HiTEX

User Manual

Für Beatriz

Version 1.1 (Draft)

MARTIN RUCKERT *Munich University of Applied Sciences*

The author has taken care in the preparation of this document, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Internet page `http://hint.userweb.mwn.de/hint/hitex.html` may contain current information, downloadable software, and news.

Copyright © 2022 by Martin Ruckert

All rights reserved.

This publication is protected by copyright, and permission must be obtained prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Martin Ruckert, Hochschule München, Fakultät für Informatik und Mathematik, Lothstrasse 64, 80335 München, Germany.

`ruckert@cs.hm.edu`

Last commit: Tue Feb 10 19:35:41 2026

Contents

1 Introduction

When I started the HINT project in 2017, I tried to keep the project as small as possible to increase the chances that I would be able to complete it. So one design decision was to keep things simple—or to quote an aphorism attributed to Albert Einstein: "Make things as simple as possible, but not simpler". The other imperative was: keep things out of the viewer if possible because I do not know how much processing power or battery power is available.

As a consequence, I focused on Donald Knuth' original TEX, disregarding all later developments like ε-TEX or LATEX, and I decided that the TEX interpreter would not need to run in the viewer. Of course TEX's line breaking routine will run in the viewer and modifications of TEX's page breaking routine. But the decision to keep the TEX interpreter out of the HINT viewer implies that HINT files do not contain token lists and that there are neither output routines nor marks. To replace them, the HINT file format includes page templates. I have described the technical means to specify page templates below and try to explain the rationale behind it, but HINT's page templates are at the time of this writing a largely untested area.

By now, the state of the HINT project is far beyond of what I had expected then, and the processing power of even low-cost mobile devices is far better than expected especially when programming the graphics card directly using OpenEGL.

The following sections will describe all the primitive control sequences that are special for HiTEX. I tried to be as close to similar primitives that have proven to be useful in other engines, notably pdfTEX, to make it easy for package writers to support the HiTEX engine.

While currently HiTEX is the only TEX engine that supports output in the HINT file format, this might not be so forever. To avoid unnecessary complications for package writers, it is strongly suggested that all such TEX engines implement the same primitives according to the same specification. The following is the first draft of this specification. All the primitives use `HINT` as a prefix to avoid name conflicts. The prefix `HINT`, as opposed to e.g. `HiTeX`, was chosen to stress the idea that these primitives are specific for the output format—not for the TEX engine.

It is common practice in other TEX engines to support the `\special` primitive to insert raw code snippets in the output. Using this primitive, it is possible to insert PostScript code into a PS file, or PDF code in a PDF output file. It is currently not planed to support this mechanism for HINT output files for two reasons: First, the development of HiTEX is closely related to the development of

the HINT file format and therefore features that are part of the HINT file format will enjoy support in HiTEX by corresponding primitives. Everything that is not available through primitives in HiTEX should be considered "internal" and might change in the future. Second, HiTEX is not considered a replacement for but a supplement to other engines. If your aim is the production of a printed book, your will target one of the engines that produce PDF output. But if, on occasion, you want to read what you wrote on a computer screen, you might just use HiTEX to process your source file. At this point you do not want to write `\special` commands for the new target; you want HiTEX as a plug-in replacement for your main target engine, even if it is not completely faithful to your final printed book.

2 HiTEX primitives

Because this is the first specification that will reach a wider user base, it is reasonable to expect changes to occur in the future. Therefore it is recommended that these primitives should not be used directly in a TEX document; instead they should be encapsulated in suitable macros so that the consequences of any change to the primitives will be local to these macros.

2.1 Syntax Description

In the following, we describe the syntax of primitive control sequences which were added to TEX.

- We use a `typewriter font` for text that occurs verbatim in the TEX document.
- We use ⟨*italics*⟩ enclosed in pointed brackets to denote symbols.
- We use rules to define the meaning of symbols. A rule starts with the symbol to be explained, followed by a colon "**:**", and then the text that this symbol stands for. A rule ends with a period "**.**".
- Optional parts of the rule's text are enclosed in [square brackets].
- Alternatives are separated by a vertical bar "|".
- Some symbols refer to text that is defined as part of standard TEX. These are explained here by an example:

  ⟨*integer*⟩ ⟶ an integer as in `\penalty`⟨*integer*⟩
  ⟨*number*⟩ ⟶ a general number as in `\kern`⟨*number*⟩`pt`
  ⟨*normal dimension*⟩ ⟶ a dimension as in `\hrule width` ⟨*normal dimension*⟩
  ⟨*dimension*⟩ ⟶ a dimension as in `\vskip 0pt plus` ⟨*dimension*⟩
  ⟨*name*⟩ ⟶ a name as in `\input` ⟨*name*⟩
  ⟨*vertical list*⟩ ⟶ a token list with matching braces as in `\vbox{`⟨*vertical list*⟩`}`
  ⟨*horizontal list*⟩ ⟶ a token list with matching braces as in
  `\hbox{`⟨*horizontal list*⟩`}`
  ⟨*general text*⟩ ⟶ a token list with matching braces as in
  `\write{`⟨*general text*⟩`}`

2.2 Version and Revision

The control sequences `\HINTversion` and `\HINTminorversion` are used to determine the major and minor version numbers of the HINT output format that is generated by HiTEX. It can be used as part of the output as in `\the\HINTversion`. The most important use, however, is testing whether the current TEX engine will in fact produce HINT output. As an example the file `ifhint.tex` contains the following code:

```
\newif\ifhint
\expandafter
  \ifx\csname HINTversion\endcsname\relax
  \hintfalse\else\hinttrue\fi
```

2.3 UTF8 Input

Starting with HiTEX version 2.0, published with TEX Live 2026, the one and only input encoding supported by HiTEX is UTF8. While other input encodings may be supported by loading special packages, it is recomended not to use such packages but instead use one of the commonly available tools, like `iconv` or `Notepad++` to convert the input files before processing them with HiTEX.

On the technical level, HiTEX converts any (multibyte) UTF8 codepoint from an input file into a single character token with a character value in the range 0 to `"10FFFF`. On output, a character token with a value within the ASCII range 0 to `"7F` is conveted using the "classic" output routines of TEX that attempt to present even non-printable characters in a readable form, while a character token with a value greater than `"7F` is converted to its multibyte UTF8 encoding.

Following the example of LuaTEX and XETEX, a number of primitives were extended and others were added to cope with the larger range of character codes. The syntax and semantics of these primitives are described in the following.

2.3.1 Working with character codes

A character code in HiTEX is no longer an ⟨*8-bit number*⟩ but an ⟨*utf character code*⟩:

⟨*utf character code*⟩ ⟶ a ⟨*number*⟩ in the range `0` to `"10FFFF`

Some traditional primitives are simply extended to handle these larger character codes. To get access to any character whatsoever, you can type `\char`⟨*utf character code*⟩. Similarly, you can write `\chardef`⟨*control sequence*⟩`=`⟨*utf character code*⟩.

New are the primitives `\Uchar` and `\Ucharcat` as defined by the XETEX manual. `\Uchar`⟨*utf character code*⟩ expands to a character token with the specified ⟨*utf character code*⟩ with category code 12. While it looks superficially like the TEX primitive `\char`, `\Uchar` is an expandable operation. `\Ucharcat`⟨*utf character code*⟩ ⟨*catcode*⟩ expands to a character token with the specified ⟨*utf character code*⟩ and the given ⟨*catcode*⟩. The values allowed for ⟨*catcode*⟩ are: 1–4, 6–8 and 10–13.

With a small exception, `\if`, `\ifcat`, and `\ifx` work for unicode characters exactly as described in the TEX book: If the argument of `\if` is a control sequence, TEX considers it to have category code 16 and character code 256 which is 1 bigger than TEX's largest character code; HiTEX will consider it to have category code 16 as well, but the character code will be `"110000` which is 1 bigger than the largest unicode character code.

HiTEX makes it possible to use any unicode character as an active character or define a single character control sequence using such a character.

TEX keeps multibyte character control sequences in a hash table and single byte control sequences in a separate table with 256 entries. Since the range of single byte UTF8 codes is only 0 to `"7F`, HiTEX needs only a smaller table with 127 entries for these controll sequences. Characters outside this range are coded with multiple bytes and go into TEX's hash table. The same schema is used for active characters. It is, however, necessary to use a separate hash table for characters that have a multibyte UTF8 coding, because, the control sequence associated with an active character is not the same thing as the single character control sequence for the same character.

TEX allows the use of the ‘ (left quote) character as a prefix to any character to specify its numeric value. For example `\count1=98` is equivalent to `\count1=‘b`. HiTEX allows this for all unicode characters. For example `\count1=8491` is equivalent to `\count1=‘Å` using the Angström symbol.

2.3.2 Setting and retrieving character information

To typeset characters correctly, TEX relies on a number of properties associated with a character: its `\catcode`, `\sfcode`, `\uccode`, `\lccode`, `\mathcode`, and `\delcode`.

While TEX traditionaly stores this information in arrays indexed by the character code, this approach is no longer appropriate if character codes span a range from 0 to 1114111 as is the case for UTF coded characters. For this reason, HiTEX uses a compressed tree representation that is compact (about 40kByte) and allows very fast access. Modifying the data stored this way is less convenient and will run out of table space if more than a few dozend character codes need to be changed. Since traditionaly character data is changed only for code vales less than 256, HiTEX assumes that most changes of `\catcode`, `\sfcode`, `\mathcode`, and `\delcode` occur in this range and keeps this data in TEX's traditional arrays where changes will not affect the compressed tables.

It is to bee seen if this approach will work in practice or some other data structure will deliver similar performance while allowing large scale across the whole range of unicode characters.

Because the compression is a complicated operation, it is not done at run time, but a separate program computes the compressed tables which are loaded into the HiTEX engine at compiletime. Therefore HiTEX is different from other unicode TEX engines: With HiTEX, it is not a good idea to load the unicode character data at runtime. The loading would destroy the compression and would sooner or

later produce and overflow error. Instead the character data is already part of the HiTEX engine when it starts with exactly the same values it would and should have after the runtime initialization.

Now let's consider the details.

2.3.3 `\catcode` *and* `\sfcode`

These primitives work pretty much as usual, except that their argument is no longer an ⟨*8-bit number*⟩ but ⟨*utf character code*⟩:

The values—category codes are in the range from 1 to 15 and space factor codes are in the range from 0 to 32767—can be retrieved like this:

⟨*internal integer*⟩ ⟶ `\catcode`⟨*utf character code*⟩
⟨*internal integer*⟩ ⟶ `\sfcode`⟨*utf character code*⟩

Setting new values is equaly simple. The assigned ⟨*number*⟩ must be in the propper range.

⟨*code assignment*⟩ ⟶ `\catcode`⟨*utf character code*⟩ [`=`] ⟨*number*⟩
⟨*code assignment*⟩ ⟶ `\sfcode`⟨*utf character code*⟩ [`=`] ⟨*number*⟩

2.3.4 `\uccode` *and* `\lccode`

These primitives are again extended to work with the full range of unicode characters. It is expected that the assignment of new lower case or upper case values is relatively rare. Sometimes it is used in certain coding tricks. So TEX's traditional tables are not retained and all information is stored in the new compressed format. Take this into accound before attempting to large scale rearangements of these assignments. Information about "title case" is currently not implemented.

⟨*internal integer*⟩ ⟶ `\lccode`⟨*utf character code*⟩
⟨*internal integer*⟩ ⟶ `\uccode`⟨*utf character code*⟩
⟨*code assignment*⟩ ⟶ `\lccode`⟨*utf character code*⟩ [`=`] ⟨*utf character code*⟩
⟨*code assignment*⟩ ⟶ `\uccode`⟨*utf character code*⟩ [`=`] ⟨*utf character code*⟩

2.3.5 `\mathcode`, `\delcode` *and friends*

Before diving deeper into this section, there is a confession due: using HiTEX with unicode fonts in math mode is a largely untested area. But the basics are implemented in the HiTEX engine.

Making the `\mathcode` primitive work nicely with unicode values requires some extra work because TEX packs three different values, the class, the family, and the character code into one integer. In hexadecimal notation a mathcode contains four hexadecimal digits: the leading hex-digit specifies the class, the next hex-digit specifies the familiy and the lowest two digits a one byte character code.

⟨*class*⟩ ⟶ ⟨*hexdigit*⟩
⟨*family*⟩ ⟶ ⟨*hexdigit*⟩

⟨*byte*⟩ ⟶ ⟨*hexdigit*⟩⟨*hexdigit*⟩
⟨*math code*⟩ ⟶ ⟨*class*⟩⟨*familiy*⟩⟨*byte*⟩

It is not strictly necessary to give the number in hexadecimal; it is possible to convert the hexadecimal number into a decimal number and write it down in decimal; or in any other format that TEX provides for the representation of numbers.

To be compatible with existing TEX code this format is retained when using `\mathcode`, only the argument of `\mathcode` now can be any unicode character.

⟨*math code*⟩ ⟶ `\mathcode`⟨*utf character code*⟩
⟨*code assignment*⟩ ⟶ `\mathcode`⟨*utf character code*⟩ [`=`] ⟨*math code*⟩

So you can say `\mathcode96` as well as `\mathcode9600` and in both cases, you will get a math code in the range 0 to `"FFFF`. If however the class stored for the character in question is not in the range 0 to `"F`, or the family is not in the range `"F`, or the character code does not fit into one byte, `\mathcode` will return zero. You can also use `\mathcode9600` to give a new math code value to the unicode character 9600, but the new value is limited to a one byte character code. Note also that you can continue to use the special math code value `"8000` to make an "active" math character.

To set or retrieve math codes with arbitrary unicode characters, HiTEX follows the example of LuaTEX and XETEX and packs a 3-bit class number, a 8-bit family number and a 21-bit unicode character into a 32-bit value: the 8 most siginficant bits contain the family, the next 3 bits contain the class, and the 21 least significant bits contain the unicode character.

⟨*Uclass*⟩ ⟶ ⟨*3-bit*⟩
⟨*Ufamily*⟩ ⟶ ⟨*8-bit*⟩
⟨*Ucode*⟩ ⟶ ⟨*21-bit*⟩
⟨*Umath code*⟩ ⟶ ⟨*Ufamily*⟩⟨*Uclass*⟩⟨*Ucode*⟩

Because the three values are not nicely aligned on the four bit boundaries of an hexadecimal notation two new primitives `\Umathcode` and `\Umathcodenum` are provided. `\Umathcodenum` is a simple extension of `\mathcode`. It returns a single number in the bit-packed format as just described and when a new value is set it must already be in the correct bit-packed format.

⟨*Umath code*⟩ ⟶ `\Umathcodenum`⟨*utf character code*⟩
⟨*code assignment*⟩ ⟶ `\Umathcodenum`⟨*utf character code*⟩ [`=`] ⟨*Umath code*⟩

`\Umathcode` is a little easier to use because it does not require the user to do the bit packing. Instead a new math code can be set by giving three separate numbers: one for the class, one for the family and one for the character code.

⟨*class number*⟩ ⟶ a ⟨*number*⟩ in the range 0 to 7
⟨*family number*⟩ ⟶ a ⟨*number*⟩ in the range 0 to 255
⟨*Umath code*⟩ ⟶ `\Umathcode`⟨*utf character code*⟩
⟨*code assignment*⟩ ⟶ `\Umathcode`⟨*utf character code*⟩ [`=`] ⟨*class number*⟩
⟨*family number*⟩ ⟨*utf character code*⟩

HiTEX's `\delcode` primitive offers no surprise. For any unicode character, you can use `\delcode` to set or retrieve its delimiter code. The traditional delimiter code is in the range -1 to `"FFFFFF`.

⟨*internal integer*⟩ ⟶ `\delcode`⟨*utf character code*⟩
⟨*code assignment*⟩ ⟶ `\delcode`⟨*utf character code*⟩ [`=`] ⟨*number*⟩

The primitives `\Udelcode` and `\Udelcodenum`, as known from XETEX or LuaTEX, are not yet implemented.

2.4 OpenType fonts

When TEX was invented, digital fonts, especially fonts suitable to typeset mathematics, were a scarce commodity. But TEX came with a companion, METAFONT, to create digital fonts. These fonts had their own glyph encoding and their own file format for glyphs (`.pk` files) and for font metrics (`.tfm` files). Soon after that, the American Mathematical Society commissioned a collection of TEX font files in the PostScript Type 1 format, which allowed the creation of PostScript (and much later PDF) output files with TEX. The PostScript Type 1 fonts largely replaced the original `.pk` files. A replacement for the font metric files was not necessary and so the `.tfm` files are still in use today.

In the meantime, the number of high quality, freely available, digital fonts has exploded. And "high quality" is no longer equivalent to "PostScript Type 1". Today, the most popular file format for fonts is the OpenType format, an extension of the slightly simpler TrueType format. So it seems natural to extend the capabilities of TEX to work with these types of font files. OpenType font files not only contain the glyphs, but they also contain the necessary font metrics. So an OpenType font file can replace both the `.pk` file and the `.tfm` file.

To define fonts, HiTEX extents the syntax and semantics of TEX's `\font` primitive. It tries to be simple and as much as possible compatible with the implementation of the `\font` primitive in LuaTEX and XETEX.

The syntax is:

⟨*font assignment*⟩ ⟶ `\font`⟨*control sequence*⟩ [`=`] ⟨*input file*⟩ [⟨*at clause*⟩]
⟨*input file*⟩ ⟶ `"`⟨*font specifier*⟩`"`
⟨*input file*⟩ ⟶ `{`⟨*font specifier*⟩`}`
⟨*input file*⟩ ⟶ ⟨*font specifier*⟩
⟨*font specifier*⟩ ⟶ ⟨*pathname*⟩
⟨*at clause*⟩ ⟶ `at` ⟨*dimen*⟩ | `scaled` ⟨*number*⟩

The ⟨*at clause*⟩ to adjust the font size is described in the TEX book and has not changed.

In modern TEX engines, an ⟨*input file*⟩ can be more than just a sequence of characters terminated by a space token or a `\relax`. If the name of the ⟨*input file*⟩ contains spaces, it is possible to enclose it in `"` (double quote) characters. In fact, if a file name contains double quote characters they switch quoting alternatingly on and off and are otherwise ignored as parts of the file name. Inside the quoted

parts, space characters are part of the file name; the first space character outside of a quoted part terminates the file name. A more convenient way to define an arbitrary ⟨*input file*⟩ is enclosing it in curly braces `{...}`. In this case the general text inside the braces—with spaces, special symbols, expanded macros, and all double quotes removed—is considered the ⟨*input file*⟩.

After HiTEX has done all this preprocessing of the ⟨*input file*⟩, the result is a ⟨*font specifier*⟩ Before we look at the more complex forms of a ⟨*font specifier*⟩ we look at the simple case of a ⟨*pathname*⟩. In this case, HiTEX passes it to the `kpathsearch` library to find a matching `.tfm` file. If the `.tfm` file is found, HiTEX will assume that the font is one of the traditional TEX fonts, using TEX's traditional encoding. Later, HiTEX will then use again the `kpathsearch` library to find and open the matching PostScript Type 1 font and if that is not possible a matching `.pk` file. Thats all.

If the `kpathsearch` library can not find a suitable `.tfm` file, HiTEX assumes that an OpenType or TrueType file should be loaded. In this case, a ⟨*font specifier*⟩ is not just simply the name of a file, but it has a very sophisticated syntax—mostly due to the attempt to be compatibel with existing implementations.

⟨*font specifier*⟩ ⟶ ⟨*font file name*⟩ [`:` ⟨*font features*⟩]
⟨*font file name*⟩ ⟶ `file:`⟨*pathname*⟩ [⟨*font selector*⟩]
⟨*font file name*⟩ ⟶ `[`⟨*pathname*⟩`]`[⟨*font selector*⟩]
⟨*font selector*⟩ ⟶ `(`⟨*number*⟩`)`
⟨*font specifier*⟩ ⟶ `name:`⟨*font name*⟩ [`:` ⟨*font features*⟩]

• Note that the use of ⟨*font features*⟩ is currently not yet implemented. Any ⟨*font features*⟩ given will be silently ignored.

• Note that font lookup using the `name:` prefix to indicate the use of "font names" as opposed to "file names" is currently not yet implemented. Currently only file name lookup is implemented.

• Note that a ⟨*font selector*⟩ must be an index number. Using the PostScript name to select a font is not yet implemented.

• Note that HiTEX in an attempt to maintain compatibility with other engines will not give up on a font specifier that neither leads to a matching `.tfm` file nor has a `file:` or `name:` prefix and is not bracketed either. Instead HiTEX will issue a warning and tries to resolve the problem. First, it tries to assume that the `file` prefix is missing, and if that does not help, it tries to assume that the `name:` prefix is missing. Only if none of this leads to a usable font file, HiTEX succumbs.

The extended font specifiers used for OpenType fonts assign special meaning to the colon "`:`", the slash "`/`", and the opening parenthesis "`(`". If these characters occur in the ⟨*pathname*⟩ of a font file, it is possible to use a bracketed notation of the ⟨*pathname*⟩ as a alternative to using the `file:` prefix. If the pathname is enclosed in square brackets, it may contain any character except the closing square bracket; otherwise it must not contain the colon "`:`", the slash "`/`", nor the opening parenthesis "`(`". Since the colon can be necessary after a drive letter in Windows and the slash serves as a directory separator in Linux, the bracketed verions is

quite common. The paraenthesis is excluded from the unbracketed version because is start the optional ⟨*font selector*⟩.

Once HiTEX has determined the ⟨*pathname*⟩, it uses the `kpathsearch` library to find a matching OpenType font and if this does not succeed, it tries to find a matching TrueType font. Once the file is found, the Harfbuzz library is used to open the font file.

• Note that it is possible to open PostScript Type 1 font files without a `.tfm` file using Harfbuzz. The use of Type 1 fonts with Harfbuzz, however, requires the use of `.afm` files, and this does not work well and is deprecated. Therefore HiTEX does not support it. For Type 1 fonts, one should use `afm2tfm` to convert the `.afm` files to `tfm` files and put the new `tfm` files in a place where the `kpathsearch` library can find them. Then run `mktexls`.

2.4.1 Embedding subsets of fonts

2.5 Images

The primitive `\HINTimage` includes an image in a document. The syntax is as follows:

`\HINTimage` [`=`] ⟨*name*⟩ [⟨*width*⟩] [⟨*height*⟩]

The optional equal sign can be added to make the code look nicer. The ⟨*name*⟩ specifies the image file. The width specification determines the width of the image. If omitted, HiTEX tries to determine the image's width from the image file. The same holds for the height specification.

⟨*width*⟩ ⟶ `width` ⟨*normal dimension*⟩
⟨*height*⟩ ⟶ `height` ⟨*normal dimension*⟩

Note that a ⟨*normal dimension*⟩ that is computed from `\hsize` or `\vsize` retains this dependency when processed by HiTEX. This allows an image to adapt to the size of the viewing area. Scaling in the HINT viewer will, however, never change the aspect ratio of an image. So it may become smaller or larger, but it will never be distorted. For this reason, HiTEX will inspect the image file to determine the aspect ratio of the stored image. The width and height values as given in the TEX file serve as the maximum values for the actual width and height. When rendering, the image will become as large as possible within the given bounds. If TEX does not specify neither width nor height, the image file must specify the absolute width and height of the image. It is considered an error if valid settings for the image's width and height can not be obtained.

2.6 Colors

Since the HINT file format is designed for on-screen viewing, the only color model supported is the RGBA model, where a color is specified by four values: the red, the green, the blue, and the alpha value. The first three determine the light intensity of the red, green, and blue component of a pixel; the alpha value determines the relative share of a color when displaying one color on top of another color. Because in practice most display devices use one byte for each of the four values that define a color, the HINTfile format stores the four color components using integer values in the range 0 to 255. Independent of the input format, HiTEX will convert all colors to this format when storing them in the output file.

2.6.1 Foreground Color

The most common color specification is the specification of a foreground color. (We will consider background colors below.) A foreground color can be specified using the following syntax:

⟨*foreground*⟩ ⟶ `FG{` ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ [⟨*integer*⟩] `}`

Note that for convenience, the alpha value is optional; if no alpha value is given, the value 255 will be used and the color is completely opaque.

Here are some examples: `FG{255 0 0}`, `FG{255 0 0 255}`, both specify the same plain opaque red; `FG{0 0 255}` is plain blue; `FG{255 255 0 127}` is a transparent yellow. Because each value fits in a single byte, the values are often given in hexadecimal notation. In TEX, hexadecimal values are written with a `"` prefix. The same colors as before are then written `FG{"FF 0 0}`, `FG{"FF 0 0 "FF}`, `FG{0 0 "FF}` and `FG{"FF "FF 0 "7F}`. Values greater than 255 or less than 0 are not allowed.

A common alternative to the color representation just described is the device independent notation where each value is a real number in the interval from 0 to 1. To keep both representations apart, the device independent representation (with the smaller numbers) uses the lowercase keyword `fg` instead of `FG`. Here is the syntax:

⟨*foreground*⟩ ⟶ `fg{` ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ [⟨*number*⟩] `}`

Using the new syntax, the colors above are written `fg{1 0 0}`, `fg{1 0 0 1}`, `fg{0 0 1}` and `fg{1 1 0 0.5}`. Values greater than 1 and less than 0 are not allowed. Note that `fg{1 1 1}` is pure white while `FG{1 1 1}` is the darkest possible gray, which on most devices is indistinguishable from pure black.

When specifying colors for computer screen, using red, green, and blue components is natural. For printing on paper, the specification using cyan, magenta, yellow, and black is the default. Since collections of named colors using the latter format are common, HiTEX allows the use of this format by prefixing the numbers with the keyword `cmyk`. Specifying the keyword `rgb` is also possible and has the same effect as giving no keyword. Using the new syntax the transparent yellow can

be written `fg{cmyk 0 0 1 0 0.5}`, `FG{cmyk 0 0 "FF 0 "7F}`, `fg{rgb 1 1 0 0.5}`, or `FG{rgb "FF "FF 0 "7F}`.

The additional syntax rules are:

⟨*foreground*⟩ ⟶ `fg{ rgb` ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ [⟨*number*⟩] `}`
⟨*foreground*⟩ ⟶ `fg{ cmyk` ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ [⟨*number*⟩] `}`
⟨*foreground*⟩ ⟶ `FG{ rgb` ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ [⟨*integer*⟩] `}`
⟨*foreground*⟩ ⟶ `FG{ cmyk` ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ [⟨*integer*⟩] `}`

2.6.2 Defining and Using Colors

As we will see, colors come in whole sets of colors. To define such a set of colors, HiTEX provides the primitive `\HINTcolor`. Its syntax is

`\HINTcolor {` ⟨*color specification*⟩ `}`

Before we give the complete definition of a ⟨*color specification*⟩, we start with some examples. In its simplest form this primitive just specifies a single color. For example `\HINTcolor{fg{0 0 0}}` specifies the foreground color black which is then used for rules and glyphs. In addition to the foreground color, you can specify a background color. For example, black text on white background is specified by `\HINTcolor{fg{0 0 0} bg{1 1 1}}` or `\HINTcolor{fg{0 0 0} BG{"FF "FF "FF}}`.

The viewer for HINT files may provide a "dark" mode, and as a document author, you can specify the colors also for dark mode. If you like white letters on dark blue background you can write `\HINTcolor{fg{0 0 0} bg{1 1 1} dark fg{1 1 1} bg{0 0 0.3}}`.

There are two more colors that an author might care about: When searching for a text, all occurrences of the search phrase are highlighted by using a different color. And while the user iterates over the occurrences on the page, one occurrence has the "focus" and is rendered again in a different color. You can specify the highlight color right after the normal text color and the focus color right after the highlight color. The same can be done for the colors in "dark" mode.

Here are the remaining rules that complete the ⟨*color specification*⟩:

⟨*color specification*⟩ ⟶ ⟨*color set*⟩ [`dark` ⟨*color set*⟩]
⟨*color set*⟩ ⟶ ⟨*color*⟩ [⟨*color*⟩ [⟨*color*⟩]]
⟨*color*⟩ ⟶ ⟨*foreground*⟩ [⟨*background*⟩]
⟨*background*⟩ ⟶ `FG{` [`rgb`] ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ [⟨*integer*⟩] `}`
⟨*background*⟩ ⟶ `fg{` [`rgb`] ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ [⟨*number*⟩] `}`
⟨*background*⟩ ⟶ `FG{ cmyk` ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ ⟨*integer*⟩ [⟨*integer*⟩] `}`
⟨*background*⟩ ⟶ `fg{ cmyk` ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ ⟨*number*⟩ [⟨*number*⟩] `}`

If some of the optional parts in the ⟨*color specification*⟩ are missing, the corresponding colors from the set of default colors, as described below, are used.

Note that the background colors for highlighted text and focus text can be given, but current viewers ignore these background specifications. Further note

that the current specification of the HINT file format limits the total number of different color specifications in a document to 255.

The colors given in `\HINTcolor` will have an immediate effect on all following rules and glyphs and the background of the enclosing box. The effect will persist until the next change of colors or until the end of the box—whatever occurs first.

The line breaking algorithm of HiTEX tracks changes in color within a paragraph and reinsert an appropriate color change at the start of every `\hbox` that contains a new line. In this way local color changes inside a paragraph can span multiple lines but do not affect the inter line glue or material that is inserted with `\vadjust`. Similarly, spliting off the initial part of a vertical box with `\vsplit` will insert a color node in the remaining part if necessary to keep the color consistent accross the split.

Special care is needed if background colors are used. Unless the background color is completely transparent with an alpha value equal to zero, the background color will fill a vertical box from left to right and a horizontal box from top to bottom. Since height, depth, and width of boxes often depend on the text that is inside, which in turn might depend on the outcome of line breaking, it is strongly recommended to use background colors with caution, and use `\strut`s to enforce a fixed height and depth of horizontal boxes.

2.6.3 Default Colors

The HINT file format specifies default values for all colors. HiTEX provides the primitive `\HINTdefaultcolor` to overwrite these default colors. This primitive must not be used after defining any custom colors using `\HINTcolor`. Its syntax is

`\HINTdefaultcolor {` ⟨*color specification*⟩ `}`

The HINT format specifies the following default colors: Normal text is black `FG{0 0 0}`, highlight text is a slightly dark red `FG{"EE 0 0}`, and focus text is slighty dark green `FG{0 "EE 0}`. The background is transparent white `BG{"FF "FF "FF 0}`. In dark mode the background is transparent black `BG{0 0 0 0}`, normal text is white `FG{"FF "FF "FF}`, and a slightly lighter red `FG{"FF "11 "11}`, and green `FG{"11 "FF "11}`, are used for highlighted and focus text.

2.6.4 Nesting Colors

A color change is limited to the enclosing box. Hence the nesting of boxes leads to a nesting of color definitions. So for example a transparent background color in the inner box will not completely replace the background color of the enclosing box but will only modify this color like seeing it through colored glas.

A color change ends not only at the end of the enclosing box, it will also end at the next use of the `\HINTcolor` or `\HINTendcolor` primitive: The `\HINTcolor` primitive will replace the current colors by a new set of colors; the `\HINTendcolor` primitive will resume the color specification that was valid just before the matching use of `\HINTcolor`. HiTEX maintains a color stack tracking local color

changes within a box or paragraph, and uses it to insert appropriate color changes so that the `\HINTendcolor` primitive will simply cancel the color change by the matching `\HINTcolor` primitive. If there is no matching `\HINTcolor` primitive, the `\HINTendcolor` primitive is silently ignored. Note that within a single box, there is at any point only a single background color: The color stack will switch from one background color to an other background color but will not overlay an "inner" background color over an "outer" background color. This is only the case when multiple boxes are nested as described above.

Here is an example: Suppose we want the TEX logo to be rendered in light red, and notes in dark green. You can write

```
\def\redTeX{\HINTcolor{fg{1 0.3 0.3}}\TeX\HINTendcolor}
\def\beginnote{\HINTcolor{fg{0 0.5 0}}}% dark green
\def\endnote{\HINTendcolor}

This is an example showing the \redTeX\ logo in red color.
\beginnote Note how the \redTeX\ logo is still red inside this
note.\endnote
```

After the first occurrence of the red TEX logo, the color will be switched back to normal black, while after the second occurrence the color will be switched back to dark green. The color switching will work as intended even if the paragraph is spread over several lines by the line breaking routine.

2.6.5 Colors for Pages

When a page get rendered in the HINT file viewer, the renderer starts with the default colors and the page is initially cleared using the default background color. If a different page color is desired, color changes can be added to the page templates.

In a vertical box, the color stack of HiTEX has a similar effect as in a horizontal box. Similar to the precautions in the line breaking routine, HiTEX will insert color changes when splitting a vertical box with `\vsplit`. Complications arise from color changes in the top level vertical list which is split into pages in the HINT viewer at runtime. Because the page builder in the viewer has no global information and should not need global information, HiTEX will insert copies of the local color information after every possible breakpoint in the top level vertical list. This will ensure that page breaks will not affect the colors of the displayed material. Note, however, that TEX considers glue (and kerns) as discardable and will remove these items from the top of a new page. Because glues and kerns are colored using the current background color, these items might be visible on a page but disappear when they follow immediately after a page break. So if you want the effect of a colored glue or kern that is not affected by a page break, you should include it inside a box or use a colored rule instead.

2.6.6 Colors for Links

The most common change in color is caused by the use of links. To support this changing of colors, the primitives `\HINTstartlink` and `\HINTendlink` (see section 2.7) cause an automatic change of the color specification. A document author can set the default colors used for links with the primitive `\HINTdefaultlinkcolor` and change the current link color with the primitive `\HINTlinkcolor`. The syntax is:

`\HINTdefaultlinkcolor {` ⟨*color specification*⟩ `}`

`\HINTlinkcolor {` ⟨*color specification*⟩ `}`

For convenience, the HINT file format specifies default colors for links as well: links use dark blue `FG{0 0 "EE}` and in dark mode light blue `FG{"11 "11 "FF}`. The primitive `\HINTdefaultlinkcolor` is used to partly or completely redefine these defaults.

Later uses of `\HINTlinkcolor` will set new current link colors. Colors that are missing in the new link color specification are taken from the corresponding default colors for links.

Whenever the `\HINTstartlink` primitive is used, its effect on the colors is equivalent to the `\HINTcolor` primitive using the current link color. This implies that the color change caused by `\HINTstartlink` is local to the enclosing box.

Whenever the `\HINTendlink` primitive is used, it will restore the color stack of HiTEX to its state before the matching `\HINTstartlink`. It is the responsibility of the TEX source code (or package) to keep the sequence of `\HINTstartlink`, `\HINTendlink`, `\HINTcolor`, and `\HINTendcolor` properly nested. A sequence like "`\HINTstartlink` ... `\HINTcolor` ... `\HINTendlink` ... `\HINTendcolor`" is possible, but it will cause `\HINTendlink` to restore the colors to those in effect before the `\HINTstartlink`. The following `\HINTendcolor` will then either restore a color of a matching `\HINTcolor` preceeding the link in the same box or it will restore the color in the outer box, or it will be ignored. In effect, the color changes inside a link stay local to the link.

2.6.7 LATEX Support

Starting with TEX Live 2025, there is a limited support for the `xcolor` package.

After `\usepackage{xcolor}` you can use the predefined standard colors; for example `\color{red}`. If you add one (or several) of the named color options `x11names`, `svgnames`, or `dvipsnames` to the package, you can also use commands like `\color{Tomato4}` (x11), `\color{BlanchedAlmond}` (svg), or `\color{Plum}` (dvips).

To define your own colors you can use for example

`\definecolor{mypink1}{rgb}{0.858, 0.188, 0.478}`,

`\definecolor{mypink2}{RGB}{219, "30, 122}`,

`\definecolor{mypink3}{cmyk}{0, 0.7808, 0.4429, 0.1412}`, or

`\definecolor{mygray}{gray}{0.2}`.

The mixing of colors is supported as well. For example a mixture of 40% green and 60% yellow look is produced by `\color{green!40!yellow}`.

The colors for links and other references can be given as options to the `hyperref` package. For example as in

```
\usepackage[linkcolor=green,urlcolor=red]{hyperref}
```

2.6.8 Differences between LaTeX and HiLaTeX

Colors and Groups

In LaTeX, colors are local to the group. So by writing "`text 1 {\color{blue} text 2 } text 3`" after `text␣1` the color of `text␣2` will change to blue and after `}` marking the end of the group, the color of `text␣3` will revert to the color of `text␣1`. HiLaTeX emulates this behaviour by inserting `\HINTendcolor` at the end of the group.

When it comes to paragraphs, the scoping rules of colors in HiLaTeX are however slightly different from the LaTeX scoping rules. In TeX and LaTeX, boxes and references all have there own group, but this is not true for paragraphs. So TeX or LaTeX will allow you to start a new group in one paragraph and end the group in the next paragraph, while it is not possible to start a group in one box and end the group in another box. As a consequence, you can switch to blue text color in the middle of a paragraph and end the blue color in the middle of the next paragraph. In HiTeX, on the other hand, when it comes to colors, paragraphs behave pretty much like boxes: The effect of a color change inside a paragraph will not extend past the end of the paragraph. The closing of the group in the next paragraph will then have no effect.

Colors in vertical Lists

The HINT file format allows color specifications in horizontal boxes and—unlike the PDF file format—in vertical boxes as well. Together with the mode switching of TeX, which goes into horizontal mode when it sees the beginning of a paragraph and back into vertical mode at the end of the paragraph, this can cause unexpected color effects.

There is for example a big difference between

```
\color{blue}
The first paragraph ...

The second paragraph ...
```

and

```
\indent
\color{blue}
```

```
The first paragraph ...


The second paragraph ...
```

In the first case, the color change is part of the vertical list and the letter "`T`" starts the paragraph. As a consequence, the color change is still in effect when the second paragraph starts. In the second case, the `\indent` command puts TEX into horizontal mode and the color change becomes part of the first paragraph. As a consequence, the color change will end with the first paragraph, as explained in the previous section.

Even more surprising is this:

```
{\color{blue} Blue} The first paragraph ...


The second paragraph ...
```

TEX finds the begining of the group `{` and the color change in vertical mode and it puts the color change into the vertical list. Then it finds the letter "`B`" and starts the paragraph. When TEX encounters the end of the group, there is no local color change *inside* the paragraph and the text continues to be blue. Even the second paragraph and all following paragraphs will continue in blue until the end of the vertical list.

The confusion that such behaviour might create has its root in TEX's mode switching which is not synchronized with TEX's grouping. While grouping is typically visible in the source text, the mode switching remains largely invisible.

Future Changes

While it may be questionable whether all the color changes shown above makes sense, it is definitely undesirable if HiLaTeX and LaTeX behave differently. As a consequence, HiLaTeX might very well change in this respect in a later version, so that HiTEX will no longer treat the begining and ending of paragraphs as the beginning and ending of a group. It is an open question how HiTEX should handle the end of a group in the middle of a paragraph ending a color change that started in the enclosing vertical list. Currently a `\HINTendcolor` at that position would be silently ignored because it can only undo local changes *inside* the paragraph. Should HiTEX instead change the color of the enclosing vertical box immediately? What does it mean to do this change immediately? At the baseline? Before the next interline glue? What are the implications for the rendering engine? How complicated can it be to look ahead for color changes that occur depply nested inside a vertical list? Would it not be better to demand the use of `\vadjust` for such an effect? Should HiTEX postpone the color change in the enclosing vertical box until the end of the paragraph?

Default Colors

Because complete color specifications are pretty long. It is important to provide usefull defaults. Currently missing elements of a color specification are taken from a single default color specification. It might be convenient to be able to provide a way to define color specifications using the current color as a basis for missing elements.

Color Numbers

The HINT file format references a color set by a number in the range 0 to 254. So HiTEX assigns each color specification a number, using the same number for two identical color specifications. One extension to the above specification of HiTEX's color primitives could be to make these numbers accessible to document authors or package programmers. For example `\the\HINTcolor` could expand to the number n of the current color set and `\HINTcolor`n would be equivalent to a use of `\HINTcolor` with a full color specification that is equivalent to the color specification belonging to n. This would be quite efficient; it would not be necessary to scan the color specification and search the existing color specifications for the matching specification with number n.

The LaTeX named colors are stored as macros, which has the advantage that loading a whole package of color names does not use any of HiTEX's color numbers. Only colors that actually get used (probably only a few) will get a color number. This works well in practice. So currently, there are no plans to implement this extension.

2.7 Links, Labels, and Outlines

A link in a HINT document refers to another location in the same document. It can be used to navigate to that location. A link is defined using the primitives `\HINTstartlink` and `\HINTendlink`. Neither of them can be used in vertical mode. The text between the start and the end of the link constitutes the visible part of the link. Depending on the user interface, clicking or tapping or otherwise activating the link (e.g. pronouncing) will navigate to the destination of the link. The user interface might provide a visual clue to make the user aware of the available links for example using a special cursor when hovering over a link. But it also may choose to leave the visual clues completely to the author of the document (e.g. using a special colors, images, or fonts).

The syntax is `\HINTstartlink` ⟨*destination*⟩ and `\HINTendlink` with

⟨*destination*⟩ ⟶ `goto` ⟨*label*⟩
⟨*label*⟩ ⟶ `name {`⟨*general text*⟩`}` | `num` ⟨*integer*⟩

As you can see, the link refers to its destination using a label which is either a name or a number. The destination can be defined by using the `\HINTdest` primitive. Forward and backward links are allowed; the definition of a label can either precede or follow the use of the label. If at the end of the document a label is undefined, a warning is given, and the label will reference the beginning of the document.

The syntax is

`\HINTdest` ⟨*label*⟩ [⟨*placement*⟩]

with

⟨*placement*⟩ ⟶ `top` | `bot`

The optional placement argument specifies how to build the page containing the destination location. `top` demands a page starting with the destination location. This is useful if the destination is for example the start of a section or chapter heading. Similarly `bot` asks for a page that ends with the destination location. The most common case is to omit the placement argument. In this case, the viewer will build a "good" page that includes the given destination. In case of a section heading, for example, it will most probably start the page with the section heading because section headings are usually preceded by a negative penalty that will convince the page builder that this is a good place to break the page. But if the section heading is immediately preceded by a chapter heading, the negative penalty found there will probably persuade the page builder to start with the chapter heading instead.

There is a special label that has the form `name {HINT.home}`. It is used to mark the "home page" of the document. User interfaces are encouraged to offer a button or keyboard shortcut to navigate to the document location labeled this way. The page should be a convenient starting point to explore the document. The typical place for this label would be the documents table of content.

The labels that identify destinations in a document can also be used to define document outlines. A document outline is a document summary given as a hierarchical list of headings where each of them refers to a specific location in the document.

The syntax is

`\HINToutline` ⟨*destination*⟩ [⟨*depth*⟩] `{`⟨*horizontal list*⟩`}`

⟨*depth*⟩ ⟶ `depth` ⟨*integer*⟩

The user interface can format the ⟨*horizontal list*⟩ much like a `\hbox` would do and display it to the user. When the user selects this text, the document will be repositioned to show the destination location in the same way as with a link. In order to support also simpler user interfaces, the current HINT backend also extracts the characters (and spaces) from the horizontal list (in top-left to bottom-right order) and makes this character string available to the user interface.

The order in which outline items are defined is important because this is the order in which they will be presented to the reader of the document. The optional depth argument allows to structure the list of outline items as a hierarchy. Outline items with a higher depth value are considered to be sub-items of items earlier in the list with lower depth values. If no depth value is given, the depth value is set to zero. It is not necessary to define depth values strictly consecutive.

2.8 Page Templates and Streams

To produce the final page, TEX uses a special piece of program called the output routine. Because a HINT file is pure data, it can not contain output routines. Instead it uses page templates to assemble pages from the main text, footnotes, floating illustrations, and other material. I start here by describing how HINT's page templates work and the special syntax used to specify them in a TEX file that is about to be processed with HiTEX. For those interested in how the design decision was made and how page templates relate to TEX's page building mechanism, a separate section follows at the end.

The syntax of a page template specification is:

`\HINTsetpage` ⟨*integer*⟩ [`=`] ⟨*name*⟩ [⟨*priority*⟩] [⟨*width*⟩] [⟨*height*⟩]
`{`⟨*vertical list*⟩ ⟨*stream definition list*⟩`}`

The ⟨*integer*⟩ specifies the page templates number in the range 1 to 255. The number 0 is reserved for the build in page template of the HINT file format, which is used if no other page template has been defined. The page template 0 can not be redefined. The ⟨*name*⟩ associates a name with the page template. The name can be displayed by the HINT viewer to help the user selecting a suitable page template.

After the name follows an optional priority; it is used to select the "best page template" if multiple templates are available. The default value is 1. The build-in template has priority 0.

⟨*priority*⟩ ⟶ `priority` ⟨*integer*⟩

After that follows an optional width and height of the full page including the margins. After subtracting `\hsize` from the width and `\vsize` from the height, the remainder is used for the margins around the displayed text. For example giving the width as 1.2`\hsize` will leave 0.1`\hsize` for the margins on both sides. In this case the margins will grow together with the available width of the display. If the width is computed by adding 20pt to `\hsize`, the margin will be 10pt on both sides. In this case the margin will not grow with the size of the display, but it will grow with the magnification factor. Of course both methods can be used together. The default is `\hsize` for the width and `\vsize` for the height so there are no margins.

The following ⟨*vertical list*⟩ defines the page itself. It should assign suitable values to `\topskip` and `\maxdepth` because the values valid at the end of the vertical list are stored in the page template and are used in the page building process. The vertical list usually also specifies the insertion of content streams using a ⟨*stream insert point*⟩.

⟨*stream insert point*⟩ ⟶ `\HINTstream` ⟨*integer*⟩

Here the ⟨*integer*⟩ must be in the range 0 to 254. The value 255 is invalid; the value 0 indicates the main body of text (what TEX's page builder would normally put into box 255 before calling the output routine). Otherwise, the ⟨*integer*⟩ is

TEX's insertion number, that is the number of TEX's box containing the insertions. As usual, this box is filled using TEX's `\insert` primitive. So after plain TEX has defined `\footins`, the footnotes for the current page can be inserted after the main body of text in the ⟨*vertical list*⟩ by saying `\HINTstream0` followed by `\HINTstream\footins`. Of course you might want to have a footnote rule and a small skip to separate the footnotes —if there are any—from the main text. This can be achieved by a suitable ⟨*stream definition*⟩ in the ⟨*stream definition list*⟩.

⟨*stream definition list*⟩ ⟶ | ⟨*stream definition list*⟩ ⟨*stream definition*⟩
⟨*stream definition*⟩ ⟶ `\HINTsetstream` ⟨*integer*⟩ [`=`] [`preferred` ⟨*integer*⟩]
[`next` ⟨*integer*⟩] [`ratio` ⟨*integer*⟩] `{`⟨*vertical list*⟩`}`

The first ⟨*integer*⟩ is the streams insertion number i, and it must match the ⟨*integer*⟩ previously used in the ⟨*stream insert point*⟩. Then follows the optional specification of a preferred stream with insertion number p, a next stream with insertion number n, and a split ratio r. If $r > 0$, the contributions to stream i are split between stream p and n in the ratio $r/1000$ for p and $1 - r/1000$ for n before contributing streams p and r to the page. Else if $p \geq 0$ any insertion to stream i is moved to stream p as long as possible, and if $n \geq 0$ we move an insert to stream n if there is "no room left" in p nor in i. How much "room" is available for the insertions is specified inside the vertical list that follows. Here `\dimen`i should be set to the maximum total height of the insertions in class i per page. `\count`i should be set to the magnification factor f, such that inserting a box of height h will contribute $h * f/1000$ to the main page; and `\skip`i should be set to the extra space needed if an insertion in class i is present.

This extra space is usually taken up by material that is inserted before and after the insertions, such as for example the footnote rule. This material can be defined by a ⟨*before list*⟩ and an ⟨*after list*⟩.

⟨*before list*⟩ ⟶ `\HINTbefore` [`=`] `{`⟨*vertical list*⟩`}`
⟨*after list*⟩ ⟶ `\HINTafter` [`=`] `{`⟨*vertical list*⟩`}`

If you are interested in the design decision that motivate the definitions that have be given in this section, you should read section 4.

3 Other Primitives

Since I consider the support for LaTeX to be crucial for the success of the HINT project, quite a few primitives have been added to HiTeX that go beyond TeX's original specification.

3.1 ε-TeX

First, the primitives of ε-TeX have been added with the exception of those primitives that deal with line breaking, with right to left reading, and with marks. Here is a list of ε-TeX primitives that are missing in HiTeX:

- `\TeXXeTstate` (current reading direction)
- `\beginL`, `\endL` (switching reading direction)
- `\beginR`, `\endR` (switching reading direction)
- `\predisplaydirection` (reading direction)
- `\lastlinefit` (line breaking)
- `\marks` (multiple marks)
- `\botmarks`, `\splitbotmarks` (multiple marks)
- `\firstmarks`, `\splitfirstmarks` (multiple marks)
- `\topmarks` (multiple marks)

3.2 LaTeX and PRoTE

Second, the primitives required to support LaTeX were added using Thierry Larondes implementation of PRoTE.

- `\Proteversion`, `\Proterevision` (version information)
- `\resettimer`, `\elapsedtime` (timing information)
- `\creationdate`, `\filemoddate`, `\filesize`, `\filedump`, `\mdfivesum` (file information)
- `\shellescape` (Currently only a dummy implementation.)

- `\setrandomseed`, `\randomseed`, `\normaldeviate`, `\uniformdeviate` (random numbers)
- `\expanddepth`, `\expanded` (programming)
- `\ifincsname`, `\ifprimitive` `\primitive` (programming)
- `\savepos`, `\lastxpos`, `\lastypos`, `\pageheight`, `\pagewidth` (Only dummy implementations since this information is not available to HiTEX at runtime.)
- `\strcmp` (comparing strings)

3.3 `kpathsearch` and `\input`

In Don Knuth's implementation of TEX, the `\input` primitive will add the extension `.tex` to any filename that does not have an extension. This implies that a file without extension cannot be opened as an input file. The usual engines do not add such an extension but pass the filename as given to `kpse_find_file` function. HiTEX does the same. The `kpathsearch` library will find files in a variety of directories and yes, it will also find files without extension. Using this library, or equivalent functionality, is just about mandatory for any engine that wants to process LATEX input.

4 Replacing TEX's Page Builder

TEX uses an output routine to finalize the page. The output outline takes the material which the page builder had accumulated in `box255` and attaches headers, footers, and floating material like figures, tables, and footnotes. The latter material is specified by insert nodes while headers and footers are often constructed using mark nodes. Running an output routine requires the full power of the TEX engine and is not part of the HINT viewer. Therefore, HINT replaces output routines by page templates. TEX can use different output routines for different parts of a book—for example the index might use a different output routine than the main body of text.

TEX uses insertions to describe floating content that is not necessarily displayed where it is specified. Three examples may illustrate this:

- Footnotes* are specified in the middle of the text but are displayed at the bottom of the page. Several footnotes on the same page are collected and displayed together. The page layout may specify a short rule to separate footnotes from the main text, and if there are many short footnotes, it may use two columns to display them. In extreme cases, the page layout may demand a long footnote to be split and continued on the next page.
- Illustrations may be displayed exactly where specified if there is enough room on the page, but may move to the top of the page, the bottom of the page, the top of next page, or a separate page at the end of the chapter.
- Margin notes are displayed in the margin on the same page starting at the top of the margin.

HINT uses page templates and content streams to achieve similar effects. But before I describe the page building mechanisms of HINT, let me summarize TEX's page builder.

* Like this one.

4.1 TEX's page building mechanism

TEX's page builder ignores leading glue, kern, and penalty nodes until the first box or rule node is encountered; whatsit nodes do not really contribute anything* to a page; mark nodes are recorded for later use. Once the first box, rule, or insert arrives, TEX makes copies of all parameters that influence the page building process and uses these copies. These parameters are the `page_goal` and the `page_max_depth`. Further, the variables `page_total`, `page_shrink`, `page_stretch`, `page_depth`, and `insert_penalties` are initialized to zero. The top skip adjustment is made when the first box or rule arrives—possibly after an insert. Now the page builder accumulates material: normal material goes into `box255` and will change `page_total`, `page_shrink`, `page_stretch`, and `page_depth`. The latter is adjusted so that is does not exceed `page_max_depth`.

The handling of inserts is more complex. TEX creates an insert class using `newinsert`. This reserves a number i and four registers: `box`i for the inserted material, `count`i for the magnification factor f, `dimen`i for the maximum size per page d, and `skip`i for the extra space needed on a page if there are any insertions of class i.

For example plain TEX allocates $n = 254$ for footnotes and sets `count254` to 1000, `dimen254` to 8in, and `skip254` to `\bigskipamount`.

An insertion node will specify the insertion class i, some vertical material, its natural height plus depth x, a `split_top_skip`, a `split_max_depth`, and a `floating_penalty`.

Now assume that an insert node with subtype 254 arrives at the page builder. If this is the first such insert, TEX will decrease the `page_goal` by the width of `skip254` and adds its stretchability and shrinkability to the total stretchability and shrinkability of the page. Later, the output routine will add some space and the footnote rule to fill just that much space and add just that much shrinkability and stretchability to the page. Then TEX will normally add the vertical material in the insert node to `box254` and decrease the `page_goal` by $x \times f/1000$.

Special processing is required if TEX detects that there is not enough space on the current page to accommodate the complete insertion. If already a previous insert did not fit on the page, simply the `floating_penalty` as given in the insert node is added to the total `insert_penalties`. Otherwise TEX will test that the total natural height plus depth of `box254` including x does not exceed the maximum size d and that the `page_total` + `page_depth` + $x \times f/1000$ − `page_shrink` ≤ `page_goal`. If one of these tests fails, the current insertion is split in such a way as to make the size of the remaining insertions just pass the tests just stated.

Whenever a glue node, or penalty node, or a kern node that is followed by glue arrives at the page builder, it rates the current position as a possible end of the page based on the shrinkability of the page and the difference between `page_total` and `page_goal`. As the page fills, the page breaks tend to become better and better until the page starts to get overfull and the page breaks get worse and worse until

* This changes when images are implemented as whatsit nodes.

they reach the point where they become `awful_bad`. At that point, the page builder returns to the best page break found so far and fires up the output routine.

4.2 HINT Page Templates

Let's look at the problems that show up when implementing a replacement for TEX's page building mechanism.

1. An insertion node can not always specify its height x because insertions may contain paragraphs that need to be broken in lines and the height of a paragraph depends in some non obvious way on its width.
2. Before the viewer can compute the height x, it needs to know the width of the insertion. Just imagine displaying footnotes in two columns or setting notes in the margin. Knowing the width, it can pack the vertical material and derive its height and depth.
3. TEX's plain format provides an insert macro that checks whether there is still space on the current page, and if so, it creates a contribution to the main text body, otherwise it creates a `topinsert`. Such a decision needs to be postponed to the HINT viewer.
4. HINT has no output routines that would specify something like the space and the rule preceding the footnote.
5. TEX's output routines have the ability to inspect the content of the boxes, split them, and distribute the content over the page. For example, the output routine for an index set in two column format might expect a box containing index entries up to a height of $2 \times$ `vsize`. It will split this box in the middle and display the top part in the left column and the bottom part in the right column. With this approach, the last page will show two partly filled columns of about equal size.
6. HINT has no mark nodes that could be used to create page headers or footers. Marks, like output routines, contain token lists and need the full TEX interpreter for processing them. Hence, HINT does not support mark nodes.

Instead of output routines, HINT uses page templates. Page templates are basically vertical boxes with ⟨*stream insert points*⟩ marking the positions where the content of the box registers, filled by the page builder, should appear. To output the page, the viewer traverses the page template, replaces the placeholders by the appropriate box content, and sets the glue.

It is only natural to treat the page's main body, inserts, and marks using the same mechanism. We call this mechanism a content stream. Content streams are identified by a stream number in the range 0 to 254; the number 255 is used to indicate an invalid stream number. The stream number 0 is reserved for the main content stream; it is always defined.

> It is planed to implement a replacement for TEX's mark nodes using different types of streams:

- normal streams correspond to TEX's inserts and accumulate content on the page,
- first streams correspond to TEX's first marks and will contain only the first insertion of the page,
- last streams correspond to TEX's bottom marks and will contain only the last insertion of the page, and
- top streams correspond to TEX's top marks. Top streams are not yet implemented.

Nodes from the content section are considered contributions to stream 0 except for insert nodes which will specify the stream number explicitly. If the stream is not defined or is not used in the current page template, its content is simply ignored.

The page builder needs a mechanism to redirect contributions from one content stream to another content stream based on the availability of space. Hence a HINT content stream can optionally specify a preferred stream number, where content should go if there is still space available, a next stream number, where content should go if the present stream has no more space available, and a split ratio if the content is to be split between these two streams before filling in the template.

Various stream parameters govern the treatment of contributions to the stream and the page building process.

- The magnification factor f: Inserting a box of height h to this stream will contribute $h \times f/1000$ to the height of the page under construction. For example, a stream that uses a two column format will have an f value of 500; a stream that specifies notes that will be displayed in the page margin will have an f value of zero.
- The height h: The extended dimension h gives the maximum height this stream is allowed to occupy on the current page. To continue the previous example, a stream that will be split into two columns will have $h = 2 \cdot$ `vsize` , and a stream that specifies notes that will be displayed in the page margin will have $h = 1 \cdot$ `vsize`. You can restrict the amount of space occupied by footnotes to the bottom quarter by setting the corresponding h value to $h = 0.25 \cdot$ `vsize`.
- The depth d: The dimension d gives the maximum depth this stream is allowed to have after formatting.
- The width w: The extended dimension w gives the width of this stream when formatting its content. For example margin notes should have the width of the margin less some surrounding space.
- The "before" list b: If there are any contributions to this stream on the current page, the material in list b is inserted *before* the material from the stream itself. For example, the short line that separates the footnotes from the main page will go, together with some surrounding space, into the list b.

- The top skip glue g: This glue is inserted between the material from list b and the first box of the stream, reduced by the height of the first box. Hence it specifies the distance between the material in b and the first baseline of the stream content.
- The "after" list a: The list a is treated like list b but its material is placed *after* the material from the stream itself.
- The "preferred" stream number p: If $p \neq 255$, it is the number of the *preferred* stream. If stream p has still enough room to accommodate the current contribution, move the contribution to stream p, otherwise keep it. For example, you can move an illustration to the main content stream, provided there is still enough space for it on the current page, by setting $p = 0$.
- The "next" stream number n: If $n \neq 255$, it is the number of the *next* stream. If a contribution can not be accommodated in stream p nor in the current stream, treat it as an insertion to stream n. For example, you can move contributions to the next column after the first column is full, or move illustrations to a separate page at the end of the chapter.
- The split ratio r: If r is positive, both p and n must be valid stream numbers and contents is not immediately moved to stream p or n as described before. Instead the content is kept in the stream itself until the current page is complete. Then, before inserting the streams into the page template, the content of this stream is formatted as a vertical box, the vertical box is split into a top fraction and a bottom fraction in the ratio $r/1000$ for the top and $(1000-r)/1000$ for the bottom, and finally the top fraction is moved to stream p and the bottom fraction to stream n. You can use this feature for example to implement footnotes arranged in two columns of about equal size. By collecting all the footnotes in one stream and then splitting the footnotes with $r = 500$ before placing them on the page into a right and left column. Even three or more columns can be implemented by cascades of streams using this mechanism.

HINT allows multiple page templates but HiTEX currently does not implement restricting them to individual page ranges and the viewer selects the page template with the highest priority. To support different output media, the page templates are named and a suitable user interface may offer the user a selection of possible page layouts. In this way, the page layout remains in the hands of the book designer, and the user has still the opportunity to pick a layout that best fits the display device.

The build-in page template with number 0 is always defined and has priority 0. It will display just the main content stream. It puts a small margin of `hsize`/8−4.5pt all around it. Given a letter size page, 8.5 inch wide, this formula yields a margin of 1 inch, matching TEX's plain format. The margin will be positive as long as the page is wider than 1/2 inch. For narrower pages, there will be no margin at all. In general, the HINT viewer will never set `hsize` larger than the width of the page and `vsize` larger than its height.

Index