The GFtoPK processor

(Version 2.4, 06 January 2014)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926, MCS-8300984, and CCR-8610181, and by the System Development Foundation. 'TEX' is a trademark of the American Mathematical Society. 'METAFONT' is a trademark of Addison-Wesley Publishing Company.

1. Introduction. This program reads a GF file and packs it into a PK file. PK files are significantly smaller than GF files, and they are much easier to interpret. This program is meant to be the bridge between METAFONT and DVI drivers that read PK files. Here are some statistics comparing typical input and output file sizes:

Font	Resolution	GF size	PK size	Reduction factor
cmr10	300	13200	5484	42%
cmr10	360	15342	6496	42%
cmr10	432	18120	7808	43%
cmr10	511	21020	9440	45%
cmr10	622	24880	11492	46%
cmr10	746	29464	13912	47%
cminch	300	48764	22076	45%

It is hoped that the simplicity and small size of the PK files will make them widely accepted.

The PK format was designed and implemented by Tomas Rokicki during the summer of 1985. This program borrows a few routines from GFtoPXL by Arthur Samuel.

The *banner* string defined here should be changed whenever GFtoPK gets modified. The *preamble_comment* macro (near the end of the program) should be changed too.

define *banner* ≡ ´This␣is␣GFtoPK,␣Version␣2.4´ { printed when the program starts }

2. Some of the diagnostic information is printed using *d_print_ln*. When debugging, it should be set the same as *print_ln*, defined later.

define *d_print_ln*(#) ≡

3. This program is written in standard Pascal, except where it is necessary to use extensions; for example, one extension is to use a default **case** as in TANGLE, WEAVE, etc. All places where nonstandard constructions are used should be listed in the index under "system dependencies."

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. The binary input comes from *gf_file*, and the output font is written on *pk_file*. All text output is written on Pascal's standard *output* file. The term *print* is used instead of *write* when this program writes on *output*, so that all such output could easily be redirected if desired.

define *print*(#) ≡ *write*(#)
define *print_ln*(#) ≡ *write_ln*(#)

program *GFtoPK*(*gf_file*, *pk_file*, *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. The following parameters can be changed at compile time to extend or reduce GFtoPK's capacity. The values given here should be quite adequate for most uses. Assuming an average of about three strokes per raster line, there are six run-counts per line, and therefore *max_row* will be sufficient for a character 2600 pixels high.

⟨ Constants in the outer block 6 ⟩ ≡
line_length = 79; { bracketed lines of output will be at most this long }
max_row = 16000; { largest index in the main *row* array }

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 }

8. If the GF file is badly malformed, the whole process must be aborted; GFtoPK will give up, after issuing an error message about the symptoms that were noticed.

Such errors might be discovered inside of subroutines inside of subroutines, so a procedure called *jump_out* has been introduced. This procedure, which simply transfers control to the label *final_end* at the end of the program, contains the only non-local **goto** statement in GFtoPK.

define *abort*(#) ≡
begin *print*(´␣´, #); *jump_out*;
end
define *bad_gf*(#) ≡ *abort*(´Bad␣GF␣file:␣´, #, ´!´)

procedure *jump_out*;
begin goto *final_end*;
end;

9. The character set. Like all programs written with the WEB system, GFtoPK 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 GFtoPK have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since GFtoPK need not deal with the controversial ASCII codes less than *´40* or greater than *´176*. If such codes appear in the GF file, they will be printed as question marks.

⟨ Types in the outer block 9 ⟩ ≡
ASCII_code = "␣" .. "~"; { a subrange of the integers }
See also sections 10 and 37.
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 38, 41, 45, 47, 48, 55, 78, 82, and 87.
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, 42, 49, 79, and 83.

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 within characters, 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 *xxx1* = 239 { for **special** strings }
define *yyy* = 243 { for **numspecial** numbers }
define *no_op* = 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^{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* ≥ 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 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 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 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.

23. 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 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 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". 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 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.

26. 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.

27. 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.

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-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 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).

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 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 $(13 - dyn\_f) * 16 + dyn\_f$. 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 we represent 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 that a procedure called *get_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.

```
@{
function pk_packed_num: integer;
  var i, j: 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;
@}
```

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 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 that allows all of the parameters of the GF file format, including vertical escapement.

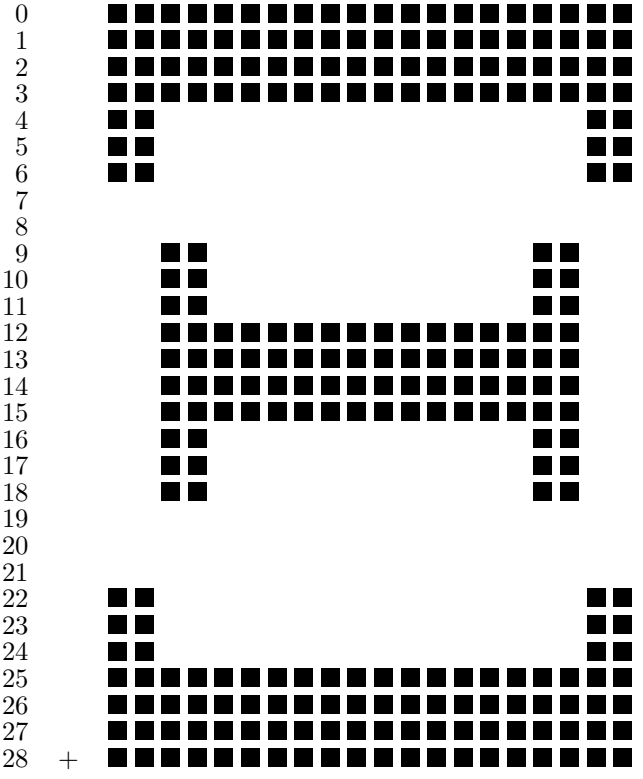
The *flag* parameter is the flag byte. The parameter *pl* (packet length) contains the offset of the byte following this character descriptor, with respect to the beginning of the *tfm* width parameter. This is given so a PK reading program can, once it has read the flag byte, packet length, and character 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 short 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 *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.)

33. 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.

34. 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 looks like this (the row numbers are chosen for convenience, and are not METAFONT's row numbers.)



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.

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 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. 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. Input and output for binary files. We have seen that a GF file is a sequence of 8-bit bytes. The bytes appear physically in what is called a '**packed file of** 0 .. 255' in Pascal lingo. The PK file is also a sequence of 8-bit bytes.

Packing is system dependent, and many Pascal systems fail to implement such files in a sensible way (at least, from the viewpoint of producing good production software). For example, some systems treat all byte-oriented files as text, looking for end-of-line marks and such things. Therefore some system-dependent code is often needed to deal with binary files, even though most of the program in this section of GFtoPK is written in standard Pascal.

We shall stick to simple Pascal in this program, for reasons of clarity, even if such simplicity is sometimes unrealistic.

⟨ Types in the outer block 9 ⟩ +≡
```
eight_bits = 0 .. 255;  { unsigned one-byte quantity }
byte_file = packed file of eight_bits;  { files that contain binary data }
```

38. The program deals with two binary file variables: *gf_file* is the input file that we are translating into PK format, to be written on *pk_file*.

⟨ Globals in the outer block 11 ⟩ +≡
```
gf_file: byte_file;  { the stuff we are GFtoPKing }
pk_file: byte_file;  { the stuff we have GFtoPKed }
```

39. To prepare the *gf_file* for input, we *reset* it.

```
procedure open_gf_file;  { prepares to read packed bytes in gf_file }
  begin reset(gf_file); gf_loc ← 0;
  end;
```

40. To prepare the *pk_file* for output, we *rewrite* it.

```
procedure open_pk_file;  { prepares to write packed bytes in pk_file }
  begin rewrite(pk_file); pk_loc ← 0; pk_open ← true;
  end;
```

41. The variable *pk_loc* contains the number of the byte about to be written to the *pk_file*, and *gf_loc* is the byte about to be read from the *gf_file*. Also, *pk_open* indicates that the packed file has been opened and is ready for output.

⟨ Globals in the outer block 11 ⟩ +≡
```
pk_loc: integer;  { where we are about to write, in pk_file }
gf_loc: integer;  { where are we in the gf_file }
pk_open: boolean;  { is the packed file open? }
```

42. We do not open the *pk_file* until after the postamble of the *gf_file* has been read. This can be used, for instance, to calculate a resolution to put in the suffix of the *pk_file* name. This also means, however, that specials in the postamble (which METAFONT never generates) do not get sent to the *pk_file*.

⟨ Set initial values 12 ⟩ +≡
```
pk_open ← false;
```

43. We shall use two simple functions to read the next byte or bytes from *gf_file*. We either need to get an individual byte or a set of four bytes.

```
function gf_byte: integer; { returns the next byte, unsigned }
  var b: eight_bits;
  begin if eof(gf_file) then bad_gf('Unexpected␣end␣of␣file!')
  else begin read(gf_file, b); gf_byte ← b;
    end;
  incr(gf_loc);
  end;
function gf_signed_quad: integer; { returns the next four bytes, signed }
  var a, b, c, d: eight_bits;
  begin read(gf_file, a); read(gf_file, b); read(gf_file, c); read(gf_file, d);
  if a < 128 then gf_signed_quad ← ((a * 256 + b) * 256 + c) * 256 + d
  else gf_signed_quad ← (((a − 256) * 256 + b) * 256 + c) * 256 + d;
  gf_loc ← gf_loc + 4;
  end;
```

44. We also need a few routines to write data to the PK file. We write data in 4-, 8-, 16-, 24-, and 32-bit chunks, so we define the appropriate routines. We must be careful not to let the sign bit mess us up, as some Pascals implement division of a negative integer differently.

```
procedure pk_byte(a : integer);
  begin if pk_open then
    begin if a < 0 then a ← a + 256;
    write(pk_file, a); incr(pk_loc);
    end;
  end;
procedure pk_halfword(a : integer);
  begin if a < 0 then a ← a + 65536;
  write(pk_file, a div 256); write(pk_file, a mod 256); pk_loc ← pk_loc + 2;
  end;
procedure pk_three_bytes(a : integer);
  begin write(pk_file, a div 65536 mod 256); write(pk_file, a div 256 mod 256); write(pk_file, a mod 256);
  pk_loc ← pk_loc + 3;
  end;
procedure pk_word(a : integer);
  var b: integer;
  begin if pk_open then
    begin if a < 0 then
      begin a ← a + ´10000000000; a ← a + ´10000000000; b ← 128 + a div 16777216;
      end
    else b ← a div 16777216;
    write(pk_file, b); write(pk_file, a div 65536 mod 256); write(pk_file, a div 256 mod 256);
    write(pk_file, a mod 256); pk_loc ← pk_loc + 4;
    end;
  end;
procedure pk_nyb(a : integer);
  begin if bit_weight = 16 then
    begin output_byte ← a * 16; bit_weight ← 1;
    end
  else begin pk_byte(output_byte + a); bit_weight ← 16;
    end;
  end;
```

45. We need the globals *bit_weight* and *output_byte* for buffering.

```
⟨ Globals in the outer block 11 ⟩ +≡
bit_weight: integer; { output bit weight }
output_byte: integer; { output byte for pk file }
```

46. Finally we come to the routines that are used for random access of the *gf_file*. To correctly find and read the postamble of the file, we need two routines, one to find the length of the *gf_file*, and one to position the *gf_file*. We assume that the first byte of the file is numbered zero.

Such routines are, of course, highly system dependent. They are implemented here in terms of two assumed system routines called *set_pos* and *cur_pos*. The call $set\_pos(f,n)$ moves to item n in file f, unless n is negative or larger than the total number of items in f; in the latter case, $set\_pos(f,n)$ moves to the end of file f. The call $cur\_pos(f)$ gives the total number of items in f, if $eof(f)$ is true; we use *cur_pos* only in such a situation.

```
procedure find_gf_length;
  begin set_pos(gf_file, -1); gf_len ← cur_pos(gf_file);
  end;
procedure move_to_byte(n : integer);
  begin set_pos(gf_file, n); gf_loc ← n;
  end;
```

47. The global *gf_len* contains the final total length of the *gf_file*.

```
⟨ Globals in the outer block 11 ⟩ +≡
gf_len: integer;  { length of gf_file }
```

48. Plan of attack. It would seem at first that converting a GF file to PK format should be relatively easy, since they both use a form of run-encoding. Unfortunately, several idiosyncrasies of the GF format make this conversion slightly cumbersome. The GF format separates the raster information from the escapement values and TFM widths; the PK format combines all information about a single character into one character packet. The GF run-encoding is on a row-by-row basis, and the PK format is on a glyph basis, as if all of the raster rows in the glyph were concatenated into one long row. The encoding of the run-counts in the GF files is fixed, whereas the PK format uses a dynamic encoding scheme that must be adjusted for each character. And, finally, any repeated rows can be marked and sent with a single command in the PK format.

There are four major steps in the conversion process. First, the postamble of the *gf_file* is found and read, and the data from the character locators is stored in memory. Next, the preamble of the *pk_file* is written. The third and by far the most difficult step reads the raster representation of all of the characters from the GF file, packs them, and writes them to the *pk_file*. Finally, the postamble is written to the *pk_file*.

The conversion of the character raster information from the *gf_file* to the format required by the *pk_file* takes several smaller steps. The GF file is read, the commands are interpreted, and the run counts are stored in the working *row* array. Each row is terminated by a *end_of_row* value, and the character glyph is terminated by an *end_of_char* value. Then, this representation of the character glyph is scanned to determine the minimum bounding box in which it will fit, correcting the *min_m*, *max_m*, *min_n*, and *max_n* values, and calculating the offset values. The third sub-step is to restructure the row list from a list based on rows to a list based on the entire glyph. Then, an optimal value of *dyn_f* is calculated, and the final size of the counts is found for the PK file format, and compared with the bit-wise packed glyph. If the run-encoding scheme is shorter, the character is written to the *pk_file* as row counts; otherwise, it is written using a bit-packed scheme.

To save various information while the GF file is being loaded, we need several arrays. The *tfm_width*, *dx*, and *dy* arrays store the obvious values. The *status* array contains the current status of the particular character. A value of 0 indicates that the character has never been defined; a 1 indicates that the character locator for that character was read in; and a 2 indicates that the raster information for at least one character was read from the *gf_file* and written to the *pk_file*. The *row* array contains row counts. It is filled anew for each character, and is used as a general workspace. The GF counts are stored starting at location 2 in this array, so that the PK counts can be written to the same array, overwriting the GF counts, without destroying any counts before they are used. (A possible repeat count in the first row might make the first row of the PK file one count longer; all succeeding rows are guaranteed to be the same length or shorter because of the *end_of_row* flags in the GF format that are unnecessary in the PK format.)

```
define virgin ≡ 0  { never heard of this character yet }
define located ≡ 1  { locators read for this character }
define sent ≡ 2  { at least one of these characters has been sent }
```

```
⟨ Globals in the outer block 11 ⟩ +≡
tfm_width: array [0 .. 255] of integer;  { the TFM widths of characters }
dx, dy: array [0 .. 255] of integer;  { the horizontal and vertical escapements }
status: array [0 .. 255] of virgin .. sent;  { character status }
row: array [0 .. max_row] of integer;  { the row counts for working }
```

49. Here we initialize all of the character *status* values to *virgin*.

```
⟨ Set initial values 12 ⟩ +≡
  for i ← 0 to 255 do status[i] ← virgin;
```

50. And, finally, we need to define the *end_of_row* and *end_of_char* values. These cannot be values that can be taken on either by legitimate run counts, even when wrapping around an entire character. Nor can they be values that repeat counts can take on. Since repeat counts can be arbitrarily large, we restrict ourselves to negative values whose absolute values are greater than the largest possible repeat count.

```
define end_of_row ≡ (−99999)  { indicates the end of a row }
define end_of_char ≡ (−99998)  { indicates the end of a character }
```

51. Reading the generic font file. There are two major procedures in this program that do all of the work. The first is *convert_gf_file*, which interprets the `GF` commands and puts row counts into the *row* array. The second, which we only anticipate at the moment, actually packs the row counts into nybbles and writes them to the packed file.

```
⟨ Packing procedures 62 ⟩;
procedure convert_gf_file;
  var i, j, k: integer;  { general purpose indices }
    gf_com: integer;  { current gf command }
    ⟨ Locals to convert_gf_file 58 ⟩
    begin open_gf_file;
    if gf_byte ≠ pre then bad_gf('First␣byte␣is␣not␣preamble');
    if gf_byte ≠ gf_id_byte then bad_gf('Identification␣byte␣is␣incorrect');
    ⟨ Find and interpret postamble 60 ⟩;
    move_to_byte(2); open_pk_file; ⟨ Write preamble 81 ⟩;
    repeat gf_com ← gf_byte;
      case gf_com of
      boc, boc1: ⟨ Interpret character 54 ⟩;
        ⟨ Specials and no_op cases 53 ⟩;
      post: ;  { we will actually do the work for this one later }
      othercases bad_gf('Unexpected␣', gf_com : 1, '␣command␣between␣characters')
      endcases;
    until gf_com = post;
    ⟨ Write postamble 84 ⟩;
    end;
```

52. We need a few easy macros to expand some case statements:

```
define four_cases(#) ≡ #, # + 1, # + 2, # + 3
define sixteen_cases(#) ≡ four_cases(#), four_cases(# + 4), four_cases(# + 8), four_cases(# + 12)
define sixty_four_cases(#) ≡ sixteen_cases(#), sixteen_cases(# + 16), sixteen_cases(# + 32),
    sixteen_cases(# + 48)
define one_sixty_five_cases(#) ≡ sixty_four_cases(#), sixty_four_cases(# + 64), sixteen_cases(# + 128),
    sixteen_cases(# + 144), four_cases(# + 160), # + 164
```

53. In this program, all special commands are passed unchanged and any *no_op* bytes are ignored, so we write some code to handle these:

```
⟨ Specials and no_op cases 53 ⟩ ≡
four_cases(xxx1): begin pk_byte(gf_com − xxx1 + pk_xxx1); i ← 0;
  for j ← 0 to gf_com − xxx1 do
    begin k ← gf_byte; pk_byte(k); i ← i ∗ 256 + k;
    end;
  for j ← 1 to i do pk_byte(gf_byte);
  end;
yyy: begin pk_byte(pk_yyy); pk_word(gf_signed_quad);
  end;
no_op:
```

This code is used in sections 51, 57, and 60.

54. Now we need the routine that handles the character commands. Again, only a subset of the gf commands are permissible inside character definitions, so we only look for these.

```
⟨ Interpret character 54 ⟩ ≡
  begin if gf_com = boc then
    begin gf_ch ← gf_signed_quad; i ← gf_signed_quad; { dispose of back pointer }
    min_m ← gf_signed_quad; max_m ← gf_signed_quad; min_n ← gf_signed_quad;
    max_n ← gf_signed_quad;
    end
  else begin gf_ch ← gf_byte; i ← gf_byte; max_m ← gf_byte; min_m ← max_m − i; i ← gf_byte;
    max_n ← gf_byte; min_n ← max_n − i;
    end;
  d_print_ln('Character␣', gf_ch : 1);
  if gf_ch ≥ 0 then gf_ch_mod_256 ← gf_ch mod 256
  else gf_ch_mod_256 ← 255 − ((−(1 + gf_ch)) mod 256);
  if status[gf_ch_mod_256] = virgin then bad_gf('no␣character␣locator␣for␣character␣', gf_ch : 1);
  ⟨ Convert character to packed form 57 ⟩;
  end
```

This code is used in section 51.

55. Communication between the procedures *convert_gf_file* and *pack_and_send_character* is done with a few global variables.

```
⟨ Globals in the outer block 11 ⟩ +≡
gf_ch: integer; { the character we are working with }
gf_ch_mod_256: integer; { locator pointer }
pred_pk_loc: integer; { where we predict the end of the character to be. }
max_n, min_n: integer; { the maximum and minimum horizontal rows }
max_m, min_m: integer; { the maximum and minimum vertical rows }
row_ptr: integer; { where we are in the row array. }
```

56. Now we are at the beginning of a character that we need the raster for. Before we get into the complexities of decoding the *paint*, *skip*, and *new_row* commands, let's define a macro that will help us fill up the *row* array. Note that we check that *row_ptr* never exceeds *max_row*; Instead of calling *bad_gf* directly, as this macro is repeated eight times, we simply set the *bad* flag true.

```
define put_in_rows(#) ≡
       begin if row_ptr > max_row then bad ← true
       else begin row[row_ptr] ← #; incr(row_ptr);
         end;
       end
```

57. Now we have the procedure that decodes the various commands and puts counts into the *row* array. This would be a trivial procedure, except for the *paint_0* command. Because the *paint_0* command exists, it is possible to have a sequence like *paint* 42, *paint_0*, *paint* 38, *paint_0*, *paint_0*, *paint_0*, *paint* 33, *skip_0*. This would be an entirely empty row, but if we left the zeros in the *row* array, it would be difficult to recognize the row as empty.

This type of situation probably would never occur in practice, but it is defined by the `GF` format, so we must be able to handle it. The extra code is really quite simple, just difficult to understand; and it does not cut down the speed appreciably. Our goal is this: to collapse sequences like *paint* 42, *paint_0*, *paint* 32 to a single count of 74, and to insure that the last count of a row is a black count rather than a white count. A buffer variable *extra*, and two state flags, *on* and *state*, enable us to accomplish this.

The *on* variable is essentially the *paint_switch* described in the `GF` description. If it is true, then we are currently painting black pixels. The *extra* variable holds a count that is about to be placed into the *row* array. We hold it in this array until we get a *paint* command of the opposite color that is greater than 0. If we get a *paint_0* command, then the *state* flag is turned on, indicating that the next count we receive can be added to the *extra* variable as it is the same color.

```
⟨ Convert character to packed form 57 ⟩ ≡
  begin bad ← false; row_ptr ← 2; on ← false; extra ← 0; state ← true;
  repeat gf_com ← gf_byte;
    case gf_com of
    ⟨ Cases for paint commands 59 ⟩;
    four_cases(skip0): begin i ← 0;
      for j ← 1 to gf_com − skip0 do i ← i ∗ 256 + gf_byte;
      if on = state then put_in_rows(extra);
      for j ← 0 to i do put_in_rows(end_of_row);
      on ← false; extra ← 0; state ← true;
      end;
    one_sixty_five_cases(new_row_0): begin if on = state then put_in_rows(extra);
      put_in_rows(end_of_row); on ← true; extra ← gf_com − new_row_0; state ← false;
      end;
    ⟨ Specials and no_op cases 53 ⟩;
    eoc: begin if on = state then put_in_rows(extra);
      if (row_ptr > 2) ∧ (row[row_ptr − 1] ≠ end_of_row) then put_in_rows(end_of_row);
      put_in_rows(end_of_char);
      if bad then abort('Ran␣out␣of␣internal␣memory␣for␣row␣counts!');
      pack_and_send_character; status[gf_ch_mod_256] ← sent;
      if pk_loc ≠ pred_pk_loc then abort('Internal␣error␣while␣writing␣character!');
      end;
    othercases bad_gf('Unexpected␣', gf_com : 1, '␣command␣in␣character␣definition')
    endcases;
  until gf_com = eoc;
  end
```

This code is used in section 54.

58. A few more locals used above and below:

```
⟨ Locals to convert_gf_file 58 ⟩ ≡
on: boolean; { indicates whether we are white or black }
state: boolean; { a state variable—is the next count the same race as the one in the extra buffer? }
extra: integer; { where we pool our counts }
bad: boolean; { did we run out of space? }
```

See also section 61.

This code is used in section 51.

59. ⟨ Cases for *paint* commands 59 ⟩ ≡

```
paint_0: begin state ← ¬state; on ← ¬on;
  end;
sixty_four_cases(paint_0 + 1), paint1 + 1, paint1 + 2: begin if gf_com < paint1 then i ← gf_com − paint_0
  else begin i ← 0;
    for j ← 0 to gf_com − paint1 do i ← i ∗ 256 + gf_byte;
    end;
  if state then
    begin extra ← extra + i; state ← false;
    end
  else begin put_in_rows(extra); extra ← i;
    end;
  on ← ¬on;
  end
```

This code is used in section 57.

60. Our last remaining task is to interpret the postamble commands. The only things that may appear in the postamble are *post_post*, *char_loc*, *char_loc0*, and the special commands. Note that any special commands that might appear in the postamble are not written to the *pk_file*. Since METAFONT does not generate special commands in the postamble, this should not be a major difficulty.

```
⟨ Find and interpret postamble 60 ⟩ ≡
  find_gf_length;
  if gf_len < 8 then bad_gf('only␣', gf_len : 1, '␣bytes␣long');
  post_loc ← gf_len − 4;
  repeat if post_loc = 0 then bad_gf('all␣223''s');
    move_to_byte(post_loc); k ← gf_byte; decr(post_loc);
  until k ≠ 223;
  if k ≠ gf_id_byte then bad_gf('ID␣byte␣is␣', k : 1);
  if post_loc < 5 then bad_gf('post␣location␣is␣', post_loc : 1);
  move_to_byte(post_loc − 3); q ← gf_signed_quad;
  if (q < 0) ∨ (q > post_loc − 3) then bad_gf('post␣pointer␣is␣', q : 1);
  move_to_byte(q); k ← gf_byte;
  if k ≠ post then bad_gf('byte␣at␣', q : 1, '␣is␣not␣post');
  i ← gf_signed_quad; { skip over junk }
  design_size ← gf_signed_quad; check_sum ← gf_signed_quad; hppp ← gf_signed_quad;
  h_mag ← round(hppp ∗ 72.27/65536); vppp ← gf_signed_quad;
  if hppp ≠ vppp then print_ln('Odd␣aspect␣ratio!');
  i ← gf_signed_quad; i ← gf_signed_quad; { skip over junk }
  i ← gf_signed_quad; i ← gf_signed_quad;
  repeat gf_com ← gf_byte;
    case gf_com of
    char_loc, char_loc0: begin gf_ch ← gf_byte;
      if status[gf_ch] ≠ virgin then bad_gf('Locator␣for␣this␣character␣already␣found.');
      if gf_com = char_loc then
        begin dx[gf_ch] ← gf_signed_quad; dy[gf_ch] ← gf_signed_quad;
        end
      else begin dx[gf_ch] ← gf_byte ∗ 65536; dy[gf_ch] ← 0;
        end;
      tfm_width[gf_ch] ← gf_signed_quad; i ← gf_signed_quad; status[gf_ch] ← located;
      end;
      ⟨ Specials and no_op cases 53 ⟩;
    post_post: ;
    othercases bad_gf('Unexpected␣', gf_com : 1, '␣in␣postamble')
    endcases;
  until gf_com = post_post
```

This code is used in section 51.

61. Just a few more locals:

```
⟨ Locals to convert_gf_file 58 ⟩ +≡
hppp, vppp: integer; { horizontal and vertical pixels per point }
q: integer; { quad temporary }
post_loc: integer; { where the postamble was }
```

62. Converting the counts to packed format. This procedure is passed the set of row counts from the GF file. It writes the character to the PK file. First, the minimum bounding box is determined. Next, the row-oriented count list is converted to a count list based on the entire glyph. Finally, we calculate the optimal *dyn_f* and send the character.

```
⟨ Packing procedures 62 ⟩ ≡
procedure pack_and_send_character;
  var i, j, k: integer;  { general indices }
    ⟨ Locals to pack_and_send_character 65 ⟩
    begin ⟨ Scan for bounding box 63 ⟩;
    ⟨ Convert row-list to glyph-list 64 ⟩;
    ⟨ Calculate dyn_f and packed size and write character 68 ⟩;
    end
```

This code is used in section 51.

63. Now we have the row counts in our *row* array. To find the real *max_n*, we look for the first non-*end_of_row* value in the *row*. If it is an *end_of_char*, the entire character is blank. Otherwise, we first eliminate all of the blank rows at the end of the character. Next, for each remaining row, we check the first white count for a new *min_m*, and the total length of the row for a new *max_m*.

```
⟨ Scan for bounding box 63 ⟩ ≡
  i ← 2; decr(row_ptr);
  while row[i] = end_of_row do incr(i);
  if row[i] ≠ end_of_char then
    begin max_n ← max_n − i + 2;
    while row[row_ptr − 2] = end_of_row do
      begin decr(row_ptr); row[row_ptr] ← end_of_char;
      end;
    min_n ← max_n + 1; extra ← max_m − min_m + 1; max_m ← 0; j ← i;
    while row[j] ≠ end_of_char do
      begin decr(min_n);
      if row[j] ≠ end_of_row then
        begin k ← row[j];
        if k < extra then extra ← k;
        incr(j);
        while row[j] ≠ end_of_row do
          begin k ← k + row[j]; incr(j);
          end;
        if max_m < k then max_m ← k;
        end;
      incr(j);
      end;
    min_m ← min_m + extra; max_m ← min_m + max_m − 1 − extra; height ← max_n − min_n + 1;
    width ← max_m − min_m + 1; x_offset ← −min_m; y_offset ← max_n;
    d_print_ln('W␣', width : 1, '␣H␣', height : 1, '␣X␣', x_offset : 1, '␣Y␣', y_offset : 1);
    end
  else begin height ← 0; width ← 0; x_offset ← 0; y_offset ← 0; d_print_ln('Empty␣raster.');
    end
```

This code is used in section 62.

64. We must convert the run-count array from a row orientation to a glyph orientation, with repeat counts for repeated rows. We separate this task into two smaller tasks, on a per row basis. But first, we define a new macro to help us fill up this new array. Here, we have no fear that we will run out of space, as the glyph representation is provably smaller than the rows representation.

```
define put_count(#) ≡
    begin row[put_ptr] ← #; incr(put_ptr);
    if repeat_flag > 0 then
      begin row[put_ptr] ← −repeat_flag; repeat_flag ← 0; incr(put_ptr);
      end;
    end
```

```
⟨ Convert row-list to glyph-list 64 ⟩ ≡
  put_ptr ← 0; row_ptr ← 2; repeat_flag ← 0; state ← true; buff ← 0;
  while row[row_ptr] = end_of_row do incr(row_ptr);
  while row[row_ptr] ≠ end_of_char do
    begin ⟨ Skip over repeated rows 66 ⟩;
    ⟨ Reformat count list 67 ⟩;
    end;
  if buff > 0 then put_count(buff);
  put_count(end_of_char)
```

This code is used in section 62.

65. Some more locals for *pack_and_send_character* used above:

```
⟨ Locals to pack_and_send_character 65 ⟩ ≡
extra: integer; { little buffer for count values }
put_ptr: integer; { next location to fill in row }
repeat_flag: integer; { how many times the current row is repeated }
h_bit: integer; { horizontal bit count for each row }
buff: integer; { our count accumulator }
```

See also sections 70 and 77.

This code is used in section 62.

66. In this short section of code, we are at the beginning of a new row. We scan forward, looking for repeated rows. If there are any, *repeat_flag* gets the count, and the *row_ptr* points to the beginning of the last of the repeated rows. Two points must be made here. First, we do not count all-black or all-white rows as repeated, as a large "paint" count will take care of them, and also there is no black to white or white to black transition in the row where we could insert a repeat count. That is the meaning of the big if statement that conditions this section. Secondly, the **while** $row[i] = row[j]$ **do** loop is guaranteed to terminate, as $j > i$ and the character is terminated by a unique *end_of_char* value.

```
⟨ Skip over repeated rows 66 ⟩ ≡
  i ← row_ptr;
  if (row[i] ≠ end_of_row) ∧ ((row[i] ≠ extra) ∨ (row[i + 1] ≠ width)) then
    begin j ← i + 1;
    while row[j − 1] ≠ end_of_row do incr(j);
    while row[i] = row[j] do
      begin if row[i] = end_of_row then
        begin incr(repeat_flag); row_ptr ← i + 1;
        end;
      incr(i); incr(j);
      end;
    end
```

This code is used in section 64.

67. Here we actually spit out a row. The routine is somewhat similar to the routine where we actually interpret the GF commands in the count buffering. We must make sure to keep track of how many bits have actually been sent, so when we hit the end of a row, we can send a white count for the remaining bits, and possibly add the white count of the next row to it. And, finally, we must not forget to subtract the *extra* white space at the beginning of each row from the first white count.

```
⟨ Reformat count list 67 ⟩ ≡
  if row[row_ptr] ≠ end_of_row then row[row_ptr] ← row[row_ptr] − extra;
  h_bit ← 0;
  while row[row_ptr] ≠ end_of_row do
    begin h_bit ← h_bit + row[row_ptr];
    if state then
      begin buff ← buff + row[row_ptr]; state ← false;
      end
    else if row[row_ptr] > 0 then
        begin put_count(buff); buff ← row[row_ptr];
        end
      else state ← true;
    incr(row_ptr);
    end;
  if h_bit < width then
    if state then buff ← buff + width − h_bit
    else begin put_count(buff); buff ← width − h_bit; state ← true;
      end
  else state ← false;
  incr(row_ptr)
```

This code is used in section 64.

68. Here is another piece of rather intricate code. We determine the smallest size in which we can pack the data, calculating *dyn_f* in the process. To do this, we calculate the size required if *dyn_f* is 0, and put this in *comp_size*. Then, we calculate the changes in the size for each increment of *dyn_f*, and stick these values in the *deriv* array. Finally, we scan through this array and find the final minimum value, which we then use to send the character data.

```
⟨ Calculate dyn_f and packed size and write character 68 ⟩ ≡
  for i ← 1 to 13 do deriv[i] ← 0;
  i ← 0; first_on ← row[i] = 0;
  if first_on then incr(i);
  comp_size ← 0;
  while row[i] ≠ end_of_char do ⟨ Process count for best dyn_f value 69 ⟩;
  b_comp_size ← comp_size; dyn_f ← 0;
  for i ← 1 to 13 do
    begin comp_size ← comp_size + deriv[i];
    if comp_size ≤ b_comp_size then
      begin b_comp_size ← comp_size; dyn_f ← i;
      end;
    end;
  comp_size ← (b_comp_size + 1) div 2;
  if (comp_size > (height * width + 7) div 8) ∨ (height * width = 0) then
    begin comp_size ← (height * width + 7) div 8; dyn_f ← 14;
    end;
  d_print_ln('Best␣packing␣is␣dyn_f␣of␣', dyn_f : 1, '␣with␣length␣', comp_size : 1);
  ⟨ Write character preamble 71 ⟩;
  if dyn_f ≠ 14 then ⟨ Send compressed format 75 ⟩
  else if height > 0 then ⟨ Send bit map 76 ⟩
```

This code is used in section 62.

69. When we enter this module, we have a count at $row[i]$. First, we add to the *comp_size* the number of nybbles that this count would require, assuming *dyn_f* to be zero. When *dyn_f* is zero, there are no one nybble counts, so we simply choose between two-nybble and extensible counts and add the appropriate value.

Next, we take the count value and determine the value of *dyn_f* (if any) that would cause this count to take either more or less nybbles. If a valid value for *dyn_f* exists in this range, we accumulate this change in the *deriv* array.

One special case handled here is a repeat count of one. A repeat count of one will never change the length of the raster representation, no matter what *dyn_f* is, because it is always represented by the nybble value 15.

```
⟨ Process count for best dyn_f value 69 ⟩ ≡
  begin j ← row[i];
  if j = −1 then incr(comp_size)
  else begin if j < 0 then
      begin incr(comp_size); j ← −j;
      end;
    if j < 209 then comp_size ← comp_size + 2
    else begin k ← j − 193;
      while k ≥ 16 do
        begin k ← k div 16; comp_size ← comp_size + 2;
        end;
      incr(comp_size);
      end;
    if j < 14 then decr(deriv[j])
    else if j < 209 then incr(deriv[(223 − j) div 15])
      else begin k ← 16;
        while (k ∗ 16 < j + 3) do k ← k ∗ 16;
        if j − k ≤ 192 then deriv[(207 − j + k) div 15] ← deriv[(207 − j + k) div 15] + 2;
        end;
    end;
  incr(i);
  end
```

This code is used in section 68.

70. We need a handful of locals:

```
⟨ Locals to pack_and_send_character 65 ⟩ +≡
dyn_f: integer; { packing value }
height, width: integer; { height and width of character }
x_offset, y_offset: integer; { offsets }
deriv: array [1 .. 13] of integer; { derivative }
b_comp_size: integer; { best size }
first_on: boolean; { indicates that the first bit is on }
flag_byte: integer; { flag byte for character }
state: boolean; { state variable }
on: boolean; { white or black? }
```

71. Now we write the character preamble information. First we need to determine which of the three formats we should use.

```
⟨ Write character preamble 71 ⟩ ≡
  flag_byte ← dyn_f * 16;
  if first_on then flag_byte ← flag_byte + 8;
  if (gf_ch ≠ gf_ch_mod_256) ∨ (tfm_width[gf_ch_mod_256] > 16777215) ∨ (tfm_width[gf_ch_mod_256] <
      0) ∨ (dy[gf_ch_mod_256] ≠ 0) ∨ (dx[gf_ch_mod_256] < 0) ∨ (dx[gf_ch_mod_256] mod 65536 ≠
      0) ∨ (comp_size > 196594) ∨ (width > 65535) ∨ (height > 65535) ∨ (x_offset > 32767) ∨ (y_offset >
      32767) ∨ (x_offset < −32768) ∨ (y_offset < −32768) then ⟨ Write long character preamble 72 ⟩
  else if (dx[gf_ch] > 16777215) ∨ (width > 255) ∨ (height > 255) ∨ (x_offset > 127) ∨ (y_offset >
      127) ∨ (x_offset < −128) ∨ (y_offset < −128) ∨ (comp_size > 1015) then
    ⟨ Write two-byte short character preamble 74 ⟩
  else ⟨ Write one-byte short character preamble 73 ⟩
```

This code is used in section 68.

72. If we must write a long character preamble, we adjust a few parameters, then write the data.

```
⟨ Write long character preamble 72 ⟩ ≡
  begin flag_byte ← flag_byte + 7; pk_byte(flag_byte); comp_size ← comp_size + 28; pk_word(comp_size);
  pk_word(gf_ch); pred_pk_loc ← pk_loc + comp_size; pk_word(tfm_width[gf_ch_mod_256]);
  pk_word(dx[gf_ch_mod_256]); pk_word(dy[gf_ch_mod_256]); pk_word(width); pk_word(height);
  pk_word(x_offset); pk_word(y_offset);
  end
```

This code is used in section 71.

73. Here we write a short short character preamble, with one-byte size parameters.

```
⟨ Write one-byte short character preamble 73 ⟩ ≡
  begin comp_size ← comp_size + 8; flag_byte ← flag_byte + comp_size div 256; pk_byte(flag_byte);
  pk_byte(comp_size mod 256); pk_byte(gf_ch); pred_pk_loc ← pk_loc + comp_size;
  pk_three_bytes(tfm_width[gf_ch_mod_256]); pk_byte(dx[gf_ch_mod_256] div 65536); pk_byte(width);
  pk_byte(height); pk_byte(x_offset); pk_byte(y_offset);
  end
```

This code is used in section 71.

74. Here we write an extended short character preamble, with two-byte size parameters.

```
⟨ Write two-byte short character preamble 74 ⟩ ≡
  begin comp_size ← comp_size + 13; flag_byte ← flag_byte + comp_size div 65536 + 4; pk_byte(flag_byte);
  pk_halfword(comp_size mod 65536); pk_byte(gf_ch); pred_pk_loc ← pk_loc + comp_size;
  pk_three_bytes(tfm_width[gf_ch_mod_256]); pk_halfword(dx[gf_ch_mod_256] div 65536);
  pk_halfword(width); pk_halfword(height); pk_halfword(x_offset); pk_halfword(y_offset);
  end
```

This code is used in section 71.

75. At this point, we have decided that the run-encoded format is smaller. (This is almost always the case.) We send out the data, a nybble at a time.

```
⟨ Send compressed format 75 ⟩ ≡
  begin bit_weight ← 16; max_2 ← 208 − 15 ∗ dyn_f; i ← 0;
  if row[i] = 0 then incr(i);
  while row[i] ≠ end_of_char do
    begin j ← row[i];
    if j = −1 then pk_nyb(15)
    else begin if j < 0 then
        begin pk_nyb(14); j ← −j;
        end;
      if j ≤ dyn_f then pk_nyb(j)
      else if j ≤ max_2 then
          begin j ← j − dyn_f − 1; pk_nyb(j div 16 + dyn_f + 1); pk_nyb(j mod 16);
          end
        else begin j ← j − max_2 + 15; k ← 16;
          while k ≤ j do
            begin k ← k ∗ 16; pk_nyb(0);
            end;
          while k > 1 do
            begin k ← k div 16; pk_nyb(j div k); j ← j mod k;
            end;
          end;
      end;
    incr(i);
    end;
  if bit_weight ≠ 16 then pk_byte(output_byte);
  end
```

This code is used in section 68.

76. This code is for the case where we have decided to send the character raster packed by bits. It uses the bit counts as well, sending eight at a time. Here we have a miniature packed format interpreter, as we must repeat any rows that are repeated. The algorithm to do this was a lot of fun to generate. Can you figure out how it works?

```
⟨ Send bit map 76 ⟩ ≡
  begin buff ← 0; p_bit ← 8; i ← 1; h_bit ← width; on ← false; state ← false; count ← row[0];
  repeat_flag ← 0;
  while (row[i] ≠ end_of_char) ∨ state ∨ (count > 0) do
    begin if state then
      begin count ← r_count; i ← r_i; on ← r_on; decr(repeat_flag);
      end
    else begin r_count ← count; r_i ← i; r_on ← on;
      end;
    ⟨ Send one row by bits 80 ⟩;
    if state ∧ (repeat_flag = 0) then
      begin count ← s_count; i ← s_i; on ← s_on; state ← false;
      end
    else if ¬state ∧ (repeat_flag > 0) then
        begin s_count ← count; s_i ← i; s_on ← on; state ← true;
        end;
    end;
  if p_bit ≠ 8 then pk_byte(buff);
  end
```

This code is used in section 68.

77. All of the remaining locals:

```
⟨ Locals to pack_and_send_character 65 ⟩ +≡
comp_size: integer; { length of the packed representation in bytes }
count: integer; { number of bits in current state to send }
p_bit: integer; { what bit are we about to send out? }
r_on, s_on: boolean; { state saving variables }
r_count, s_count: integer; { ditto }
r_i, s_i: integer; { and again. }
max_2: integer; { the highest count that fits in two bytes }
```

78. We make the *power* array global.

```
⟨ Globals in the outer block 11 ⟩ +≡
power: array [0 .. 8] of integer; { easy powers of two }
```

79. We initialize the power array.

```
⟨ Set initial values 12 ⟩ +≡
  power[0] ← 1;
  for i ← 1 to 8 do power[i] ← power[i − 1] + power[i − 1];
```

80. Here we are at the beginning of a row and simply output the next *width* bits. We break the possibilities up into three cases: we finish a byte but not the row, we finish a row, and we finish neither a row nor a byte. But, first, we insure that we have a *count* value.

```
⟨ Send one row by bits 80 ⟩ ≡
  repeat if count = 0 then
      begin if row[i] < 0 then
        begin if ¬state then repeat_flag ← −row[i];
        incr(i);
        end;
      count ← row[i]; incr(i); on ← ¬on;
      end;
    if (count ≥ p_bit) ∧ (p_bit < h_bit) then
      begin { we end a byte, we don't end the row }
      if on then buff ← buff + power[p_bit] − 1;
      pk_byte(buff); buff ← 0; h_bit ← h_bit − p_bit; count ← count − p_bit; p_bit ← 8;
      end
    else if (count < p_bit) ∧ (count < h_bit) then
        begin { we end neither the row nor the byte }
        if on then buff ← buff + power[p_bit] − power[p_bit − count];
        p_bit ← p_bit − count; h_bit ← h_bit − count; count ← 0;
        end
      else begin { we end a row and maybe a byte }
        if on then buff ← buff + power[p_bit] − power[p_bit − h_bit];
        count ← count − h_bit; p_bit ← p_bit − h_bit; h_bit ← width;
        if p_bit = 0 then
          begin pk_byte(buff); buff ← 0; p_bit ← 8;
          end;
        end;
  until h_bit = width
```

This code is used in section 76.

81. Now we are ready for the routine that writes the preamble of the packed file.

```
define preamble_comment ≡ 'GFtoPK␣2.4␣output␣from␣'
define comm_length = 23 { length of preamble_comment }
define from_length = 6 { length of its '␣from␣' part }
```

⟨ Write preamble 81 ⟩ ≡

```
pk_byte(pk_pre); pk_byte(pk_id); i ← gf_byte; { get length of introductory comment }
repeat if i = 0 then j ← "." else j ← gf_byte;
  decr(i); { some people think it's wise to avoid goto statements }
until j ≠ "␣"; { remove leading blanks }
incr(i); { this many bytes to copy }
if i = 0 then k ← comm_length − from_length
else k ← i + comm_length;
if k > 255 then pk_byte(255) else pk_byte(k);
for k ← 1 to comm_length do
  if (i > 0) ∨ (k ≤ comm_length − from_length) then pk_byte(xord[comment[k]]);
print('''');
for k ← 1 to i do
  begin if k > 1 then j ← gf_byte;
  print(xchr[j]);
  if k < 256 − comm_length then pk_byte(j);
  end;
print_ln('''');
pk_word(design_size); pk_word(check_sum); pk_word(hppp); pk_word(vppp)
```

This code is used in section 51.

82. Of course, we need an array to hold the comment.

⟨ Globals in the outer block 11 ⟩ +≡

```
comment: packed array [1 .. comm_length] of char;
```

83. ⟨ Set initial values 12 ⟩ +≡

```
comment ← preamble_comment;
```

84. Writing the postamble is even easier.

⟨ Write postamble 84 ⟩ ≡

```
pk_byte(pk_post);
while (pk_loc mod 4 ≠ 0) do pk_byte(pk_no_op)
```

This code is used in section 51.

85. Once we are finished with the GF file, we check the status of each character to insure that each character that had a locator also had raster information.

⟨ Check for unrasterized locators 85 ⟩ ≡

```
for i ← 0 to 255 do
  if status[i] = located then print_ln('Character␣', i : 1, '␣missing␣raster␣information!')
```

This code is used in section 86.

86. Finally, the main program.

```
begin initialize; convert_gf_file; ⟨ Check for unrasterized locators 85 ⟩;
print_ln(gf_len : 1, '␣bytes␣packed␣to␣', pk_loc : 1, '␣bytes.');
final_end: end.
```

87. A few more globals.

```
⟨ Globals in the outer block 11 ⟩ +≡
check_sum: integer;   { the checksum of the file }
design_size: integer;   { the design size of the font }
h_mag: integer;   { the pixel magnification in pixels per inch }
i: integer;
```

88. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `GFtoPK` work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

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