1007

The Portable C Library

M. E. Lesk

Bell Laboratories,
Murray Hill, New Jersey 07974

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 routines on the different systems. It is the sole current reference to libraries for GCOS and OS, and supplements the UNIX programmer's manual. A variety of programming aids available for C users, such as measurement and debugging aids, are also described.

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. Appendix 1 provides an index to the currently available routines.

We will refer to several subroutine libraries in the following sections. Each host computer provides a non-portable *system library,* described in the appropriate system reference manuals [2,3,4]. To simplify the task of writing portable software, the *portable C library* described here permits basic input and output operations on multiple files to be performed without using system-dependent calls. This library will be provided, at least as an alternative, in each C run-time environment. Additional features are offered in specific environments; when these routines are available in several places an effort is made to define them compatibly. These supplementary capabilities may be provided in future C systems, although many are only of interest on a particular operating system, and some routines may be difficult to implement in future run-time environments.

In general, statements in this memo apply to all of the systems where C is implemented. When this is not true, a three-column format is used, in which the left column is applicable to UNIX, the middle column to GCOS and the right column to OS/370.

UNIX	GCOS	OS/370

Two simple subroutines will be adequate for many users. *Getchar ()* returns a character from the standard input, usually the teletype; and *putchar (c)* writes a character on the standard output, also usually the teletype. *Putchar* returns as its value the character written. By C convention, '\0' indicates end of file or end of string. Thus the program

```
main ()
{
while (putchar(getchar()) != '\0');
}
```

will echo lines typed at it.

To compile and run this program, it should first be entered with the local editor into a file named

prog.c	*prog.c*	*prog.text*

The command dialog to compile and execute would appear as follows:

```
% cc prog.c -lp          SYSTEM ?  ./cc prog.c h=     READY
% a.out                  SYSTEM ?  go .program        ccg prog
```

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. Non-portable routines are described in sections 8U, 8G and 8I. The commands to compile a C program on UNIX, GCOS and IBM TSO are described in sections 9U, 9G and 9I respectively. Some useful utilities and support software are described in sections 10 and 11. Finally, when you get into trouble with all of this, the C debugging facilities are described in section 12.

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.

Thus, suppose the program above to be stored in executable form on the file

prog	*prog.h*	*cllm.load(prog)*

If it is invoked by the command line

prog <data	*/prog.h <data*	*call cllm.load(prog) 'prog <data'*

the file *data* is listed on the teletype. If invoked by

prog >newdata	*/prog.h >newdata*	*call cllm.load(prog) 'prog >newdata'*

lines typed at the terminal are written into the file *newdata;* and if invoked by

prog <old >new	*/prog.h <old >new*	*call cllm.load(prog) 'prog <old >new'*

the contents of the file *old* are copied into the file *new;* in the last two examples, the files *newdata* and *new* will be created if not present.

The running program, of course, has access to the command line via the main program arguments, except for ">file" or "<file" arguments, which are handled by the start-up routine. If you intend to provide arguments on the command line, your main program should begin

```
main (argc, argv)
int argc; char *argv[];
{
```

where *argc* will be set to the number of command line arguments, and *argv* will be a vector of pointers to the successive arguments as character strings; note that *argv[0]* normally gives the command name by which the program was invoked.

In GCOS batch the command line is read from filecode CZ; it will be supplied for any batch job submitted by the *./sh* driver. This "pseudo-shell" is described in section 10.

Note that on OS/370 the "command line" is the string normally called the PARM string in IBM terminology.

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 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

In general, C requires some modifications to the standard system runtime environment to support recursion and call-by-value. An effort is made to minimize the effect of this on programmers accustomed to the standard system environments, as well as providing the portable facilities needed. The language is almost exactly the same on all machines, except for essential machine differences such as word length and number of characters per word.

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.	ASCII code is used. Characters range from 0 to +511 in numeric value, there is no sign extension on character to integer conversion, and right shifts are logical. The "first" character in a word is stored in the left-most quarter word.	EBCDIC character code is used. Characters range from 0 to +255, no sign extension, and logical right shifts. The "first" character in a word is stored in the leftmost quarter word. Floating point, and the operators =*, =/, =%, =>>, and =<< are not yet implemented.

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

External names may be upper and lower case, up to seven characters long. There may be multiple external definitions (uninitialized) of the same name.	External names must be unique in the first six characters of a one-case alphabet. There may be only one external definition of a given name; all other uses must be references.	External names are one-case, but may be up to eight characters long. There may be only one external definition of a given name; all other uses must be references.

The C alphabet for identifier names includes the upper and lower case letters, the digits, and the underline. To conform with loader requirements,

underline is left alone.	underline is translated to '.' in external names.	underline is translated to '#' in external names.

A serious problem faced by C users in non-UNIX environments is calling non-C library routines. In general, these are likely to conform to FORTRAN specifications. The basic incompatibilities arise from the FORTRAN view that subroutines arguments are passed by address, not value, and that passing an array name is the same as passing the first element of the array.

The operating system is written in C, and communication with it is easy. It is not possible to communicate with FORTRAN, since the FORTRAN compiler does not actually produce independent object programs.	Scalar integers and doubles can be passed to and from FORTRAN. Vectors can be passed from C to FORTRAN by passing an argument of *p[0]*, where *p* is an integer pointer to the beginning of the vector. Passing an array from FORTRAN to C requires use of the *xnargs* routine described in section 8.	It is not now possible to communicate with standard OS programs without editing the assembly code produced by the compiler. The IBM calling sequence for C routines is being changed; when the new sequence is installed, the rules for FORTRAN-C interchange will be similar to the rules on GCOS.

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.

On OS/370 opening files with the 'a' option (to append to the end) is not yet implemented.

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 the given operating system:

EOT (control-D)	FS (control-\)	/*

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	If you write more than 511 characters without a newline, one will silently be inserted. No actual writing of characters takes place until a newline is written.	If you write more than 255 characters without a newline, one will silently be inserted. No actual writing of characters takes place until a newline is written. Two consecutive newlines have a blank inserted between them.

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:

All Systems

```
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.

All output is buffered into lines, and calling *cflush* causes a newline to be inserted.

All output is buffered into lines, and calling *cflush* causes a newline to be inserted.

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.

An appropriate error indication is the octal value of two BCD characters marking a GCOS abort code.

An appropriate error indication is either 4, 8, 12, or 16 to suggest the usual OS levels of errors.

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.

Gets removes trailing blanks from the input line, to compensate for their insertion by the operating system when it is padding records to fixed length.

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 (^), 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.

All GCOS records comprising a number of characters not divisible by 4 are extended to whole machine words; but the padding is carefully removed when the records are read.

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.

The third argument of "i" is required. Programs which write and read mixed mode data files will be supported on GCOS someday, but don't hold your breath.

An ordinary file descriptor may be used for binary I-O, but record lengths are limited to 255 bytes. Do not mix character and binary I-O except at record boundaries.

*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.

On GCOS, *cwrite* may only be used on files opened with the "i" option.

*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.

On UNIX, where there are no records, 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.	On GCOS, this is the number of items actually contained in the next logical record. *Cread* may only be used on files opened with the "i" option.	On OS/370, this is the number of items actually contained in the next logical record.

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.

	String must be 80 characters or less. This routine should not be used in batch operation.	The *string* should be in upper case. This routine will not work in batch.

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.		The form of the call is *nargs (ndecl)* where *ndecl* is the number of declared arguments.

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. Failure to obtain space is variously indicated:

Returns −1.	GC abort.	804 or 80A abend.

*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.

Ftoa is not available on OS/370.

ATOF (char-string)

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

Atof is not available on OS/370.

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.

Not yet implemented on TSO.

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.

The name may also be spelled *intso ().*

WDLENG ()

This deliberately different routine distinguishes the local system.

Returns 16.	Returns 36.	Returns 32.

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* and on GCOS the name *gcos* is defined before starting to compile your program.

8. FEATURES RESTRICTED TO ONE OPERATING SYSTEM

You can read this section even if you intend to write portable programs, but I don't know why you would want to. It describes routines tied to one of the three environments where C operates. These include both generally useful routines which we do not know how to define elegantly and efficiently for all operating systems, and also routines which no one would want to execute except on the specific system for which they are defined. Some of the routines in this section are of extremely specialized interest. The procedures *copen* and *cexit*, which have been described earlier in a restricted way, are defined fully in this section.

8U. UNIX SPECIFIC ROUTINES

Since most of the UNIX operating system and its software are written in C, a great many more library routines are available here for system-specific operations. Users should consult the *UNIX Programmer's Manual* for details[2].

8G. GCOS SPECIFIC ROUTINES

On GCOS, where C has been used for systems programming tasks, there is a supply of subroutines to permit access to operating system control points. Furthermore, the variety of file types on GCOS requires additional flexibility in the I-O routines to cater to those not satisfied with default file handling.

COPEN (filename, type, options)

The function *copen* has a number of other possible option arguments to produce very specific results in terms of the GCOS file system.

In analyzing the *filename*, file creation and accessing use the BFOR library *jfilac* subroutine; the overall operation follows the computation center's standard conventions:

First, throw away any permissions or alternate names, (as defined by the standard GCOS conventions; since the second argument specifies whether reading or writing is required, the permissions are redundant). Then, if *filename* does not contain a slash:

a. A file already accessed with this name is searched for in the AFT.
b. If this is unsuccessful, an attempt is made to access a permanent file with the specified name in the user's catalog.
c. If this fails and write or append operation was requested, a temporary file is created.

If *filename* contains a slash:

a. An attempt is made to access a permanent file by this name.
b. If this is unsuccessful and write or append operation was requested, a permanent file is created.

Files with the same name from different catalogs may be open simultaneously; this is done by using funny AFT names ("0." to "9.") for permanent files. When a file is closed, *cclose* deaccesses permanent files; thus the user never sees the funny aliases.

In batch, the treatment of permanent files is similar to their treatment in time-sharing. When accessed, they are given funny filecodes rather than funny AFT names. Temporary files in batch are created as needed for filenames not containing a '/'; note that such files disappear at the end of the batch activity and are only useful if they are written, closed, and read within one activity. For example, the command line *prog >junk* in TSS writes the output onto a temporary file *junk* which can later be picked up from the AFT and read; but in batch, if *junk* is not in your catalog you have just thrown away the output. (See below to find out how to provide a command line in batch.) Any file name beginning with '*' refers to an externally defined filecode; the second and third characters of the name define the filecode. When writing, temporary files are created for non-existent filecodes. Since these do not have save disposition, however, multi-activity jobs should use permanent files or filecodes defined by $ FILE control cards to pass information from one activity to the next. At the moment, parameters on $ FFILE control cards are ignored.

The *options* argument is a string made up of a combination of the following characters (other characters are ignored), which usually specifies a very GCOS-specific option for file specification:

- t: assign to teletype. In time-sharing, this file is assigned to the user's terminal; the filename is ignored. In batch, lines written on the file have printer slew added.
- 2: indicates that this is a standard system format BCD file. This option is not necessary on a file being read, but is the only way to write a BCD file. Normally such files are written in media code 2; if the slew option 's' is present they are written with code 3. Media code 2 files are padded to 80 characters to be a card image. This can be suppressed by using media code 0 (see below).
- s: printer slew is present or is to be generated. This option is only relevant for mode 2 files; and it should be omitted if no slew is desired or present. Unfortunately, GEFRC will not recognize the presence of slew on an input file, so that this option is required to read a BCD file with slew (media code 3). Note that *qed* writes BCD files without slew but Fortran usually writes them with slew.
- i: binary stream file. This file is defined to be a stream of machine-oriented rather than character-oriented quantities. See section 6 for discussion of binary stream files.
- 3: media code 3 file, i.e. BCD with printer slew. This is equivalent to "2s".
- 0: media code 2 file, i.e. BCD, no slew, not padded to a card image.
- d: discuss file accessing errors at terminal, rather than just returning an error code. This permits the user to correct file opening errors at run time.
- l: leave this file in AFT after it is closed; normally permanent files not originally present are deleted.
- f: full AFT name given; this suppresses the normal creation of C altnames for files accessed by C. If you don't understand this, you don't need it.

When *copen* returns a negative value, there has been an error trying to open the file. The specific negative number is usually that provided by the system file-access routine. If you specify the *d* option to *copen*, a message will be typed on the terminal, and you will have a chance to adjust the open request. Note in batch that the abort code MZ (not on the green cards) refers to an error in the Fortran file-accessing package.

There is one unusual filecode in batch operation: the command line, if required by the program, is expected on filecode CZ. This filecode is created by the *./sh* program if used to spawn batch jobs; alternatively, you may provide a $ DATA CZ card in your deck. This permits, for example, redirection of the standard input and output in batch.

CGETI (buff, len, fd)

Cgeti places the next record from unit *fd* into *buff*, and returns the number of integers read; 0 on eof. The unit *fd* must have been opened with the "i" option and may not be addressed by *cgetc* calls or by *ungetc*. The second argument indicates the length of your input buffer, which should be large enough for the records being read.

CPUTI (buff, len, fd)

Cputi writes *len* words from *buff* onto unit *fd* as a single GCOS record. The file *fd* must have been opened with the "i" option, and *cputc* must not be used on this file.

BACK ([fd])

The file *fd* is backspaced one logical record. Since *copen* rewinds files as they are opened, these routines need not normally be used. The direction of the file (input or output) is retained after reverse motion. See the GEFRC manual [3] or experts to understand such operations as backspacing the teletype. The default file descriptor is *cin.*

XNARGS (ptrs)

The addresses of the arguments to the calling program are placed in the array of pointers *ptrs.* The function returns the number of arguments. It does not matter how many arguments are declared; *xnargs* depends only on the actual invocation. This routine permits a C program being called by FORTRAN to pick up the positions of arrays being passed as arguments.

DRLDRL (number, arg1, arg2, ...)

This call causes the execution of the instruction sequence

```
DRL  number
arg1
arg2
...
```

with the A and Q registers taken from the external integers *.a.reg* and *.q.reg* and restored to these cells after the derail returns. For example, *drldrl (5)* (derail return) exits without closing files, etc. If you don't understand this routine, you don't need it.

MMEMME (number, arg1, arg2, ...)

This routine is the same as *drldrl* except that it executes the batch system call instruction instead of the time-sharing instruction.

8I. IBM SPECIFIC ROUTINES

This section describes routines which only apply to the IBM OS/370 C system. It is likely that as C usage on this system expands, additional routines will be added.

COPEN (name, direction, options)

The *name* given is only taken as a file name in TSO. In batch OS, dynamic file accessing is impossible, and the name is taken as a ddname. The only legal format for a ddname (and thus for a batch *name* argument to *copen)* is "ft??f001" where the question marks are replaced by digits. This name must be defined on a DD job control language statement. Such names are treated similarly in TSO (the data set must have been allocated to this name) but any other name is also legal in TSO, and is allocated by the run-time library. The format of new output files is TSO *.text* style. As on GCOS, the IBM version of *copen* takes an optional third argument to specify some details of file handling. In particular it is occasionally necessary to override the default operation of placing a blank at the head of each output line being sent to a printer. This suppresses carriage control so that C programs written without reference to it work correctly. The possible characters (given either as a single character argument, or as a member of a string) are

e	"edit mode": on output, changes lower case to upper case and tab characters into blanks.
b	Places a blank at the beginning of each output line (for carriage control on a printer).
n	Does not place a blank at the beginning of each output line, even if this stream is headed for a printer.

GENREG (regno, ndecl)

Genreg returns the value register *regno* had on entry to the calling function; *ndecl* is the number of declared arguments in the calling function.

9. CC — THE C COMPILER COMMAND

This section describes the commands to compile and load C programs on the three different operating systems. On all systems, C is designed to be used with input files of variable line length and upper/lower case ASCII character set. Files should not be line numbered. Any standard ASCII terminal suffices to enter a C program, but a terminal restricted to upper case or the PL/I character set will be almost impossible to use with C.

9U. UNIX COMPILING COMMANDS

This description of the C compiling command on UNIX is taken from the *UNIX Programmer's Manual* (© Bell Laboratories 1972, 1973, 1974, 1975). Reprinted by permission.

The format of the C compiling command on UNIX is

```
cc [ −c ] [ −p ] [ −f ] [ −O ] [ −S ] [ −P ] file ...
```

The UNIX C compiler accepts three types of arguments: source programs, compiler options, and other loader-directed information.

Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

The following compiler option flags are interpreted by *cc:*

−c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

−p Arrange for the compiler to produce code which counts the number of times each routine is called; also, if loading takes place, replace the standard startup routine by one which automatically creates the data file of profiling information during execution of the object program. An execution profile can then be generated by use of the UNIX *prof* command.

−f In systems without hardware floating-point, use a version of the C compiler which handles floating-point constants and loads the object program with the floating-point interpreter. Do not use if the hardware is present.

−O Invoke an object-code optimizer.

−S Compile the named C programs, and leave the assembler-language output on corresponding files suffixed '.s'.

−P Run only the macro preprocessor on the named C programs, and leave the output on corresponding files suffixed '.i'.

Other arguments are taken to be either loader flag arguments, or C-compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name *a.out.*

9G. THE GCOS C COMPILER COMMAND

The commmand *./cc* compiles and loads C programs. To compile *x.c, y.c* and *z.c*, placing the result on a permanent random library *clib*, type

./cc x.c y.c z.c r=/clib

or to run *a.c* with *clib* and put the H* (executable version) on *prog*, type

./cc a.c clib x=prog

The command line arguments are separated by blanks. They are divided into three categories. Unlike the *cc* command on UNIX, the syntax of a file name is never meaningful; source programs need not end in '.c', for example. The types of arguments are:

1. random libraries: saved for loading.
2. sequential files: sent to the C compiler.
3. strings with '=' in them: command options.

Command options are (the text in parentheses is optional):

1. **r**(anlib)=name: merge file for compiler output; also used as a library for loading.
2. **h**(star)=name or just =name: time-sharing H* file to be created. The file can be run directly with the system command loader or the *go* command. If *name* is omitted, the default is *.program.*
3. **x**(ecute)=name: do an *h=name* and then run the program.
4. **f**(ortran,orty,ortrex)=name or **g**(map)=name: program written in Fortran or assembler to be included in library or H* file.
5. **b**(atchhstar)=name: batch H* file to be created. Such a file may not be executed in time-sharing.
6. **o**(ptions)=name: use *bfor* options on file *name.*
7. **m**(ap)=name: put assembly listings and load map on file *name.*
8. **u**(se)=name: program references needed for loading.
9. **e**(ntry)=name: replace normal starting point with *name.*
10. **k**(eep)=name: place the GMAP translation of the next source program being compiled on file *name*, which will be made a permanent file in your catalog.
11. **d**(ebug)=**no**: suppress the compilation of symbol tables and the loading of the debugger (see section 12).
12. **l**(imits)=string: make a $ LIMITS card for *bfor. String* should be something like ",26k".

If you give no options on your command line, but only filenames, the default is *r=.library.* Be careful with the order of libraries and options on the command line; they are used in the sequence given, and an options file, if any, is inserted at the place requested. Note that no default options file is used, unlike *bfor.* Loading involves the library *cc/c.t.lib* preceded, in batch only, by *cc/c.b.lib;* the normal entry point is ...... (the GCOS standard). The *bfor* libraries are also used.

9I. IBM COMPILING COMMANDS

Standard TSO command lists have been defined for C compilation and loading. They are installed in the standard system procedure library. Supposing that the program is on a file *prog.text*, to compile, say

cc prog

and to compile and execute say

```
ccg prog
```

The C run-time library is stored on the library *sys3.clib* for linking or loading. C requires the FORTRAN runtime library *sys1.fortlib* as well. On TSO it is not now possible to compile more than one source program at a time. The dialog to compile and run the programs p1.text, p2.text, and p3.text would appear as follows:

```
READY
cc p1
READY
cc p2
READY
cc p3
READY
loadgo (p1.obj p2.obj p3.obj) lib('sys3.clib', 'sys1.fortlib')
```

To use the C compiler on TSO, the user should specify a 250K region at logon.

In batch OS/370 it is not possible to run the C compiler, because the preprocessor command "include" implies dynamic file attaching, which is only supported in time-sharing under OS/370. The IBM C compiler is still under development, and it is likely that some of the restrictions in this paper will be removed. In particular, floating point and the associated library routines will be included, and the calling sequence will be altered for easier compatibility with other programming languages.

10. GCOS PSEUDO-SHELL

The GCOS program **./sh** provides the syntax of the UNIX command language on the GCOS operating system. It may be used conveniently with either C or BFOR programs. To simplify the distinction between batch and time-sharing, it handles object programs most conveniently in the form of random libraries. It will generate, as required, batch or time-sharing versions of such programs and execute them.

For example, suppose that the simple file copy routine discussed in section 1 is stored on random library *prog.r* by a command such as

```
./cc prog r=/prog.r
```

or equivalent. In that case, the following sequence would submit first a batch and then a time-sharing job to perform file copies:

```
SYSTEM ?./sh
=prog.r <data >newdata &
=prog.r <oldlist >newlist
```

./sh is an imitation of the UNIX shell for GCOS. For example, to release all files with names ending in ".g", type *./sh rele *.g.* To make an archive file of all files with names *s1.c, s2.c, s3.c*, etc., type *./sh ./arch c arfile s?.c* To execute commands on file junk, type *./sh <junk.* To load and run the random library *prog.r*, type *./sh prog.r.* To spawn a batch job to execute the same program, type *./sh prog.r &*

The input line is divided by spaces, semicolons, and commas. Each piece is examined for the characters ?, [, and *; if it contains one of them, it is replaced by the permanent quick-access file names it matches. The UNIX rules are used: ? matches any single character; * matches anything; [chars] matches any of the characters in chars; [a-d] matches any character between a and d inclusive. Thus, *a?* is replaced by all two-letter file names beginning with a; *a** is replaced by all file names beginning with a; **.g* is replaced by all file names ending in .g; *a[125-8]* is replaced by any of the file names *a1, a2, a5, a6, a7, a8* which exist, and so on. Backslash may be used to escape the special characters; for example,

*./sh bfor run ** libr=mylib,use=main*

If all you want is a list of names, you can use *./sh ./null pattern* to get a list of all file names matching *pattern.*

The result is reassembled into a GCOS command line, printed and executed. You need not worry about the details of command reassembly; the shell follows the command loader's requirements. These are: for the commands *list, rele, purge* or *remo* the first two items are separated by blank and the rest by semicolon; for the commands *bfor, edit, yrun* or *jrun* the first three items are separated by blank and the rest by semicolon; for the command *get* the first two items are separated by blank and the remainder by comma; for command *filsys* the first three items are separated by blank and the rest by comma; otherwise all items are separated by blanks. If the assembled line exceeds 80 characters it is not executed due to a deficiency in the command loader.

If the first item on the line to be executed is a random library, it is loaded and then executed. The loading is done by *bfor;* the libraries used are each input argument, until the first non-library, followed by *cc/c.t.lib,* and *bfor/syslib. Use* commands are given for ...... and *main.* This suffices for most C and *bfor* jobs. The output is put on temporary file *.pg* and executed with *go.* Remaining (i.e. non-library) command line arguments are fed to the resulting program. If the first item on the command line is a random library, and the last is '&' or '&' followed by a string in parentheses, a batch job is spawned to run the program. The batch job uses *cc/c.b.lib* in addition to *cc/c.t.lib,* and *bfor/batchlib* in place of *bfor/syslib,* and the command line is provided on filecode *cz.* If a string in parentheses followed the '&', it is used as an option string to *jrun;* thus the line

prog.r <input >output &(s1)

runs *prog.r* in batch with diverted input and output and at service grade 1. For the benefit of FORTRAN users, the shell recognizes such constructions as *5<file* or *6>file* and causes the *file* to be accessed with altname or filecode corresponding to the given number. This permits FORTRAN programs running with the shell to divert their output. Any number, of course, may be used with the shell, but GCOS FORTRAN only recognizes filecodes from 1 to 39.

If *./sh* has no arguments, it reads lines and executes each in turn. Since it is a C program, it can thus be used to execute a file by *./sh <filename.* Alternatively, it can be used in front of the GCOS command loader as a filter by just saying *./sh.* In this case, you type lines which are translated and then executed. To get out of the pseudo-shell, type a blank line.

If you have more than 150 quick-access files, only the 150 most recently created are used with *, ? and []. AFT names are not used; thus you may have trouble with temporary files or alternate names, neither of which are known to the shell. Because of the escapes, \ must sometimes be input as \\. The shell accepts command files with arguments as on UNIX.

11. MEASUREMENT FACILITIES

In general, these are derived from the operating system, and thus vary from machine to machine.

The standard UNIX profiler can be invoked by compiling your program with the *−p* flag of the compiler, e.g. "cc −p file.c". After running the resulting program, the *prof(I)* command will indicate the amount of time spent in each subroutine.

If you load the library *./hist.r* with your programs at the end of the run a table is printed showing the number of times every C subroutine was called. If you want to histogram parts of your program, you may call the entry *.phist* at any time; it will print the current counts and reset them to zero.

There are no specifically C-oriented measurement facilities on the OS system. The GCOS histogrammer is not yet installed.

12. DEBUGGING FACILITIES

On GCOS and UNIX there are symbolic post-mortem dump facilities for C programs. The UNIX debugger, written by D. M. Ritchie, is only outlined below; more detail is available in the *UNIX Programmer's Manual.* Errors in a running C program cause a brief message to be typed, followed by the option of entering the debugger.

When a C program faults, the core image is written to a file and the system returns to command level. The debugger can be entered by typing *cdb* (with optional arguments giving the names of the load module and core image files if non-standard).

The debugger is entered automatically after a fault; or, by calling *x_dbug()* it can be entered deliberately. In batch, the debugger is entered on fault, prints a symbolic dump, and exits.

No C debugging facilities have been implemented yet on OS/370.

Interactive examination of the values of variables is possible; in general, each input line to the debugger specifies data to be printed as of the time of the fault.

The debugger is used interactively. The first request should probably be $, which presents a list of executing subroutines and their arguments in octal.

On entrance, the debugger asks you to choose between interactive examination or a full dump typed at the terminal or directed to a file. The first step if interaction is chosen is to present the list of active routines.

The typical debugger command names a variable and asks for its value to be printed. The general format is

subroutine:variable/format

where *format* is one of the letters *i, c, f, d* or *s* to specify integer, character, floating, double, or string printout.

If the format is omitted octal integer is assumed.

If the format is omitted the type of the variable is used to decide on the format.

A command may also be given as

number/format

to examine a particular memory location by address. If the number begins with 0 it is taken as an octal address; otherwise it is a decimal address.

The slash is required.

The slash may be omitted. Non-word-aligned character data may be accessed with addresses of the form *number.d* where *d* is 0, 1, 2 or 3 to select the bytes of a word.

To find out the address of a variable, the format

subroutine:variable=

may be used. Other possible inquiries are

subroutine:variable"

(equivalent to *variable/s*) and

subroutine:variable'

(which is the same as *subroutine:variable/c).*

An address or variable name may be followed by *,number* which causes the indicated number of memory locations, beginning with the address or variable, to be printed. A format may follow.

Not implemented: the size of arrays is known from the symbol tables and when an array is named its contents are all printed.

On GCOS there is a concept of "current routine". The name of the subroutine need only be given when it changes, and the command *name:* is accepted to change it. The commands *#up* and *#down* move up and down the stack of active routines ("up" is towards the operating system, in the direction of returns; "down" is in the direction of calls).

Not implemented: subroutine names must be given for each internal variable.

All of the variables of a routine may be printed with the command *#print.*

To exit from the debugger, type an end-of-file at it.

On GCOS the command *#quit* also exits.

In general, the cost of the debugger is small.

Since the debugger operates only on a post-mortem core image, there is no cost associated with the running program.

The debugging routines require about 2500 additional words of memory, plus the cost of the symbol tables, which run three words per variable. The *d=no* option of *./cc* will suppress the debugger.

17. ACKNOWLEDGMENTS

Without the work of Dennis Ritchie, of course, this entire subject would be non-existent. In addition, Steve Johnson and Tom Peterson have written the GCOS and IBM TSO compilers respectively. Special thanks are due to Steve Johnson for his active interest in and contributions to the library and C run-time support on several machines. The assistance of Roger Faulkner with GCOS C is also gratefully acknowledged.

18. REFERENCES

1. D. M. Ritchie, *C Reference Manual.* Section 1, this report.

2. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual.* Available from Bell Laboratories under license agreement.

3. C. S. Roberts and R. A. Faulkner, *TSS GEFRC — File and Record Control Subroutines For Use with GECOS Time-Sharing,* (private communication) and N. P. Nelson, *BFOR,* (private communication) August 1973, describe the routines used to support the C runtime system.

4. J. F. Gimpel, *Sys2.Fortlib Newsletter,* (private communication) describes subroutines used for C run-time support.

Appendix 1
List of the C Library Routines

Name	Section
ABORT (code)	7
ATOF (char-string)	7
BACK ([fd])	8G
CALLOC (n, sizeof(object))	7
CCLOSE (fd)	4
CCLOSE (fd)	6
CEOF (fd)	4
CEXIT ([errcode])	4
CFLUSH (fn)	4
CFREE (ptr, n, sizeof(*ptr))	7
CGETC (fd)	4
CGETI (buff, len, fd)	8G
COPEN (filename, type)	4
COPEN (filename, type, options)	8G
COPEN (name, direction, options)	8I
COPEN (name, direction, "i")	6
CPUTC (char , fd)	4
CPUTI (buff, len, fd)	8G
CREAD (ptr, ptr+1, nitems, fd)	6
CWRITE (ptr, ptr+1, nitems, fd)	6
DRLDRL (number, arg1, arg2, ...)	8G
FTOA (floating-number, char-string , precision , format)	7
GENREG (regno, ndecl)	8I
GETCHAR ()	4
GETS (s)	4
INTSS ()	7
MMEMME (number, arg1, arg2, ...)	8G
NARGS ()	7
PRINTF ([fd,] control-string, arg1, arg2, ...)	5
PUTCHAR (ch)	4
PUTS (s)	4
REW (fd)	7
SCANF ([fd,] control-string, arg1, arg2, ...)	5
SYSTEM (string)	7
TMPNAM (str)	7
UNGETC (char , fd)	4
WDLENG ()	7
XNARGS (ptrs)	8G