The GFtoPK processor

(Version 2.4, 06 January 2014)

Editor's Note: The present variant of this C/WEB source file has been modified for use in the TEX Live system.

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.

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926, MCS-8300984, and CCR-8610181, and by the System Development Foundation.

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

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

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

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

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

define *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.

define *print*(#) ≡
  if *verbose* **then** *write*(*stdout*, #)
define *print_ln*(#) ≡
  if *verbose* **then** *write_ln*(*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_init_prog*(´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.

⟨ 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*.

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

Such errors might be discovered inside of subroutines inside of subroutines, so we might want to *abort* the program with an error message.

define *abort*(#) ≡
  begin *write_ln*(*stderr*, #); *uexit*(1);
  end
define *bad_gf*(#) ≡ *abort*(´Bad␣GF␣file:␣´, #, ´!´)

10* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like `GFtoPK`. So we shall assume that the Pascal system being used for `GFtoPK` has a character set containing at least the standard visible characters of ASCII code (`"!"` through `"~"`).

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name. In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the output file. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *char* $\equiv 0\,..\,255$
define *text_char* $\equiv$ *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*;

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** $a \leftarrow a + 65536$;
  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* + ´*10000000000*; *a* ← *a* + ´*10000000000*; *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;

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

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

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;

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

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

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

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

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

⟨ Globals in the outer block 11 ⟩ +≡
tfm_width: **array** [0 .. 255] **of** *integer*; { the TFM widths of characters }
dx, *dy*: **array** [0 .. 255] **of** *integer*; { the horizontal and vertical escapements }
status: **array** [0 .. 255] **of** *virgin* .. *sent*; { character status }
row: ↑*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 ⟩ +≡
row ← *xmalloc_array*(*integer*, *MAX_ROW*); *max_row* ← *MAX_ROW*;
for $i \leftarrow 0$ **to** 255 **do** *status*[i] ← *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_ln('Reallocated␣row␣array␣to␣', new_row : 1, '␣items␣from␣', 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␣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, boc1: ⟨Interpret character 54⟩;
        ⟨Specials and no_op cases 53⟩;
      post: ; {we will actually do the work for this one later}
      othercases bad_gf('Unexpected␣', gf_com : 1, '␣command␣between␣characters')
      endcases;
    until gf_com = post;
    ⟨Write postamble 84⟩;
    end;
```

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

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

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

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

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

⟨Convert character to packed form 57*⟩ ≡
  begin *row_ptr* ← 2; *on* ← *false*; *extra* ← 0; *state* ← *true*;
  repeat *gf_com* ← *gf_byte*;
    case *gf_com* **of**
    ⟨Cases for *paint* commands 59⟩;
    four_cases(*skip0*): **begin** *i* ← 0;
      for *j* ← 1 **to** *gf_com* − *skip0* **do** *i* ← *i* ∗ 256 + *gf_byte*;
      if *on* = *state* **then** *put_in_rows*(*extra*);
      for *j* ← 0 **to** *i* **do** *put_in_rows*(*end_of_row*);
      on ← *false*; *extra* ← 0; *state* ← *true*;
      end;
    sixty_four_cases(*new_row_0*): *do_the_rows* ← *true*;
    sixty_four_cases(*new_row_64*): *do_the_rows* ← *true*;
    thirty_seven_cases(*new_row_128*): *do_the_rows* ← *true*;
    ⟨Specials and *no_op* cases 53⟩;
    eoc: **begin if** *on* = *state* **then** *put_in_rows*(*extra*);
      if (*row_ptr* > 2) ∧ (*row*[*row_ptr* − 1] ≠ *end_of_row*) **then** *put_in_rows*(*end_of_row*);
      put_in_rows(*end_of_char*); *pack_and_send_character*; *status*[*gf_ch_mod_256*] ← *sent*;
      if *pk_loc* ≠ *pred_pk_loc* **then** *abort*(`´Internal␣error␣while␣writing␣character!´`);
      end;
    othercases *bad_gf*(`´Unexpected␣´`, *gf_com* : 1, `´␣command␣in␣character␣definition´`)
    endcases;
    if *do_the_rows* **then**
      begin *do_the_rows* ← *false*;
      if *on* = *state* **then** *put_in_rows*(*extra*);
      put_in_rows(*end_of_row*); *on* ← *true*; *extra* ← *gf_com* − *new_row_0*; *state* ← *false*;
      end;
  until *gf_com* = *eoc*;
  end

This code is used in section 54.

58* A few more locals used above and below:

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

See also section 61.

This code is used in section 51*.

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

define *preamble_comment* ≡ `'GFtoPK␣2.4␣output␣from␣'`
define *comm_length* = 0 { length of *preamble_comment* }
define *from_length* = 0 { length of its `'␣from␣'` part }

⟨ Write preamble 81* ⟩ ≡
pk_byte(*pk_pre*); *pk_byte*(*pk_id*); $i \leftarrow$ *gf_byte*; { get length of introductory comment }
repeat if $i = 0$ **then** $j \leftarrow$ `"."` **else** $j \leftarrow$ *gf_byte*;
  decr(i); { some people think it's wise to avoid **goto** statements }
until $j \neq$ `"␣"`; { remove leading blanks }
incr(i); { this many bytes to copy }
if $i = 0$ **then** $k \leftarrow comm\_length - from\_length$
else $k \leftarrow i + comm\_length$;
if $k > 255$ **then** *pk_byte*(255) **else** *pk_byte*(k);
for $k \leftarrow 1$ **to** *comm_length* **do**
  if $(i > 0) \lor (k \le comm\_length - from\_length)$ **then** *pk_byte*(*xord*[*comment*[k]]);
print(`''''`);
for $k \leftarrow 1$ **to** i **do**
  begin if $k > 1$ **then** $j \leftarrow$ *gf_byte*;
  print(*xchr*[j]);
  if $k < 256 - comm\_length$ **then** *pk_byte*(j);
  end;
print_ln(`''''`);
pk_word(*design_size*); *pk_word*(*check_sum*); *pk_word*(*hppp*); *pk_word*(*vppp*)

This code is used in section 51*.

83* This module is empty in the C version.

86* Finally, the main program.

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

88*. System-dependent changes. Parse a Unix-style command line.

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 89* ⟩;
  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; { Else it was a flag; *getopt* has already done the assignment. }
  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 *write_ln*(*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.

⟨ 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.

⟨ Define the option table 89* ⟩ +≡
  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*);

91* Print progress information?

⟨ Define the option table 89* ⟩ +≡
$long\_options[current\_option].name \leftarrow$ `'verbose'`; $long\_options[current\_option].has\_arg \leftarrow 0$;
$long\_options[current\_option].flag \leftarrow address\_of(verbose)$; $long\_options[current\_option].val \leftarrow 1$;
$incr(current\_option)$;

92* ⟨ Globals in the outer block 11 ⟩ +≡
$verbose$: c_int_type;

93* ⟨ Initialize the option variables 93* ⟩ ≡
$verbose \leftarrow false$;

This code is used in section 88*.

94* An element with all zeros always ends the list.

⟨ Define the option table 89* ⟩ +≡
$long\_options[current\_option].name \leftarrow 0$; $long\_options[current\_option].has\_arg \leftarrow 0$;
$long\_options[current\_option].flag \leftarrow 0$; $long\_options[current\_option].val \leftarrow 0$;

95* Global filenames.

⟨ Globals in the outer block 11 ⟩ +≡
gf_name, pk_name: $const\_c\_string$;

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.