The PKtype processor

(Version 2.3, 23 April 2020)
The banner string defined here should be changed whenever PKtype gets modified.

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

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

```pascal
define print_in(#) ≡ write_in(output,#)
define print(#) ≡ write(output,#)
define typ_file ≡ stdout
define t_print_in(#) ≡ write_in(typ_file,#)
define t_print(#) ≡ write(typ_file,#)
```

```pascal
program PKtype(input, output);
type ⟨ Types in the outer block 9 ⟩
var ⟨ Globals in the outer block 11 ⟩
   ⟨ Define parse_arguments 56* ⟩
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('PKTYPE', 0, nil, nil);
      parse_arguments; print(banner); print_in(version_string);
      ⟨ Set initial values 12 ⟩
      end;
```

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

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

We use a call to the external C exit to avoid a non-local goto.

```pascal
define abort(#) ≡
   begin print_in(#); uexit(1)
   end
```
10* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like PKtype. So we shall assume that the Pascal system being used for PKtype has a character set containing at least the standard visible characters of ASCII code ("!" through "~").

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

\[
\begin{align*}
\text{define } & \text{char } \equiv 0 \ldots 255 \quad \text{the data type of characters in text files} \\
\text{define } & \text{text_char } \equiv \text{char} \quad \text{ordinal number of the smallest element of text_char} \\
\text{define } & \text{first_text_char } = 0 \quad \text{ordinal number of the largest element of text_char} \\
\text{define } & \text{last_text_char } = 127 \quad \text{packed file of text_char}
\end{align*}
\]
31* ⟨Globals in the outer block 11⟩ +≡

\texttt{pk\_file: byte\_file; \{ where the input comes from \}}

32* In C, do path searching.

\textbf{procedure} \texttt{open\_pk\_file;} \{ prepares to read packed bytes in \texttt{pk\_file} \}
\begin{itemize}
\item \texttt{begin} \{ Don’t use \texttt{kpse\_find\_pk}; we want the exact file or nothing. \}
\item \texttt{pk\_file ← kpse\_open\_file(cmdline(1), kpse\_pk\_format); cur\_loc ← 0;}
\item \texttt{end;}
\end{itemize}

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

⟨Globals in the outer block 11⟩ +≡

\texttt{pk\_name: c\_string; \{ name of input and output files \}}
\texttt{cur\_loc: integer; \{ how many bytes have we read? \}}
We shall use a set of simple functions to read the next byte or bytes from `pk_file`. There are seven possibilities, each of which is treated as a separate function in order to minimize the overhead for subroutine calls. We comment out the ones we don’t need.

```c
#define pk_byte ≡ get_byte
#define pk_loc ≡ cur_loc

function get_byte: integer;  { returns the next byte, unsigned }
  var b: eight_bits;
  begin if eof (pk_file) then get_byte ← 0
  else begin
    read (pk_file, b); incr (cur_loc); get_byte ← b;
  end;
end;
@{
function signed_byte: integer;  { returns the next byte, signed }
  var b: eight_bits;
  begin
    read (pk_file, b); incr (cur_loc);
    if b < 128 then signed_byte ← b else signed_byte ← b - 256;
  end;
}@

function get_two_bytes: integer;  { returns the next two bytes, unsigned }
  var a, b: eight_bits;
  begin
    read (pk_file, a);
    read (pk_file, b);
    cur_loc ← cur_loc + 2;
    get_two_bytes ← a * 256 + b;
  end;
@{
function signed_pair: integer;  { returns the next two bytes, signed }
  var a, b: eight_bits;
  begin
    read (pk_file, a);
    read (pk_file, b);
    cur_loc ← cur_loc + 2;
    if a < 128 then signed_pair ← a * 256 + b
    else signed_pair ← (a - 256) * 256 + b;
  end;
}@{

function get_three_bytes: integer;  { returns the next three bytes, unsigned }
  var a, b, c: eight_bits;
  begin
    read (pk_file, a);
    read (pk_file, b);
    read (pk_file, c);
    cur_loc ← cur_loc + 3;
    get_three_bytes ← (a * 256 + b) * 256 + c;
  end;
}@{

function signed_trio: integer;  { returns the next three bytes, signed }
  var a, b, c: eight_bits;
  begin
    read (pk_file, a);
    read (pk_file, b);
    read (pk_file, c);
    cur_loc ← cur_loc + 3;
    if a < 128 then signed_trio ← (a * 256 + b) * 256 + c
    else signed_trio ← ((a - 256) * 256 + b) * 256 + c;
  end;
}@{

function signed_quad: integer;  { returns the next four bytes, signed }
  var a, b, c, d: eight_bits;
  begin
    read (pk_file, a);
    read (pk_file, b);
    read (pk_file, c);
    read (pk_file, d);
    cur_loc ← cur_loc + 4;
    if a < 128 then signed_quad ← ((a * 256 + b) * 256 + c) * 256 + d
    else signed_quad ← (((a - 256) * 256 + b) * 256 + c) * 256 + d;
  end;
}@
```
This module was needed when output was directed to \textit{typ\_file}. It is not needed when output goes to \textit{stdout}.

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

\begin{itemize}
  \item \textbf{define} \texttt{get\_16} \equiv \texttt{get\_two\_bytes}
  \item \textbf{define} \texttt{get\_32} \equiv \texttt{signed\_quad}
\end{itemize}
52* If any specials are found, we write them out here.

define four_cases(#) \equiv #, # + 1, # + 2, # + 3

procedure skip_specials;
  var i, j: integer;
begin repeat
  flag_byte \leftarrow pk_byte;
  if flag_byte \geq 240 then
    case flag_byte of
      four_cases(pk_xx1): begin t_print((pk_loc - 1) : 1, \``Special:```); i \leftarrow 0;
        for j \leftarrow pk_xx1 to flag_byte do i \leftarrow 256 \ast i + pk_byte;
        for j \leftarrow 1 to i do t_print(xchr[pk_byte]);
        t_print(ln(````); end;
      pk_yyy: begin t_print((pk_loc - 1) : 1); t_print ln(````Num_special:```); get_32 : 1); end;
      pk_post: t_print ln((pk_loc - 1) : 1, \``Postamble``);
      pk_no_op: t_print ln((pk_loc - 1) : 1, \``No_op``);
      pk_pre, pk_undefined: abort(````Unexpected```, flag_byte : 1, ```!```);
    endcases
  until (flag_byte < 240) \lor (flag_byte = pk_post);
end;
Terminal communication. There isn’t any.

So there is no **procedure** dialog.
§55* The main program. Now that we have all the pieces written, let us put them together.

begin initialize; open pk_file; ⟨Read preamble 38⟩;
skip_specials;
while flag_byte ≠ pk_post do
  begin ⟨Unpack and write character 40⟩;
    skip_specials;
  end;
j ← 0;
while ¬eof(pk_file) do
  begin i ← pk_byte;
    if i ≠ pk_no_op then abort(‘Bad byte at end of file: i: 1);
    t_print ln((pk_loc - 1): 1, ‘No op’); incr(j);
  end;
t_print ln(pk_loc: 1, ‘bytes read from packed file.’);
end.
56* System-dependent changes. Parse a Unix-style command line.

```c
#define argument_is(#) (strcmp(long_options[option_index].name, #) == 0)

(Define parse_arguments 56*)

procedure parse_arguments;
const n_options = 2;  /* Pascal won’t count array lengths for us. */
var long_options: array [0..n_options] of getopt_struct;
  getopt_return: integer; option_index: c_int_type; current_option: 0..n_options;
begin (Define the option table 57*);
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
  else if getopt_return_val = ”?" then
    begin usage(my_name);
    end
  else if argument_is(’help’) then
    begin usage_help(PKTYPE_HELP, nil);
    end
  else if argument_is(’version’) then
    begin print_version_and_exit(banner, nil, ’Tomas_Rokicki’, nil);
    end:  /* Else it was just 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. */
if (optind + 1 ≠ argc) then
  begin writeLn(stderr, my_name, ’: 0Need exactly 0one file argument.*); usage(my_name);
  end;
end;
```

This code is used in section 4*.

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

```c
(Define the option table 57*)

#define 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 58* and 59*.

This code is used in section 56*.

58* Another of the standard options.

```c
(Define the option table 57*)

#define 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);
```

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

```c
(Define the option table 57*)

#define long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```
60* 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: 2, 4, 5, 6, 8, 10, 31, 32, 33, 34, 35, 36, 52, 53, 54, 55, 56, 57, 58, 59, 60.

-help: 57*
-version: 58*
a: 34*
abort: 8* 23, 38, 40, 50, 52* 55*
address_of: 56*
argv: 56*
argument_is: 56*
argv: 4* 56*
ASCII_code: 9, 11.
b: 34*
Bad byte at end of file: 55*
Bad packet length: 40.
banner: 9* 4* 56*
bit_weight: 45, 47, 48.
boolean: 41, 45, 46, 51.
byte_file: 30, 31*.
c: 34*
c_int_type: 56*
c_string: 33*
car: 40, 41, 42, 43, 44.
cc: 25.
char: 10*
checksum: 38, 39.
chr: 10* 11, 13.
 cmdline: 32*.
count: 50, 51.
cs: 16.
cur_loc: 32* 33* 34*.
current_option: 56* 57* 58* 59*.
d: 34*
dealloc: 7, 23.
design_size: 38, 39.
dialog: 54*.
dm: 25.
do_nothing: 7, 56*.
ds: 16.
dx: 25, 40, 41, 42, 43, 44.
dxs: 41.
dy: 25, 40, 41, 42, 43, 44.
dys: 41.
dyn_f: 21, 22, 23, 24, 25, 28, 29, 40, 41, 48, 49.
dys: 41.
eight_bits: 30, 34*, 45, 47.
else: 3.
end: 3.
end_of_first_char: 40, 41, 42, 43, 44.
endcases: 3.
eof: 34* 55*.
false: 50.
first_text_char: 10* 13.
Wrong version of PK file: 38.

PK type changes for C

usage: 56*
usage_help: 56*
val: 57*, 58*, 59*
value: 46.
version_string: 4*
voff: 25, 27.
vppp: 16, 38, 39.
width: 24, 40, 41, 42, 43, 44, 49, 50.
write: 4*
write_ln: 4*, 56*.

Second repeat count...: 23.

pk_type: 4*
PKTYPE_HELP: 56*

pre command missing: 38.
print: 4*
print Ln: 4*, 8*, 38.
print_version_and_exit: 56*.
read: 34*.
repeat_count: 23, 46, 50, 51.
round: 38.
rows_left: 50, 51.
scaled: 16.

send_out: 23, 46, 50.
signed_byte: 34*.
signed_pair: 34*.
signed_quad: 34*, 36*.
signed_trio: 34*.
skip_specials: 52*, 55*.
status: 41.
stderr: 56*.
stdout: 4*, 35*.
strcmp: 56*.

system dependencies: 6*, 10*, 30, 31*, 34*.
t_print: 4*, 38, 40, 46, 49, 50, 52*.
t_print_Ln: 4*, 38, 40, 46, 49, 50, 52*, 55*.
temp: 45.
term_pos: 37, 46, 50.
text_char: 10*, 11.
text_file: 10*.
tfm: 25, 26, 29.
tfm_width: 40, 41, 42, 43, 44.
tfms: 41.
true: 23.
turn_on: 40, 46, 50, 51.
uxit: 8*.

Unexpected bbb: 52*.
PK type changes for C

-names of the sections-

- Create normally packed raster 50 Used in section 48.
- Define the option table 57*, 58*, 59* Used in section 56*.
- Define parse_arguments 56* Used in section 4*.
- Get raster by bits 49 Used in section 48.
- Globals in the outer block 11, 31*, 33*, 37, 39, 41, 47, 51 Used in section 4*.
- Packed number procedure 23 Used in section 46.
- Read and translate raster description 48 Used in section 40.
- Read extended short character preamble 43 Used in section 40.
- Read long character preamble 42 Used in section 40.
- Read preamble 38 Used in section 55*.
- Read short character preamble 44 Used in section 40.
- Set initial values 12, 13 Used in section 4*.
- Types in the outer block 9, 10*, 30 Used in section 4*.
- Unpack and write character 40 Used in section 55*.