The PKtoGF processor

(Version 1.1, 22 April 2020)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926 and MCS-8300984, and by the System Development Foundation.

1. Introduction. This program takes a packed, or PK file, and converts it into the standard GF format. The resulting GF file is standard in every way, and is essentially identical to the GF file from which the PK file was produced in the first place. Note that, however, GF to PK to GF is not an exact identity transformation, as the new GF file will have a different preamble string and the actual minimum bounding box will be used, instead of a possibly larger bounding box in the original GF file.

2. The *banner* string defined here should be changed whenever PKtoGF gets modified. You should update the preamble comment as well.

```
define banner ≡ 'This␣is␣PKtoGF,␣Version␣1.1'   { printed when the program starts }
define preamble_comment ≡ 'PKtoGF␣1.1␣output'
define comm_length ≡ 17
```

3. This program is written in standard Pascal, except where it is necessary to use extensions; for example, PKtoGF must read files whose names are dynamically specified, and that would be impossible in pure Pascal.

```
define othercases ≡ others:    { default for cases not listed explicitly }
define endcases ≡ end    { follows the default case in an extended case statement }
format othercases ≡ else
format endcases ≡ end
```

4. Both the input and output come from binary files. On line interaction is handled through Pascal's standard *input* and *output* files.

```
define print_ln(#) ≡ write_ln(output,#)
define print(#) ≡ write(output,#)
program PKtoGF(input,output);
  label ⟨ Labels in the outer block 5 ⟩
  const ⟨ Constants in the outer block 6 ⟩
  type ⟨ Types in the outer block 9 ⟩
  var ⟨ Globals in the outer block 11 ⟩
  procedure initialize;   { this procedure gets things started properly }
    var i: integer;   { loop index for initializations }
    begin print_ln(banner);
    ⟨ Set initial values 12 ⟩
    end;
```

5. If the program has to stop prematurely, it goes to the '*final_end*'.

```
define final_end = 9999   { label for the end of it all }
⟨ Labels in the outer block 5 ⟩ ≡
  final_end;
```

This code is used in section 4.

6. These constants determine the maximum length of a file name and the length of the terminal line, as well as the maximum number of run counts allowed per line of the GF file. (We need this to implement repeat counts.)

```
⟨ Constants in the outer block 6 ⟩ ≡
  name_length = 80;   { maximum length of a file name }
  terminal_line_length = 132;   { maximum length of an input line }
  max_counts = 400;   { maximum number of run counts in a raster line }
```

This code is used in section 4.

7. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1   { increase a variable by unity }
define decr(#) ≡ # ← # − 1   { decrease a variable by unity }
define do_nothing ≡          { empty statement }
```

8. It is possible that a malformed packed file (heaven forbid!) or some other error might be detected by this program. Such errors might occur in a deeply nested procedure, so the procedure called *jump_out* has been added to transfer to the very end of the program with an error message.

```
define abort(#) ≡
        begin print_ln('␣', #); jump_out;
        end
procedure jump_out;
  begin goto final_end;
  end;
```

9. The character set. Like all programs written with the WEB system, PKtoGF can be used with any character set. But it uses ASCII code internally, because the programming for portable input-output is easier when a fixed internal code is used.

The next few sections of PKtoGF have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since PKtoGF need not deal with the controversial ASCII codes less than *´40*.

⟨ Types in the outer block 9 ⟩ ≡
```
ASCII_code = "␣" .. "~";  { a subrange of the integers }
```
See also sections 10 and 38.

This code is used in section 4.

10. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like GFtoPK. So we shall assume that the Pascal system being used for GFtoPK has a character set containing at least the standard visible characters of ASCII code ("!" through "~").

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name. In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the output file. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

```
define text_char ≡ char  { the data type of characters in text files }
define first_text_char = 0  { ordinal number of the smallest element of text_char }
define last_text_char = 127  { ordinal number of the largest element of text_char }
```
⟨ Types in the outer block 9 ⟩ +≡
```
text_file = packed file of text_char;
```

11. The GFtoPK processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions.

⟨ Globals in the outer block 11 ⟩ ≡
```
xord: array [text_char] of ASCII_code;  { specifies conversion of input characters }
xchr: array [0 .. 255] of text_char;  { specifies conversion of output characters }
```
See also sections 39, 41, 48, 50, 55, 57, 63, 67, and 69.

This code is used in section 4.

12. Under our assumption that the visible characters of standard ASCII are all present, the following assignment statements initialize the *xchr* array properly, without needing any system-dependent changes.

```
⟨ Set initial values 12 ⟩ ≡
  for i ← 0 to '37 do xchr[i] ← '?';
  xchr['40] ← '␣'; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
  xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← ''';
  xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
  xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
  xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
  xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
  xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
  xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
  xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
  xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
  xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
  xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
  xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
  xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
  xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
  xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
  xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
  xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
  xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
  xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
  xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
  xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
  xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
  xchr['175] ← '}'; xchr['176] ← '~';
  for i ← '177 to 255 do xchr[i] ← '?';
```

See also sections 13, 51, and 58.

This code is used in section 4.

13. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*.

```
⟨ Set initial values 12 ⟩ +≡
  for i ← first_text_char to last_text_char do xord[chr(i)] ← '40;
  for i ← "␣" to "~" do xord[xchr[i]] ← i;
```

14. 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 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.

15. 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.

16. 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.

define *gf_id_byte* = 131 { identifies the kind of GF files described here }

17. Here are the opcodes that GFtoPK actually refers to.

define *paint_0* = 0 { beginning of the *paint* commands }
define *paint1* = 64 { move right a given number of columns, then black ↔ white }
define *boc* = 67 { beginning of a character }
define *boc1* = 68 { abbreviated *boc* }
define *eoc* = 69 { end of a character }
define *skip0* = 70 { skip no blank rows }
define *skip1* = 71 { skip over blank rows }
define *new_row_0* = 74 { move down one row and then right }
define *max_new_row* = 238 { move down one row and then right }
define *no_op* = 247 { noop }
define *xxx1* = 239 { for **special** strings }
define *yyy* = 243 { for **numspecial** numbers }
define *nop* = 244 { no operation }
define *char_loc* = 245 { character locators in the postamble }
define *char_loc0* = 246 { character locators in the postamble }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *post_post* = 249 { postamble ending }
define *undefined_commands* ≡ 250, 251, 252, 253, 254, 255

18. 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[≥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*.)

19. 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^{20} 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* ≥ 0 in order to get a GF file.

20. 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., *´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. GFtoPK first reads the postamble, and then scans the file from front to back.

21. Packed file format. 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 expressly 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 are separated into two sets, those that introduce a character definition, and those that do not. The values that introduce a character definition comprise the range from 0 to 239; byte values above 239 are interpreted commands. Bytes which 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 interpreted 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.

22. 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 defined. 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.

23. First, the command bytes shall 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 mustn'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 which 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". Usually, the name of the PK file is formed by concatenating the font name (e.g., cmr10) with the resolution at which the font is prepared in pixels per inch multiplied by the magnification factor, and the letters PK. For instance, cmr10 at 300 dots per inch should be named CMR10.300PK; at one thousand dots per inch and magstephalf, it should be named CMR10.1095PK.

24. We put a few of the above opcodes into definitions for symbolic use by this program.

define *pk_id* = 89 { the version of PK file described }
define *pk_xxx1* = 240 { **special** commands }
define *pk_yyy* = 244 { **numspecial** commands }
define *pk_post* = 245 { postamble }
define *pk_no_op* = 246 { no operation }
define *pk_pre* = 247 { preamble }

25. 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 the top of the character to the bottom. This is essentially the way the GF format works. Instead of presenting each row individually, however, let us 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 be easily reconstructed. In addition, we do not need special commands to mark the end of one row and the beginning of the next.

Next, let us put 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 which encloses all 'black' pixels of a character. Let us 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 seems 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.

26. The analysis of the 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.

27. Now let us 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, however. 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.

28. 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 .. *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-byte nybble values will run from $dyn\_f + 1$ .. $(13 - dyn\_f) * 16 + dyn\_f$. 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 which will pack each individual character in the smallest number of nybbles. Legal values of *dyn_f* run from 0 (with no one-byte run counts) to 13 (with no two-byte run counts).

29. Our only remaining task in the coding of packed numbers is the large run counts. We use a scheme suggested by D. E. Knuth which will simply and elegantly represent 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 $(13 - dyn\_f) * 16 + dyn\_f$. 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 $(13 - dyn\_f) * 16 + dyn\_f + 1$ and adding 16, and then representing the result according to the scheme above.

30. The final algorithm for decoding the run counts based on the above scheme might look like this, assuming 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.

```
⟨ Packed number procedure 30 ⟩ ≡
function pk_packed_num: integer;
  var i, j, k: integer;
  begin i ← get_nyb;
  if i = 0 then
    begin repeat j ← get_nyb; incr(i);
    until j ≠ 0;
    while i > 0 do
      begin j ← j * 16 + get_nyb; decr(i);
      end;
    pk_packed_num ← j − 15 + (13 − dyn_f) * 16 + dyn_f;
    end
  else if i ≤ dyn_f then pk_packed_num ← i
    else if i < 14 then pk_packed_num ← (i − dyn_f − 1) * 16 + get_nyb + dyn_f + 1
      else begin if i = 14 then repeat_count ← pk_packed_num
        else repeat_count ← 1;
        pk_packed_num ← pk_packed_num;
        end;
  end;
```

This code is used in section 62.

31. 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 $(width * height + 7)$ **div** 8. 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.

32. 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 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, and which one 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 `aminch`.

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 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 $(flag \bmod 4) * 256 + pl$; for the extended format, it might be calculated as $(flag \bmod 4) * 65536 + pl$.

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 *dy* 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 which occupies the unit square in METAFONT; the METAFONT reference point is the lower left hand corner of this pixel. (See the example below.)

33. TEX requires that all characters which have the same character codes modulo 256 also have 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.

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.

34. As an illustration of the PK format, the character Ξ from the font amr10 at 300 dots per inch will be encoded. (Note: amr fonts are obsolete, and the reference to this character is retained from an older version of the Computer Modern fonts solely for illustration.) This character was chosen because it illustrates some of the borderline cases. The raster for the character looks like this (the row numbers are chosen for convenience, and are not METAFONT's row numbers.)

```
 0      MMMMMMMMMMMMMMMMMMMMMMM
 1      MMMMMMMMMMMMMMMMMMMMMMM
 2      MMMMMMMMMMMMMMMMMMMMMMM
 3      MMMMMMMMMMMMMMMMMMMMMMM
 4      MM                   MM
 5      MM                   MM
 6      MM                   MM
 7
 8
 9         MM             MM
10         MM             MM
11         MM             MM
12         MMMMMMMMMMMMMMMMMM
13         MMMMMMMMMMMMMMMMMM
14         MMMMMMMMMMMMMMMMMM
15         MMMMMMMMMMMMMMMMMM
16         MM             MM
17         MM             MM
18         MM             MM
19
20
21
22      MM                   MM
23      MM                   MM
24      MM                   MM
25      MMMMMMMMMMMMMMMMMMMMMMM
26      MMMMMMMMMMMMMMMMMMMMMMM
27      MMMMMMMMMMMMMMMMMMMMMMM
28   *  MMMMMMMMMMMMMMMMMMMMMMM
```

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 repeat counts are calculated. The rows thus removed are rows 5, 6, 10, 11, 13, 14, 15, 17, 18, 23, and 24.

35. 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 which in one pass over the count list can determine the best value of *dyn_f*. 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. 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.

36. 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:

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`

37. This format was written by Tomas Rokicki in August, 1985.

38. Input and output. There are two types of files that this program must deal with—standard text files and files of bytes (packed files and generic font files.) For our purposes, we shall consider an eight-bit byte to consist of the values 0 .. 255. If your system does not pack these values to a byte, it is no major difficulty; you must only insure that the input function *pk_byte* can read packed bytes, and that the output function *gf_byte* packs the bytes to be shipped.

```
⟨ Types in the outer block 9 ⟩ +≡
  eight_bits = 0 .. 255;  { packed file byte }
  byte_file = packed file of eight_bits;  { for packed file words }
```

39.
```
⟨ Globals in the outer block 11 ⟩ +≡
gf_file, pk_file: byte_file;  { the I/O streams }
```

40. To prepare these files for input, we *reset* them. An extension of Pascal is needed in the case of *gf_file*, since we want to associate it with external files whose names are specified dynamically (i.e., not known at compile time). The following code assumes that '*reset*(f, s)' does this, when f is a file variable and s is a string variable that specifies the file name. If *eof*(f) is true immediately after *reset*(f, s) has acted, we assume that no file named s is accessible.

```
procedure open_gf_file;  { prepares to write packed bytes in a gf_file }
  begin rewrite(gf_file, gf_name); gf_loc ← 0;
  end;
procedure open_pk_file;  { prepares the input for reading }
  begin reset(pk_file, pk_name); pk_loc ← 0;
  end;
```

41. We need a place to store the names of the input and output files, as well as a byte counter for the output file.

```
⟨ Globals in the outer block 11 ⟩ +≡
gf_name, pk_name: packed array [1 .. name_length] of char;  { names of input and output files }
gf_loc, pk_loc: integer;  { how many bytes have we sent? }
```

42. We need a procedure that will write a byte to the `GF` file. If the particular system requires buffering, here is the place to do it.

```
procedure gf_byte(i : integer);
  begin gf_file↑ ← i; put(gf_file); incr(gf_loc);
  end;
```

43. We also need a function that will get a single byte from the `PK` file. Again, buffering may be done in this procedure.

```
function pk_byte: eight_bits;
  var nybble, temp: eight_bits;
  begin temp ← pk_file↑; get(pk_file); pk_loc ← pk_loc + 1; pk_byte ← temp;
  end;
```

44. Now we are ready to open the files and write the identification of the pixel file.

```
⟨ Open files 44 ⟩ ≡
  open_pk_file; open_gf_file
```

This code is used in section 73.

45. As we are reading the packed file, we often need to fetch 16 and 32 bit quantities. Here we have two procedures to do this.

```
function signed_byte: integer;
  var a: integer;
  begin a ← pk_byte;
  if a > 127 then a ← a − 256;
  signed_byte ← a;
  end;
function get_16: integer;
  var a: integer;
  begin a ← pk_byte; get_16 ← a * 256 + pk_byte;
  end;
function signed_16: integer;
  var a: integer;
  begin a ← signed_byte; signed_16 ← a * 256 + pk_byte;
  end;
function get_32: integer;
  var a: integer;
  begin a ← get_16;
  if a > 32767 then a ← a − 65536;
  get_32 ← a * 65536 + get_16;
  end;
```

46. As we are writing the `GF` file, we often need to write signed and unsigned, one, two, three, and four-byte values. These routines give us that capability.

```
procedure gf_sbyte(i : integer);
  begin if i < 0 then i ← i + 256;
  gf_byte(i);
  end;
procedure gf_16(i : integer);
  begin gf_byte(i div 256); gf_byte(i mod 256);
  end;
procedure gf_24(i : integer);
  begin gf_byte(i div 65536); gf_16(i mod 65536);
  end;
procedure gf_quad(i : integer);
  begin if i ≥ 0 then
    begin gf_byte(i div 16777216);
    end
  else begin i ← (i + 1073741824) + 1073741824; gf_byte(128 + (i div 16777216));
    end;
  gf_24(i mod 16777216);
  end;
```

47. Character unpacking. Now we deal with unpacking characters into the GF representation.

```
⟨ Unpack and write character 47 ⟩ ≡
  dyn_f ← flag_byte div 16; flag_byte ← flag_byte mod 16; turn_on ← flag_byte ≥ 8;
  if turn_on then flag_byte ← flag_byte − 8;
  if flag_byte = 7 then ⟨ Read long character preamble 52 ⟩
  else if flag_byte > 3 then ⟨ Read extended short character preamble 53 ⟩
    else ⟨ Read short character preamble 54 ⟩;
  ⟨ Calculate and check min_m, max_m, min_n, and max_n 56 ⟩;
  ⟨ Save character locator 60 ⟩;
  ⟨ Write character preamble 59 ⟩;
  ⟨ Read and translate raster description 65 ⟩;
  gf_byte(eoc); last_eoc ← gf_loc;
  if end_of_packet ≠ pk_loc then abort('Bad␣pk␣file!␣␣Bad␣packet␣length.')
```

This code is used in section 73.

48. We need a whole lot of globals used but not defined up there.

```
⟨ Globals in the outer block 11 ⟩ +≡
i, j: integer; { index pointers }
end_of_packet: integer; { where we expect the end of the packet to be }
dyn_f: integer; { dynamic packing variable }
car: integer; { the character we are reading }
tfm_width: integer; { the TFM width of the current character }
x_off, y_off: integer; { the offsets for the character }
```

49. Now we read and check the preamble of the PK file. In the preamble, we find the *hppp*, *design_size*, *checksum*. We write the relevant parameters to the GF file, including the preamble comment.

```
⟨ Read preamble 49 ⟩ ≡
  if pk_byte ≠ pk_pre then abort('Bad␣pk␣file!␣␣pre␣command␣missing.');
  gf_byte(pre);
  if pk_byte ≠ pk_id then abort('Wrong␣version␣of␣packed␣file!.');
  gf_byte(gf_id_byte); j ← pk_byte;
  for i ← 1 to j do hppp ← pk_byte;
  gf_byte(comm_length);
  for i ← 1 to comm_length do gf_byte(xord[comment[i]]);
  design_size ← get_32; checksum ← get_32; hppp ← get_32; vppp ← get_32;
  if hppp ≠ vppp then print_ln('Warning:␣␣aspect␣ratio␣not␣1:1!');
  magnification ← round(hppp * 72.27 * 5/65536); last_eoc ← gf_loc
```

This code is used in section 73.

50. Of course, we need to define the above variables.

```
⟨ Globals in the outer block 11 ⟩ +≡
comment: packed array [1 .. comm_length] of char;
magnification: integer; { resolution at which pixel file is prepared }
design_size: integer; { design size in FIXes }
checksum: integer; { checksum of pixel file }
hppp, vppp: integer; { horizontal and vertical points per inch }
```

51.

```
⟨ Set initial values 12 ⟩ +≡
  comment ← preamble_comment;
```

52. Now, the character preamble reading modules. First, we have the general case: the long character preamble format.

⟨ Read long character preamble 52 ⟩ ≡
```
begin packet_length ← get_32; car ← get_32; end_of_packet ← packet_length + pk_loc;
tfm_width ← get_32; hor_esc ← get_32; ver_esc ← get_32; c_width ← get_32; c_height ← get_32;
word_width ← (c_width + 31) div 32; x_off ← get_32; y_off ← get_32;
end
```
This code is used in section 47.

53. This module reads the character preamble with double byte parameters.

⟨ Read extended short character preamble 53 ⟩ ≡
```
begin packet_length ← (flag_byte − 4) * 65536 + get_16; car ← pk_byte;
end_of_packet ← packet_length + pk_loc; i ← pk_byte; tfm_width ← i * 65536 + get_16;
hor_esc ← get_16 * 65536; ver_esc ← 0; c_width ← get_16; c_height ← get_16;
word_width ← (c_width + 31) div 32; x_off ← signed_16; y_off ← signed_16;
end
```
This code is used in section 47.

54. Here we read the most common character preamble, that with single byte parameters.

⟨ Read short character preamble 54 ⟩ ≡
```
begin packet_length ← flag_byte * 256 + pk_byte; car ← pk_byte; end_of_packet ← packet_length + pk_loc;
i ← pk_byte; tfm_width ← i * 65536 + get_16; hor_esc ← pk_byte * 65536; ver_esc ← 0;
c_width ← pk_byte; c_height ← pk_byte; word_width ← (c_width + 31) div 32; x_off ← signed_byte;
y_off ← signed_byte;
end
```
This code is used in section 47.

55. Some more globals:

⟨ Globals in the outer block 11 ⟩ +≡
```
c_height, c_width: integer; { sizes of the character glyphs }
word_width: integer; { width of character in raster words }
hor_esc, ver_esc: integer; { the character escapement }
packet_length: integer; { the length of the packet in bytes }
last_eoc: integer; { the last end of character }
```

56. The GF format requires the minimum and maximum m and n values in the postamble, so we generate them here. One thing that should be noted, here. The value $max\_n - min\_n$ will be the height of the character glyph, but for the width, you need to use $max\_m - min\_m - 1$, because of the peculiarities of the GF format.

```
⟨ Calculate and check min_m, max_m, min_n, and max_n 56 ⟩ ≡
  if (c_height = 0) ∨ (c_width = 0) then
    begin c_height ← 0; c_width ← 0; x_off ← 0; y_off ← 0;
    end;
  min_m ← −x_off;
  if min_m < mmin_m then mmin_m ← min_m;
  max_m ← c_width + min_m;
  if max_m > mmax_m then mmax_m ← max_m;
  min_n ← y_off − c_height + 1; max_n ← y_off;
  if min_n > max_n then min_n ← max_n;
  if min_n < mmin_n then mmin_n ← min_n;
  if max_n > mmax_n then mmax_n ← max_n
```

This code is used in section 47.

57. We have to declare the variables which hold the bounding box. We also need the arrays that hold the back pointers to the characters, the horizontal and vertical escapements, and the TFM widths.

```
⟨ Globals in the outer block 11 ⟩ +≡
min_m, max_m, min_n, max_n: integer;
mmin_m, mmax_m, mmin_n, mmax_n: integer;
char_pointer, s_tfm_width: array [0 .. 255] of integer;
s_hor_esc, s_ver_esc: array [0 .. 255] of integer;
this_char_ptr: integer;
```

58. We initialize these bounding box values to be ridiculous, and say that there were no characters seen yet.

```
⟨ Set initial values 12 ⟩ +≡
  mmin_m ← 999999; mmin_n ← 999999; mmax_m ← −999999; mmax_n ← −999999;
  for i ← 0 to 255 do char_pointer[i] ← −1;
```

59. This module takes care of the simple job of writing the character preamble, after picking one to fit.

```
⟨ Write character preamble 59 ⟩ ≡
  begin if (char_pointer[car mod 256] = −1) ∧ (car ≥ 0) ∧ (car < 256) ∧ (max_m ≥ 0) ∧ (max_m <
        256) ∧ (max_n ≥ 0) ∧ (max_n < 256) ∧ (max_m ≥ min_m) ∧ (max_n ≥ min_n) ∧ (max_m <
        min_m + 256) ∧ (max_n < min_n + 256) then
    begin char_pointer[car mod 256] ← this_char_ptr; gf_byte(boc1); gf_byte(car);
    gf_byte(max_m − min_m); gf_byte(max_m); gf_byte(max_n − min_n); gf_byte(max_n);
    end
  else begin gf_byte(boc); gf_quad(car); gf_quad(char_pointer[car mod 256]);
    char_pointer[car mod 256] ← this_char_ptr; gf_quad(min_m); gf_quad(max_m); gf_quad(min_n);
    gf_quad(max_n);
    end;
  end
```

This code is used in section 47.

60. In this routine we either save or check the current character parameters.

```
⟨ Save character locator 60 ⟩ ≡
  begin i ← car mod 256;
  if (char_pointer[i] = −1) then
    begin s_ver_esc[i] ← ver_esc; s_hor_esc[i] ← hor_esc; s_tfm_width[i] ← tfm_width;
    end
  else begin if (s_ver_esc[i] ≠ ver_esc) ∨ (s_hor_esc[i] ≠ hor_esc) ∨ (s_tfm_width[i] ≠ tfm_width) then
      print_ln('Two␣characters␣mod␣', i : 1, '␣have␣mismatched␣parameters');
    end;
  end
```

This code is used in section 47.

61. And another module to write out those character locators we have so carefully saved up the information for.

```
⟨ Write character locators 61 ⟩ ≡
  for i ← 0 to 255 do
    if char_pointer[i] ≠ −1 then
      begin if (s_ver_esc[i] = 0)∧(s_hor_esc[i] ≥ 0)∧(s_hor_esc[i] < 16777216)∧(s_hor_esc[i] mod 65536 = 0)
          then
        begin gf_byte(char_loc0); gf_byte(i); gf_byte(s_hor_esc[i] div 65536);
        end
      else begin gf_byte(char_loc); gf_byte(i); gf_quad(s_hor_esc[i]); gf_quad(s_ver_esc[i]);
        end;
      gf_quad(s_tfm_width[i]); gf_quad(char_pointer[i]);
      end
```

This code is used in section 68.

62. Now we have the most important part of the program, where we actually interpret the commands in the raster description. First of all, we need a procedure to get a single nybble from the file, as well as one to get a single bit. We also use the *pk_packed_num* procedure defined in the PK file description.

```
function get_nyb: integer;
  var temp: eight_bits;
  begin if bit_weight = 0 then
    begin input_byte ← pk_byte; bit_weight ← 16;
    end;
  temp ← input_byte div bit_weight; input_byte ← input_byte − temp ∗ bit_weight;
  bit_weight ← bit_weight div 16; get_nyb ← temp;
  end;
function get_bit: boolean;
  var temp: boolean;
  begin bit_weight ← bit_weight div 2;
  if bit_weight = 0 then
    begin input_byte ← pk_byte; bit_weight ← 128;
    end;
  temp ← input_byte ≥ bit_weight;
  if temp then input_byte ← input_byte − bit_weight;
  get_bit ← temp;
  end; ⟨ Packed number procedure 30 ⟩
```

63. Now, the globals to help communication between these procedures, and a buffer for the raster row counts.

```
⟨ Globals in the outer block 11 ⟩ +≡
input_byte: eight_bits;  { the byte we are currently decimating }
bit_weight: eight_bits;  { weight of the current bit }
nybble: eight_bits;  { the current nybble }
row_counts: array [0 .. max_counts] of integer;  { where the row is constructed }
rcp: integer;  { the row counts pointer }
```

64. Actually, if the character is a bit mapped character, then we make it look like run counts by determining the appropriate values ourselves. Thus, we have a routine which gets the next count value, below.

```
⟨ Get next count value into count 64 ⟩ ≡
  begin turn_on ← ¬turn_on;
  if dyn_f = 14 then
    begin count ← 1; done ← false;
    while ¬done do
      begin if count_down ≤ 0 then done ← true
      else if (turn_on = get_bit) then count ← count + 1
        else done ← true;
      count_down ← count_down − 1;
      end;
    end
  else count ← pk_packed_num;
  end
```

This code is used in section 65.

65. And the main procedure.

```
⟨ Read and translate raster description 65 ⟩ ≡
  if (c_width > 0) ∧ (c_height > 0) then
    begin bit_weight ← 0; count_down ← c_height * c_width − 1;
    if dyn_f = 14 then turn_on ← get_bit;
    repeat_count ← 0; x_to_go ← c_width; y_to_go ← c_height; cur_n ← c_height; count ← 0;
    first_on ← turn_on; turn_on ← ¬turn_on; rcp ← 0;
    while y_to_go > 0 do
      begin if count = 0 then ⟨ Get next count value into count 64 ⟩;
      if rcp = 0 then first_on ← turn_on;
      while count ≥ x_to_go do
        begin row_counts[rcp] ← x_to_go; count ← count − x_to_go;
        for i ← 0 to repeat_count do
          begin ⟨ Output row 66 ⟩;
          y_to_go ← y_to_go − 1;
          end;
        repeat_count ← 0; x_to_go ← c_width; rcp ← 0;
        if (count > 0) then first_on ← turn_on;
        end;
      if count > 0 then
        begin row_counts[rcp] ← count;
        if rcp = 0 then first_on ← turn_on;
        rcp ← rcp + 1;
        if rcp > max_counts then
          begin print_ln('A␣character␣had␣too␣many␣run␣counts'); jump_out;
          end;
        x_to_go ← x_to_go − count; count ← 0;
        end;
      end;
    end
```

This code is used in section 47.

66. This routine actually outputs a row to the `GF` file.

```
⟨ Output row 66 ⟩ ≡
  if (rcp > 0) ∨ first_on then
    begin j ← 0; max ← rcp;
    if ¬turn_on then max ← max − 1;
    if cur_n − y_to_go = 1 then
      begin if first_on then gf_byte(new_row_0)
      else if row_counts[0] < 165 then
          begin gf_byte(new_row_0 + row_counts[0]); j ← j + 1;
          end
        else gf_byte(skip0);
      end
    else if cur_n > y_to_go then
        begin if cur_n − y_to_go < 257 then
          begin gf_byte(skip1); gf_byte(cur_n − y_to_go − 1);
          end
        else begin gf_byte(skip1 + 1); gf_16(cur_n − y_to_go − 1);
          end;
        if first_on then gf_byte(paint_0);
        end
      else if first_on then gf_byte(paint_0);
    cur_n ← y_to_go;
    while j ≤ max do
      begin if row_counts[j] < 64 then gf_byte(paint_0 + row_counts[j])
      else if row_counts[j] < 256 then
          begin gf_byte(paint1); gf_byte(row_counts[j]);
          end
        else begin gf_byte(paint1 + 1); gf_16(row_counts[j]);
          end;
      j ← j + 1;
      end;
    end
```

This code is used in section 65.

67. Here we need the array which counts down the number of bits, and the current state flag.

```
⟨ Globals in the outer block 11 ⟩ +≡
count_down: integer;   { have we run out of bits yet? }
done: boolean;   { are we done yet? }
max: integer;   { the maximum number of counts to output }
repeat_count: integer;   { how many times to repeat the next row? }
x_to_go, y_to_go: integer;   { how many columns/rows left? }
turn_on, first_on: boolean;   { are we black here? }
count: integer;   { how many bits of current color left? }
cur_n: integer;   { what row are we at? }
```

68. To finish the GF file, we write out a postamble, including the character locators that we stored away.

```
⟨ Write GF postamble 68 ⟩ ≡
  j ← gf_loc; gf_byte(post); gf_quad(last_eoc); gf_quad(design_size); gf_quad(checksum); gf_quad(hppp);
  gf_quad(vppp); gf_quad(mmin_m); gf_quad(mmax_m); gf_quad(mmin_n); gf_quad(mmax_n);
  ⟨ Write character locators 61 ⟩;
  gf_byte(post_post); gf_quad(j); gf_byte(gf_id_byte);
  for i ← 0 to 3 do gf_byte(223);
  while gf_loc mod 4 ≠ 0 do gf_byte(223)
```

This code is used in section 73.

69. We need the *flag_byte* variable.

```
⟨ Globals in the outer block 11 ⟩ +≡
flag_byte: integer;   { command or character flag byte }
```

70. Another necessary procedure skips over any specials between characters and before and after the postamble. (It echoes the specials exactly.)

```
procedure skip_specials;
  var i, j, k: integer;
  begin this_char_ptr ← gf_loc;
  repeat flag_byte ← pk_byte;
    if flag_byte ≥ 240 then
      case flag_byte of
      240, 241, 242, 243: begin i ← 0; gf_byte(flag_byte − 1);
        for j ← 240 to flag_byte do
          begin k ← pk_byte; gf_byte(k); i ← 256 ∗ i + k;
          end;
        for j ← 1 to i do gf_byte(pk_byte);
        end;
      244: begin gf_byte(243); gf_quad(get_32);
        end;
      245: begin end;
      246: begin end;
      247, 248, 249, 250, 251, 252, 253, 254, 255: abort('Unexpected␣', flag_byte : 1, '!');
      endcases;
  until (flag_byte < 240) ∨ (flag_byte = pk_post);
  end;
```

71. Terminal communication. We must get the file names and determine whether input is to be in hexadecimal or binary. To do this, we use the standard input path name. We need a procedure to flush the input buffer. For most systems, this will be an empty statement. For other systems, a *print_ln* will provide a quick fix. We also need a routine to get a line of input from the terminal. On some systems, a simple *read_ln* will do. Finally, a macro to print a string to the first blank is required.

```
define flush_buffer ≡
        begin end
define get_line(#) ≡
        if eoln(input) then read_ln(input);
      i ← 1;
      while ¬(eoln(input) ∨ eof(input)) do
        begin #[i] ← input↑; incr(i); get(input);
        end;
      #[i] ← ´␣´
```

72.

```
procedure dialog;
  var i: integer;  { index variable }
    buffer: packed array [1 .. name_length] of char;  { input buffer }
  begin for i ← 1 to name_length do
    begin gf_name[i] ← ´␣´; pk_name[i] ← ´␣´;
    end;
  print(´Input␣file␣name:␣␣´); flush_buffer; get_line(pk_name); print(´Output␣file␣name:␣␣´);
  flush_buffer; get_line(gf_name);
  end;
```

73. The main program. Now that we have all the pieces written, let us put them together.

```
  begin initialize; dialog; ⟨ Open files 44 ⟩;
  ⟨ Read preamble 49 ⟩;
  skip_specials;
  while flag_byte ≠ pk_post do
    begin ⟨ Unpack and write character 47 ⟩;
    skip_specials;
    end;
  while ¬eof(pk_file) do i ← pk_byte;
  ⟨ Write GF postamble 68 ⟩;
  print_ln(pk_loc : 1, ´␣bytes␣unpacked␣to␣´, gf_loc : 1, ´␣bytes.´);
final_end: end.
```

74. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `PKtoGF` work at a particular installation. Any additional routines should be inserted here.

75. Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

⟨ Calculate and check *min_m*, *max_m*, *min_n*, and *max_n* 56 ⟩ Used in section 47.
⟨ Constants in the outer block 6 ⟩ Used in section 4.
⟨ Get next count value into *count* 64 ⟩ Used in section 65.
⟨ Globals in the outer block 11, 39, 41, 48, 50, 55, 57, 63, 67, 69 ⟩ Used in section 4.
⟨ Labels in the outer block 5 ⟩ Used in section 4.
⟨ Open files 44 ⟩ Used in section 73.
⟨ Output row 66 ⟩ Used in section 65.
⟨ Packed number procedure 30 ⟩ Used in section 62.
⟨ Read and translate raster description 65 ⟩ Used in section 47.
⟨ Read extended short character preamble 53 ⟩ Used in section 47.
⟨ Read long character preamble 52 ⟩ Used in section 47.
⟨ Read preamble 49 ⟩ Used in section 73.
⟨ Read short character preamble 54 ⟩ Used in section 47.
⟨ Save character locator 60 ⟩ Used in section 47.
⟨ Set initial values 12, 13, 51, 58 ⟩ Used in section 4.
⟨ Types in the outer block 9, 10, 38 ⟩ Used in section 4.
⟨ Unpack and write character 47 ⟩ Used in section 73.
⟨ Write `GF` postamble 68 ⟩ Used in section 73.
⟨ Write character locators 61 ⟩ Used in section 68.
⟨ Write character preamble 59 ⟩ Used in section 47.