The GFtoPK processor

(Version 2.4, 06 January 2014)
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:

<table>
<thead>
<tr>
<th>Font</th>
<th>Resolution</th>
<th>GF size</th>
<th>PK size</th>
<th>Reduction factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmr10</td>
<td>300</td>
<td>13200</td>
<td>5484</td>
<td>42%</td>
</tr>
<tr>
<td>cmr10</td>
<td>360</td>
<td>15342</td>
<td>6496</td>
<td>42%</td>
</tr>
<tr>
<td>cmr10</td>
<td>432</td>
<td>18120</td>
<td>7808</td>
<td>43%</td>
</tr>
<tr>
<td>cmr10</td>
<td>511</td>
<td>21020</td>
<td>9440</td>
<td>45%</td>
</tr>
<tr>
<td>cmr10</td>
<td>622</td>
<td>24880</td>
<td>11492</td>
<td>46%</td>
</tr>
<tr>
<td>cminch</td>
<td>300</td>
<td>48764</td>
<td>22076</td>
<td>45%</td>
</tr>
</tbody>
</table>

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.

```c
#define my_name ≡ 'gftopk'
#define banner ≡ 'This is GFtoPK, Version 2.4'  { printed when the program starts }
```

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. Since the terminal output is really not very interesting, it is produced only when the −v command line flag is presented.

```c
#define print(#{
    if verbose then write(stdout, #)
#define print_ln(#{
    if verbose then writeLn(stdout, #)
```

program GFtoPK(*gf_file, pk_file, output*);

const ⟨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 initial main row array }

This code is used in section 4*.
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 we might want to abort the program with an error message.

```
define abort(#) ≡
    begin write ln(stderr,#); uexit(1);
    end

define bad_gf(#) ≡ abort(ʻBad GF file:ʻ#,ʻ!ʻ)
```
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.

```plaintext
define char ≡ 0 .. 255
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;`
In C, we do path searching based on the user’s environment or the default paths.

```c
procedure open_gf_file;  { prepares to read packed bytes in gf_file }
    begin
        gf_file ← kpse_open_file(gf_name, kpse_gf_format);
        gf_loc ← 0;
    end;
end;
```

To prepare the pk_file for output, we rewrite it.

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

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.

Output is handled through `putbyte` which is supplied by web2c.

```c
define pk_byte(#) ≡
    begin
        putbyte(#, pk_file);
        incr(pk_loc)
    end
```

```c
procedure pk_halfword(a : integer);
    begin
        if a < 0 then
            begin
                a ← a + 65536;
                putbyte(a div 256, pk_file);
                putbyte(a mod 256, pk_file);
                pk_loc ← pk_loc + 2;
            end
        else
            begin
                b ← a div 16777216;
                putbyte(b, pk_file);
                putbyte(a mod 256, pk_file);
                pk_loc ← pk_loc + 3;
            end;
    end;
end;
```

```c
procedure pk_three_bytes(a : integer);
    begin
        putbyte(a div 65536 mod 256, pk_file);
        putbyte(a div 256 mod 256, pk_file);
        putbyte(a mod 256, pk_file);
        pk_loc ← pk_loc + 3;
    end;
end;
```

```c
procedure pk_word(a : integer);
    var b: integer;
    begin
        if a < 0 then
            begin
                a ← a + 10000000000; a ← a + 10000000000; b ← 128 + a div 16777216;
            end
        else
            begin
                b ← a div 16777216;
                putbyte(b, pk_file);
                putbyte(a div 65536 mod 256, pk_file);
                putbyte(a div 256 mod 256, pk_file);
                putbyte(a mod 256, pk_file);
                pk_loc ← pk_loc + 4;
            end;
    end;
end;
```

```c
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;
end;
```
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.

\[
\begin{align*}
\text{define } & \quad \text{find_gf_length } \equiv \text{gf_len } \leftarrow \text{gf_length} \\
\text{function } & \quad \text{gf_length: integer;} \\
& \quad \begin{align*}
\text{begin } & \quad \text{xfseek(} \text{gf_file, } 0, 2, \text{gf_name); } \\
& \quad \quad \text{gf_len } \leftarrow \text{xftell(} \text{gf_file, } \text{gf_name); } \\
\text{end;}
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{procedure } & \quad \text{move_to_byte(} n : \text{integer); } \\
& \quad \begin{align*}
\text{begin } & \quad \text{xfseek(} \text{gf_file, } n, 0, \text{gf_name); } \\
\text{end;}
\end{align*}
\end{align*}
\]
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 an 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 reconstruct 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.)

\[
\text{define } \text{virgin} \equiv 0 \quad \{ \text{never heard of this character yet} \}
\]
\[
\text{define } \text{located} \equiv 1 \quad \{ \text{locators read for this character} \}
\]
\[
\text{define } \text{sent} \equiv 2 \quad \{ \text{at least one of these characters has been sent} \}
\]
\[
\text{(Globals in the outer block 11)} +\equiv
\text{tfm_width: array [0..255] of integer; } \{ \text{the TFM widths of characters} \}
\text{dx, dy: array [0..255] of integer; } \{ \text{the horizontal and vertical escapements} \}
\text{status: array [0..255] of virgin .. sent; } \{ \text{character status} \}
\text{row: ↑integer; } \{ \text{the row counts for working} \}
\text{max_row: integer; } \{ \text{largest index in the main row array} \}
\]

Plan of attack. 49*

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

\[
\text{(Set initial values 12)} +\equiv
\text{row } \leftarrow \text{malloc_array}(\text{integer, } \text{MAX\_ROW}); \text{max\_row } \leftarrow \text{MAX\_ROW};
\text{for } i \leftarrow 0 \text{ to 255 do status}[i] \leftarrow \text{virgin};
\]
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 row_overflow;
  var new_row: integer;
  begin new_row ← max_row + MAX_ROW;
    print_in(’Reallocated_row_array_to_u’, new_row : 1, ’u_items_from_u’, max_row : 1,’.’);
    row ← xrealloc_array(row, integer, new_row); max_row ← new_row;
  end;

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_isincorrect’);
    ⟨Find and interpret postamble 60⟩;
    move_to_byte(2); open_pk_file; ⟨Write preamble 81*⟩;
    repeat gf_com ← gf_byte; do_the_rows ← false;
      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,’u_command_between_u_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), sixty_cases(# + 48)

define thirty_seven_cases(#) ≡ sixteen_cases(#), sixteen_cases(# + 16), four_cases(# + 32), # + 36

define new_row_64 = new_row_0 + 64

define new_row_128 = new_row_64 + 64

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 row_overflow;
    row[row_ptr] ← #; incr(row_ptr);
  end
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_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{align*}
\text{begin} & \quad \text{row_ptr} \leftarrow 2; \quad \text{on} \leftarrow \text{false}; \quad \text{extra} \leftarrow 0; \quad \text{state} \leftarrow \text{true}; \\
\text{repeat} & \quad \text{gf_com} \leftarrow \text{gf_byte}; \\
\text{case} & \quad \text{gf_com of} \\
& \quad \text{four_cases}((\text{skip}0)): \quad \text{begin} \; i \leftarrow 0; \\
& \quad \quad \text{for} \; j \leftarrow 1 \; \text{to} \; \text{gf_com} - \text{skip}0 \; \text{do} \; i \leftarrow i \cdot 256 + \text{gf_byte}; \\
& \quad \quad \text{if} \; \text{on} = \text{state} \; \text{then} \; \text{put_in_rows(extra)}; \\
& \quad \quad \text{for} \; j \leftarrow 0 \; \text{to} \; i \; \text{do} \; \text{put_in_rows(\text{end_of_row})}; \\
& \quad \quad \text{on} \leftarrow \text{false}; \quad \text{extra} \leftarrow 0; \quad \text{state} \leftarrow \text{true}; \\
& \quad \text{end}; \\
& \quad \text{sixty_four_cases(new_row_0)}: \quad \text{do_the_rows} \leftarrow \text{true}; \\
& \quad \text{sixty_four_cases(new_row_64)}: \quad \text{do_the_rows} \leftarrow \text{true}; \\
& \quad \text{thirty_seven_cases(new_row_128)}: \quad \text{do_the_rows} \leftarrow \text{true}; \\
& \quad \text{\{Specials and no_op cases 53\}}; \\
& \quad \text{eoc}: \quad \text{\{begin if on = state then put_in_rows(extra)\}}; \\
& \quad \quad \text{if} \; (\text{row_ptr} > 2) \wedge (\text{row}[\text{row_ptr} - 1] \neq \text{end_of_row}) \; \text{then} \; \text{put_in_rows(\text{end_of_row})}; \\
& \quad \quad \text{put_in_rows(\text{end_of_char})}; \quad \text{pack_and_send_character}; \quad \text{status[\text{ch_mod_256}] left sent}; \\
& \quad \quad \text{if} \; \text{pk_loc} \neq \text{pred_pk_loc} \; \text{then} \; \text{abort(\text{"Internal_error while writing character!"})}; \\
& \quad \text{end}; \\
& \quad \text{othercases bad_gf(\text{"Unexpected\_i\_command\_in\_character\_definition\"})} \\
& \quad \text{endcases}; \\
& \quad \text{if} \; \text{do_the_rows} \; \text{then} \\
& \quad \quad \text{\{begin do_the_rows left false\}}; \\
& \quad \quad \text{if} \; \text{on} = \text{state} \; \text{then} \; \text{put_in_rows(extra)}; \\
& \quad \quad \text{put_in_rows(\text{end_of_row})}; \quad \text{on} \leftarrow \text{true}; \quad \text{extra} \leftarrow \text{gf_com} - \text{new_row_0}; \quad \text{state} \leftarrow \text{false}; \\
& \quad \; \text{end}; \\
& \quad \text{until} \; \text{gf_com} = \text{eoc}; \\
& \text{end}
\end{align*}
\]

This code is used in section 54.
A few more locals used above and below:

\[
\langle \text{Locals to convert_gf_file 58*} \rangle \equiv \\
\text{do_the_rows: boolean;} \\
\text{on: boolean;} \quad \{ \text{indicates whether we are white or black} \} \\
\text{state: boolean;} \quad \{ \text{a state variable— is the next count the same race as the one in the extra buffer?} \} \\
\text{extra: integer;} \quad \{ \text{where we pool our counts} \}
\]

See also section 61.

This code is used in section 51*.
Now we are ready for the routine that writes the preamble of the packed file.

```c
define preamble_comment ≡ "GFtoPK.2.4.output.from";
define comm_length = 0  { length of preamble_comment }
define from_length = 0  { length of its "from" part }

⟨Write preamble⟩ ≡ 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 ≠ "\n";  { 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('\\n');
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('\\n');
  pk_word(design_size); pk_word(check_sum); pk_word(hppp); pk_word(vppp)
```

This code is used in section 51*.

This module is empty in the C version.

Finally, the main program.

```c
begin initialize; convert_gf_file;  { Check for unrasterized locators 85};
print ln(gf_len: 1, "\nbytes\npacked\nto\", pk_loc: 1, "\nbytes.\n"); end.
```
88* System-dependent changes. Parse a Unix-style command line.

\[
\text{define} \quad \text{argument_is(\#)} \equiv (\text{strcmp}(\text{long_options}[\text{option_index}].name, \#) = 0)
\]

\[
\text{define} \quad \text{do_nothing} \equiv \{ \text{empty statement} \}
\]

(Define parse_arguments 88*)

\[
\text{procedure} \quad \text{parse_arguments};
\]

\[
\text{const} \quad \text{n_options} = 3; \quad \{ \text{Pascal won’t count array lengths for us.} \}
\]

\[
\text{var} \quad \text{long_options} : \text{array} \[0..\text{n_options}] \text{of getopt_struct};
\]

\[
\text{getopt_return_val} : \text{integer}; \text{option_index} : \text{c_int_type}; \text{current_option} : 0..\text{n_options};
\]

\[
\text{begin} \quad \{ \text{Initialize the option variables 93*} \};
\]

(Define the option table 89*)

\[
\text{repeat} \quad \text{getopt_return_val} \leftarrow \text{getopt_long_only(\text{argv}, \text{argv}, \text{long_options}, \text{address_of(\text{option_index})})};
\]

\[
\text{if} \quad \text{getopt_return_val} = -1 \text{ then}
\]

\[
\quad \text{begin do_nothing}; \quad \{ \text{End of arguments; we exit the loop below.} \}
\]

\[
\quad \text{end}
\]

\[
\text{else if} \quad \text{getopt_return_val} = “?” \text{ then}
\]

\[
\quad \text{begin usage(\text{my_name});} \quad \{ \text{getopt has already given an error message.} \}
\]

\[
\quad \text{end}
\]

\[
\text{else if} \quad \text{argument_is(\text{“help”}) then}
\]

\[
\quad \text{begin usage_help(\text{GFTOPK_HELP, nil});}
\]

\[
\text{end}
\]

\[
\text{else if} \quad \text{argument_is(\text{“version”}) then}
\]

\[
\quad \text{begin print_version_and_exit(\text{banner, nil, “Tomasz Rokicki”, nil});}
\]

\[
\quad \text{end}; \quad \{ \text{Else it was a flag; getopt has already done the assignment.} \}
\]

\[
\text{until} \quad \text{getopt_return_val} = -1; \quad \{ \text{Now optind is the index of first non-option on the command line. We}
\]

\[
\text{must have one or two remaining arguments.} \}
\]

\[
\text{if} \quad (\text{optind} + 1 \neq \text{argc}) \land (\text{optind} + 2 \neq \text{argc}) \text{ then}
\]

\[
\quad \text{begin write_in(\text{stderr, my_name, “: Needs one or two file arguments.”); usage(my_name);}
\]

\[
\quad \text{end}; \quad \{ \text{If an explicit output filename isn’t given, construct it from gf_name.} \}
\]

\[
\text{if} \quad \text{optind} + 2 = \text{argc} \text{ then}
\]

\[
\quad \text{begin pk_name} \leftarrow \text{cmdline(optind + 1)};
\]

\[
\quad \text{end}
\]

\[
\text{else begin pk_name} \leftarrow \text{basename_change_suffix(gf_name, “gf”, “pk”)};
\]

\[
\quad \text{end}; \quad \text{end}
\]

This code is used in section 4*.

89* Here are the options we allow. The first is one of the standard GNU options.

(Define the option table 89*)

\[
\text{current_option} \leftarrow 0; \quad \text{long_options}[\text{current_option}].name \leftarrow \text{“help”};
\]

\[
\text{long_options}[\text{current_option}].has_arg \leftarrow 0; \quad \text{long_options}[\text{current_option}].flag \leftarrow 0;
\]

\[
\text{long_options}[\text{current_option}].val \leftarrow 0; \quad \text{incr(\text{current_option})};
\]

See also sections 90*, 91*, and 94*.

This code is used in section 88*.

90* Another of the standard options.

(Define the option table 89*)

\[
\text{long_options}[\text{current_option}].name \leftarrow \text{“version”}; \quad \text{long_options}[\text{current_option}].has_arg \leftarrow 0;
\]

\[
\text{long_options}[\text{current_option}].flag \leftarrow 0; \quad \text{long_options}[\text{current_option}].val \leftarrow 0; \quad \text{incr(\text{current_option})};
\]
Print progress information?

\[\text{Define the option table}\]

\[
\begin{align*}
\text{long_options[\text{current_option}].name} & \leftarrow \text{verbose}; \\
\text{long_options[\text{current_option}].has_arg} & \leftarrow 0; \\
\text{long_options[\text{current_option}].flag} & \leftarrow \text{address_of(\text{verbose});} \\
\text{long_options[\text{current_option}].val} & \leftarrow 1; \\
\text{incr(\text{current_option});}
\end{align*}
\]

\[\text{(Globals in the outer block)}\]

\[\begin{align*}
\text{verbose: c_int_type;}
\end{align*}\]

\[\text{(Initialize the option variables)}\]

\[
\begin{align*}
\text{verbose} & \leftarrow \text{false}; \\
\text{This code is used in section 88*}.
\end{align*}
\]

An element with all zeros always ends the list.

\[\text{(Define the option table)}\]

\[
\begin{align*}
\text{long_options[\text{current_option}].name} & \leftarrow 0; \\
\text{long_options[\text{current_option}].has_arg} & \leftarrow 0; \\
\text{long_options[\text{current_option}].flag} & \leftarrow 0; \\
\text{long_options[\text{current_option}].val} & \leftarrow 0;
\end{align*}
\]

Global filenames.

\[\text{(Globals in the outer block)}\]

\[
\begin{align*}
\text{gf\_name, pk\_name: const c_string;}
\end{align*}
\]
96*  Index.  Pointers to error messages appear here together with the section numbers where each identifier is used.

The following sections were changed by the change file: 1, 4, 5, 6, 8, 10, 39, 40, 44, 46, 48, 49, 51, 52, 56, 57, 58, 81, 83, 86, 88, 89, 90, 91, 92, 93, 94, 95, 96.

- help: 89*
  - version: 90*
  a: 43.
  abort: 8* 57*.
  address_of: 88* 91*.
  all 223's: 60.
  argc: 88*.
  argument is: 88*.
  argv: 4* 88*.
  ASCII_code: 9, 11.
  b: 43, 44*.
  b_comp_size: 68, 70.
  backpointers: 19.
  bad: 56*.
  Bad GF file: 8*.
  bad_gf: 8* 43, 51* 54, 56* 57* 60.
  banner: 1* 4* 88*.
  basename_change_suffix: 88*.
  bit_weight: 44* 45, 75.
  black: 15, 16.
  loc: 14, 16, 17, 18, 19, 51* 54.
  loc1: 16, 17, 51*.
  boolean: 41, 58* 70, 77.
  buff: 64, 65, 67, 76, 80.
  byte is not post: 60.
  byte_file: 37, 38.
  c: 43.
  c_int_type: 88* 92*.
  cc: 32.
  char: 10* 82.
  char_loc: 16, 17, 19, 60.
  char_loc0: 16, 17, 60.
  check sum: 18.
  check_sum: 60, 81* 87.
  Chinese characters: 19.
  chr: 10* 11, 13.
  cmdline: 88*.
  comm_length: 81* 82.
  comment: 81* 82.
  comp_size: 68, 69, 71, 72, 73, 74, 77.
  const_c_string: 95*.
  convert_gf_file: 51* 55, 86*.
  count: 76, 77, 80.
  cs: 18, 23.
  cur_pos: 46*.
  current_option: 88* 89* 90* 91* 94*.
  d: 43.
  d_print_in: 2, 54, 63, 68.
  debugging: 2.
  decr: 7, 30, 60, 63, 69, 76, 81*.
  del_m: 16.
  del_n: 16.
  deriv: 68, 69, 70.
  design size: 18.
  design_size: 60, 81* 87.
  dm: 16, 32.
  do nothing: 88*.
  do_the_rows: 51* 57* 58*.
  ds: 18, 23.
  dx: 16, 19, 32, 48* 60, 71, 72, 73, 74.
  dy: 16, 19, 32, 48* 60, 71, 72.
  eight_bits: 37, 43.
  else: 3.
  end: 3.
  end_of_char: 48* 50, 57* 63, 64, 66, 68, 75, 76.
  end_of_row: 48* 50, 57* 63, 64, 66, 67.
  endcases: 3.
  eoc: 14, 16, 17, 18, 57*.
  eof: 43, 46*.
  extra: 57* 58* 59, 63, 65, 66, 67.
  false: 42, 51* 57* 59, 67, 76, 93*.
  find_gf_length: 46* 60.
  First byte is not preamble: 51*.
  first_on: 68, 70, 71.
  first_text_char: 10* 13.
  flag: 32, 89* 90* 91* 94*.
  flag_byte: 70, 71, 72, 73, 74.
  four_cases: 52* 53, 57*.
  from_length: 81*.
  Fuchs, David Raymond: 20.
  get_nyb: 30.
  getopt: 88*.
  getopt_long_only: 88*.
  getopt_return_val: 88*.
  getopt_struct: 88*.
  gf_byte: 43, 51* 53, 54, 57* 59, 60, 81*.
  gf_ch: 54, 55, 60, 71, 72, 73, 74.
  gf_ch_mod_256: 54, 55, 57* 71, 72, 73, 74.
  gf_com: 51* 53, 54, 57* 59, 60.
  gf_file: 4* 38, 39* 41, 42, 43, 46* 47, 48*.
  gf_id_byte: 16, 51* 60.
  gf_len: 46* 47, 60, 86*.
  gf_length: 46*.
  gf_loc: 39* 41, 43.
gf_name: 39, 46, 88, 95
gf_signed_quad: 43, 53, 54, 60.
GPtoPK: 4
GFTOPK_HELP: 88
h_bit: 65, 67, 76, 80.
h_mag: 60, 87.
has_arg: 89, 90, 91, 94
height: 31, 63, 68, 70, 71, 72, 73, 74.
hoff: 32, 34.
hcpp: 18, 23, 60, 61, 81
i: 4, 30, 51, 62, 87.
ID byte is wrong: 60.
Identification byte incorrect: 51
incr: 7, 30, 43, 44, 56, 63, 64, 66, 67, 68, 69, 75, 80, 81, 90, 91
initialize: 4, 86
integer: 4, 30, 41, 43, 44, 45, 46, 47, 48, 49, 51, 55, 58, 61, 62, 65, 70, 77, 78, 87, 88
Internal error: 57
j: 30, 51, 62.
Knuth, Donald Ervin: 29.
kpse_gf_format: 39
kpse_init_prog: 4
kpse_open_file: 39
kpse_set_program_name: 4
last_text_char: 10, 13.
line_length: 6
located: 48, 60, 85.
Locator...already found: 60.
long_options: 88, 89, 90, 91, 94
max_m: 16, 18, 48, 54, 55, 63.
max_n: 16, 18, 48, 54, 55, 63.
max_new_row: 17.
max_row: 6, 48, 49, 51, 56
MAX_ROW: 6, 49, 51
max2: 75, 77.
min_m: 16, 18, 48, 54, 55, 63.
min_n: 16, 18, 48, 54, 55, 63.
missing raster information: 85.
mvew_to_byte: 46, 51, 60.
my_name: 1, 4, 88
n_options: 88
name: 88, 89, 90, 91, 94
new_row: 51, 56
new_row_0: 16, 17, 52, 57
new_row_1: 16.
new_row_128: 52, 57
new_row_164: 16.
new_row_64: 52, 57
no character locator...: 54.
no_op: 16, 17, 19, 53.
Odd aspect ratio: 60.
on: 57, 58, 59, 70, 76, 80.
only n bytes long: 60.
open_gf_file: 39, 51
open_pk_file: 40, 51
optind: 88
option_index: 88
ord: 11.
ooriental characters: 19.
othercases: 3.
output: 4
output_byte: 44, 45, 75.
p_bit: 76, 77, 80.
pack_and_send_character: 55, 57, 62, 65.
paint: 56, 57
paint_switch: 15, 16, 57
paint_0: 16, 17, 57, 59.
paint1: 16, 17, 59.
paint2: 16.
paint3: 16.
parse_arguments: 4, 88
pk_byte: 44, 53, 72, 73, 74, 75, 76, 80, 81, 84
pk_file: 4, 38, 40, 41, 42, 44, 48, 60
pk_halfword: 44, 74
pk_id: 24, 81
pk_loc: 40, 41, 44, 57, 72, 73, 74, 84, 86
pk_name: 40, 88, 95
pk_no_op: 23, 24, 84
pk_nyb: 44, 75
pk_open: 40, 41, 42
pk_packed_num: 30
pk_post: 23, 24, 84
pk_pre: 23, 24, 81
pk_three_bytes: 44, 73, 74
pk_word: 44, 53, 72, 81
pk_xxx1: 23, 24, 53
pl: 32.
post: 14, 16, 17, 18, 20, 51, 60.
post location is: 60.
post pointer is wrong: 60.
post_loc: 60, 61.
post_post: 16, 17, 18, 20, 60.
power: 78, 79, 80.
pre: 14, 16, 17, 51
preamble_comment: 1, 81
pred_pk_loc: 55, 57, 72, 73, 74
print: 4, 81
print_len: 2, 3, 51, 60, 81, 85, 86
print_version_and_exit: 88

(Calculate \textit{dyn}_f and packed size and write character 68) Used in section 62.
(Cases for \textit{paint} commands 59) Used in section 57*.
(Check for unrasterized locators 85) Used in section 86*.
(Constants in the outer block 6*) Used in section 4*.
(Convert character to packed form 57*) Used in section 54.
(Convert row-list to glyph-list 64) Used in section 62.
(Define the option table 89*, 90*, 91*, 94*) Used in section 88*.
(Define \textit{parse_arguments} 88*) Used in section 4*.
(Find and interpret postamble 60) Used in section 51*.
(Globals in the outer block 11, 38, 41, 45, 47, 48*, 55, 78, 82, 87, 92*, 95*) Used in section 4*.
(Initialize the option variables 93*) Used in section 88*.
(Interpret character 54) Used in section 88*.
(Locals to \textit{convert.gf.file} 58*, 61) Used in section 51*.
(Locals to \textit{pack_and_send_character} 65, 70, 77) Used in section 62.
(Packing procedures 62) Used in section 51*.
(Process count for best \textit{dyn}_f value 69) Used in section 68.
(Reformat count list 67) Used in section 64.
(Scan for bounding box 63) Used in section 62.
(Send bit map 76) Used in section 68.
(Send compressed format 75) Used in section 68.
(Send one row by bits 80) Used in section 76.
(Set initial values 12, 13, 42, 49*, 79) Used in section 4*.
(Skip over repeated rows 66) Used in section 64.
(Specials and \textit{no_op} cases 53) Used in sections 51*, 57*, and 60.
(Write character preamble 71) Used in section 68.
(Write long character preamble 72) Used in section 71.
(Write one-byte short character preamble 73) Used in section 71.
(Write postamble 84) Used in section 51*.
(Write preamble 81*) Used in section 51*.
(Write two-byte short character preamble 74) Used in section 71.