The PKtype processor

(Version 2.3, 23 April 2020)

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

1. Introduction. This program reads a `PK` file, verifies that it is in the correct format, and writes it in textual format.

2. The *banner* string defined here should be changed whenever `PKtype` gets modified.

```
define banner ≡ 'This␣is␣PKtype,␣Version␣2.3'   { printed when the program starts }
```

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

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

4. Both the input and output come from binary files. On line interaction is handled through Pascal's standard *input* and *output* files. Two macros are used to write to the type file, so this output can easily be redirected.

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

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

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

This code is used in section 4.

6. These constants determine the maximum length of a file name and the length of the terminal line, as well as the widest character that can be translated.

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

This code is used in section 4.

7. Here are some macros for common programming idioms.

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

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

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

9. The character set. Like all programs written with the WEB system, PKtype 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 PKtype have therefore been copied from the analogous ones in the WEB system routines. They have been considerably simplified, since PKtype need not deal with the controversial ASCII codes less than *´40* or greater than *´176*.

⟨ Types in the outer block 9 ⟩ ≡
  ASCII_code = "␣" .. "~"; { a subrange of the integers }

See also sections 10 and 30.

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 PKtype. So we shall assume that the Pascal system being used for PKtype 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 PKtype 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 31, 33, 37, 39, 41, 47, and 51.

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 section 13.

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. 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.

15. 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.

16. 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`.

17. 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 }
define *pk_undefined* ≡ 248, 249, 250, 251, 252, 253, 254, 255

18. 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.

19. 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.

20. 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.

21. 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).

22. 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.

23. The final algorithm for decoding the run counts based on the above scheme looks 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.

```
⟨ Packed number procedure 23 ⟩ ≡
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 repeat_count ≠ 0 then abort('Second␣repeat␣count␣for␣this␣row!');
        repeat_count ← 1; { prevent recursion more than one level }
        if i = 14 then repeat_count ← pk_packed_num;
        send_out(true, repeat_count); pk_packed_num ← pk_packed_num;
        end;
  end;
```

This code is used in section 46.

24. 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.

25. 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.

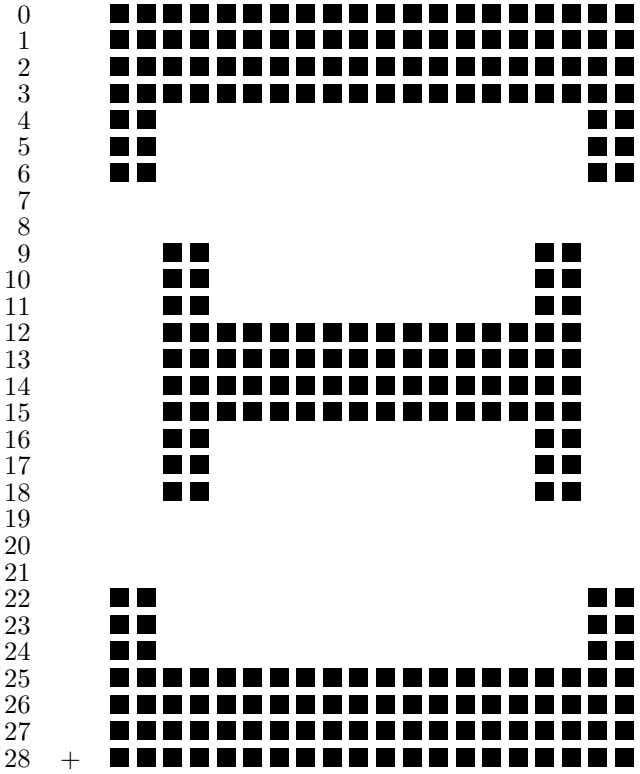
The *flag* parameter is the flag byte. The parameter *pl* (packet length) contains the offset of the byte following this character descriptor, with respect to the beginning of the *tfm* width parameter. This is given so a PK reading program can, once it has read the flag byte, packet length, and character 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 $(flag \textbf{ mod } 4) * 256 + pl$; for the short extended format, it might be calculated as $(flag \textbf{ mod } 4) * 65536 + pl$.

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 *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.)

26. 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.

27. 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.

28. 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.

29. 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`

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

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

31.
```
⟨ Globals in the outer block 11 ⟩ +≡
pk_file: byte_file;  { where the input comes from }
typ_file: text_file;  { where the final output goes }
```

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

```
procedure open_pk_file;  { prepares the input for reading }
  begin reset(pk_file, pk_name); pk_loc ← 0;
  end;
procedure open_typ_file;  { prepares to write text data to the typ_file }
  begin rewrite(typ_file, typ_name);
  end;
```

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

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

34. We also need a function that will get a single byte from the `pk` file.

```
function pk_byte: eight_bits;
  var temp: eight_bits;
  begin temp ← pk_file↑; get(pk_file); incr(pk_loc); pk_byte ← temp;
  end;
```

35. Now we are ready to open the files.

```
⟨ Open files 35 ⟩ ≡
  open_pk_file; open_typ_file; t_print_ln(banner); t_print('Input␣file:␣'); i ← 1;
  while pk_name[i] ≠ '␣' do
    begin t_print(pk_name[i]); incr(i);
    end;
  t_print_ln('␣')
```

This code is used in section 55.

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

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

37. We still need the *term_pos* variable.

```
⟨ Globals in the outer block 11 ⟩ +≡
term_pos: integer;  { current terminal position }
```

38. Now we read and check the preamble of the PK file. In the preamble, we find the *hppp*, *design_size*, *checksum*.

```
⟨ Read preamble 38 ⟩ ≡
  if pk_byte ≠ pk_pre then abort('Bad␣PK␣file:␣␣pre␣command␣missing!');
  if pk_byte ≠ pk_id then abort('Wrong␣version␣of␣PK␣file!');
  j ← pk_byte; t_print('''');
  for i ← 1 to j do t_print(xchr[pk_byte]);
  t_print_ln(''''); design_size ← get_32; t_print_ln('Design␣size␣=␣', design_size : 1);
  checksum ← get_32; t_print_ln('Checksum␣=␣', checksum : 1); hppp ← get_32; vppp ← get_32;
  t_print('Resolution:␣horizontal␣=␣', hppp : 1, '␣␣vertical␣=␣', vppp : 1);
  magnification ← round(hppp * 72.27/65536); t_print_ln('␣␣(', magnification : 1, '␣dpi)');
  if hppp ≠ vppp then print_ln('Warning:␣␣aspect␣ratio␣not␣1:1!')
```

This code is used in section 55.

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

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

40. Character unpacking. Here we simply unpack the character, writing the information we glean to the *typ_file*.

```
⟨ Unpack and write character 40 ⟩ ≡
  t_print((pk_loc − 1) : 1, ´:␣␣Flag␣byte␣=␣´, flag_byte : 1); dyn_f ← flag_byte div 16;
  flag_byte ← flag_byte mod 16; turn_on ← flag_byte ≥ 8;
  if turn_on then flag_byte ← flag_byte − 8;
  if flag_byte = 7 then ⟨ Read long character preamble 42 ⟩
  else if flag_byte > 3 then ⟨ Read extended short character preamble 43 ⟩
    else ⟨ Read short character preamble 44 ⟩;
  t_print_ln(´␣␣Character␣=␣´, car : 1, ´␣␣Packet␣length␣=␣´, packet_length : 1);
  t_print_ln(´␣␣Dynamic␣packing␣variable␣=␣´, dyn_f : 1);
  t_print(´␣␣TFM␣width␣=␣´, tfm_width : 1, ´␣␣dx␣=␣´, dx : 1);
  if dy ≠ 0 then t_print_ln(´␣␣dy␣=␣´, dy : 1)
  else t_print_ln(´␣␣´);
  t_print_ln(´␣␣Height␣=␣´, height : 1, ´␣␣Width␣=␣´, width : 1, ´␣␣X-offset␣=␣´, x_off : 1,
      ´␣␣Y-offset␣=␣´, y_off : 1); ⟨ Read and translate raster description 48 ⟩;
  if end_of_packet ≠ pk_loc then abort(´Bad␣PK␣file:␣Bad␣packet␣length!´)
```

This code is used in section 55.

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

```
⟨ Globals in the outer block 11 ⟩ +≡
i, j: integer; { index pointers }
flag_byte: integer; { the byte that introduces the character definition }
end_of_packet: integer; { where we expect the end of the packet to be }
width, height: integer; { width and height of character }
x_off, y_off: integer; { x and y offsets of character }
tfm_width: integer; { character tfm width }
tfms: array [0 .. 255] of integer; { character tfm widths }
dx, dy: integer; { escapement values }
dxs, dys: array [0 .. 255] of integer; { escapement values }
status: array [0 .. 255] of boolean; { has the character been seen? }
dyn_f: integer; { dynamic packing variable }
car: integer; { the character we are reading }
packet_length: integer; { the length of the character packet }
```

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

```
⟨ Read long character preamble 42 ⟩ ≡
  begin packet_length ← get_32; car ← get_32; end_of_packet ← packet_length + pk_loc;
  packet_length ← packet_length + 9; tfm_width ← get_32; dx ← get_32; dy ← get_32; width ← get_32;
  height ← get_32; x_off ← get_32; y_off ← get_32;
  end
```

This code is used in section 40.

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

```
⟨ Read extended short character preamble 43 ⟩ ≡
  begin packet_length ← (flag_byte − 4) ∗ 65536 + get_16; car ← pk_byte;
  end_of_packet ← packet_length + pk_loc; packet_length ← packet_length + 4; i ← pk_byte;
  tfm_width ← i ∗ 65536 + get_16; dx ← get_16 ∗ 65536; dy ← 0; width ← get_16; height ← get_16;
  x_off ← get_16; y_off ← get_16;
  if x_off > 32767 then x_off ← x_off − 65536;
  if y_off > 32767 then y_off ← y_off − 65536;
  end
```

This code is used in section 40.

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

```
⟨ Read short character preamble 44 ⟩ ≡
  begin packet_length ← flag_byte ∗ 256 + pk_byte; car ← pk_byte; end_of_packet ← packet_length + pk_loc;
  packet_length ← packet_length + 3; i ← pk_byte; tfm_width ← i ∗ 65536 + get_16; dx ← pk_byte ∗ 65536;
  dy ← 0; width ← pk_byte; height ← pk_byte; x_off ← pk_byte; y_off ← pk_byte;
  if x_off > 127 then x_off ← x_off − 256;
  if y_off > 127 then y_off ← y_off − 256;
  end
```

This code is used in section 40.

45. Now we have the most important part of the program, where we actually interpret the commands in the raster description. First of all, we need a procedure to get a single nybble from the file, as well as one to get a single bit.

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

46. We also need a function to write output to the screen. We put as many counts on a line as possible, to reduce the volume of output. Each count will appear as a number, with white counts enclosed by parentheses and repeat counts by brackets.

```
procedure send_out(repeat_count : boolean; value : integer);
  var i, len: integer;
  begin i ← 10; len ← 1;
  while value ≥ i do
    begin incr(len); i ← i * 10;
    end;
  if repeat_count ∨ ¬turn_on then len ← len + 2;
  if term_pos + len > 78 then
    begin term_pos ← len + 2; t_print_ln(´␣´); t_print(´␣␣´);
    end
  else term_pos ← term_pos + len;
  if repeat_count then t_print(´[´, value : 1, ´]´)
  else if turn_on then t_print(value : 1)
    else t_print(´(´, value : 1, ´)´);
  end; ⟨ Packed number procedure 23 ⟩
```

47. Now, the globals to help communication between these procedures.

```
⟨ Globals in the outer block 11 ⟩ +≡
input_byte: eight_bits; { the byte we are currently decimating }
bit_weight: eight_bits; { weight of the current bit }
nybble: eight_bits; { the current nybble }
```

48. And the main procedure.

```
⟨ Read and translate raster description 48 ⟩ ≡
  bit_weight ← 0;
  if dyn_f = 14 then ⟨ Get raster by bits 49 ⟩
  else ⟨ Create normally packed raster 50 ⟩
```

This code is used in section 40.

49. If $dyn\_f = 14$, then we need to get the raster representation one bit at a time.

```
⟨ Get raster by bits 49 ⟩ ≡
  begin for i ← 1 to height do
    begin t_print(´␣␣´);
    for j ← 1 to width do
      if get_bit then t_print(´*´)
      else t_print(´.´);
    t_print_ln(´␣´);
    end;
  end
```

This code is used in section 48.

50. Otherwise, we translate the bit counts into the raster rows. *count* contains the number of bits of the current color, and *turn_on* indicates whether or not they should be black. *rows_left* contains the number of rows to be sent.

```
⟨ Create normally packed raster 50 ⟩ ≡
  begin term_pos ← 2; t_print(´␣␣´); rows_left ← height; h_bit ← width; repeat_count ← 0;
  while rows_left > 0 do
    begin count ← pk_packed_num; send_out(false, count);
    if count ≥ h_bit then
      begin rows_left ← rows_left − repeat_count − 1; repeat_count ← 0; count ← count − h_bit;
      h_bit ← width; rows_left ← rows_left − count div width; count ← count mod width;
      end;
    h_bit ← h_bit − count; turn_on ← ¬turn_on;
    end;
  t_print_ln(´␣´);
  if (rows_left ≠ 0) ∨ (h_bit ≠ width) then abort(´Bad␣PK␣file:␣More␣bits␣than␣required!´);
  end
```

This code is used in section 48.

51. We need to declare the repeat flag, bit counter, and color flag here.

```
⟨ Globals in the outer block 11 ⟩ +≡
repeat_count: integer; { how many times to repeat the next row? }
rows_left: integer; { how many rows left? }
turn_on: boolean; { are we black here? }
h_bit: integer; { what is our horizontal position? }
count: integer; { how many bits of current color left? }
```

52. If any specials are found, we write them out here.

```
  define four_cases(#) ≡ #, # + 1, # + 2, # + 3
procedure skip_specials;
  var i, j: integer;
  begin repeat flag_byte ← pk_byte;
    if flag_byte ≥ 240 then
      case flag_byte of
      four_cases(pk_xxx1): begin t_print((pk_loc − 1) : 1, ´:␣␣Special:␣´´´); i ← 0;
        for j ← pk_xxx1 to flag_byte do i ← 256 ∗ i + pk_byte;
        for j ← 1 to i do t_print(xchr[pk_byte]);
        t_print_ln(´´´´);
        end;
      pk_yyy: t_print_ln((pk_loc − 1) : 1, ´:␣␣Num␣special:␣´, get_32 : 1);
      pk_post: t_print_ln((pk_loc − 1) : 1, ´:␣␣Postamble´);
      pk_no_op: t_print_ln((pk_loc − 1) : 1, ´:␣␣No␣op´);
      pk_pre, pk_undefined: abort(´Unexpected␣´, flag_byte : 1, ´!´);
      endcases;
  until (flag_byte < 240) ∨ (flag_byte = pk_post);
  end;
```

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

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

54.

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

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

```
  begin initialize; dialog; ⟨ Open files 35 ⟩;
  ⟨ Read preamble 38 ⟩;
  skip_specials;
  while flag_byte ≠ pk_post do
    begin ⟨ Unpack and write character 40 ⟩;
    skip_specials;
    end;
  j ← 0;
  while ¬eof(pk_file) do
    begin i ← pk_byte;
    if i ≠ pk_no_op then abort('Bad␣byte␣at␣end␣of␣file:␣', i : 1);
    t_print_ln((pk_loc − 1) : 1, ':␣␣No␣op');  incr(j);
    end;
  t_print_ln(pk_loc : 1, '␣bytes␣read␣from␣packed␣file.');
final_end: end.
```

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

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