The Portable C Library (on UNIX) *

M. E. Lesk

1. INTRODUCTION

The C language [1] now exists on three operating systems. A set of library routines common to PDP 11 UNIX, Honeywell 6000 GCOS, and IBM 370 OS has been provided to improve program portability. This memorandum describes the UNIX implementation of the portable routines.

The programs defined here were chosen to follow the standard routines available on UNIX, with alterations to improve transferability to other computer systems. It is expected that future C implementations will try to support the basic library outlined in this document. It provides character stream input and output on multiple files; simple accessing of files by name; and some elementary formatting and translating routines. The remainder of this memorandum lists the portable and non-portable library routines and explains some of the programming aids available.

The I/O routines in the C library fall into several classes. Files are addressed through intermediate numbers called *file-descriptors* which are described in section 2. Several default file-descriptors are provided by the system; other aspects of the system environment are explained in section 3.

Basic character-stream input and output involves the reading or writing of files considered as streams of characters. The C library includes facilities for this, discussed in section 4. Higher-level character stream operations permit translation of internal binary representations of numbers to and from character representations, and formatting or unpacking of character data. These operations are performed with the subprograms in section 5. Binary input and output routines permit data transmission without the cost of translation to or from readable ASCII character representations. Such data transmission should only be directed to files or tapes, and not to printers or terminals. As is usual with such routines, the only simple guarantee that can be made to the programmer seeking portability is that data written by a particular sequence of binary writes, if read by the exactly matching sequence of binary reads, will restore the previous contents of memory. Other reads or writes have system-dependent effects. See section 6 for a discussion of binary input and output. Section 7 describes some further routines in the portable library. These include a storage allocator and some other control and conversion functions.

2. FILE DESCRIPTORS

Except for the standard input and output files, all files must be explicitly opened before any I/O is performed on them. When files are opened for writing, they are created if not already present. They must be closed when finished, although the normal *cexit* routine will take care of that. When opened a disc file or device is associated with a file descriptor, an integer between 0 and 9. This file descriptor is used for further I/O to the file.

Initially you are given three file descriptors by the system: 0, 1, and 2. File 0 is the standard input; it is normally the teletype in time-sharing or input data cards in batch. File 1 is the standard output; it is normally the teletype in time-sharing or the line printer in batch. File 2 is the error file; it is an output file, normally the same as file 1, except that when file 1 is diverted via a command line '>' operator, file 2 remains attached to the original destination, usually the terminal. It is used for error message output. These popular UNIX conventions are considered part of the C library specification. By closing 0 or 1, the default input or output may be re-directed; this can also be done on the command line by >*file* for output or <*file* for input.

Associated with the portable library are two external integers, named *cin* and *cout.* These are respectively the numbers of the standard input unit and standard output unit. Initially 0 and 1 are used, but you

* This document is an abbreviated form of ''The Portable C Library'', by M. E. Lesk, describing only the UNIX section of the library.

may redefine them at any time. These cells are used by the routines *getchar, putchar, gets,* and *puts* to select their I-O unit number.

3. THE C ENVIRONMENT

The C language is almost exactly the same on all machines, except for essential machine differences such as word length and number of characters per word. On UNIX ASCII character code is used. Characters range from −128 to +127 in numeric value, there is sign extension when characters are assigned to integers, and right shifts are arithmetic. The ''first'' character in a word is stored in the right half word.

More serious problems of compatibility are caused by the loaders on the different operating systems.

UNIX permits external names to be in upper and lower case, up to seven characters long. There may be multiple external definitions (uninitialized) of the same name.

The C alphabet for identifier names includes the upper and lower case letters, the digits, and the underline. It is not possible for C programs to communicate with FORTRAN programs.

4. BASIC CHARACTER STREAM ROUTINES

These routines transfer streams of characters in and out of C programs. Interpretation of the characters is left to the user. Facilities for interpreting numerical strings are presented in section 5; and routines to transfer binary data to and from files or devices are discussed in section 6. In the following routine descriptions, the optional argument *fd* represents a file-descriptor; if not present, it is taken to be 0 for input and 1 for output. When your program starts, remember that these are associated with the ''standard'' input and output files.

COPEN (filename, type)

Copen initiates activity on a file; if necessary it will create the file too. Up to 10 files may be open at one time. When called as described here, *copen* returns a filedescriptor for a character stream file. Values less than zero returned by *copen* indicate an error trying to open the file. Other calls to *copen* are described in sections 6 and 7.

Arguments :

Filename: a string representing a file name, according to the local operating system conventions. All accept a string of letters and digits as a legal file name, although leading digits are not recommended on GCOS.
Type: a character 'r', 'w', or 'a' meaning read, write, or append. Note that the type is a single character, whereas the file name must be a string.

CGETC (fd)

Cgetc returns the next character from the input unit associated with *fd*. On end of file *cgetc* returns '\0'. To signal end of file from the teletype, type the special symbol appropriate to UNIX: EOT (control-D)

CPUTC (ch , fd)

Cputc writes a character onto the given output unit. *Cputc* returns as its value the character written.

Output for disk files is buffered in 512 character units, irrespective of newlines; teletype output goes character by character

CCLOSE (fd)

Activity on file *fd* is terminated and any output buffers are emptied. You usually don't have to call *cclose; cexit* will do it for you on all open files. However, to write some data on a file and then read it back in, the correct sequence is:

```
fd = copen (''file'', 'w');
write on fd ...
cclose (fd);
fd = copen(''file'', 'r');
read from fd ...
```

CFLUSH (fn)

To get buffer flushing, but retain the ability to write more on the file, you may call this routine.

Normally, output intended for the teletype is not buffered and this call is not needed.

CEXIT ([errcode])

Cexit closes all files and then terminates execution. If a non-zero argument is given, this is assumed to be an error indication or other returned value to be signalled to the operating system.

Cexit **must** be called explicitly; a return from the main program is not adequate.

CEOF (fd)

Ceof returns nonzero when end of file has been reached on input unit *fd.*

GETCHAR ()

Getchar is a special case of *cgetc;* it reads one character from the standard input unit. *Getchar ( )* is defined as *cgetc (cin);* it should not have an argument.

PUTCHAR (ch)

Putchar (ch) is the same as *cputc (ch, cout);* it writes one character on the standard output.

GETS (s)

Gets reads everything up to the next newline into the string pointed to by *s.* If the last character read from this input unit was newline, then *gets* reads the next line, which on GCOS and IBM corresponds exactly to a logical record. The terminating newline is replaced by '\0'. The value of *gets* is *s,* or 0 if end of file.

PUTS (s)

Copies the string *s* onto the standard output unit. The terminating '\0' is replaced by a newline character. The value of *puts* is *s.*

UNGETC (ch , fd)

Ungetc pushes back its character argument to the unit *fd,* which must be open for input. After *ungetc ('a', fd); ungetc ('b', fd);* the next two characters to be read from *fd* will be 'b' and then 'a'. Up to 100 characters may be pushed back on each file. This subroutine permits a program to read past the end of its input, and then restore it for the next routine to read. It is impossible to change an external file with *ungetc;* its purpose is only for internal communications, most particularly *scanf,* which is described in section 5. Note that *scanf* actually requires only one character of ''unget'' capability; thus it is possible that future implementors may change the specification of the *ungetc* routine.

5. HIGH-LEVEL CHARACTER STREAM ROUTINES

These two routines, *printf* for output and *scanf* for input, permit simple translation to and from character representations of numerical quantities. They also allow generation or interpretation of formatted lines.

PRINTF ([fd,] control-string, arg1, arg2, ...)

PRINTF ([−1, output-string,] control-string, arg1, arg2, ...)

Printf converts, formats, and prints its arguments under control of the control string. The control string contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to *printf*.

Each conversion specification is introduced by the character '%'. Following the '%', there may be:

— an optional minus sign '−' which specifies left adjustment of the converted argument in the indicated field;

— an optional digit string specifying a minimum field width; if the converted argument has fewer characters than the field width it will be padded on the left (or right, if the left adjustment indicator has been given) to make up the field width; the padding character is blank normally and zero if the field width was specified with a leading zero (note that this does not imply an octal field width);

— an optional period '.' which serves to separate the field width from the next digit string;

— an optional digit string (the precision) which specifies the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point of a floating or double number.

— an optional length modifier 'l' which indicates that the corresponding data item is a *long* rather than an *int.*

— a character which indicates the type of conversion to be applied.

The conversion characters and their meanings are:

d The argument is converted to decimal notation.

o The argument is converted to octal notation.

x The argument is converted to hexadecimal notation.

u The argument is converted to unsigned decimal notation. This is only implemented (or useful) on UNIX.

c The argument is taken to be a single character.

s The argument is taken to be a string and characters from the string are printed until a null character is reached or until the number of characters indicated by the precision specification is exhausted.

e The argument is taken to be a float or double and converted to decimal notation of the form *[−] m.nnnnnnE [−] xx* where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22.

f The argument is taken to be a float or double and converted to decimal notation of the form *[−]mmm.nnnnn* where the length of the string of *n*'s is specified by the precision. The default precision is 6 and the maximum is 22. Note that the precision does not determine the number of significant digits printed in **f** format.

If no recognizable conversion character appears after the '%', that character is printed; thus '%' may be printed by use of the string "%%".

As an example of *printf,* the following program fragment

```
int i, j; float x; char *s;
i = 35; j=2; x= 1.732; s = "ritchie";
printf ("%d %f %s\n", i, x, s);
printf ("%o, %4d or %−4d%5.5s\n", i, j, j, s);
```

would print

```
35  1.732000  ritchie
043,      2  or  2     ritch
```

If *fd* is not specified, output is to unit *cout.* It is possible to direct output to a string instead of to a file. This is indicated by −1 as the first argument. The second argument should be a pointer to the string. *Printf* will put a terminating '\0' onto the string.

SCANF ([fd,] control-string, arg1, arg2, ...)

SCANF ([−1, input-string,] control-string, arg1, arg2, ...)

Scanf reads characters, interprets them according to a format, and stores the results in its arguments. It expects as arguments:
1. An optional file-descriptor or input-string, indicating the source of the input characters; if omitted, file *cin* is used;
2. A control string, described below;
3. A set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs or newlines, which are ignored.
2. Ordinary characters (not %) which are expected to match the next non-space character of the input stream (where space characters are defined as blank, tab or newline).
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character ∗, an optional numerical maximum field width, and a conversion character.

A conversion specification is used to direct the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by the ∗ character. An input field is defined as a string of non-space characters; it extends either to the next space character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. Pointers, rather than variable names, are required by the ''call-by-value'' semantics of the C language. The following conversion characters are legal:

% indicates that a single % character is expected in the input stream at this point; no assignment is done.

d indicates that a decimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

o indicates that an octal integer is expected in the input stream; the corresponding argument should be a integer pointer.

x indicates that a hexadecimal integer is expected in the input stream; the corresponding argument should be an integer pointer.

s indicates that a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating '\0', which will be added. The input field is terminated by a space character or a newline.

c indicates that a single character is expected; the corresponding argument should be a character pointer; the next input character is placed at the indicated spot. The normal skip over space characters is suppressed in this case; to read the next non-space character, try *%1s.*

e or **f** indicates that a floating point number is expected in the input stream; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float. The input format for *floats* is a string of numbers possibly containing a decimal point, followed by an optional exponent field containing an E or e followed by a possibly signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (ˆ), the input field is all

characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character which is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters *d, o* and *x* may be preceded by *l* to indicate that a pointer to *long* rather than *int* is expected. Similarly, the conversion characters *e* or *f* may be preceded by *l* to indicate that a pointer to *double* rather than *float* is in the argument list. The character *h* will function similarly in the future to indicate *short* data items.

For example, the call

```
int i; float x; char name[50];
scanf ( "%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E-1  thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain *"thompson\0"*. Or,

```
int i; float x; char name[50];
scanf ("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i,* 789.0 to *x,* skip "0123", and place the string "56\0" in *name.* The next call to *cgetc* will return 'a'.

Scanf returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, −1 is returned; note that this is different from 0, which means that the next input character does not match what you called for in the control string. *Scanf,* if given a first argument of −1, will scan a string in memory given as the second argument. For example, if you want to read up to four numbers from an input line and find out how many there were, you could try

```
int a[4], amax;
char line[100];
amax = scanf(-1, gets(line), "%d%d%d%d", &a[0], &a[1], &a[2], &a[3]);
```

6. BINARY STREAM ROUTINES

These routines write binary data, not translated to printable characters. They are normally efficient but do not produce files that can be printed or easily interpreted. No special information is added to the records and thus they can be handled by other programming systems *if* you make the departure from portability required to tell the other system how big a C item (integer, float, structure, etc.) really is in machine units.

COPEN (name, direction, "i")

When *copen* is called with a third argument as above, a binary stream filedescriptor is returned. Such a file descriptor is required for the remaining subroutines in this section, and may not be used with the routines in the preceding two sections. The first two arguments operate exactly as described in section 3; further details are given in section 7. An ordinary file descriptor may be used for binary I-O, but binary and character I-O may not be mixed unless *cflush* is called at each switch to binary I-O. The third argument to *copen* is ignored.

*CWRITE (ptr, sizeof(*ptr), nitems, fd)*

Cwrite writes *nitems* of data beginning at *ptr* on file *fd. Cwrite* writes blocks of binary information, not translated to printable form, on a file. It is intended for machine-oriented bulk storage of intermediate data. Any kind of data may be written with this command, but only the corresponding *cread* should be expected to make any sense of it on return. The first argument is a pointer to the beginning of a vector of any kind of data. The second argument tells *cwrite* how big the items are. The third argument specifies the number of the items to be written; the fourth indicates where.

CREAD (ptr, sizeof(∗ptr), nitems, fd)

Cread reads up to *nitems* of data from file *fd* into a buffer beginning at *ptr.* *Cread* returns the number of items read.
The returned number of items will be equal to the number requested by *nitems* except for reading certain devices (e.g. the teletype or magnetic tape) or reading the final bytes of a disk file.

Again, the second argument indicates the size of the data items being read.

CCLOSE (fd)

The same description applies as for character-stream files.

7. OTHER PORTABLE ROUTINES

REW (fd)

Rewinds unit *fd.* Buffers are emptied properly and the file is left open.

SYSTEM (string)

The given *string* is executed as if it were typed at the terminal.

NARGS ( )

A subroutine can call this function to try to find out how many arguments it was called with. Normally, *nargs()* returns the number of arguments plus 3 for every *float* or *double* argument and plus one for every *long* argument. If the new UNIX feature of separated instruction and data space areas is used, *nargs()* doesn't work at all.

CALLOC (n, sizeof(object))

Calloc returns a pointer to new storage, allocated in space obtained from the operating system. The space obtained is well enough aligned for any use, i.e. for a double-precision number. Enough space to store *n* objects of the size indicated by the second argument is provided. The *sizeof* is executed at compile time; it is not in the library. A returned value of −1 indicates failure to obtain space.

*CFREE (ptr, n, sizeof(*ptr))*

Cfree returns to the operating system memory starting at *ptr* and extending for *n* units of the size given by the third argument. The space should have been obtained through *calloc*. On UNIX you can only return the exact amount of space obtained by *calloc;* the second and third arguments are ignored.

FTOA (floating-number, char-string, precision, format)

Ftoa (floating to ASCII conversion) converts floating point numbers to character strings. The *format* argument should be either 'f' or 'e'; 'e' is default. See the explanation of *printf* in section 5 for a description of the result.

ATOF (char-string)

Returns a floating value equal to the value of the ASCII character string argument, interpreted as a decimal floating point number.

TMPNAM (str)

This routine places in the character array expected as its argument a string which is legal to use as a file name and which is guaranteed to be unique among all jobs executing on the computer at the same time. It is thus appropriate for use as a temporary file name, although the user may wish to move it into an appropriate directory. The value of the function is the address of the string.

ABORT (code)

Causes your program to terminate abnormally, which typically results in a dump by the operating system.

INTSS ()

This routine tells you whether you are running in foreground or background. The definition of ‘‘foreground’’ is that the standard input is the terminal.

WDLENG ( )

This returns 16 on UNIX. C users should be aware that the preprocessor normally provides a defined symbol suitable for distinguishing the local system; thus on UNIX the symbol *unix* is defined before starting to compile your program.