Software) are offering. It is impressive that they plan to offer phone support for their products and, if contacted by enough interested VERSATEC users, would be willing to consider providing a true spooler for the VERSATEC.

We are planning to have a VMS/TEX session as part of the upcoming TUG meeting at Stanford. If you have suggestions on subjects that should be discussed that are VMS specific, be sure to contact me.

* * * * * * * * * * *

NOTICE TO VAX/VMS USERS

David Kellerman
Barry Smith

Early this year, we left Oregon Software, forming a partnership to continue our work with TEX and WEB. We are continuing to handle the distribution of TEX software for VAX/VMS systems (Oregon Software will no longer distribute TEX).

Our new address:

Kellerman and Smith
2343 SE 45th Avenue
Portland, Oregon 97215
(503) 232-4799

The following software is currently available:

- **TEX 1.0 (TEX82).** TEX itself is compatible with the version put together by David Fuchs, and offers some additional performance and friendly features (the PLAIN format is really preloaded, for example). The system interface to VMS is cleaner, especially for batch processing. For those who commonly use canned macros, we've packaged INITEX so that you can easily create saved images with new preloaded formats. Included on one 2400′ tape are the TEXware programs, the WEB system, all program sources and executable images, and the new AM fonts at twenty-one different magnifications. (We include a copy of The TEXbook.)
- **VERTEX.** This is an all-new WEB language Versatec driver program for model 1200 and V-80 printers. VERTEX uses the VMS Command Language Definition facility and has a bewildering number of options. It can print TEX78 or TEX 1.0 DVI files, with the old or new PXL font files, in either landscape or portrait orientation. Like TEX, you can easily preload your set of common PXL images at your site. VERTEX is provided in executable image format on a 600′ tape.
- **IMPRINT.** This is a print spooler that, in various versions, will drive the Imagen printers (IMPRINT-10, 8/300, 5/840, 60/240). IMPRINT uses the VAX/VMS print queueing facilities and is completely compatible with the standard PRINT command. It prints files in Printer, Daisy, Tektronix, Impress, and DVI formats without intermediate processing. Even more than VERTEX, IMPRINT has a ridiculous array of options, as well as several layers of site and user-dependent defaults to simplify commands. IMPRINT is provided in executable image format on a 600′ tape, and is also available directly from the Imagen Corporation.

All of the above software includes a user's guide, system manager's installation guide, 90-day unconditional warranty, telephone support for the same periosm, and domestic shipping via UPS 2nd-day air. International orders will be billed for air-freight costs, and must include a written statement that the software will not be re-exported.

Prices? TEX is $200 (US), VERTEX is $400, a package with both TEX and VERTEX is $500, and IMPRINT (IMPRINT-10, 8/300) is $1,200 ($900 for educational institutions). We will be offering support, maintenance, and update services in the near future.

Our current projects are a true spooler for the Versatec, and a VAX/VMS spooling interface to the Compugraphics 8400/8600 photo-typesetters. We're open to other requests.

* * * * * * * * * * *

Fonts

* * * * * * * * * * *

Editor's note: The two documents on the following pages are extracted from a work-in-progress—the new Metafont—and are thus subject to change. Nonetheless, they give the flavor of the new approach to device-independent font definition, and should be useful for (as David Fuchs has put it in his report on page 22) 'planning ahead'.

METAFONT GENERIC FONT FILE FORMAT

1. Generic font file format. 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 seven 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 the "black" pixels of a character, 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 byte is number 0, then comes number 1, and so on.

2. 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 several quantities: (a) the current row number, y; (b) the current column number, x; and (c) the current starting-column number, z. 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 $|x|$, $|y|$, or $|z|$ to exceed 4095, 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 row 0 and column 0 is METAFONT's unit square, comprising the region of the plane whose coordinates both lie between 0 and 1. Negative values of y correspond to rows of pixels *below* the baseline.

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

3. 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 y, in columns x through $x+d-1$ inclusive. Then, in any case, complement the *paint_switch* and advance x 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_x*[4] *max_x*[4] *min_y*[4] *max_y*[4] z[4]. Beginning of a character: Here c is the character code, and p points to the previous *boc* command (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.) All x-coordinates of black pixels in the character that follows will be $\ge$ *min_x* and $\le$ *max_x*; all y-coordinates of black pixels will be $\ge$ *min_y* and $\le$ *max_y*. Finally, z is the leftmost potentially black column in row *max_y*; it satisfies $min\_x \le z \le max\_x$. When a GF-reading program sees a *boc*, it can use *min_x*, *max_x*, *min_y*, and *max_y* to initialize the bounds of an array. Then it sets $y \leftarrow max\_y$, *paint_switch* $\leftarrow$ *black*, and initializes its x and z registers to the stated value of z.

eoc 68. 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*.

skip1 69 m[1]. Decrease y by $m + 1$, set $x \leftarrow z$, and set *paint_switch* $\leftarrow$ *black*. This is a way to produce m all-white rows.

skip2 70 m[2]. Same as *skip1*, but m can be as large as 65535.

skip3 71 m[3]. Same as *skip1*, but m can be as large as $2^{24} - 1$. METAFONT obviously never needs this command.

new_row 72 u[4]. Decrease y by 1 and set $z \leftarrow z + u$; then set $x \leftarrow z$ and *paint_switch* $\leftarrow$ *black*. (It's a general way to finish one row and begin another.)

left_z_83 through *left_z_1* (opcodes 73 to 155). Same as *new_row*, with $u = -83$ through -1, respectively.

right_z_0 156. Same as *skip1* with $m = 0$ or *new_row* with $u = 0$.

right_z_1 through *right_z_83* (opcodes 157 to 239). Same as *new_row*, with $u = +1$ through $+83$, respectively. METAFONT generates a *new_row* command only when $|u| > 83$.

nop 240. No operation, do nothing. Any number of *nop*'s may occur between GF commands, but a *nop* cannot be inserted between a command and its parameters or between two parameters.

xxx1 241 k[1] x[k]. This command is undefined in general; it functions as a $(k + 2)$-byte *nop* 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 can appear anywhere. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 242 k[2] x[k]. Like *xxx1*, but $0 \le k < 65536$.

xxx3 243 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 244 k[4] x[k]. Like *xxx1*, but k can be ridiculously large; k mustn't be negative.

yyy 245 n[4]. This command is undefined in general; it functions as a 5-byte *nop* 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.

char_loc 246 c[1] v[4] w[4] p[4]. This command will appear only in the postamble, which will be explained shortly.

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 129. 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.

define *gf_id_byte* = 129 { identifies the kind of GF files described here }

4. 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_x*[4] *max_x*[4] *min_y*[4] *max_y*[4]
⟨ character locators ⟩
post_post *q*[4] *i*[1] 223's[≥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 the TFM file that METAFONT produces (or would produce) on this run. 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_x*, *max_x*, *min_y*, and *max_y*, which bound the values that x and y assume in all of the characters of this GF file.

5. Character locators are introduced by *char_loc* commands, which contain a character residue c, a character device width v, 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 device width v is the value of METAFONT's **chardw** parameter, rounded to the nearest integer, i.e., the number of pixels that the font designer wishes the character to occupy when it is typeset within a word.

The character width w duplicates the information in the TFM file; it is a *fix_word* value relative to the design size, and it should be independent of magnification.

The backpointer p points to the character's *boc*, or to the first of a sequence of consecutive *nop* or *xxx* or *yyy* 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*.

6. 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 129, 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). 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.

Unfortunately, however, standard PASCAL does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so GF format has been designed to work most efficiently with modern operating systems. But if GF files have to be processed under the restrictions of standard PASCAL, one can simply read them from front to back. This will be adequate for most applications. However, the postamble-first approach would facilitate a program that merges two GF files, replacing data from one that is overridden by corresponding data in the other.