Software

Packed (PK) Font File Format

Pixel files, the output of METAFONT79, are now considered obsolete; their use is being discouraged. This is not surprising; as a font file format, they are rather limiting. They tend to be too large, do not include device character width information, and only allow 128 characters. Yet, the generic font file format offered in their place (with new METAFONT) has some other limitations. GF files tend to be almost as large as the pixel files they replace, and the character width data is separated from the raster data. A new format, called the packed, or PK format, is therefore being introduced.

This new file format is less than half the size of the GF format. Since input/output time usually dominates execution time when reading font files, the smaller size can also lead to a performance improvement. The new format is easier to interpret than the GF format. The minimum bounding box for the bitmaps is supplied; the horizontal and vertical size do not need to be checked against the actual bits of the character. The raster data and width data for each character is given in the same character 'packet'. Finally, the font parameters (checksum, design size, etc.) are given at the beginning of the font file, rather than at the end, so random file access is not needed.

An additional advantage is that the length of each character packet is given at the beginning of the packet. This means that the character packets can be directly stored into the memory of a driver without interpretation, and they can be interpreted on a demand basis. A recent sampling of font usage at Texas A&M University on their Electrical Engineering VAX showed that 18 characters per declared font were used on the average. Thus, a driver using the packed format might not need to interpret 110 of the 128 characters in a font, on the average.

The use of the packed file format does impose an extra step of processing between METAFONT and the driver, however. Also, the packed file format is based to a large extent on the four bit nybble rather than the eight bit byte, so individual bytes often need to be split as the file is being read in. In addition, the generic font format allows **special**s and **numspecial**s within character raster definitions; the packed file format does not. METAFONT never generates **special**s inside of character raster definitions, so this should not pose any problems.

The size improvement of packed files over pixel files is almost five to one for a large collection of fonts. 323 fonts at three hundred dots per inch and various magnifications were reduced by 79% when converted to the packed form. This makes the packed format ideal for microcomputer systems or any system where disk space is at a premium.

The packed file format is part of the METAFONTware, with which it will be distributed. Programs currently on the distribution are GFtoPK, which converts generic font files to packed files; PKtoPX, which converts packed files to pixel files; PXtoPK, which converts pixel files to packed files, and PKtype, which lists and verifies a packed file.

The rest of this article was extracted from PKtype, and is a full description of the packed file format.

1. 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.

2. 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.

3. 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 musn't be negative.

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

pk_post 245. Beginning of the postamble. This command is followed by just enough *pk_no_op* commands to make the file a multiple of four bytes long; zero through three are usual, but four are also 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 $1/2^{16}$ 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., amr10) with the resolution at which the font is prepared in pixels per inch multiplied by the magnification factor, and the letters PK. For instance, amr10 at 300 dots per inch should be named AMR10.300PK; at one thousand dots per inch and magstephalf, it should be named AMR10.1095PK.

4. 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 }
```

5. 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.

6. 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.

7. 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.

8. 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).

9. 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.

10. 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.

```
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 repeat_count ≠ 0 then
        abort('Extra repeat count!');
      if i = 14 then
        repeat_count ← pk_packed_num
      else repeat_count ← 1;
      send_out(true, repeat_count);
      pk_packed_num ← pk_packed_num;
      end;
end;
```

11. 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.

12. 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 nybbles 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 16) 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 8) 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 huge 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* **mod** 4) $*$ 256 + *pl*; for the extended format, it might be calculated as (*flag* **mod** 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.)

13. 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. The current version of TEX (1.5) cannot output character codes greater than 255 anyway.

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.

14. 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.)

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

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.

15. 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.

16. 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

Tomas Rokicki
Rokicki@SU-Score.Arpa
Department of Computer Science
Stanford University
Stanford, CA 94305