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 *banner* ≡ `'This␣is␣GFtoPK,␣C␣Version␣2.3'` { 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 term_out ≡ stdout  { standard output }
define print(#) ≡
        if verbose then write(term_out,#)
define print_ln(#) ≡
        if verbose then write_ln(term_out,#)
program GFtoPK;
  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 set_paths(GF_FILE_PATH_BIT);
    ⟨ 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.

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 a procedure called *jump_out* has been introduced. This procedure, which simply transfers control to the label *final_end* at the end of the program, contains the only non-local **goto** statement in GFtoPK.

```
define abort(#) ≡
        begin verbose ← true; print_ln(#); uexit(1);
        end
define bad_gf(#) ≡ abort('Bad␣GF␣file:␣',#,'!')
```

37* Input and output for binary files. We have seen that a `GF` file is a sequence of 8-bit bytes. The bytes appear physically in what is called a '**packed file of** 0 .. 255' in Pascal lingo. The `PK` file is also a sequence of 8-bit bytes.

Packing is system dependent, and many Pascal systems fail to implement such files in a sensible way (at least, from the viewpoint of producing good production software). For example, some systems treat all byte-oriented files as text, looking for end-of-line marks and such things. Therefore some system-dependent code is often needed to deal with binary files, even though most of the program in this section of `GFtoPK` is written in standard Pascal.

We shall stick to simple Pascal in this program, for reasons of clarity, even if such simplicity is sometimes unrealistic.

⟨ Types in the outer block 9 ⟩ +≡
```
eight_bits = 0 .. 255;  { unsigned one-byte quantity }
byte_file = packed file of eight_bits;  { files that contain binary data }
UNIX_file_name = packed array [1 .. FILENAMESIZE] of char;
```

38* The program deals with two binary file variables: *gf_file* is the input file that we are translating into `PK` format, to be written on *pk_file*.

⟨ Globals in the outer block 11 ⟩ +≡
```
gf_file: byte_file;  { the stuff we are GFtoPKing }
pk_file: byte_file;  { the stuff we have GFtoPKed }
verbose: boolean;  { chatter about the conversion? }
pk_arg: integer;  { where we may be looking for the name of the pk_file }
gf_name: UNIX_file_name;
```

39* In C, we use the external *test_read_access* procedure, which also does path searching based on the user's environment or the default path. In the course of this routine we also check the command line for the `-v` flag, and make other checks to see that it is worth running this program at all.

```
  define usage ≡ abort('Usage: gftopk [-v] <gf file> [pk file].')
procedure open_gf_file; { prepares to read packed bytes in gf_file }
  var j: integer;
  begin verbose ← false; pk_arg ← 3;
  if (argc < 2) ∨ (argc > 4) then usage;
  argv(1, gf_name);
  if gf_name[1] = xchr["-"] then
    begin if gf_name[2] = xchr["v"] then
      begin verbose ← true; argv(2, gf_name); incr(pk_arg)
      end
    else usage;
    end;
  print_ln(banner);
  if test_read_access(gf_name, GFFILEPATH) then
    begin reset(gf_file, gf_name)
    end
  else begin print_pascal_string(gf_name); abort(': GF file not found.');
    end;
  gf_loc ← 0;
  end;
procedure open_pk_file; { prepares to write packed bytes in pk_file }
  var j, k: integer; pk_name: UNIX_file_name;
  begin if argc = pk_arg then argv(argc − 1, pk_name)
  else begin j ← FILENAMESIZE; k ← 1;
    while (j > 1) ∧ (gf_name[j] ≠ xchr["/"]) do
      decr(j);
    if gf_name[j] = xchr["/"] then incr(j); { to avoid picking up the / }
    print(xchr["["]); print(xchr["␣"]);
    while (j < FILENAMESIZE) ∧ (¬(gf_name[j] = xchr["."]) ∨ (gf_name[j] = xchr["␣"])) do
      begin pk_name[k] ← gf_name[j]; print(xchr[xord[gf_name[j]]]); incr(j); incr(k)
      end;
    while (j < FILENAMESIZE) ∧ ¬(gf_name[j] = xchr["g"]) do
      begin if gf_name[j] = xchr["␣"] then abort('No `gf'' in suffix!');
      pk_name[k] ← gf_name[j]; print(xchr[xord[gf_name[j]]]); incr(k); incr(j)
      end;
    print(xchr[xord[gf_name[j]]]); incr(j); print(xchr[xord[gf_name[j]]]); print(xchr["␣"]);
    print(xchr["-"]); print(xchr[">"]); print(xchr["␣"]); pk_name[k] ← xchr["p"]; incr(k);
    pk_name[k] ← xchr["k"]; incr(k); pk_name[k] ← xchr["␣"];
    for j ← 1 to k do print(xchr[xord[pk_name[j]]]);
    print_ln(xchr["]"])
    end;
  rewrite(pk_file, pk_name); pk_loc ← 0; pk_open ← true
  end;
```

44.* 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 + ´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 ≡ gf_len ← gf_length
function gf_length: integer;
  begin checked_fseek(gf_file, 0, 2); gf_length ← ftell(gf_file);
  end;
procedure move_to_byte(n : integer);
  begin checked_fseek(gf_file, n, 0);
  end;
```

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 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
```

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 bad ← false; 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);
      if bad then abort('Ran␣out␣of␣internal␣memory␣for␣row␣counts!');
      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, '␣character␣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 }
bad: boolean;  { did we run out of space? }
```

See also section 61.

This code is used in section 51*.

82:* Of course, we need an array to hold the comment.

⟨ Globals in the outer block 11 ⟩ +≡
comment: **packed array** [1 .. *comm_length* + 1] **of** *char*;

83:* ⟨ Set initial values 12 ⟩ +≡
vstrcpy(*comment* + 1, *preamble_comment*);

86:* Finally, the main program.

begin *initialize*; *convert_gf_file*; ⟨ Check for un-rasterized locaters 85 ⟩;
print_ln(*gf_len* : 1, ´␣`bytes␣packed␣to␣´, *pk_loc* : 1, ´␣`bytes.´);
end.

89* 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, 8, 37, 38, 39, 44, 46, 51, 52, 57, 58, 82, 83, 86, 89.

⟨ Calculate *dyn_f* and packed size and write character 68 ⟩ Used in section 62.
⟨ Cases for *paint* commands 59 ⟩ Used in section 57*.
⟨ Check for un-rasterized locaters 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.
⟨ Find and interpret postamble 60 ⟩ Used in section 51*.
⟨ Globals in the outer block 11, 38*, 40, 45, 47, 48, 55, 78, 82*, 87 ⟩ Used in section 4*.
⟨ 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 *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, 41, 49, 79, 83* ⟩ Used in section 4*.
⟨ Skip over repeated rows 66 ⟩ Used in section 64.
⟨ Specials and *no_op* cases 53 ⟩ Used in sections 51*, 57*, and 60.
⟨ Types in the outer block 9, 10, 37* ⟩ Used in section 4*.
⟨ 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.

The GFtoPK processor

(Version 2.3, 29 July 1990)

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.