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.

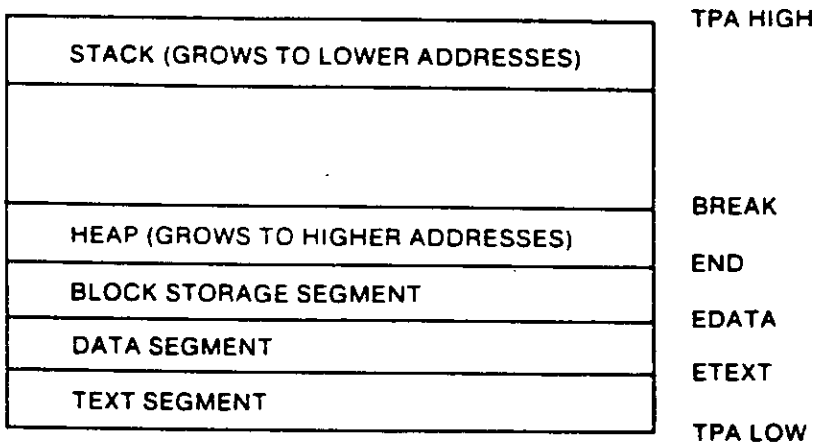

Figure 1-1. Memory Layout

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.

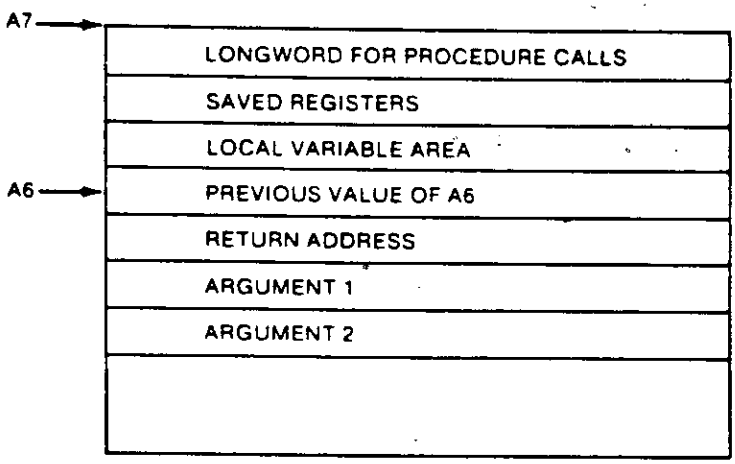

Figure 1-2. C Stack Frame

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

tan, atan Functions

The tan function returns the trigonometric tangent of a floating-point number. The atan function returns the trigonometric arctangent of a floating-point number. You must express arguments to tan in radians.

Calling Sequence:

```
FLOAT  tan(),atan();
FLOAT  val,rval,ret;

ret = tan(rval);
ret = atan(val);
```

Arguments:

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

val a floating-point number

Returns:

ret the tangent or arctangent of the argument value expressed in radians

Note:

The best precision results with arguments that are less than two pi. You can pass numbers declared as either float or double to tan and atan.

ttyname Function

The ttyname function returns a pointer to the null-terminated filename of the terminal device associated with an open file descriptor.

Calling Sequence:

```
BYTE *name,*ttyname();
WORD fd;

name = ttyname(fd);
```

Arguments:

fd	an open file descriptor

Returns:

A pointer to the null-terminated string CON: if the file descriptor is open and attached to the CP/M-68K console device. Otherwise, zero (NULL) returns.

ungetc Function

The ungetc function pushes a character back to an input stream. The next getc, getw, or getchar operation incorporates the character. One character of buffering is guaranteed if something has been read from the stream. The fseek function erases any pushed-back characters. You cannot ungetc EOF (-1).

Calling Sequence:

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

ret = ungetc(c,stream);
```

Arguments:

c	the character to push back
stream	the stream address

Returns:

ret	c if the character is successfully pushed back
-1	error

unlink Function

The unlink function deletes a named file from the file system. The removal operation fails if the file is open or nonexistent.

Calling Sequence:

```
WORD ret;
BYTE *name;

ret = unlink(name);
```

Arguments:

name	the null-terminated filename

Returns:

0	success
-1	failure

write Function

The write function transfers data to a file opened by file descriptor. Transfer begins at the present file pointer, as set by previous transfers or by the lseek function. You can write any arbitrary number of bytes to the file. The number of bytes actually written returns. If the number of bytes written does not match the number requested, an error occurred.

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

Calling Sequence:

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

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

Arguments:

fd	the open file descriptor
buffer	the starting buffer address
bytes	the number of bytes to write

Returns:

ret	the number of bytes actually written
-1	errors

Note:

Due to the buffering scheme used, all data is not written to the file until the file is closed.

End of Section 2

Section 3
C Style Guide

To make your C language programs portable, readable, and easy to maintain, follow the stylistic rules presented in this section. However, no rule can predict every situation; use your own judgment in applying these principles to unique cases.

3.1 Modularity

Modular programs reduce porting and maintenance costs. Modularize your programs, so that all routines that perform a specified function are grouped in a single module. This practice has two benefits: first, the maintenance programmer can treat most modules as black boxes for modification purposes; and second, the nature of data structures is hidden from the rest of the program. In a modular program, you can change any major data structure by changing only one module.

3.1.1 Module Size

A good maximum size for modules is 500 lines. Do not make modules bigger than the size required for a given function.

3.1.2 Intermodule Communication

Whenever possible, modules should communicate through procedure calls. Avoid global data areas. Where one or more compilations require the same data structure, use a header file.

3.1.3 Header Files

In separately combined files, use header files to define types, symbolic constants, and data structures the same way for all modules. The following list gives rules for using header files.

- Use the '#include "file.h"' format for header files that are project-specific. Use '#include <file.h>' for system-wide files. Never use device or directory names in an include statement.
- Do not nest include files.
- Do not define variables other than global data references in a header file. Never initialize a global variable in a header file.
- When writing macro definitions, put parentheses around each use of the parameters to avoid precedence mix-ups.

3.2 Mandatory Coding Conventions

To make your programs portable, you must adhere strictly to the conventions presented in this section. Otherwise, the following problems can occur:

- The length of a C int variable varies from machine to machine. This can cause problems with representation and with binary I/O that involves int quantities.
- The byte order of multibyte binary variables differs from machine to machine. This can cause problems if a piece of code views a binary variable as a byte stream.
- Naming conventions and the maximum length of identifiers differ from machine to machine. Some compilers do not distinguish between upper- and lower-case characters.
- Some compilers sign-extend character and short variables to int during arithmetic operations; some compilers do not.
- Some compilers view a hex or octal constant as an unsigned int; some do not. For example, the following sequence does not always work as expected:

```
LONG data;
    .
    .
    .
printf("%ld\n",(data & 0xffff));
```

The printf statement prints the lower 16 bits of the long data item data. However, some compilers sign-extend the hex constant 0xffff.

- You must be careful of evaluation-order dependencies, particularly in compound BOOLEAN conditions. Failure to parenthesize correctly can lead to incorrect operation.

3.2.1 Variable and Constant Names

Local variable names should be unique to eight characters. Global variable names and procedure names should be unique to six characters. All variable and procedure names should be completely lower-case.

Usually, names defined with a #define statement should be entirely upper-case. The only exceptions are functions defined as macros, such as getc and isascii. These names should also be unique to eight characters.

You should not redefine global names as local variables within a procedure.

3.2.2 Variable Typing

Using standard types is unsafe in programs designed to be portable due to the differences in C compiler standard type definitions. Instead, use a set of types and storage classes defined with typedef or #define. The following tables define C language types and storage classes.

Table 3-1. Type Definitions

Type	*C Base Type*	
LONG	signed long	(32 bits)
WORD	signed short	(16 bits)
UWORD	unsigned short	(16 bits)
BOOLEAN	short	(16 bits)
BYTE	signed char	(8 bits)
UBYTE	unsigned char	(8 bits)
VOID	void (function return)	
DEFAULT	int	(16/32 bits)

Table 3-2. Storage Class Definitions

Class	*C Base Class*
REG	register variable
LOCAL	auto variable
MLOCAL	module static variable
GLOBAL	global variable definition
EXTERN	global variable reference

Additionally, you must declare global variables at the beginning of the module. Define local variables at the beginning of the function in which they are used. You must always specify the storage class and type, even though the C language does not require this.

3.2.3 Expressions and Constants

Write all expressions and constants to be implementation-independent. Always use parentheses to avoid ambiguities. For example, the construct

```
if(c = getchar() = = '\n')
```

does not assign the value returned by getchar to c. Instead, the value returned by getchar is compared to '\n', and c receives the value 0 or 1 (the true/false output of the comparison). The value that getchar returns is lost. Putting parentheses around the assignment solves the problem:

```
if((c = getchar()) = = '\n')
```

Write constants for masking, so that the underlying int size is irrelevant. In the following example,

```
LONG data;
    .
    .
    .
printf("%ld/n",(data & 0xffffL);
```

the long masking constant solves the previous problem for all compilers. Specifying the one's complement often yields the desired effect, for example, ~0xff instead of 0xff00.

For portability, character constants must consist of a single character. Place multi-character constants in string variables.

Commas that separate arguments in functions are not operators. Evaluation order is not guaranteed. For example, the following function call

```
printf("%d %d\n",i++,i++);
```

can perform differently on different machines.

3.2.4 Pointer Arithmetic

Do not manipulate pointers as ints or other arithmetic variables. C allows the addition or subtraction of an integer to or from a pointer variable. Do not attempt logical operations, such as AND or OR, on pointers. A pointer to one type of object can convert to a pointer to a smaller data type with complete generality. Converting a pointer to a larger data type can yield alignment problems.

You can test pointers for equality with other pointer variables and constants, notably NULL. Arithmetic comparisons, such as >=, do not work on all compilers and can generate machine-dependent code.

When you evaluate the size of a data structure, remember that the compiler might leave holes in a data structure to allow for alignment. Always use the sizeof operator.

3.2.5 String Constants

Allocate strings so that you can easily convert programs to foreign languages. The preferred method is to use an array of pointers to constant strings, which is initialized in a separate file. This way, each string reference then references the proper element of the pointer array.

Never modify a specific location in a constant string, as in the following example:

```
BYTE    string[] ="BDOS Error On x:";
        .

        .
string[14] = 'A';
```

Foreign-language equivalents are not likely to be the same length as the English version of a message.

Never use the high-order bit of an ASCII string for bit flags. Extended character sets make extensive use of the characters above 0x7F.

3.2.6 Data and BSS Sections

Usually, C programs have three sections: text (program instructions), data (initialized data), and BSS (uninitialized data). Avoid modifying initialized data if at all possible. Programs that do not modify the data segment can aid the swapping performance and disk utilization of a multiuser system.

Also, if a program does not modify the data segment, you can place the program in ROM with no conversion. This means that the program does not modify initialized static variables. This restriction does not apply to the modification of initialized automatic variables.

3.2.7 Module Layout

The following list tells you what to include in a module.

- At the beginning of the file, place a comment describing the following items:
  - the purpose of the module
  - the major outside entry points to the module
  - any global data areas that the module requires
  - any machine or compiler dependencies
- Include file statements.
- Module-specific #define statements.
- Global variable references and definitions. Every variable should include a comment describing its purpose.
- Procedure definitions. Each procedure definition should contain the following items:
  - A comment paragraph, describing the procedure's function, input parameters, and return parameters. Describe any unusual coding techniques here.
  - The procedure header. The procedure return type must be explicitly specified. Use VOID when a function returns no value.
  - Argument definitions. You must explicitly declare storage class and variable type.
  - Local variable definitions. Define all local variables before any executable code. You must explicitly declare storage class and variable type.
  - Procedure code.

Refer to Appendix C for a sample program.

Appendix A
Summary of BIOS Functions

Table A-1 lists the BIOS functions supported by CP/M-68K. For more details on these functions, refer to the *CP/M-68K Operating System System Guide*.

Table A-1. Summary of BIOS Functions

Function	*F#*	*Description*
Init	0	Called for Cold Boot
Warm Boot	1	Called for Warm Start
Const	2	Check for Console Character Ready
Conin	3	Read Console Character In
Conout	4	Write Console Character Out
List	5	Write Listing Character Out
Auxiliary Output	6	Write Character to Auxiliary Output Device
Auxiliary Input	7	Read from Auxiliary Input Device
Home	8	Move to Track 00
Seldsk	9	Select Disk Drive
Settrk	10	Set Track Number
Setsec	11	Set Sector Number
Setdma	12	Set DMA Offset Address
Read	13	Read Selected Sector
Write	14	Write Selected Sector
Listst	15	Return List Status
Sectran	16	Sector Translate
Get Memory Region Table Address	18	Address of Memory Region Table
Get I/O Byte	19	Get I/O Mapping Byte
Set I/O Byte	20	Set I/O Mapping Byte
Flush Buffers	21	Writes Modified Buffers
Set Exception Vector	22	Sets Exception Vector

End of Appendix A

Table A-1. (continued)

Number	*Name*	*Error Message*
21	-	Error Undefined on CP/M-68K
22	EINVAL	Invalid argument
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
26	-	Error Undefined on CP/M-68K
27	EFBIG	File too big
28	ENOSPC	No space left on device
29	-	Error Undefined on CP/M-68K
30	EROFS	Read-Only file system
31	-	Error Undefined on CP/M-68K
32	-	Error Undefined on CP/M-68K
33	-	Error Undefined on CP/M-68K
34	-	Error Undefined on CP/M-68K
35	ENODSPC	No directory space

The file <errno.h> also includes the names for all errors defined with UNIX V7. Therefore, programs that reference these definitions need not be changed.

End of Appendix A

Appendix B
Customizing the C Compiler

Compiling a C program requires three compiler passes. The output of the compiler is assembly language, which must be assembled and linked to produce a program that runs. The compiler, assembler, linker load modules, C library, and the system include files need a substantial amount of disk storage space, minimizing storage space. This appendix discusses compiler operation and suggests ways to minimize the disk storage requirements for compiling.

B.1 Compiler Operation

The C compiler has three components: the preprocessor (CP68), the parser (C068), and the code generator (C168). The assembler (AS68) and the linker (LO68) also help generate an executable program. The following list tells you how these components operate.

1. The preprocessor, CP68, takes the original source file and produces a file with all #define and #include statements resolved. The preprocessor command line takes the form:

   CP68 [-I d:] file.C file.I

   The -I flag indicates that the next argument is a CP/M-68K drive specification. This drive is used for all library include statements of the form #include <file>. Drive specifications can also appear in the filename portion of an #include statement, but this procedure is not recommended. File.C is the source file, and file.I is the output file.
2. The parser, C068, takes the file produced by the preprocessor and creates an intermediate code file. The command line takes the form:

   C068 file.I file.IC file.ST

   File.I is the output from the preprocessor. File.IC is the intermediate code file that C168 uses. File.ST is a temporary file that collects constant data for inclusion at the end of the intermediate code file.

3. The code generator, C168, takes the intermediate code file from C068 and produces an assembly-language source file. The command line takes the form:

   C168 file.IC file.S [-LD]

   File.IC is the intermediate code output from C068. File.S is the assembly-language output file. The -L flag indicates that the compilation assumes all address variables are 32 bits. The default is 16-bit addresses. The -D flag causes the compiler to include the line numbers from the source file (file.C) as comments in the generated assembly language. This is useful for debugging.

4. The assembler, AS68, translates the compiler output to a form that the linkage editor can use. The command line takes the form:

   AS68 -L -U [-F d:] [-S d:] file.S

   The -L option indicates to the linkage editor that addresses are considered 32-bit quantities. The -U option means that undefined symbols are considered external references. The -F option specifies a drive that the assembler uses for temporary files. The -S option specifies a drive that the assembler uses for the initialization file (AS68SYMB.DAT). File.S is the output of C168, and file.O is produced by the assembler.

5. The linker, LO68, produces an executable file from the output of one or more assembler runs. You must also include a start-up file and the C library when linking C programs. The linker command line takes the form:

   LO68 -R [-F d:] -O file.68K S.O file.O clib

   The -R option specifies that the file be relocatable. Relocatable files run on any CP/M-68K system. The -F option allows you to place linker temporary files on a disk drive other than the default. The -O file.68K construct makes the linker place the executable output in file.68K. S.O is the run-time start-up routine. You must include this file as the first file in every C program link. File.O is the output of the assembler. Specify multiple files between S.O and clib if you want separate compilation. clib is the C library file.

B.2 Supplied submit Files

CP/M-68K includes two submit files, c.sub and clink.sub, that compile and link C programs (see Section 1.1). Usually, these files are located on the default drive. However, you can edit these files to specify different disk drives for any of the following drives:

- The disk drive on which the compiler passes, assembler, and linker reside.
- The disk drive that the #include <file> statements in the C preprocessor reference.
- The disk drive with the assembler initialization file.
- The disk drive on which the assembler and linker create temporary files.
- The disk drive containing the C library file.

B.3 Saving Disk Space

You can do the following things to conserve disk space:

- Use the reloc utility on all the load modules, the compiler, assembler, linker, and editor. This significantly reduces file size and load time.
- Place all the load modules on one disk and use another disk for sources and temporary files. This requires two drives.
- On single-density disk systems, you must place the C library file and linker on a separate disk and swap disks before linking.

B.4 Gaining Speed

Along with the items in Section B.3, you can speed compilation by implementing the following:

- Put the assembler temp files on a different drive from the source and object files.
- Put the linker temp files on a different drive from the object input, C library, and load module output.
- Use the linker -S (suppress symbol table) and -T (absolute load module) switches in place of the -R flag. If you do this, the resulting program cannot run on an arbitrary CP/M-68K system.

End of Appendix B

Appendix C
Sample C Module

The modules in this appendix are written and documented in C code that follows the style conventions discussed in Section 3.

```
/*****************************************************************************/
/*                                                                           */
/*                      _ P r i n t f  M o d u l e                           */
/*                      --------------------------                           */
/*                                                                           */
/*      This module is called through the single entry point "_printf" to    */
/*      perform the conversions and output for the library functions:        */
/*                                                                           */
/*            printf  - Formatted print to standard output                   */
/*            fprintf - Formatted print to stream file                       */
/*            sprintf - Formatted print to string                            */
/*                                                                           */
/*      The calling routines are logically a part of this module, but are    */
/*      compiled separately to save space in the user's program when only    */
/*      one of the library routines is used.                                 */
/*                                                                           */
/*      The following routines are present:                                  */
/*                                                                           */
/*            _printf           Internal printf conversion / output          */
/*            _prnt8            Octal conversion routine                     */
/*            _prntx            Hex conversion routine                       */
/*            __conv            Decimal ASCII to binary routine              */
/*            _putstr           Output character to string routine           */
/*            _prnt1            Decimal conversion routine                   */
/*                                                                           */
/*      The following routines are called:                                   */
/*                                                                           */
/*            strlen            Compute length of a string                   */
/*            putc              Stream output routine                        */
/*            ftoa              Floating point output conversion routine     */
/*                                                                           */
/*                                                                           */
/*      This routine depends on the fact that the argument list is always    */
/*      composed of LONG data items.                                         */
/*                                                                           */
/*      Configured for Whitesmith's C on VAX.  "putc" arguments are          */
/*      reversed from UNIX.                                                  */
/*                                                                           */
/*****************************************************************************/

/*
 *      Include files:
 */
#include        <stdio.h>                       /* Just the standard stuff */
```

```
/*
 *      Local DEFINEs
 */
#define HIBIT   31                              /* High bit number of LONG  */


/*
 *      Local static data:
 */                                             /*****************************/
        MLOCAL BYTE     *_ptrbf = 0;            /*  Buffer.Pointer           */
        MLOCAL BYTE     *_ptrst = 0;            /*  -> File/string (if any) */
        MLOCAL BYTE     *__fmt  = 0;            /*  Format Pointer           */
                                                /*****************************/
```

```
/*****************************************************************************
*
*               P R I N T F   I N T E R N A L   R O U T I N E
*               ---------------------------------------------
*
*       Routine "_printf" is used to handle all "printf" functions, including
*       "sprintf", and "fprintf".
*
*       Calling Sequence:
*
*               _printf(fd,func,fmt,arg1);
*
*       Where:
*
*               fd              Is the file or string pointer.
*               func            Is the function to handle output.
*               fmt             Is the address of the format string.
*               arg1            Is the address of the first arg.
*
*
*       Returns:
*
*               Number of characters output
*
*       Buss:
*
*       It is assumed that args are contiguous starting at "arg1", and that
*       all are the same size (LONG), except for floating point.
*
*
*****************************************************************************/
_printf(fd,f,fmt,a1)                    /*********************************/
        LONG    fd;                     /* Not really, but ...           */
        LONG    (*f)();                 /* Function pointer              */
        BYTE    *fmt;                   /* -> Format string              */
        LONG    *a1;                    /* -> Arg list                   */
{                                       /*********************************/
        LOCAL BYTE      c;              /* Format character temp         */
        LOCAL BYTE      *s;             /* Output string pointer         */
        LOCAL BYTE      adj;            /* Right/left adjust flag        */
        LOCAL BYTE      buf[30];        /* Temporary buffer              */
                                        /*********************************/
        LOCAL LONG      *adx;           /* Arg Address temporary         */
        LOCAL LONG      x;              /* Arg Value   temporary         */
        LOCAL LONG      n;              /* String Length Temp            */
        LOCAL LONG      m;              /* Field  Length Temporary       */
        LOCAL LONG      width;          /* Field width                   */
        LOCAL LONG      prec;           /* Precision for "%x.yf"         */
        LOCAL LONG      padchar;        /* '0' or ' ' (padding)          */
        LOCAL DOUBLE    zz;             /* Floating temporary            */
        LOCAL DOUBLE    *dblptr;        /* Floating temp. address        */
        LOCAL LONG      ccount;         /* Character count               */
        EXTERN          _putstr();      /* Reference function            */
                                        /*********************************/
```

```
        ccount = 0;                                 /*****************************/
        _ptrbf = buf;                               /* Initially no characters */
        adx = al;                                   /* Set buffer pointer      */
         _ptrst = fd;                               /* Copy address variable   */
         __fmt = fmt;                               /* Copy file descriptor    */
                                                    /* Copy format address     */
        if(*__fmt == 'L' || *__fmt == 'l')          /*****************************/
               __fmt++;                             /* Skip long output        */
                                                    /*      conversions        */
                                                    /*                         */
/*****************************************************                         */
/* This is the main format conversion loop.  Load a character from the       */
/* format string.  If the character is '%', perform the appropriate          */
/* conversion.  Otherwise, just output the character.                        */
/*****************************************************                         */
                                                    /*                         */
        while( c = *__fmt++ )                       /* Pick up next format char*/
        {                                           /*                         */
          if(c != '%')                              /*****************************/
          {                                         /*                         */
            (*f)(fd,c);                             /* If not '%', just output */
            ccount++;                               /* Bump character count    */
          }                                         /*****************************/
            else                                    /* It is a '%',            */
          {                                         /*            convert      */
            x = *adx++;                             /* x = address of next arg */
                                                    /*****************************/
            if( *__fmt == '-' )                     /* Check for left adjust   */
            {                                       /*****************************/
              adj = 'l';                            /* Is left, set flag       */
              __fmt++;                              /* Bump format pointer     *
            }                                       /*                         */
            else                                    /* Right adjust            */
              adj = 'r';                            /*****************************/
                                                    /*                         */
            padchar=(*__fmt=='0') ? '0' : ' ';      /* Select Pad character    */
                                                    /*****************************/
              width = __conv();                     /* Convert width (if any)  */
                                                    /*****************************/
            if( *__fmt == '.')                      /* '.' means precision spec*/
            {                                       /*                         */
              ++__fmt;                              /* Bump past '.'           */
              prec = __conv();                      /* Convert precision spec  */
            }                                       /*                         */
            else                                    /* None specified          */
              prec = 0;                             /*****************************/
                                                    /*                         */
            s = 0;                                  /* Assume no output string */
            switch ( c = *__fmt++ )                 /* Next char is conversion */
            {                                       /*                         */
              case 'D':                             /* Decimal                 */
              case 'd':                             /*                         */
              _prtl(x);                             /* Call decimal print rtn  */
              break;                                /* Go do output            */
                                                    /*****************************/
```

```
    case 'o':                          /* Octal                    */
    case 'O':                          /*       Print              */
      _prnt8(x);                       /* Call octal printer       */
      break;                           /* Go do output             */
                                       /****************************/
    case 'x':                          /* Hex                      */
    case 'X':                          /*      Print               */
      _prntx(x);                       /* Call conversion routine  */
      break;                           /* Go do output             */
                                       /****************************/
    case 'S':                          /* String                   */
    case 's':                          /*         Output?          */
      s=x;                             /* Yes, (easy)              */
      break;                           /* Go finish up             */
                                       /****************************/
    case 'C':                          /* Character                */
    case 'c':                          /*      Output?             */
      *_ptrbf++ = x&0377;              /* Just load buffer         */
      break;                           /* Go output                */
                                       /****************************/
    case 'E':                          /* Floating point?          */
    case 'e':                          /*                          */
    case 'F':                          /*                          */
    case 'f':                          /*                          */
      dblptr = adx-1;                  /* Assumes 64 bit float!    */
      zz = *dblptr;                    /* Load value               */
      adx =+ 1;                        /* Bump past second word    */
      ftoa (zz, buf, prec, c);         /* Call floating conversion*/
      prec = 0;                        /* Fake out padding routine*/
      s = buf;                         /* Just like string print   */
      break;                           /* Go Output                */
                                       /****************************/
    default:                           /* None of the above?       */
      (*f)(fd,c);                      /* Just Output              */
     ccount++;                         /* Count it.                */
      adx--;                           /* Fix arg address          */
}                                      /* End switch               */
                                       /****************************/
if (s == 0)                            /* If s = 0, string is in   */
{                                      /* "buf",                   */
  *_ptrbf = '0';                       /*   Insure termination     */
  s = buf;                             /*   Load address           */
}                                      /****************************/
                                       /*                          */
n = strlen (s);                        /* Compute converted length*/
n = (prec<n && prec != 0) ? prec : n;/* Take min(prec,n)         */
m = width-n;                           /* m is # of pad characters*/
                                       /****************************/
if (adj == 'r')                        /* For right adjust,        */
   while (m-- > 0)                     /* Pad in front             */
   {                                   /*                          */
     (*f)(fd,padchar);                 /* Thusly                   */
     ccount++;                         /* Count it                 */
   }                                   /*                          */
                                       /****************************/
```

```
            while (n--)
            {
                (*f)(fd,*s++);
                ccount++;
            }

            while (m-- > 0)
            {
                (*f)(fd,padchar);
                ccount++;
            }
            _ptrbf = buf;
          }
        }
        if((*f) == _putstr)
              (*f)(fd,'0');

        return(ccount);
}
```

```
/* Output Converted          */
/*                           */
/*                    Data   */
/* Count it                  */
/*                           */
/*****************************/
/* If left adjust,           */
/*                           */
/*                 Pad       */
/* Count padded characters   */
/*****************************/
/* Reset buffer pointer      */
/* End else                  */
/* End while                 */
/* If string output,         */
/* Drop in terminator char   */
/*****************************/
/* Return appropriate value*/
/* End _printf               */
/*****************************/
```

```
/*****************************************************************************/
/*                                                                           */
/*                  _ P R N T 8   P R O C E D U R E                          */
/*                  -------------------------------                          */
/*                                                                           */
/*      Routine "_prnt8" converts a binary LONG value to octal ascii.        */
/*      The area at "_ptrbf" is used.                                        */
/*                                                                           */
/*      Calling Sequence:                                                    */
/*                                                                           */
/*            _prnt8(n);                                                     */
/*                                                                           */
/*      "n" is the number to be converted.                                   */
/*                                                                           */
/*      Returns:                                                             */
/*                                                                           */
/*            (none)                                                         */
/*                                                                           */
/*****************************************************************************/
VOID _prnt8 (n)                          /*                                 */
      LONG      n;                       /* Number to convert               */
{                                        /*                                 */
      REG WORD      p;                   /* Counts bits                     */
      REG WORD      k;                   /* Temporary 3-bit value           */
      REG WORD      sw;                  /* Switch 1 => output              */
                                         /*********************************/
      if (n==0)                          /* Handle 0 as special case        */
      {                                  /*                                 */
            *_ptrbf++ = '0';             /* Put in one zero                 */
            return;                      /*      And quit                   */
      }                                  /*                                 */
                                         /*********************************/
      sw = 0;                            /* Indicate no output yet          */
                                         /*                                 */
      for (p=HIBIT; p >= 0; p -= 3)      /* Use 3 bits at a time            */
                                         /*                                 */
      if ((k = (n>>p)&07) || sw)         /* Need to output yet?             */
      {                                  /*                                 */
            if (p==HIBIT)                /* 1st digit has only 2 bits       */
                  k = k & 02;            /* Mask appropriately              */
            *_ptrbf++ = '0' + k;         /* ASCIIfy digit                   */
            sw = 1;                      /* Set output flag                 */
      }                                  /* End if                          */
}                                        /* End _prnt8                      */
                                         /*********************************/
```

```
/*****************************************************************************/
/*                                                                           */
/*                  _ P r n t x   F u n c t i o n                            */
/*                  ---------------------------                              */
/*                                                                           */
/*      The "_prntx" function converts a binary LONG quantity to hex ASCII   */
/*      and stores the result in "*_ptrbf". Leading zeros are suppressed.     */
/*                                                                           */
/*      Calling sequence:                                                    */
/*                                                                           */
/*            _prntx(n);                                                     */
/*                                                                           */
/*      where "n" is the value to be converted.                              */
/*                                                                           */
/*      Returns:                                                             */
/*                                                                           */
/*            (none)                                                         */
/*                                                                           */
/*****************************************************************************/
VOID _prntx (n)                             /*                               */
     LONG      n;                           /* 32 bits                       */
{                                           /*******************************/
     REG LONG       d;                      /* A digit                       */
     REG LONG       a;                      /* Temporary value               */
                                            /*******************************/
     if (a = n>>4)                          /* Peel off low 4 bits           */
          _prntx ( a & 0xfffffff);          /* If <> 0, print first          */
     d = n&017;                             /* Take low four bits            */
     *_ptrbf++ =  d > 9 ? 'A'+d-10 : '0' + d;/* ASCIIfy into buffer           */
}                                           /*******************************/
```

```
/*****************************************************************************/
/*                                                                           */
/*                  __Conv  Function                                         */
/*                  ----------------                                         */
/*                                                                           */
/*      Function "__conv" is used to convert a decimal ASCII string in       */
/*      the format to binary.                                                */
/*                                                                           */
/*      Calling Sequence:                                                    */
/*                                                                           */
/*            val = __conv();                                                */
/*                                                                           */
/*      Returns:                                                             */
/*                                                                           */
/*            "val" is the converted value                                   */
/*            Zero is returned if no value                                   */
/*                                                                           */
/*****************************************************************************/
LONG __conv()                                   /*                           */
{                                               /***************************/
      REG   BYTE      c;                        /* Character temporary     */
      REG   LONG      n;                        /* Accumulator             */
                                                /***************************/
      n = 0;                                    /* Zero found so far       */
      while(((c= *__fmt++) >= '0')              /* While c is a digit      */
            && (c <= '9'))                      /*                         */
            n = n*10+c-'0';                     /* Add c to accumulator    */
      __fmt--;                                  /* Back up format pointer to*/
                                                /* character skipped above */
      return(n);                                /* See, wasn't that simple? */
}                                               /***************************/
```

```
/*************************************************************************/
/*                                                                       */
/*                  _ P u t s t r   F u n c t i o n                      */
/*                  -------------------------------                      */
/*                                                                       */
/*      Function "_putstr" is used by "sprintf" as the output function   */
/*      argument to "_printf".  A single character is copied to the buffer*/
/*      at "_ptrst".                                                     */
/*                                                                       */
/*      Calling Sequence:                                                */
/*                                                                       */
/*                  _putstr(str,chr);                                    */
/*                                                                       */
/*      where "str" is a dummy argument necessary because the other output*/
/*      functions have two arguments.                                    */
/*                                                                       */
/*      Returns:                                                         */
/*                                                                       */
/*            (none)                                                     */
/*                                                                       */
/*************************************************************************/
VOID _putstr(str,chr)                          /*                       */
      REG BYTE      chr;                       /* The output character  */
      BYTE          *str;                      /* Dummy argument        */
{                                              /***********************/
      *_ptrst++ = chr;                         /* Output the character  */
      return(0);                               /* Go back               */
}                                              /***********************/
```

```
/*****************************************************************************/
/*                                                                           */
/*                  _ P r t 1   F u n c t i o n                              */
/*                  ---------------------------                              */
/*                                                                           */
/*      Function "_prt1" converts a LONG binary quantity to decimal ASCII    */
/*      at the buffer pointed to by "_ptrbf".                                */
/*                                                                           */
/*      Calling Sequence:                                                    */
/*                                                                           */
/*            _prt1(n);                                                      */
/*                                                                           */
/*      where "n" is the value to be converted.                              */
/*                                                                           */
/*      Returns:                                                             */
/*                                                                           */
/*            (none)                                                         */
/*                                                                           */
/*****************************************************************************/
VOID _prt1(n)                                  /*                           */
      REG LONG      n;                         /* Conversion input          */
{                                              /*****************************/
      REG LONG      digs[15];                  /* store digits here         */
      REG LONG      *dpt;                      /* Points to last digit      */
                                               /*****************************/
      dpt = digs;                              /* Initialize digit pointer  */
                                               /*****************************/
      if (n >= 0)                              /* Fix                       */
         n = -n;                               /*      up                   */
      else                                     /*           sign            */
         *_ptrbf++ = '-';                      /*           stuff           */
                                               /*****************************/
      for (; n != 0; n = n/10)                 /* Divide by 10 till zero    */
       *dpt++ = n%10;                          /* Store digit (reverse ord)*/
                                               /*****************************/
      if (dpt == digs)                         /* Zero value?               */
         *dpt++ = 0;                           /* Yes, store 1 zero digit   */
                                               /*****************************/
      while (dpt != digs)                      /* Now convert to ASCII      */
      {                                        /*                           */
         --dpt;                                /* Decrement pointer         */
         *_ptrbf++ =  '0' - *dpt;              /* Note digits are negative!*/
      }                                        /*                           */
}                                              /*****************************/
```

End of Appendix C

Appendix D
Error Messages

This appendix lists the error messages returned by the components of the CP/M-68K C compiler, the C Parser, C068, the C Co-generator, C168, the C Preprocessor, CP68, and by the CP/M-68K C Run-time Library. The sections are arranged alphabetically. Error messages are listed within each section in alphabetical order with explanations and suggested user responses.

D.1 C068 Error Messages

The CP/M-68K C Parser, C068, returns two types of error messages: diagnostic error messages and messages indicating errors in the internal logic of C068. Both types of error messages take the general form:

*line no. error message text

The asterisk (*) indicates that the error message comes from C068. The "error message text" describes the error. You must correct any errors you receive from C068 before invoking C168. Uncorrected errors from C068 cause erroneous error messages to occur when you run C168.

D.1.1 Diagnostic Error Messages

These error messages occur mostly in response to syntax errors in the source code. Refer to your C language manual for a complete discussion of the C language syntax.

The error messages are listed in Table D-1 in alphabetical order with short explanations and suggested user responses.

Table D-1. C068 Diagnostic Error Messages

Message	*Meaning*
`*line no. address of register`	You have attempted to take the address of a register. Correct the source code before you recompile it.
`*line no. assignable operand required`	On the line indicated, the operand to the left of the equals sign in the assignment statement is not a valid operand. Supply a valid operand. This error might occur because the operand is a constant instead of a variable.
`*line no. bad character constant`	A character constant on the line indicated is invalid. The character constant must be a single character between quotes. A control character, more than one character, or a symbol that is not a character will cause this error to occur.
`*line no. bad indirection`	You attempted to reference by address instead of by value, but the expression you used is not an address. Supply a value or a valid address before you recompile the source code.
`*line no. can't open filename`	Either the filename or the drive code is incorrect. Specify the correct drive code and filename before you recompile the source code.
`*line no. case not inside a switch block`	The case on the line indicated is not inside a switch block. Correct the source code before you recompile it.
`*line no. character constant too long`	The character constant on the line indicated is too long. A character constant must be a single character between quotes. Correct the source code before you recompile it.
`*line no. constant required`	The operation on the line indicated requires a constant. Correct the error before you recompile the source code.

Table D-1. (continued)

Message	*Meaning*
`*line no. declaration syntax`	The syntax of the declaration on the line indicated is incorrect. Refer to your C language manual. Correct the syntax before you recompile the source code.
`*line no. default not inside a switch block`	The default on the line indicated is not inside a switch block. Correct the source code before you recompile it.
`*line no. dimension table overflow`	There are too many dimensions, at or prior to the line indicated, for the dimension table. The dimension table does not have space for more than 8 or 9 dimensions. Structures count as dimensions. Rewrite the source code to use fewer dimensions and structures before you recompile it.
`*line no. duplicate case value`	Two cases for the same switch are identical. Eliminate one of the cases before you recompile the source code.
`*line no. expected label`	A go to statement on the line indicated does not have a label. Supply the missing label before you recompile the source code.
`*line no. expression too complex`	Due to internal limitations in C068, the expression on the line indicated is too complex to be evaluated. Simplify the expression before recompiling the source code.
`*line no. external definition syntax`	The syntax of the external definition on the line indicated is incorrect. Correct the syntax before you recompile the source code. Refer to your C language manual for the correct syntax.
`*line no. field overflows byte`	The bit field asks for more bits than fit in an 8-bit byte. Reduce the number of bits in the bit field before you recompile the source code.

Table D-1. (continued)

Message	*Meaning*
*line no. field overflows word	The word field asks for more bytes than fit in a word. Reduce the number of bytes in the byte field before you recompile the source code.
*line no. floating point not supported	CP/M-68K does not support floating point. Rewrite the source code before you recompile it.
*line no. function body syntax	There is no bracket at the beginning of the function on the line indicated. Supply the missing bracket before you recompile the source code.
*line no. illegal call	You attempted to call something that is not a function. Correct the source code before you recompile it.
*line no. illegal function declaration	The storage class of the function declared in the line indicated is illegal. The only two storage classes allowed for functions are static and external. Correct the declaration before you recompile the source code.
*line no. illegal register specification	The register specification in the line indicated is illegal. Structures and arrays cannot be put into a register. Correct the source code before you recompile it.
*line no. illegal type conversion	You made an incompatible assignment. This error commonly occurs when attempting to convert a pointer, 32 bits, to an int, 16 bits. Correct the source code before you recompile it.
*line no. indirection on function invalid	You attempted to use the indirection operator (*) on a function. Correct the source code before you recompile it.

Table D-1. (continued)

Message	*Meaning*
`*line no. initializer alignment`	This message usually indicates a missing initializer value, or values out of order. Check the initializer list and correct it before you recompile the source code.
`*line no. initializer list too long`	The initializer list is too long for C068. Shorten the list before you recompile the source code.
`*line no. invalid break statement`	The break statement on the line indicated is not inside a loop or a switch. Correct the source code before you recompile it.
`*line no. invalid character`	There is an invalid character in the collating sequence in the line indicated. Control characters or members of the extended character set are not valid characters. Correct the source code before you recompile it.
`*line no. invalid continue statement`	The continue statement on the line indicated is not inside a loop. This error might occur when you have used a continue statement in a switch. A continue statement is only valid in a loop. Correct the source code before reinvoking C068.
`*line no. invalid conversion`	You attempted an incompatible assignment, for example, a pointer, 32 bits, and an int, 16 bits. Correct the source code before you recompile it.
`*line no. invalid data type`	The line indicated contains an expression that attempts to equate two incompatible quantities, for example, an int, 16 bits, and a pointer, 32 bits. Correct the source code before you recompile it.

Table D-1. (continued)

Message	*Meaning*
`*line no. invalid declarator`	The declarator in the line indicated is not a recognizable language element. Supply a valid declarator before you recompile the source code.
`*line no. invalid expression`	The expression in the line indicated contains a syntax error. Correct the syntax of the expression before you recompile the source code.
`*line no. invalid field size`	The field in the line indicated is less than or equal to zero. Correct the field size before you recompile the source code.
`*line no. invalid field type description`	You attempted to put a pointer or a long into a bit field. Correct the source code before you recompile it.
`*line no. invalid for statement`	The for statement in the line indicated contains a syntax error. Refer to your C language manual for the correct syntax of a for statement. Correct the statement before you recompile the source code.
`*line no. invalid initializer`	The initializer you specified in the line indicated is not a constant. You can only initialize to a constant. Correct the source code before you recompile it.
`*line no. invalid label`	You used a variable name as a label in the line indicated. Correct the source code before you recompile it.
`*line no. invalid long declaration`	You attempted to declare something long that cannot be long, for example, a character. Correct the source code before you recompile it.

Table D-1. (continued)

Message	*Meaning*
`*line no. invalid operand type`	The expression in the line indicated contains an invalid operand. Correct the source code before you recompile it.
`*line no. invalid register specification`	You attempted to put something larger than allowed into a register, for example, a structure or a function. Correct the source code before you recompile it.
`*line no. invalid short declaration`	You attempted to declare something short that cannot be short. Correct the source code before you recompile it.
`*line no. invalid storage class`	You specified an invalid storage class in a declaration. Refer to your C language manual for the allowed storage classes. Correct the source code before you recompile it.
`*line no. invalid structure declaration: name`	The size of the structure indicated by the variable `name` has a size less than or equal to zero. Correct the source code before you recompile it.
`*line no. invalid structure member name`	The structure reference in the line indicated is not a member of any structure. Correct the source code before you recompile it.
`*line no. invalid structure prototype: name`	In the line indicated you reference a structure name that is not a prototype. Correct the source code before you recompile it.
`*line no. invalid type declaration`	The type declared in the line indicated is invalid. Refer to your C language manual for a discussion of valid types. Correct the source code before you recompile it.

Table D-1. (continued)

Message	*Meaning*
`*line no. invalid typedef statement`	The line indicated contains a statement with more than one typedef keyword. Only one typedef is allowed per statement. Correct the source code before you recompile it.
`*line no. invalid unsigned declaration`	The quantity you declared unsigned in the line indicated might not be unsigned. Only an int can be unsigned. Correct the declaration before you recompile the source code.
`*line no. invalid ?: operator syntax`	This message indicates an error in the use of the ?: conditional operator in the line indicated. Refer to your C language manual for the correct syntax. Correct the source code before you recompile it.
`*line no. label redeclaration: label`	You used the same label for two separate items. Correct the source code before you recompile it.
`*line no. missing colon`	You left out a colon. Supply a colon in the correct location before you recompile the source code.
`*line no. missing { in initialization`	You neglected to put in the left curly brace in the initialization of an array or structure. Supply the missing brace before you recompile the source code.
`*line no. missing }`	You left the right curly brace out of the initialization of an array or structure. Supply the missing brace before you recompile the source code.
`*line no. missing while`	The do statement at the line indicated is missing a while at the end. Supply the missing while before you recompile the source code.

Table D-1. (continued)

Message	*Meaning*
*line no. missing semicolon	A semicolon is missing from the line indicated. Supply the missing semicolon before you recompile the source code.
*line no. no structure name	You referred to a structure in the line indicated without giving the structure name. Correct the source code before you recompile it.
*line no. no */ before EOF	The last comment in the source code is missing its final delimiter. Supply the missing delimiter before you recompile the source code.
*line no. not a structure: name	The structure referenced in the line indicated is not a structure. Correct the source code before you recompile it.
*line no. not in parameter list: x	In the line indicated, you declared the something indicated by the variable x to be an argument to a function, but x is not in the function parameter list. Correct the source code before you recompile it.
*line no. parenthesized expression syntax	The line indicated contains a syntax error in the parenthesized expression. Correct the source code before you recompile it.
*line no. redeclaration: symbol	A symbol has been declared twice. Remove one of the declarations before recompiling the source code.
*line no. string cannot cross line	The character string at the line indicated continues beyond one line. The closing quote to a character string must be on the same line as the opening quote, unless you use a backslash (\) at the end of the first line to indicate that the line continues. Correct the source code before you recompile it.

Table D-1. (continued)

Message	*Meaning*
*line no. string too long	The string at the line indicated is longer than 255 characters. A string cannot be longer than 255 characters on a single line. Break the string and use a continuation, indicated by a backslash (\) at the end of the line to be continued.
*line no. structure declaration syntax	The syntax of the structure declaration on the line indicated is incorrect. Correct the syntax before reinvoking C068.
*line no. structure operation not yet implemented	On the line indicated, you assigned a structure to another structure. Assigning a structure to another structure is not yet supported by the CP/M-68K C compiler. Correct the source code before reinvoking C068.
*line no. structure table overflow	There are too many structures in your program for the structure tables. Eliminate some structures before reinvoking the C compiler.
*line no. symbol table overflow	Your program uses too many symbols for the space available on the symbol table. Eliminate some symbols before reinvoking the C compiler.
*line no. temp creation error	The drive code or filename of the temporary file referenced in the line indicated is incorrect. Specify the correct drive code and filename before you recompile the source code.
*line no. too many cases in switch	The switch at the line indicated has too many cases. Eliminate some cases before you recompile the source code.

Table D-1. (continued)

Message	*Meaning*
`*line no. too many initializers`	The initializer list in the line indicated contains more initializers than there are members of the array being initialized. Correct the list before you recompile the source code.
`*line no. too many params`	The function declaration at the line indicated contains too many parameters. Rewrite the source code before you recompile the source code.
`*line no. undefined label: label`	The label indicated by the variable `label` has not been defined. Correct the source code before you recompile it.
`*line no. undefined symbol: symbol`	The symbol indicated by the variable `symbol` is undefined. Correct the source code before you recompile it.
`*line no. unexpected EOF`	This error usually occurs when there is no right curly brace (}) after a function, or when there are mismatched comment delimiters. Locate and correct the error before you recompile the source code.
`*line no. usage: c068 source asm str`	The syntax of the C compiler command line is incorrect. The correct syntax is given in the error message. Reenter the command line using a valid syntax.
`*line no. { not matched by }`	A left curly brace ({) is not matched by a right curly brace. This error frequently occurs in an initialization sequence. Supply the missing brace before you recompile the source code.

Table D-1. (continued)

Message	*Meaning*
`*line no. ="char" assumed`	You have user a = + type operation with an invalid character. When an invalid character occurs after the = sign, C068 puts in = = instead of =. Correct the source code before you recompile the source code.
`*line no. & operand illegal`	You attempted to take the address of something that is not a variable, for example, a register. Correct the source code and recompile it.

D.1.2 Internal Logic Errors

These messages indicate fatal errors in the internal logic of C068:

```
*line no. can't copy filename
*line no. invalid keyword
*line no. too many chars pushed back
*line no. too many tokens pushed back
```

Contact the place you purchased your system for assistance. Provide the following information:

- Indicate the version of the operating system you are using.
- Describe your system's hardware configuration.
- Provide sufficient information to reproduce the error. Indicate which program was running at the time the error occurred. If possible, also provide a disk with a copy of the program.

D.2 C168 Error Messages

The CP/M-68K C Co-generator, C168, returns two types of fatal error messages: diagnostic error messages and messages indicating errors in the internal logic of C168. Both types of error messages take the general form:

```
**line no. error message text
```

The asterisks (**) indicate that the error message comes from C168. The error message text describes the error. If you run C168 before correcting any errors you received from C068, you receive erroneous errors from C168.

D.2.1 Fatal Diagnostic Errors

The C168 fatal, diagnostic error messages are listed in Table D-2 in alphabetical order, with explanations and suggested user responses.

Table D-2. C168 Fatal Diagnostic Errors

Message	*Meaning*
`**line no. can't create filename`	Either the drive code or the filename for the file indicated by the variable `filename` is incorrect. Ensure that you are requesting the correct drive code and filename before you recompile the source code.
`**line no. can't open filename`	Either the drive code or the filename for the file indicated by the variable `filename` is incorrect. Ensure that you are requesting the correct drive code and filename before you recompile the source code.
`**line no. divide by zero`	You attempted to divide by zero in the line indicated. Correct the source code before you recompile it.
`**line no. expression too complex`	An expression on the line indicated is too complex for C168. Simplify the expression before you recompile the source code.

Table D-2. (continued)

Message	*Meaning*
**line no. modulus by zero	The second operand of the percent operator in the line indicated is zero. Correct the source code before you recompile it.
**line no. structure operation not implemented	The operation you attempted with a structure in the line indicated is illegal. Correct the source code before you recompile it.
**line no. usage: c168 icode asm [-DLmec]	The command line syntax is incorrect. The correct command line syntax is given in the error message. Correct the syntax before you reenter the command line.

D.2.2 Internal Logic Errors

The following messages indicate fatal errors in the internal logic of C168:

```
**line no. cdsize: invalid type
**line no. code skeleton error: op
**line no. hard long to register
**line no. intermediate code error
**line no. invalid initialization
**line no. invalid operator op
**line no. invalid register expression
**line no. invalid storage class sc
**line no. no code table for op
**line no. skelmatch type: stype
```

If you receive one of these messages, contact the place where you purchased your system for assistance. Provide the following information:

- Indicate the version of the operating system you are using.
- Describe your system's hardware configuration.
- Provide sufficient information to reproduce the error. Indicate which program was running at the time the error occurred. If possible, also provide a disk with a copy of the program.

D.3 CP68 Error Messages

The CP/M-68K C Preprocessor, CP68, returns two types of fatal error messages: diagnostic error messages and messages indicating errors in the internal logic of CP68. Both types of error messages take the general form:

```
# line no. error message text
```

The pound sign (#) indicates that the error message comes from CP68. The "error message text" describes the error.

D.3.1 Diagnostic Error Messages

A fatal diagnostic error message prevents CP68 from processing your file. The CP68 diagnostic error messages are listed in Table D-3 with explanations and suggested user responses.

Table D-3. CP68 Diagnostic Error Messages

Message	*Meaning*
`#line no. argument buffer overflow`	An argument list in the line indicated contains too many characters for the space allocated to the argument buffer. Reduce the number of characters in the argument list before rerunning CP68.
`#line no. bad argument: arg`	In the line indicated, the argument represented by the variable `arg` contains an invalid character. Replace or eliminate the invalid character before rerunning CP68.

Table D-3. (continued)

Message	*Meaning*
`*line no. bad character octal no.`	The line indicated contains an illegal character. The ASCII code of the invalid character is represented by the variable `octal no.` Examine the line indicated to locate the error. Replace the character before rerunning CP68.
`*line no. bad define name: name`	The name indicated by the variable `name` contains one or more invalid characters. Examine the name to locate the error. Replace the invalid characters before rerunning CP68.
`*line no. bad include file`	The syntax of the `*include` statement is incorrect. The `*include` statement must follow one of the following two formats: #include <filename> #include "filename" Rewrite the statement before rerunning CP68.
`*line no. bad include file name`	In the line indicated, the filename in the `*include` statement contains either an invalid character or more than 8 characters, the maximum allowed. Supply a valid filename before rerunning CP68.
`*line no. can't open fname`	The `*include` statement in the line indicated contains an invalid or nonexistent filename. Check the filename before rerunning CP68.
`*line no. can't open infile`	CP68 cannot open the input file indicated by the variable `infile`. Either the drive code or the filename is incorrect. Check the drive code and the filename before rerunning CP68.
`*line no. can't open outfile`	CP68 cannot open the output file indicated by the variable `outfile`. Either the drive code is incorrect, or the disk to which CP68 is writing is full. Check the drive code. If it is correct, the file is full. Erase unnecessary files, if any, or insert a new disk before rerunning CP68.

Table D-3. (continued)

Message	*Meaning*
`*line no. condition stack overflow`	The source code contains too many nested #if's for the space allocated to the condition stack. The stack overflowed before the line indicated. Rewrite the source code before rerunning CP68.
`*line no. define recursion`	A name or variable on the line indicated has been defined in terms of itself. Redefine the name before rerunning CP68.
`*line no. define table overflow`	The source code contains one or a combination of the following: too many names, too many long names, too many expressions, or too many large expressions. The space allocated to the define table was filled before the line indicated. Simplify and rewrite the source code before rerunning CP68.
`*line no. expression operator stack overflow`	An expression in the line indicated contains too many operations for the space allocated to the expression operator stack. Eliminate or consolidate some operations before rerunning CP68.
`*line no. expression stack overflow`	An expression in the line indicated contains too many terms for the space allocated to the expression stack. Eliminate or consolidate some terms before rerunning CP68.
`*line no. expression syntax`	The syntax of an expression in the line indicated is incorrect. Examine the line to locate the error. Correct the syntax before rerunning CP68.
`*line no. includes nested too deeply`	The `*include` statement in the line indicated contains more than 7 nested include files, the maximum allowed. Rewrite the source code so that no one `*include` statement contains more than 7 nested include files.

Table D-3. (continued)

Message	*Meaning*
`*line no. invalid #else`	A `#else` statement occurs in the source code without a preceding `#if` statement. Supply the missing `#if` statement or eliminate the `#else` statement before rerunning CP68.
`*line no. invalid #endif`	A `#endif` statement occurs in the source code without a preceding `#if` statement. Supply the missing `#if` statement or eliminate the `#endif` statement before rerunning CP68.
`*line no. invalid preprocessor command`	The command in the line indicated is either not valid for CP68 or is incorrectly formatted. Correct the command before rerunning CP68.
`*line no. line overflow`	The line indicated contains more than 255 characters, the maximum allowed. Reduce the line to no more than 255 characters before rerunning CP68.
`*line no. macro argument too long`	An argument name in the line indicated contains more than 8 characters, the maximum allowed. Use no more than 8 characters for the argument name, and rerun CP68.
`*line no. no */ before EOF`	A comment in the source code is missing the closing */. Supply the missing */ before rerunning CP68.
`*line no. string cannot cross line`	A string in the line indicated is missing a closing quotation mark. Supply the missing quotation mark before rerunning CP68.
`*line no. string too long`	The line indicated contains a string greater than 255 characters, the maximum allowed. Shorten the string to no more than 255 characters before rerunning CP68.

Table D-3. (continued)

Message	*Meaning*
`*line no. symbol table overflow`	The source code uses too many symbols for the space allocated to the symbol table. The symbol table was filled prior to the line indicated. Eliminate some symbols before rerunning CP68.
`*line no. too many arguments`	One of the names in the line indicated contains more than 9 arguments, the maximum allowed. Reduce the number of arguments to no more than 9 per name before rerunning CP68.
`*line no. unexpected EOF`	This message indicates an incomplete program. Examine the source code to locate the error. Correct before rerunning CP68.
`*line no. unmatched conditional`	A `*if` statement occurs in the source code without a matching `*endif` statement. Supply the missing `*endif` statement before rerunning CP68.
`*line no. usage: c68 [-i x:] inputfile outputfile`	This message indicates incorrect syntax in the command line. The correct syntax is given. Correct the command line before rerunning CP68. Refer to your C manual for an explanation of the command line syntax.

D.3.2 Internal Logic Errors

CP68 returns only one message indicating an error in the internal logic of CP68:

```
*line no. too many characters pushed back
```

If you receive this message, contact the place where you purchased your system for assistance. Provide the following information:

- Indicate the version of the operating system you are using.
- Describe your system's hardware configuration.
- Provide sufficient information to reproduce the error. Indicate which program was running at the time the error occurred. If possible, also provide a disk with a copy of the program.

D.4 C-Run-time Library Error Messages

The C-Run-time Library returns only one fatal error message, stack overflow. The stack overflow message means the program you are trying to include in the C-Run-time Library is too big. Reduce the size of the program.

End of Appendix D

Index