DIGITAL RESEARCH™

C
Language
Programming Guide
for CP/M-68K™

Foreword

The C language under CP/M-68K™ is easy to read, easy to maintain, and highly portable. CP/M-68K can run most applications written in C for the UNIX® operating system, except programs that use the UNIX fork/exec multitasking primitives or that read UNIX file structures.

The C Language Programming Guide for CP/M-68K is not a tutorial. This manual describes how to program in C under the CP/M-68K operating system, and is best used by programmers familiar with the C language as described in *The C Programming Language* (Kernighan and Ritchie, 1978).

The commonly accepted standard for C language programming is the Portable C Compiler (PCC), written by Stephen C. Johnson. Many versions of the UNIX operating system use PCC, including the Zilog®, ONYX™, Xenix®, Berkeley UNIX, and UNIQ™ systems.

The CP/M-68K C compiler differs from PCC on the following points:

- The CP/M-68K C int (default) data type is 16 bits long. Pointers are 32 bits long. All function definitions and function calls that use long (32-bit ints) and pointer parameters must use the proper declarations.
- long, int, and char register variables are assigned to D registers. Five such registers are available in each procedure.
- Any register variable used as a pointer is assigned to an A register. There are three such registers available in each procedure.
- All local declarations in a function body must precede the first executable statement of the function.
- The CP/M-68K C compiler handles structure initialization as if the structure were an array of short integers, as in UNIX version 6.
- The first eight characters of variable and function names must be unique. The first seven characters of external names must be unique.
- The CP/M-68K C compiler does not support floating point.
- The CP/M-68K C compiler does not support structure assignment, structure arguments, and structures returned from procedures.
- The CP/M-68K C compiler does not support initialization of automatic variables.
- The CP/M-68K C compiler does not support enumeration types.

Section 1 of this manual describes the conventions of using C language under CP/M-68K. Section 2 discusses C language compatibility with UNIX version 7 and provides a dictionary of C library routines for CP/M-68K. Section 3 presents a style guide for coding C language programs.

Appendix A is a table of CP/M-68K error codes. Appendix B discusses compiler components, tells you how to operate the compiler, and suggests ways to conserve the disk space used for compiling. Appendix C presents sample C modules that are written and documented according to the style conventions outlined in Section 3.

Table of Contents

Table of Contents
(continued)

Table of Contents
(continued)

Appendixes

Tables and Figures

Tables

Figures

Section 1
Using C Language Under CP/M-68K

1.1 Compiling a CP/M-68K C Program

To create an executable C program under CP/M-68K, use the C.SUB and CLINK.SUB command files. The C.SUB file invokes the C compiler and the CLINK.SUB file invokes the linker. Use the following command line format to invoke the C compiler. Note that the command keyword SUBMIT is optional and that the source file must have a C filetype. You must not specify the C filetype in the compiler command line.

```
A>[SUBMIT] C filename
```

The compiler produces an object file with a O filetype. The linker uses the object file to create the executable program. Use the following command line format to invoke the linker. Again, the command keyword SUBMIT is optional. You must not specify the O filetype in the linker command line for the object file.

```
A>[SUBMIT] CLINK filename
```

You can specify multiple object files for linking into an executable program. For example, the first three command lines below compile source files named ONE.C, TWO.C, and THREE.C. The last command line links the three object files that the compiler creates into an executable program named ONE.68K

```
A>submit c one
A>submit c two
A>submit c three
A>submit clink one two three
```

To link C programs that use floating point math, substitute the CLINKF file for CLINK in the preceding example. CLINKF uses the Motorola FFP floating point format which is considered the fastest. To compile and link programs that use IEEE floating point format, substitute the CE file for C and the CLINKE file for CLINK in the preceding examples.

1.2 Memory Layout

The memory allocation of C programs running under CP/M-68K is similar to that of UNIX C programs. A program consists of three segments: the text segment or program instruction area, the data segment for initialized data, and the BSS or block storage segment for uninitialized data. There are two dynamic memory areas: the stack and the heap. Procedure calls and automatic variables use the stack. Data structures such as symbol tables use the heap. The brk, sbrk, malloc, and free C functions manage the heap. Figure 1-1 shows how each of the areas are arranged in memory.

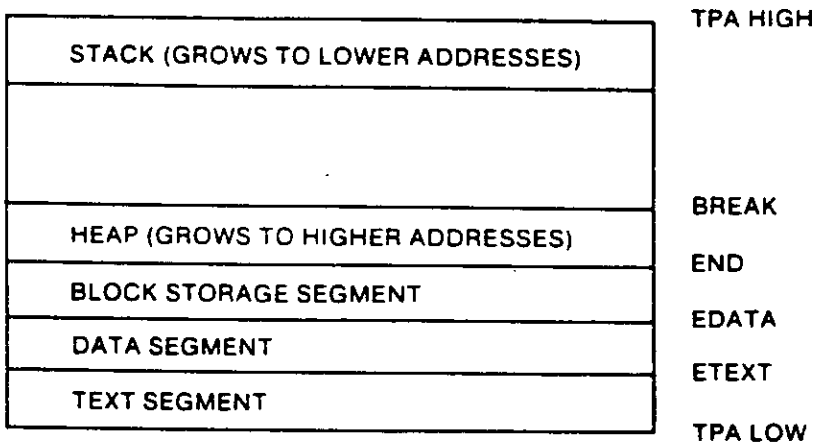

Figure 1-1. Memory Layout

The linker determines the locations etext, edata, and end. These locations are the ending addresses of the text, data, and BSS segments. The break location is the first unused address following the heap.

1.3 Calling Conventions

The JSR instruction (jump to subroutine) calls a C language procedure. Register A6 acts as the frame pointer to reference local storage. Arguments are pushed onto the A7 stack in reverse order. Word and character arguments occupy 16 bits. Long, floating point, and pointer arguments occupy 32 bits. All function values return in register D0. Functions that specify no return value actually return an undefined value.

For example, the following sequence

```
xyx() {
        long     a;
        int      b;
        char     x;
        register y;
            .
            .
        b = blivot(x,a);
}
```

generates the following codes:

```
_xyz:
        link     a6,#-8          * Space for a,b,x
        movem.l  d6-d7,-(27)     *d7 used for y
            .                    *d6 reserves space
            .
            .
        move.l   -4(a6),(a7)     * Load parameter a
        move.b   -8(a6),d0       * Load parameter x
        ext.w    d0              * Extend to word size
        move.w   d0,-(a7)        * Push it
        jsr      _blivot         * Call subroutine
        add.l    #2,a7           * Pop argument list
        move.w   d0,6(a6)        * Store return parameter
        tst.l    (a7)+           * Purge longword
        movem.l  (a7)+,d7        * Unsave registers
        unlk     a6              * Restore frame pointer
        rts                      * Return to caller
```

C code, in which all arguments are the same length, might not work without modification because of the varying length of arguments on the stack.

The compiler adds an underline character, _, to the beginning of each external variable or function name. This means that all external names in C must be unique in seven characters.

The compiler-generated code maintains a long word at the top of the stack for use in subroutine calls. This shortens the stack-popping code required on return from a procedure call. The movem.l instruction, which saves the registers, contains an extra register to allocate this space.

The compiler uses registers D3 through D7, and A3 through A5, for register variables. A procedure called from a C program must save and restore these registers, if they are used. The compiler-generated code saves only those registers used. Registers D0 through D2, and A0 through A2, are scratch registers and can be modified by the called procedure.

1.4 Stack Frame

Figure 1-2 illustrates the standard C stack frame.

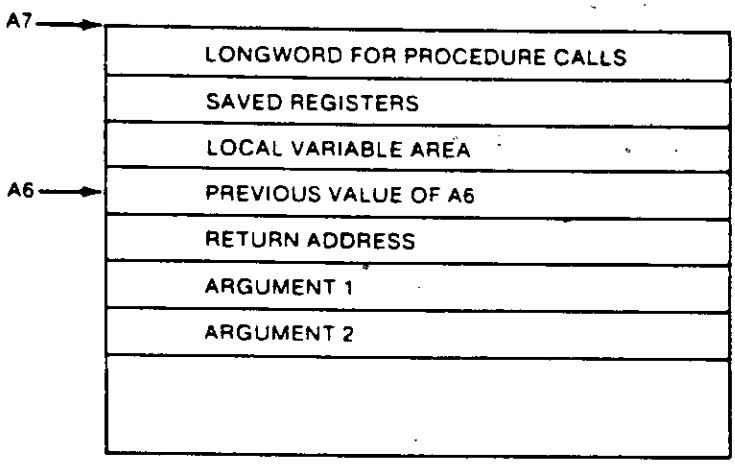

Figure 1-2. C Stack Frame

Arguments are either two or four bytes depending on the argument type. The compiler generated code uses register A6 to reference all variables on the stack.

1.5 Command Line Interface

The standard C argc/argv interface for arguments typed on the command line also works under CP/M-68K. For example, the command

```
command arg1 arg2 arg3 ... argn
```

produces the following interface setup:

```
argc     =     n+1
arg[0]         "C Runtime"
arg[1]         "arg1"
arg[2]         "arg2"
          .
          .
          .
argv[n]        argn
```

You cannot obtain the command name under CP/M-68K. Therefore, the argv[0] argument always contains the string "C Runtime".

Strings that contain the characters * or ? are interpreted as wildcarded filenames. The C runtime start-up routine scans the directory and expands each wildcarded filename into a list of filenames that match the specification. To pass a string that contains * or ? characters to a C program, enclose the string in single or double quotation marks. Similarly, enclose argument strings that contain embedded blanks in quotation marks to pass them to a C program as a single element of argv[].

1.6 I/O Conventions

UNIX C programs use two types of file and device I/O: regular and stream files. A unique number called the file descriptor identifies regular files. In CP/M-68K, file numbers range from 0 to 15. The address of a user control block in the run-time system identifies stream files. Unlike regular files, stream files use a form of intermediate buffering that makes single-byte I/O more efficient.

Under UNIX, you can reference peripheral devices, such as terminals and printers, as files using the special names /dev/tty for terminal and /dev/lp for printer. Under CP/M-68K, CON: is for the console device and LST: is for the listing device.

CP/M-68K stores ASCII files with a carriage return line feed after each line. A CTRL-Z (0x1a) character indicates end-of-file. C programs usually end lines with only a line feed. This means that in C for CP/M-68K, read and write operations to ASCII files must insert and delete carriage-return characters. The CTRL-Z must be deleted on read and inserted on close for such files. These operations are not desirable for binary files. CP/M-68K C includes an extra entry point to all file open and creat calls to distinguish between ASCII and binary files.

1.7 Standard Files

C programs begin execution with three files already open: the standard input, standard output, and standard error files. You can access these files as either stream or regular files in a C program. The usual C library routines close and reopen the standard files. The following definitions are in the <stdio.h> file.

Table 1-1. Standard File Definitions

File	File Descriptor	Stream Name
standard input	STDIN	stdin
standard output	STDOUT	stdout
standard error	STDERR	stderr

1.8 I/O Redirection

You can redirect C program standard I/O using the < and > characters. For example, the following command executes the file TEST.68K. The standard input comes from file DAT and the standard output goes to the listing device. The argument list is C, D, E, and F.

```
A>TEST <DAT >LST: C D E F
```

You cannot place spaces between the < or > characters and the filename that the character refers to. Note that you cannot redirect the standard error file.

You can append information to an existing file using the following specification:

```
>>filename
```

The standard output from the program specified by the filename appears after the original contents of the file.

End of Section 1

Section 2
C Language Library Routines

The CP/M-68K C library is a collection of routines for I/O, dynamic memory allocation, system traps, and data conversion.

2.1 Compatibility with UNIX V7

The C library is compatible with UNIX version 7, allowing programs to move easily from UNIX to CP/M-68K. CP/M-68K C simulates many UNIX operating system calls and features. However, CP/M-68K does not support the following C functions that UNIX implements:

- the fork/exec, kill, lock, nice, pause, ptrace, sync, and wait primitives
- the acct system call
- the alarm function, or the stime, time, ftime, and times system calls
- the dup and dup2 duplicate file descriptor functions
- the getuid, getgid, geteuid, getegid, setuid, and setgid functions
- the indir indirect system call
- the ioctl, stty, and gtty system calls
- the link system call
- the chdir, chroot, mknod, mount, umount, mpx, pipe, pkon, pkoff, profil, sync, stat, fstat, umask, and utime system calls
- the phys system call

The following UNIX library functions are not available under CP/M-68K:

- Assert
- Crypt
- DBM
- Getenv
- Getgrent, getlogin, getpw, and getpwent functions
- l3tol, ltol3
- monitor
- itom, madd, msub, mult, mdiv, min, mout, pow, gcd, and rpow
- nlist
- pkopen, pkclose, pkread, pkwrite, and pkfail
- plot
- popen, pclose
- sleep
- system
- ttyslot

The CP/M-68K C language library does not contain the floating-point routines available under UNIX.

Entry points have been added to file open and creat calls to distinguish between ASCII and binary files. Byte level end-of-file is unavailable for binary files. ASCII files, however, are compatible with UNIX, and with the CP/M-68K text editors and utilities that use ASCII files.

The C Programming Guide for CP/M-68K does not separate the UNIX system calls and library functions; all calls are library functions under CP/M-68K.

2.2 Library Functions under CP/M-68K

The remainder of this section alphabetically lists library routines that C supports under CP/M-68K. The C compiler accepts entry in upper- and lower-case; however, type all library routines in lower-case, as shown in the calling sequences.

abort Function

The abort function terminates the current program with an error. The error is system dependent. The 3000 uses an illegal instruction trap. This invokes DDT-68K™, if the debugger is loaded with the object program.

Calling Sequence:

```
WORD code;

abort(code);
```

Arguments:

code loads into register D0 before abort

Returns:

The abort function never returns.

abs Function

The abs function takes the absolute value of a single argument. This function is implemented as a macro in <stdio.h>; arguments with side effects do not work as you expect. For example, the call

```
a = abs(*x++);
```

increments x twice.

Calling Sequence:

```
WORD val;
WORD ret;

ret = abs(val);
```

Arguments:

val	the input value

Returns:

ret	the absolute value of val

access Function

The access function checks whether the calling program can access a specified file. Under CP/M-68K, the file is accessible if it exists.

Calling Sequence:

```
BYTE *name;
WORD  mode;
WORD  ret;

ret = access(name,mode);
```

Arguments:

name points to the null-terminated filename

mode can be one of four values:

4	checks read access
2	checks write access
1	checks execute access
0	checks directory path access

CP/M-68K ignores the 0 argument

Returns:

ret 0 if file access is allowed or -1 if not allowed

Note:

CP/M-68K only checks to see if the specified file exists.

atoi, atof, atol Functions

The atoi, atof, and atol functions convert an ASCII digit string to an integer, float, or long binary number, respectively. The atoi and atol functions convert digit strings of the form [-][+]dddddd... The atof function converts digit strings of the form [-][+]ddddd.ddd[e[-]dd]. Each "d" is a decimal digit. The compiler ignores all leading spaces, but permits a leading sign. Conversion proceeds until the number of digits in the string is exhausted. Each function returns a 0 when there are no more digits to convert.

Calling Sequence:

```
BYTE    *string;
WORD    ival,atoi();
LONG    lval,atol();
FLOAT   fval,atof();

ival = atoi(string);
lval = atol(string);
fval  = atof(string);
```

Arguments:

string	a pointer to a null-terminated string that contains the number to convert

Returns:

ival	atoi returns the converted string as an integer
lval	atol returns the converted string as a long binary number
fval	atof returns the converted string as a single-precision floating-point number

Note:

The atoi, atol, and atof functions do not detect or report overflow. Therefore, you cannot specify a limit to the number of contiguous digits processed or determine the number of digits a function processes.

brk, sbrk Functions

The brk and sbrk functions extend the heap portion of the user program. The brk function sets the upper bound of the program, called the break in UNIX terminology, to an absolute address. The sbrk function extends the program by an incremental amount.

Calling Sequence:

```
WORD brk();
BYTE *addr,*sbrk();
WORD ret;
BYTE *start;

ret =  brk(addr);
start = sbrk(incr);
```

Arguments:

addr	the desired new break address
incr	the incremental number of bytes desired

Returns:

0	success (brk)
-1	failure (brk)
start	begins the allocated area (sbrk)
0	failure (sbrk)

calloc, malloc, realloc, free Functions

The calloc, malloc, realloc, and free functions manage the dynamic area between the region and the stack.

The malloc function allocates an area of contiguous bytes aligned on a word boundary and returns the address of this area. Malloc uses the sbrk function to allocate additional heap space, if necessary.

The calloc function allocates space for an array of elements, whose size is given in bytes.

The realloc function changes the size of a block. The address of the block returns.

The free function releases a block previously allocated by malloc.

Calling Sequence:

```
WORD size,number;
BYTE *addr,*malloc(),*calloc(),*realloc();

addr = malloc(size);
addr = calloc(number,size);
addr = realloc(addr,size);
free(addr);
```

Arguments:

size	the number of bytes desired
number	the number of elements desired
addr	points to the allocated region

Returns:

Address of the allocated region if successful, 0 if unsuccessful.

Note:

Freeing a bogus address can be disastrous.

ceil Function

The ceil function returns the smallest integer that is greater than the argument you specify. For example, ceil(1.5) returns 2.0. The return value is a floating-point number.

Calling Sequence:

```
FLOAT  ceil();
FLOAT  arg;
FLOAT  ret;

ret = ceil(arg);
```

Arguments:

arg	a floating-point number

Returns:

ret	a floating-point number

chmod, chown Functions

Under UNIX, the chmod and chown system calls allow you to change the protection and owner ID of an existing file. CP/M-68K treats these calls as NO-OPS if the file exists.

Calling Sequence:

```
BYTE *name;
WORD mode,owner,group,ret;

ret = chmod(name,mode);
ret = chown(name,owner,group);
```

Arguments:

name	the affected filename (null-terminated)
mode	the new mode for the file
owner	the new owner of the file
group	the new group number

Returns:

ret	0 if the file exists
	-1 if the file does not exist

close Function

The close function terminates access to a file or device. This routine acts on files opened with the open or creat functions. Specify a file descriptor, not a stream, for the operation. The fclose function closes stream files.

Calling Sequence:

```
WORD fd,ret;

ret = close(fd);
```

Arguments:

fd	the file descriptor to be closed

Returns:

0	successful close
-1	unknown file descriptor

cos, sin Functions

The cos function returns the trigonometric cosine of a floating-point number. The sin function returns the trigonometric sine of a floating-point number. You must express all arguments in radians.

Calling Sequence:

```
FLOAT  cos(),sin();
FLOAT  val,ret;

ret = cos(val);
ret = sin(val);
```

Arguments:

val	a floating-point number that expresses an angle in radians

Returns:

ret	the cosine or sine of the argument value expressed in radians

Note:

The best results occur with arguments that are less than 2 pi. You can pass numbers declared as either float or double to cos and sin.

creat, creata, creatb Functions

The creat function adds a new file to a disk directory. The file can then be referenced by the file descriptor, but not as a stream file. The creat and creata functions create an ASCII file. The creatb function creates a binary file.

Calling Sequence:

```
BYTE *name;
WORD mode,fd;

fd = creat(name,mode);
fd = creata(name,mode);
fd = creatb(name,mode);
```

Arguments:

name	the filename string, null-terminated
mode	the UNIX file mode, ignored by CP/M-68K

Returns:

fd	The file descriptor for the opened file. A file descriptor is an int quantity that denotes an open file in a read, write, or lseek call.
-1	Returned if there are any errors.

Note:

UNIX programs that use binary files compile successfully, but execute improperly.

ctype Functions

The file <ctype.h> defines a number of functions that classify ASCII characters. These functions indicate whether a character belongs to a certain character class, returning nonzero for true and zero for false. The following table defines ctype functions.

Table 2-1. ctype Functions

Function	Meaning
isalpha(c)	c is a letter.
isupper(c)	c is upper-case.
islower(c)	c is lower-case.
isdigit(c)	c is a digit.
isalnum(c)	c is alphanumeric.
isspace(c)	c is a white space character.
ispunct(c)	c is a punctuation character.
isprint(c)	c is a printable character.
iscntrl(c)	c is a control character.
isascii(c)	c is an ASCII character (< 0x80).

The white space characters are the space (0x20), tab (0x09), carriage return (0x0d), line-feed (0x0a), and form-feed (0x0c) characters. Punctuation characters are not control or alphanumeric characters. The printing characters are the space (0x20) through the tilde (0x7e). A control character is less than a space (0x20).

Calling Sequence:

```
#include <ctype.h>

WORD ret;
BYTE c; /* or WORD c; */

ret = isalpha(c);
ret = isupper(c);
ret = islower(c);
ret = isdigit(c);
ret = isalnum(c);
ret = isspace(c);
ret = ispunct(c);
ret = isprint(c);
ret = iscntrl(c);
ret = isascii(c);
```

Arguments:

c	the character to be classified

Returns:

ret	= 0 for false
ret	<>0 for true

Note:

These functions are implemented as macros; arguments with side effects, such as *p++, work incorrectly in some cases. Bogus values return if arguments are not ASCII characters. For example, >0x7f.

end, etext, edata Locations

The linkage editor defines the labels end, etext, and edata as the first location past the BSS, text, and data regions, respectively. The program-break location, which is the last used location, is initially set to end. However, many library functions alter this location. sbrk(0) can retrieve the break.

etoa, ftoa Functions

The etoa and ftoa functions convert a floating-point number to an ASCII string. Both functions return the address of the converted string buffer. The string returned in the buffer takes the form [-]d.ddddde[-]dd. Each "d" is a decimal digit.

Calling Sequence:

```
FLOAT  fval;
BYTE   *ftoa(),*etoa(),*buf,*ret;
WORD   prec;

ret = etoa(fval,buf,prec);
ret = ftoa(fval,buf,prec);
```

Arguments:

fval	the floating point number to be converted
buf	the address of the buffer for the digit string
prec	the number of digits to appear to the right of the decimal point in the converted string

Returns:

ret	the address of the buffer for the converted, null-terminated string

exit, _exit Functions

The exit function passes control to CP/M-68K. An optional completion code, which CP/M-68K ignores, might return. exit deallocates all memory and closes any open files. exit also flushes the buffer for stream output files.

The _exit function immediately returns control to CP/M-68K, without flushing or closing open files.

Calling Sequence:

```
WORD code;

exit(code);
_exit(code);
```

Arguments:

code optional return code

Returns:

no returns

exp Function

The exp function returns the constant e raised to a specified exponent. The constant e is the base of natural logarithims equal to 2.71828182845905.

Calling Sequence:

```
FLOAT  exp();
FLOAT  fval,ret;

ret = exp(fval);
```

Arguments:

fval the exponent expressed as a floating-point number

Returns:

ret the value of e raised to the specified exponent

Note:

You can pass numbers declared as either float or double to exp.

fabs Function

The fabs function returns the absolute value of a floating-point number.

Calling Sequence:

```
FLOAT  fabs();
FLOAT  fval;
FLOAT  retval;

retval = fabs(fval);
```

Arguments:

fval a floating point number

Returns:

retval the absolute value of the floating-point number

fclose, fflush Functions

The fclose and fflush functions close and flush stream files. The stream address identifies the stream to be closed.

Calling Sequence:

```
WORD ret;
FILE *stream;

ret = fclose(stream);
ret = fflush(stream);
```

Arguments:

stream	the stream address

Returns:

0	successful
-1	bad stream address or write failure

feof, ferror, clearerr, fileno Functions

These functions manipulate file streams in a system-independent manner.

The feof function returns nonzero if a specified stream is at end-of-file, and zero if it is not.

The ferror function returns nonzero when an error has occurred on a specified stream. The clearerr function clears this error. This is useful for functions such as putw, where no error indication returns for output failures.

The fileno function returns the file descriptor associated with an open stream.

Calling Sequence:

```
WORD ret;
FILE *stream;
WORD fd;

ret = feof(stream);
ret = ferror(stream);
clearerr(stream);
fd = fileno(stream);
```

Arguments:

stream	the stream address

Returns:

ret	a zero or nonzero indicator
fd	the returned file descriptor

floor Function

The floor function returns the largest integer that is less than the argument you specify. The returned value is a floating-point number. For example, floor(1.5) returns 1.0.

Calling Sequence:

```
FLOAT  floor();
FLOAT  fval;
FLOAT  retval;

retval = floor(fval);
```

Arguments:

fval	a floating-point number

Returns:

retval	a floating-point integer value

fmod Function

The fmod function returns the floating-point modulus (remainder) from a division of two arguments. fmod divides the first argument by the second and returns the remainder.

Calling Sequence:

```
FLOAT  fmod();
FLOAT  x,y;
FLOAT  ret;

ret = fmod(x,y);
```

Arguments:

x	a floating-point dividend
y	a floating-point divisor

Returns:

ret	the modulus as a floating-point number

fopen, freopen, fdopen Functions

The fopen, freopen, and fdopen functions associate an I/O stream with a file or device.

The fopen and fopena functions open an existing ASCII file for I/O as a stream. The fopenb function opens an existing binary file for I/O as a stream.

The freopen and freopa functions substitute a new ASCII file for an open stream. The freopb function substitutes a new binary file for an open stream.

The fdopen function associates a file that file descriptor opened, using open or creat, with a stream.

Calling Sequence:

```
FILE *fopen(),fopena(),fopenb();
FILE *freopen(),freopa(),freopb();
FILE *fdopen();
FILE *stream;
BYTE *name,*access;
WORD fd;

stream = fopen(name,access);
stream = fopena(name,access);
stream = fopenb(name,access);
stream = freopen(name,access,stream);
stream = freopa(name,access,stream);
stream = freopb(name,access,stream);
stream = fdopen(fd,access);
```

Arguments:

name	the null-terminated filename string
stream	the stream address
access	the access string:

r	read the file
w	write the file
a	append to a file

Returns:

stream	successful if stream address open
0	unsuccessful

Note:

UNIX programs that use fopen on binary files compile and link correctly, but execute improperly.

fread, fwrite Functions

The fread and fwrite functions transfer a stream of bytes between a stream file and primary memory.

Calling Sequence:

```
WORD nitems;
BYTE *buff;
WORD size;
FILE *stream;

nitems =  fread(buff,size,nitems,stream);
nitems = fwrite(buff,size,nitems,stream);
```

Arguments:

buff	the primary memory buffer address
size	the number of bytes in each item
nitems	the number of items to transfer
stream	an open stream file

Returns:

nitems	the number of items read or written
0	error, including EOF

fseek, ftell, rewind Functions

The fseek, ftell, and rewind functions position a stream file.

The fseek function sets the read or write pointer to an arbitrary offset in the stream. The rewind function sets the read or write pointer to the beginning of the stream. These calls have no effect on the console device or the listing device.

The ftell function returns the present value of the read or write pointer in the stream. This call returns a meaningless value for nonfile devices.

Calling Sequence:

```
WORD ret;
FILE *stream;
LONG offset,ftell();
WORD ptrname;

ret = fseek(stream,offset,ptrname);
ret = rewind(stream);
offset = ftell(stream);
```

Arguments:

stream	the stream address
offset	a signed offset measured in bytes
ptrname	the interpretation of offset:
	0 => from beginning of file
	1 => from current position
	2 => from end of file

Returns:

ret	0 for success, -1 for failure
offset	present offset in stream

Note:

ASCII file seek and tell operations do not account for carriage returns that are eventually deleted. CTRL-Z characters at the end of the file are correctly handled.

getc, getchar, fgetc, getw, getl Functions

The getc, getchar, fgetc, getw, and getl functions perform input from a stream.

The getc function reads a single character from an input stream. This function is implemented as a macro in <stdio.h>, and arguments should not have side effects.

The getchar function reads a single character from the standard input. It is identical to getc(stdin) in all respects.

The fgetc function is a function implementation of getc, used to reduce object code size.

The getw function reads a 16-bit word from the stream, high byte first. This is compatible with the read function call. No special alignment is required.

The getl function reads a 32-bit long from the stream, in 68000 byte order. No special alignment is required.

Calling Sequence:

```
WORD ichar;
FILE *stream;
WORD iword;
LONG ilong,getl();

ichar = getc(stream);
ichar = getchar();
ichar = fgetc(stream);
iword = getw(stream);
ilong = getl(stream);
```

Arguments:

stream	the stream address

Returns:

ichar	character read from stream
iword	word read from stream
ilong	longword read from stream
-1	on read failures

Note:

Error return from getchar is incompatible with UNIX prior to version 7. Error return from getl or getw is a valid value that might occur in the file normally. Use feof or ferror to detect end-of-file or read errors.

getpass Function

The getpass function reads a password from the console device. A prompt is output, and the input read without echoing to the console. A pointer returns to a 0- to 8-character null-terminated string.

Calling Sequence:

```
BYTE *prompt;
BYTE *getpass;
BYTE *pass;

pass = getpass(prompt);
```

Arguments:

prompt a null-terminated prompt string

Returns:

pass points to the password read

Note:

The return value points to static data whose content is overwritten by each call.

getpid Function

The getpid function is a bogus routine that returns a false process ID. This routine is strictly for UNIX compatibility; serves no purpose under CP/M-68K. The return value is unpredictable in some implementations.

Calling Sequence:

```
WORD pid;

pid = getpid();
```

Arguments:

no arguments.

Returns:

pid false process ID

gets, fgets Functions

The gets and fgets functions read strings from stream files. fgets reads a string including a newline (line-feed) character. gets deletes the newline, and reads only from the standard input. Both functions terminate the strings with a null character.

You must specify a maximum count with fgets, but not with gets. This count includes the terminating null character.

Calling Sequence:

```
BYTE *addr;
BYTE *s;
BYTE *gets(),*fgets();
WORD n;
FILE *stream;

addr = gets(s);
addr = fgets(s,n,stream);
```

Arguments:

s	the string buffer area address
n	the maximum character count
stream	the input stream

Returns:

addr	the string buffer address

index, rindex Functions

The index and rindex functions locate a given character in a string. index returns a pointer to the first occurrence of the character. rindex returns a pointer to the last occurrence.

Calling Sequence:

```
BYTE c;
BYTE *s;
BYTE *ptr;
BYTE *index(),*rindex();

ptr = index(s,c);
ptr = rindex(s,c);
```

Arguments:

s	a null-terminated string pointer
c	the character for which to look

Returns:

ptr	the desired character address
0	character not in the string

isatty Function

A CP/M-68K program can use the isatty function to determine whether a file descriptor is attached to the CP/M-68K console device (CON:).

Calling Sequence:

```
WORD fd;
WORD ret;

ret = isatty(fd);
```

Arguments:

fd	an open file descriptor

Returns:

1	fd attached to CON:
0	fd not attached to CON:

log Function

The log function returns the natural logarithm (log base e) of a floating-point number.

Calling Sequence:

```
FLOAT  log();
FLOAT  fval,ret;

ret = log(fval);
```

Arguments:

fval	a floating-point number

Returns:

ret	the natural logarithim of the floating-point number

Note:

You can pass numbers declared as either float or double to log.

lseek, tell Functions

The lseek function positions a file referenced by the file descriptor to an arbitrary offset. Do not use this function with stream files, because the data in the stream buffer might be invalid. Use the fseek function instead.

The tell function determines the file offset of an open file descriptor.

Calling Sequence:

```
WORD fd;
WORD ptrname;
LONG offset,lseek(),tell(),ret;

ret = lseek(fd,offset,ptrname);
ret = tell (fd);
```

Arguments:

fd	the open file descriptor
offset	a signed byte offset in the file
ptrname	the interpretation of offset:
	0 => from the beginning of the file
	1 => from the current file position
	2 => from the end of the file

Returns:

ret	resulting absolute file offset
-1	error

Note:

Incompatible with versions 1 through 6 of UNIX.

mktemp Function

The mktemp function creates a temporary filename. The calling argument is a character string ending in 6 X characters. The temporary filename overwrites these characters.

Calling Sequence:

```
BYTE *string;
BYTE *mktemp();

string = mktemp(string);
```

Arguments:

string the address of the template string

Returns:

string the original address argument

open, opena, openb Functions

The open and opena functions open an existing ASCII file by file descriptor. The openb function opens an existing binary file. The file can be opened for reading, writing, or updating.

Calling Sequence:

```
BYTE *name;
WORD mode;
WORD fd;

fd = open(name,mode);
fd = opena(name,mode);
fd = openb(name,mode);
```

Arguments:

name	the null-terminated filename string
mode	the access desired:
	0 => Read-Only
	1 => Write-Only
	2 => Read-Write (update)

Returns:

fd	the file descriptor for accessing the file
-1	open failure

Note:

UNIX programs that use binary files compile correctly, but execute improperly.

perror Function

The perror function writes a short message on the standard error file that describes the last system error encountered. First an argument string prints, then a colon, then the message.

CP/M-68K C simulates the UNIX notion of an external variable, errno, that contains the last error returned from the operating system. Appendix A contains a list of the possible values of errno and of the messages that perror prints.

Calling Sequence:

```
BYTE *s;
WORD err;
err = perror(s);
```

Arguments:

s	the prefix string to be printed

Returns:

err	value of "ERRNO" before call

Note:

Many messages are undefined on CP/M-68K.

pow Function

The pow function returns the value of a number raised to a specified power; pow uses two floating-point arguments. The first argument is the mantissa and the second argument is the exponent.

Calling Sequence:

```
FLOAT  pow();
FLOAT  x,y;
FLOAT  ret;

ret = pow(x,_);
```

Arguments:

x a floating-point mantissa

y a floating-point exponent

Returns:

ret the value of the mantissa raised to the exponent

printf, fprintf, sprintf Functions

The printf functions format data for output. The printf function outputs to the standard output stream. The fprintf function outputs to an arbitrary stream file. The sprintf function outputs to a string (memory).

Calling Sequence:

```
WORD ret;
BYTE *fmt;
FILE *stream;
BYTE *string;
BYTE *sprintf(),rs;
/* Args can be any type */

ret    = printf (fmt,arg1,arg2 ...);
ret    = fprintf(stream,fmt,arg1,arg2 ...);
rs     = sprintf(string,fmt,arg1,arg2 ...);
```

Arguments:

fmt	format string with conversion specifiers
argn	data arguments to be converted
stream	output stream file
string	buffer address

Returns:

ret	number of characters output -1 if error
rs	buffer string address null if error

Conversion Operators

A percent sign, %, in the format string indicates the start of a conversion operator. Values to be converted come in order from the argument list. Table 2-2 defines the valid conversion operators.

Table 2-2. Conversion Operators

Operator	Meaning
d	Converts a binary number to decimal ASCII and inserts in output stream.
o	Converts a binary number to octal ASCII and inserts in output stream.
x	Converts a binary number to hexadecimal ASCII and inserts in output stream.
c	Uses the argument as a single ASCII character.
s	Uses the argument as a pointer to a null-terminated ASCII string, and inserts the string into the output stream.
u	Converts an unsigned binary number to decimal ASCII and inserts in output stream.
%	Prints a % character.

You can insert the following optional directions between the % character and the conversion operator:

- A minus sign justifies the converted output to the left, instead of the default right justification.
- A digit string specifies a field width. This value gives the minimum width of the field. If the digit string begins with a 0 character, zero padding results instead of blank padding. An asterisk takes the value of the width field as the next argument in the argument list.
- A period separates the field width from the precision string.
- A digit string specifies the precision for floating-point conversion, which is the number of digits following the decimal point. An asterisk takes the value of the precision field from the next argument in the argument list.
- The character l or L specifies that a 32-bit long value be converted. A capitalized conversion code does the same thing.

putc, putchar, fputc, putw, putl Functions

The putc, putchar, fputc, putw, and putl functions output characters and words to stream files.

The putc function outputs a single 8-bit character to a stream file. This function is implemented as a macro in <stdio.h>, so do not use arguments with side effects. The fputc function provides the equivalent function as a real function.

The putchar function outputs a character to the standard output stream file. This function is also implemented as a macro in <stdio.h>. Avoid using side effects with putchar.

The putw function outputs a 16-bit word to the specified stream file. The word is output high byte first, compatible with the write function call.

The putl function outputs a 32-bit longword to the stream file. The bytes are output in 68000 order, as with the write function call.

Calling Sequence:

```
BYTE c;
FILE *stream;
WORD w,ret;
LONG lret,putl(),l;

ret  = putc(c,stream);
ret  = fputc(c,stream);
ret  = putchar(c);
ret  = putw(w,stream);
lret = putl(l,stream);
```

Arguments:

c	the character to be output
stream	the output stream address
w	the word to be output
l	the long to be output

Returns:

ret	the word or character output
lret	the long output with putl
-1	an output error

Note:

A -1 return from putw or putl is a valid integer or long value. Use ferror to detect write errors.

puts, fputs Functions

The puts and fputs functions output a null-terminated string to an output stream.

The puts function outputs the string to the standard output, and appends a newline character.

The fputs function outputs the string to a named output stream. The fputs function does not append a newline character.

Neither routine copies the trailing null to the output stream.

Calling Sequence:

```
WORD ret;
BYTE *s;
FILE *stream;

ret = puts(s);
ret = fputs(s,stream);
```

Arguments:

s	the string to be output
stream	the output stream

Returns:

ret	the last character output
-1	error

Note:

The newline incompatibility is required for compatibility with UNIX.

qsort Function

The qsort function is a quick sort routine. You supply a vector of elements and a function to compare two elements, and the vector returns sorted.

Calling Sequence:

```
WORD ret;
BYTE *base;
WORD number;
WORD size;
WORD compare();

ret = qsort(base,number,size,compare);
```

Arguments:

base	the base address of the element vector
number	the number of elements to sort
size	size of each element in bytes
compare	the address of the comparison function

This function is called by the following:

```
ret = compare(a,b);
```

The return is:

```
< 0  if a < b
= 0  if a = b
> 0  if a > b
```

Returns:

0	always

rand, srand Functions

The rand and srand functions constitute the C language random number generator. Call srand with the seed to initialize the generator. Call rand to retrieve random numbers. The random numbers are C int quantities.

Calling Sequence:

```
WORD seed;
WORD rnum;

rnum = srand(seed);
rnum = rand();
```

Arguments:

seed	an int random number seed

Returns:

rnum	desired random number

read Function

The read function reads data from a file opened by the file descriptor using open or creat. You can read any number of bytes, starting at the current file pointer.

Under CP/M-68K, the most efficient reads begin and end on 128-byte boundaries.

Calling Sequence:

```
WORD ret;
WORD fd;
BYTE *buffer;
WORD bytes;

ret = read(fd,buffer,bytes);
```

Arguments:

fd	a file descriptor open for read
buffer	the buffer address
bytes	the number of bytes to be read

Returns:

ret	number of bytes actually read
-1	error

scanf, fscanf, sscanf Functions

The scanf functions convert input format. The scanf function reads from the standard input, fscanf reads from an open stream file, and sscanf reads from a null-terminated string.

Calling Sequence:

```
BYTE *format,*string;
WORD nitems;
FILE *stream;
/* Args can be any type */

nitems = scanf(format,arg1,arg2 ...);
nitems = fscanf(stream,format,arg1,arg2 ...);
nitems = sscanf(string,format,arg1,arg2 ...);
```

Arguments:

format	the control string
argn	pointers to converted data locations
stream	an open input stream file
string	null-terminated input string

Returns:

nitems	the number of items converted
-1	I/O error

Control String Format

The control string consists of the following items:

- Blanks, tabs, or newlines (line feeds) that match optional white space in the input.
- An ASCII character (not %) that matches the next character of the input stream.
- Conversion specifications, consisting of a leading %, an optional * (which suppresses assignment), and a conversion character. The next input field is converted and assigned to the next argument, up to the next inappropriate character in the input or until the field width is exhausted.

Conversion characters indicate the interpretation of the next input field. The following table defines valid conversion characters.

Table 2-3. Valid Conversion Characters

Character	Meaning
%	A single % matches in the input at this point; no conversion is performed.
d	Converts a decimal ASCII integer and stores it where the next argument points.
o	Converts an octal ASCII integer.
x	Converts a hexadecimal ASCII integer.
s	A character string, ending with a space, is input. The argument pointer is assumed to point to a character array big enough to contain the string and a trailing null character, which are added.
c	Stores a single ASCII character, including spaces. To find the next nonblank character, use %1s.
[	Stores a string that does not end with spaces. The character string is enclosed in brackets. If the first character after the left bracket is not ^, the input is read until the scan comes to the first character not within the brackets. If the first character after the left bracket is ^, the input is read until the first character within the brackets.

Note:

You cannot determine the success of literal matches and suppressed assignments.

setjmp, longjmp Functions

The setjmp and longjmp functions execute a nonlocal GOTO. The setjmp function initially specifies a return location. You can then call longjmp from the procedure that invoked setjmp, or any subsequent procedure. longjmp simulates a return from setjmp in the procedure that originally invoked setjmp. A setjmp return value passes from the longjmp call. The procedure invoking setjmp must not return before longjmp is called.

Calling Sequence:

```
#include <setjmp.h>
WORD      xret,ret;
jmp_buf   env;
          .
          .
          .
xret = setjmp(env);
          .
          .
          .
longjmp(env,ret);
```

Arguments:

env	contains the saved environment
ret	the desired return value from setjmp

Returns:

xret	0 when setjmp invoked initially copied from ret when longjmp called

Note:

awkward

signal Function

The signal function connects a C function with a 68000 exception condition. Each possible exception condition is indicated by a number. The following table defines exception conditions.

Table 2-4. 68000 Exception Conditions

Number	Condition
4	Illegal instruction trap. Includes illegal instructions, privilege violation, and line A and line F traps.
5	Trace trap.
6	Trap instruction other than 2 or 3: used by BDOS and BIOS.
8	Arithmetic traps: zero divide, CHK instruction, and TRAPV instruction.
10	BUSERR (nonexistent memory) or addressing (boundary) error trap.

All other values are ignored for compatibility with UNIX.

Returning from the procedure activated by the signal resumes normal processing. The library routines preserve registers and condition codes.

Calling Sequence:

```
WORD ret,sig;
WORD func();

ret = signal(sig,func);
```

Arguments:

sig	the signal number given above
func	the address of a C function

Returns:

ret	0 if no error, -1 if sig out of range

sinh, tanh Function

The sinh function returns the trigonometric hyperbolic sine of a floating-point number. The tanh function returns the trigonometric hyperbolic tangent of a floating-point number. You must express all arguments in radians.

Calling Sequence:

```
FLOAT  sinh(),tanh();
FLOAT  fval,ret;

ret = sinh(fval);
ret = tanh(fval);
```

Arguments:

fval	a floating-point number that expresses an angle in radians

Returns:

ret	the hyperbolic sine or hyperbolic tangent of the argument value expressed in radians

Note:

You can pass numbers declared as either float or double to sinh and tanh.

sqrt Function

The sqrt function returns the square root of a floating-point number.

Calling Sequence:

```
FLOAT  sqrt();
FLOAT  fval,ret;

ret = sqrt(fval);
```

Arguments:

fval	a floating-point number

Returns:

ret	the square root of the specified argument

Note:

You can pass numbers declared as either float or double to sqrt.

strcat, strncat Functions

The strcat and strncat functions concatenate strings. The strcat function concatenates two null-terminated strings. The strncat function copies a specified number of characters.

Calling Sequence:

```
BYTE *s1,*s2,*ret;
BYTE *strcat(),*strncat();
WORD n;

ret = strcat(s1,s2);
ret = strncat(s1,s2,n);
```

Arguments:

s1	the first string
s2	the second string, appended to s1
n	the maximum number of characters in s1

Returns:

ret	a pointer to s1

Note:

The strcat (s1,s1) function never terminates and usually destroys the operating system because the end-of-string marker is lost, so strcat continues until it runs out of memory, including the memory occupied by the operating system.

strcmp, strncmp Functions

The strcmp and strncmp functions compare strings. The strcmp function uses null termination, and strncmp limits the comparison to a specified number of characters.

Calling Sequence:

```
BYTE *s1,*s2;
WORD val,n;

val =  strcmp(s1,s2);
val = strncmp(s1,s2,n);
```

Arguments:

s1	a null-terminated string address
s2	a null-terminated string address
n	the maximum number of characters to compare

Returns:

val — the comparison result:

```
< 0 => s1 < s2
= 0 => s1 = s2
> 0 => s1 > s2
```

Note:

Different machines and compilers interpret the characters as signed or unsigned.

strcpy, strncpy Functions

The strcpy and strncpy functions copy one null-terminated string to another. The strcpy function uses null-termination, while strncpy imposes a maximum count on the copied string.

Calling Sequence:

```
BYTE *s1,*s2,*ret;
BYTE *strcpy(),*strncpy();
WORD n;

ret = strcpy(s1,s2);
ret = strncpy(s1,s2,n);
```

Arguments:

s1	the destination string
s2	the source string
n	the maximum character count

Returns:

ret	the address of s1

Note:

If the count is exceeded in strncpy, the destination string is not null-terminated.

strlen Function

The strlen function returns the length of a null-terminated string.

Calling Sequence:

```
BYTE *s;
WORD len;

len = strlen(s);
```

Arguments:

s	the string address

Returns:

len	the string length

swab Function

The swab function copies one area of memory to another. The high and low bytes in the destination copy are reversed. You can use this function to copy binary data from a PDP-11™ or VAX™ to the 68000. The number of bytes to swap must be even.

Calling Sequence:

```
WORD ret;
BYTE *from,*to;
WORD nbytes;

ret = swab(from,to,nbytes);
```

Arguments:

from	the address of the source buffer
to	the address of the destination
nbytes	the number of bytes to copy

Returns:

ret	always 0