hash_size` and `hash_prime`) and the hyphenation exception dictionary (parameter `hyph_size`).

### 33.2.2 Preloaded fonts

During a run of  $\text{T}_{\text{E}}\text{X}$  the only information needed about fonts is the data that is found in the `tfm` files (see below). Since a run of  $\text{T}_{\text{E}}\text{X}$ , especially if the input contains math material, can easily access 30–40 fonts, the disk access for all the `tfm` files can become significant. Therefore the plain format and  $\text{\LaTeX}$  load these metrics files in  $\text{IniT}_{\text{E}}\text{X}$ . A  $\text{T}_{\text{E}}\text{X}$  version using such a format does not need to load any `tfm` files.

On the other hand, if a format has the possibility of accessing a range of typefaces, it may be advantageous to have metrics files loaded on demand during the actual run of  $\text{T}_{\text{E}}\text{X}$ .

### 33.2.3 The plain format

The first format written for  $\text{T}_{\text{E}}\text{X}$ , and the basis for all later ones, is the plain format, described in the  $\text{T}_{\text{E}}\text{X}$  book. It is a mixture of

- definitions and macros one simply cannot live without such as the initial `\catcode` assignments, all of the math delimiter definitions, and the `\new...` macros;
- constructs that are useful, but for which  $\text{\LaTeX}$  and other packages use a different implementation, such as the tabbing environment; and

- some macros that are insufficient for any but the simplest applications: `\item` and `\beginsection` are in this category.

It is the first category which Knuth meant to serve as a foundation for future macro packages, so that they can live peacefully together (see Chapter 31). This idea is reflected in the fact that the name ‘plain’ is not capitalized: it is the basic set of macros.

### 33.2.4 The L<sup>A</sup>T<sub>E</sub>X format

The L<sup>A</sup>T<sub>E</sub>X format, written by Leslie Lamport of Digital Equipment Corporation and described in [29], was released around 1985. The L<sup>A</sup>T<sub>E</sub>X format, using its own version of `plain.tex` (called `lplain.tex`), is not compatible with plain T<sub>E</sub>X; a number of plain macros are not available. Still, it contains large parts of the plain format (even when they overlap with its own constructs).

L<sup>A</sup>T<sub>E</sub>X is a powerful format with facilities such as marginal notes, floating objects, cross referencing, and automatic table of contents generation. Its main drawback is that the ‘style files’ which define the actual layout are quite hard to write (although L<sup>A</sup>T<sub>E</sub>X is in the process of a major revision, in which this problem will be tackled; see [34] and [33]). As a result, people have had at their disposal mostly the styles written by Leslie Lamport, the layout of which is rather idiosyncratic. See [6] for a successful attempt to replace these styles.

### 33.2.5 Mathematical formats

There are two formats with extensive facilities for mathematics typesetting: AmsT<sub>E</sub>X [43] (which originated at the American Mathematical Society) and LAMST<sub>E</sub>X [44]. The first of these includes more facilities than plain T<sub>E</sub>X or L<sup>A</sup>T<sub>E</sub>X for typesetting mathematics, but it lacks features such as automatic numbering and cross-referencing, available in L<sup>A</sup>T<sub>E</sub>X, for instance. LAMST<sub>E</sub>X, then, is the synthesis of AmsT<sub>E</sub>X and L<sup>A</sup>T<sub>E</sub>X. Also it includes still more features for mathematics, such as complicated tables and commutative diagrams.

### 33.2.6 Other formats

Other formats than the above exist: for instance, Phyzzx [51], TeXsis [35], Macro T<sub>E</sub>X [15], eplain [4], and T<sub>E</sub>Xt1 [13]. Typically, such formats provide the facilities of L<sup>A</sup>T<sub>E</sub>X, but try to be more easily adaptable by the user. Also, in general they have been written with the intention of being an add-on product to the plain format.

This book was, in its incarnation published by Addison-Wesley, also written in an ‘other format’: the *Lollipop* format. This format does not contain user macros, but the tools with which a style designer can program them; see [12]. The current version of this book is written in  $\text{\LaTeX}$ .

## 33.3 The dvi file

The dvi file (this term stands for ‘device independent’) contains the output of a  $\text{\TeX}$  run: it contains compactly dumped representations of boxes that have been sent there by `\shipout⟨box⟩`. The act of shipping out usually occurs inside the output routine, but this is not necessarily so.

### 33.3.1 The dvi file format

A dvi file is a byte-oriented file, consisting of a preamble, a postamble, and a list of pages.

Access for subsequent software to a completed dvi file is strictly sequential in nature: the pages are stored as a backwards linked list. This means that only two ways of accessing are possible:

- given the start of a page, the next can be found by reading until an end-of-page code is encountered, and
- starting at the end of the file pages can be read backwards at higher speed, as each beginning-of-page code contains the byte position of the previous one.

The preamble and postamble contain

- the magnification of the document (see below),
- the unit of measurement used for the document, and
- possibly a comment string.

The postamble contains in addition a list of the font definitions that appear on the pages of the file.

Neither the preamble nor the postamble of the file contains a table of byte positions of pages. The full definition of the dvi file format can be found in [23].

### 33.3.2 Page identification

Whenever a `\shipout` occurs,  $\text{\TeX}$  also writes the values of counters 0–9 to the dvi file and the terminal. Ordinarily, only counter 0, the page number, is used, and the other counters are zero. Those zeros are not output to the

terminal. The other counters can be used to indicate further structure in the document. Log output shows the non-zero counters and the zero counters in between.

### 33.3.3 Magnification

The *magnification* of a document can be indicated by the  $\langle$ integer parameter $\rangle$  `\mag`, which specifies 1000 times the magnification ratio.

The `dvi` file contains the value of `\mag` for the document in its preamble and postamble. If no true dimensions are used the `dvi` file will look the same as when no magnification would have been used, except for the `\mag` value in the preamble and the postamble.

Whenever a true dimension is used it is divided by the value of `\mag`, so that the final output will have the dimension as prescribed by the user. The `\mag` parameter cannot be changed after a true dimension has been used, or after the first page has been shipped to the `dvi` file.

Plain T<sub>E</sub>X has the `\magnification` macro for globally sizing the document, without changing the physical size of the page:

```
\def\magnification{\afterassignment\m@g\count@}
\def\m@g{\mag\count@
\hsize6.5truein\vsizer8.9truein\dimen\footins8truein}
```

The explanation for this is as follows: the command `\m@g` is saved with an `\afterassignment` command, and the magnification value (which is 1000 times the actual magnification factor) is assigned to `\count@`. After this assignment, the macro `\m@g` assigns the magnification value to `\mag`, and the horizontal and vertical size are reset to their original values `6.5truein` and `8.9truein`. The `\footins` is also reset.

## 33.4 Specials

T<sub>E</sub>X is to a large degree machineindependent, but it still needs a hook for machine-dependent extensions. This is done through *specials*. The `\special` command writes a  $\langle$ balanced text $\rangle$  to the `dvi` file which T<sub>E</sub>X does not interpret like other token lists: it assumes that the printer driver knows what to do with it. The `\special` command is not supposed to change the  $x$  and  $y$  position on the page, so that the implementation of T<sub>E</sub>X remains independent of the actual *device driver* that handles the `\special`.

The most popular application of specials is probably the inclusion of graphic material, written in some page description language, such as *PostScript*. The



size of the graphics can usually be determined from the file containing it (in the case of encapsulated PostScript through the ‘bounding box’ data), so  $\text{\TeX}$  can leave space for such material.

## 33.5 Time

$\text{\TeX}$  has four parameters, `\year`, `\month`, `\day`, and `\time`, that tell the *time* and *date* when the current job started. After this, the parameters are not updated. The user can change them without this having any effect.

All four parameters are integers; the `\time` parameter gives the number of minutes since midnight that the current job started.

## 33.6 Fonts

Font information is split in the  $\text{\TeX}$  system into the metric information (how high, wide, and deep is a character), and the actual description of the characters in a font.  $\text{\TeX}$ , the formatter, needs only the metric information; printer drivers and screen previewers need the character descriptions. With this approach it is for instance possible for  $\text{\TeX}$  to use with relative ease the resident fonts of a printer.

### 33.6.1 Font metrics

The metric information of  $\text{\TeX}$ ’s fonts is stored in `tfm` files, which stands for ‘ $\text{\TeX}$  font metrics’ files. Metrics files contain the following information (see [23] for the full definition):

- the design size of a font;
- the values for the `\fontdimen` parameters (see Chapter 4);
- the height, depth, width, and italic correction of individual characters;
- kerning tables;
- ligature tables;
- information regarding successors and extensions of math characters (see Chapter 21).

Metrics files use a packed format, but they can be converted to and from a readable format by the auxiliary programs `tftopl` and `pltotf` (see [26]). Here `pl` stands for ‘property list’, a term deriving from the programming language Lisp.

Files in p1 format are just text, so they can easily be edited; after conversion they can then again be used as tfm files.

### 33.6.2 Virtual fonts

With *virtual fonts* (see [24]) it is possible that what looks like one font to T<sub>E</sub>X resides in more than one physical font file. Also, virtual fonts can be used to change in effect the internal organization of font files.

For T<sub>E</sub>X itself, the presence of virtual fonts makes no difference: everything is still based on tfm files containing metric information. However, the screen or printer driver that displays the resulting dvi file on the screen or on a printer will search for files with extension .vf to determine how characters are to be interpreted. The vf file can, for instance, instruct the driver to interpret a character as a certain position in a certain font file, to interpret a character as more than one position (a way of forming accented characters), or to include \special information (for instance to set gray levels).

Readable variants of vf files have extension vpl, analogous to the p1 files for the tfm files; see above. Conversion between vf and vpl files can be performed with the vftovp and vptovf programs.

However, because virtual fonts are a matter for *device drivers*, no more details will be given in this book.

### 33.6.3 Font files

Character descriptions are stored in three types of files.

**gf** Generic Font files. This is the file type that the Metafont program generates. There are not many previewers or printer drivers that use this type of file directly.

**pxl** Pixel files. The pxl format is a pure bitmap format. Thus it is easy to generate pxl files from, for instance, scanner images.

This format should be superseded by the pk format. Pixel files can become rather big, as their size grows quadratically in the size of the characters.

**pk** Packed files. Pixel files can be packed by a form of run-length encoding: instead of storing the complete bitmap only the starting positions and lengths of ‘runs’ of black and white pixels are stored. This makes the size of pk files approximately linear in the size of the characters. However, a previewer or printer driver using a packed font file has to unpack it before it is able to use it.

The following conversion programs exist: `gftopxl`, `gftopk`, `pktopxl`, `pxltopk`.

### 33.6.4 Computer Modern

The only family of typefaces that comes with T<sub>E</sub>X in the standard distribution is the *Computer Modern* family of typefaces. This is an adaptation (using the terminology of [42]) by Donald Knuth of the Monotype Modern 8A typeface that was used for the first volume of his *Art of Computer Programming* series. The ‘modern faces’ all derive from the types that were cut between 1780 and 1800 by Firmin Didot in France, Giambattista Bodoni in Italy, and Justus Erich Walbaum in Germany. After the first two, these types are also called ‘Didone’ types. This name was coined in the Vox classification of types [50]. Ultimately, the inspiration for the Didone types is the ‘Romain du Roi’, the type that was designed by Nicolas Jaugeon around 1692 for the French Imprimerie Royale.

Didone types are characterized by a strong vertical orientation, and thin hairlines. The vertical accent is strengthened by the fact that the insides of curves are flattened. The result is a clear and brilliant page, provided that the printing is done carefully and on good quality paper. However, they are quite vulnerable; [48] compares them to the distinguished but fragile furniture from the same period, saying one is afraid to use either, ‘for both seem in danger of breaking in pieces’. With the current proliferation of low resolution (around 300 dot per inch) printers, the Computer Modern is a somewhat unfortunate choice.

Recently, Donald Knuth has developed a new typeface (or rather, a sub-family of typefaces) by changing parameters in the Computer Modern family. The result is a so-called ‘Egyptian’ typeface: Computer Concrete [22]. The name derives from the fact that it was intended primarily for the book *Concrete Mathematics*. Egyptian typefaces (they fall under the ‘Mécanes’ in the Vox classification, meaning constructed, not derived from written letters) have a very uniform line width and square serifs. They do not have anything to do with Egypt; such types happened to be popular in the first half of the nineteenth century when Egyptology was developing and popular.

## 33.7 T<sub>E</sub>X and web

The T<sub>E</sub>X program is written in web, a programming language that can be considered as a subset of *Pascal*, augmented with a preprocessor.

T<sub>E</sub>X makes no use of some features of Pascal, in order to facilitate porting to Pascal systems other than the one it was originally designed for, and even to enable automatic translation to other programming languages such as C. For instance, it does not use the Pascal `With` construct. Also, procedures do not have output parameters; apart from writing to global variables, the only way values are returned is through `Function` values.

Actually, web is more than a superset of a subset of Pascal (and to be more precise, it can also be used with other programming languages); it is a ‘system of structured documentation’. This means that the web programmer writes pieces of program code, interspersed with their documentation, in one file. This idea of ‘literate programming’ was introduced in [19]; for more information, see [41].

Two auxiliary programs, Tangle and Weave, can then be used to strip the documentation and convert web into regular Pascal, or to convert the web file into a T<sub>E</sub>X file that will typeset the program and documentation.

Portability of web programs is achieved by the ‘change file’ mechanism. A change file is a list of changes to be made to the web file; a bit like a stream editor script. These changes can comprise both adaptations of the web file to the particular Pascal compiler that will be used, and bug fixes to T<sub>E</sub>X. Thus the `TeX.web` file need never be edited.

## 33.8 The T<sub>E</sub>X Users Group

T<sub>E</sub>X users have joined into several users groups over the last decade. Many national or language users groups exist, and a lot of them publish newsletters. The oldest of all T<sub>E</sub>X users groups is simply called that: the T<sub>E</sub>X Users Group, or *TUG*, and its journal is called *TUGboat*. You can reach them at

T<sub>E</sub>X Users Group

P.O. Box 2311

Portland, OR 97208-2311, USA

or electronically at `office@tug.org` on the Internet.

## 第 34 章 Tracing

T<sub>E</sub>X's workings are often quite different from what the programmer expected, so there are ways to discover how T<sub>E</sub>X arrived at the result it did. The `\tracing...` commands all write *statistics* information of a certain kind to the log file (and to the terminal if `\tracingonline` is positive), and a number of `\show...` commands can be used to ask the current status or value of various items of T<sub>E</sub>X.

In the following list, only `\show` and `\showthe` display their output on the terminal by default, other `\show...` and `\tracing...` commands write to the log file. They will write in addition to the terminal if `\tracingonline` is positive.

- `\meaning` Give the meaning of a control sequence as a string of characters.
- `\show` Display the meaning of a control sequence.
- `\showthe` Display the result of prefixing a token with `\the`.
- `\showbox` Display the contents of a box.
- `\showlists` Display the contents of the partial lists currently built in all modes. This is treated on page 79.
- `\tracingcommands` If this is 1 T<sub>E</sub>X displays primitive commands executed; if this is 2 or more the outcome of conditionals is also recorded.
- `\tracingmacros` If this is 1, T<sub>E</sub>X shows expansion of macros that are performed and the actual values of the arguments; if this is 2 or more (token parameter)s such as `\output` and `\everypar` are also traced.
- `\tracingoutput` If this is positive, the log file shows a dump of boxes that are shipped to the dvi file.
- `\showboxdepth` The number of levels of box dump that are shown when boxes are displayed.
- `\showboxbreadth` Number of successive elements on each level that are shown when boxes are displayed.

- `\tracingonline` If this parameter is positive,  $\TeX$  will write trace information to the terminal in addition to the log file.
- `\tracingparagraphs` If this parameter is positive,  $\TeX$  generates a trace of the line breaking algorithm.
- `\tracingpages` If this parameter is positive,  $\TeX$  generates a trace of the page breaking algorithm.
- `\tracinglostchars` If this parameter is positive,  $\TeX$  gives diagnostic messages whenever a character is accessed that is not present in a font.  
Plain default: 1.
- `\tracingrestores` If this parameter is positive,  $\TeX$  will report all values that are restored when a group ends.
- `\tracingstats` If this parameter is 1,  $\TeX$  reports at the end of the job the usage of various internal arrays; if it is 2, the memory demands are given whenever a page is shipped out.

### 34.1 Meaning and content: `\show`, `\showthe`, `\meaning`

The meaning of control sequences, and the contents of those that represent internal quantities, can be obtained by the primitive commands `\show`, `\showthe`, and `\meaning`.

The control sequences `\show` and `\meaning` are similar: the former will give output to the log file and the terminal, whereas the latter will produce the same tokens, but they are placed in  $\TeX$ 's input stream.

The meaning of a primitive command of  $\TeX$  is that command itself:

```
\show\baselineskip
```

gives

```
\baselineskip=\baselineskip
```

The meaning of a defined quantity is its definition:

```
\show\pageno
```

gives

```
\pageno=\count0
```

The meaning of a macro is its parameter text and replacement text:

```
\def\foo#1?#2\par{\set{#1!}\set{#2?}}
\show\foo
```

gives

```
\foo=macro:
#1?#2\par ->\set {#1!}\set {#2?}
```

For macros without parameters the part before the arrow (the parameter text) is empty.

The `\showthe` command will display on the log file and terminal the tokens that `\the` produces. After `\show`, `\showthe`, `\showbox`, and `\showlists`  $\TeX$  asks the user for input; this can be prevented by specifying `\scrollmode`. Characters generated by `\meaning` and `\the` have category 12, except for spaces (see page 32); the value of `\escapechar` is used when control sequences are represented.

## 34.2 Show boxes: `\showbox`, `\tracingoutput`

If `\tracingoutput` is positive the log file will receive a dumped representation of all boxes that are written to the dvi file with `\shipout`. The same representation is used by the command `\showbox<8-bit number>`.

In the first case  $\TeX$  will report ‘Completed box being shipped out’; in the second case it will enter `\errorstopmode`, and tell the user ‘OK. (see the transcript file)’. If `\tracingonline` is positive, the box is also displayed on the terminal; if `\scrollmode` has been specified,  $\TeX$  does not stop for input.

The upper bound on the number of nested boxes that is dumped is `\showboxdepth`; each time a level is visited at most `\showboxbreadth` items are shown, the remainder of the list is summarized with `etc.` For each box its height, depth, and width are indicated in that order, and for characters it is stated from what font they were taken.

例子: *After*

```
\font\tenroman=cmr10 \tenroman
\setbox0=\hbox{g}
\showbox0
```

*the log file will show*

```
\hbox(4.30554+1.94444)x5.00002
.\tenroman g
```

*indicating that the box was 4.30554pt high, 1.94444pt deep, and 5.00002pt wide, and that it contained a character ‘g’ from the font \tenroman. Note that the fifth decimal of all sizes may be rounded because  $\TeX$  works with multiples of  $2^{-16}$ pt.*

The next example has nested boxes,

```
\vbox{\hbox{g}\hbox{o}}
```

and it contains `\baselineskip` glue between the boxes. After a `\showbox` command the log file output is:

```
\vbox(16.30554+0.0)x5.00002
.\hbox(4.30554+1.94444)x5.00002
..\tenroman g
.\glue(\baselineskip) 5.75002
.\hbox(4.30554+0.0)x5.00002
..\tenroman o
```

Each time a new level is entered an extra dot is added to the front of the line. Note that  $\TeX$  tells explicitly that the glue is `\baselineskip` glue; it inserts names like this for all automatically inserted glue. The value of the `baselineskip` glue here is such that the baselines of the boxes are at 12 point distance.

Now let us look at explicit (user) glue.  $\TeX$  indicates the ratio by which it is stretched or shrunk.

例子:

```
\hbox to 20pt {\kern10pt \hskip0pt plus 5pt}
```

*gives (indicating that the available stretch has been multiplied by 2.0):*

```
\hbox(0.0+0.0)x20.0, glue set 2.0
.\kern 10.0
.\glue 0.0 plus 5.0
```

and

```
\hbox to 0pt {\kern10pt \hskip0pt minus 20pt}
```

*gives (the shrink has been multiplied by 0.5)*

```
\hbox(0.0+0.0)x0.0, glue set - 0.5
.\kern 10.0
.\glue 0.0 minus 20.0
```

*respectively.*

This is an example with infinitely stretchable or shrinkable glue:

```
\hbox(4.00000+0.14000)x15.0, glue set 9.00000fil
```

This means that the horizontal box contained `fil` glue, and it was set such that its resulting width was 9pt.

Underfull boxes are dumped like all other boxes, but the usual ‘Underfull hbox detected at line...’ is given. Overfull horizontal boxes contain a vertical rule of width `\overfullrule`:

```
\hbox to 5pt {\kern10pt}
```

gives



```
\hbox(0.0+0.0)x5.0
.\kern 10.0
.\rule(**)x5.0
```

Box leaders are not dumped completely:

```
.\leaders 40.0
..\hbox(4.77313+0.14581)x15.0, glue set 9.76852fil
...\tenrm a
...\glue 0.0 plus 1.0fil
```

is the dump for

```
\leaders\hbox to 15pt{\tenrm a\hfil}\hskip 40pt
```

Preceding or trailing glue around the leader boxes is also not indicated.

## 34.3 Global statistics

The parameter `\tracingstats` can be used to force  $\text{\TeX}$  to report at the end of the job the global use of resources. Some production versions of  $\text{\TeX}$  may not have this option.

As an example, here are the statistics for this book:

Here is how much of TeX's memory you used:

String memory (bounded by ‘pool size’):

```
877 strings out of 4649
9928 string characters out of 61781
```

Main memory, control sequences, font memory:

```
53071 words of memory out of 262141
2528 multiletter control sequences out of 9500
20137 words of font info for 70 fonts,
      out of 72000 for 255
```

Hyphenation:

```
14 hyphenation exceptions out of 607
```

Stacks: input, nest, parameter, buffer, and save stack respectively,

```
17i,6n,19p,245b,422s stack positions out of
300i,40n,60p,3000b,4000s
```

# 第 35 章    Errors, Catastrophes, and Help

When  $\text{\TeX}$  is running, various errors can occur. This chapter treats how errors in the input are displayed, and what sort of overflow of internal data structures of  $\text{\TeX}$  can occur.

`\errorcontextlines` ( $\text{\TeX}$ 3 only) Number of additional context lines shown in error messages.

`\errmessage` Report an error, giving the parameter of this command as message.

`\errhelp` Tokens that will be displayed if the user asks further help after an `\errmessage`.

## 35.1 Error messages

When  $\text{\TeX}$  is running in `\errorstopmode` (which it usually is; see Chapter 32 for the other running modes), errors occurring are reported on the user terminal, and  $\text{\TeX}$  asks the user for further instructions. Errors can occur either because of some internal condition of  $\text{\TeX}$ , or because a macro has issued an `\errmessage` command.

If an error occurs  $\text{\TeX}$  shows the input line on which the error occurred. If the offending command was not on that line but, for instance, in a macro that was called – possibly indirectly – from that line, the line of that command is also shown. If the offending command was indirectly called, an additional `\errorcontextlines` number of lines is shown with the preceding macro calls.

A value of `\errorcontextlines = 0` causes ... to be printed as the sole indication that there is a context. Negative values inhibit even this.

For each macro in the sequence that leads to the offending command,  $\text{\TeX}$

attempts to display some preceding and some following tokens. First one line is displayed ending with the – indirectly – offending command; then, one line lower some following tokens are given.

例子:

```
This paragraph ends \vship1cm with a skip.
```

*gives*

```
! Undefined control sequence.
1.1 This paragraph ends \vship
                                1cm with a skip.
```

If  $\text{\TeX}$  is not running in some non-stop mode, the user is given the chance to do some *error patching*, or to ask for further information. In general the following options are available:

⟨**return**⟩  $\text{\TeX}$  will continue processing. If the error was something innocent that  $\text{\TeX}$  could either ignore or patch itself, this is the easy way out.

h Give further details about the error. If the error was caused by an `\err-` message command, the `\errhelp` tokens will be displayed here.

i Insert. The user can insert some material. For example, if a control sequence is misspelled, the correct command can sometimes be inserted, as

```
i\vskip
```

for the above example. Also, this is an opportunity for inserting `\show` commands to inspect  $\text{\TeX}$ 's internal state. However, if  $\text{\TeX}$  is in the middle of scanning something complicated, such commands will not be executed, or will even add to the confusion.

s (`\scrollmode`) Scroll further errors, but display the messages.  $\text{\TeX}$  will patch any further errors. This is a handy option, for instance if the error occurs in an alignment, because the number of subsequent errors tends to be rather large.

r (`\nonstopmode`) Run without stopping.  $\text{\TeX}$  will never stop for user interaction.

q (`\batchmode`) Quiet running.  $\text{\TeX}$  will never stop for user interaction, and does not give any more terminal output.

x Exit. Abort this run of  $\text{\TeX}$ .

e Edit. This option is not available on all  $\text{\TeX}$  system. If it is, the run of  $\text{\TeX}$  is aborted, and an editor is started, opening with the input file, maybe even on the offending line.

## 35.2 Overflow errors

Harsh reality imposes some restrictions on how elaborate  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 's workings can get. Some restrictions are imposed by compile-time constants, and are therefore fairly loose, but some depend strongly on the actual computer implementation.

Here follows the list of all categories of overflow that prompt  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  to report ‘Capacity exceeded’. Most bounds involved are (determined by) compile-time constants; their values given here in parentheses are those used in the source listing of  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  in [25]. Actual values may differ, and probably will. Remember that  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  was developed in the good old days when even big computers were fairly small.

### 35.2.1 Buffer size (500)

Current lines of all files that are open are kept in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ 's input buffer, as are control sequence names that are being built with `\csname . . . \endcsname`.

### 35.2.2 Exception dictionary (307)

The maximum number of hyphenation exceptions specified by `\hyphenation` must be a prime number. Two arrays with this many halfwords are allocated.

Changing this number makes formats incompatible; that is,  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  can only use a format that was made by an  $\mathrm{IniT}_{\mathrm{E}}\mathrm{X}$  with the same value for this constant.

### 35.2.3 Font memory (20,000)

Information about fonts is stored in an array of memory words. This is easily overflowed by preloading too many fonts in  $\mathrm{IniT}_{\mathrm{E}}\mathrm{X}$ .

### 35.2.4 Grouping levels

The number of open groups should be recordable in a quarter word. There is no compile-time constant corresponding to this.

### 35.2.5 Hash size (2100)

Maximum number of control sequences. It is suggested that this number should not exceed 10% of the main memory size. The values in  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  and  $\mathrm{IniT}_{\mathrm{E}}\mathrm{X}$  should agree; also the `hash_prime` values should agree.

This value is rather low; for macro packages that are more elaborate than plain  $\text{\TeX}$  a value of about 3000 is more realistic.

### 35.2.6 Number of strings (3000)

The maximum number of strings must be recordable in a half word.

### 35.2.7 Input stack size (200)

For each input source an item is allocated on the input stack. Typical input sources are input files (but their simultaneous number is more limited; see below), and token lists such as token variables, macro replacement texts, and alignment templates. A macro with ‘runaway recursion’ (for example, `\def\mac{{\mac}}`) will overflow this stack.

$\text{\TeX}$  performs some optimization here: before the last call in a token list all token lists ending with this call are cleared. This process is similar to ‘resolving tail recursion’ (see Chapter 11).

### 35.2.8 Main memory size (30,000)

Almost all ‘dynamic’ objects of  $\text{\TeX}$ , such as macro definition texts and all material on the current page, are stored in the main memory array. Formats may already take 20 000 words of main memory for macro definitions, and complicated pages containing for instance the  $\text{\LaTeX}$  picture environment may easily overflow this array.

$\text{\TeX}$ ’s main memory is divided in words, and a half word is supposed to be able to address the whole of the memory. Thus on current 32-bit computers the most common choice is to let the main memory size be at most 64K bytes. A half word address can then be stored in 16 bits, half a machine word.

However, so-called ‘Big  $\text{\TeX}$ ’ implementations exist that have a main memory larger than 64K words. Most compilers will then allocate 32-bit words for addressing this memory, even if (say) 18 bits would suffice. Big  $\text{\TeX}$ s therefore become immediately a lot bigger when they cross the 64K threshold. Thus they are usually not found on microcomputers, although virtual memory schemes for these are possible; see for instance [45].

$\text{\TeX}$  can have a bigger main memory than  $\text{\InitEX}$ ; see Chapter 33 for further details.

### 35.2.9 Parameter stack size (60)

Macro parameters may contain macro calls with further parameters. The number of parameters that may occur nested is bounded by the parameter stack size.

### 35.2.10 Pattern memory (8000)

Hyphenation patterns are stored in a trie array. The default size of 8000 hyphenation patterns seems sufficient for English or Italian, for example, but it is not for Dutch or German.

### 35.2.11 Pattern memory ops per language

The number of hyphenation ops (see the literature about hyphenation: [30] and appendix H of [25]) should be recordable in a quarter word. There is no compile-time constant corresponding to this.  $\text{\TeX}$  version 2 had the same upper bound, but gave no error message in case of overflow. Again, for languages such as Dutch and German this bound is too low. There are versions of  $\text{\TeX}$  that have a higher bound here.

### 35.2.12 Pool size (32,000)

Strings are error messages and control sequence names. They are stored using one byte per character.  $\text{\TeX}$  has initially about 23 000 characters worth of strings.

The pool will overflow if a user defines a large number of control sequences on top of a substantial macro package. However, even if the user does not define any new commands overflow may occur: crossreferencing schemes also work by defining control sequences. For large documents a pool size of 40 000 or 60 000 is probably sufficient.

### 35.2.13 Save size (600)

Quantities that are assigned to inside a group must be restored after the end of that group. The save stack is where the values to be restored are kept; the size of the save stack limits the number of values that can be restored.

Alternating global and local assignments to a value will lead to ‘save stack build-up’: for each local assignment following a global assignment the previous

value of the variable is saved. Thus an alternation of such assignments will lead to an unnecessary proliferation of items on the save stack.

#### 35.2.14 Semantic nest size (40)

Each time  $\text{\TeX}$  switches to a mode nested inside another mode (for instance when processing an `\hbox` inside a `\vbox`) the current state is pushed on the semantic nest stack. The semantic nest size is the maximum number of levels that can be pushed.

#### 35.2.15 Text input levels (6)

The number of nested `\input` files has to be very limited, as the current lines are all kept in the input buffer.

## 第 36 章 The Grammar of T<sub>E</sub>X

Many chapters in this book contain pieces of the grammar that defines the formal syntax of T<sub>E</sub>X. In this chapter the structure of the rewriting rules of the grammar is explained, and some key notions are presented.

In the T<sub>E</sub>X book a grammar appears in Chapters 24–27. An even more rigorous grammar of T<sub>E</sub>X can be found in [1]. The grammar presented in this book is virtually identical to that of the T<sub>E</sub>X book.

### 36.1 Notations

Basic to the grammar are

**grammatical terms** These are enclosed in angle brackets:

`<term>`

**control sequences** These are given in typewriter type with a backslash for the escape character:

`\command`

Lastly there are

**keywords** Also given in typewriter type

`keyword`

This is a limited collection of words that have a special meaning for T<sub>E</sub>X in certain contexts; see below.

The three elements of the grammar are used in syntax rules:

`<snark> → boojum | <empty>`

This rule says that the grammatical entity `<snark>` is either the keyword `boojum`, or the grammatical entity `<empty>`.

There are two other notational conventions. The first is that the double quote is used to indicate hexadecimal (base 16) notation. For instance `"ab56`



stands for  $10 \times 16^3 + 11 \times 16^2 + 5 \times 16^1 + 6 \times 16^0$ . The second convention is that subscripts are used to denote category codes. Thus  $a_{12}$  denotes an ‘a’ of category 12.

## 36.2 Keywords

A keyword is sequence of characters (or character tokens) of any category code but 13 (active). Unlike the situation in control sequences,  $\text{\TeX}$  does not distinguish between lowercase and uppercase characters in keywords. Uppercase characters in keywords are converted to lowercase by adding 32 to them; the `\lccode` and `\uccode` are not used here. Furthermore, any keyword can be preceded by optional spaces.

Thus both `true cm` and `truecm` are legal. By far the strangest example, however, is provided by the grammar rule

$$\langle \text{fil unit} \rangle \longrightarrow \text{fil} \mid \langle \text{fil unit} \rangle 1$$

which implies that `fil L 1` is also a legal  $\langle \text{fil dimen} \rangle$ . Strange errors can ensue from this; see page 137 for an example.

Here is the full list of all keywords: `at`, `bp`, `by`, `cc`, `cm`, `dd`, `depth`, `em`, `ex`, `fil`, `height`, `in`, `l`, `minus`, `mm`, `mu`, `pc`, `plus`, `pt`, `scaled`, `sp`, `spread`, `to`, `true`, `width`.

## 36.3 Specific grammatical terms

Some grammatical terms appear in a lot of rules. One such term is  $\langle \text{optional spaces} \rangle$ . The term *optional space* is probably clear enough, but here is the formal definition:

$$\langle \text{optional spaces} \rangle \longrightarrow \langle \text{empty} \rangle \mid \langle \text{space token} \rangle \langle \text{optional spaces} \rangle$$

which amounts to saying that  $\langle \text{optional spaces} \rangle$  is zero or more space tokens.

Other terms may not be so immediately obvious. Below are some of them.

### 36.3.1 $\langle \text{equals} \rangle$

In assignments the equals sign is optional; therefore there is a term

$$\langle \text{equals} \rangle \longrightarrow \langle \text{optional spaces} \rangle \mid \langle \text{optional spaces} \rangle =_{12}$$

in  $\text{\TeX}$ ’s grammar.

### 36.3.2 $\langle \text{filler} \rangle$ , $\langle \text{general text} \rangle$

More obscure than the  $\langle \text{optional spaces} \rangle$  is the combination of spaces and `\relax` tokens that is allowed in some places, for instance

```
\setbox0= \relax\box1
```

The quantity involved is

$$\langle \text{filler} \rangle \longrightarrow \langle \text{optional spaces} \rangle \mid \langle \text{filler} \rangle \backslash \text{relax} \langle \text{optional spaces} \rangle$$

One important occurrence of  $\langle \text{filler} \rangle$  is in

$$\langle \text{general text} \rangle \longrightarrow \langle \text{filler} \rangle \{ \langle \text{balanced text} \rangle \langle \text{right brace} \rangle$$

A  $\langle \text{general text} \rangle$  follows such control sequences as `\message`, `\uppercase`, or `\mark`. The braces around the  $\langle \text{balanced text} \rangle$  are explained in the next point.

### 36.3.3 $\{ \}$ and $\langle \text{left brace} \rangle \langle \text{right brace} \rangle$

The T<sub>E</sub>X grammar uses a perhaps somewhat unfortunate convention for braces. First of all

`{` and `}`

stand for braces that are either explicit open/close group characters, or control sequences defined by `\let`, such as

```
\let\bgroup={ \let\egroup=}
```

The grammatical terms

$$\langle \text{left brace} \rangle \text{ and } \langle \text{right brace} \rangle$$

stand for explicit open/close group characters, that is, characters of categories 1 and 2 respectively.

Various combinations of these two kinds of braces exist. Braces around boxes can be implicit:

$$\backslash \text{hbox} \langle \text{box specification} \rangle \{ \langle \text{horizontal mode material} \rangle \}$$

Around a macro definition there must be explicit braces:

$$\langle \text{definition text} \rangle \longrightarrow \langle \text{parameter text} \rangle \langle \text{left brace} \rangle \langle \text{balanced text} \rangle \langle \text{right brace} \rangle$$

Finally, the  $\langle \text{general text} \rangle$  that was mentioned above has to be explicitly closed, but it can be implicitly opened:

$$\langle \text{general text} \rangle \longrightarrow \langle \text{filler} \rangle \{ \langle \text{balanced text} \rangle \langle \text{right brace} \rangle$$

The closing brace of a  $\langle \text{general text} \rangle$  has to be explicit, since a general text is a token list, which may contain `\egroup` tokens. T<sub>E</sub>X performs expansion to find the opening brace of a  $\langle \text{general text} \rangle$ .

### 36.3.4 $\langle\text{math field}\rangle$

In math mode various operations such as subscripting or applying `\underline` take an argument that is a  $\langle\text{math field}\rangle$ : either a single symbol, or a group. Here is the exact definition.

$$\begin{aligned}\langle\text{math field}\rangle &\longrightarrow \langle\text{math symbol}\rangle \mid \langle\text{filler}\rangle\{\langle\text{math mode material}\rangle\} \\ \langle\text{math symbol}\rangle &\longrightarrow \langle\text{character}\rangle \mid \langle\text{math character}\rangle\end{aligned}$$

See page 44 for  $\langle\text{character}\rangle$ , and page 206 for  $\langle\text{math character}\rangle$ .

## 36.4 Differences between T<sub>E</sub>X versions 2 and 3

In 1989 Knuth released T<sub>E</sub>X version 3.0, which is the first real change in T<sub>E</sub>X since version 2.0, which was released in 1986 (version 0 of T<sub>E</sub>X was released in 1982; see [18] for more about the history of T<sub>E</sub>X). All intermediate versions were merely bug fixes.

The main difference between versions 2 and 3 lies in the fact that 8-bit input has become possible. Associated with this, various quantities that used to be 127 or 128 have been raised to 255 or 256 respectively. Here is a short list. The full description is in [20].

All ‘codes’ (`\catcode`, `\sfcode`, et cetera; see page 47) now apply to 256 character codes instead of 128.

A character with code `\endlinechar` is appended to the line unless this parameter is negative or more than 255 (this was 127) (see page 27).

No escape character is output by `\write` and other commands if `\escapechar` is negative or more than 255 (this was 127) (see page 33).

The  $\sim$  replacement mechanism has been extended (see page 30).

Parameters `\language`, `\inputlineno`, `\errorcontextlines`, `\lefthyphenmin`, `\righthyphenmin`, `\badness`, `\holdinginserts`, `\emergencystretch`, and commands `\noboundary`, `\setlanguage` have been added.

The value of `\outputpenalty` is no longer zero if the page break was not at a penalty item; instead it is 10 000 (see page 247).

The plain format has also been updated, mostly with default settings for parameters such as `\lefthyphenmin`, but also a few macros have been added.

## 第 37 章 Glossary of T<sub>E</sub>X Primitives

This chapter gives the list of all primitives of T<sub>E</sub>X. After each control sequence the grammatical category of the command or parameter is given, plus a short description. For some commands the syntax of their use is given.

For parameters the class to which they belong is given. Commands that have no grammatical category in the T<sub>E</sub>X book are denoted either ‘<expandable command>’ or ‘<primitive command>’ in this list.

Grammatical terms such as <equals> and <optional space> are explained in Chapter 36.

`\-` <horizontal command> Discretionary hyphen; this is equivalent to `\discretionary""{-}{-}{-}`. Can be used to indicate hyphenatable points in a word. Chapter 19.

`\char32` <horizontal command> Control space. Insert the same amount of space as a space token would if `\spacefactor = 1000`. Chapter 2,20.

`\char47` <primitive command> Italic correction: insert a kern specified by the preceding character. Each character has an italic correction, possibly zero, specified in the tfm file. For slanted fonts this compensates for overhang. Chapter 4.

`\above<dimen>` <generalized fraction command> Fraction with specified bar width. Chapter 23.

`\abovedisplayshortskip` <glue parameter> Glue above a display if the line preceding the display was short. Chapter 24.

`\abovedisplayskip` <glue parameter> Glue above a display. Chapter 24.

`\abovewithdelims<delim1><delim2><dimen>` <generalized fraction command> Generalized fraction with delimiters. Chapter 23.

`\accent<8-bit number><optional assignments><character>` <horizontal command> Command to place accents on characters. Chapter 3.

`\adjdemerits` <integer parameter> Penalty for adjacent not visually

compatible lines. Default 10 000 in plain  $\text{\TeX}$ . Chapter 19.

$\backslash\text{advance}\langle\text{numeric variable}\rangle\langle\text{optional by}\rangle\langle\text{number}\rangle\langle\text{arithmetic assignment}\rangle$   
Arithmetic command to increase or decrease a  $\langle\text{numeric variable}\rangle$ , that is, a  
 $\langle\text{count variable}\rangle$ ,  $\langle\text{dimen variable}\rangle$ ,  $\langle\text{glue variable}\rangle$ , or  $\langle\text{muglue variable}\rangle$ .  
Chapter 7,8.

$\backslash\text{afterassignment}\langle\text{token}\rangle\langle\text{primitive command}\rangle$  Save the next token for  
execution after the next assignment. Only one token can be saved this way.  
Chapter 12.

$\backslash\text{aftergroup}\langle\text{token}\rangle\langle\text{primitive command}\rangle$  Save the next token for insertion  
after the current group. Several tokens can be saved this way. Chapter 10.

$\backslash\text{atop}\langle\text{dimen}\rangle\langle\text{generalized fraction command}\rangle$  Place objects over one another.  
Chapter 23.

$\backslash\text{atopwithdelims}\langle\text{delim}_1\rangle\langle\text{delim}_2\rangle\langle\text{generalized fraction command}\rangle$  Place  
objects over one another with delimiters. Chapter 23.

$\backslash\text{badness}\langle\text{internal integer}\rangle$  ( $\text{\TeX}$ 3 only) Badness of the most recently  
constructed box. Chapter 5.

$\backslash\text{baselineskip}\langle\text{glue parameter}\rangle$  The ‘ideal’ baseline distance between  
neighbouring boxes on a vertical list; 12pt in plain  $\text{\TeX}$ . Chapter 15.

$\backslash\text{batchmode}\langle\text{interaction mode assignment}\rangle$   $\text{\TeX}$  patches errors itself and  
performs an emergency stop on serious errors such as missing input files, but  
no terminal output is generated. Chapter 32.

$\backslash\text{begingroup}\langle\text{primitive command}\rangle$  Open a group that must be closed with  
 $\backslash\text{endgroup}$ . Chapter 10.

$\backslash\text{belowdisplayshortskip}\langle\text{glue parameter}\rangle$  Glue below a display if the line  
preceding the display was short. Chapter 24.

$\backslash\text{belowdisplayskip}\langle\text{glue parameter}\rangle$  Glue below a display. Chapter 24.

$\backslash\text{binoppenalty}\langle\text{integer parameter}\rangle$  Penalty for breaking after a binary  
operator not enclosed in a subformula. Plain  $\text{\TeX}$  default: 700. Chapter 23.

$\backslash\text{bookmark}\langle\text{expandable command}\rangle$  The last mark on the current page.  
Chapter 28.

$\backslash\text{box}\langle\text{8-bit number}\rangle\langle\text{box}\rangle$  Use a box register, emptying it. Chapter 5.

$\backslash\text{boxmaxdepth}\langle\text{dimen parameter}\rangle$  Maximum allowed depth of boxes.  
Default  $\backslash\text{maxdimen}$  in plain  $\text{\TeX}$ . Chapter 5.

$\backslash\text{brokenpenalty}\langle\text{integer parameter}\rangle$  Additional penalty for breaking a page  
after a hyphenated line. Default 100 in plain  $\text{\TeX}$ . Chapter 27.

`\catcode``<8-bit number>` `<internal integer>`; the control sequence itself is a `<codename>`. Access category codes. Chapter 2.

`\char``<number>` `<character>` Explicit denotation of a character to be typeset. Chapter 3.

`\chardef``<control sequence>``<equals>``<number>` `<shorthand definition>` Define a control sequence to be a synonym for a character code. Chapter 3.

`\cleaders` `<leaders>` As `\leaders`, but with box leaders any excess space is split into equal glue items before and after the leaders. Chapter 9.

`\closein``<4-bit number>` `<primitive command>` Close an input stream. Chapter 30.

`\closeout``<4-bit number>` `<primitive command>` Close an output stream. Chapter 30.

`\clubpenalty` `<integer parameter>` Additional penalty for breaking a page after the first line of a paragraph. Default 150 in plain T<sub>E</sub>X. Chapter 27.

`\copy``<8-bit number>` `<box>` Use a box register and retain the contents. Chapter 5.

`\count``<8-bit number>` `<internal integer>`; the control sequence itself is a `<register prefix>`. Access count registers. Chapter 7.

`\countdef``<control sequence>``<equals>``<8-bit number>` `<shorthand definition>`; the control sequence itself is a `<registerdef>`. Define a control sequence to be a synonym for a `\count` register. Chapter 7.

`\cr` `<primitive command>` Terminate an alignment line. Chapter 25.

`\crrcr` `<primitive command>` Terminate an alignment line if it has not already been terminated by `\cr`. Chapter 25.

`\csname` `<expandable command>` Start forming the name of a control sequence. Has to be balanced with `\endcsname`. Chapter 11.

`\day` `<integer parameter>` The day of the current job. Chapter 33.

`\deadcycles` `<special integer>` Counter that keeps track of how many times the output routine has been called without a `\shipout` taking place. If this number reaches `\maxdeadcycles` T<sub>E</sub>X gives an error message. Plain T<sub>E</sub>X default: 25. Chapter 28.

`\def` `<def>` Start a macro definition. Chapter 11.

`\defaultshyphenchar` `<integer parameter>` Value of `\hyphenchar` when a font is loaded. Default value in plain T<sub>E</sub>X is `\-`. Chapter 4,19.

`\defaultskewchar` `<integer parameter>` Value of `\skewchar` when a font is

loaded. Default value in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  is  $-1$ . Chapter 21.

`\delcode` $\langle 8\text{-bit number} \rangle \langle \text{internal integer} \rangle$ ; the control sequence itself is a  $\langle \text{codename} \rangle$ . Access the code specifying how a character should be used as delimiter after `\left` or `\right`. Chapter 21.

`\delimiter` $\langle 27\text{-bit number} \rangle \langle \text{math character} \rangle$  Explicit denotation of a delimiter. Chapter 21.

`\delimiterfactor`  $\langle \text{integer parameter} \rangle$  1000 times the part of a delimited formula that should be covered by a delimiter. Plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  default: 901. Chapter 21.

`\delimitershortfall`  $\langle \text{integer parameter} \rangle$  Size of the part of a delimited formula that is allowed to go uncovered by a delimiter. Plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  default: 5pt. Chapter 21.

`\dimen` $\langle 8\text{-bit number} \rangle \langle \text{internal dimen} \rangle$ ; the control sequence itself is a  $\langle \text{register prefix} \rangle$ . Access `dimen` registers. Chapter 8.

`\dimendef` $\langle \text{control sequence} \rangle \langle \text{equals} \rangle \langle 8\text{-bit number} \rangle \langle \text{shorthand definition} \rangle$ ; the control sequence itself is a  $\langle \text{registerdef} \rangle$ . Define a control sequence to be a synonym for a `\dimen` register. Chapter 8.

`\discretionary` $\{pre\text{-}break\}\{post\text{-}break\}\{no\text{-}break\}$   $\langle \text{horizontal command} \rangle$  Specify the way a character sequence is split up at a line break. Chapter 19.

`\displayindent`  $\langle \text{dimen parameter} \rangle$  Distance by which the box, in which the display is centred, is indented owing to hanging indentation. This value is set automatically for each display. Chapter 24.

`\displaylimits`  $\langle \text{primitive command} \rangle$  Restore default placement for limits. Chapter 23.

`\displaystyle`  $\langle \text{primitive command} \rangle$  Select the display style of math typesetting. Chapter 23.

`\displaywidowpenalty`  $\langle \text{integer parameter} \rangle$  Additional penalty for breaking a page before the last line above a display formula. Default 50 in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ . Chapter 27.

`\displaywidth`  $\langle \text{dimen parameter} \rangle$  Width of the box in which the display is centred. This value is set automatically for each display. Chapter 24.

`\divide` $\langle \text{numeric variable} \rangle \langle \text{optional by} \rangle \langle \text{number} \rangle \langle \text{arithmetic assignment} \rangle$  Arithmetic command to divide a  $\langle \text{numeric variable} \rangle$  (see `\advance`). Chapter 7.

`\doublehyphendemerits`  $\langle \text{integer parameter} \rangle$  Penalty for consecutive lines ending with a hyphen. Default 10 000 in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ . Chapter 19.

`\dp` $\langle 8\text{-bit number} \rangle \langle \text{internal dimen} \rangle$ ; the control sequence itself is a  $\langle \text{box dimension} \rangle$ . Depth of the box in a box register. Chapter 5.

`\dump`  $\langle \text{vertical command} \rangle$  Dump a format file; possible only in IniT<sub>E</sub>X, not allowed inside a group.

Chapter 33.

`\edef`  $\langle \text{def} \rangle$  Start a macro definition; the replacement text is expanded at definition time. Chapter 11.

`\else`  $\langle \text{expandable command} \rangle$  Select  $\langle \text{false text} \rangle$  of a conditional or default case of `\ifcase`. Chapter 13.

`\emergencystretch`  $\langle \text{dimen parameter} \rangle$  (T<sub>E</sub>X3 only) Assumed extra stretchability in lines of a paragraph in third pass of the line-breaking algorithm. Chapter 19.

`\end`  $\langle \text{vertical command} \rangle$  End this run. Chapter 32.

`\endcsname`  $\langle \text{expandable command} \rangle$  Delimit the name of a control sequence that was begun with `\csname`. Chapter 11.

`\endgroup`  $\langle \text{primitive command} \rangle$  End a group that was opened with `\begingroup`. Chapter 10.

`\endinput`  $\langle \text{expandable command} \rangle$  Terminate inputting the current file after the current line. Chapter 30.

`\endlinechar`  $\langle \text{integer parameter} \rangle$  The character code of the end-of-line character appended to input lines. IniT<sub>E</sub>X default: 13. Chapter 2.

`\eqno` $\langle \text{math mode material} \rangle \mathbb{S} \mathbb{S} \langle \text{eqno} \rangle$  Place a right equation number in a display formula. Chapter 24.

`\errhelp`  $\langle \text{token parameter} \rangle$  Tokens that will be displayed if the user asks for help after an `\errmessage`. Chapter 35.

`\errmessage` $\langle \text{general text} \rangle \langle \text{primitive command} \rangle$  Report an error and give the user opportunity to act. Chapter 35.

`\errorcontextlines`  $\langle \text{integer parameter} \rangle$  (T<sub>E</sub>X3 only) Number of additional context lines shown in error messages. Chapter 35.

`\errorstopmode`  $\langle \text{interaction mode assignment} \rangle$  Ask for user input on the occurrence of an error. Chapter 32.

`\escapechar`  $\langle \text{integer parameter} \rangle$  Number of the character that is used when control sequences are being converted into character tokens. IniT<sub>E</sub>X default: 92. Chapter 3.

`\everycr`  $\langle \text{token parameter} \rangle$  Token list inserted after every `\cr` or



non-redundant `\crrc`. Chapter 25.

`\everydisplay`  $\langle$ token parameter $\rangle$  Token list inserted at the start of a display. Chapter 24.

`\everyhbox`  $\langle$ token parameter $\rangle$  Token list inserted at the start of a horizontal box. Chapter 5.

`\everyjob`  $\langle$ token parameter $\rangle$  Token list inserted at the start of each job. Chapter 32.

`\everymath`  $\langle$ token parameter $\rangle$  Token list inserted at the start of non-display math. Chapter 23.

`\everypar`  $\langle$ token parameter $\rangle$  Token list inserted in front of paragraph text. Chapter 16.

`\everyvbox`  $\langle$ token parameter $\rangle$  Token list inserted at the start of a vertical box. Chapter 5.

`\exhyphenpenalty`  $\langle$ integer parameter $\rangle$  Penalty for breaking a horizontal line at a discretionary in the special case where the prebreak text is empty. Default 50 in plain  $\text{\TeX}$ . Chapter 19.

`\expandafter`  $\langle$ expandable command $\rangle$  Take the next two tokens and place the expansion of the second after the first. Chapter 12.

`\fam`  $\langle$ integer parameter $\rangle$  The number of the current font family. Chapter 22.

`\fi`  $\langle$ expandable command $\rangle$  Closing delimiter for all conditionals. Chapter 13.

`\finalhyphendemerits`  $\langle$ integer parameter $\rangle$  Penalty added when the penultimate line of a paragraph ends with a hyphen. Plain  $\text{\TeX}$  default 5000. Chapter 19.

`\firstmark`  $\langle$ expandable command $\rangle$  The first mark on the current page. Chapter 28.

`\floatingpenalty`  $\langle$ integer parameter $\rangle$  Penalty amount added to `\insertpenalties` when an insertion is split. Chapter 29.

`\font` $\langle$ control sequence $\rangle$  $\langle$ equals $\rangle$  $\langle$ file name $\rangle$  $\langle$ at clause $\rangle$   $\langle$ simple assignment $\rangle$  Associate a control sequence with a tfm file. When used on its own, this control sequence is a  $\langle$ font $\rangle$ , denoting the current font. Chapter 4.

`\fontdimen` $\langle$ number $\rangle$  $\langle$ font $\rangle$   $\langle$ internal dimen $\rangle$  Access various parameters of fonts. Chapter 4.

`\fontname` $\langle$ font $\rangle$   $\langle$ primitive command $\rangle$  The external name of a font. Chapter 4.

`\futurelet` $\langle$ control sequence $\rangle$  $\langle$ token $_1$  $\rangle$  $\langle$ token $_2$  $\rangle$   $\langle$ let assignment $\rangle$  Assign the

meaning of  $\langle \text{token}_2 \rangle$  to the  $\langle \text{control sequence} \rangle$ . Chapter 11.

`\gdef`  $\langle \text{def} \rangle$  Synonym for `\global\def`. Chapter 11.

`\global`  $\langle \text{prefix} \rangle$  Make the next definition, arithmetic statement, or assignment global. Chapter 10,11.

`\globaldefs`  $\langle \text{integer parameter} \rangle$  Override `\global` specifications: a positive value of this parameter makes all assignments global, a negative value makes them local. Chapter 10.

`\halign` $\langle \text{box specification} \rangle \{ \langle \text{alignment material} \rangle \}$   $\langle \text{vertical command} \rangle$   
Horizontal alignment. Display alignment:

$$$\halign\langle \text{box specification} \rangle \{ \dots \} \langle \text{optional assignments} \rangle $$$

Chapter 25.

`\hangafter`  $\langle \text{integer parameter} \rangle$  If positive, this denotes the number of lines before indenting starts; if negative, its absolute value is the number of indented lines starting with the first line of the paragraph. The default value of 1 is restored after every paragraph. Chapter 18.

`\hangindent`  $\langle \text{dimen parameter} \rangle$  If positive, this indicates indentation from the left margin; if negative, this is the negative of the indentation from the right margin. The default value of 0pt is restored after every paragraph. Chapter 18.

`\hbadness`  $\langle \text{integer parameter} \rangle$  Threshold below which T<sub>E</sub>X does not report an underfull or overfull horizontal box. Plain T<sub>E</sub>X default: 1000. Chapter 5.

`\hbox` $\langle \text{box specification} \rangle \{ \langle \text{horizontal material} \rangle \}$   $\langle \text{box} \rangle$  Construct a horizontal box. Chapter 5.

`\hfil`  $\langle \text{horizontal command} \rangle$  Horizontal skip equivalent to `\hskip 0cm plus 1fil`. Chapter 8.

`\hfill`  $\langle \text{horizontal command} \rangle$  Horizontal skip equivalent to `\hskip 0cm plus 1fill`. Chapter 8.

`\hfilneg`  $\langle \text{horizontal command} \rangle$  Horizontal skip equivalent to `\hskip 0cm minus 1fil`. Chapter 8.

`\hfuzz`  $\langle \text{dimen parameter} \rangle$  Excess size that T<sub>E</sub>X tolerates before it considers a horizontal box overfull. Plain T<sub>E</sub>X default: 0.1pt. Chapter 5.

`\hoffset`  $\langle \text{dimen parameter} \rangle$  Distance by which the page is shifted to the right of the reference point which is at one inch from the left margin. Chapter 26.

`\holdinginserts`  $\langle \text{integer parameter} \rangle$  (only T<sub>E</sub>X3) If this is positive,

insertions are not placed in their boxes when the `\output` tokens are inserted. Chapter 29.

`\hrule` `<vertical command>` Rule that spreads in horizontal direction. Chapter 9.

`\hsize` `<dimen parameter>` Line width used for text typesetting inside a vertical box. Chapter 5,18.

`\hskip``<glue>` `<horizontal command>` Insert in horizontal mode a glue item. Chapter 8.

`\hss` `<horizontal command>` Horizontal skip equivalent to `\hskip 0cm plus 1fil minus 1fil`. Chapter 8.

`\ht``<8-bit number>` `<internal dimen>`; the control sequence itself is a `<box dimension>`. Height of the box in a box register. Chapter 5.

`\hyphenation``<general text>` `<hyphenation assignment>` Define hyphenation exceptions for the current value of `\language`. Chapter 19.

`\hyphenchar``<font>` `<internal integer>` Number of the character behind which a `\discretionary{}{}{}` is inserted. Chapter 4,19.

`\hyphenpenalty` `<integer parameter>` Penalty associated with break at a discretionary in the general case. Default 50 in plain  $\TeX$ . Chapter 19.

`\if``<token1>``<token2>` `<expandable command>` Test equality of character codes. Chapter 13.

`\ifcase``<number>``<case0>``\or... \or``<casen>``\else``<other cases>``\fi` `<expandable command>` Enumerated case statement. Chapter 13.

`\ifcat``<token1>``<token2>` `<expandable command>` Test whether two characters have the same category code. Chapter 13.

`\ifdim``<dimen1>``<relation>``<dimen2>` `<expandable command>` Compare two dimensions. Chapter 13.

`\ifeof``<4-bit number>` `<expandable command>` Test whether a file has been fully read, or does not exist. Chapter 30.

`\iffalse` `<expandable command>` This test is always false. Chapter 13.

`\ifhbox``<8-bit number>` `<expandable command>` Test whether a box register contains a horizontal box. Chapter 5.

`\ifhmode` `<expandable command>` Test whether the current mode is (possibly restricted) horizontal mode. Chapter 13.

`\ifinner` `<expandable command>` Test whether the current mode is an internal mode. Chapter 13.

`\ifmmode`  $\langle$ expandable command $\rangle$  Test whether the current mode is (possibly display) math mode. Chapter 13.

`\ifnum` $\langle$ number<sub>1</sub> $\rangle$  $\langle$ relation $\rangle$  $\langle$ number<sub>2</sub> $\rangle$   $\langle$ expandable command $\rangle$  Test relations between numbers. Chapter 13.

`\ifodd` $\langle$ number $\rangle$   $\langle$ expandable command $\rangle$  Test whether a number is odd. Chapter 13.

`\iftrue`  $\langle$ expandable command $\rangle$  This test is always true. Chapter 13.

`\ifvbox` $\langle$ 8-bit number $\rangle$   $\langle$ expandable command $\rangle$  Test whether a box register contains a vertical box. Chapter 5.

`\ifvmode`  $\langle$ expandable command $\rangle$  Test whether the current mode is (possibly internal) vertical mode. Chapter 13.

`\ifvoid` $\langle$ 8-bit number $\rangle$   $\langle$ expandable command $\rangle$  Test whether a box register is empty. Chapter 13,5.

`\ifx` $\langle$ token<sub>1</sub> $\rangle$  $\langle$ token<sub>2</sub> $\rangle$   $\langle$ expandable command $\rangle$  Test equality of macro expansion, or equality of character code and category code. Chapter 13.

`\ignorespaces`  $\langle$ primitive command $\rangle$  Expands following tokens until something other than a  $\langle$ space token $\rangle$  is found. Chapter 2.

`\immediate`  $\langle$ primitive command $\rangle$  Prefix to have output operations executed right away. Chapter 30.

`\indent`  $\langle$ primitive command $\rangle$  Switch to horizontal mode and insert box with width `\parindent`. This command is automatically inserted before a  $\langle$ horizontal command $\rangle$  in vertical mode. Chapter 16.

`\input` $\langle$ file name $\rangle$   $\langle$ expandable command $\rangle$  Read a specified file as T<sub>E</sub>X input. Chapter 30.

`\inputlineno`  $\langle$ internal integer $\rangle$  (T<sub>E</sub>X3 only) Number of the current input line. Chapter 30.

`\insert` $\langle$ 8-bit number $\rangle$  $\{$  $\langle$ vertical mode material $\rangle$  $\}$   $\langle$ primitive command $\rangle$  Start an insertion item. Chapter 29.

`\insertpenalties`  $\langle$ special integer $\rangle$  Total of penalties for split insertions. Inside the output routine the number of held-over insertions. Chapter 29.

`\interlinepenalty`  $\langle$ integer parameter $\rangle$  Penalty for breaking a page between lines of a paragraph. Default 0 in plain T<sub>E</sub>X. Chapter 27.

`\jobname`  $\langle$ expandable command $\rangle$  Name of the main T<sub>E</sub>X file being processed. Chapter 32.

`\kern` $\langle$ dimen $\rangle$   $\langle$ kern $\rangle$  Add a kern item of the specified  $\langle$ dimen $\rangle$  to the list; this

can be used both in horizontal and vertical mode. Chapter 8.

`\language`  $\langle$ integer parameter $\rangle$  (T<sub>E</sub>X3 only) Choose a set of hyphenation patterns and exceptions. Chapter 19.

`\lastbox`  $\langle$ box $\rangle$  Register containing the last element added to the current list, if this was a box. Chapter 5.

`\lastkern`  $\langle$ internal dimen $\rangle$  If the last item on the list was a kern, the size of this. Chapter 8.

`\lastpenalty`  $\langle$ internal integer $\rangle$  If the last item on the list was a penalty, the value of this. Chapter 27.

`\lastskip`  $\langle$ internal glue $\rangle$  or  $\langle$ internal muglue $\rangle$ . If the last item on the list was a skip, the size of this. Chapter 8.

`\lccode` $\langle$ 8-bit number $\rangle$   $\langle$ internal integer $\rangle$ ; the control sequence itself is a  $\langle$ codename $\rangle$ . Access the character code that is the lowercase variant of a given code. Chapter 3.

`\leaders` $\langle$ box or rule $\rangle\langle$ vertical/horizontal/mathematical skip $\rangle$   $\langle$ leaders $\rangle$  Fill a specified amount of space with a rule or copies of box. Chapter 9.

`\left`  $\langle$ primitive command $\rangle$  Use the following character as an open delimiter. Chapter 21.

`\lefthyphenmin`  $\langle$ integer parameter $\rangle$  (T<sub>E</sub>X3 only) Minimum number of characters before a hyphenation. Chapter 19.

`\leftskip`  $\langle$ glue parameter $\rangle$  Glue that is placed to the left of all lines. Chapter 18.

`\leqno` $\langle$ math mode material $\rangle$  $\$$  $\$$   $\langle$ eqno $\rangle$  Place a left equation number in a display formula. Chapter 24.

`\let` $\langle$ control sequence $\rangle\langle$ equals $\rangle\langle$ token $\rangle$   $\langle$ let assignment $\rangle$  Define a control sequence to a token, assign its meaning if the token is a command or macro. Chapter 11.

`\limits`  $\langle$ primitive command $\rangle$  Place limits over and under a large operator. This is the default position in display style. Chapter 23.

`\linepenalty`  $\langle$ integer parameter $\rangle$  Penalty value associated with each line break. Default 10 in plain T<sub>E</sub>X. Chapter 19.

`\lineskip`  $\langle$ glue parameter $\rangle$  Glue added if distance between bottom and top of neighbouring boxes is less than `\lineskiplimit`. Default 1pt in plain T<sub>E</sub>X. Chapter 15.

`\lineskiplimit`  $\langle$ dimen parameter $\rangle$  Distance to be maintained between the

bottom and top of neighbouring boxes on a vertical list. Default 0pt in plain T<sub>E</sub>X. Chapter 15.

`\long`  $\langle$ prefix $\rangle$  Indicate that the arguments of the macro to be defined are allowed to contain `\par` tokens. Chapter 11.

`\looseness`  $\langle$ integer parameter $\rangle$  Number of lines by which this paragraph has to be made longer (or, if negative, shorter) than it would be ideally. Chapter 19.

`\lower` $\langle$ dimen $\rangle$  $\langle$ box $\rangle$   $\langle$ primitive command $\rangle$  Adjust vertical positioning of a box in horizontal mode. Chapter 5.

`\lowercase` $\langle$ general text $\rangle$   $\langle$ primitive command $\rangle$  Convert the argument to its lowercase form. Chapter 3.

`\mag`  $\langle$ integer parameter $\rangle$  1000 times the magnification of the document. Default 1000 in InT<sub>E</sub>X. Chapter 33.

`\mark` $\langle$ general text $\rangle$   $\langle$ primitive command $\rangle$  Specify a mark text. Chapter 28.

`\mathaccent` $\langle$ 15-bit number $\rangle$  $\langle$ math field $\rangle$   $\langle$ primitive command $\rangle$  Place an accent in math mode. Chapter 21,23.

`\mathbin` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as a binary operation. Chapter 23.

`\mathchar` $\langle$ 15-bit number $\rangle$   $\langle$ primitive command $\rangle$  Explicit denotation of a mathematical character. Chapter 21.

`\mathchardef` $\langle$ control sequence $\rangle$  $\langle$ equals $\rangle$  $\langle$ 15-bit number $\rangle$   $\langle$ shorthand definition $\rangle$  Define a control sequence to be a synonym for a math character code. Chapter 21.

`\mathchoice` $\{D\}\{T\}\{S\}\{SS\}$   $\langle$ primitive command $\rangle$  Give four variants of a formula for the four styles of math typesetting. Chapter 23.

`\mathclose` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as a closing symbol. Chapter 23.

`\mathcode` $\langle$ 8-bit number $\rangle$   $\langle$ internal integer $\rangle$ ; the control sequence itself is a  $\langle$ codename $\rangle$ . Code of a character determining its treatment in math mode. Chapter 21.

`\mathinner` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as an inner formula. Chapter 23.

`\mathop` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as a large operator. Chapter 23.

`\mathopen` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function

as an opening symbol. Chapter 23.

`\mathord` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as an ordinary object. Chapter 23.

`\mathpunct` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as a punctuation symbol. Chapter 23.

`\mathrel` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Let the following  $\langle$ math field $\rangle$  function as a relation. Chapter 23.

`\mathsurround`  $\langle$ dimen parameter $\rangle$  Kern amount placed before and after in-line formulas. Chapter 23.

`\maxdeadcycles`  $\langle$ integer parameter $\rangle$  The maximum number of times that the output routine is allowed to be called without a `\shipout` occurring. `InitEX` default: 25. Chapter 28.

`\maxdepth`  $\langle$ dimen parameter $\rangle$  Maximum depth of the page box. Default 4pt in plain `TEX`. Chapter 26.

`\meaning`  $\langle$ expandable command $\rangle$  Give the meaning of a control sequence as a string of characters. Chapter 34.

`\medmuskip`  $\langle$ muglue parameter $\rangle$  Medium amount of mu glue. Default value in plain `TEX`: 4mu plus 2mu minus 4mu Chapter 23.

`\message` $\langle$ general text $\rangle$   $\langle$ primitive command $\rangle$  Write a message to the terminal. Chapter 30.

`\mkern`  $\langle$ primitive command $\rangle$  Insert a kern measured in mu units. Chapter 23.

`\month`  $\langle$ integer parameter $\rangle$  The month of the current job. Chapter 33.

`\moveleft` $\langle$ dimen $\rangle$  $\langle$ box $\rangle$   $\langle$ primitive command $\rangle$  Adjust horizontal positioning of a box in vertical mode. Chapter 5.

`\moveright` $\langle$ dimen $\rangle$  $\langle$ box $\rangle$   $\langle$ primitive command $\rangle$  Adjust horizontal positioning of a box in vertical mode. Chapter 5.

`\mskip`  $\langle$ mathematical skip $\rangle$  Insert glue measured in mu units. Chapter 23.

`\multiply` $\langle$ numeric variable $\rangle$  $\langle$ optional by $\rangle$  $\langle$ number $\rangle$   $\langle$ arithmetic assignment $\rangle$  Arithmetic command to multiply a  $\langle$ numeric variable $\rangle$  (see `\advance`). Chapter 7.

`\muskip` $\langle$ 8-bit number $\rangle$   $\langle$ internal muglue $\rangle$ ; the control sequence itself is a  $\langle$ register prefix $\rangle$ . Access skips measured in mu units. Chapter 23.

`\muskipdef` $\langle$ control sequence $\rangle$  $\langle$ equals $\rangle$  $\langle$ 8-bit number $\rangle$   $\langle$ shorthand definition $\rangle$ ; the control sequence itself is a  $\langle$ registerdef $\rangle$ . Define a control sequence to be a synonym for a `\muskip` register. Chapter 23.

`\newlinechar`  $\langle$ integer parameter $\rangle$  Number of the character that triggers a new line in `\write` and `\message` statements. Plain T<sub>E</sub>X default -1; L<sup>A</sup>T<sub>E</sub>X default 10. Chapter 30.

`\noalign` $\langle$ filler $\rangle$  $\{$  $\langle$ vertical (horizontal) mode material $\rangle$  $\}$   $\langle$ primitive command $\rangle$  Specify vertical (horizontal) material to be placed in between rows (columns) of an `\halign` (`\valign`). Chapter 25.

`\noboundary`  $\langle$ horizontal command $\rangle$  (T<sub>E</sub>X3 only) Omit implicit boundary character. Chapter 4.

`\noexpand` $\langle$ token $\rangle$   $\langle$ expandable command $\rangle$  Do not expand the next token. Chapter 12.

`\noindent`  $\langle$ primitive command $\rangle$  Switch to horizontal mode with an empty horizontal list. Chapter 16.

`\nolimits`  $\langle$ primitive command $\rangle$  Place limits of a large operator as subscript and superscript expressions. This is the default position in text style. Chapter 23.

`\nonscript`  $\langle$ primitive command $\rangle$  Cancel the next glue item if it occurs in `scriptstyle` or `scriptscriptstyle`. Chapter 23.

`\nonstopmode`  $\langle$ interaction mode assignment $\rangle$  T<sub>E</sub>X fixes errors as best it can, and performs an emergency stop when user interaction is needed. Chapter 32.

`\nulldelimiterspace`  $\langle$ dimen parameter $\rangle$  Width taken for empty delimiters. Default 1.2pt in plain T<sub>E</sub>X. Chapter 21.

`\nullfont`  $\langle$ fontdef token $\rangle$  Name of an empty font that T<sub>E</sub>X uses in emergencies. Chapter 4.

`\number` $\langle$ number $\rangle$   $\langle$ expandable command $\rangle$  Convert a  $\langle$ number $\rangle$  to decimal representation. Chapter 7.

`\omit`  $\langle$ primitive command $\rangle$  Omit the template for one alignment entry. Chapter 25.

`\openin` $\langle$ 4-bit number $\rangle$  $\langle$ equals $\rangle$  $\langle$ filename $\rangle$   $\langle$ primitive command $\rangle$  Open a stream for input. Chapter 30.

`\openout` $\langle$ 4-bit number $\rangle$  $\langle$ equals $\rangle$  $\langle$ filename $\rangle$   $\langle$ primitive command $\rangle$  Open a stream for output. Chapter 30.

`\or`  $\langle$ primitive command $\rangle$  Separator for entries of an `\ifcase`. Chapter 13.

`\outer`  $\langle$ prefix $\rangle$  Indicate that the macro being defined should occur on the outer level only. Chapter 11.

`\output`  $\langle$ token parameter $\rangle$  Token list with instructions for shipping out



pages. Chapter 28.

`\outputpenalty`  $\langle$ integer parameter $\rangle$  Value of the penalty at the current page break, or 10 000 if the break was not at a penalty. Chapter 27,28.

`\over`  $\langle$ generalized fraction command $\rangle$  Fraction. Chapter 23.

`\overfullrule`  $\langle$ dimen parameter $\rangle$  Width of the rule that is printed to indicate overfull horizontal boxes. Plain  $\text{\TeX}$  default: 5pt. Chapter 5.

`\overline` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Overline the following  $\langle$ math field $\rangle$ . Chapter 23.

`\overwithdelims` $\langle$ delim $_1$  $\rangle$  $\langle$ delim $_2$  $\rangle$   $\langle$ generalized fraction command $\rangle$  Fraction with delimiters. Chapter 23.

`\pagedepth`  $\langle$ special dimen $\rangle$  Depth of the current page. Chapter 27.

`\pagefilllstretch`  $\langle$ special dimen $\rangle$  Accumulated third-order stretch of the current page. Chapter 27.

`\pagefillstretch`  $\langle$ special dimen $\rangle$  Accumulated second-order stretch of the current page. Chapter 27.

`\pagefilstretch`  $\langle$ special dimen $\rangle$  Accumulated first-order stretch of the current page. Chapter 27.

`\pagegoal`  $\langle$ special dimen $\rangle$  Goal height of the page box. This starts at `\vsize`, and is diminished by heights of insertion items. Chapter 27.

`\pageshrink`  $\langle$ special dimen $\rangle$  Accumulated shrink of the current page. Chapter 27.

`\pagestretch`  $\langle$ special dimen $\rangle$  Accumulated zeroth-order stretch of the current page. Chapter 27.

`\pagetotal`  $\langle$ special dimen $\rangle$  Accumulated natural height of the current page. Chapter 27.

`\par`  $\langle$ primitive command $\rangle$  Close off a paragraph and go into vertical mode. Chapter 17.

`\parfillskip`  $\langle$ glue parameter $\rangle$  Glue that is placed between the last element of the paragraph and the line end. Plain  $\text{\TeX}$  default: 0pt plus 1fil. Chapter 17.

`\parindent`  $\langle$ dimen parameter $\rangle$  Size of the indentation box added in front of a paragraph. Chapter 16,18.

`\parshape`  $\langle$ internal integer $\rangle$  Command for general paragraph shapes:

`\parshape` $\langle$ equals $\rangle n\ i_1\ \ell_1\ \dots\ i_n\ \ell_n$

specifies a number of lines  $n$ , and  $n$  pairs of an indentation and line length.

Chapter 18.

`\parskip`  $\langle$ glue parameter $\rangle$  Amount of glue added to vertical list when a paragraph starts; default value `0pt plus 1pt` in plain T<sub>E</sub>X. Chapter 16.

`\patterns` $\langle$ general text $\rangle$   $\langle$ hyphenation assignment $\rangle$  Define a list of hyphenation patterns for the current value of `\language`; allowed only in IniT<sub>E</sub>X. Chapter 19.

`\pausing`  $\langle$ integer parameter $\rangle$  Specify that T<sub>E</sub>X should pause after each line that is read from a file. Chapter 32.

`\penalty`  $\langle$ primitive command $\rangle$  Specify desirability of not breaking at this point. Chapter 19,27.

`\postdisplaypenalty`  $\langle$ integer parameter $\rangle$  Penalty placed in the vertical list below a display. Chapter 24.

`\predisplaypenalty`  $\langle$ integer parameter $\rangle$  Penalty placed in the vertical list above a display. Plain T<sub>E</sub>X default: 10 000. Chapter 24.

`\predisplaysize`  $\langle$ dimen parameter $\rangle$  Effective width of the line preceding the display. Chapter 24.

`\pretolerance`  $\langle$ integer parameter $\rangle$  Tolerance value for a paragraph that uses no hyphenation. Default 100 in plain T<sub>E</sub>X. Chapter 19.

`\prevdepth`  $\langle$ special dimen $\rangle$  Depth of the last box added to a vertical list as it is perceived by T<sub>E</sub>X. Chapter 15.

`\prevgraf`  $\langle$ special integer $\rangle$  The number of lines in the paragraph last added to the vertical list. Chapter 19.

`\radical` $\langle$ 24-bit number $\rangle$   $\langle$ primitive command $\rangle$  Command for setting things such as root signs. Chapter 21.

`\raise` $\langle$ dimen $\rangle$  $\langle$ box $\rangle$   $\langle$ primitive command $\rangle$  Adjust vertical positioning of a box in horizontal mode. Chapter 5.

`\read` $\langle$ number $\rangle$ `to` $\langle$ control sequence $\rangle$   $\langle$ simple assignment $\rangle$  Read a line from a stream into a control sequence. Chapter 30.

`\relax`  $\langle$ primitive command $\rangle$  Do nothing. Chapter 12.

`\relpenalty`  $\langle$ integer parameter $\rangle$  Penalty for breaking after a binary relation, not enclosed in a subformula. Plain T<sub>E</sub>X default: 500. Chapter 23.

`\right`  $\langle$ primitive command $\rangle$  Use the following character as a closing delimiter. Chapter 21.

`\righthyphenmin`  $\langle$ integer parameter $\rangle$  (T<sub>E</sub>X3 only) Minimum number of characters after a hyphenation. Chapter 19.

`\rightskip`  $\langle$ glue parameter $\rangle$  Glue that is placed to the right of all lines.

Chapter 18.

`\romannumeral` $\langle$ number $\rangle$   $\langle$ expandable command $\rangle$  Convert a positive  $\langle$ number $\rangle$  to lowercase roman representation. Chapter 7.

`\scriptfont` $\langle$ 4-bit number $\rangle$   $\langle$ family member $\rangle$ ; the control sequence itself is a  $\langle$ font range $\rangle$ . Access the scriptfont of a family. Chapter 22.

`\scriptscriptfont` $\langle$ 4-bit number $\rangle$   $\langle$ family member $\rangle$ ; the control sequence itself is a  $\langle$ font range $\rangle$ . Access the scriptscriptfont of a family. Chapter 22.

`\scriptscriptstyle`  $\langle$ primitive command $\rangle$  Select the scriptscript style of math typesetting. Chapter 23.

`\scriptspace`  $\langle$ dimen parameter $\rangle$  Extra space after subscripts and superscripts. Default .5pt in plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ . Chapter 23.

`\scriptstyle`  $\langle$ primitive command $\rangle$  Select the script style of math typesetting. Chapter 23.

`\scrollmode`  $\langle$ interaction mode assignment $\rangle$   $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  patches errors itself, but will ask the user for missing files. Chapter 32.

`\setbox` $\langle$ 8-bit number $\rangle$  $\langle$ equals $\rangle$  $\langle$ box $\rangle$   $\langle$ simple assignment $\rangle$  Assign a box to a box register. Chapter 5.

`\setlanguage` $\langle$ number $\rangle$   $\langle$ primitive command $\rangle$  ( $\mathrm{T}_{\mathrm{E}}\mathrm{X}3$  only) Insert a whatsit resetting the current language to the  $\langle$ number $\rangle$  specified. Chapter 19.

`\sfcode` $\langle$ 8-bit number $\rangle$   $\langle$ internal integer $\rangle$ ; the control sequence itself is a  $\langle$ codename $\rangle$ . Access the value of the `\spacefactor` associated with a character. Chapter 20.

`\shipout` $\langle$ box $\rangle$   $\langle$ primitive command $\rangle$  Ship a box to the dvi file. Chapter 28.

`\show` $\langle$ token $\rangle$   $\langle$ primitive command $\rangle$  Display the meaning of a token on the screen. Chapter 34.

`\showbox` $\langle$ 8-bit number $\rangle$   $\langle$ primitive command $\rangle$  Write the contents of a box to the log file. Chapter 34.

`\showboxbreadth`  $\langle$ integer parameter $\rangle$  Number of successive elements that are shown when `\tracingoutput` is positive, each time a level is visited. Plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  default: 5. Chapter 34.

`\showboxdepth`  $\langle$ integer parameter $\rangle$  The number of levels that are shown when `\tracingoutput` is positive. Plain  $\mathrm{T}_{\mathrm{E}}\mathrm{X}$  default: 3. Chapter 34.

`\showlists`  $\langle$ primitive command $\rangle$  Write to the log file the contents of the partial lists currently built in all modes. Chapter 6.

`\showthe``<internal quantity>` `<primitive command>` Display on the terminal the result of prefixing a token with `\the`. Chapter 34.

`\skewchar``<font>` `<internal integer>` Font position of an after-placed accent. Chapter 21.

`\skip``<8-bit number>` `<internal glue>`; the control sequence itself is a `<register prefix>`. Access skip registers Chapter 8.

`\skipdef``<control sequence>``<equals>``<8-bit number>` `<shorthand definition>`; the control sequence itself is a `<registerdef>`. Define a control sequence to be a synonym for a `\skip` register. Chapter 8.

`\spacefactor` `<special integer>` 1000 times the ratio by which the stretch component of the interword glue should be multiplied. Chapter 20.

`\spaceskip` `<glue parameter>` Interword glue if non-zero. Chapter 20.

`\span` `<primitive command>` Join two adjacent alignment entries, or (in preamble) expand the next token. Chapter 25.

`\special``<general text>` `<primitive command>` Write a token list to the dvi file. Chapter 33.

`\splitbotmark` `<expandable command>` The last mark on a split-off page. Chapter 28.

`\splitfirstmark` `<expandable command>` The first mark on a split-off page. Chapter 28.

`\splitmaxdepth` `<dimen parameter>` Maximum depth of a box split off by a `\vsplit` operation. Default 4pt in plain T<sub>E</sub>X. Chapter 5,26.

`\splittopskip` `<glue parameter>` Minimum distance between the top of what remains after a `\vsplit` operation, and the first item in that box. Default 10pt in plain T<sub>E</sub>X. Chapter 27.

`\string``<token>` `<expandable command>` Convert a token to a string of one or more characters. Chapter 3.

`\tabskip` `<glue parameter>` Amount of glue in between columns (rows) of an `\halign` (`\valign`). Chapter 25.

`\textfont``<4-bit number>` `<family member>`; the control sequence itself is a `<font range>`. Access the `\textfont` of a family. Chapter 22.

`\textstyle` `<primitive command>` Select the text style of math typesetting. Chapter 23.

`\the``<internal quantity>` `<primitive command>` Expand the value of various quantities in T<sub>E</sub>X into a string of (character) tokens. Chapter 12.

`\thickmuskip` *<muglue parameter>* Large amount of mu glue. Default value in plain  $\TeX$ : 5mu plus 5mu. Chapter 23.

`\thinmuskip` *<muglue parameter>* Small amount of mu glue. Default value in plain  $\TeX$ : 3mu. Chapter 23.

`\time` *<integer parameter>* Number of minutes after midnight that the current job started. Chapter 33.

`\toks`*<8-bit number>* *<register prefix>* Access a token list register. Chapter 14.

`\toksdef`*<control sequence>**<equals>**<8-bit number>* *<shorthand definition>*; the control sequence itself is a *<registerdef>*. Assign a control sequence to a `\toks` register. Chapter 14.

`\tolerance` *<integer parameter>* Tolerance value for lines in a paragraph that does use hyphenation. Default 200 in plain  $\TeX$ , 10 000 in  $\text{\texttt{Ini}\TeX}$ . Chapter 19.

`\topmark` *<expandable command>* The last mark of the previous page. Chapter 28.

`\topskip` *<glue parameter>* Minimum distance between the top of the page box and the baseline of the first box on the page. Default 10pt in plain  $\TeX$ . Chapter 26.

`\tracingcommands` *<integer parameter>* When this is 1,  $\TeX$  displays primitive commands executed; when this is 2 or more the outcome of conditionals is also recorded. Chapter 34.

`\tracinglostchars` *<integer parameter>* If this parameter is positive,  $\TeX$  gives diagnostic messages whenever a character is accessed that is not present in a font. Plain  $\TeX$  default: 1. Chapter 34.

`\tracingmacros` *<integer parameter>* If this is 1, the log file shows expansion of macros that are performed and the actual values of the arguments; if this is 2 or more *<token parameter>*s such as `\output` and `\everypar` are also traced. Chapter 34.

`\tracingonline` *<integer parameter>* If this parameter is positive,  $\TeX$  will write trace information also to the terminal. Chapter 34.

`\tracingoutput` *<integer parameter>* If this parameter is positive, the log file shows a dump of boxes that are shipped to the dvi file. Chapter 34.

`\tracingpages` *<integer parameter>* If this parameter is positive,  $\TeX$  generates a trace of the page-breaking algorithm. Chapter 34.

`\tracingparagraphs` *<integer parameter>* If this parameter is positive,  $\TeX$  generates a trace of the line-breaking algorithm. Chapter 34.

`\tracingrestores`  $\langle$ integer parameter $\rangle$  If this parameter is positive, T<sub>E</sub>X will report all values that are restored when a group level ends. Chapter 34.

`\tracingstats`  $\langle$ integer parameter $\rangle$  If this parameter is positive, T<sub>E</sub>X reports at the end of the job the usage of various internal arrays. Chapter 34.

`\uccode` $\langle$ 8-bit number $\rangle$   $\langle$ internal integer $\rangle$ ; the control sequence itself is a  $\langle$ codename $\rangle$ . Access the character code that is the uppercase variant of a given code. Chapter 3.

`\uchyph`  $\langle$ integer parameter $\rangle$  Positive if hyphenating words starting with a capital letter is allowed. Plain T<sub>E</sub>X default 1. Chapter 19.

`\underline` $\langle$ math field $\rangle$   $\langle$ math atom $\rangle$  Underline the following  $\langle$ math field $\rangle$ . Chapter 23.

`\unhbox` $\langle$ 8-bit number $\rangle$   $\langle$ horizontal command $\rangle$  Unpack a box register containing a horizontal box, appending the contents to the list, and emptying the register. Chapter 5.

`\unhcopy` $\langle$ 8-bit number $\rangle$   $\langle$ horizontal command $\rangle$  The same as `\unhbox`, but do not empty the register. Chapter 5.

`\unkern`  $\langle$ primitive command $\rangle$  Remove the last item of the list if this was a kern. Chapter 8.

`\unpenalty`  $\langle$ primitive command $\rangle$  Remove the last item of the list if this was a penalty. Chapter 27.

`\unskip`  $\langle$ primitive command $\rangle$  Remove the last item of the list if this was a skip. Chapter 8.

`\unvbox` $\langle$ 8-bit number $\rangle$   $\langle$ vertical command $\rangle$  Unpack a box register containing a vertical box, appending the contents to the list, and emptying the register. Chapter 5.

`\unvcopy` $\langle$ 8-bit number $\rangle$   $\langle$ vertical command $\rangle$  The same as `\unvbox`, but do not empty the register. Chapter 5.

`\uppercase` $\langle$ general text $\rangle$   $\langle$ primitive command $\rangle$  Convert the argument to its uppercase form. Chapter 3.

`\vadjust` $\langle$ filler $\rangle$  $\{ \langle$ vertical mode material $\rangle \}$   $\langle$ primitive command $\rangle$  Specify in horizontal mode material for the enclosing vertical list. Chapter 6.

`\valign` $\langle$ box specification $\rangle$  $\{ \langle$ alignment material $\rangle \}$   $\langle$ horizontal command $\rangle$  Vertical alignment. Chapter 25.

`\vbadness`  $\langle$ integer parameter $\rangle$  Threshold below which overfull and underfull vertical boxes are not shown. Plain T<sub>E</sub>X default: 1000. Chapter 5.

`\vbox<box specification>{<vertical material>} <primitive command>`

Construct a vertical box with reference point on the last item. Chapter 5.

`\vcenter<box specification>{<vertical material>} <primitive command>`

Construct a vertical box vertically centred on the math axis. Chapter 23.

`\vfil <vertical command>` Vertical skip equivalent to `\vskip 0cm plus 1fil`. Chapter 8.

`\vfill <vertical command>` Vertical skip equivalent to

`\vskip 0cm plus 1fill`. Chapter 8.

`\vfилneg <vertical command>` Vertical skip equivalent to

`\vskip 0cm minus 1fil`. Chapter 8.

`\vfuzz <dimen parameter>` Excess size that T<sub>E</sub>X tolerates before it considers a vertical box overfull. Plain T<sub>E</sub>X default: 0.1pt. Chapter 5.

`\voffset <dimen parameter>` Distance by which the page is shifted down from the reference point, which is one inch from the top of the page. Chapter 26.

`\vrule <horizontal command>` Rule that spreads in vertical direction.

Chapter 9.

`\vsize <dimen parameter>` Height of the page box. Chapter 5,26.

`\vskip<glue> <vertical command>` Insert in vertical mode a glue item.

Chapter 8.

`\vsplit<8-bit number>to<dimen> <primitive command>` Split off the top part of a vertical box. Chapter 5,27.

`\vss <vertical command>` Vertical skip equivalent to

`\vskip 0cm plus 1fil minus 1fil`. Chapter 8.

`\vtop<box specification>{<vertical material>} <primitive command>`

Construct a vertical box with reference point on the first item. Chapter 5.

`\wd<8-bit number> <internal dimen>;` the control sequence itself is a <box dimension>. Width of the box in a box register. Chapter 5.

`\widowpenalty <integer parameter>` Additional penalty for breaking a page before the last line of a paragraph. Default 150 in plain T<sub>E</sub>X. Chapter 27.

`\write<number><general text> <primitive command>` Generate a whatsit item containing a token list to be written to the terminal or to a file. Chapter 30.

`\xdef <def>` Synonym for `\global\edef`. Chapter 11.

`\xleaders <leaders>` As `\leaders`, but with box leaders any excess space is spread equally between the boxes. Chapter 9.

`\xspaceskip <glue parameter>` Interword glue if non-zero and

`\spacefactor`  $\geq 2000$ . Chapter [20](#).

`\year`  $\langle$ integer parameter $\rangle$  The year of the current job. Chapter [33](#).



## 第 38 章 编码表格

38.1 字符编码表

38.1.1 ASCII 字符编码

ASCII CHARACTER CODES

dec

CHAR

hex

oct

b7 b6 b5 BITS b4 b3 b2 b1	0 0 0 0	0 0 0 1	0 1 0 0	0 1 0 1	1 0 0 0	1 0 0 1	1 1 0 0	1 1 0 1
	CONTROL		SYMBOLS NUMBERS		UPPERCASE		LOWERCASE	
0 0 0 0	0 NUL	16 DLE	32 SP	48 0	64 @	80 P	96 ‘	112 p
0 0 0 1	1 SOH	17 DC1	33 !	49 1	65 A	81 Q	97 a	113 q
0 0 1 0	2 STX	18 DC2	34 ”	50 2	66 B	82 R	98 b	114 r
0 0 1 1	3 ETX	19 DC3	35 #	51 3	67 C	83 S	99 c	115 s
0 1 0 0	4 EOT	20 DC4	36 \$	52 4	68 D	84 T	100 d	116 t
0 1 0 1	5 ENQ	21 NAK	37 %	53 5	69 E	85 U	101 e	117 u
0 1 1 0	6 ACK	22 SYN	38 &	54 6	70 F	86 V	102 f	118 v
0 1 1 1	7 BEL	23 ETB	39 ’	55 7	71 G	87 W	103 g	119 w
1 0 0 0	8 BS	24 CAN	40 (	56 8	72 H	88 X	104 h	120 x
1 0 0 1	9 HT	25 EM	41 )	57 9	73 I	89 Y	105 i	121 y
1 0 1 0	10 LF	26 SUB	42 *	58 :	74 J	90 Z	106 j	122 z
1 0 1 1	11 VT	27 ESC	43 +	59 ;	75 K	91 [	107 k	123 {
1 1 0 0	12 FF	28 FS	44 ,	60 <	76 L	92 \	108 l	124
1 1 0 1	13 CR	29 GS	45 -	61 =	77 M	93 ]	109 m	125 }
1 1 1 0	14 SO	30 RS	46 .	62 >	78 N	94 ^	110 n	126 ~
1 1 1 1	15 SI	31 US	47 /	63 ?	79 O	95 -	111 o	127 DEL

38.1.2 T<sub>E</sub>X 字符编码

CHAR

dec

hex

oct

T<sub>E</sub>X CHARACTER CODES

b7 b6 b5 BITS b4 b3 b2 b1	0000	0001	0100	0101	1000	1001	1100	1101
	CONTROL		SYMBOLS NUMBERS		UPPERCASE		LOWERCASE	
0000	0 <sup>0</sup> ^@	16 <sup>0</sup> ^^P	32 <sup>0</sup> SP	48 <sup>0</sup> 0	64 <sup>0</sup> @	80 <sup>0</sup> P	96 <sup>0</sup> ‘	112 <sup>0</sup> p
0001	1 <sup>1</sup> ^^A	17 <sup>1</sup> ^^Q	20 <sup>1</sup> !	30 <sup>1</sup> 1	40 <sup>1</sup> A	50 <sup>1</sup> Q	60 <sup>1</sup> a	70 <sup>1</sup> q
0010	2 <sup>2</sup> ^^B	18 <sup>2</sup> ^^R	21 <sup>2</sup> ”	31 <sup>2</sup> 2	41 <sup>2</sup> B	51 <sup>2</sup> R	61 <sup>2</sup> b	71 <sup>2</sup> r
0011	3 <sup>3</sup> ^^C	19 <sup>3</sup> ^^S	22 <sup>3</sup> #	32 <sup>3</sup> 3	42 <sup>3</sup> C	52 <sup>3</sup> S	62 <sup>3</sup> c	72 <sup>3</sup> s
0100	4 <sup>4</sup> ^^D	20 <sup>4</sup> ^^T	23 <sup>4</sup> \$	33 <sup>4</sup> 4	43 <sup>4</sup> D	53 <sup>4</sup> T	63 <sup>4</sup> d	73 <sup>4</sup> t
0101	5 <sup>5</sup> ^^E	21 <sup>5</sup> ^^U	24 <sup>5</sup> %	34 <sup>5</sup> 5	44 <sup>5</sup> E	54 <sup>5</sup> U	64 <sup>5</sup> e	74 <sup>5</sup> u
0110	6 <sup>6</sup> ^^F	22 <sup>6</sup> ^^V	25 <sup>6</sup> &	35 <sup>6</sup> 6	45 <sup>6</sup> F	55 <sup>6</sup> V	65 <sup>6</sup> f	75 <sup>6</sup> v
0111	7 <sup>7</sup> ^^G	23 <sup>7</sup> ^^W	26 <sup>7</sup> ’	36 <sup>7</sup> 7	46 <sup>7</sup> G	56 <sup>7</sup> W	66 <sup>7</sup> g	76 <sup>7</sup> w
1000	8 <sup>8</sup> ^^H	24 <sup>8</sup> ^^X	27 <sup>8</sup> (	37 <sup>8</sup> 8	47 <sup>8</sup> H	57 <sup>8</sup> X	67 <sup>8</sup> h	77 <sup>8</sup> x
1001	9 <sup>9</sup> ^^I	25 <sup>9</sup> ^^Y	28 <sup>9</sup> )	38 <sup>9</sup> 9	48 <sup>9</sup> I	58 <sup>9</sup> Y	68 <sup>9</sup> i	78 <sup>9</sup> y
1010	10 <sup>A</sup> ^^J	26 <sup>1A</sup> ^^Z	29 <sup>2A</sup> *	39 <sup>3A</sup> :	49 <sup>4A</sup> J	59 <sup>5A</sup> Z	69 <sup>6A</sup> j	79 <sup>7A</sup> z
1011	11 <sup>B</sup> ^^K	27 <sup>1B</sup> ^^[	30 <sup>2B</sup> +	40 <sup>3B</sup> ;	50 <sup>4B</sup> K	60 <sup>5B</sup> [	70 <sup>6B</sup> k	80 <sup>7B</sup> {
1100	12 <sup>C</sup> ^^L	28 <sup>1C</sup> ^^\ 34	31 <sup>2C</sup> ,	41 <sup>3C</sup> <	51 <sup>4C</sup> L	61 <sup>5C</sup> \ 134	71 <sup>6C</sup> l	81 <sup>7C</sup>
1101	13 <sup>D</sup> ^^M	29 <sup>1D</sup> ^^] 35	32 <sup>2D</sup> —	42 <sup>3D</sup> =	52 <sup>4D</sup> M	62 <sup>5D</sup> ]	72 <sup>6D</sup> m	82 <sup>7D</sup> }
1110	14 <sup>E</sup> ^^N	30 <sup>1E</sup> ^^ 36	33 <sup>2E</sup> .	43 <sup>3E</sup> >	53 <sup>4E</sup> N	63 <sup>5E</sup> ^	73 <sup>6E</sup> n	83 <sup>7E</sup> ~
1111	15 <sup>F</sup> ^^O	31 <sup>1F</sup> ^^ 37	34 <sup>2F</sup> /	44 <sup>3F</sup> ?	54 <sup>4F</sup> O	64 <sup>5F</sup> - 137	74 <sup>6F</sup> o	84 <sup>7F</sup> ^^? 177

38.2 计算机现代字体

38.2.1 计算机现代罗马字体

COMPUTER MODERN ROMAN FONT LAYOUT

0 Γ 0	16 l 10 20	32 ˘ 20 40	48 0 30 60	64 @ 40 100	80 P 50 120	96 ‘ 60 140	112 p 70 160
1 Δ 1	17 J 11 21	33 ! 21 41	49 1 31 61	65 A 41 101	81 Q 51 121	97 a 61 141	113 q 71 161
2 Θ 2	18 ˙ 12 22	34 ” 22 42	50 2 32 62	66 B 42 102	82 R 52 122	98 b 62 142	114 r 72 162
3 Λ 3	19 ˘ 13 23	35 # 23 43	51 3 33 63	67 C 43 103	83 S 53 123	99 c 63 143	115 s 73 163
4 Ξ 4	20 ˘ 14 24	36 \$ 24 44	52 4 34 64	68 D 44 104	84 T 54 124	100 d 64 144	116 t 74 164
5 Π 5	21 ˘ 15 25	37 % 25 45	53 5 35 65	69 E 45 105	85 U 55 125	101 e 65 145	117 u 75 165
6 Σ 6	22 - 16 26	38 & 26 46	54 6 36 66	70 F 46 106	86 V 56 126	102 f 66 146	118 v 76 166
7 Υ 7	23 ° 17 27	39 , 27 47	55 7 37 67	71 G 47 107	87 W 57 127	103 g 67 147	119 w 77 167
8 Φ 10	24 ˘ 18 30	40 ( 28 50	56 8 38 70	72 H 48 110	88 X 58 130	104 h 68 150	120 x 78 170
9 Ψ 11	25 ß 19 31	41 ) 29 51	57 9 39 71	73 I 49 111	89 Y 59 131	105 i 69 151	121 y 79 171
10 Ω 12	26 æ 1A 32	42 * 2A 52	58 : 3A 72	74 J 4A 112	90 Z 5A 132	106 j 6A 152	122 z 7A 172
11 ff 13	27 œ 1B 33	43 + 2B 53	59 ; 3B 73	75 K 4B 113	91 [ 5B 133	107 k 6B 153	123 — 7B 173
12 fi 14	28 ø 1C 34	44 , 2C 54	60 i 3C 74	76 L 4C 114	92 “ 5C 134	108 l 6C 154	124 — 7C 174
13 fl 15	29 Æ 1D 35	45 - 2D 55	61 = 3D 75	77 M 4D 115	93 ] 5D 135	109 m 6D 155	125 ” 7D 175
14 ffi 16	30 Œ 1E 36	46 · 2E 56	62 ¿ 3E 76	78 N 4E 116	94 ^ 5E 136	110 n 6E 156	126 ~ 7E 176
15 ffl 17	31 Ø 1F 37	47 / 2F 57	63 ? 3F 77	79 O 4F 117	95 · 5F 137	111 o 6F 157	127 .. 7F 177

38.2.2 计算机现代打字机字体

COMPUTER MODERN TYPEWRITER FONT LAYOUT

0	16	32	48	64	80	96	112
0	0	10	20	30	40	50	60
1	17	33	49	65	81	97	113
1	1	11	21	31	41	51	61
2	18	34	50	66	82	98	114
2	2	12	22	32	42	52	62
3	19	35	51	67	83	99	115
3	3	13	23	33	43	53	63
4	20	36	52	68	84	100	116
4	4	14	24	34	44	54	64
5	21	37	53	69	85	101	117
5	5	15	25	35	45	55	65
6	22	38	54	70	86	102	118
6	6	16	26	36	46	56	66
7	23	39	55	71	87	103	119
7	7	17	27	37	47	57	67
8	24	40	56	72	88	104	120
8	10	18	28	38	48	58	68
9	25	41	57	73	89	105	121
9	11	19	29	39	49	59	69
10	26	42	58	74	90	106	122
A	12	1A	2A	3A	4A	5A	6A
11	27	43	59	75	91	107	123
B	13	1B	2B	3B	4B	5B	6B
12	28	44	60	76	92	108	124
C	14	1C	2C	3C	4C	5C	6C
13	29	45	61	77	93	109	125
D	15	1D	2D	3D	4D	5D	6D
14	30	46	62	78	94	110	126
E	16	1E	2E	3E	4E	5E	6E
15	31	47	63	79	95	111	127
F	17	1F	2F	3F	4F	5F	6F

38.2.3 计算机现代意大利字体

COMPUTER MODERN ITALIC FONT LAYOUT

0	<i>Γ</i>	16	<i>ı</i>	32	<i>˘</i>	48	<i>ø</i>	64	<i>@</i>	80	<i>P</i>	96	<i>‘</i>	112	<i>p</i>
0	0	10	20	20	40	30	60	40	100	50	120	60	140	70	160
1	<i>Δ</i>	17	<i>j</i>	33	<i>!</i>	49	<i>1</i>	65	<i>A</i>	81	<i>Q</i>	97	<i>a</i>	113	<i>q</i>
1	1	11	21	21	41	31	61	41	101	51	121	61	141	71	161
2	<i>Θ</i>	18	<i>˙</i>	34	<i>”</i>	50	<i>2</i>	66	<i>B</i>	82	<i>R</i>	98	<i>b</i>	114	<i>r</i>
2	2	12	22	22	42	32	62	42	102	52	122	62	142	72	162
3	<i>Λ</i>	19	<i>˘</i>	35	<i>#</i>	51	<i>3</i>	67	<i>C</i>	83	<i>S</i>	99	<i>c</i>	115	<i>s</i>
3	3	13	23	23	43	33	63	43	103	53	123	63	143	73	163
4	<i>Ξ</i>	20	<i>˘</i>	36	<i>£</i>	52	<i>4</i>	68	<i>D</i>	84	<i>T</i>	100	<i>d</i>	116	<i>t</i>
4	4	14	24	24	44	34	64	44	104	54	124	64	144	74	164
5	<i>Π</i>	21	<i>˘</i>	37	<i>%</i>	53	<i>5</i>	69	<i>E</i>	85	<i>U</i>	101	<i>e</i>	117	<i>u</i>
5	5	15	25	25	45	35	65	45	105	55	125	65	145	75	165
6	<i>Σ</i>	22	<i>˘</i>	38	<i>£</i>	54	<i>6</i>	70	<i>F</i>	86	<i>V</i>	102	<i>f</i>	118	<i>v</i>
6	6	16	26	26	46	36	66	46	106	56	126	66	146	76	166
7	<i>Υ</i>	23	<i>˘</i>	39	<i>’</i>	55	<i>γ</i>	71	<i>G</i>	87	<i>W</i>	103	<i>g</i>	119	<i>w</i>
7	7	17	27	27	47	37	67	47	107	57	127	67	147	77	167
8	<i>Φ</i>	24	<i>˘</i>	40	<i>(</i>	56	<i>8</i>	72	<i>H</i>	88	<i>X</i>	104	<i>h</i>	120	<i>x</i>
8	8	18	30	28	50	38	70	48	110	58	130	68	150	78	170
9	<i>Ψ</i>	25	<i>β</i>	41	<i>)</i>	57	<i>9</i>	73	<i>I</i>	89	<i>Y</i>	105	<i>i</i>	121	<i>y</i>
9	9	19	31	29	51	39	71	49	111	59	131	69	151	79	171
10	<i>Ω</i>	26	<i>æ</i>	42	<i>*</i>	58	<i>:</i>	74	<i>J</i>	90	<i>Z</i>	106	<i>j</i>	122	<i>z</i>
A	12	1A	32	2A	52	3A	72	4A	112	5A	132	6A	152	7A	172
11	<i>ff</i>	27	<i>æ</i>	43	<i>+</i>	59	<i>;</i>	75	<i>K</i>	91	<i>l</i>	107	<i>k</i>	123	<i>—</i>
B	13	1B	33	2B	53	3B	73	4B	113	5B	133	6B	153	7B	173
12	<i>fi</i>	28	<i>ø</i>	44	<i>,</i>	60	<i>i</i>	76	<i>L</i>	92	<i>“</i>	108	<i>l</i>	124	<i>—</i>
C	14	1C	34	2C	54	3C	74	4C	114	5C	134	6C	154	7C	174
13	<i>fl</i>	29	<i>Æ</i>	45	<i>-</i>	61	<i>=</i>	77	<i>M</i>	93	<i>]</i>	109	<i>m</i>	125	<i>”</i>
D	15	1D	35	2D	55	3D	75	4D	115	5D	135	6D	155	7D	175
14	<i>ffi</i>	30	<i>Œ</i>	46	<i>·</i>	62	<i>¿</i>	78	<i>N</i>	94	<i>^</i>	110	<i>n</i>	126	<i>~</i>
E	16	1E	36	2E	56	3E	76	4E	116	5E	136	6E	156	7E	176
15	<i>ffl</i>	31	<i>Ø</i>	47	<i>/</i>	63	<i>?</i>	79	<i>O</i>	95	<i>˙</i>	111	<i>o</i>	127	<i>ˆ</i>
F	17	1F	37	2F	57	3F	77	4F	117	5F	137	6F	157	7F	177

38.2.4 计算机现代符号字体

COMPUTER MODERN SYMBOL FONT

0	—	0	16	⋈	20	32	←	40	48	/	60	64	ℵ	100	80	ℙ	120	96	┌	140	112	√	160
0			10		20	30		40	30		60	40		100	50		120	60		140	70		160
1	·	1	17	≡	21	33	→	41	49	∞	61	65	ℳ	101	81	ℚ	121	97	┐	141	113	∏	161
2	×	2	18	⊆	22	34	↑	42	50	∈	62	66	ℬ	102	82	ℛ	122	98	└	142	114	▽	162
3	*	3	19	⊇	23	35	↓	43	51	∋	63	67	ℭ	103	83	ℳ	123	99	┘	143	115	∫	163
4	÷	4	20	≤	24	36	↔	44	52	△	64	68	ℰ	104	84	ℴ	124	100	┌	144	116	□	164
5	◇	5	21	≥	25	37	↗	45	53	▽	65	69	ℰ	105	85	ℴ	125	101	┐	145	117	□	165
6	±	6	22	≲	26	38	↘	46	54	/	66	70	ℱ	106	86	ℴ	126	102	{	146	118	⊆	166
7	⊕	7	23	≳	27	39	≈	47	55	⋈	67	71	ℱ	107	87	ℴ	127	103	}	147	119	⊇	167
8	⊕	10	24	≈	30	40	←	50	56	∇	70	72	ℋ	110	88	ℳ	130	104	<	150	120	§	170
9	⊖	11	25	≈	31	41	⇒	51	57	∃	71	73	ℐ	111	89	ℴ	131	105	>	151	121	†	171
10	⊗	12	26	⊂	32	42	↑	52	58	┐	72	74	ℐ	112	90	ℴ	132	106		152	122	‡	172
11	⊗	13	27	⊃	33	43	↓	53	59	∅	73	75	ℐ	113	91	ℴ	133	107		153	123	¶	173
12	⊙	14	28	⋈	34	44	↔	54	60	ℛ	74	76	ℒ	114	92	∩	134	108	↕	154	124	♣	174
13	○	15	29	⋈	35	45	↖	55	61	ℳ	75	77	ℳ	115	93	⊕	135	109	↕	155	125	◇	175
14	◦	16	30	↖	36	46	↙	56	62	⊤	76	78	ℳ	116	94	∧	136	110	↖	156	126	♥	176
15	●	17	31	↖	37	47	∝	57	63	⊥	77	79	ℴ	117	95	∇	137	111	↖	157	127	♠	177
F			1F		37	2F		57	3F		77	4F		117	5F		137	6F		157	7F		177

38.2.5 计算机现代数学扩展字体

COMPUTER MODERN MATH EXTENSION FONT

0	16	32	48	64	80	96	112
0 ( 0	10 ( 20	20 ( 40	30 / 60	40 \ 100	50 $\Sigma$ 120	60 $\Pi$ 140	70 $\checkmark$ 160
1 ) 1	11 ) 21	21 ) 41	31 \ 61	41 / 101	51 $\Pi$ 121	61 $\Pi$ 141	71 $\checkmark$ 161
2 [ 2	12 ( 22	22 [ 42	32 [ 62	42   102	52 $\int$ 122	62 $\wedge$ 142	72 $\checkmark$ 162
3 ] 3	13 ) 23	23 ] 43	33   63	43   103	53 $\cup$ 123	63 $\wedge$ 143	73 $\checkmark$ 163
4 L 4	14 [ 24	24 [ 44	34 L 64	44 $\langle$ 104	54 $\cap$ 124	64 $\wedge$ 144	74 $\checkmark$ 164
5 J 5	15 ] 25	25 ] 45	35 J 65	45 $\rangle$ 105	55 $\oplus$ 125	65 $\sim$ 145	75   165
6 [ 6	16 L 26	26 [ 46	36   66	46 $\sqcup$ 106	56 $\wedge$ 126	66 $\sim$ 146	76 $\lceil$ 166
7 ] 7	17 J 27	27 ] 47	37   67	47 $\sqcup$ 107	57 $\vee$ 127	67 $\sim$ 147	77 $\parallel$ 167
8 { 10	18 [ 30	28 { 50	38 / 70	48 $\oint$ 110	58 $\Sigma$ 130	68 [ 150	78 $\uparrow$ 170
9 } 11	19 ] 31	29 } 51	39 \ 71	49 $\oint$ 111	59 $\Pi$ 131	69 ] 151	79 $\downarrow$ 171
A $\langle$ 12	1A { 32	2A $\langle$ 52	3A \ 72	4A $\odot$ 112	5A $\int$ 132	6A L 152	7A $\curvearrowright$ 172
B $\rangle$ 13	1B } 33	2B $\rangle$ 53	3B / 73	4B $\odot$ 113	5B $\cup$ 133	6B ] 153	7B $\curvearrowleft$ 173
C   14	1C $\langle$ 34	2C / 54	3C } 74	4C $\oplus$ 114	5C $\cap$ 134	6C [ 154	7C $\curvearrowright$ 174
D $\parallel$ 15	1D $\rangle$ 35	2D \ 55	3D } 75	4D $\oplus$ 115	5D $\oplus$ 135	6D ] 155	7D $\curvearrowleft$ 175
E / 16	1E / 36	2E / 56	3E ' 76	4E $\otimes$ 116	5E $\wedge$ 136	6E { 156	7E $\uparrow$ 176
F \ 17	1F \ 37	2F \ 57	3F   77	4F $\otimes$ 117	5F $\vee$ 137	6F } 157	7F $\downarrow$ 177



### 38.3 Plain T<sub>E</sub>X 数学符号

#### 38.3.1 字符的数学码

下面列出的字符是用这个赋值定义的：

```
\mathcode⟨8-bit number⟩⟨equals⟩⟨15-bit number⟩
```

Character	\mathcode	Class	Family	Hex position
.	"013A	ordinary	1	3A
/	"013D			3D
\	"026E		2	6E
	"026A			6A
+	"202B	binary operation	0	2B
-	"2200		2	00
*	"2203			03
:	"303A	relation	0	3A
=	"303D			3D
<	"313C		1	3C
>	"313E			3E
(	"4028	open symbol	0	28
[	"405B			5B
{	"4266		2	66
!	"5021	closing symbol	0	21
)	"5029			29
?	"503F			3F
]	"505D			5D
}	"5267		2	67
;	"603B	punctuation	0	3B
,	"613B		1	3B
	"8000			
'	"8000			
-	"8000			

#### 38.3.2 字符的定界码

下面列出的字符是用这个赋值定义的：

```
\delcode⟨8-bit number⟩⟨equals⟩⟨24-bit number⟩
```

它们可以与 \left 和 \right 一起使用。

Character	\delcode	small variant		large variant	
		Family	Hex position	Family	Hex position
(	"028300	0	28	3	00
)	"029301	0	29	3	01
[	"05B302	0	5B	3	02
]	"05D303	0	5D	3	03
<	"26830A	2	68	3	0A
>	"26930B	2	69	3	0B
/	"02F30E	0	2F	3	0E
	"26A30C	2	6A	3	0C
\	"26E30F	2	6E	3	0F

38.3.3 定义普通符号

下面列出的字符是用这个赋值定义的：

`\mathchardef⟨control sequence⟩⟨equals⟩⟨15-bit number⟩`

Symbol	Control Sequence	\mathcode	Family	Hex position
$\partial$	<code>\partial</code>	"0140	1	40
$\flat$	<code>\flat</code>	"015B		5B
$\natural$	<code>\natural</code>	"015C		5C
$\sharp$	<code>\sharp</code>	"015D		5D
$\ell$	<code>\ell</code>	"0160		60
$\imath$	<code>\imath</code>	"017B		7B
$\jmath$	<code>\jmath</code>	"017C		7C
$\wp$	<code>\wp</code>	"017D		7D
$\prime$	<code>\prime</code>	"0230	2	30
$\infty$	<code>\infty</code>	"0231		31
$\triangle$	<code>\triangle</code>	"0234		34
$\forall$	<code>\forall</code>	"0238		38
$\exists$	<code>\exists</code>	"0239		39
$\neg$	<code>\neg</code>	"023A		3A
$\emptyset$	<code>\emptyset</code>	"023B		3B
$\Re$	<code>\Re</code>	"023C		3C
$\Im$	<code>\Im</code>	"023D		3D
$\top$	<code>\top</code>	"023E		3E
$\bot$	<code>\bot</code>	"023F		3F
$\aleph$	<code>\aleph</code>	"0240		40

$\nabla$	<code>\nabla</code>	"0272	72
$\clubsuit$	<code>\clubsuit</code>	"027C	7C
$\diamond$	<code>\diamondsuit</code>	"027D	7D
$\heartsuit$	<code>\heartsuit</code>	"027E	7E
$\spadesuit$	<code>\spadesuit</code>	"027F	7F

38.3.4 定义巨算符

下面列出的字符是用这个赋值定义的：

```
\mathchardef<control sequence>(equals)<15-bit number>
```

Symbol	Control Sequence	<code>\mathcode</code>	Family	Hex position
$\int\!\!\int$	<code>\smallint</code>	"1273	2	73
$\sqcup\!\!\sqcup$	<code>\bigsqcup</code>	"1346	3	46
$\oint\!\!\oint$	<code>\ointop</code>	"1348		48
$\odot\!\!\odot$	<code>\bigodot</code>	"134A		4A
$\oplus\!\!\oplus$	<code>\bigoplus</code>	"134C		4C
$\otimes\!\!\otimes$	<code>\bigotimes</code>	"134E		4E
$\sum\!\!\sum$	<code>\sum</code>	"1350		50
$\prod\!\!\prod$	<code>\prod</code>	"1351		51
$\int\!\!\int\!\!\int$	<code>\intop</code>	"1352		52
$\cup\!\!\cup$	<code>\bigcup</code>	"1353		53
$\cap\!\!\cap$	<code>\bigcap</code>	"1354		54
$\uplus\!\!\uplus$	<code>\biguplus</code>	"1355		55
$\wedge\!\!\wedge$	<code>\bigwedge</code>	"1356		56
$\vee\!\!\vee$	<code>\bigvee</code>	"1357		57
$\coprod\!\!\coprod$	<code>\coprod</code>	"1360		60

38.3.5 定义二元运算符

下面列出的字符是用这个赋值定义的：

```
\mathchardef<control sequence>(equals)<15-bit number>
```

Symbol	Control Sequence	<code>\mathcode</code>	Family	Hex position
$\triangleright$	<code>\trianglerightright</code>	"212E	1	2E
$\triangleleft$	<code>\triangleleftleft</code>	"212F		2F

*	\star	"213F		3F
·	\cdot	"2201	2	01
×	\times	"2202		02
*	\ast	"2203		03
÷	\div	"2204		04
◇	\diamond	"2205		05
±	\pm	"2206		06
∓	\mp	"2207		07
⊕	\oplus	"2208		08
⊖	\ominus	"2209		09
⊗	\otimes	"220A		0A
⊘	\oslash	"220B		0B
⊙	\odot	"220C		0C
◯	\bigcirc	"220D		0D
◦	\circ	"220E		0E
•	\bullet	"220F		0F
△	\bigtriangleup	"2234		34
▽	\bigtriangledown	"2235		35
∪	\cup	"225B		5B
∩	\cap	"225C		5C
⊕	\uplus	"225D		5D
∧	\wedge	"225E		5E
∨	\vee	"225F		5F
\	\setminus	"226E		6E
ℳ	\mathcal M	"226F		6F
ℐ	\mathcal I	"2271		71
⊔	\sqcup	"2274		74
⊓	\sqcap	"2275		75
†	\dagger	"2279		79
‡	\ddagger	"227A		7A

38.3.6 定义二元关系符

下面列出的字符是用这个赋值定义的：

`\mathchardef⟨control sequence⟩⟨equals⟩⟨15-bit number⟩`

Symbol	Control Sequence	\mathcode	Family	Hex position
↵	\leftharpoonup	"3128	1	28
↶	\leftharpoondown	"3129		29

$\rightarrow$	<code>\rightharpoonup</code>	"312A	2A
$\rightarrow$	<code>\rightharpoondown</code>	"312B	2B
$($	<code>\smile</code>	"315E	5E
$)$	<code>\frown</code>	"315F	5F
$\asymp$	<code>\asymp</code>	"3210	2 10
$\equiv$	<code>\equiv</code>	"3211	11
$\subseteq$	<code>\subseteq</code>	"3212	12
$\supseteq$	<code>\supseteq</code>	"3213	13
$\leq$	<code>\leq</code>	"3214	14
$\geq$	<code>\geq</code>	"3215	15
$\preceq$	<code>\preceq</code>	"3216	16
$\succeq$	<code>\succeq</code>	"3217	17
$\sim$	<code>\sim</code>	"3218	18
$\approx$	<code>\approx</code>	"3219	19
$\subset$	<code>\subset</code>	"321A	1A
$\supset$	<code>\supset</code>	"321B	1B
$\ll$	<code>\ll</code>	"321C	1C
$\gg$	<code>\gg</code>	"321D	1D
$\prec$	<code>\prec</code>	"321E	1E
$\succ$	<code>\succ</code>	"321F	1F
$\leftarrow$	<code>\leftarrow</code>	"3220	20
$\rightarrow$	<code>\rightarrow</code>	"3221	21
$\leftrightarrow$	<code>\leftrightarrow</code>	"3224	24
$\nearrow$	<code>\nearrow</code>	"3225	25
$\searrow$	<code>\searrow</code>	"3226	26
$\simeq$	<code>\simeq</code>	"3227	27
$\Leftarrow$	<code>\Leftarrow</code>	"3228	28
$\Rightarrow$	<code>\Rightarrow</code>	"3229	29
$\Leftrightarrow$	<code>\Leftrightarrow</code>	"322C	2C
$\nwarrow$	<code>\nwarrow</code>	"322D	2D
$\swarrow$	<code>\swarrow</code>	"322E	2E
$\propto$	<code>\propto</code>	"322F	2F
$\in$	<code>\in</code>	"3232	32
$\ni$	<code>\ni</code>	"3233	33
$/$	<code>\not</code>	"3236	36
$\mapsto$	<code>\mapsto</code>	"3237	37
$\perp$	<code>\perp</code>	"323F	3F
$\vdash$	<code>\vdash</code>	"3260	60

$\dashv$	<code>\dashv</code>	"3261	61
$\mid$	<code>\mid</code>	"326A	6A
$\parallel$	<code>\parallel</code>	"326B	6B
$\sqsubseteq$	<code>\sqsubseteq</code>	"3276	76
$\sqsupseteq$	<code>\sqsupseteq</code>	"3277	77

38.3.7 定义定界符

下面列出的字符是用这个赋值定义的：

```
\def<control sequence>{\delimiter<27-bit number>}
```

Delimiters			
Symbol	Control Sequence	Hex code	Function
$\{$	<code>\lmoustache</code>	"4000340	open symbol
$\}$	<code>\rmoustache</code>	"5000341	closing symbol
$\{$	<code>\lgroup</code>	"400033A	open symbol
$\}$	<code>\rgroup</code>	"500033B	closing symbol
$\uparrow$	<code>\arrowvert</code>	"33C	ordinary
$\Uparrow$	<code>\Arrowvert</code>	"33D	ordinary
$\downarrow$	<code>\bracevert</code>	"33E	ordinary
$\Downarrow$	<code>\Vert</code>	"26B30D	ordinary
$\updownarrow$	<code>\vert</code>	"26A30C	ordinary
$\uparrow$	<code>\uparrow</code>	"3222378	relation
$\downarrow$	<code>\downarrow</code>	"3223379	relation
$\updownarrow$	<code>\updownarrow</code>	"326C33F	relation
$\Uparrow$	<code>\Uparrow</code>	"322A37E	relation
$\Downarrow$	<code>\Downarrow</code>	"322B37F	relation
$\updownarrow$	<code>\Updownarrow</code>	"326D377	relation
$\backslash$	<code>\backslash</code>	"26E30F	ordinary
$\rangle$	<code>\rangle</code>	"526930B	closing symbol
$\langle$	<code>\langle</code>	"426830A	open symbol
$\}$	<code>\rbrace</code>	"5267309	closing symbol
$\{$	<code>\lbrace</code>	"4266308	open symbol
$\lceil$	<code>\rceil</code>	"5265307	closing symbol
$\rfloor$	<code>\lceil</code>	"4264306	open symbol
$\rfloor$	<code>\rfloor</code>	"5263305	closing symbol
$\lfloor$	<code>\lfloor</code>	"4262304	open symbol

# 索引

- ^^ replacement, 30
- tfm files, 53
- dvi file, 288
- ~, 204
- accents, 45
- accents in math mode, 212
- alignment tab, 238
- alignments, 236–245
  - rules in, 243–244
- arithmetic, 89
  - on glue, *see* glue, arithmetic on
- assignment
  - box size, 65
  - font, 54
  - global, 54, 65, 114
  - local, 114
- badness, 101
  - and line breaking, 191
  - calculation, 101
- box
  - bounding, 55
- boxes, 57–74
- boxes
  - text in, 70
- braces, 115–116
- breakpoints
  - computation of, 252–253
- breakpoints in math lists, 226
- category
  - 0, 27, 29, 137
  - 1, 27, 68, 114, 118
  - 2, 27, 68, 114, 115, 118
  - 3, 27, 219, 230
  - 4, 27, 238
  - 5, 27, 37
  - 6, 28, 119
  - 7, 28, 31, 220
  - 8, 28, 220
  - 9, 28
  - 10, 28, 29, 31, 36, 50, 53, 122, 141
  - 11, 28, 29, 31, 45, 85, 154
  - 12, 28, 36, 45, 50, 53, 85, 88, 90, 98, 122, 141, 152, 154, 297
  - 13, 28, 125, 307
  - 14, 28
  - 15, 28, 30
  - 16, 29, 46, 151
- character
  - active, and \noexpand, 138
  - codes, 41
  - extendable, 210
  - hyphen, 195
  - implicit, 44
  - parameter, 122
- code
  - lowercase, *see* lowercase, code
  - uppercase, *see* uppercase, code
- codenames, 48

- commands
  - horizontal, 77
  - vertical, 77
- Computer Modern, 292
- conditionals, 149
  - evaluation of, 155
- cramped styles, 219
- date, 290
- delimiter
  - size, 210–211
- delimiter code, 209
- delimiters, 209–211
- demerits, 192
- device driver, 290
- device drivers, 291
- discardable items, 76
- discretionary hyphen, 195
- discretionary item, 195
- display alignment, 237
- display math, 230
- displays
  - non-centred, 234
- equation numbering, 233–234
- error patching, 301
- escape
  - character, *see* character, escape
- expansion, 132
  - expandable constructs, 133
- extension font, 228
- files, 270
- fixed-point
  - arithmetic, 90
- floating-point
  - arithmetic, 90
- font
  - dimensions, 54
- font families, 213
- font files, 291–292
- font metrics, 290
- font tables, 334
- fonts, 51
- format file, 284
- formula
  - axis of, 223
  - centring of, 223
- frenchspacing, 205
- generalized fractions, 225
- glue, 92
  - arithmetic on, 96
  - setting, 101
  - shrink component of, 99
  - stretch component of, 99
- horizontal alignment, 237
- hyphenation, 196
- I/O
  - asynchronous, 273
  - file, 270–276
  - screen, 273
- indentation
  - hanging, 184–185
- Init<sub>E</sub>X, 283
- input files, 270
- insertions, 264
- integer, 83
- italic correction, 55
- job, 280–281
- kerning, 55
- keywords, 307
- language, 199
  - current, 199
- languages, 196
- L<sub>A</sub>T<sub>E</sub>X, 287



- leaders, 107
  - rule, 108
- ligatures, 56
- line
  - end, 26
  - input, 26
  - width, 184
- line breaking, 189
  - badness, 191
- list
  - horizontal, 76
  - vertical, 76
- lists
  - horizontal
    - breakpoints in, 191
- log file, 281
- Lollipop, 288
- lowercase
  - code, 47
- machine dependence, 232
- machine independence, 26
- magnification, 289
- marks, 259
- math characters, 208–209
- math classes, 221
- math mode, 219
  - display, 219
  - non-display, 219
- math shift character, 219
- math spacing, 223–225
- math styles, 219
- math symbols, lists of, 339
- math units, 223
- migrating material, 80
- mode, 75
  - horizontal, 76
  - internal vertical, 78
  - restricted horizontal, 78
  - vertical, 76
- mu glue, 223
- number
  - conversion, 88
- numbers, 83
- output routine, 257–263
- overflow errors, 302–305
- page
  - breaking, 249–256
  - depth, 247–248
  - height, 247–248
  - length, 250–251
  - numbering, 261
- page positioning, 246
- paragraph
  - breaking into lines, 189
  - end, 179–182
  - shape, 183–188
  - start, 173–178
- Pascal, 293
- penalties in math mode, 226
- point
  - scaled, 87
- PostScript, 290
- prefixes
  - macro, 118
- quad, 223
- radical, 211
- registers
  - allocation of, 278–279
- roman numerals, 88
- rules, 105
- run modes, 282
- shrink, 99
- slant

- per point, 46
- space
  - control, 203
  - factor, 202
  - optional, 307
- spacefactor code, 204
- spacing, 201
- specials, 290
- statistics, 295
- streams, 271
- stretch, 99
- subscript, 220
- successor, 211
- superscript, 220
- symbol font, 227–228
- table, character codes, 333
- table, ascii, 332
- tables, 236
- tables, font, 334
- $\text{\TeX}$ , 283
- $\text{\TeX}$ , big, 303
- $\text{\TeX}$ , version 2, 309
- $\text{\TeX}$ , version 3, 67, 198, 268, 309
- tie, 204
- time, 290
- token
  - list, 163
- tracing, 295–299
- TUG, 293
- TUGboat, 293
- units of measurement, 97
- uppercase
  - code, 47
- vertical alignment, 237
- Vir $\text{\TeX}$ , 283
- virtual fonts, 291
- web, 293
- whatsits, 273
- 保存堆栈, 113
- 原始命令, 125
- 参数, 119
  - 字符, 122
  - 定界的, 120
  - 非定界, 120
- 参量, 119
- 命令
  - 原始的, 125
- 基线
  - 距离, 168
- 备选内容, 250
- 字符, 32
  - 空格符, 34
  - 转义符, 31, 33
- 宏, 117
  - 外部的, 118
  - 定义, 118
- 当前页面, 250
- 惩罚
  - 竖直模式中, 251
- 抄录模式, 131
- 控制
  - 控制序列, 29
  - 控制空格, 29
- 活动字符, 125
- 状态
  - 内部, 29
- 空格
  - 可选空格, 34
  - 滑稽空格, 36
  - 空格记号, 36
- 竖直模式的断点, 251
- 类别码, 27

粘连

行间, [168](#)

编组, [113](#)

定界符, [114](#)

花括号

显式, [114](#)

隐式, [114](#)

行

空白行, [33](#)

终止符, [37](#)

记号

空格记号, [34](#)

输入

栈, [126](#)

递归, [126](#)

页面

构建器, [250](#)

## 参考文献

- [1] W. Appelt. *T<sub>E</sub>X fr Fortgeschrittene*. Addison-Wesley Verlag, 1988. [302](#)
- [2] B. Beeton. Controlling <ctrl-M>; ruling the depths. *TUGboat*, 9:182–183, 1988. [37](#)
- [3] B. Beeton. Additional font and glyph attributes for processing of mathematics, 1991. document N1174 Rev., of ISO/IEC JTC1/SC18/WG8. [224](#)
- [4] K. Berry. Eplain. *TUGboat*, 11:571–572, 1990. [284](#)
- [5] J. Braams. Babel, a language option for L<sup>A</sup>T<sub>E</sub>X. *TUGboat*, 12:291–301, 1991. [194](#)
- [6] J. Braams, V. Eijkhout, and N.A.F.M. Poppelier. The development of national L<sup>A</sup>T<sub>E</sub>X styles. *TUGboat*, 10:401–406, 1989. [284](#)
- [7] M.J. Downes. Line breaking in \unhboxed text. *TUGboat*, 11:605–612. [69](#)
- [8] V. Eijkhout. An indentation scheme. *TUGboat*, 11:613–616. [174](#)
- [9] V. Eijkhout. A paragraph skip scheme. *TUGboat*, 11:616–619. [175](#)
- [10] V. Eijkhout. Unusual paragraph shapes. *TUGboat*, 11:51–53. [72](#), [186](#)
- [11] V. Eijkhout. Oral T<sub>E</sub>X. *TUGboat*, 12:272–276, 1991. [146](#), [158](#)
- [12] V. Eijkhout and A. Lenstra. The document style designer as a separate entity. *TUGboat*, 12:31–34, 1991. [285](#)
- [13] D. Guenther. T<sub>E</sub>X T1 goes public domain. *TUGboat*, 11:54–55, 1990. [284](#)
- [14] *Hart's Rules for Compositors and Readers at the Oxford University Press*. Oxford University Press, 1983. 39th edition. [203](#)
- [15] A. Hendrikson. *MacroT<sub>E</sub>X, A T<sub>E</sub>X Macro Toolkit*. T<sub>E</sub>X\_nology Inc, 1991. [284](#)

- [16] A. Jeffrey. Lists in  $\text{T}_{\text{E}}\text{X}$ 's mouth. *TUGboat*, 11:237–245, 1990. [146](#)
- [17] D.E. Knuth. *Computer Modern Typefaces*. Addison-Wesley. [53](#)
- [18] D.E. Knuth. The errors of  $\text{T}_{\text{E}}\text{X}$ . *Software Practice and Experience*, 19:607–681. [305](#)
- [19] D.E. Knuth. Literate programming. *Computer J.*, 27:97–111. [290](#)
- [20] D.E. Knuth. The new versions of  $\text{T}_{\text{E}}\text{X}$  and Metafont. *TUGboat*, 10:325–327. [54](#), [305](#)
- [21] D.E. Knuth. A torture test for  $\text{T}_{\text{E}}\text{X}$ . Technical report, Stanford Computer Science Report 1027, Stanford, California. [88](#)
- [22] D.E. Knuth. Typesetting concrete mathematics. *TUGboat*, 10:31–36. [289](#)
- [23] D.E. Knuth. *T\_{\text{E}}\text{X}: the Program*. Addison-Wesley, 1986. [42](#), [52](#), [280](#), [285](#), [287](#)
- [24] D.E. Knuth. Virtual fonts: more fun for grand wizards. *TUGboat*, 11:13–23, 1990. [50](#), [288](#)
- [25] D.E. Knuth. *The T\_{\text{E}}\text{X} book*. Addison-Wesley, reprinted with corrections 1989. [36](#), [47](#), [298](#), [300](#)
- [26] D.E. Knuth and D.R. Fuchs.  $\text{T}_{\text{E}}\text{X}$  ware. Technical report, 1986. Stanford Computer Science report 86–1097. [287](#)
- [27] D.E. Knuth and M.F. Plass. Breaking paragraphs into lines. *Software practice and experience*, 11:1119–1184, 1981. [188](#), [190](#)
- [28] G. Kuiken. Additional hyphenation patterns. *TUGboat*, 11:24–25, 1990. [195](#)
- [29] L. Lamport. *L\_{\text{A}}\text{T}\_{\text{E}}\text{X}, a Document Preparation System*. Addison-Wesley, 1986. [284](#)
- [30] F.M. Liang. *Word hy-phen-a-tion by com-pu-ter*. PhD thesis, 1983. [194](#), [300](#)
- [31] S. Maus. Looking ahead for a  $\langle\text{box}\rangle$ . *TUGboat*, 11:612–613, 1990. [134](#)
- [32] S. Maus. An expansion power lemma. *TUGboat*, 12:277, 1991. [146](#)
- [33] F. Mittelbach and R. Schpf.  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}3$ . *TUGboat*, 12. [284](#)

- [34] F. Mittelbach and R. Schpf. With  $\LaTeX$  into the nineties. *TUGboat*, 10:681–690, 1989. [284](#)
- [35] E. Myers and F.E. Paige.  $\TeX$  sis –  $\TeX$  macros for physicists. Macros and manual available by anonymous ftp from lifshitz.ph.utexas.edu (128.83.131.57). [284](#)
- [36] H. Partl. German  $\TeX$ . *TUGboat*, 9:70–72, 1988. [194](#)
- [37] Z. Rubinstein. Printing annotated chess literature in natural notation. *TUGboat*, 10:387–390, 1989. [125](#)
- [38] D. Salomon. Output routines: Examples and techniques. part i: Introduction and examples. *TUGboat*, 11:69–85, 1990. [260](#)
- [39] D. Salomon. Output routines: Examples and techniques. part ii: OTR techniques. *TUGboat*, 11:212–236, 1990. [260](#)
- [40] D. Salomon. Output routines: Examples and techniques. part iii: Insertions. *TUGboat*, 11:588–605, 1990. [266](#)
- [41] W. Sewell. *Weaving a Program: Literate Programming in WEB*. Van Nostrand Reinhold, 1989. [290](#)
- [42] R. Southall. Designing a new typeface with metafont. In  *$\TeX$  for scientific documentation, Lecture Notes in Computer Science 236*. Springer Verlag, 1984. [50](#), [289](#)
- [43] M. Spivak. *The Joy of  $\TeX$* . American Mathematical Society, 1986. [284](#)
- [44] M. Spivak. *LAMST $\TeX$ , the Synthesis*. The  $\TeX$  plorators Corporation, 1989. [284](#)
- [45] K. Thull. The virtual memory management of public  $\TeX$ . *TUGboat*, 10:15–22, 1989. [299](#)
- [46] J. Tschichold. *Ausgewählte Aufsätze über Fragen der Gestalt des Buches und der Typographie*. Birkhäuser Verlag, 1975. [180](#)
- [47] P. Tutelaers. A font and a style for typesetting chess using  $\LaTeX$  or plain  $\TeX$ . *TUGboat*, 13, 1991. [125](#)
- [48] D.B. Updike. *Printing Types*. Harvard University Press, 1937. (reprinted 1980, New York NY: Dover Publications). [289](#)

- 
- [49] S. von Bechtolsheim. A tutorial on \futurelet. *TUGboat*, 9:276–278, 1988. [127](#)
- [50] M. Vox. *Caractère*, 1955. [289](#)
- [51] M. Weinstein. Everything you wanted to know about phyzzx but didn't know to ask. Technical report, 1984. Stanford Linear Accelerator Publication, SLAC-TN-84-7. [284](#)
- [52] J.V. White. *Graphic Design for the Electronic Age*. Watson-Guptill, 1988. [169](#)

# 版本历史

## Version 1.4

Reinstated a couple of figures (baseline distance, plus one one paragraph shape that didn't make it into the original book.)

## Version 1.3

Finally managed to reinstate the tables chapter.  
Starting to add more concepts to the index.

## Version 1.2

Added chapter references to glossary.  
Fixed a bunch of typographic accidents.

## Version 1.1

Small remark about `\afterassignment` after macro definitions.  
Trouble with indexing macros fixed, I hope.  
Separate letter and a4 versions.  
Better intro for the chapter [20](#) on spacing.