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>cmr10</td>
<td>746</td>
<td>29464</td>
<td>13912</td>
<td>47%</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.

```plaintext
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.

```plaintext
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*⟩
type ⟨Types in the outer block 9⟩
var ⟨Globals in the outer block 11⟩
  ⟨Define parse_arguments 88*⟩
procedure initialize;  { this procedure gets things started properly }
  var i: integer;  { loop index for initializations }
  begin kpse_set_program_name(argv[0], my.name); kpse_initProg('GFTOPK', 0, nil, nil);
    parse_arguments; print(banner); print.Ln(version_string);  ⟨Set initial values 12⟩
  end;

5* This module is deleted, because it is only useful for a non-local goto, which we can’t use in C.

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

```plaintext
⟨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.

```plaintext
define abort(#) ≡
    begin write_ln(stderr, #); uexit(1);
    end

define bad_gf(#) ≡ abort(´Bad GF file:` , #, ´!´)
```
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 \texttt{char} for the data type associated with the characters in text files, while other Pascals consider \texttt{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 \texttt{text\_char} to stand for the data type of the characters in the output file. We shall also assume that \texttt{text\_char} consists of the elements \texttt{chr(first\_text\_char)} through \texttt{chr(last\_text\_char)}, inclusive. The following definitions should be adjusted if necessary.

\begin{verbatim}
define char \equiv 0..255 { the data type of characters in text files }
define text_char \equiv char { ordinal number of the smallest element of text_char }
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 }
\end{verbatim}

\texttt{text\_file = packed file of text\_char;
§14  GF to PK changes C

39* In C, we do path searching based on the user’s environment or the default paths.

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;

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

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;

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

Output is handled through putbyte which is supplied by web2c.

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

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

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;

procedure pk_word(a: integer);
  var b: integer;
  begin if a < 0 then
      begin a ← a + ¹10000000000; a ← a + ¹10000000000; b ← 128 + a div 16777216;
      end
  else 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;

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;
Finally we come to the routines that are used for random access of the \textit{gf\_file}. To correctly find and read the postamble of the file, we need two routines, one to find the length of the \textit{gf\_file}, and one to position the \textit{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 \textit{set\_pos} and \textit{cur\_pos}. The call \textit{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, \textit{set\_pos}(f, n) moves to the end of file \(f\). The call \textit{cur\_pos}(f) gives the total number of items in \(f\), if \textit{eof}(f) is true; we use \textit{cur\_pos} only in such a situation.

\begin{verbatim}
define find\_gf\_length \equiv gf\_len \leftarrow gf\_length

function gf\_length: integer;
    begin xfseek(gf\_file, 0, 2, gf\_name); gf\_length \leftarrow xftell(gf\_file, gf\_name);
    end;

procedure move\_to\_byte(n: integer);
    begin xfseek(gf\_file, n, 0, gf\_name);
    end;
\end{verbatim}
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 restructure the row list from a list based on rows to a list based on the entire glyph. Then, an optimal value of dyn_f is calculated, and the final size of the counts is found for the PK file format, and compared with the bit-wise packed glyph. If the run-encoding scheme is shorter, the character is written to the pk_file as row counts; otherwise, it is written using a bit-packed scheme.

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

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

(Globals in the outer block 11) +≡

```plaintext
tfm_width: array [0 .. 255] of integer;  { the TFM widths of characters }
dx, dy: array [0 .. 255] of integer;  { the horizontal and vertical escapements }
status: array [0 .. 255] of virgin .. sent;  { character status }
row: ↑integer;  { the row counts for working }
max_row: integer;  { largest index in the main row array }
```

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

(Set initial values 12) +≡

```plaintext
row ← xmalloc_array(integer, MAX_ROW); max_row ← MAX_ROW;
for i ← 0 to 255 do status[i] ← virgin;
```
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);

```plaintext
procedure row_overflow;
    var new_row: integer;

    begin new_row ← max_row + MAX_ROW;
            print Ln("Reallocated_row_array to", new_row : 1, "items from", max_row : 1, ");
    row ← xrealloc(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 is incorrect");
        ⟨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, bocl: ⟨Interpret character 54⟩;
                ⟨Specials and no_op cases 53⟩;
                post: ; ⟨we will actually do the work for this one later⟩;
                othercases bad_gf("Unexpected", gf_com : 1, "command between characters")
            endcases;
        until gf_com = post;
        ⟨Write postamble 84⟩;
    end;
```

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

```plaintext
define four_cases(#) ≡ #, # + 1, # + 2, # + 3
define sixteen_cases(#) ≡ four_cases(#), four_cases(# + 4), four_cases(# + 8), four_cases(# + 12)
define sixty_four_cases(#) ≡ sixteen_cases(#), sixteen_cases(# + 16), sixteen_cases(# + 32),
    sixteen_cases(# + 48)
define 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
```

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.

```plaintext
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 recognize the row as empty.

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.

\[\text{Convert character to packed form } 57^* \equiv \]
\[
\text{begin row_ptr } \leftarrow 2; \text{ on } \leftarrow \text{false}; \text{ extra } \leftarrow 0; \text{ state } \leftarrow \text{true}; \\
\text{repeat } \text{gf_com } \leftarrow \text{gf_byte} ; \\
\text{case } \text{gf_com of} \\
\quad \text{\{Cases for paint commands 59\}}; \\
\quad \text{four_cases}(\text{skip0}) : \text{begin } i \leftarrow 0; \\
\quad \quad \text{for } j \leftarrow 1 \text{ to } \text{gf_com } \text{−} \text{skip0} \text{ do } i \leftarrow i \times 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(end_of_row)}; \\
\quad \quad \text{on } \leftarrow \text{false}; \text{ extra } \leftarrow 0; \text{ state } \leftarrow \text{true}; \\
\quad \text{end}; \\
\quad \text{sixty_four_cases(new_row_0)} ; \text{ do_the_rows } \leftarrow \text{true}; \\
\quad \text{sixty_four_cases(new_row_64)} ; \text{ do_the_rows } \leftarrow \text{true}; \\
\quad \text{thirty_seven_cases(new_row_128)} ; \text{ do_the_rows } \leftarrow \text{true}; \\
\quad \text{\{Specials and no_op cases 53\}}; \\
\quad \text{eoc} : \text{begin if } \text{on } = \text{state} \text{ then } \text{put_in_rows(extra)}; \\
\quad \quad \text{if } (\text{row_ptr } > 2) \land (\text{row}[\text{row_ptr } - 1] \neq \text{end_of_row}) \text{ then } \text{put_in_rows(end_of_row)}; \\
\quad \quad \text{put_in_rows(end_of_char)} ; \text{ pack_and_send_character}; \text{ status[\text{gf_ch_mod_256}] } \leftarrow \text{sent}; \\
\quad \quad \text{if } \text{pk_loc } \neq \text{pred_pk_loc} \text{ then abort(’Internal error while writing character!’);} \\
\quad \text{end}; \\
\quad \text{othercases bad_gf(’Unexpected’, } \text{gf_com : 1, ’command in character definition’)} \\
\text{endcases}; \\
\quad \text{if } \text{do_the_rows } \text{ then} \\
\quad \quad \text{begin do_the_rows } \leftarrow \text{false}; \\
\quad \quad \text{if } \text{on } = \text{state} \text{ then } \text{put_in_rows(extra)}; \\
\quad \quad \text{put_in_rows(end_of_row)}; \text{ on } \leftarrow \text{true}; \text{ extra } \leftarrow \text{gf_com } \text{−} \text{new_row_0}; \text{ state } \leftarrow \text{false}; \\
\quad \text{end}; \\
\quad \text{until } \text{gf_com } = \text{eoc}; \\
\text{end} \]

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

\[
\langle \text{Locals to convert\_gf\_file 58}\rangle \equiv
\]

\[
do\_the\_rows: \text{boolean};
\]

\[
on: \text{boolean}; \quad \{\text{indicates whether we are white or black}\}\]

\[
state: \text{boolean}; \quad \{\text{a state variable—is the next count the same race as the one in the extra buffer?}\}\]

\[
extra: \text{integer}; \quad \{\text{where we pool our counts}\}\]

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\ 81\textsuperscript{*}\} ≡
\pkbyte{pk_pre}; \pkbyte{pk_id}; i \leftarrow gf_byte; \{ get length of introductory comment \}
\repeat if i = 0 then j \leftarrow "." else j \leftarrow gf_byte;
\until j \neq "."; \{ remove leading blanks \}
in(i); \{ this many bytes to copy \}
if i = 0 then k \leftarrow \text{comm_length} - \text{from_length}
else k \leftarrow i + \text{comm_length};
if k > 255 then \pkbyte{255} else \pkbyte{k};
for k \leftarrow 1 \to \text{comm_length} do
  if (i > 0) \lor (k \leq \text{comm_length} - \text{from_length}) then \pkbyte{xord[comment[k]]};
  print(\`");
  for k \leftarrow 1 \to i do
    begin if k > 1 then j \leftarrow gf_byte;
      print(xchr[j]);
    if k < 256 - \text{comm_length} then \pkbyte{j};
    print\ln(\`");
    \pkword{design_size}; \pkword{check_sum}; \pkword{hppp}; \pkword{vppp}
\}
This code is used in section 51\textsuperscript{*}.

This module is empty in the C version.

Finally, the main program.

\begin initialize; convert_gf_file; \{ Check for unrasterized locators 85 \};
print\ln(gf_len : 1, \\bytes_{\text{packed to}}\”, pk_loc : 1, \\bytes.”);\end
```
88* System-dependent changes. Parse a Unix-style command line.

```pascal
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
define do_nothing ≡  { empty statement }
(Define parse_arguments 88*) ≡
procedure parse_arguments;
  const n_options = 3;  { Pascal won’t count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
  getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
begin { Initialize the option variables 93*};
  (Define the option table 88*);
  repeat getopt_return_val ≡ getopt_long_only(argc, argv, ‘’, long_options, address_of(option_index));
    if getopt_return_val = -1 then
      begin do_nothing;  { End of arguments; we exit the loop below. }
      end
    else if getopt_return_val = "?” then
      begin usage(my_name);  { getopt has already given an error message. }
      end
    else if argument_is(‘help’) then
      begin usage_help(GFTOPK_HELP, nil);
      end
    else if argument_is(‘version’) then
      begin print_version_and_exit(banner, nil, ‘Tomas_Rokicki’, nil);
      end
  until getopt_return_val = -1;  { Now optind is the index of first non-option on the command line. We
  must have one or two remaining arguments. }
if (optind + 1 ≠ argc) ∧ (optind + 2 ≠ argc) then
  begin writeLn(stderr, my_name, ‘: Need one or two file arguments.’); usage(my_name);
  end;
gf_name ← cmdline(optind);  { If an explicit output filename isn’t given, construct it from gf_name. }
if optind + 2 = argc then
  begin pk_name ← cmdline(optind + 1);
  end
else begin pk_name ← basename_change_suffix(gf_name, ‘gf’, ‘pk’);
  end;
end;
```

This code is used in section 4*.

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

```pascal
(Define the option table 89*) ≡
current_option ← 0; long_options[current_option].name ← ‘help’;
long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
long_options[current_option].val ← 0; incr(current_option);
```

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

This code is used in section 88*.

90* Another of the standard options.

```pascal
(Define the option table 90*) ≡
long_options[current_option].name ← ‘version’; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```
Print progress information?

\[
\begin{align*}
\text{Define the option table } & 89^* \equiv \\
\text{long_options}[\text{current_option}].\text{name} & \leftarrow \text{"verbose"}; \\
\text{long_options}[\text{current_option}].\text{has_arg} & \leftarrow 0; \\
\text{long_options}[\text{current_option}].\text{flag} & \leftarrow \text{address_of}(\text{verbose}); \\
\text{long_options}[\text{current_option}].\text{val} & \leftarrow 1; \\
\text{incr(} & \text{current_option});
\end{align*}
\]

( Globals in the outer block 11 ) +≡

\[
\text{verbose: c_int_type;}
\]

( Initialize the option variables 93* ) ≡

\[
\text{verbose} \leftarrow \text{false};
\]

This code is used in section 88*.

An element with all zeros always ends the list.

\[
\begin{align*}
\text{Define the option table } & 89^* \equiv \\
\text{long_options}[\text{current_option}].\text{name} & \leftarrow 0; \\
\text{long_options}[\text{current_option}].\text{has_arg} & \leftarrow 0; \\
\text{long_options}[\text{current_option}].\text{flag} & \leftarrow 0; \\
\text{long_options}[\text{current_option}].\text{val} & \leftarrow 0;
\end{align*}
\]

Global filenames.

\[
\begin{align*}
\text{Globals in the outer block } & 11 \equiv \\
\text{gf_name, pk_name: const c_string;}
\end{align*}
\]
96* Index. Points 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_qf: 8* 43, 51* 54, 56* 57* 60.
banner: 1* 4* 88*.
basename_change_suffix: 88*.
bias: 11, 12, 13, 14, 15, 16.
bias: 14, 16, 17, 18, 19, 31, 54.
bias: 16, 17, 51*.
boolean: 41, 58* 70, 77.
buf: 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_printLn: 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.
ec: 14, 16, 17, 18, 57*.
ecf: 43, 46*.
false: 42, 51*, 57*, 59, 67, 76, 93*.
find_gf_length: 46*, 60.
First byte is not preambles: 51*.
first_on: 68, 70, 71.
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.
GFltPK: 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.
hppp: 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*, 89*, 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.
Japanese characters: 19.
k: 51* 62.
Knuth, Donald Ervin: 29.
kpse_gf_format: 39*.
kpse_init_prog: 4*.
kpse_open_file: 39*.
kpse_set_program_name: 4*.
lstext_string: 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* 49*, 51*.
max_row: 6*, 48*, 49*, 51*, 56*.
max2: 75, 77.
min_m: 16, 18, 48*, 54, 55, 63.
min_n: 16, 18, 48*, 54, 55, 63.
missing raster information: 85.
move_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.
optical characters: 19.
othercases: 3.
others: 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_xxx: 23, 24, 53.
pk_yyy: 23, 24, 53.
plot: 32.
post: 14, 16, 17, 18, 20, 51*, 60.
pot location is: 60.
pot pointer is wrong: 60.
pot_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*.
printloon: 2, 4*, 51*, 60, 81*, 85, 86*.
print_version_and_exit: 88*.
usage: 19.
put_count: 64, 67.
put_in_rows: \textit{56*, 57*, 59}.
put_ptr: 64, 65.
putbyte: 44*.
q: 61.
r_count: 76, 77.
r_i: 76, 77.
r_on: 76, 77.
read: 43.
repeat_count: 30.
repeat_flag: 64, 65, 66, 76, 80.
rewrite: 40*.
rewritebin: 40*.
Rokicki, Tomas Gerhard Paul: 1*.
rond: 60.
row_overflow: \textit{51*, 56*}.
row_ptr: \textit{55, 56*, 57*, 63, 64, 66, 67}.
s_count: 76, 77.
s_i: 76, 77.
s_on: 76, 77.
Samuel, Arthur Lee: 1*.
scaled: 16, 18, 19, 23.
sent: \textit{48*, 57*}.
s_pos: 46*.
sixteen_cases: 52*.
sixty_four_cases: \textit{52*, 57*, 59}.
skip: 56*.
skip_0: 57*.
skip0: 16, 17, 57*.
skip1: 16, 17*.
skip2: 16.
skip3: 16.
state: \textit{57*, 58*, 59, 64, 67, 70, 76, 80}.
status: \textit{48*, 49*, 54, 57*, 60, 85}.
stderr: \textit{8*, 88*}.
stdout: \textit{4*}.
strcmp: stdout.
strcmp: stderr.
true: \textit{40*, 57*, 64, 67, 76}.
true: \textit{40*, 57*, 64, 67, 76}.
uexit: 8*.
undefined_commands: 17.
Unexpected command: 51*, 57*, 60.
Unexpected end of file: 43.
usage: 88*.
usage_help: 88*.
val: \textit{89*, 90*, 91*, 94*}.
verbose: \textit{4*, 91*, 92*, 93*}.
version_string: \textit{4*}.
virgin: \textit{48*, 49*, 54, 60}.
voif: 32, 34.
ppp: 18, 23, 60, 61, 81*.
white: 16.
width: 31, 63, 66, 67, 68, 70, 71, 72, 73, 74, 76, 80.
write: 4*.
write_in: \textit{4*, 8*, 88*}.
x_offset: 63, 70, 71, 72, 73, 74.
xchr: \textit{11, 12, 13, 81*}.
xfseek: 46*.
xf.tell: 46*.
xmalloc_array: 49*.
xord: \textit{11, 13, 81*}.
xrealloc_array: 51*.
xxx1: 16, 17, 53.
xxx2: 16.
xxx3: 16.
xxx4: 16.
y_offset: 63, 70, 71, 72, 73, 74.
yyy: 16, 17, 19, 23, 53.
(Calculate $\text{dyn}_f$ and packed size and write character 68) Used in section 62.
(Cases for paint commands 59) Used in section 57*.
(Define the option table 89*, 90*, 91*, 94*) Used in section 88*.
(Define parse_arguments 88*) Used in section 4*.
(Find and interpret postamble 60) Used in section 88*.
(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 51*.
(Locals to convert_gf_file 58*, 61) Used in section 51*.
(Locals to pack_and_send_character 65, 70, 77) Used in section 62.
(Packing procedures 62) Used in section 51*.
(Process count for best $\text{dyn}_f$ value 69) Used in section 68.
(Reformat count list 67) Used in section 64.
(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.