The DVI Driver Standard, Level 0

The TUG DVI Driver Standards Committee

Abstract

The TUG DVI Driver Standard defines functional and interface requirements for computer programs (DVI processors) that read and translate files in the DVI page description language. This document is the subset of the DVI standard (level 0) applying to minimally functional DVI processors. The specifications here should be considered a minimum requirement; developers are encouraged to write drivers exceeding these specifications.

(The version of the Level 0 Standard presented here is draft 0.05. It has been reviewed by the TUG DVI Driver Standards Committee and is now being presented to the TUG membership at large for review.)

The complete standard will be presented as a series of "tiers" requiring increasingly stringent control over the output of DVI processors.

1 Purpose of the level-0 standard

The level-0 standard (henceforth called *standard*) is meant to be a base standard to which all DVI-processing programs must adhere. It provides a base level of support for both DVI-to-output-device translators (so called *drivers*) and DVI-to-DVI preprocessors (*e.g.,* `dviselect`). The standard hereafter calls such DVI-processing programs "DVI *processors*" or just "*processors.*" This standard allows all reasonable documents to be rendered (i.e., printed or displayed) accurately. When we refer to accurate rendering, we mean that the when the data generated by the DVI processor(s) are transmitted to a output device the latter shall produce a page accurately depicting the page described by the DVI file (disregarding resolution effects and output technology).

The basis for many of the specifications in this standard is the possible output of TeX82 although some requirements are based on assumptions that cannot occur with TeX82-based output; functions which can be implemented via a pre-processor are generally omitted (*e.g.,* page selection and sorting).

2 The DVI file

As a rule, DVI processors must be able to read and *interpret* any valid DVI file as specified in appendix A. They shall also correctly *render* any DVI file which falls within the following limits. If these requirements cannot be met due to limitations of the computer or the output device they shall be fulfilled as completely as possible and the limitations documented. Aside from this exception, these specifications are a *minimum*; good processors will probably be able to handle DVI files exceeding these limits (DVI files which exceed the limits are likely to be rare, but might still occur).

Explanation: This exception above is necessary because certain popular output devices have varying capacity depending on the amount of on-board memory or similar conditions. For example, an HP LaserJet Plus with 512 KB of memory is capable of holding in memory only 3056 distinct downloaded characters; a full page bitmap is also not possible with this configuration.

2.1 DVI commands

The DVI processor must be able to interpret every DVI command listed in Appendix A.

Explanation: Some commands, *e.g., put4*, are generally used for conditions outside those enumerated below; despite this, DVI-translating programs are expected to accurately interpret these commands and execute them if they do specify an action within the specified minimum limits.

2.2 Characters

2.2.1 Number of characters in a font

The DVI processor must be able to handle fonts which have characters at any code in the range $0 \le c < 256$.

Explanation: Some printers with download possibilities will require fonts with more than a given number of characters to be broken into two or more device fonts when downloaded to the printer. Please note that this requirement is not subject to the exception for device limitations of section 2.

2.2.2 Character size

The DVI processor must be able to render any character up to a size of 600 pt (horizontal) by 800 pt (vertical) unless this is not possible due to device constraints as outlined in section 2.

Explanation: This size is the glyph size, not the size given in the TFM files. These two sizes are not connected; especially it's important that the glyph might be outside the bounding box given by the dimensions of the TFM files.

2.2.3 Number of characters per page

The DVI processor must be able to render a page containing as many as 20 000 characters unless this is not possible due to device constraints as outlined in section 2.

2.2.4 Unusual characters

The DVI processor must correctly render (a) characters with empty bitmaps (*e.g.,* the SLITEX fonts) including characters whose horizontal escapement is 0, (b) characters whose printable image is wider than its horizontal escapement, and (c) characters with a negative horizontal escapement.

2.3 Rules

2.3.1 Rule size

The DVI processor must be able to render rules of any size up to 600 pt (horizontal) by 800 pt (vertical) unless this is not possible due to device constraints as outlined in section 2.

2.3.2 Placement of rules on the page

The lower left corner of a rule is to be placed on the page at the location given by rounding the current DVI coordinates as indicated in section 2.6.2. The height and width of the rule is given by the formula $\lceil Kn \rceil$ where n is the dimension in DVI units and K is a constant which converts from DVI units to device units.[1]

Explanation: It's important to remember that no rule is rendered if $n \leq 0$, as specified in appendix A.

2.4 Number of rules per page

The DVI processor must be able to render a page containing as many as 1000 rules unless this is not possible due to device constraints as outlined in section 2.

2.5 Stack

The DVI processor must be able to handle DVI files whose *push*/*pop* stack is up to 100 levels deep.

2.6 Positioning on the page

2.6.1 Location of the origin

The point $(0, 0)$ in DVI coordinates is to be located at a point one inch (25.4 mm) from the top of the page and one inch (25.4 mm) from the left side of the page.

Explanation: While the default margin given in this circumstance is somewhat inconvenient for users of non-U.S.-sized paper, the advantage of having a universally standard default location of $(0, 0)$ and the widespread assumption of the given default margin in most macro packages outweighs the inconveniences. For some DVI processors (*e.g.,* screen previewers), this specification refers to a virtual page and not the physical output.

[1] Devices with aspect ratios unequal to one will need to maintain separate constants for vertical and horizontal dimensions.

2.6.2 Changes in position due to characters and rules

The definition of DVI files refers to six registers, (h, v, w, x, y, z), which hold integer values in DVI units. In practice, we also need registers *hh* and *vv*, the pixel analogs of h and v, since it is not always true that $hh = \text{pixel\_round}(h)$ or $vv = \text{pixel\_round}(v)$ where $\text{pixel\_round}(n)$ is defined as $\text{sign}(Kn) \cdot \lfloor \text{abs}(Kn) + 0.5 \rfloor$ with $\text{sign}(i)$ resulting in -1 if $i < 0$ and in 1 otherwise.

Whenever the DVI processor encounters an instruction that changes the current position, it updates h and v using pure DVI units. If the change in position is due to a command which sets a character, the processor adds the horizontal escapement value from the PK or GF file to *hh* to get the new value for *hh*.

For a horizontal movement of x DVI units from any other command, *hh* will be set to $hh + \text{pixel\_round}(x)$ if $x < \mathit{word\_space}$ for a horizontal movement to the right or if $x > -\mathit{back\_space}$ for a horizontal movement to the left. *word_space* is defined as $\mathit{space} - \mathit{space\_shrink}$, and *back_space* is defined as *0.9quad* if the processors uses TFM files. If the processors does not use TFM files the design size of the current font in the DVI file (after all necessary magnifications have been applied) may be used for a *quad*, and *word_space* may be approximated by *0.2quad*. If x exceeds the bounds outlined above, *hh* is set to be $\text{pixel\_round}(h+x)$. In this way, rounding errors are absorbed by interword spaces.

For a vertical movement of y DVI units, *vv* is set similarly except that *vv* is set to $vv + \text{pixel\_round}(y)$ if $-0.8\mathit{quad} < y < 0.8\mathit{quad}$ and set to $\text{pixel\_round}(v + y)$ otherwise. This allows vertical rounding errors to be absorbed in the interline spacing while still allowing fractions and super- and subscripts to be printed consistently.

After any horizontal movement, a final check is made as to whether $\mathit{dist} > \mathit{max\_drift}$ with *dist* defined as $\text{abs}(hh - \text{pixel\_round}(Kh))$. If it is, then *hh* is set to $\text{pixel\_round}(Kh) + \text{sign}(\mathit{dist}) \cdot \mathit{max\_drift}$. A similar check is made with *vv* and *v*. *max_drift* should be set to 2 for output devices with device units smaller than or equal to 0.005 in (0.127 mm), 1 for output devices with device units greater than 0.005 in (0.127 mm) but less than or equal to 0.01 in (0.254 mm) and 0 for output devices with device units greater than 0.01 in (0.254 mm).

Explanation: This method for tracking the positions is oriented towards the typesetting of text. It does not fix positioning problems with lines consisting completely of characters of a fixed-width font, where one line consists only of characters without any movements and the next line contains movements. Other problematic areas are the creation of line graphics with fonts with line segments. These line segments may not align.

2.6.3 Range of movement

The DVI processor should be able to handle movements in the DVI file up to a total of $2^{31} - 1$ DVI units in any direction from the origin.

2.6.4 Objects off the page

Any printable object which would lie entirely off the physical page should not be rendered but any changes to positioning should still be taken into consideration. Any printable object which would lie partially off the physical page should either be clipped so that portion of the object that lies off the page is not printed or omitted entirely, unless this is not possible due to device constraints as outlined in section 2.

Explanation: Because some output devices do unpredictable things when objects are rendered partially or completely off the edge of the page, it is up to the DVI processor writer to make sure that objects printed partially off the page are handled correctly.

2.7 Fonts

2.7.1 Font numbers

The DVI processor must be able to accept font numbers (the parameter k given by a *fnt_def* command) in the range $0 \le k < 256$.

2.7.2 Distinct fonts

The DVI processor must be able to handle any document containing 64 or fewer distinct fonts.

2.8 Specials

Specials are the parameters to the DVI commands *xxx1*, *xxx2*, *xxx3*, and *xxx4*. This standard does not define the meaning of any special, future tiers may. Specials not officially defined by the DVI processor standards committee should be flagged with a warning when read from the DVI file. If any specials are encountered that are ignored by the processor, the processor must issue a warning message. These warning messages may optionally be turned off at run time.

3 Configuration

It must be possible for the installer of a DVI processor to configure such things as the location and naming scheme of fonts, default paper size, etc. without having to recompile or relink the processor.

Explanation: "etc." means "make as many things configurable as possible." This should be more detailled (Hint due to Karl Berry).

4 Font files

4.1 Font formats

The DVI processor must be able to read PK fonts with the location specifiable at run time. The PK format is given in appendix C. GF support is optional. The GF format is given in appendix B.

Explanation: The PK format is the preferred format for bitmap fonts because (a) it is the most compact format in the TeX world and (b) included in the PK format are pieces of information about the font (*e.g.,* the horizontal escapement in pixels for each character) which are essential for fulfilling the typesetting requirements of section 2.6.2.

4.2 The scaling number

The magnification and resolution of a font are combined into a scaling number in one of two ways:

Resolution number The resolution number is given by *resolution* × *magnification* where both values are as above. This is the preferred specification for GF and PK files.

Magnification number The magnification number is given by 5 × *resolution* × *magnification* where the resolution is given in dots per inch (on devices with a aspect ratio unequal to one, the horizontal resolution should be used) and a magnification of 1 indicates normal sizing.

4.3 Magnifications

4.3.1 Minimum set of magnifications

The DVI processor must be able to use at least fonts at the following magnifications of its target resolution: 1.0 (`magstep0`), 1.095 (`magstep0.5`), 1.2 (`magstep1`), 1.44 (`magstep2`), 1.728 (`magstep3`), 2.074 (`magstep4`), 2.488 (`magstep5`), 2.986 (`magstep6`), 3.583 (`magstep7`), 4.300 (`magstep8`), and 5.160 (`magstep9`).

Explanation: The term `magstep`*n* stems from the TeX and METAFONT control sequence with the same name. It's meaning is 1.2^n.

This list should not be taken as an exhaustive list. DVI processor authors are encouraged to support all possible magnifications.

4.3.2 Margin of error

If a DVI file requests a font at a size that does not exist, but the requested size is within 0.2 % of a supported magnification with the font at that size existing, the DVI processor must use the latter font without warning.

Explanation: TeX and METAFONT compute font magnifications with different precisions. Further, calculations done by TeX and/or a DVI processor are subject to roundoff errors. The margin prescribed is sufficient for accomodating most of these errors. It is *not* intended to compensate for fonts requested at an incorrect size.

4.4 Missing fonts

If a font is missing the DVI processor must continue processing and, after issuing an appropriate warning message, deal with the missing font in one of three ways:

1. Insert appropriate white space where characters of the font would appear.
2. Insert black rectangles of the size of the character given in the TFM file for the font.
3. Print the characters from that font at a different size or from another font at the same size.

If methods 1 or 2 are used and the processor is unable to locate size information for the font in question, then the processor may simply ignore any character setting command that occur while the current font is that font.

Under no circumstances should a missing font cause a fatal error.

A Device-Independent File Format

A.1 Introduction

The form of DVI files was designed by David R. Fuchs in 1979. Almost any reasonable typesetting device can be driven by a program that takes DVI files as input, and a lot of such DVI-to-whatever programs have been written. Thus, it is possible to print the output of document compilers like TeX on many different kinds of equipment.

A DVI file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the '*set_rule*' command has two parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two's complement notation. For example, a two-byte-long distance parameter has a value between -2^{15} and $2^{15}-1$.

A DVI file consists of a "preamble," followed by a sequence of one or more "pages," followed by a "postamble." The preamble is simply a *pre* command, with its parameters that define the dimensions used in the file; this must come first. Each "page" consists of a *bop* command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an *eop* command. The pages appear in the order that they were generated, not in any particular numerical order. If we ignore *nop* commands and *fnt_def* commands (which are allowed between any two commands in the file), each *eop* command is immediately followed by a *bop* command, or by a *post* command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in DVI commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a *bop* command points to the previous *bop*; this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a DVI file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the *bop* that starts in byte 1000 points to 100 and the *bop* that starts in byte 2000 points to 1000. (The very first *bop*, i.e., the one that starts in byte 100, has a pointer of -1.)

The DVI format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a DVI-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font f is an integer; this value is changed only by *fnt* and *fnt_num* commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, h and v. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of (h, v) would be $(h, -v)$. (c) The current spacing amounts are given by four numbers w, x, y, and z, where w and x are used for horizontal spacing and where y and z are used for vertical spacing. (d) There is a stack containing (h, v, w, x, y, z) values; the DVI commands *push* and *pop* are used to change the current level of operation. Note that the current font f is not pushed and popped; the stack contains only information about positioning.

The values of h, v, w, x, y, and z are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing h by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below.

A.2 Summary of DVI commands

Here is a list of all the commands that may appear in a DVI file. Each command is specified by its symbolic name (*e.g.*, *bop*), its opcode byte (*e.g.*, 139), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '$p[4]$' means that parameter p is four bytes long.

set_char_0 0
Typeset character number 0 from font f such that the reference point of the character is at (h, v). Then increase h by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that h will advance after this command; but h usually does increase.

set_char_1 through *set_char_127* (opcodes 1 to 127)
Do the operations of *set_char_0*; but use the character whose number matches the opcode, instead of character 0.

set1 128 $c[1]$
Same as *set_char_0*, except that character number c is typeset. TEX82 uses this command for characters in the range $128 \le c < 256$.

set2 129 $c[2]$
Same as *set1*, except that c is two bytes long, so it is in the range $0 \le c < 65536$. TEX82 never uses this command, which is intended for processors that deal with oriental languages; but a DVI processor should allow character codes greater than 255. The processor may then assume that these characters have the same width as the character whose respective codes are c mod 256.

set3 130 $c[3]$
Same as *set1*, except that c is three bytes long, so it can be as large as $2^{24} - 1$.

set4 131 $c[+4]$
Same as *set1*, except that c is four bytes long, possibly even negative. Imagine that.

set_rule 132 $a[+4]$ $b[+4]$
Typeset a solid black rectangle of height a and width b, with its bottom left corner at (h, v). Then set $h \leftarrow h + b$. If either $a \le 0$ or $b \le 0$, nothing should be typeset. Note that if $b < 0$, the value of h will decrease even though nothing else happens. Programs that typeset from DVI files should be careful to make the rules line up carefully with digitized characters, as explained in connection with the *rule_pixels* subroutine below.

put1 133 $c[1]$
Typeset character number c from font f such that the reference point of the character is at (h, v). (The 'put' commands are exactly like the 'set' commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

put2 134 $c[2]$
Same as *set2*, except that h is not changed.

put3 135 $c[3]$
Same as *set3*, except that h is not changed.

put4 136 $c[+4]$
Same as *set4*, except that h is not changed.

put_rule 137 $a[+4]$ $b[+4]$
Same as *set_rule*, except that h is not changed.

nop 138
No operation, do nothing. Any number of *nop*'s may occur between DVI commands, but a *nop* cannot be inserted between a command and its parameters or between two parameters.

bop 139 $c_0[+4]$ $c_1[+4]$ $\ldots$ $c_9[+4]$ $p[+4]$
Beginning of a page: Set $(h, v, w, x, y, z) \leftarrow (0, 0, 0, 0, 0, 0)$ and set the stack empty. Set the current font f to an undefined value. The ten c_i parameters can be used to identify pages, if a user wants to print only part of a DVI file; TEX82 gives them the values of `\count0` ... `\count9` at the time `\shipout` was invoked for this page. The parameter p points to the previous *bop* command in the file, where the first *bop* has $p = -1$.

eop 140
End of page: Print what you have read since the previous *bop*. At this point the stack should be empty.

push 141
Push the current values of (h, v, w, x, y, z) onto the top of the stack; do not change any of these values. Note that f is not pushed.

pop 142
Pop the top six values off of the stack and assign them to (h, v, w, x, y, z). The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a *pop* command.

right1 143 $b[+1]$
Set $h \leftarrow h + b$, i.e., move right b units. The parameter is a signed number in two's complement notation, $-128 \le b < 128$; if $b < 0$, the reference point actually moves left.

right2 144 $b[+2]$
Same as *right1*, except that b is a two-byte quantity in the range $-32768 \le b < 32768$.

right3 145 $b[+3]$
Same as *right1*, except that b is a three-byte quantity in the range $-2^{23} \le b < 2^{23}$.

right4 146 $b[+4]$
Same as *right1*, except that b is a four-byte quantity in the range $-2^{31} \le b < 2^{31}$.

w0 147
Set $h \leftarrow h + w$; i.e., move right w units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how w gets particular values.

w1 148 $b[+1]$
Set $w \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current w spacing and moves right by b.

w2 149 $b[+2]$
Same as *w1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

w3 150 $b[+3]$
Same as *w1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

w4 151 $b[+4]$
Same as *w1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

x0 152
Set $h \leftarrow h + x$; i.e., move right x units. The 'x' commands are like the 'w' commands except that they involve x instead of w.

x1 153 $b[+1]$
Set $x \leftarrow b$ and $h \leftarrow h + b$. The value of b is a signed quantity in two's complement notation, $-128 \le b < 128$. This command changes the current x spacing and moves right by b.

x2 154 $b[+2]$
Same as *x1*, but b is a two-byte-long parameter, $-32768 \le b < 32768$.

x3 155 $b[+3]$
Same as *x1*, but b is a three-byte-long parameter, $-2^{23} \le b < 2^{23}$.

x4 156 $b[+4]$
Same as *x1*, but b is a four-byte-long parameter, $-2^{31} \le b < 2^{31}$.

down1 157 $a[+1]$
Set $v \leftarrow v + a$, i.e., move down a units. The parameter is a signed number in two's complement notation, $-128 \le a < 128$; if $a < 0$, the reference point actually moves up.

down2 158 $a[+2]$
Same as *down1*, except that a is a two-byte quantity in the range $-32768 \le a < 32768$.

down3 159 $a[+3]$
Same as *down1*, except that a is a three-byte quantity in the range $-2^{23} \le a < 2^{23}$.

down4 160 $a[+4]$
Same as *down1*, except that a is a four-byte quantity in the range $-2^{31} \le a < 2^{31}$.

y0 161
Set $v \leftarrow v + y$; i.e., move down y units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how y gets particular values.

y1 162 $a[+1]$
Set $y \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation, $-128 \le a < 128$. This command changes the current y spacing and moves down by a.

y2 163 $a[+2]$
Same as *y1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

y3 164 $a[+3]$
Same as *y1*, but a is a three-byte-long parameter, $-2^{23} \le a < 2^{23}$.

y4 165 $a[+4]$
Same as *y1*, but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

z0 166
Set $v \leftarrow v + z$; i.e., move down z units. The 'z' commands are like the 'y' commands except that they involve z instead of y.

z1 167 $a[+1]$
Set $z \leftarrow a$ and $v \leftarrow v + a$. The value of a is a signed quantity in two's complement notation, $-128 \le a < 128$. This command changes the current z spacing and moves down by a.

z2 168 $a[+2]$
Same as *z1*, but a is a two-byte-long parameter, $-32768 \le a < 32768$.

z3 169 $a[+3]$
Same as *z1*, but a is a three-byte-long parameter, $-2^{23} \le a < 2^{23}$.

z4 170 $a[+4]$
Same as *z1*, but a is a four-byte-long parameter, $-2^{31} \le a < 2^{31}$.

fnt_num_0 171
Set $f \leftarrow 0$. Font 0 must previously have been defined by a *fnt_def* instruction, as explained below.

fnt_num_1 through *fnt_num_63* (opcodes 172 to 234)
Set $f \leftarrow 1, \ldots, f \leftarrow 63$, respectively.

fnt1 235 $k[1]$
Set $f \leftarrow k$. TeX82 uses this command for font numbers in the range $64 \le k < 256$.

fnt2 236 $k[2]$
Same as *fnt1*, except that k is two bytes long, so it is in the range $0 \le k < 65536$. TeX82 never generates this command, but large font numbers may prove useful for specifications of color or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

fnt3 237 $k[3]$
Same as *fnt1*, except that k is three bytes long, so it can be as large as $2^{24} - 1$.

fnt4 238 $k[+4]$
Same as *fnt1*, except that k is four bytes long; this is for the really big font numbers (and for the negative ones).

xxx1 239 $k[1]$ $x[k]$
This command is undefined in general; it functions as a $(k+2)$-byte *nop* unless special `DVI`-reading programs are being used. TeX82 generates *xxx1* when a short enough `\special` appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$
Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]\ x[k]$
Like *xxx1*, but $0 \le k < 2^{24}$.

xxx4 242 $k[4]\ x[k]$
Like *xxx1*, but k can be ridiculously large. TEX82 uses *xxx4* when *xxx1* would be incorrect.

fnt_def1 243 $k[1]\ c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l]$
Define font k, where $0 \le k < 256$; font definitions will be explained shortly.

fnt_def2 244 $k[2]\ c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l]$
Define font k, where $0 \le k < 65536$.

fnt_def3 245 $k[3]\ c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l]$
Define font k, where $0 \le k < 2^{24}$.

fnt_def4 246 $k[+4]\ c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l]$
Define font k, where $-2^{31} \le k < 2^{31}$.

pre 247 $i[1]\ num[4]\ den[4]\ mag[4]\ k[1]\ x[k]$
Beginning of the preamble; this must come at the very beginning of the file. Parameters i, *num*, *den*, *mag*, k, and x are explained below.

post 248
Beginning of the postamble, see below.

post_post 249
Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

A.3 The preamble

The preamble contains basic information about the file as a whole. As stated above, there are six parameters:

$$i[1]\ num[4]\ den[4]\ mag[4]\ k[1]\ x[k].$$

The i byte identifies `DVI` format; currently this byte is always set to 2. (The value $i = 3$ is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting. Some day we will set $i = 4$, when `DVI` format makes another incompatible change—perhaps in the year 2048.)

The next two parameters, *num* and *den*, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the `DVI` file could be multiplied in order to get lengths in units of 10^{-7} meters. (For example, there are exactly 7227 TEX points in 254 centimeters, and TEX82 works with scaled points where there are 2^{16} sp in a point, so TEX82 sets $num = 25400000$ and $den = 7227 \cdot 2^{16} = 473628672$.)

The *mag* parameter is what TEX82 calls `\mag`, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore $mn/1000d$. Note that if a TEX source document does not call for any '`true`' dimensions, and if you change it only by specifying a different `\mag` setting, the `DVI` file that TEX creates will be completely unchanged except for the value of *mag* in the preamble and postamble. (Fancy `DVI`-reading programs allow users to override the *mag* setting when a `DVI` file is being printed.)

Finally, k and x allow the `DVI` writer to include a comment, which is not interpreted further. The length of comment x is k, where $0 \le k < 256$.

A.4 Font definitions

Font definitions for a given font number k contain further parameters

$$c[4]\ s[4]\ d[4]\ a[1]\ l[1]\ n[a+l].$$

The four-byte value c is the check sum that TEX (or whatever program generated the `DVI` file) found in the `TFM` file for this font; c should match the check sum of the font found by programs that read this `DVI` file.

Parameter s contains a fixed-point scale factor that is applied to the character widths in font k; font dimensions in `TFM` files and other font files are relative to this quantity, which is always positive and less than 2^{27}. It is given in the same units as the other dimensions of the `DVI` file. Parameter d is similar to s; it is the "design size," and (like s) it is given in `DVI` units. Thus, font k is to be used at $mag \cdot s/1000d$ times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length $a + l$. The number a is the length of the "area" or directory, and l is the length of the font name itself; the standard local system font area is supposed to be used when $a = 0$. The n field contains the area in its first a bytes.

Font definitions must appear before the first use of a particular font number. Once font k is defined, it must not be defined again; however, we shall see below that font definitions appear in the postamble as well as in the pages, so in this sense each font number is defined exactly twice, if at all. Like *nop* commands, font definitions can appear before the first *bop*, or between an *eop* and a *bop*.

A.5 The postamble

The last page in a `DVI` file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that TEX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The postamble has the form

post $p[4]\ num[4]\ den[4]\ mag[4]\ l[4]\ u[4]\ s[2]\ t[2]$
$\langle$ font definitions $\rangle$
post_post $q[4]\ i[1]$ 223's$[\ge 4]$

Here p is a pointer to the final *bop* in the file. The next three parameters, *num*, *den*, and *mag*, are duplicates of the quantities that appeared in the preamble.

Parameters l and u give respectively the height-plus-depth of the tallest page and the width of the widest page, in the same units as other dimensions of the file. These numbers might be used by a `DVI`-reading program to position individual "pages" on large sheets of film or paper; however, the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design `DVI`-to-printer software that attempts cleverly to center the output; a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore l and u are often ignored.

Parameter s is the maximum stack depth (i.e., the largest excess of *push* commands over *pop* commands) needed to process this file. Then comes t, the total number of pages (*bop* commands) present.

The postamble continues with font definitions, which are any number of *fnt_def* commands as described above, possibly interspersed with *nop* commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a *fnt* command, and once in the postamble.

The last part of the postamble, following the *post_post* byte that signifies the end of the font definitions, contains q, a pointer to the *post* command that started the postamble. An identification byte, i, comes next; this currently equals 2, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., *´337* in octal). TEX puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a DVI file makes it feasible for DVI-reading programs to find the postamble first, on most computers, even though TEX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the DVI reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the DVI reader discovers all the information needed for typesetting the pages. Note that it is also possible to skip through the DVI file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since DVI files used in production jobs tend to be large.

B Generic Font File Format

B.1 Introduction

The most important output produced by a typical run of METAFONT is the "generic font" (GF) file that specifies the bit patterns of the characters that have been drawn. The term *generic* indicates that this file format doesn't match the conventions of any name-brand manufacturer; but it is easy to convert GF files to the special format required by almost all digital phototypesetting equipment. There's a strong analogy between the DVI files written by TEX and the GF files written by METAFONT; and, in fact, the file formats have a lot in common.

A GF file is a stream of 8-bit bytes that may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the '*boc*' (beginning of character) command has six parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters can be either positive or negative, hence they range in value from -2^{31} to $2^{31}-1$. As in TFM files, numbers that occupy more than one byte position appear in BigEndian order, and negative numbers appear in two's complement notation.

A GF file consists of a "preamble," followed by a sequence of one or more "characters," followed by a "postamble." The preamble is simply a *pre* command, with its parameters that introduce the file; this must come first. Each "character" consists of a *boc* command, followed by any number of other commands that specify "black" pixels, followed by an *eoc* command. The characters appear in the order that METAFONT generated them. If we ignore no-op commands (which are allowed between any two commands in the file), each *eoc* command is immediately followed by a *boc* command, or by a *post* command; in the latter case, there are no more characters in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in GF commands are "pointers." These are four-byte quantities that give the location number of some other byte in the file; the first file byte is number 0, then comes number 1, and so on.

The GF format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information relative instead of absolute. When a GF-reading program reads the commands for a character, it keeps track of two quantities: (a) the current column number, m; and (b) the current row number, n. These are 32-bit signed integers, although most actual font formats produced from GF files will need to curtail this vast range because of practical limitations. (METAFONT output will never allow $|m|$ or $|n|$ to get extremely large, but the GF format tries to be more general.)

How do GF's row and column numbers correspond to the conventions of TEX and METAFONT? Well, the "reference point" of a character, in TEX's view, is considered to be at the lower left corner of the pixel in row 0 and column 0. This point is the intersection of the baseline with the left edge of the type; it corresponds to location $(0,0)$ in METAFONT programs. Thus the pixel in GF row 0 and column 0 is METAFONT's unit square, comprising the region of the plane whose coordinates both lie between 0 and 1. The pixel in GF row n and column m consists of the points whose METAFONT coordinates (x,y) satisfy $m \le x \le m+1$ and $n \le y \le n+1$. Negative values of m and x correspond to columns of pixels *left* of the reference point; negative values of n and y correspond to rows of pixels *below* the baseline.

Besides m and n, there's also a third aspect of the current state, namely the *paint_switch*, which is always either *black* or *white*. Each *paint* command advances m by a specified amount d, and blackens the intervening pixels if *paint_switch* = *black*; then the *paint_switch* changes to the opposite state. GF's commands are designed so that m will never decrease within a row, and n will never increase within a character; hence there is no way to whiten a pixel that has been blackened.

B.2 Summary of GF commands

Here is a list of all the commands that may appear in a GF file. Each command is specified by its symbolic name (*e.g.*, *boc*), its opcode byte (*e.g.*, 67), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '$d[2]$' means that parameter d is two bytes long.

paint_0 0
: This is a *paint* command with $d = 0$; it does nothing but change the *paint_switch* from *black* to *white* or vice versa.

paint_1 through *paint_63* (opcodes 1 to 63)
: These are *paint* commands with $d = 1$ to 63, defined as follows: If *paint_switch* = *black*, blacken d pixels of the current row n, in columns m through $m + d - 1$ inclusive. Then, in any case, complement the *paint_switch* and advance m by d.

paint1 64 $d[1]$
: This is a *paint* command with a specified value of d; METAFONT uses it to paint when $64 \le d < 256$.

paint2 65 $d[2]$
: Same as *paint1*, but d can be as high as 65535.

paint3 66 $d[3]$
: Same as *paint1*, but d can be as high as $2^{24} - 1$. METAFONT never needs this command, and it is hard to imagine anybody making practical use of it; surely a more compact encoding will be desirable when characters can be this large. But the command is there, anyway, just in case.

boc 67 $c[+4]$ $p[+4]$ $min\_m[+4]$ $max\_m[+4]$ $min\_n[+4]$ $max\_n[+4]$
: Beginning of a character: Here c is the character code, and p points to the previous character beginning (if any) for characters having this code number modulo 256. (The pointer p is -1 if there was no prior character with an equivalent code.) The values of registers m and n defined by the instructions that follow for this character must satisfy $min\_m \le m \le max\_m$ and $min\_n \le n \le max\_n$. (The values of max_m and min_n need not be the tightest bounds possible.) When a GF-reading program sees a *boc*, it can use min_m, max_m, min_n, and max_n to initialize the bounds of an array. Then it sets $m \leftarrow min\_m$, $n \leftarrow max\_n$, and $paint\_switch \leftarrow white$.

boc1 68 $c[1]$ $del\_m[1]$ $max\_m[1]$ $del\_n[1]$ $max\_n[1]$
: Same as *boc*, but p is assumed to be -1; also $del\_m = max\_m - min\_m$ and $del\_n = max\_n - min\_n$ are given instead of min_m and min_n. The one-byte parameters must be between 0 and 255, inclusive. (This abbreviated *boc* saves 19 bytes per character, in common cases.)

eoc 69
: End of character: All pixels blackened so far constitute the pattern for this character. In particular, a completely blank character might have *eoc* immediately following *boc*.

skip0 70
: Decrease n by 1 and set $m \leftarrow min\_m$, $paint\_switch \leftarrow white$. (This finishes one row and begins another, ready to whiten the leftmost pixel in the new row.)

skip1 71 $d[1]$
: Decrease n by $d + 1$, set $m \leftarrow min\_m$, and set $paint\_switch \leftarrow white$. This is a way to produce d all-white rows.

skip2 72 $d[2]$
: Same as *skip1*, but d can be as large as 65535.

skip3 73 $d[3]$
: Same as *skip1*, but d can be as large as $2^{24} - 1$. METAFONT obviously never needs this command.

new_row_0 74
: Decrease n by 1 and set $m \leftarrow min\_m$, $paint\_switch \leftarrow black$. (This finishes one row and begins another, ready to *blacken* the leftmost pixel in the new row.)

new_row_1 through *new_row_164* (opcodes 75 to 238)
: Same as *new_row_0*, but with $m \leftarrow min\_m + 1$ through $min\_m + 164$, respectively.

xxx1 239 $k[1]$ $x[k]$
: This command is undefined in general; it functions as a $(k+2)$-byte *no_op* unless special GF-reading programs are being used. METAFONT generates *xxx* commands when encountering a **special** string; this occurs in the GF file only between characters, after the preamble, and before the postamble. However, *xxx* commands might appear anywhere in GF files generated by other processors. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2]$ $x[k]$
: Like *xxx1*, but $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$
: Like *xxx1*, but $0 \le k < 2^{24}$. METAFONT uses this when sending a **special** string whose length exceeds 255.

xxx4 242 $k[4]$ $x[k]$
: Like *xxx1*, but k can be ridiculously large; k mustn't be negative.

yyy 243 $y[+4]$
: This command is undefined in general; it functions as a 5-byte *no_op* unless special GF-reading programs are being used. METAFONT puts *scaled* numbers into *yyy*'s, as a result of **numspecial** commands; the intent is to provide numeric parameters to *xxx* commands that immediately precede.

no_op 244
: No operation, do nothing. Any number of *no_op*'s may occur between GF commands, but a *no_op* cannot be inserted between a command and its parameters or between two parameters.

char_loc 245 $c[1]$ $dx[+4]$ $dy[+4]$ $w[+4]$ $p[+4]$
: This command will appear only in the postamble, which will be explained shortly.

char_loc0 246 $c[1]$ $dm[1]$ $w[+4]$ $p[+4]$
: Same as *char_loc*, except that dy is assumed to be zero, and the value of dx is taken to be $65536 * dm$, where $0 \le dm < 256$.

pre 247 $i[1]$ $k[1]$ $x[k]$
: Beginning of the preamble; this must come at the very beginning of the file. Parameter i is an identifying number for GF format, currently 131. The other information is merely commentary; it is not given special interpretation like *xxx* commands are. (Note that *xxx* commands may immediately follow the preamble, before the first *boc*.)

post 248
: Beginning of the postamble, see below.

post_post 249
: Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

B.3 The postamble

The last character in a GF file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that METAFONT has accumulated. The postamble has the form

post $p[4]$ $ds[4]$ $cs[4]$ $hppp[4]$ $vppp[4]$ $min\_m[+4]$ $max\_m[+4]$ $min\_n[+4]$ $max\_n[+4]$
⟨character locators⟩
post_post $q[4]$ $i[1]$ 223's$[\ge 4]$

Here p is a pointer to the byte following the final *eoc* in the file (or to the byte following the preamble, if there are no characters); it can be used to locate the beginning of *xxx* commands that might have preceded the postamble. The *ds* and *cs* parameters give the design size and check sum, respectively, which are exactly the values put into the header of any TFM file that shares information with this GF file. Parameters *hppp* and *vppp* are the ratios of pixels per point, horizontally and vertically, expressed as *scaled* integers (i.e., multiplied by 2^{16}); they can be used to correlate the font with specific device resolutions, magnifications, and "at sizes." Then come *min_m*, *max_m*, *min_n*, and *max_n*, which bound the values that registers m and n assume in all characters in this GF file. (These bounds need not be the best possible; *max_m* and *min_n* may, on the other hand, be tighter than the similar bounds in *boc* commands. For example, some character may have $min\_n = -100$ in its *boc*, but it might turn out that n never gets lower than -50 in any character; then *min_n* can have any value ≤ -50. If there are no characters in the file, it's possible to have $min\_m > max\_m$ and/or $min\_n > max\_n$.)

Character locators are introduced by *char_loc* commands, which specify a character residue c, character escapements (dx, dy), a character width w, and a pointer p to the beginning of that character. (If two or more characters have the same code c modulo 256, only the last will be indicated; the others can be located by following backpointers. Characters whose codes differ by a multiple of 256 are assumed to share the same font metric information, hence the TFM file contains only residues of character codes modulo 256. This convention is intended for oriental languages, when there are many character shapes but few distinct widths.)

The character escapements (dx, dy) are the values of METAFONT's **chardx** and **chardy** parameters; they are in units of *scaled* pixels; i.e., dx is in horizontal pixel units times 2^{16}, and dy is in vertical pixel units times 2^{16}. This is the intended amount of displacement after typesetting the character; for DVI files, dy should be zero, but other document file formats allow nonzero vertical escapement.

The character width w duplicates the information in the TFM file; it is 2^{24} times the ratio of the true width to the font's design size.

The backpointer p points to the character's *boc*, or to the first of a sequence of consecutive *xxx* or *yyy* or *no_op* commands that immediately precede the *boc*, if such commands exist; such "special" commands essentially belong to the characters, while the special commands after the final character belong to the postamble (i.e., to the font as a whole). This convention about p applies also to the backpointers in *boc* commands, even though it wasn't explained in the description of *boc*.

Pointer p might be -1 if the character exists in the TFM file but not in the GF file. This unusual situation can arise in METAFONT output if the user had $proofing < 0$ when the character was being shipped out, but then made $proofing \ge 0$ in order to get a GF file.

The last part of the postamble, following the *post_post* byte that signifies the end of the character locators, contains q, a pointer to the *post* command that started the postamble. An identification byte, i, comes next; this currently equals 131, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., "DF in hexadecimal). METAFONT puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a GF file makes it feasible for GF-reading programs to find the postamble first, on most computers, even though METAFONT wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the GF reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the GF reader can discover all the information needed for individual characters.

C Packed File Format

C.1 Introduction

The packed file format is a compact representation of the data contained in a GF file. The information content is the same, but packed (PK) files are almost always less than half the size of their GF counterparts. They are also easier to convert into a raster representation because they do not have a profusion of *paint*, *skip*, and *new_row* commands to be separately interpreted. In addition, the PK format expressedly forbids **special** commands within a character. The minimum bounding box for each character is explicit in the format, and does not need to be scanned for as in the GF format. Finally, the width and escapement values are combined with the raster information into character "packets," making it simpler in many cases to process a character.

A PK file is organized as a stream of 8-bit bytes. At times, these bytes might be split into 4-bit nybbles or single bits, or combined into multiple byte parameters. When bytes are split into smaller pieces, the 'first' piece is always the most significant of the byte. For instance, the first bit of a byte is the bit with value 128; the first nybble can be found by dividing a byte by 16. Similarly, when bytes are combined into multiple byte parameters, the first byte is the most significant of the parameter. If the parameter is signed, it is represented by two's-complement notation.

The set of possible eight-bit values is separated into two sets, those that introduce a character definition, and those that do not. The values that introduce a character definition range from 0 to 239; byte values above 239 are interpreted as commands. Bytes that introduce character definitions are called flag bytes, and various fields within the byte indicate various things about how the character definition is encoded. Command bytes have zero or more parameters, and can never appear within a character definition or between parameters of another command, where they would be interpeted as data.

A PK file consists of a preamble, followed by a sequence of one or more character definitions, followed by a postamble. The preamble command must be the first byte in the file, followed immediately by its parameters. Any number of character definitions may follow, and any command but the preamble command and the postamble command may occur between character definitions. The very last command in the file must be the postamble.

The packed file format is intended to be easy to read and interpret by device drivers. The small size of the file reduces the input/output overhead each time a font is loaded. For those drivers that load and save each font file into memory, the small size also helps reduce the memory requirements. The length of each character packet is specified, allowing the character raster data to be loaded into memory by simply counting bytes, rather than interpreting each command; then, each character can be interpreted on a demand basis. This also makes it possible for a driver to skip a particular character quickly if it knows that the character is unused.

C.2 Summary of PK commands

First, the command bytes will be presented; then the format of the character definitions will be defined. Eight of the possible sixteen commands (values 240 through 255) are currently defined; the others are reserved for future extensions. The commands are listed below. Each command is specified by its symbolic name (*e.g.*, *pk_no_op*), its opcode byte, and any parameters. The parameters are followed by a bracketed number telling how many bytes they occupy, with the number preceded by a plus sign if it is a signed quantity. (Four byte quantities are always signed, however.)

pk_xxx1 240 $k[1]$ $x[k]$
: This command is undefined in general; it functions as a $(k+2)$-byte *no_op* unless special PK-reading programs are being used. METAFONT generates *xxx* commands when encountering a **special** string. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

pk_xxx2 241 $k[2]$ $x[k]$
: Like *pk_xxx1*, but $0 \le k < 65536$.

pk_xxx3 242 $k[3]$ $x[k]$
: Like *pk_xxx1*, but $0 \le k < 2^{24}$. METAFONT uses this when sending a **special** string whose length exceeds 255.

pk_xxx4 243 $k[4]$ $x[k]$
: Like *pk_xxx1*, but k can be ridiculously large; k musn't be negative.

pk_yyy 244 $y[4]$
: This command is undefined in general; it functions as a five-byte *no_op* unless special PK reading programs are being used. METAFONT puts *scaled* numbers into *yyy*'s, as a result of **numspecial** commands; the intent is to provide numeric parameters to *xxx* commands that immediately precede.

pk_post 245
: Beginning of the postamble. This command is followed by enough *pk_no_op* commands to make the file a multiple of four bytes long. Zero through three bytes are usual, but any number is allowed. This should make the file easy to read on machines that pack four bytes to a word.

pk_no_op 246
: No operation, do nothing. Any number of *pk_no_op*'s may appear between PK commands, but a *pk_no_op* cannot be inserted between a command and its parameters, between two parameters, or inside a character definition.

pk_pre 247 $i[1]$ $k[1]$ $x[k]$ $ds[4]$ $cs[4]$ $hppp[4]$ $vppp[4]$
Preamble command. Here, i is the identification byte of the file, currently equal to 89. The string x is merely a comment, usually indicating the source of the PK file. The parameters ds and cs are the design size of the file in $1/2^{20}$ points, and the checksum of the file, respectively. The checksum should match the TFM file and the GF files for this font. Parameters $hppp$ and $vppp$ are the ratios of pixels per point, horizontally and vertically, multiplied by 2^{16}; they can be used to correlate the font with specific device resolutions, magnifications, and "at sizes."

C.3 Packing algorithms

The PK format has two conflicting goals: to pack character raster and size information as compactly as possible, while retaining ease of translation into raster and other forms. A suitable compromise was found in the use of run-encoding of the raster information. Instead of packing the individual bits of the character, we instead count the number of consecutive 'black' or 'white' pixels in a horizontal raster row, and then encode this number. Run counts are found for each row from left to right, traversing rows from the top to bottom. This is essentially the way the GF format works. Instead of presenting each row individually, however, we concatenate all of the horizontal raster rows into one long string of pixels, and encode this row. With knowledge of the width of the bit-map, the original character glyph can easily be reconstructed. In addition, we do not need special commands to mark the end of one row and the beginning of the next.

Next, we place the burden of finding the minimum bounding box on the part of the font generator, since the characters will usually be used much more often than they are generated. The minimum bounding box is the smallest rectangle that encloses all 'black' pixels of a character. We also eliminate the need for a special end of character marker, by supplying exactly as many bits as are required to fill the minimum bounding box, from which the end of the character is implicit.

Let us next consider the distribution of the run counts. Analysis of several dozen pixel files at 300 dots per inch yields a distribution peaking at four, falling off slowly until ten, then a bit more steeply until twenty, and then asymptotically approaching the horizontal. Thus, the great majority of our run counts will fit in a four-bit nybble. The eight-bit byte is attractive for our run-counts, as it is the standard on many systems; however, the wasted four bits in the majority of cases seem a high price to pay. Another possibility is to use a Huffman-type encoding scheme with a variable number of bits for each run-count; this was rejected because of the overhead in fetching and examining individual bits in the file. Thus, the character raster definitions in the PK file format are based on the four-bit nybble.

An analysis of typical pixel files yielded another interesting statistic: Fully 37 % of the raster rows were duplicates of the previous row. Thus, the PK format allows the specification of repeat counts, which indicate how many times a horizontal raster row is to be repeated. These repeated rows are taken out of the character glyph before individual rows are concatenated into the long string of pixels.

For elegance, we disallow a run count of zero. The case of a null raster description should be gleaned from the character width and height being equal to zero, and no raster data should be read. No other zero counts are ever necessary. Also, in the absence of repeat counts, the repeat value is set to be zero (only the original row is sent.) If a repeat count is seen, it takes effect on the current row. The current row is defined as the row on which the first pixel of the next run count will lie. The repeat count is set back to zero when the last pixel in the current row is seen, and the row is sent out.

This poses a problem for entirely black and entirely white rows, however. Let us say that the current row ends with four white pixels, and then we have five entirely empty rows, followed by a black pixel at the beginning of the next row, and the character width is ten pixels. We would like to use a repeat count, but there is no legal place to put it. If we put it before the white run count, it will apply to the current row. If we put it after, it applies to the row with the black pixel at the beginning. Thus, entirely white or entirely black repeated rows are always packed as large run counts (in this case, a white run count of 54) rather than repeat counts.

Now we turn our attention to the actual packing of the run counts and repeat counts into nybbles. There are only sixteen possible nybble values. We need to indicate run counts and repeat counts. Since the run counts are much more common, we will devote the majority of the nybble values to them. We therefore indicate a repeat count by a nybble of 14 followed by a packed number, where a packed number will be explained later. Since the repeat count value of one is so common, we indicate a repeat one command by a single nybble of 15. A 14 followed by the packed number 1 is still legal for a repeat one count. The run counts are coded directly as packed numbers.

For packed numbers, therefore, we have the nybble values 0 through 13. We need to represent the positive integers up to, say, $2^{31}-1$. We would like the more common smaller numbers to take only one or two nybbles, and the infrequent large numbers to take three or more. We could therefore allocate one nybble value to indicate a large run count taking three or more nybbles. We do this with the value 0.

We are left with the values 1 through 13. We can allocate some of these, say dyn_f, to be one-nybble run counts. These will work for the run counts $1, \ldots, dyn\_f$. For subsequent run counts, we will use a nybble greater than dyn_f, followed by a second nybble, whose value can run from 0 through 15. Thus, the two-nybble values will run from $dyn\_f + 1, \ldots, (13 - dyn\_f) * 16 + dyn\_f$. We have our definition of large run count values now, being all counts greater than $(13 - dyn\_f) * 16 + dyn\_f$.

We can analyze our several dozen pixel files and determine an optimal value of *dyn_f*, and use this value for all of the characters. Unfortunately, values of *dyn_f* that pack small characters well tend to pack the large characters poorly, and values that pack large characters well are not efficient for the smaller characters. Thus, we choose the optimal *dyn_f* on a character basis, picking the value that will pack each individual character in the smallest number of nybbles. Legal values of *dyn_f* run from 0 (with no one-nybble run counts) to 13 (with no two-nybble run counts).

Our only remaining task in the coding of packed numbers is the large run counts. We use a scheme suggested by D. E. KNUTH that simply and elegantly represents arbitrarily large values. The general scheme to represent an integer i is to write its hexadecimal representation, with leading zeros removed. Then we count the number of digits, and prepend one less than that many zeros before the hexadecimal representation. Thus, the values from one to fifteen occupy one nybble; the values sixteen through 255 occupy three, the values 256 through 4095 require five, etc.

For our purposes, however, we have already represented the numbers one through $(13 - \mathit{dyn\_f}) * 16 + \mathit{dyn\_f}$. In addition, the one-nybble values have already been taken by our other commands, which means that only the values from sixteen up are available to us for long run counts. Thus, we simply normalize our long run counts, by subtracting $(13 - \mathit{dyn\_f}) * 16 + \mathit{dyn\_f} + 1$ and adding 16, and then we represent the result according to the scheme above.

C.4 Decoding PK files

The final algorithm for decoding the run counts based on the above scheme might look like the Pascal routine in figure 1, assuming that a procedure called *pk_nyb* is available to get the next nybble from the file, and assuming that the global *repeat_count* indicates whether a row needs to be repeated. Note that this routine is recursive, but since a repeat count can never directly follow another repeat count, it can only be recursive to one level.

For low resolution fonts, or characters with 'gray' areas, run encoding can often make the character many times larger. Therefore, for those characters that cannot be encoded efficiently with run counts, the PK format allows bit-mapping of the characters. This is indicated by a *dyn_f* value of 14. The bits are packed tightly, by concatenating all of the horizontal raster rows into one long string, and then packing this string eight bits to a byte. The number of bytes required can be calculated by $\lfloor(\mathit{width} * \mathit{height} + 7)/8\rfloor$. This format should only be used when packing the character by run counts takes more bytes than this, although, of course, it is legal for any character. Any extra bits in the last byte should be set to zero.

At this point, we are ready to introduce the format for a character descriptor. It consists of three parts: a flag byte, a character preamble, and the raster data. The most significant four bits of the flag byte yield the *dyn_f* value for that character. (Notice that only values of 0 through 14 are legal for *dyn_f*, with 14 indicating a bit mapped character; thus, the flag bytes do not conflict with the command bytes, whose upper nybble is always 15.) The next bit (with weight 8) indicates whether the first run count is a black count or a white count, with a one indicating a black count. For bit-mapped characters, this bit should be set to a zero. The next bit (with weight 4) indicates whether certain later parameters (referred to as size parameters) are given in one-byte or two-byte quantities, with a one indicating that they are in two-byte quantities. The last two bits are concatenated on to the beginning of the packet-length parameter in the character preamble, which will be explained below.

However, if the last three bits of the flag byte are all set (normally indicating that the size parameters are two-byte values and that a 3 should be prepended to the length parameter), then a long format of the character preamble should be used instead of one of the short forms.

Therefore, there are three formats for the character preamble; the one that is used depends on the least significant three bits of the flag byte. If the least significant three bits are in the range zero through three, the short format is used. If they are in the range four through six, the extended short format is used. Otherwise, if the least significant bits are all set, then the long form of the character preamble is used. The preamble formats are explained below.

Short form *flag*[1] *pl*[1] *cc*[1] *tfm*[3] *dm*[1] *w*[1] *h*[1] *hoff*[+1] *voff*[+1].
If this format of the character preamble is used, the above parameters must all fit in the indicated number of bytes, signed or unsigned as indicated. Almost all of the standard TeX font characters fit; the few exceptions are fonts such as `cminch`.

Extended short form *flag*[1] *pl*[2] *cc*[1] *tfm*[3] *dm*[2] *w*[2] *h*[2] *hoff*[+2] *voff*[+2].
Larger characters use this extended format.

Long form *flag*[1] *pl*[4] *cc*[4] *tfm*[4] *dx*[4] *dy*[4] *w*[4] *h*[4] *hoff*[4] *voff*[4].
This is the general format which allows all of the parameters of the GF file format, including vertical escapement.

The *flag* parameter is the flag byte. The parameter *pl* (packet length) contains the offset of the byte following this character descriptor, with respect to the beginning of the *tfm* width parameter. This is given so a PK reading program can, once it has read the flag byte, packet length, and character code (*cc*), skip over the character by simply reading this many more bytes. For the two short forms of the character preamble, the last

```
function pk_packed_num: integer;
    var i, j: integer;
    begin $i \leftarrow get\_nyb$;
    if $i = 0$ then
        begin
        repeat $j \leftarrow get\_nyb$; incr(i);
        until $j \neq 0$;
        while $i > 0$ do
            begin $j \leftarrow j * 16 + get\_nyb$; decr(i);
            end;
        $pk\_packed\_num \leftarrow j - 15 + (13 - dyn\_f) * 16 + dyn\_f$;
        end
    else if $i \le dyn\_f$ then $pk\_packed\_num \leftarrow i$
        else if $i < 14$ then $pk\_packed\_num \leftarrow (i - dyn\_f - 1) * 16 + get\_nyb + dyn\_f + 1$
            else begin
                if $repeat\_count \neq 0$ then abort('Second␣repeat␣count␣for␣this␣row!');
                $repeat\_count \leftarrow 1$;   { prevent recursion more than one level }
                if $i = 14$ then ! $repeat\_count \leftarrow pk\_packed\_num$;
                send_out(true, repeat_count); $pk\_packed\_num \leftarrow pk\_packed\_num$;
                end;
    end;
```

Figure 1: Algorithm for decoding run counts in a PK file

two bits of the flag byte should be considered the two most-significant bits of the packet length. For the short format, the true packet length might be calculated as $(\mathit{flag} \bmod 4) \cdot 256 + pl$; for the short extended format, it might be calculated as $(\mathit{flag} \bmod 4) \cdot 65536 + pl$.

The w parameter is the width and the h parameter is the height in pixels of the minimum bounding box. The *dx* and *dy* parameters are the horizontal and vertical escapements, respectively. In the short formats, *dy* is assumed to be zero and *dm* is *dx* but in pixels; in the long format, *dx* and *dy* are both in pixels multiplied by 2^{16}. The *hoff* is the horizontal offset from the upper left pixel to the reference pixel; the *voff* is the vertical offset. They are both given in pixels, with right and down being positive. The reference pixel is the pixel that occupies the unit square in METAFONT; the METAFONT reference point is the lower left hand corner of this pixel. (See the example below.)

TeX requires all characters that have the same character codes modulo 256 to have also the same *tfm* widths and escapement values. The PK format does not itself make this a requirement, but in order for the font to work correctly with the TeX software, this constraint should be observed. (The standard version of TeX cannot output character codes greater than 255, but extended versions do exist.)

Following the character preamble is the raster information for the character, packed by run counts or by bits, as indicated by the flag byte. If the character is packed by run counts and the required number of nybbles is odd, then the last byte of the raster description should have a zero for its least significant nybble.

C.5 An example character

As an illustration of the PK format, the character Ξ from the font amr10 at 300 dots per inch will be encoded. This character was chosen because it illustrates some of the borderline cases. The raster for the character is shown in figure 2. The width of the minimum bounding box for this character is 20; its height is 29. The '+' represents the reference pixel; notice how it lies outside the minimum bounding box. The *hoff* value is −2, and the *voff* is 28.

The first task is to calculate the run counts and repeat counts. The repeat counts are placed at the first transition (black to white or white to black) in a row, and are enclosed in brackets. White counts are enclosed in parentheses. It is relatively easy to generate the counts list:

82 [2] (16) 2 (42) [2] 2 (12) 2 (4) [3]
16 (4) [2] 2 (12) 2 (62) [2] 2 (16) 82

Note that any duplicated rows that are not all white or all black are removed before the run counts are calculated. The rows thus removed are rows 5, 6, 10, 11, 13, 14, 15, 17, 18, 23, and 24.

The next step in the encoding of this character is to calculate the optimal value of *dyn_f*. The details of how this calculation is done are not important here; suffice it to say that there is a simple algorithm that can determine the best value of *dyn_f* in one pass over the count list. For this character, the optimal value turns out to be 8 (atypically low). Thus, all count values less than or equal to 8 are packed in one nybble; those from nine to $(13 - 8) * 16 + 8$ or 88 are packed in two nybbles.

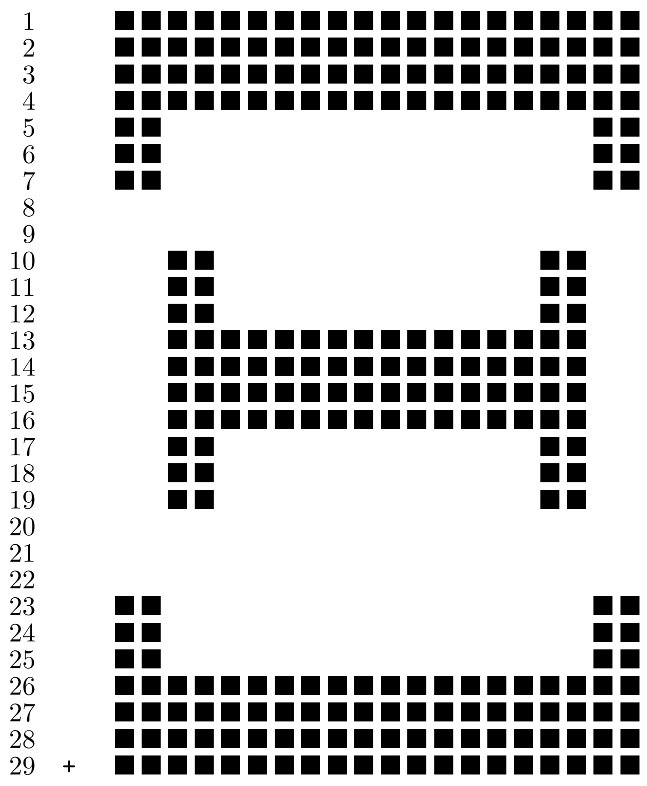

Figure 2: Character Ξ of `amr10` (the row numbers are chosen for convenience, and are not METAFONT's row numbers.)

The run encoded values now become (in hex, separated according to the above list):

```
D9 E2 97 2 B1 E2 2 93 2 4 E3
97 4 E2 2 93 2 C5 E2 2 97 D9
```

which comes to 36 nybbles, or 18 bytes. This is shorter than the 73 bytes required for the bit map, so we use the run count packing.

The short form of the character preamble is used because all of the parameters fit in their respective lengths. The packet length is therefore 18 bytes for the raster, plus eight bytes for the character preamble parameters following the character code, or 26. The *tfm* width for this character is 640796, or `9C71C` in hexadecimal. The horizontal escapement is 25 pixels. The flag byte is 88 hex, indicating the short preamble, the black first count, and the *dyn_f* value of 8. The final total character packet, in hexadecimal, is given in figure 3.

D Font metric data

D.1 Introduction

The idea behind `TFM` files is that typesetting routines like TeX need a compact way to store the relevant information about several dozen fonts, and computer centers need a compact way to store the relevant information about several hundred fonts. `TFM` files are compact, and

Flag byte	88		
Packet length	1A		
Character code	04		
tfm width	09	C7	1C
Horizontal escapement (pixels)	19		
Width of bit map	14		
Height of bit map	1D		
Horizontal offset (signed)	FE		
Vertical offset	1C		
Raster data	D9	E2	97
	2B	1E	22
	93	24	E3
	97	4E	22
	93	2C	5E
	22	97	D9

Figure 3: `PK` character packet for Ξ of `amr10`

most of the information they contain is highly relevant, so they provide a solution to the problem.

The information in a `TFM` file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words; but TEX uses the byte interpretation, and so do we. Note that the bytes are considered to be unsigned numbers.

D.2 Summary of `TFM` files

D.2.1 The header

The first 24 bytes (6 words) of a `TFM` file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

lf = length of the entire file, in words;
lh = length of the header data, in words;
bc = smallest character code in the font;
ec = largest character code in the font;
nw = number of words in the width table;
nh = number of words in the height table;
nd = number of words in the depth table;
ni = number of words in the italic correction table;
nl = number of words in the lig/kern table;
nk = number of words in the kern table;
ne = number of words in the extensible character table;
np = number of font parameter words.

They are all nonnegative and less than 2^{15}. We must have $bc - 1 \le ec \le 255$, $ne \le 256$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

D.2.2 `TFM` data

The rest of the `TFM` file may be regarded as a sequence of ten data arrays having the informal specification

header: **array** $[0 \to lh - 1]$ **of** *stuff*
char_info: **array** $[bc \to ec]$ **of** *char_info_word*
width: **array** $[0 \to nw - 1]$ **of** *fix_word*
height: **array** $[0 \to nh - 1]$ **of** *fix_word*
depth: **array** $[0 \to nd - 1]$ **of** *fix_word*
italic: **array** $[0 \to ni - 1]$ **of** *fix_word*
lig_kern: **array** $[0 \to nl - 1]$ **of** *lig_kern_command*
kern: **array** $[0 \to nk - 1]$ **of** *fix_word*
exten: **array** $[0 \to ne - 1]$ **of** *extensible_recipe*
param: **array** $[1 \to np]$ **of** *fix_word*

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two's complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048. We will see below, however, that all but one of the *fix_word* values will lie between -16 and $+16$.

The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, and for `TFM` files to be used with Xerox printing software it must contain at least 18 words, allocated as described below. When different kinds of devices need to be interfaced, it may be necessary to add further words to the header block.

header[0] is a 32-bit check sum that TEX will copy into the `DVI` output file whenever it uses the font. Later on when the `DVI` file is printed, possibly on another computer, the actual font that gets used is supposed to have a check sum that agrees with the one in the `TFM` file used by TEX. In this way, users will be warned about potential incompatibilities. (However, if the check sum is zero in either the font file or the `TFM` file, no check is made.) The actual relation between this check sum and the rest of the `TFM` file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a *fix_word* containing the design size of the font, in units of TEX points (7227 TEX points = 254 cm). This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a "10 point" font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TEX user asks for a font '`at` δ `pt`', the effect is to override the design size and replace it by δ, and to multiply the x and y coordinates of the points in the font image by a factor of δ divided by the design size. *All other dimensions in the* `TFM` *file are fix_word numbers in design-size units.* Thus, for example, the value of *param*[6], one `em` or `\quad`, is often the *fix_word* value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, *header*[1] and *param*[1] are the only *fix_word* entries in the whole `TFM` file whose first byte might be something besides 0 or 255.

header[2...11], if present, contains 40 bytes that identify the character coding scheme. The first byte, which must be between 0 and 39, is the number of subsequent ASCII bytes actually relevant in this string, which is intended to specify what character-code-to-symbol convention is present in the font. Examples are `ASCII` for standard ASCII, `TeX text` for fonts like `cmr10` and `cmti9`, `TeX math extension` for `cmex10`, `XEROX text` for

Xerox fonts, `GRAPHIC` for special-purpose non-alphabetic fonts, `UNSPECIFIED` for the default case when there is no information. Parentheses should not appear in this name. (Such a string is said to be in BCPL format.)

header[12 . . . 16], if present, contains 20 bytes that name the font family (*e.g.*, `CMR` or `HELVETICA`), in BCPL format. This field is also known as the "font identifier."

header[17], if present, contains a first byte called the *seven_bit_safe_flag*, then two bytes that are ignored, and a fourth byte called the *face*. If the value of the fourth byte is less than 18, it has the following interpretation as a "weight, slope, and expansion": Add 0 or 2 or 4 (for medium or bold or light) to 0 or 1 (for roman or italic) to 0 or 6 or 12 (for regular or condensed or extended). For example, 13 is 0+1+12, so it represents medium italic extended. A three-letter code (*e.g.*, `MIE`) can be used for such *face* data.

header[18 . . . whatever] might also be present; the individual words are simply called *header*[18], *header*[19], etc., at the moment.

Next comes the *char_info* array, which contains one *char_info_word* per character. Each *char_info_word* contains six fields packed into four bytes as follows.

first byte *width_index* (8 bits)

second byte *height_index* (4 bits) times 16, plus *depth_index* (4 bits)

third byte *italic_index* (6 bits) times 4, plus *tag* (2 bits)

fourth byte *remainder* (8 bits)

The actual width of a character is *width*[*width_index*], in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the `TFM` format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

Incidentally, the relation $width[0] = height[0] = depth[0] = italic[0] = 0$ should always hold, so that an index of zero implies a value of zero. The *width_index* should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between *bc* and *ec* and has a nonzero *width_index*.

The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.

$tag = 0$ (*no_tag*) means that *remainder* is unused.

$tag = 1$ (*lig_tag*)
means that this character has a ligature/kerning program starting at *lig_kern*[*remainder*].

$tag = 2$ (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.

$tag = 3$ (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in *exten*[*remainder*].

The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word is a *lig_kern_command* of four bytes.

first byte *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.

second byte *next_char*: "if *next_char* follows the current character, then perform the operation and stop, otherwise continue."

third byte *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.

fourth byte *remainder*.

In a kern step, an additional space equal to $kern[256(op\_byte - 128) + remainder]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a+2b+c$ where $0 \le a \le b+c$ and $0 \le b, c \le 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

Notice that if $a = 0$ and $b = 1$, the current character is unchanged; if $a = b$ and $c = 1$, the current character is changed but the next character is unchanged.

If the very first instruction of the *lig_kern* array has $skip\_byte = 255$, the *next_char* byte is the so-called right boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has $skip\_byte = 255$, there is a special ligature/kerning program for a left boundary character, beginning at location $256\,op\_byte + remainder$. The interpretation is that TeX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has $skip\_byte > 128$, the program actually begins in location $256\,op\_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with *skip_byte* > 128 in the *lig_kern* array must have $256\,op\_byte + remainder < nl$. If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature command is performed.

Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

The final portion of a `TFM` file is the *param* array, which is another sequence of *fix_word* values.

$param[1] = slant$ is the amount of italic slant, which is used to help position accents. For example, $slant = 0.25$ means that when you go up one unit, you also go 0.25 units to the right. The *slant* is a pure number; it's the only *fix_word* other than the design size itself that is not scaled by the design size.

$param[2] = space$ is the normal spacing between words in text. Note that character `"␣"` in the font need not have anything to do with blank spaces.

$param[3] = space\_stretch$ is the amount of glue stretching between words.

$param[4] = space\_shrink$ is the amount of glue shrinking between words.

$param[5] = x\_height$ is the height of letters for which accents don't have to be raised or lowered.

$param[6] = quad$ is the size of one em in the font.

$param[7] = extra\_space$
is the amount added to $param[2]$ at the ends of sentences.

When the character coding scheme is `TeX math symbols`, the font is supposed to have 15 additional parameters called *num1*, *num2*, *num3*, *denom1*, *denom2*, *sup1*, *sup2*, *sup3*, *sub1*, *sub2*, *supdrop*, *subdrop*, *delim1*, *delim2*, and *axis_height*, respectively. When the character coding scheme is `TeX math extension`, the font is supposed to have six additional parameters called *default_rule_thickness* and *big_op_spacing1* through *big_op_spacing5*.