DIGITAL
RESEARCH®

C
Language
Programmer's Guide
for the CP/M-86®
Family of Operating Systems

C
Language
Programmer's Guide
for the
CP/M-86® Family
# of Operating Systems

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
TWX 910 360 5001

COPYRIGHT

Readers are granted permission to include the example programs, either in whole or in part, in their own programs.

DISCLAIMER

TRADEMARKS

The C Language Programmer's Guide for the CP/M-86 Family of Operating Systems was prepared using the Digital Research TEX™ Text Formatter and printed in the United States of America.

****
* Second Edition: October 1983 *
****

Foreword

Digital Research C™ is a full-function implementation of the standard C programming language. This implementation runs under the CP/M-86® operating system based on the Intel® 8086/8088 family of microprocessors. Unlike many other C language implementations, Digital Research C enables you to write programs that are completely portable between CP/M® and the UNIX® operating system.

The Digital Research C system consists of two executable components: the C compiler and the reverse preprocessor. The system subroutine libraries support two 8086/8088 program memory models: small and big. System libraries are compatible with UNIX Version 7. Compiler options accommodate direct control over many aspects of the compilation process and provide programmer access to useful compiler generated listings and interlistings. An extensive error warning and reporting system expedites program development with explicit diagnostic messages. Programmer's utilities for use with the Digital Research C system include the LINK-86™ linkage editor, the LIB-86™ library utility, the XREF-86™ assembly language cross-reference utility program, and the RASM-86™ relocatable assembler. Three documents supply the necessary information for using the C language, C system software, and the programmer's utilities.

- Digital Research. C Language Programmer's Guide for the CP/M-86 Family of Operating Systems. Pacific Grove, California: Digital Research, 1983 (cited as Programmer's Guide).
- Digital Research. Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems. Pacific Grove, California: Digital Research, 1983 (cited as Programmer's Utilities Guide).
- Kernighan, Brian W., and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

The Programmer's Guide consists of seven sections and six appendixes. The manual provides all the information you need to operate the C system software.

- Section 1 defines the computer resources you need to use Digital Research C and the individual components that make up the C software system. A simple demonstration program helps you get your C system up and running.
- Section 2 explains how to use the C compiler. A second demonstration program provides a more detailed description of the compiling, linking, and running procedure.

- Section 3 describes each function in the C system library. A directory at the beginning of the section helps you locate specific function descriptions easily.
- Section 4 explains the use of files and other input/output conventions.
- Section 5 explains how to interface assembly routines with C modules.
- Section 6 describes internal data representations.
- Section 7 explains the use of overlays.

Appendixes include a listing of error messages, summaries of system library functions and compiler options, a useful programming style guide, and some sample C source code modules.

This programmer's guide does not attempt to describe features of the C language. The Kernighan and Ritchie manual provides both an excellent C language reference section for the experienced programmer and a tutorial introduction to help the novice programmer get started in C. The Programmer's Utilities Guide presents in-depth explanations of the Digital Research linkage editor, library utility, and relocatable assembler. Together, the three manuals provide all the information you need to use Digital Research C to its fullest potential.

Digital Research is interested in your comments on programs and documentation. Please use the Software Performance Reports and the Reader Comment Card enclosed in each product package to help us provide you with better software products.

Table of Contents

1 Getting Started with C

2 Operating C

3 C System Library

Table of Contents
(continued)

Table of Contents
(continued)

4 Input/Output Conventions

5 Assembler Routine Interfacing

6 Internal Data Representation

7 Overlays

Appendixes

Tables, Figures, and Listings

Tables

Figures

Tables, Figures, and Listings
(continued)

Listings

Section 1
Getting Started with C

1.1 System Requirements

To operate the C compiler, you must have all of the following computer resources. Memory requirements specified below are minimum values.

- 8086 or 8088 CPU running CP/M-86.
- 108K of user program area in addition to the space occupied by the operating system.
- Enough disk space to hold the compiler during compilation.
- Enough disk space to hold the temporary file that the compiler generates. Typically, the temporary space required is one-third to one-half the size of your source file plus include files.
- Enough disk space to hold the object file that the compiler creates.

1.2 Run-time Requirements

You must have all of the following computer resources to execute programs compiled with the C compiler and linked with the system subroutine libraries. Memory requirements specified below are minimum values.

- 8086 or 8088 CPU running CP/M-86
- from 10K to 32K bytes for system library modules plus space for your program code

1.3 C Components

The Digital Research C software system consists of the two executable components of the C compiler, two versions of the system subroutine library, and two sample test programs. The executable components are the C compiler and the reverse preprocessor program. The system libraries support two 8086/8088 memory models: small and big models. Refer to Section 2.4 for an explanation of memory models. The C system also provides five special purpose files that you can include in a C program with the #include directive. Table 1-1 describes all the files on your C product disks.

Programmer's utilities for use with the Digital Research C system include the LINK-86 linkage editor, the LIB-86 library utility, the XREF-86 assembly language cross-reference utility program, and the RASM-86 relocatable assembler.

Table 1-1. C System Disk Files

Component	Disk File	Description
compiler	DRC.CMD	compiler supervisory module
	DRC860.CMD	preprocessor
	DRC861.CMD	parser and code generator
	DRC862.CMD	listing/disassembly file merge utility
	DRC.ERR	error messages
	DRCRPP.CMD	reverse preprocessor program
	R.CMD	program to run parts of the compiler
#include files	STDIO.H	macro definitions for standard input and output; contains a directive that includes PORTAB.H
	PORTAB.H	macro definitions for program portability
	CTYPE.H	ASCII character classification routines
	SETJMP.H	nonlocal program jump routine
	ERRNO.H	macro definitions for the perror function
libraries	CLEARS.L86	small model system subroutine library
	CLEARL.L86	big model system subroutine library
sample programs	SAMPLE.C	C program ready to compile, link, and run
	TEST.C	C test program that ensures proper component functioning
	STARTUP.A86	Sample start-up routine
	READ.ME	Updated notes on C software and documentation. Use TYPE command to display notes.

1.4 Minimum Configuration

You must have the following four files to execute the C compiler in minimum configuration:

• DRC.CMD	compiler supervisory module
• DRC860.CMD	preprocessor
• DRC861.CMD	parser and code generator
• R.CMD	program to run other parts of the compiler

The compiler supervisory module, DRC.CMD, executes the other compiler modules in the proper sequence. DRC860.CMD is the preprocessor, and DRC861.CMD is the parser and code generator.

The compiler error messages file, DRC.ERR, and the listing/disassembly file-merge utility, DRC862.CMD, are not required during compilation. If DRC.ERR is not on-line during compilation, the compiler returns error messages by number only. DRC862.CMD merges listing and disassembly files together for a complete interlist display when you use the compiler interlist option.

The listing file contains the source code lines from your C program. The disassembly file contains compiler generated assembly code. The assembly code the compiler generates is actually an approximation of the correct 8086/88 assembly code. It is provided for debugging purposes only.

If DRC862.CMD is not on-line during compilation, the interlist option is ineffective and the compiler outputs the listing and disassembly files separately.

Following compilation, the linkage editor requires one of the system library files to create the executable program. The library files do not have to be on-line during compilation. The following diagram illustrates the minimum C system in operation.

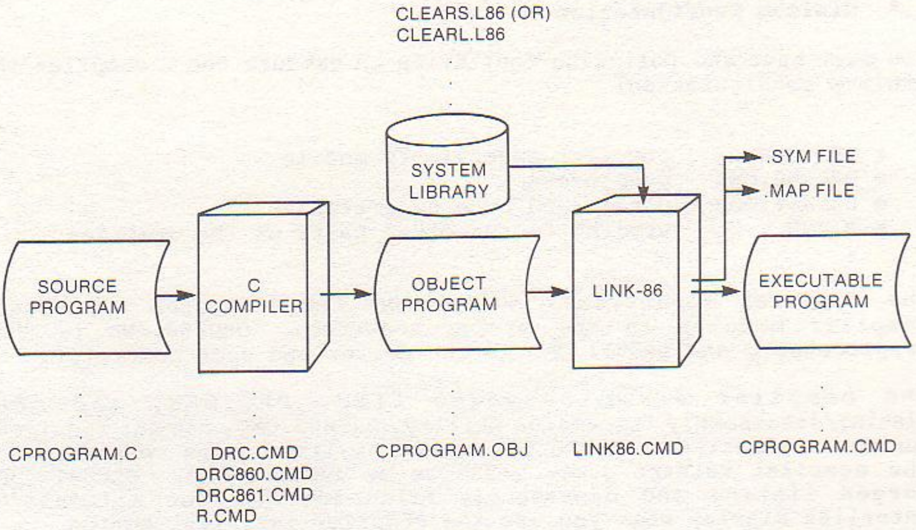


Figure 1-1. Minimum C System Operation

1.5 A Simple Demonstration

The following simple demonstration illustrates the standard procedure used to create an executable program written in C. If you are an experienced C programmer, you might want to skip this section and continue with Section 2.

The following instructions assume you already know how to use your operating system. The instructions are for C on a CP/M-86-based system with two floppy-disk drives.

First, make back-up copies of your master C product and programmer's utilities disks, and store the original disks in a safe place. Your operating system disk should be in drive A.

1) Create a C work disk.

   Using a file copy program, such as PIP, create a C work disk that contains the four compiler files required for minimum configuration, the linkage editor, the small model system library, and SAMPLE.C. If you do not have enough room on disk for all the files, you can place the linkage editor and library on a separate disk. Your work disk or disks should contain all of the following files:

DRC.CMD	compiler supervisory module
DRC860.CMD	preprocessor
DRC861.CMD	parser and code generator
R.CMD	program to run other parts of the compiler
LINK86.CMD	linkage editor
CLEARS.L86	small model system library
SAMPLE.C	sample program

With your operating system disk in drive A, place your new C work disk that contains SAMPLE.C in drive B. SAMPLE.C uses a simple for-loop and the printf function to print a short series of messages. You can display the SAMPLE.C source program on your terminal with the CP/M-86 TYPE command. Make sure drive B is the default drive and enter the following command:

```
B>TYPE SAMPLE.C
```

The following output appears on your terminal screen:

```
main()
{
     int val;

     for (val = 0; val <= 3; val++)
         printf("%d TESTING C\n", val);

     printf("\n");
     printf("FINISHED!\n");
}

B>
```

2) Compile the program.

To compile the SAMPLE.C source program, enter the following command. Be sure drive B is the default drive.

```
B>DRC SAMPLE
```

Note that you do not have to specify the .C filetype for the source program. First, the compiler searches the default drive for SAMPLE with no filetype. When the compiler cannot find SAMPLE, it automatically searches for SAMPLE.C.

The compiler displays a sign-on banner as shown in the following display. Sign-on banners might vary slightly for different versions of the compiler. Next the preprocessor and parser/code generator modules display short messages indicating execution. At the end of compilation, the compiler displays a memory allocation message. The memory allocation message indicates the amount of space that the compiler allocates for the different parts of the program. Values in the memory allocation message might vary slightly for different versions of the C compiler. Section 2.1.4 describes the different parts of the memory allocation message in more detail.

```
-----------------------------------------------------------
Digital Research C                            Version X.X
Serial No. XXXX-XXXX-XXXXXX        All Rights Reserved
Copyright (c) 1983              Digital Research, Inc.
-----------------------------------------------------------

Digital Research C  Version X.X -- Preprocessor

Digital Research C  Version X.X -- Code Gen

sample.c:      code:   67     static:   27      extern:  160

B>
```

The compiler compiles SAMPLE.C according to the 8086 small memory model by default. Refer to Section 2.4 for an explanation of memory models. The compiler then creates the relocatable object file for the program. The compiler names the object file with the same filename as the input source file but with a .OBJ filetype. This is the default object file naming convention. A directory for disk B should have the new file SAMPLE.OBJ. If you are using two separate work disks, copy SAMPLE.OBJ onto the disk that contains LINK-86 and the small model system library. Place the LINK-86 disk in drive B.

3) Link the program.

The SAMPLE.C file is compiled according to the small memory model. Therefore, you must link SAMPLE.OBJ with the small model system subroutine library.

You do not have to specify the library name explicitly in the linker command line if the library file is on the default drive. The compiler creates a special object record in SAMPLE.OBJ. This record contains information that tells LINK-86 which system library to search for any required routines. To run LINK-86, enter the following command. Be sure drive B is the default drive.

```
B>LINK86 SAMPLE
```

LINK-86 assumes a filetype of .OBJ for the object file you specify in the command line. LINK-86 displays a sign-on message and some allocation messages on your terminal as shown in the following display. Values in the allocation messages might vary slightly for programs compiled with different versions of the C compiler.

```
------------------------------------------------------
LINK-86 Linkage Editor                     Version X.X
Serial No. XXXX-XXXX-XXXXXX    All Rights Reserved
Copyright (c) 1982,1983      Digital Research, Inc.
------------------------------------------------------

CODE    02F2E
DATA    00DBC

USE FACTOR:  04%

B>
```

If you get no error messages, the program has been linked successfully. LINK-86 creates a directly executable program. The directory for disk B should have the new command file SAMPLE.CMD. Refer to Section 7 of the Programmer's Utilities Guide to learn how LINK-86 works.

4) Run the program.

To run the SAMPLE.CMD program, enter the following command. Be sure drive B is the default drive. Notice that you do not have to specify the .CMD filetype.

```
B>SAMPLE
```

The following output appears on your terminal screen:

```
0 TESTING C
1 TESTING C
2 TESTING C
3 TESTING C

FINISHED

B>
```

If your C software does not seem to operate correctly, check the system requirements listed in Section 1.1 and the run-time requirements listed in Section 1.2. Make sure your equipment complies with the specified guidelines. Section 2.5 provides a more detailed explanation of the compiling, linking, and running procedures.

End of Section 1

Section 2
Operating C

The Digital Research C compiler is especially suited to commercial systems and applications development. Enhanced diagnostic features, such as compiler information message display, error reporting, and a listing/disassembly file-merge utility, provide expanded visibility of compiler-generated information to simplify debugging and program maintenance.

2.1 Compiler Operation

To use the full C compiler configuration, the following six files must be on-line:

• DRC.CMD	compiler supervisory module
• DRC860.CMD	preprocessor
• DRC861.CMD	parser and code generator
• DRC862.CMD	listing/disassembly file-merge utility
• R.CMD	program loader utility
• DRC.ERR	compiler error messages

You can place DRC860.CMD, DRC861.CMD, and DRC862.CMD on different drives for space considerations using the -0, -1, and - 2 compiler command line options, respectively. Command line options are described in Section 2.1.3. DRC.CMD and DRC.ERR must both be on the default drive. Your source program file can be on any logical drive.

The compiler takes a C source program as input and generates an object program in the Intel relocatable object file format. During compilation, the compiler creates temporary work files named CTEMP.TOK and COBJ.TMP. Unless compilation is unsuccessful, you never see a temporary file listed in a directory. The compiler erases the files automatically when compilation is finished.

The size of a temporary file varies with the size of your source program. The total amount of temporary space required during compilation is approximately one-third to one-half the size of your source file or files. If you do not have enough work space on disk for the compiler, you can break up large programs into modules and compile each module separately.

2.1.1 Compiler Command Lines

The command line invokes the compiler, specifies the source file to compile, and passes special instructions to the compiler in the form of command line compiler options. A command line cannot exceed 128 characters. Compiler command lines use the following general format:

```
DRC  source file   option switches
```

Note that you do not have to specify the .C filetype explicitly for the source program in the command line. The compiler assumes a .C filetype unless you specify otherwise.

2.1.2 Stopping the Compiler

To stop the compiler during processing, press any console key. The compiler displays the following message:

```
Stop DRC (Y/N)?
```

Type a lowercase or uppercase Y to stop processing. The compiler immediately returns control to the operating system. If you type any character except Y, the compiler resumes processing.

2.1.3 Compiler Command Line Options

Command line option switches are reserved characters (letters and digits) that send special instructions to the compiler. An option switch specification consists of a dash followed by the reserved character. You cannot place spaces between the dash and the reserved character. However, you must place at least one space between each dash/character combination that you use in a command line.

Notice that certain option switches require an additional parameter. You cannot place spaces between the option character and the parameter. Under CP/M-86, you can enter command line option switches in lowercase or uppercase and you can place option switches anywhere in a command line. For example, the following three command line examples produce the same results.

```
B>DRC PROGRAM.C -OPROGBIG.OBJ -B -F -H
B>drc -b -f -h -oprogbig.obj program
B>DRC -B -OPROGBIG.OBJ PROGRAM -f -h
```

The rest of Section 2.1.3 describes the command line option switches in alphabetical order. Table 2-1 is a summarized description of the option switches listed alphabetically.

Table 2-1. Compiler Command Line Options

Option	Description
-a|files|	Invoke LINK-86 automatically. "files" are the object files and libraries to link. Specify the filename and [I] for a LINK-86 command line input file.
-b	Enable big memory model. (Default is small model.)
-d|name|	Define "name" as the value 1. Works like #define in the source code, but defines names in lowercase only.
-f	Use 8087 math coprocessor.
-h	Suppress sign-on banner.
-i|drive:|	Search specified disk drive for #include files.
-j	Disable short/long jump optimizer.
-l|name|	Generate program listing. Send listing to "name". (Default "name" is CON:).
-n	Disable code optimizer for faster compilation.
-o|filename|	Specify name for object file. If the filename does not contain a period, ".OBJ" will be appended.
-p	Execute preprocessor module only. Place output in file CTEMP.TOK.
-q|number|	Set number of code generator nodes to save space in symbol table. (Default is 500; minimum is 100.)
-r|name|	Request program interlisting (reverse assembly). Send interlisting to "name". (Default "name" is CON:).
-v|number|	Set compiler message display level. Should appear before other switches in command line. "number" can range from 1 to 5 to produce the following information: -v1 Display general information messages only.

Table 2-1. (continued)

Option	Description
	-v2 Display a # character as compiler processes each function. -v3 Display function name as compiler processes each function. -v4 Display start/end messages for #include files. -v5 Display filename and line number as compiler processes each line.
-w|number|	Set error message display level. "number" can be 0, 1, or 2. Default is -w0. -w0 Display all error messages. -w1 Suppress error warning messages. -w2 Suppress all error messages.
-x	Call an assembly routine to save and restore registers rather than generate code to do it in-line. Program compiles smaller but runs slower. Use with small model only.
-z|drive:|	Place temporary work files on specified disk drive.
-0|drive:|	Specify location of compiler preprocessor module (DRC860.CMD).
-1|drive:|	Specify location of compiler parser and code generator module (DRC861.CMD).
-2|drive:|	Specify location of compiler listing/disassembly file merge utility (DRC862.CMD).
-3|drive:|	Specify location of LINK-86 (LINK86.CMD).

-a option

The -a option switch executes LINK-86 automatically at the end of compilation. You must specify object files and any libraries other than the system library after the -a in the compiler command line. Alternatively, you can specify a LINK-86 input file using the INPUT option. Refer to Section 7.11, "Command Input File Options," in the Programmer's Utilities Guide for more information on the LINK-86 INPUT option.

The following command line example compiles a program named PROGRAM.C and automatically links the object file that the compiler creates with the small model system library. Notice that you do not have to specify the object file name or library file name explicitly after the -A if that object file is the only file to be linked with the system library on the default drive.

```
B>DRC PROGRAM -A
```

In this example, the compiler, LINK-86, and the small model system library are all on the default drive (B:). You can use the -3 option switch to specify a drive other than the default drive for LINK-86.

The compiler compiles PROGRAM.C according to the small memory model and names the object file PROGRAM.OBJ both by default. The compiler creates a special object record in PROGRAM.OBJ that tells LINK-86 which system library to search for required routines depending on which memory model you specify for compilation. Note that the appropriate system library file must be on the default drive. Otherwise, LINK-86 displays the NO FILE error message, indicating that you must specify the library and drive location explicitly. The object record automatically specifies the LINK-86 SEARCH option for library files. Therefore, LINK-86 only links modules from the system library that are referenced in PROGRAM.C. Without the SEARCH option, LINK-86 links in the entire system library, making the executable program unnecessarily large. The preceding example creates an executable program named PROGRAM.CMD.

To link multiple object files and libraries, you must specify each filename explicitly after the -A, including the name of the object file that the compiler creates. Use commas to separate each filespec after the -A in the command line. The following example compiles the program named PROGRAM.C, then links the object file that the compiler creates with an object file named PROGTWO.OBJ and the small model system library. Notice that you do not have to specify the .OBJ filetype explicitly for the object files after the -A.

```
B>DRC PROGRAM -APROGRAM,PROGTWO
```

The next example is identical to the first example, except the small model system library is on the D drive. Notice that you must specify the object file that the compiler creates explicitly after the -A whenever you have additional explicit filespecs.

```
B>DRC PROGRAM -APROGRAM,D:CLEARL.L86[S]
```

Remember, in this case, you must specify the object file that the compiler creates, the library file drive location, the library filename, and the LINK-86 SEARCH option explicitly.

The last example is exactly the same as the first example, except LINK-86 is on drive D. In this case, the system library is on the default drive. You do not have to specify the object file and library file explicitly.

```
B>DRC PROGRAM -A -3D:
```

Remember, a compiler command line cannot exceed 128 characters.

-b option

The -b option switch enables compilation according to the big memory model. (Refer to Section 2.4 for a description of memory models.) For example, the following command line compiles PROGRAM.C according to the big model:

```
B>DRC PROGRAM -B
```

The compiler creates the object file with the same filename as the input source file, but with an .OBJ filetype. This is the default object file naming convention. In this example, the filename is PROGRAM. You should rename PROGRAM.OBJ to better identify the file as a big model object file. A name such as PROGBIG.OBJ is easier to identify. Alternatively, you can use the -o option to rename the object file at compile time as shown in the following example:

```
B>DRC PROGRAM -OPROGBIG.OBJ -B
```

-d option

The -d option switch works like a #define in the source code. However, any name that you specify after the -d in the command line equates to the value 1. For example, the following command line directs the compiler to equate the name factor with the value 1 in the source file PROGRAM.C.

```
B>DRC PROGRAM -Dfactor
```

The next example directs the compiler to equate the name ref_22 with the value 1 in the source file PROGRAM.C.

```
B>DRC PROGRAM -Dref_22
```

-f option

The -f option switch directs the compiler to use the Intel 8087 math coprocessor for floating-point arithmetic as shown in the following example:

```
B>DRC PROGRAM -F
```

You must have the 8087 microprocessor to use the -f option. If you do not specify -f, the compiler calls routines in the system library for floating-point math. If you execute a program compiled with the -f option on a computer that does not have an 8087 math coprocessor, the program does not execute properly.

-h option

The -h option switch directs the compiler to suppress the standard compiler sign-on banner and other compiler module sign-on messages. The compiler supervisory module, DRC.CMD, displays the sign-on banner by default. Following the banner, the preprocessor and parser/code generator modules display their sign-on messages by default as shown below. Sign-on banners might differ slightly for different versions of the compiler. The memory allocation message appears last. When you specify -h, the compiler only displays the memory allocation message.

```
-------------------------------------------------
Digital Research C                    Version X.X
Serial No. XXX-XXXX-XXXXXX   All Rights Reserved
Copyright (c) 1983        Digital Research, Inc.
-------------------------------------------------

Digital Research C  Version X.X -- Preprocessor

Digital Research C  Version X.X -- Code Gen

test.c:           code:   350 static:   695 extern:   36
```

The following command line example directs the compiler not to display the sign-on banner or module sign-on messages during the compilation of PROGRAM.C. The compiler only displays the memory allocation message.

```
B>DRC PROGRAM -H
```

-i option

The -i option switch directs the compiler to search a specified disk drive for #include files. #include files facilitate the handling of declarations and groups of #define definitions. You specify #include files with the #include directive. Refer to Chapter 4.11, "The C Preprocessor," in The C Programming Language for more information on file inclusion. The following example directs the compiler to search drive C: for #include files specified in PROGRAM.C.

```
B>DRC PROGRAM -IC:
```

-j option

The -j option switch directs the compiler to disable the short/long jump optimizer. The jump optimizer converts long program branches to short branches wherever possible in the program. The result is a smaller object file. If you disable the jump optimizer, the result is faster compilation, but a bigger object file. The following example disables the jump optimizer for the compilation of PROGRAM.C.

```
B>DRC PROGRAM -J
```

-l option

The -l option switch directs the compiler to generate a listing of the source program. The preprocessor module, DRC860.CMD, actually generates the listing. You can specify a device name to which to send the listing. The compiler sends the listing to the console (CON:) by default. The following example directs the compiler to send the listing to the printer (LST:).

```
B>DRC PROGRAM -LLST:
```

-n option

The -n option switch directs the compiler to disable the code optimizer as shown in the following example:

```
B>DRC PROGRAM -N
```

If you use the -n option, the result is faster compilation, but a bigger object file.

-o option

Use the -o option switch to specify a name for the object file the compiler creates. The compiler creates the object file with the same filename as the input source file, but with an .OBJ filetype. This is the default object file naming convention. The following command line example compiles PROGRAM.C and renames the object file ONE.OBJ.

```
B>DRC PROGRAM -OONE.OBJ
```

-p option

The -p option switch directs the compiler to only execute the preprocessor module, DRC860.CMD. The preprocessor creates a temporary work file named CTEMP.TOK. If you use the -p option, the compiler stops after executing the preprocessor module and leaves CTEMP.TOK on disk. The reverse preprocessor program, DRCRPP.CMD, accepts the data in CTEMP.TOK as input and generates a modified version of your original input source file. Refer to Section 2.3 for additional information on the reverse preprocessor. The following example executes the preprocessor and creates CTEMP.TOK for PROGRAM.C.

```
B>DRC PROGRAM.C -P
```

-q option

Use the -q option switch to set the number of code generator nodes for optimization. Nodes are the pieces of data the code generator uses to build assembly instructions. The code optimizer works on groups of nodes. You can set the size of these node groups.

The default number of code generator nodes is 500. Use a smaller number of nodes to save space in the symbol table. The minimum value is 100. Use a larger number of nodes to optimize larger portions of code. The following example sets the number of code generator nodes for PROGRAM.C to 1000. A value of 1000 actually provides maximum optimization.

```
B>DRC PROGRAM -Q1000
```

-r option

The -r option directs the compiler to generate a program interlisting. An interlisting is a combination of the source code lines from your C program and the compiler-generated assembly code. DRC862.CMD is the listing disassembly file merge utility module. It merges the listing and disassembly files together for a complete interlist display when you use the -r option.

The listing file contains the source code lines from your C program. The disassembly file contains compiler generated assembly code. The assembly code the compiler generates is only an approximation of 8086/88 assembly code. It is provided for debugging purposes only.

If DRC862.CMD is not on-line during compilation, the interlist option is ineffective and the compiler outputs the listing and disassembly files separately. You can use the -2 option switch to specify a drive other than the default drive for DRC862.CMD.

Following compilation, the linkage editor requires one of the system library files to create the executable program. The library files do not have to be on-line during compilation.

You can specify a device name to which to send the interlisting. The compiler sends the interlisting to the console (CON:) by default. The following example directs the compiler to generate an interlisting and sends it to the printer (LST:).

```
B>DRC PROGRAM -RLST:
```

-v option

The compiler can produce a variety of messages other than sign-on and error messages to provide general compilation information and to indicate different stages of compilation. Use the -v option to set the compiler information message display level. Specify the -v as the first option switch in a command line. The -v option does not affect the display of sign-on and error messages.

You can specify a number ranging from 1 to 5 after the -v to select the various types of information message display. If you do not specify a number as shown in the following example, the compiler assumes -v1 and displays only general information messages:

```
B>DRC PROGRAM -V
```

General information includes messages such as "Using program.obj as output file" and messages relating to other option switches. This is why you must specify the -v as the first option switch in a command line. The -v must be able to read the rest of the command line to display the appropriate messages. The -v parameters 1 through 5 produce the following compiler information messages:

-v1	Display general information messages only.
-v2	Display a # character as compiler processes each function.
-v3	Display function name as compiler processes each function.
-v4	Display start/end messages for include files.
-v5	Display filename and line number as compiler processes each line.

Each -v parameter except -v2 and -v3 operates in a hierarchical fashion. In other words, when you specify -v2, the compiler automatically activates -v1, and so on. However, the -v2 and -v3 switches are mutually exclusive. When you specify -v3, the compiler automatically activates -v1 but not -v2. Note that when you specify -v4 or -v5, the compiler activates the -v3 switch and not -v2. For example, the following command line directs the compiler to display all messages corresponding to -v5, -v4, -v3, and -v1 switches:

```
B>DRC PROGRAM -V5
```

-w option

Use the -w option switch to set the compiler error message display level. Compiler error messages can be divided into two different categories: error reports and error warnings. Error reports indicate mistakes in your source program, such as syntax errors and improper data type specifications. Error warnings effectively indicate that some error can occur if you do not take some corrective action. Refer to Section 2.1.5 for additional information on error messages.

You can specify a number ranging from 0 to 2 after the -w to select the display level. The parameters 0 through 2 produce the following results:

-w0	Display all error messages.
-w1	Suppress error warning messages.
-w2	Suppress all error messages.

For example, the following command line directs the compiler to display only error reports:

```
B>DRC PROGRAM -W1
```

-x option

Use the -x option with the small memory model to call special assembly routines from the system library that save and restore registers for interfacing C modules and assembly routines. You cannot access these routines explicitly from a program. If you do not use -x or if you use the big memory model, the compiler generates code to save and restore registers in-line. Using -x with the small model, your program is slightly smaller but runs a little more slowly. Refer to Section 5 for more information on interfacing assembly routines with C modules. The following command line example calls the special assembly routines to save and restore registers in PROGRAM.C.

```
B>DRC PROGRAM -X
```

-z option

During compilation, the compiler creates temporary work files named CTEMP.TOK and COBJ.TMP. If you do not have enough work space on disk for the temporary files, you can use the -z option switch to place them on a specified disk drive. You specify the drive after the -z in the command line. The compiler erases the files automatically when compilation is finished. The following example directs the compiler to place the temporary files on the D: drive.

```
B>DRC PROGRAM -ZD:
```

-0 option

Use the -0 option switch to specify a drive other than the default drive for the compiler preprocessor module, DRC860.CMD. This option is handy if you do not have enough room on one disk for all the compiler modules. The following example informs the compiler supervisory module that the preprocessor is on the F: drive.

```
B>DRC PROGRAM -0F:
```

-1 option

Use the -1 option switch to specify a drive other than the default drive for the compiler parser and code generator module, DRC861.CMD. This option is handy if you do not have enough room on one disk for all the compiler modules. The following example informs the compiler supervisory module that the parser and code generator module is on the D: drive.

```
B>DRC PROGRAM -1D:
```

Note that the compiler writes the temporary files on the default drive unless you specify otherwise, using the -z option switch.

-2 option

Use the -2 option switch in conjunction with -r to specify a drive other than the default drive for the compiler listing/disassembly file merge utility, DRC862.CMD. This option is handy if you do not have enough room on one disk for all the compiler modules. The following example directs the compiler to generate a program interlisting and informs the compiler supervisory module that the listing/disassembly file merge utility is on the C: drive. The compiler sends the interlisting to the console (CON:) by default.

```
B>DRC PROGRAM -R -2C:
```

-3 option

Use the -3 option switch in conjunction with -a to specify a drive other than the default drive for the link editor, LINK86.CMD. This option is handy if you do not have enough room on one disk for both the compiler and link editor. The following command line example automatically invokes LINK-86 after compilation, but informs the compiler that LINK-86 is on the D: drive.

```
B>DRC PROGRAM -APROGRAM -3D:
```

2.1.4 Memory Allocation Data

At the end of compilation, the compiler displays a single message that indicates the amount of memory used for certain memory areas. The message appears as shown below:

```
|filename:|    code: nnnn      static: nnnn      extern: nnnn
```

The filename is the name of the input source file. The three numbers represented in the preceding example by nnnn are decimal values that indicate the number of bytes used for each memory area. The static area includes all variables specifically declared as static and all literal character strings. The external area includes all variables declared explicitly or implicitly external.

2.1.5 Error Messages

Compiler error messages can be divided into two different categories: error reports and error warnings. Error reports indicate mistakes in your source program, such as syntax errors and improper data type specifications. Error reports include messages, such as number 52, "Right parenthesis) is missing" and number 7, "Conflicting data type specified for a function."

Error warnings effectively indicate that an error can occur if you do not take some corrective action. For example, error message 83 is a warning that suggests caution using the indirection operator with integers. Error message 83, listed in Appendix C, reads as follows:

> 83 WARNING: Indirection for non-pointers is not portable.
>
> Integers can be indirected successfully in Digital Research C (small model only) and PDP-11 C. Indirection is not portable. This is an ERROR WARNING message.

If your program uses the indirection operator with integers and is configured according to the big memory model, an error can occur. Some warnings, such as number 95, "WARNING: Subscript is truncated to short int," simply inform you of a certain activity taking place during compilation.

Each compiler error message corresponds to an assigned error number. Refer to Appendix C for a summary of error messages listed in numerical order. Appendix C also provides suggestions on how to correct certain errors.

The compiler displays both types of error messages in the following format:

```
filename:  line number:  Error  number:  message text
```

The filename is the name of your input source file. The line number indicates which line in the source program contains the error. The number that follows the word Error in the message corresponds to the assigned error message number listed in Appendix C. The message text is a literal description of the error. The message text does not display if the DRC.ERR file is not on-line during compilation.

You can use compiler option switch -w to change the error message display level. You can have the compiler display all messages, suppress only the warning messages, or suppress all messages. Refer to Section 2.1.3, concerning the use of compiler option switches.

The C compiler detects a maximum of 10 errors, then aborts compilation. The compiler only displays one error message for each source code line. The compiler counts multiple errors in one source line as a single error.

2.2 Start-up Routines and Stand-alone Programs

A start-up routine controls the execution of a program. It sets up the operating environment for program execution by initializing the stack pointer, segment registers, and heap. The start-up routine is contained in the system library and linked into the executable program automatically.

The standard start-up routine in the system library is named _START. _START sets up the operating environment to execute a program under CP/M-86. After setting up the environment, _START calls the program's "main" routine for execution. The main routine in all C programs is a function named main(). The "main" routine returns control to _START at the conclusion of execution. Lastly, _START cleans up the environment by flushing buffers, closing files, freeing storage, and returning control to the operating system.

The source file named STARTUP.A86 on one of your C product disks is an example of a start-up routine written in assembly language. Study STARTUP.A86 to learn more about start-up routines.

You can also compile and link programs intended for stand-alone execution. Stand-alone programs do not use the support services of an operating system, but interface directly with the system hardware. In other words, a stand-alone program is a systems level program such as an operating system.

A stand-alone program accesses certain machine support subroutines in the system library, such as the long divide and long shift routines. You cannot access machine support subroutines explicitly. The compiler generates code to access them implicitly.

To create an executable stand-alone program, object modules created with the C compiler must be linked with the appropriate system library and a start-up routine that sets up the desired target operating environment. The _START module in the system library sets up the operating environment for CP/M-86. You must write a new start-up routine for your new target environment. When linking the program, your new start-up routine module must appear first in the LINK-86 command line. As a result, LINK-86 does not link in the standard _START that is contained in the system library.

LINK-86 produces an .CMD file that is executable in your desired target environment. The following LINK-86 command line example creates a stand-alone program named PROG from the object modules MOD1.OBJ, MOD2.OBJ, and MOD3.OBJ. The file STARTUP1.OBJ contains the start-up routine, _START, specially written for the target environment. LINK-86 searches the default drive automatically for the proper system library.

```
B>LINK86 PROG=STARTUP1,MOD1,MOD2,MOD3
```

Note that the start-up module must appear first in the LINK-86 command line. The system library must always appear after the new start-up module in the command line if the system library is specified explicitly. The object modules can appear in any order.

You can specify a different drive location for the system library in the link command line. The following example links the three object modules with the big model library. The library is on the d drive.

```
B>LINK86 PROG=STARTUP1,MOD1,MOD2,MOD3,d:CLEARL.L86[S]
```

The LINK-86 search option, [S], after the library specification selects only the routines from the library that the program requires. If you omit the search option, LINK-86 links the entire library into the .CMD file, making the executable program unnecessarily large. You can use the LINK-86 MAP option to make sure the proper routines are loaded from CLEAR. See Section 7 of the Programmer's Utilities Guide for more information on LINK-86.

2.3 Reverse Preprocessor Operation

The reverse preprocessor program, DRCRPP.CMD, is useful to determine how the compiler's preprocessor module, DRC860.CMD, handles macro instruction expansions. This can be handy when the compiler reports confusing error messages pertaining to macros.

The preprocessor module creates a temporary work file during compilation named CTEMP.TOK. If you use the -p compiler option, the compiler stops after executing the preprocessor module and leaves CTEMP.TOK on disk. The reverse preprocessor program accepts the data in CTEMP.TOK as input and generates a modified version of your original compiler input file. The reverse preprocessor incorporates all #include files and expands all macro instructions to generate the modified file. Use the following command line to invoke the reverse preprocessor:

```
B>DRCRPP <CTEMP.TOK
```

The < character specifies that the input for the reverse preprocessor program comes from the CTEMP.TOK file. Refer to Section 4.4 for information on input/output redirection.

2.4 Memory Models

The 8086/8088 microprocessor can address up to one million bytes of memory. Each address in memory points to one of three different memory areas: program code, data, or stack. For each of the memory areas, the 8086/8088 has a segment base register that points to the base address of the corresponding area in memory.

- The code segment register (CS) points to the base of the program code.
- The data segment register (DS) points to the base of an available data area.
- The stack segment register (SS) points to the base of the stack area.
- The extra segment register (ES) points to the base of another area, most often the heap.

C programs can have varying amounts of code, data, stack, and heap. Memory models determine the size of the different areas and the initial values for segment registers. For example, a memory model called the small model supports separate code and data segments each limited to 64K bytes. The C compiler supports two different memory models providing a wide range of program configurations that take full advantage of the 8086/8088 microprocessor architecture.

- small
- big

For a more complete understanding of memory models, read Section 7 in the Programmer's Utilities Guide on LINK-86 first. Section 7.5 in the utilities guide explains how LINK-86 combines the different program segments into groups and positions them in the executable .CMD file. Section 7.5.2 in the Programmer's Utilities Guide defines the terms CGROUP (code group) and DGROUP (data group).

2.4.1 Small Memory Model

The C compiler compiles all programs according to the small memory model by default. The small model defines a separate code group (CGROUP) and data group (DGROUP). Neither group can exceed 64K bytes. The C compiler automatically generates the group names CGROUP and DGROUP. For assembly language modules, use the RASM-86 GROUP directive to place segments into the proper group. Refer to Section 3.3, "The GROUP Directive," in the Programmer's Utilities Guide for more information.

LINK-86 places all segments belonging to the CGROUP in the code section of the .CMD file and all segments belonging to the DGROUP in the data section of the .CMD file. All data segments, including all common segments allocated with external variables are located together in low memory within the DGROUP, as shown in Figure 2-1. A dynamically allocated data area called the heap grows up in memory towards the stack. The heap is positioned on top of the data segments. The stack grows from the top of the data section down towards the heap.

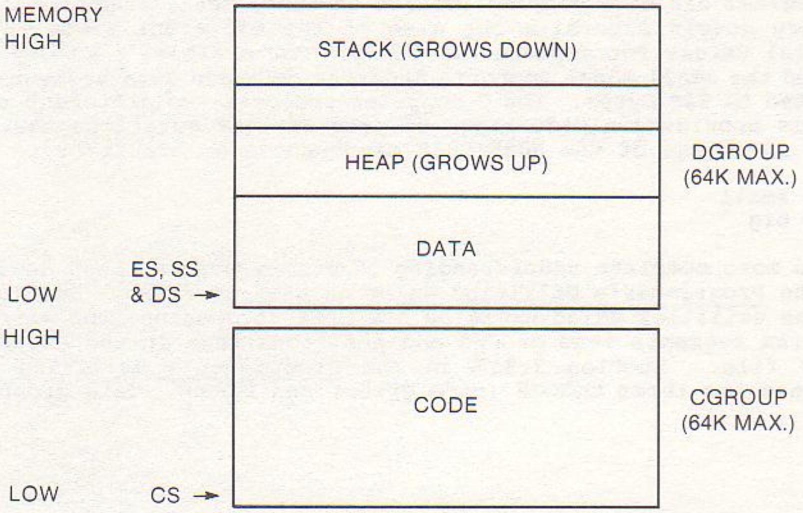


Figure 2-1. Small Memory Model

2.4.2 Big Memory Model

Use the big model for programs that use a maximum of 64K bytes of data, a maximum of 64K bytes of stack, but require a large code section and heap. To specify big model compilation, use the -b command line compiler option. All program data segments including all common segments allocated with external variables are located together within the DGROUP (data group), as shown in Figure 2-2.

The compiler does not group code segments in the CGROUP (code group). All program code segments are separate segments with a unique name. No individual code segment can exceed 64K bytes. The total amount of code is limited to the amount of available memory. Do not use the RASM-86 GROUP directive to place code segments for an assembly program into the CGROUP as you would for the small model. All code must occupy separate segments with a unique name.

The stack occupies a separate segment limited to 64K. The initial size of the stack is determined in the run-time start-up routine. The final stack size can be adjusted at link time using the LINK-86 command line options. The heap data occupies the extra segment. The heap size is limited only by the amount of available memory and is adjustable at link time.

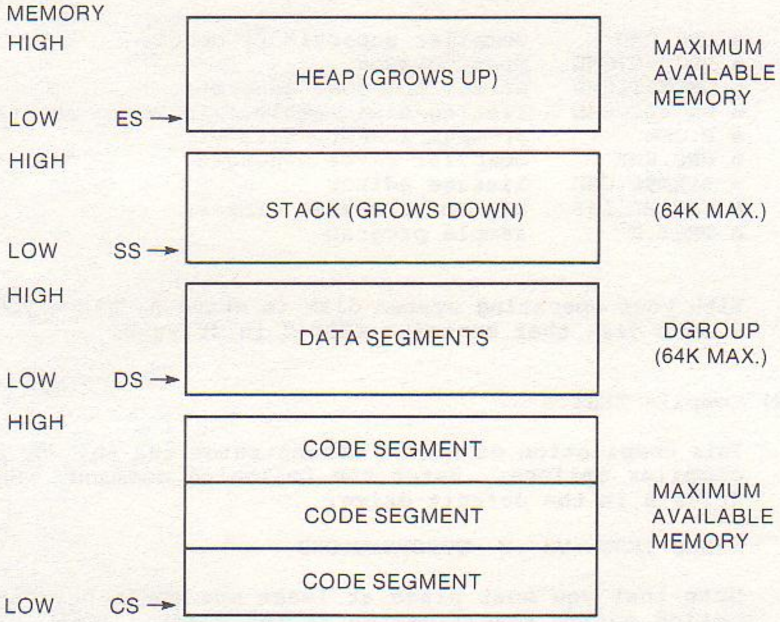


Figure 2-2. Big Memory Model

2.5 Compiling, Linking, and Running TEST.C

The TEST.C program on your C product disks serves two purposes. First, TEST.C is a C language source program that demonstrates how to compile, link, and run a program in greater detail than the simple demonstration in Section 1. Second, the program tests the compiler, linker, and libraries with simple math routines to ensure proper functioning of each component.

Be sure to make a copy of your C product and programmer's utilities disks. Store the original product disks in a safe place. You should already be familiar with your operating system and file copy program. The following instructions are for a CP/M-86-based system with two floppy-disk drives.

1) Create a C work disk.

Using a file copy program, such as PIP, create a C work disk that contains the five compiler files, the linkage editor, the big model subroutine library, and TEST.C. If you do not have enough room on disk for all the files, you can place the linkage editor and library on a separate disk. Your work disk or disks should contain all the following files:

• DRC.CMD	compiler supervisory module
• DRC860.CMD	preprocessor
• DRC861.CMD	parser and code generator
• DRC862.CMD	listing/disassembly file merge utility
• R.CMD	program loader utility
• DRC.ERR	compiler error messages
• LINK86.CMD	linkage editor
• CLEARL.L86	big model system library
• TEST.C	sample program

With your operating system disk in drive A, place your new C work disk that contains TEST.C in drive B.

2) Compile TEST.C

This compilation of TEST.C demonstrates the -b, -o, and -v compiler options. Enter the following command. Be sure drive B is the default drive.

```
B>DRC TEST -V3 -B -OTESTBIG.OBJ
```

Note that you must place at least one space between each option switch specification in the command line. The -b option switch directs the compiler to compile TEST.C according to the big memory model. The -v3 option switch tells the compiler to display the name of each program module and function as the compiler processes it. Remember, -v3 automatically activates -v1. Therefore, the compiler also displays general information messages.

However, -v2 and -v3 are mutually exclusive. Refer to Section 2.1.3 for more information on option switches. The -o option tells the compiler to name the object file TESTBIG.OBJ to better identify the file as a big model object file.

Note that you do not have to specify the .C filetype for the source program explicitly in the command line. If you do not specify a filetype, the compiler first searches for the filename with no filetype. If the compiler cannot find the filename with no filetype, it automatically searches for that filename with a .C filetype. If the compiler cannot find the filename with a .C filetype, it prints the message "Unable to open filename.C for output."

The following output should appear on your terminal screen:

```
--------------------------------------------------
Digital Research C                     Version X.X
Serial No. XXXX-XXXX-XXXXXX    All Rights Reserved
Copyright (c) 1983         Digital Research, Inc.
--------------------------------------------------


Digital Research C  Version X.X -- Preprocessor

Big Computation Model enabled
Using test.obj as output file

Digital Research C  Version X.X -- Code Gen

Processing: main

test.c:     code:    350  static:    695  extern:   36

B>
```

The compiler displays the sign-on banner first. During processing, the preprocessor and the parser/code generator modules display their sign-on messages indicating execution. The compiler displays general information messages and the name of each module and function processed in the TEST.C program, as requested with the -v3 option switch. In the preceding display, "Big Computation Model enabled" and "Using test.obj as output file" are general information messages. "Processing: main" indicates that there is only one function in TEST.C named "main." The memory allocation message appears last. Section 2.1.4 describes the different parts of the memory allocation message. If you get no error messages, TEST.C has been compiled successfully. Section 2.1.5 explains error messages.

The compiler creates the relocatable object program named TESTBIG.OBJ according to the big memory model. If you are using two separate work disks, copy TESTBIG.OBJ onto the disk that contains LINK-86 and the big model library.

3) Linking TEST.OBJ

The compiler creates a special object record in TESTBIG.OBJ. The record contains information that tells LINK-86 which system library to search for any required routines. LINK-86 searches the library automatically if the library is on the default drive, as it is in this example. Enter the following command. Be sure drive B is the default drive.

```
B>LINK86 TESTBIG
```

LINK-86 assumes a .OBJ filetype for the object files in the command line unless you specify otherwise. A sign-on banner and some memory allocation messages display on your terminal as shown below. The values in the allocation messages might vary for programs compiled with different versions of the C compiler.

```
-----------------------------------------------------------
LINK86 Linkage Editor                          Version X.XX
Serial No. XXXX-XXXX-XXXXXX         All Rights Reserved
Copyright (c) 1982,1983         Digital Research, Inc.
-----------------------------------------------------------

CODE    03E88
DATA    010A8

USE FACTOR:  07%

B>
```

If you get no error messages, the program has been linked successfully. LINK-86 creates the directly executable program. A directory for disk B should have the new command file TESTBIG.CMD.

If the system library is not on the default drive, LINK-86 displays the NO FILE error message, indicating that you must specify the appropriate drive explicitly in the link command line. For example, the following command line links TESTBIG.OBJ with the big model system library on the D: drive.

```
B>LINK86 TESTBIG, D:CLEARL.L86[S]
```

The SEARCH option, [S], after the library name, tells LINK-86 to select only the routines from the library that the program requires. If you omit the SEARCH option, LINK-86 links the entire library into the .CMD file, making the executable program unnecessarily large.

A LINK-86 command line input file is handy if you want to avoid having to type a long, complicated command line over and over. Command line input files work with the LINK-86 INPUT option. Refer to Section 7.11, "Command Input File Options," in the Programmer's Utilities Guide for more information.

4) Running TESTBIG.CMD

To execute TESTBIG.CMD, enter the following command. Be sure drive B is the default drive. Notice that you do not have to specify the .CMD filetype explicitly for TESTBIG.CMD.

```
B>TESTBIG
```

The following output should appear on your terminal.

```
***********************************************
**       WELCOME TO DIGITAL RESEARCH C       **
**                                           **
**  This sample program tests the C compiler, **
**  linker, and libraries.  If the number in  **
**  parentheses matches the number to the     **
**  immediate left, each component is working **
**  properly.                                 **
***********************************************

Test        int math: 4567 * 10 = 45670 (45670)
Test long int math: 1234 * 4567 = 5635678 (5635678)
Test float     math: 1.234 + 0.001 = 1.235 (1.235)
Test double    math: 5635678.0 / 1234.0 = 4567.0 (4567.0)

Good Luck!

B>
```

You can compile, link, and run the TEST.C program according to either of the two memory models: small or big. Be sure to specify a different object filename to distinguish one model from another. You can use the -o compiler option to name the object file during compilation.

If your C software does not operate correctly, check the system requirements listed in Section 1.1 and the run-time requirements listed in Section 1.2. Make sure your equipment complies with the specified guidelines.

If you still cannot get your software to operate correctly, fill out the Software Performance Report included in your C product package. Describe your problem in detail and mail the report to the Digital Research Technical Support Center. A prompt reply will follow.

End of Section 2

Section 3
C System Library

The run-time subroutine library for use with the Digital Research C system is called CLEAR. CLEAR stands for Common Language Environment And Run-time. CLEAR is a collection of subroutines for input/output, dynamic memory allocation, system traps, and data conversion. CLEAR is configured for both 8086/8088 memory models: small and big. Refer to Section 2.4 for a description of memory models.

- CLEARS.L86 (small model version)
- CLEARL.L86 (big model version)

Both CLEAR library files are on your C product disks.

3.1 UNIX V7 Compatibility

The CLEAR system library is compatible with UNIX Version 7, allowing programs to move easily between UNIX and CP/M-86. The system library simulates many UNIX operating system calls and features. However, CLEAR does not support the following UNIX operating system calls:

- 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 in C for CP/M-86:

- 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
- signal
- sleep
- system
- ttyslot

Entry points have been added to file open and creat calls to distinguish between ASCII and binary files.

3.2 System Library Routines

This section presents the system library subroutines that you can reference explicitly in a C program. Subroutines in the system library that have one or two underscores preceding the function name or that have the function name in capital letters are not accessible directly. They are designed for access internally by other functions in the library.

The remainder of this section alphabetically lists and explains C language functions. Each explanation demonstrates proper use of the function with four categories of information:

1) Declarations: examples of proper variable type and storage class declarations

2) Calling Syntax: the proper format used to reference the function

3) Arguments: a description of the different parameters enclosed in parentheses that follow each function name

4) Returns: a description of what each function returns

In certain cases, a function may not return a value.

The declarations in this section use standard C type and storage class specifiers. However, the file PORTAB.H contains a set of variable type declaration keywords (Table 3-1) and storage class declaration keywords (Table 3-2) that you can use to ensure consistent internal representation of data types across different processors.

Declaration keywords in PORTAB.H are macro definitions specified with #define. Using standard type specifiers can be unsafe in programs designed to be portable because of variations in internal representation among different compilers. For example, an integer declared with the keyword int might be 16-bits long on one processor and 32-bits on a different processor. However, an integer declared with the macro WORD is 16-bits on any processor. The standard I/O file STDIO.H already includes PORTAB.H. Therefore, if your program does not include STDIO.H, you must include PORTAB.H explicitly to use the macros shown in Tables 3-1 and 3-2.

The specifier FILE used in this section is defined in STDIO.H. Refer to Chapter 7.6, "File Access," in The C Programming Language for additional information.

Refer to Chapter 4.11, "The C Preprocessor," in The C Programming Language for more information on file inclusion and macro substitution. The following tables show the portability macros for variable types and storage classes defined in PORTAB.H.

Table 3-1. Variable Type Macro Definitions

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

Table 3-2. Storage Class Macro Definitions

Macro	Standard Class
REG	register variable
LOCAL	auto variable
MLOCAL	module static variable
GLOBAL	global variable definition
EXTERN	global variable reference

Identifiers in C can use both uppercase and lowercase characters. However, you must type all library function names in lowercase, as shown in the calling syntax portion of each function explanation.

With some care, you can make direct calls to the operating system from a C program with the __BDOS routine. The routine uses two arguments as shown in the following syntax diagram.

```
    ret = __BDOS (arg1, arg2)
```

The first argument is the BDOS function number as defined in the operating system. The first argument has an integer data type. The second argument depends on which BDOS function you call. The data type for the second argument varies depending on the requirements of the specific function call.

The __BDOS function returns a value of character data type. If your program requires a return value other than a character, you must write your own assembly language routine that interfaces with the operating system to make the change. Refer to your operating system programmer's guide for a description of BDOS function calls.

abs Function

The abs function returns the absolute value of a number. The abs function is implemented as a macro in STDIO.H. Therefore, arguments that involve side effects might not work as expected and should not be used. For instance, a call to abs that uses the ++ operator in the argument increments the argument value twice. a = abs(*x++) increments the value x twice. Do not declare functions that are implemented as macros.

Declarations:

```
int  val;
int  ret;  /* can be any type */
```

Calling Syntax:

```
ret = abs(val);
```

Arguments:

val -- the input value can be any number

Returns:

ret -- the absolute value of val, can be any type

access Function

The access function checks whether the calling program can access a specified file. Under CP/M-86, the file is accessible if it exists.

Declarations:

```
char *name;
int  mode;
int  ret, access();
```

Calling Syntax:

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

Arguments:

name -- points to a 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-86 ignores a mode value of 0.

Returns:

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

Note: CP/M-86 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 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. See Chapter 2.7, "Type Conversions," in The C Programming Language for related information.

Declarations:

```
char   *string;
int    iret, atoi();
long   lret, atol();
double fret, atof();
```

Calling Syntax:

```
iret = atoi(digit string);
lret = atol(digit string);
fret = atof(digit string);
```

Arguments:

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

Returns:

iret -- atoi returns the converted string as an integer.

lret -- atol returns the converted string as a long binary number.

fret -- atof returns the converted string as a double-precision floating-point number.

Each function returns 0 when there are no digits in the string.

Note: the atoi, atof, and atol 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 your program.

Use the brk function to set a new upper bound for the heap. The upper bound is an address called the break in UNIX terminology. A valid break address is one that does not exceed the maximum extent of the heap.

Use the sbrk function to extend the heap by an incremental number of bytes.

Declarations:

```
int   ret, brk();
char  *addr;
char  *strt, *sbrk();
int   incr;
```

Calling Syntax:

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

Arguments:

addr -- the new break address

incr -- the incremental number of bytes to extend the heap

Returns:

ret -- brk returns a 0 if successful, or a -1 if it fails.

strt -- sbrk returns a pointer that marks the beginning of the heap extension, or a -1 if it fails.

calloc, malloc, zalloc, realloc, free Functions

The malloc, zalloc, calloc, realloc, and free functions manage a block of dynamic area in memory called the heap. The heap is an area of contiguous bytes aligned on a word boundary.

The malloc (memory allocation) function allocates a word-aligned area in the heap and returns the starting address of the area. The argument to malloc is the number of bytes to allocate.

The zalloc (zero allocation) function is just like the malloc function except that it also zeros out the storage.

The calloc (chunk allocation) function allocates space for an array in the heap and returns the starting address of the array. The first argument to calloc is the number of entries in the array. The second argument is the size in bytes of each entry.

The realloc function changes the size of a previously allocated area. If possible, realloc uses free space adjacent to the original area. Otherwise, realloc allocates a new, larger area. Realloc copies the data from the old area to the new area, then frees the old area. Realloc returns a pointer to the new area.

The free function releases an area previously allocated with the allocation functions described above.

Declarations:

```
int   size, number;
char  *ret,    *addr,    *malloc(),    *zalloc(),    *calloc();
char  *realloc();
```

Calling Syntax:

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

Arguments:

```
size   -- the number of bytes to allocate
number -- the number of array elements to allocate
addr   -- points to the beginning of the allocated region
```

Returns:

ret -- the starting address of the allocated region if successful, and 0 if the function fails

chmod, chown Functions

Under UNIX, the chmod and chown system calls allow you to change the protection mode and owner ID of an existing file. CP/M-86 does not support protection mode or owner ID, so these calls have no effect. They are included for UNIX compatibility.

Declarations:

```
char  *name;
int   mode, owner, group, ret, chmod(), chown();
```

Calling Syntax:

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

Arguments:

name -- points to a null terminated filename
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, or -1 if the file does not exist

close Function

The close function terminates access to a file or device. The close function acts on files that have been accessed with the open or creat functions. You must specify a file descriptor for the close function, not a stream address. The fclose function closes stream files. See Chapter 8.3, "Open, Creat, Close, Unlink," in The C Programming Language for related information.

Declarations:

```
     int  ret, close(), fd;
```

Calling Syntax:

```
     ret = close(fd);
```

Arguments:

fd -- the file descriptor of the file to close

Returns:

ret -- 0 if the function succeeds, or -1 if the function detects an unknown file descriptor

cos, sin Functions

The cos function returns the trigonometric cosine for double-precision floating-point numbers. The sin function returns the trigonometric sine for double-precision floating-point numbers. You must express all arguments in radians.

Declarations:

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

Calling Syntax:

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

Arguments:

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

Returns:

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

Note: you can pass numbers declared as either float or double to cos and sin. If you pass a float, C will automatically convert it to a double.

creat, creata, creatb Functions

The creat, creata, and creatb functions create new disk files for regular, low-level access. All three functions return a unique number called a file descriptor. The file descriptor is a positive short integer used to identify a file in a C program. Under CP/M-86 the file descriptor can range from 0 to 15. Refer to Chapter 8.1, "File Descriptors," in The C Programming Language for more information on file descriptors.

There is no difference between creat and creata. Both functions create ASCII files. Use creatb to create binary files. Chapter 8.2, "Low Level I/O--Read and Write," in The C Programming Language has related information on the creat function.

Declarations:

```
char  *name;
int   mode;
int   fd, creat(), creata(), creatb();
```

Calling Syntax:

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

Arguments:

name -- a null-terminated filename string

mode -- the UNIX file mode, ignored by CP/M-86

Returns:

fd -- the file descriptor for the opened file or -1 if an error occurs

Note: ASCII files use a CTRL-Z to indicate the end-of-file. Binary files do not use an end-of-file marker. Therefore, CP/M-86 cannot directly detect the end of binary files. UNIX programs that use creat with binary files compile successfully, but might execute improperly.

ctype Functions

The file CTYPE.H defines a number of functions that classify ASCII characters. These functions test whether a character belongs to a certain character class. Each function returns a 0 if the classification test is false and a nonzero value if the test is true. See Chapter 7.9, "Some Miscellaneous Functions," in The C Programming Language for related information on ctype functions. The following table defines the ctype functions.

Table 3-3. ctype Functions

Function	Meaning
isalpha(c)	c is a letter
isupper(c)	c is uppercase
islower(c)	c is lowercase
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 any character less than a space character (0x20).

Declarations:

```
#include <ctype.h>

int   ret;
char  c;  /* or int c; */
```

Calling Syntax:

```
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 classify

Returns:

ret -- 0 if the classification test is false, or a nonzero value if the test is true

Note: the ctype functions are implemented as macros. Therefore, arguments that involve side effects, such as *p++, might not work as expected and should be avoided. The functions return meaningless values if arguments are not ASCII characters. Do not declare functions that are implemented as macros.

execl Function

The execl function passes control from an executing C program to another C program. You can chain any number of C programs for execution. However, once you pass control to a new program, you cannot effectively return to the original program. The new program overlays the original program in memory. Therefore, if you chain back to the original program, all data from the first execution are lost.

Specify the name of a file that contains the program to chain to and any arguments that the new program needs during execution. You must have at least one argument in addition to the filename. The argument must point to a null-terminated string that is the same as the filename string. This calling syntax procedure is based on UNIX conventions.

Declarations:

```
int  execl();
char *name, *arg1, *arg2;
```

Calling Syntax:

```
ret = execl(name, arg1, arg2, ..., NULLPTR);
```

Arguments:

name   -- a pointer to a null-terminated filename string
argX   -- pointers to null-terminated character strings
NULLPTR -- macro defined in PORTAB.H equal to 0

Returns:

ret -- -1 if the function fails

Note: if execl returns to the original program, an error has occurred. The function returns a -1 and the errno external variable is set to indicate the error. Refer to the perror function for additional information.

exit, _exit Functions

The exit function passes control to CP/M-86. An optional completion code might return. The completion code is operating system-dependent. CP/M-86 ignores the code. 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-86, without flushing buffers, closing open files, or deallocating memory. See Chapter 7.7, "Error Handling--Stderr and Exit," in The C Programming Language for related information.

Declarations:

```
int  code;
```

Calling Syntax:

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

Arguments:

code -- the optional, system-dependent completion code

Returns:

No return values

exp Function

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

Declarations:

```
double val;
double ret;
double exp();
```

Calling Syntax:

```
ret = exp(val);
```

Arguments:

val -- the exponent expressed as a double-precision 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. If you pass a float, C will automatically convert it to a double.

fabs Function

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

Declarations:

```
double ret;
double fabs();
double val;
```

Calling Syntax:

```
ret = fabs(val);
```

Arguments:

val -- a double-precision floating-point number

Returns:

ret -- the absolute value of the floating-point number

Note: you can pass numbers declared as either float or double to fabs. If you pass a float, C will automatically convert it to a double.

fclose, fflush Functions

The fclose function writes all data in a stream file to disk and closes the file. The fflush function writes all data in a stream file to disk but leaves the file open. A pointer identifies the stream to close. See Chapter 7.6, "File Access," in The C Programming Language for related information.

Declarations:

```
int  ret, fclose(), fflush();
FILE *stream;
```

Calling Syntax:

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

Arguments:

stream -- a pointer to a stream file control structure

Returns:

ret -- 0 if the function succeeds, or -1 if the function encounters a bad stream address or a write failure

feof, ferror, clearerr, fileno Functions

The feof, ferror, clearerr, fileno functions enable stream file manipulation in a system-independent manner.

Use the feof function to detect the end-of-file in a stream.

Use the ferror function to detect errors in a stream file. The clearerr function clears any error detected. This is most 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. See the fdopen function.

Declarations:

```
int  ret, fd;
int  feof(), ferror(), fileno();
FILE *stream;
```

Calling Syntax:

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

Arguments:

stream -- a pointer to a stream file control structure

Returns:

ret -- feof returns a nonzero value if the specified stream is at the end-of-file, and zero if it is not.

ret -- ferror returns a nonzero value if an error occurs in a specified stream file.

clearerr returns no value

fd -- fileno returns the file descriptor associated with the specified file.

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 are exactly the same. Both 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. If the specified file does not exist, the fopen, fopena, and fopenb functions create the file. See Chapter 7.6, "File Access," and Chapter 8.5, "Example--An Implementation of Fopen and Getc," in The C Programming Language for related information on fopen.

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 adds a stream file control structure to a file opened for regular access.

Declarations:

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

Calling Syntax:

```
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   -- a pointer to a null-terminated filename string
stream -- a pointer to a stream file control structure
access -- the access string can be one of three characters:

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

Returns:

stream -- the stream address if the function succeeds, or 0 if the function fails

Note: ASCII files use a CTRL-Z to indicate the end-of-file. Binary files do not use an end-of-file marker. Therefore, CP/M-86 cannot directly detect the end of binary files. UNIX programs that use fopen, freopen, or fdopen with binary files compile and link correctly but might execute improperly.

fread, fwrite Functions

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

fread transfers bytes from a stream file to memory. fwrite transfers bytes from memory to a stream file.

Declarations:

```
int  fread(), fwrite();
char *buff;
int  size, nitems;
FILE *stream;
```

Calling Syntax:

```
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 -- points to an open stream file

Returns:

nitems -- the number of items read or written, or 0 if an error occurs, including EOF

fseek, ftell, rewind Functions

The fseek, ftell, and rewind functions position the read/write pointer in a stream file. fseek and rewind have no effect on a console or listing device. ftell returns a meaningless value for nonfile devices.

The fseek function sets the read/write pointer to an arbitrary offset in the stream.

The rewind function sets the read/write pointer to the beginning of the stream.

The ftell function returns the present position of the read/write pointer in the stream.

Declarations:

```
int   ret, fseek(), rewind();
FILE  *stream;
long  offset, ftell();
int   ptrname;
```

Calling Syntax:

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

Arguments:

stream -- points to a stream file
offset -- a signed offset measured in bytes
ptrname -- The offset can start from one of three points:

0 from beginning of file
1 from current position
2 from end of file

Returns:

ret -- 0 if the function succeeds and -1 if it fails
offset -- current position of the pointer in the stream

Note: ASCII file seek and tell operations do not account for carriage returns. The functions ignore carriage returns. A CTRL-Z character at the end of the file is handled properly.

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. Arguments do not create side effects.

The getchar function reads a single character from the standard input. It is identical to getc(stdin) in all respects. See Chapter 7.2, "Standard Input and Output--Getchar and Putchar," and Chapter 7.6, "File Access," in The C Programming Language for related information on getc and getchar.

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

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

The getl function reads a 32-bit long integer from a stream, in 8086 byte order. No special alignment is required.

Declarations:

```
int   icharac, getchar(), fgetc();
FILE  *stream;
int   iword, getw();
long  ilong,getl();
```

Calling Syntax:

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

Arguments:

stream -- a pointer to a stream file control structure

Returns:

ichar -- the character read from the stream
iword -- the word read from the stream
ilong -- the long word read from the stream, or -1 if a read failure occurs

Note: errors that return from getchar are incompatible with UNIX prior to UNIX Version 7. Errors that return from getl or getw are valid values that might normally occur in a file. Use feof or ferror to detect an end-of-file or read error.

getpass Function

The getpass function reads a password from the console device.

The function issues a specified prompt, then reads the input response without echoing the input to the console. The function returns a pointer that points to the password, which is a null-terminated string. The string can contain eight or fewer characters.

Declarations:

```
char *prompt;
char *getpass;
char *pass;
```

Calling Syntax:

```
pass = getpass(prompt);
```

Arguments:

prompt -- a pointer to a null-terminated prompt string

Returns:

pass -- a pointer to the password

Note: the return value points to static data that is overwritten upon each call to getpass.

getpid Function

The getpid function is provided for UNIX V7 compatibility and serves no purpose under CP/M-86. Under UNIX, getpid returns a dummy process ID. Under CP/M-86, the return value is unpredictable.

Declarations:

```
int  pid, getpid();
```

Calling Syntax:

```
pid = getpid();
```

Arguments:

getpid uses no arguments

Returns:

pid -- a dummy process-ID on single-tasking operating systems

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 character count with fgets, but not with gets. This count includes the terminating null character. See Chapter 7.8, "Line Input and Output," in The C Programming Language for related information on fgets.

Declarations:

```
char *addr;
char *stg;
char *gets(), *fgets();
int  max;
FILE *stream;
```

Calling Syntax:

```
addr = gets(stg);
addr = fgets(stg, max, stream);
```

Arguments:

```
stg    -- pointer to a null terminated string
max    -- the maximum character count
stream -- points to the input stream
```

Returns:

```
addr -- the string address
```

index, rindex, strchr, strrchr Functions

The index, rindex, strchr, and strrchr functions locate a specified character in a string. index and strchr return a pointer to the first occurrence of the character. rindex and strrchr return a pointer to the last occurrence of the character. See Chapter 4.1, "Basics," in The C Programming Language for related information on index.

Declarations:

```
char charac;
char *stg;
char *ptr;
char *index(), *rindex(), *strchr(), *strrchr();
```

Calling Syntax:

```
ptr = index(stg, charac);
ptr = rindex(stg, charac);
ptr = strchr(stg, charac);
ptr = strchr(stg, charac);
```

Arguments:

stg -- pointer to a null-terminated string
charac -- the character to look for

Returns:

ptr -- the address of the specified character, or 0 if the character does not occur in the string

Note: strchr is identical to index, and strrchr is identical to rindex. They are alternate names for the same entry points, but strchr and strrchr are the preferred entry point names. index and rindex are included for compatibility with UNIX Version 7.

Under UNIX Level III, rindex has been eliminated and index is a function similar to but not quite the same as strchr. Under UNIX Level III, the second argument to index is a pointer to a null-terminated string instead of a single character argument. Plan to convert index and rindex to strchr and strrchr respectively for compatibility with later releases of UNIX.

isatty Function

A CP/M-86 program can use the isatty function to determine whether a file descriptor is attached to the CP/M-86 console device (CON:).

Declarations:

```
int  ret, isatty(), fd;
```

Calling Syntax:

```
ret = isatty(fd);
```

Arguments:

fd -- an open file descriptor

Returns:

ret -- 1 if the file descriptor is attached to CON:, and 0 if not attached to CON:

log, log10 Functions

The log function returns the natural logarithm of a double-precision floating-point number. The log10 function returns the base 10 logarithm of a double-precision floating-point number.

Declarations:

```
double val;
double ret;
double log();
double log10();
```

Calling Syntax:

```
ret = log(val);
ret = log10(val);
```

Arguments:

val -- a double-precision floating-point number

Returns:

ret -- the natural or base 10 logarithm of the double-precision floating-point number

Note: you can pass numbers declared as either float or double to log and log10. If you pass a float, C will automatically convert it to a double.

lseek, tell Functions

The lseek function positions a file referenced with a 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 with stream files. See Chapter 8.4, "Random Access--Seek and Lseek," in The C Programming Language for related information.

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

Declarations:

```
int   fd;
int   ptrname;
long  offset;
long  ret, lseek(), tell();
```

Calling Syntax:

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

Arguments:

fd -- the open file descriptor
offset -- a signed byte offset in the file
ptrname -- the offset interpretation, which can be one of three numbers:

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, or -1 if an error occurs

Note: these functions are 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 six upper- or lowercase X characters. The temporary filename overwrites these characters. If you specify no x characters in the argument string, the original filename remains unchanged. If you specify fewer than six x characters in the argument string, unpredicable results occur.

Declarations:

```
char *string;
char *mktemp();
```

Calling Syntax:

```
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 with a file descriptor. The openb function opens an existing binary file. You can open a file for reading, writing, or updating. See Chapter 8.3, "Open, Creat, Close, Unlink," in The C Programming Language for related information.

Declarations:

```
char *name;
int  mode;
int  fd, open(), opena(), openb();
```

Calling Syntax:

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

Arguments:

name -- points to a null-terminated filename string
mode -- type of access, can be one of three values:

0 - Read-Only
1 - Write-Only
2 - Read-Write (update)

Returns:

fd -- the file descriptor to access the file or -1 if the function fails

Note: ASCII files use a CTRL-Z to indicate the end-of-file. Binary files do not use an end-of-file marker. Therefore, CP/M-86 cannot directly detect the end of binary files. UNIX programs that use open with binary files compile correctly, but might execute improperly.

perror Function

The perror function writes a short message on the standard error file that describes the last operating system error to occur. The function prints a prefix string specified as a perror argument, then a colon and the error message.

The system library simulates the UNIX notion of an external variable, errno, that contains the last error to return from the operating system.

The perror function and the errno external variable report errors that occur during a CP/M-86 system call. The #include file ERRNO.H contains symbolic definitions for the errors that CP/M-86 returns. The ERRNO.H file also includes the names for all errors defined in UNIX V7. Therefore, you do not have to change programs that reference these definitions. The following table lists error numbers, symbolic names, and messages available that perror can report.

Table 3-4. perror Error Codes

Number	Name	Error Message
0	-	Error undefined on CP/M-86
1	-	Error undefined on CP/M-86
2	ENOENT	No such file
3	-	Error undefined on CP/M-86
4	-	Error undefined on CP/M-86
5	EIO	I/O error
6	-	Error undefined on CP/M-86
7	E2BIG	Arg list too long
8	-	Error undefined on CP/M-86
9	EBADF	Bad file number
10	-	Error undefined on CP/M-86
11	-	Error undefined on CP/M-86
12	ENOMEM	Not enough core
13	EACCES	Permission denied
14	-	Error undefined on CP/M-86
15	-	Error undefined on CP/M-86
16	-	Error undefined on CP/M-86
17	-	Error undefined on CP/M-86
18	-	Error undefined on CP/M-86
19	-	Error undefined on CP/M-86
20	-	Error undefined on CP/M-86
21	-	Error undefined on CP/M-86
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-86

Table 3-4. (continued)

Number	Name	Error Message
27	EFBIG	File too big
28	ENOSPC	No space left on device
29	-	Error undefined on CP/M-86
30	EROFS	Read-Only file system
31	-	Error undefined on CP/M-86
32	-	Error undefined on CP/M-86
33	-	Error undefined on CP/M-86
34	-	Error undefined on CP/M-86
35	ENODSPC	No directory space

Declarations:

```
char *s;
```

Calling Syntax:

```
perror(stg);
```

Arguments:

stg -- points to the prefix string to print

Returns:

perror does not return a value.

Note: certain error messages are defined in UNIX but not in CP/M-86.

printf, fprintf, sprintf Functions

The printf functions convert and format data for output. To reference a printf function, you specify a format string and a series of arguments to format. The format string consists of a series of conversion specifications. The number of conversion specifications in the format string must match the number of arguments that follow. Each conversion specification corresponds to one argument. The function converts and formats each argument consecutively as listed in the function reference. See the following page for more information on format strings.

The printf function outputs to the standard output file. The fprintf function outputs to an arbitrary stream file. The sprintf function outputs to a string (memory). Refer to Chapter 7.3, "Formatted Output--Printf," and Chapter 7.6, "File Access," in The C Programming Language for more details on these three functions.

Declarations:

```
int   ret, printf(), fprintf(), *sprintf();
char  *format;
FILE  *stream;
char  *string;
/* Args can be any type */
```

Calling Syntax:

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

Arguments:

format -- the format string
argX   -- the data arguments to format
stream -- points to a stream file opened for output
string -- points to a string buffer

Returns:

ret -- the number of characters output, or -1 if an error occurs

Format Strings:

A percent sign, %, in the format string indicates the start of a conversion specification. After the percent sign, you can use a combination of reserved formatting symbols, digits strings, and conversion characters to specify a particular format for the data. Conversion characters operate primarily on numeric data and must appear last in a conversion specification. If a character after the percent sign is not a reserved formatting symbol, digit, or conversion character, the function prints that character literally. Table 3-5 defines the conversion characters.

Table 3-5. Output Conversion Characters

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.

Table 3-5. (continued)

Operator	Meaning
f	Converts a float or double number to ASCII decimal and inserts in output stream. The precision string controls the number of decimal places. Default is six digits to the right of the decimal point.
e	Same as f, except number converts to scientific notation.
g	Converts float or double numbers using the d, f, or e conversion character depending on which yields the full precision with a minimum number of characters. The input value must be float or double.

You can use the following reserved formatting symbols and digit strings between the percent sign and a conversion character. Remember, the conversion character must appear last in a conversion specification.

- 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 output 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 that follows a period specifies the precision for floating-point conversion. The precision is the number of digits that follow the decimal point. An asterisk tells the function to use the value of the precision field from the next argument in the argument list.
- The character L or l specifies a 32-bit long value. For example, in a small model, where a pointer is 16 bits, you would say printf("%4x",p); but in a big model, where a pointer is 32 bits, you would say printf("%8lx",p);

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. Therefore, do not use arguments that involve side effects. Do not declare functions that are implemented as macros. The fputc function is equivalent to putc. However, fputc is not implemented as a macro.

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 functions that involve side effects with putchar. The C Programming Language, Chapter 7.6, "File Access," has related information on putc, and Chapter 7.2, "Standard Input and Output--Gerchar and Putchar," has related information on putchar.

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

The putl function outputs a 32-bit longword to a specified stream file. The bytes are output in 8086 order like the write function call.

Declarations:

```
char  charac;
FILE  *stream;
int   ret, fputc(), wrd, putw();
long  lret, putl(), lng;
```

Calling Syntax:

```
ret  = putc(charac, stream);
ret  = fputc(charac, stream);
ret  = putchar(charac);
ret  = putw(wrd, stream);
lret = putl(lng, stream);
```

Arguments:

charac -- the character to output
stream -- points to the output stream file
wrd -- the word to output
lng -- the long to output

Returns:

ret -- the word or character output, -1 indicates an output error

lret -- the long output with putl, -1 indicates 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. Neither routine copies the trailing null to the output stream.

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

The fputs function outputs the string to a specified output stream. The fputs function does not append a newline character. Chapter 7.8, "Line Input and Output," in The C Programming Language has related information on fputs.

Declarations:

```
int  ret, puts(), fputs();
char *stg;
FILE *stream;
```

Calling Syntax:

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

Arguments:

```
stg    -- the string to be output
stream -- the output stream
```

Returns:

```
ret -- the last character output or -1 if an error occurs
```

Note: the difference between puts and fputs is required for compatibility with UNIX.

qsort Function

The qsort function is a quick sort routine. You supply a vector of elements and a comparison function that compares two elements. The qsort function sorts the elements in the vector according to your comparison function.

A vector is a series of elements specified by a base address, the number of elements in the vector, and the size of each element in bytes. A call to the comparison function that you write must use the following format:

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

Your comparison function must return values according to the following criteria:

```
return value is < 0  if a < b
return value is = 0  if a = b
return value is > 0  if a > b
```

Declarations:

```
int  ret, qsort();
char *base;
int  number;
int  size;
int  compare();
```

Calling Syntax:

```
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 user written comparison function

Returns:

ret -- qsort always returns a value of 0.

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.

Declarations:

```
int  srand(), seed;
int  rnum, rand();
```

Calling Syntax:

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

Arguments:

seed -- an int random number seed

Returns:

rnum -- a random int number

read Function

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

Under CP/M-86, the most efficient reads begin and end on 128-byte boundaries.

See Chapter 8.2, "Low Level I/O--Read and Write," in The C Programming Language for related information.

Declarations:

```
int  ret, read();
int  fd;
char *buffer;
unsigned bytes;
```

Calling Syntax:

```
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, or -1 if an error occurs
```

scanf, fscanf, sscanf Functions

The scanf functions convert data for input. The functions read characters from an input source, convert them according to a format string, and store them in specified arguments. Arguments must be pointers. The functions continue to read characters until the input field width is exhausted.

To reference a scanf function, you specify the format string and a series of arguments. The format string consists of a series of conversion specifications. The number of conversion specifications in the format string must match the number of arguments that follow. Each conversion specification corresponds to one argument. See below for more information on format strings.

The scanf function reads from the standard input, fscanf reads from an open stream file, and sscanf reads from a null-terminated string. Refer to Chapter 7.4, "Formatted Input--Scanf," in The C Programming Language for related information.

Declarations:

```
char *format, *string;
int  nitems, scanf(), fscanf(), sscanf();
FILE *stream;
/* args can be pointers of any type */
```

Calling Syntax:

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

Arguments:

```
format -- the control string
argX   -- pointers to locations to store converted data
stream -- a stream file opened for input
string -- null-terminated input string
```

Returns:

```
nitems -- the number of items converted, or -1 if an I/O error
          occurs
```

Format String:

Format strings for scanf functions consist of the following items:

- Blanks, tabs, or newlines (line-feeds) that match optional white space in the input data.
- An ASCII character (not %) that matches the next character of the input stream.
- Conversion specifications, consisting of a leading percent sign, %, an optional asterisk, *, and a conversion character. The asterisk tells the function to suppress assignment of the data and skip to the next one. Conversion characters indicate the interpretation of the input field. Table 3-6 defines valid conversion characters.

Table 3-6. Input Conversion Characters

Character	Meaning
%	A single percent sign, %, 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 and stores it where the next argument points.
x	Converts a hexadecimal ASCII integer and stores it where the next argument points. Can also be used to input a pointer value. Use one %x for small model and two for big model to input both the offset and segment values.
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.

Table 3-6. (continued)

Character	Meaning
h	Converts a short integer. The corresponding argument must be a pointer to a short integer.
e or f	Converts a string to floating-point binary. The corresponding argument should be double.
[ ]	Indicates a string that is not delimited with spaces. The specified character string must be enclosed in the brackets. If the first character after the left bracket is not ^, the string is read up to the first character outside the right bracket. If the first character after the left bracket is ^, the string is read up to the first character that is in the set of charcters that remains between the brackets.

setbuf Function

The setbuf function assigns buffering to an input/output stream. Use setbuf after the stream has been opened but before it is read or written.

By using setbuf, you can specify a character array for a buffer in place of the automatically allocated buffer. If you specify the constant pointer NULL, input/output will be completely unbuffered.

Declarations:

```
int     setbuf();
FILE    *stream;
char    *buffer;
```

Calling Syntax:

```
ret = setbuf(stream, buffer);
```

Arguments:

stream -- a pointer to a stream file
buffer -- character array to serve a buffer

Returns:

ret -- 0 if the function is successful and -1 if it fails

setjmp, longjmp Functions

The setjmp and longjmp functions enable a program to execute a nonlocal GOTO. Use the setjmp function to save the program environment at a specific point in the flow of execution and to specify a return location for the longjmp call. You can then call longjmp from any point after the setjmp call.

The longjmp function simulates a return from a call to setjmp. First, longjmp returns a value to setjmp as specified in the second argument in the longjmp call. Secondly, execution continues at the instruction immediately following the setjmp call in the program.

If the function that invokes setjmp returns, the saved environment becomes invalid and longjmp cannot be used with it. The setjmp function saves the program environment in a variable of type jmp_buf. The type jmp_buf is defined in the include file setjmp.h.

Declarations:

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

Calling Syntax:

```
          .
          .
xret = setjmp(env);
          .
          .
longjmp(env, ret);
```

Arguments:

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

Returns:

xret -- 0 when setjmp is called initially, then copied from ret when longjmp is called

sqrt Function

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

Declarations:

```
double sqrt();
double val;
double ret;
```

Calling Syntax:

```
ret = sqrt(val);
```

Arguments:

val -- a double-precision floating-point number

Returns:

ret -- the square root of the specified double-precision floating-point number

Note: you can pass numbers declared as either float or double to sqrt. If you pass a float, C will automatically convert it to a double.

strcat, strncat Functions

The strcat and strncat functions concatenate strings.

The strcat function concatenates two null-terminated strings. The strncat function concatenates a null-terminated string and a specified maximum number of characters from a second string.

See Chapter 2.8, "Increment and Decrement Operators," in <u>The C Programming Language</u> for related information on the strcat function.

<u>Declarations:</u>

```
char *stgl, *stg2, *ret;
char *strcat(), *strncat();
int  max;
```

<u>Calling Syntax:</u>

```
ret = strcat(stgl, stg2);
ret = strncat(stgl, stg2, max);
```

<u>Arguments:</u>

stgl -- the first string
stg2 -- the second string, (appended to stgl)
max -- the maximum number of characters from stg2 to append to stgl

<u>Returns:</u>

ret -- points to the first string appended to the second

Note: if you use strcat(stgl,stgl), the function does not terminate and usually destroys the operating system. The end-of-string marker becomes lost. Therefore, strcat continues until it runs out of memory, including memory the operating system occupies.

strcmp, strncmp Functions

The strcmp and strncmp functions compare strings.

The strcmp function compares two null-terminated strings. strncmp limits the comparison to a specified number of characters in each string.

See Chapter 5.5, "Character Pointers and Functions," in <u>The C Programming Language</u> for related information on the strcmp function.

Declarations:

```
char  *stg1, *stg2;
int   val, strcmp(), strncmp(), max;
```

Calling Syntax:

```
val = strcmp(stg1, stg2);
val = strncmp(stg1, stg2, max);
```

Arguments:

```
stg1 -- a null-terminated string address
stg2 -- a null-terminated string address
max  -- the maximum number of characters to compare
```

Returns:

```
val  -- the number of characters:

        < 0 if stg1 < stg2
        = 0 if stg1 = stg2
        > 0 if stg1 > stg2
```

Note: different machines and compilers might 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 stops copying the source string after the null character is copied. strncpy specifies a maximum number of characters to copy. strncpy truncates or null-pads the source string depending on the specified number of characters to copy.

See Chapter 5.5, "Character Pointers and Functions," in The C Programming Language for related information on the strcpy function.

Declarations:

```
char *stgl, *stg2, *ret;
char *strcpy(), *strncpy();
int  n;
```

Calling Syntax:

```
ret = strcpy(stgl, stg2);
ret = strncpy(stgl, stg2, max);
```

Arguments:

stgl -- the destination string
stg2 -- the source string
max -- the exact number of characters to copy from the source string

Returns:

ret -- points to the first string

Note: if the source string exceeds the specified number of characters to copy, the destination string is not null-terminated.

strlen Function

The strlen function returns the length of a null-terminated string. See Chapter 2.3, "Constants," and 5.3, "Pointers and Arrays," in The C Programming Language for additional information.

Declarations:

```
     char *stg;
     int  len, strlen();
```

Calling Syntax:

```
     len = strlen(stg);
```

Arguments:

```
     stg -- points to a string
```

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. The number of bytes to swap must be an even number. See Chapter 5.2, Pointers and Function Arguments," in The C Programming Language for related information.

Declarations:

```
int   ret, swab();
char  *from, *to;
int   nbytes;
```

Calling Syntax:

```
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 -- swab always returns 0
```

tan, atan Functions

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

Declarations:

```
double val;
double ret;
double tan();
double atan();
```

Calling Syntax:

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

Arguments:

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

Returns:

ret -- the tan or arctangent expressed in radians of the argument value

Note: you can pass numbers declared as either float or double to tan and atan. If you pass a float, C will automatically convert it to a double.

toascii, tolower, toupper Functions

The toascii, tolower, and toupper functions are character conversion functions implemented as macros in the include file CTYPE.H. You must include CTYPE.H in any program that uses any of these three functions. Do not declare functions that are implemented as macros. Arguments that involve side effects might work incorrectly and should be avoided.

The tolower function converts an uppercase letter to the corresponding lowercase letter. The toupper function converts a lowercase letter to the corresponding uppercase letter. The toascii function simply turns off all bits in a character representation that are not part of a standard ASCII character. toascii is provided for compatibility with other systems.

Declarations:

```
#include <ctype.h>
int ret;
```

Calling Syntax:

```
ret = tolower(charac);
ret = toupper(charac);
ret = toascii(charac);
```

Arguments:

charac -- a single character to convert

Returns:

ret -- the converted character

Note: tolower and toupper can accept character arguments represented by integers in the range 0 to 255.

ttyname Function

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

Declarations:

```
char *name, *ttyname();
int  fd;
```

Calling Syntax:

```
name = ttyname(fd);
```

Arguments:

fd -- an open file descriptor

Returns:

name -- If the file descriptor is open and attached to the CP/M-86 console device, the function returns a pointer to the null-terminated string CON:. Otherwise, the function returns a NULL character.

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 use ungetc with EOF (-1). See Chapter 7.9, "Some Miscellaneous Functions," in The C Programming Language for related information.

Declarations:

```
char  charac;
FILE  *stream;
int   ret, ungetc();
```

Calling Syntax:

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

Arguments:

charac -- the character to push back
stream -- the stream address

Returns:

ret -- If the character is successfully pushed back, the function returns charac. If an error occurs, the function returns -1.

unlink Function

The unlink function deletes a named file from the file system. The function fails if the file is open or nonexistent. See Chapter 8.3, "Open, Creat, Close, Unlink," in The C Programming Language for related information.

Declarations:

```
int   ret, unlink();
char  *name;
```

Calling Syntax:

```
ret = unlink(name);
```

Arguments:

name -- points to a null-terminated filename

Returns:

ret -- 0 if the function succeeds, or -1 if the function fails

write Function

The write function transfers data to a file opened with a 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 has occurred.

Under CP/M-86, the most efficient writes begin and end on 128-byte boundaries.

See Chapter 8.2, "Low Level I/O--Read and Write," in The C Programming Language for related information.

Declarations:

```
int  ret, write();
int  fd;
char *buffer;
unsigned bytes;
```

Calling Syntax:

```
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, or -1 if an error occurs

Note: due to the buffering scheme used, some data might not be written to the file until the file is closed.

End of Section 3

Section 4
Input/Output Conventions

In C, all input and output is done by reading and writing files. Even peripheral devices such as your terminal are treated as files in a C program. A C program can access files in two different ways: as a regular file or as a stream file. C provides three input/output files called the standard I/O files that simplify input/output to your terminal and other common I/O sources.

The C Programming Language does not use the terms regular and stream. However, the manual provides complete descriptions of both types of file access. In this section, we refer to the appropriate chapters in The C Programming Language that contain additional information.

4.1 Regular File Access

Regular file access is considered low-level I/O because it provides no added services, such as data buffering. Regular access is a direct entry into the operating system. See Chapter 8.2, "Low Level I/O--Read and Write," in The C Programming Language for more information on low-level I/O.

To create a disk file for regular access, use the creat, creata, and creatb functions. All three functions return a unique number called a file descriptor. The file descriptor is a positive short integer used to identify the file in a C program. Under CP/M-86, the file descriptor can range from 0 to 15. Refer to Chapter 8.1, "File Descriptors," in The C Programming Language for more information on file descriptors.

Use creat and creata to create ASCII files and creatb to create binary files. CP/M-86 stores ASCII files with a carriage return and line-feed at the end of each line and a CTRL-Z character (0x1a) at the end-of-file. However, C programs under UNIX normally end lines with only a line-feed and do not mark the end-of-file. This means that for C programs under CP/M-86 to be compatible with C programs under UNIX the read and write functions for ASCII files respectively delete and insert carriage return characters. Also, the read functions for ASCII files delete the terminating CTRL-Z, and the close functions for ASCII files insert the CTRL-Z at the end-of-file. For binary files under CP/M-86, there is no automatic way to detect or mark the end-of-file. The program must keep track of the end-of-file position.

Use the open, opena, and openb functions to open existing files for regular access. All three functions return a file descriptor. You cannot use these functions to create new files.

The following list contains all the system library functions you can use for regular file access.

Functions for Regular File Access

close
creat
creata
creatb
lseek
open
opena
openb
read
tell
unlink
write

4.2 Stream File Access

Unlike regular file access, stream file access employs a form of local buffering, making single-byte I/O more efficient. Stream file access uses a 512-byte buffer, which corresponds to a physical blocksize on many peripheral devices.

A stream is identified by a pointer to a data control structure that contains all the information relevant to the stream. Refer to Chapter 7.6, "File Access," in The C Programming Language for additional information.

The following list contains all the system library functions you can use for stream file access. The page number refers to the function descriptions in Section 3.

Functions for Stream File Access

fclose
fdopen
feof
ferror
fflush
fgetc
fgets
fileno
fopen
fopena
fopenb
fprintf
fputc
fputs
fread
freopen
freopena
freopenb
fscanf
fseek
ftell
fwrite
getc
getchar
getl
gets
getw
printf
putc
putchar
putl
puts
putw
rewind
scanf
ungetc

4.3 Peripheral Devices

Peripheral devices, such as your terminal or printer, are treated as files in C. Like UNIX, peripheral devices under CP/M-86 use special names for identification in a program.

- CON: stands for a console device.
- LST: stands for a listing device.

4.4 Standard I/O Files

C provides three files that simplify I/O procedures from common sources, such as your terminal. The three files are the standard input, standard output, and standard error files. You can access these files as either regular or stream files. A C program begins execution with all three files open and initially connected to your terminal. Therefore, a program can handle terminal I/O without having to open files explicitly. As described below, you can indicate a source other than the terminal and redirect I/O with the < and > characters.

The standard I/O uses routines from the system library, CLEAR. The file STDIO.H contains certain macro definitions and variables used by the system library routines for opening, closing, reading, and writing the standard I/O files. You must include STDIO.H in any source program that references a system library function. Remember, STDIO.H already includes the portability file PORTAB.H. Table 4-1 shows the definitions in STDIO.H for the standard I/O files. You can list STDIO.H to examine the entire file.

Table 4-1. Standard I/O File Definitions

File	File Descriptor	Stream Name
standard input	0	stdin
standard output	1	stdout
standard error	2	stderr

You can redirect the flow of standard input and output from a command line using the < and > characters. Specify a filename or device after the < character to indicate an input source other than the terminal. The following example executes a file named PROG.CMD, with the standard input coming from a different file named INDAT.

```
prog <indat
```

Specify a filename or device after the > character to indicate an output destination other than the terminal. The following example executes a file named PROG.CMD with standard input coming from a different file named INDAT and standard output going to the list device:

```
prog <indat >lst:
```

Refer to The C Programming Language for more information on I/O redirection.

End of Section 4

Section 5
Assembler Routine Interfacing

RASM-86 is an 8086/8088 relocatable assembler that uses a compatible subset of the ASM-86™ assembly language. You can use RASM-86 to write assembly language programs that interface with C modules. RASM-86 generates relocatable object files compatible for linking with LINK-86. Refer to your Programmer's Utilities Guide for complete explanations of RASM-86 and LINK-86.

This section defines the conventions and guidelines you must observe to properly interface C functions with assembly language routines. Section 5.4 presents a RASM-86 routine to assemble and a sample C module that you can link with the routine. The information presented in this section is for the experienced assembly language programmer.

5.1 External Naming Conventions

Names for external functions and varibles are significant up to eight characters in Digital Research C, although you can use many more characters. The number of significant characters in external names varies for different compilers. For example, UNIX C compilers place an underscore character at the beginning of all external names, effectively making the names significant to seven characters only. For portability considerations, it is preferable to limit all external names to seven significant characters.

Digital Research C does not prefix an underscore on external names. However, certain routines in the system library have one or two underscores preceding the function name. You must not attempt to reference these routines directly with the exception of _exit. They are designed for access internally by other functions in the library.

RASM-86 converts all characters to uppercase unless you use the RASM-86 $nc switch. However, the C compiler distinguishes between uppercase and lowercase. Therefore, you must use all uppercase characters to specify assembler function names and to declare assembly routine variables in your C source code. The following examples show some proper and improper external names:

ASM_ROUTINE()	Proper function name. C compiler recognizes the name as ASM_ROUT.
asm_routine()	Improper function name. RASM-86 converts the name to uppercase unless you use the RASM-86 $nc switch.

`int CALCVALUE;`	Proper variable name. C compiler truncates the name to CALCVALU.
`int CALC_VALUE;`	Proper variable name. C compiler truncates the name to CALC_VAL.
`int calc_val;`	Improper variable name. RASM-86 converts the name to uppercase unless you use the RASM-86 $nc switch.

5.2 Calling an Assembly Routine from a C Module

Three steps are required to call an assembly language routine from a C module:

1) Declare the function names external in the C source code using the C language extern declaration. Refer to Chapter 1.10, "Scope: External Variables," in *The C Programming Language* for more information on C external declarations.

   Remember, RASM-86 converts all characters to uppercase unless you use the RASM-86 $nc switch. Therefore, function names declared in the C program must be in uppercase if you do not use the switch. For example, the following C declarations specify FUNC_1, FUNC_2, and FUNC_3 as external functions:

```
extern    int    FUNC_1();
extern    int    FUNC_2();
extern    long   FUNC_3();
              .
              .
              .
```

2) Declare the function names PUBLIC in the assembly routine using the RASM-86 PUBLIC directive. Refer to Section 3.7 in the *Programmer's Utilities Guide* for information on the PUBLIC directive.

```
PUBLIC   FUNC_1
PUBLIC   FUNC_2
PUBLIC   FUNC_3
  .
  .
  .
```

3) Call or reference the assembly routines from the C module.

5.3 Calling a C Module from an Assembly Routine

Two steps are required to call a C function from an assembly routine:

1) Declare the C function names external in the assembly routine using the RASM-86 EXTRN directive. Refer to Section 3.8 in the Programmer's Utilities Guide for information on the EXTRN directive. The following RASM-86 directive statements specify FUNC_1, FUNC_2, and FUNC_3 as external:

```
EXTRN   FUNC_1:NEAR
EXTRN   FUNC_2:NEAR
EXTRN   FUNC_3:NEAR
  .
  .
  .
```

   Notice that the functions are labeled NEAR in the preceding EXTRN directives. You must use the NEAR label for modules designed according to the 8086 small memory model. For modules designed according to the big memory model, you must use the FAR label.

2) Call the C functions from the assembly routine. Notice that you do not have to explicitly declare the C functions as public in the C code.

5.4 Argument Passing

The following conventions apply to the passing of arguments in C:

- C functions pass arguments on the hardware stack.
- The compiler places each argument on the stack reading from right to left.
- All arguments pass by value.
- Arguments that evaluate to one byte pass as a word value with the low-order byte of the word containing the one-byte value.
- Multiword values such as long integers, floats, and doubles, pass with the high-order words pushed before the low-order words.
- The called assembly routine must save and restore the contents of the SI and DI registers if the routine uses those registers.

- Under the small memory model, a pointer is a word that contains an offset value. Under the big memory model, a pointer is a double word that contains an offset value in the low-order word and a segment number in the high-order word. When a big model pointer is passed as an argument, the high-order word is pushed first.
- Calls to assembly functions under the big model use far calls and returns. Calls to assembly functions under the small model use near calls and returns.

When a C program calls a C function or an assembly routine, the compiler generates a standard entry/exit protocol that performs necessary manipulation of register contents. The extry/exit protocol is shown below:

```
FUNCTION:                    ;start of entry protocol
          PUSH    BP         ;save old frame pointer
          MOV     BP,SP      ;set up new frame pointer
          PUSH    DI         ;save register variables in DI and SI
          PUSH    SI
          SUB     SP,nnn     ;allocate any necessary local variables
                             ;end of entry protocol
                .
                . (the called function body)
                .
                                ;start of exit protocol
          LEA     SP,-4[BP]     ;reset stack pointer for pop
          POP     SI            ;restore register variables
          POP     DI
          POP     BP            ;restore frame pointer
          RET                   ;use RETF for big model
```

Figures 5-1 and 5-2 show the stack upon entry to a hypothetical assembly language function named TESTFUNC. Figure 5-1 shows the stack for the small memory model. Figure 5-2 shows the stack for the big model. TESTFUNC has six parameters to pass as shown below:

```
TESTFUNC(var_a, var_b, var_c, var_d, var_e, "greetings")
```

The variables var_a through var_e have the following type definitions:

```
int var_a;
long var_b;
char var_c;
float var_d;
double var_e;
```

STACK	OFFSET FROM REGISTER BP
CALLER BP	+0
RETURN ADDRESS	+2
VAR_A WORD VALUE	+4
VAR_B LOW WORD	+6
VAR_B HIGH WORD	+8
VAR_C WORD VALUE	+10
VAR_D WORD 0 (LOW)	+12
VAR_D WORD 1	+14
VAR_D WORD 2	+16
VAR_D WORD 3 (HIGH)	+18
VAR_E WORD 0 (LOW)	+20
VAR_E WORD 1	+22
VAR_E WORD 2	+24
VAR_E WORD 3 (HIGH)	+26
POINTER TO "greetings"	+28

Figure 5-1. Stack for Small Model

STACK	OFFSET FROM REGISTER BP
CALLER BP	+0
RETURN ADDRESS LOW WORD	+2
RETURN ADDRESS HIGH WORD	+4
VAR_A WORD VALUE	+6
VAR_B LOW WORD	+8
VAR_B HIGH WORD	+10
VAR_C WORD VALUE	+12
VAR_D WORD 0 (LOW)	+14
VAR_D WORD 1	+16
VAR_D WORD 2	+18
VAR_D WORD 3 (HIGH)	+20
VAR_E WORD 0 (LOW)	+22
VAR_E WORD 1	+24
VAR_E WORD 2	+26
VAR_E WORD 3 (HIGH)	+28
POINTER TO "greetings"	+30

Figure 5-2. Stack for Big Model

The compiler statically allocates space for the string constant "greetings", and passes a pointer to this static location as indicated in Figures 5-1 and 5-2. Note that floats always convert to double before being passed as arguments.

Remember, under the small model, a pointer is a word value. The value is an offset from the data segment base address. Under the big model, a pointer is a double word value. The high-order word is the segment base address. The low-order word is an offset from the segment base address.

Unlike most languages implemented for the 8086/8088, the calling C routine removes the arguments from the stack after returning from the called routine. The compiler generates an ADD SP,<nnn> instruction in the C program immediately following the call to the assembly routine. The <nnn> stands for the number of bytes pushed onto the stack. The instruction modifies the stack pointer, effectively removing the arguments. If you call a C routine from an assembly routine, you must modify the stack pointer explicitly in the assembly routine to remove the arguments.

5.5 Function Return Values

The values that a function returns are passed back to the calling program in certain registers. Table 5-1 shows which registers contain the return values for each C data type.

Table 5-1. Function Return Registers

Data Type	Registers
int, char, short, pointer (small model)	AX
long, float, pointer (big model)	High word in BX (segment for pointer) Low word in AX (offset for pointer)
double	High word in DX High middle word in CX Low middle word in BX Low word in AX

5.6 Accessing External Data

The C compiler places each C program variable declared explicitly or implicitly external into a data segment with the common attribute. To access external variables from an assembly module, or to define such variables in an assembly module for access from a C module, you must define each variable as a separate common data segment in the assembly module. Consequently, the variable name becomes the segment name in the assembly module. You cannot reference segment names in an assembly routine. Therefore, you must assign new variable names and allocate space for each one. Three steps are required to access external data.

1) Declare the variables that the C program is to share with the assembly routine as external variables in the C source program. For example, consider a hypothetical assembly language function call from a C program.

   ```
   ret = NEXTFUNC(VAR_1, VAR_2, VAR_3, VAR_4, VAR_5)
   ```

   NEXTFUNC and its five parameters must be declared external in the calling C program. Remember, RASM-86 converts all characters to uppercase unless you use the RASM-86 $nc switch.

   ```
   extern   float  NEXTFUNC();

   extern   int  VAR_1;
   extern   long  VAR_2;
   extern   char  VAR_3;
   extern   float  VAR_4;
   extern   double  VAR_5;
   ```

2) Declare TESTFUNC as PUBLIC in the assembly routine. Then declare each parameter as a separate data segment within the assembly routine. You cannot declare the variables PUBLIC in the assembly routine. You must declare each data segment with a RASM-86 COMMON combine type. Refer to Section 3.2.3 in the Programmer's Utilities Guide for more information on combine types.

   Once the variable names are declared as segments, you can not reference them as local variables in the assembly routine. Therefore, if you plan to reference the variables as local in the routine, you must assign different names for proper access. Also, you must allocate storage for the variables using the appropriate RASM-86 allocation directives. Refer to Sections 3.14 through 3.17 in the Programmer's Utilities Guide for information on allocation directives.

   ```
                PUBLIC TESTFUNC

   VAR_1        DSEG    COMMON
   VAR_ONE      RW 1

   VAR_2        DSEG    COMMON
   VAR_TWO      RW 2

   VAR_3        DSEG    COMMON
   VAR_THREE    RB 1

   VAR_4        DSEG    COMMON
   VAR_FOUR     RW 2
   ```

```
VAR_5       DSEG   COMMON
VAR_FIVE    RW 4
```

3) Group all the COMMON data segments together into the data group (dgroup) using the RASM-86 GROUP directive. Refer to Section 3.3 in the Programmer's Utilities Guide for more information on the GROUP directive.

```
DGROUP   GROUP   DATA, VAR.1, VAR.2, VAR.3, VAR.4, VAR.5
```

End of Section 5

Section 6
Internal Data Representation

There are four fundamental data types in the C language:

- character
- integer
- single-precision floating-point
- double-precision floating-point

This section describes the internal data representation that Digital Research C uses for each type.

6.1 Character Storage

C stores a character value as a single, 8-bit, unsigned binary number, as shown in Figure 6-1. Character values are always positive integers ranging from 0 to 255. Use the declaration keyword char to declare character data.

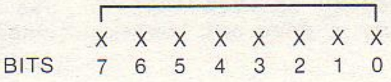


Figure 6-1. Character Storage

6.2 Integer Storage

There are two different sizes for integers: short and long. Short integers can be either signed or unsigned. C stores a short signed integer value as a 16-bit, two's complement binary number. Short signed integers range from -32768 to +32767, inclusive. You can use the keywords int or short to declare short signed integers. Use the keyword unsigned with int to declare unsigned short integers. Unsigned integers range from 0 to 65535. Figure 6-2 shows the storage format for short integers.

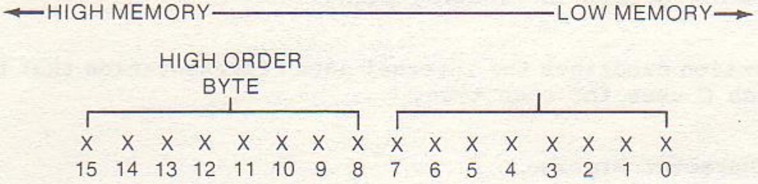


Figure 6-2. Short Integer Storage

C stores a long integer as a 32-bit, two's complement binary number, as shown in Figure 6-3. Long integers are always signed numbers ranging from -2147483648 to +2147483647. Use the keyword long to declare long integers. C does not implement unsigned long integers.

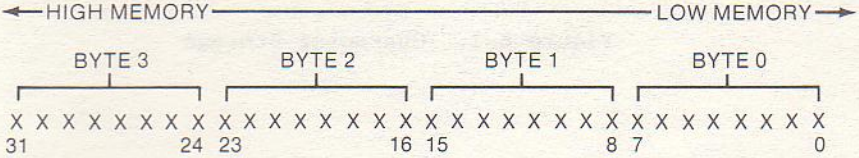


Figure 6-3. Long Integer Storage

6.3 Single-precision Floating-point

C stores a single-precision floating-point number in four consecutive bytes using the IEEE format. The 32-bits contain three fields as shown in Figure 6-4: a sign bit, an 8-bit biased exponent, and a 23-bit mantissa with a 24th implicit normalized bit.

The normalized bit is always 1 for nonzero numbers. The bit is recognized implicitly and is not stored. The binary point is situated to the immediate right of the implicit normalized bit. The exponent has a bias of 7F hexadecimal (127 decimal). Therefore, the hexadecimal number 80 represents an exponent of +1. The hexadecimal number 7E represents an exponent of -1. The mantissa is precise to 7 decimal digits. Single-precision floating-point numbers range from 1.18 times 10 to the minus 38th power up to 3.40 times 10 to the 38th power (1.18*10**-38 <= |x| <=3.40*10**38).

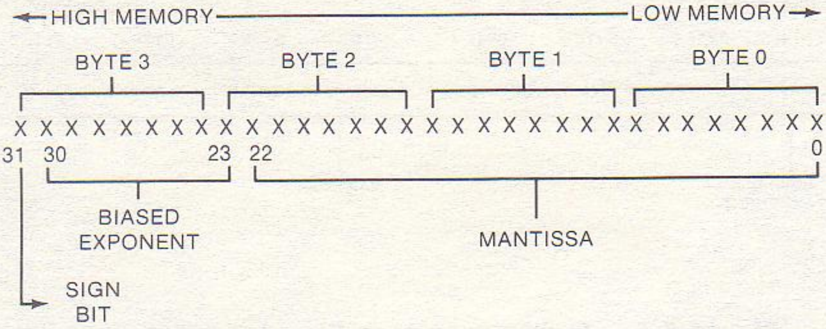


Figure 6-4. Single-precision Floating-point Storage

6.4 Double-precision Floating-point

C stores a double-precision floating-point number in eight consecutive bytes using the IEEE format. The 64-bits contain three fields: a sign bit, an 11-bit biased exponent, and a 52-bit mantissa with a 53rd implicit normalized bit.

The normalized bit is always 1 for nonzero numbers. The bit is recognized implicitly and is not stored. The binary point is situated to the immediate left of the implicit normalized bit.

The exponent has a bias of 3FF hexadecimal (1023 decimal). Therefore, the hexadecimal number 400 represents an exponent of +1. The hexadecimal number 3FE represents an exponent of -1. The mantissa is precise to 15 decimal digits. Double-precision floating-point numbers range from 9.46 times 10 to the minus 308th power up to 1.80 times 10 to the 308th power (9.46*10**-308 <= |x| <= 1.80*10**308). C performs all floating-point arithmetic in double-precision. Figure 6-5 shows the format for double-precision floating-point numbers.

When a program specifies single-precision numbers in an expression, C pads the fractional portion of those numbers with zeros, effectively lengthening the numbers to double-precision. When a double-precision number converts to single-precision, C first rounds the double-precision number before truncating it to single-precision length.

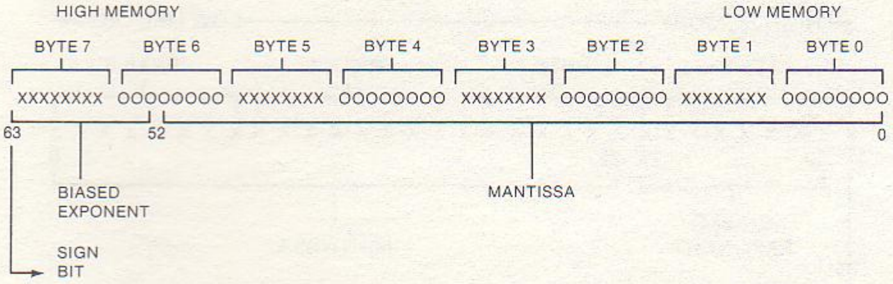


Figure 6-5. Double-precision Floating-point Storage

6.5 Pointer

In small model, a pointer is represented as a 16-bit offset. The associated segment is the data segment, in register DS. In big model, a pointer is represented as a 32-bit value: a 16-bit offset and a 16 bit segment. Also, in big model, the heap can be more than 64K bytes, but no individual allocation on the heap can be larger than 64K bytes because the offset is only 16 bits.

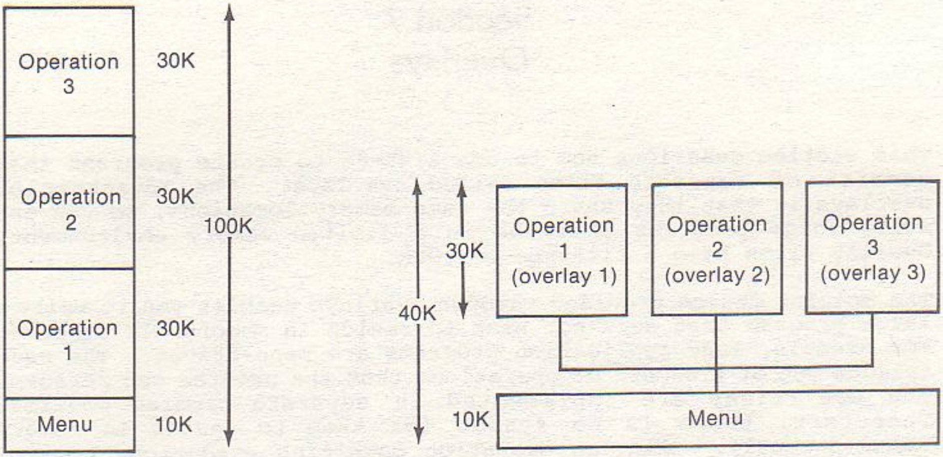
End of Section 6

Section 7
Overlays

This section describes how to use LINK-86 to create programs that consist of separate files called overlays. The advantage of overlays is that they share the same memory locations, so you can write large programs that run in a limited memory environment. Overlay files have a filetype of .OVR.

The modular design provided through overlays enables you to write a large program that does not need to reside in memory all at once. For example, many application programs are menu-driven. The user selects one of a number of operations that the program can perform. The operations are implemented in separate program modules. Therefore, there is no reason for them to reside in memory simultaneously. When an operation completes execution, control returns to the menu portion of the program. The user can then select another operation. Using overlays, you can divide such a program into separate operation subprograms, which can be stored on disk and loaded into memory only when required.

Figure 7-1 illustrates the concept of overlays. Suppose a menu-driven application program consists of three separate user-selectable operations. If each operation consists of a module that requires 30K of memory and the menu portion requires 10K, then the total memory required for the program is 100K, as shown on the left in Figure 7-1. However, if the three operation modules are designed using overlays, as shown on the right in Figure 7-1, the program requires only 40K for execution because all three functions share the same memory locations at different times.



Without Overlays **Separate Overlays**

Figure 7-1. Using Overlays in a Large Program

You can also create overlays in the form of a tree structure, where each overlay can call other overlays. Section 7.2 describes the command line syntax for creating nested overlays.

Figure 7-2 illustrates such an overlay structure. The top of the highest overlay determines the total amount of memory required. In Figure 7-2, the highest overlay is SUB4. Note that this is substantially less memory than would be required if all the operation modules and suboperation modules had to reside in memory simultaneously.

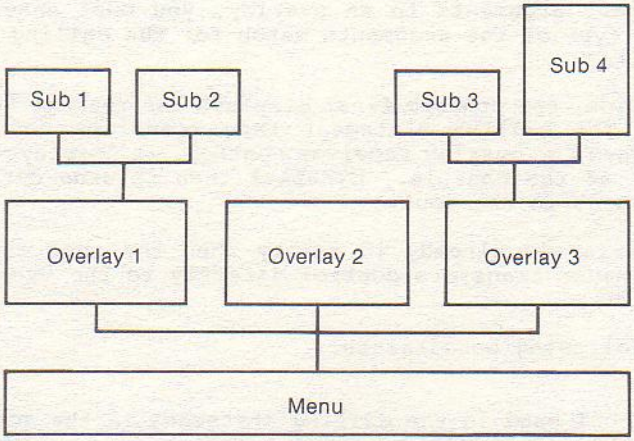


Figure 7-2. Tree Structure of Overlays

7.1 Writing Programs That Use Overlays

There are two restrictions for programs that use overlays. The first restriction is that all overlays must be on the default drive. The second restriction is that the overlay names are determined at compile time and cannot be changed at run-time.

For example, the following C source program is a root module named MAIN.C, which uses one overlay named OVERLAY1.

```
main()
{
  printf("root\n");
  overlay1("overlay1\n");
  printf("return from overlay\n");
}
```

OVERLAY1 is defined as follows:

```
overlay1(s);
char *s;
{
  printf(s);
}
```

When you pass arguments to an overlay, you must ensure that the number and type of the arguments match for the calling program and the overlay.

Upon execution, the program first displays the message "root" at the console. The calling statement then transfers control to the overlay. When the overlay receives control, it displays the message "overlay1" at the console. OVERLAY1 then returns control to the next statement in the root.

If the overlay is already in memory when the root calls it, the overlay manager transfers control directly to the overlay without reloading it.

Note the following constraints:

- The label used in the calling statement is the actual name of the .OVR file loaded by the overlay manager, so the two names must match.
- The name of the entry point to an overlay need not match the name used in the calling statement. You should use the same name to avoid confusion.
- The overlay manager loads overlays only from the drive that was the default drive when the root module began execution. The overlay manager disregards any changes to the default drive that occur after the root module begins execution.
- The names of the overlays are fixed. To change the names of the overlays, you must edit, compile, and relink the program.
- No nonstandard statements are needed in your source program. Therefore, you can postpone the decision on whether or not to create overlays until link-time.

7.2 LINK-86 Command Lines for C Overlays

You specify overlays in the LINK-86 command line by enclosing each overlay specification in parentheses. The overlay manager is included automatically from the CLEAR library.

You can specify an overlay using any of the following forms:

```
LINK86 ROOT (OVERLAY1)
LINK86 ROOT (OVERLAY1,PART2,PART3)
LINK86 ROOT (OVERLAY1=PART1,PART2,PART3)
```

The first form produces the file OVERLAY1.OVR from the file OVERLAY1.OBJ. The second form produces the file OVERLAY1.OVR from OVERLAY1.OBJ, PART2.OBJ and PART3.OBJ. The third form produces the file OVERLAY1.OVR from PART1.OBJ, PART2.OBJ and PART3.OBJ.

In the command line, a left parenthesis indicates the start of a new overlay specification and also indicates the end of the group preceding it. All files to be included at any overlay must appear together, without any intervening overlay specifications. You can use spaces to improve readability, and commas can separate parts of a single overlay. However, do not use commas to set off the overlay specifications from the root module or from each other. Also, overlays must be last on the command line.

For example, the following command line is invalid:

```
A>LINK86 ROOT(OVERLAY1),MOREROOT
```

The correct form of the command is as follows.

```
A>LINK86 ROOT,MOREROOT(OVERLAY1)
```

To nest overlays, you must specify them in the command line with nested parentheses as shown below. SUB1, SUB2, SUB3, and SUB4 are nested overlays in the following example.

```
A>LINK86 MENU,(FUNC1(SUB1)(SUB2)(FUNC2)(FUNC3(SUB3)(SUB4))
```

7.3 General Overlay Constraints

The following general constraints apply when you use LINK-86 to create overlays:

- Each overlay has only one entry point. The overlay manager assumes that this entry point is at the beginning of the overlay.
- You cannot make an upward reference from a module to an entry point in an overlay that is higher on the tree. The only exception is a reference to the main entry point of the overlay as described above. You can make downward references to entry points in overlays lower on the tree or in the root module.
- Common segments that are declared in one module cannot be initialized by a module higher in the tree. LINK-86 ignores any attempt to do so.

End of Section 7

Appendix A
System Library Routine Summary

There are two versions of the CLEAR (Common Language Environment And Run-time) library. CLEAR is configured for both 8086/8088 memory models: small and big. Refer to Section 2.4 for a description of memory models. Both CLEAR files are on your C product disks.

- CLEARS.L86 (small memory version)
- CLEARL.L86 (big memory version)

Most of the modules in the system library are accessible directly from your C program using the appropriate function names. However, certain routines in the system library are designed for access internally by other functions in the library and cannot be accessed explicitly from a program. All module names in the system library are in capital letters but function names are in lowercase. Module names might vary slightly for different versions of the system libraries.

Each function in the system libraries performs a certain task. Groups of functions fall into related categories according to what each one does. For example, one group of functions pertains to stream I/O. Another group operates on strings. There are seven general categories for system functions.

- Regular File Access Functions
- Stream File Access Functions
- String Functions
- ASCII Character Macros
- Memory Management Functions
- Double-precision Floating-point Functions
- Utility Functions

The following lists present the system library functions in the appropriate category. Refer to Section 3 for a complete description of each function. Remember, function names are in lowercase.

Regular File Access Functions

close
creat
creata
creatb
lseek
open
opena
openb
read
tell
unlink
write

Stream File Access Functions

clearerr
fclose
fdopen
feof
ferror
fflush
fgetc
fgets
fileno
fopen
fopena
fopenb
fprintf
fputc
fputs
fread
freopen
freopena
frepoenb
fscanf
fseek
ftell
fwrite
getc
getchar
getl
gets
getw
printf
putc
putchar
putl
puts
putw
rewind
scanf
setbuf
ungetc

String Functions

atof
atoi
atol
index
mktemp
rindex
sprintf
sscanf
strcat
strchr
strcmp
strcpy
strlen
strncat
strncmp
strncpy
strrchr

ASCII Character Macros

isalnum
isalpha
isascii
iscntrl
isdigit
islower
isprint
ispunct
isspace
isupper
toascii
tolower
toupper

Memory Management Functions

brk
calloc
free
malloc
realloc
sbrk
zalloc

Double-precision Floating-point Functions

atan
atof
cos
exp
fabs
log
log10
sin
sqrt
tan

Utility Functions

abs
access
__BDOS
chmod
chown
exit
_exit
getpass
getpid
isatty
longjmp
perror
qsort
rand
setjmp
srand
swab
ttyname

End of Appendix A

Appendix B
Compiler Option Summary

Command line option switches are reserved letters that send special instructions to the compiler. An option switch specification consists of a dash followed by the option letter. You cannot place spaces between the dash and the letter. However, you must place at least one space between each dash/option letter combination that you use in the command line. Option switch specifications must follow the source file in the command line. The following table lists the compiler command line option switches and a brief description of each. Notice that certain option switches require an additional parameter.

Table B-1. Compiler Command Line Options

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td>-a|files|</td><td>Invoke LINK86 automatically. |files| are the object files and libraries to link. Specify the filename and [I] for a LINK86 command line input file.</td></tr>
<tr><td>-b</td><td>Enable big memory model. (Default is small model.)</td></tr>
<tr><td>-d|name|</td><td>Define |name| as the value 1. Works like #define in the source code. Defines names in lowercase only.</td></tr>
<tr><td>-f</td><td>Use 8087 math coprocessor.</td></tr>
<tr><td>-h</td><td>Suppress sign-on messages.</td></tr>
<tr><td>-i|drive:|</td><td>Search specified disk drive for #include files.</td></tr>
<tr><td>-j</td><td>Disable short/long jump optimizer.</td></tr>
<tr><td>-l|name|</td><td>Generate program listing. Send listing to |name|. (Default |name| is CON:)</td></tr>
<tr><td>-n</td><td>Disable code optimizer for faster compilation.</td></tr>
</table>

Table B-1. (continued)

Option	Description
-o|filename|	Specify name for object file. If filename does not contain a period, .OBJ will be appended to filename.
-p	Execute preprocessor module only. Place output in file CTEMP.TOK.
-q|number|	Set number of code generator modes to save space in symbol table. (Default is 500; minimum is 100.)
-r|name|	Request program interlisting (reverse assembly). Send interlisting to |name|. (Default |name| is CON:)
-v|number|	Set compiler message display level. Should appear before other switches in command line. |number| can range from 1 to 5 to produce the following information: -v1 Display general information messages only. -v2 Display a # character as compiler processes each function. -v3 Display function name as compiler processes each function. -v4 Display start/end messages for #include files. -v5 Display filename and line number as compiler processes each line.
-w|number|	Set error message display level. |number| can be 0, 1, or 2. -w0 Display all error messages. -w1 Suppress error warning messages. -w2 Suppress all error messages.
-x	Call an assembly routine to save and restore registers rather than generate code to do it in-line. Program compiles smaller but runs slower. Use with small model only.
-z|drive:|	Place temporary work files on specified disk drive.

Table B-1. (continued)

Option	Description
-0|drive:|	Specify location of compiler preprocessor module (DRC860.CMD).
-1|drive:|	Specify location of compiler parser and code generator module (DRC861.CMD).
-2|drive:|	Specify location of compiler listing/disassembly file merge utility (DRC862.CMD).
-3|drive:|	Specify location of LINK86 (LINK86.CMD).

End of Appendix B

Appendix C
Error Messages

Compiler error messages can be divided into two different categories: error reports and error warnings. Error reports indicate mistakes in your source program, such as syntax errors and improper data type specifications. Error reports include messages such as "Right brace } is missing" and "Same statement label used more than once."

Error warnings effectively indicate that some error might occur if you do not take some corrective action. For example, error message 83 is a warning that suggests caution using the indirection operator with integers. Some warnings, such as number 95 "Subscript is truncated to short int", simply inform you of a certain activity taking place during compilation.

You can use compiler option switch -w to reset the error message display level. You can have the compiler display all messages, suppress only the warning messages, or suppress all messages. Refer to Section 2.1.3 for information on how to use compiler option switches.

All compiler error messages correspond to an assigned error number. Table C-1 presents the C error messages listed in numerical order. Each entry shows the message text, the most common cause of the error, and a suggestion for fixing the error. Error warnings are clearly distinguished from error reports in Table C-1.

Table C-1. Error Messages

Error	Meaning
1	Out of memory. An allocate function returns NULL. Compilation stops because available memory is exhausted. Reduce the number of functions compiled in a single module.
2	Identifier not specified in type or storage class declaration. The compiler read a type or storage class declaration keyword, but could not find a corresponding identifier. Correct the syntax error in the source program.

Table C-1. (continued)

Error	Meaning
3	A public function definition is declared external. Do not declare public function definitions external. Declare the public function name external using the keyword extern in any module that calls the function. Remove the keyword extern from the public function definition.
4	Parentheses () missing in function declaration. The compiler read an opening brace { indicating the start of a function body, but did not find a corresponding parameter list in the function declaration. You must place parentheses, (), after the function name in any function declaration that does not have parameters.
5	<identifier>...not declared as a function. The identifier shown in the message text was referenced as a function. The identifier has been declared, but not as a function. Change the declaration or use different names for variables and functions.
6	Two functions have the same name. The program uses one function name to identify two different functions. Change one of the function names.
7	Conflicting data type specified for a function. The compiler detects conflicting data type references for a function. This often happens when a function is declared implicitly as an integer and is declared later as returning a noninteger type. Declare the function to return the appropriate data type before its first use.

Table C-1. (continued)

Error	Meaning
8	Data type not specified in variable declaration. The compiler read a variable name that has no type or storage class declaration. Functions automatically default to an integer return value. Declare the variable name with an appropriate type, storage class, or both.
9	Global variable declared with "register" storage class. The register storage class is not meaningful for extern or static variables. Delete the register storage class in the variable declaration.
10	Conflicting data type specified for a function. The compiler detects conflicting data type references for a function. This often happens when a function is declared implicitly as an integer and is declared later as returning a noninteger type. Declare the function to return the appropriate data type before its first use.
11	Conflicting storage class keywords in declaration. The compiler read a declaration with conflicting storage classes, such as auto register or static extern. Specify only one storage class in each declaration.
12	Conflicting data type keywords in declaration. The compiler read a declaration with conflicting data types, such as float int or long float. Specify only one data type in each declaration.

Table C-1. (continued)

Error	Meaning
13	Use the keywords unsigned/long/short with int only. You can use the type qualifiers unsigned, long, and short on int data items only. Change any incorrect declarations in your source program.
14	Do not use the "unsigned long" type declaration. Some compilers permit use of the "unsigned long" type declaration. You cannot use it in Digital Research C. Delete the keyword "unsigned".
15	Conflicting type qualifiers "short/long" in declaration. You cannot use the type qualifiers short and long in the same variable or function declaration. Choose one type qualifier for each declaration.
16	Conflicting definitions for structure tag identifier. The program uses an identifier as a structure tag. That identifier is already defined as something else. Change the first declaration of the identifier or choose a different structure tag.
17	Identifier or left brace { missing in struct or union declaration. Each struct and union declaration requires either a left brace { or an identifier. Correct the syntax error in the source code.

Table C-1. (continued)

Error	Meaning
18	Storage class specified in struct or union declaration. You cannot declare elements of a struct or union with storage class keywords (static, extern, register, auto). Delete storage class keywords in all struct and union declarations.
19	Data type not specified in struct or union declaration. Each struct or union declarations must contain a data type specification. The compiler did not find one. Specify an appropriate type
20	Use integer constants to define bit field width. You can use only integers to define the width of bit fields in a struct or union. Change the constant to an integer.
21	Conflicting offsets or data type declared for <identifier>. In Digital Research C, all struct and union fields exist in the same name space. If you use the name in multiple declarations, each field must be unique or exist at the same offset with the same type.
22	A comma or semicolon is missing. A required comma or semicolon is missing. Correct the syntax error in the source code.
23	Internal compiler error. Compiler error. Contact the Digital Research Technical Support Center.

Table C-1. (continued)

Error	Meaning
24	Do not use "static" or "extern" to declare parameters. You cannot use the storage class keywords static or extern to declare parameters, because parameters pass on the stack and cannot be allocated statically. You can use auto and register in parameter declarations. Delete any incorrect storage class declarations for parameters.
25	Data type not specified in parameter declaration. You must declare a data type for each parameter. Specify an appropriate type.
26	Do not use abstract declarator in parameter declaration. You must supply a identifier to declare parameters. For example, "int ;" is incorrect. Correct the syntax error in the source code.
27	<identifier>...not specified as a parameter. The compiler read a declaration for an identifier that is not in the list of parameters. Probably a syntax or typing mistake. Correct the syntax error in your source code or move the declaration after the opening left brace.
28	<identifier>...must be pointer or scalar. You cannot declare structures as parameters in Digital Research C. Change your source program.
29	Identifier not specified in parameter declaration. Parameter declaration syntax requires an identifier. The compiler cannot find one. Correct the syntax error in the source code.

Table C-1. (continued)

Error	Meaning
30	Function body is specified as a parameter. You can declare functions as parameters but those functions cannot contain statements. Correct the syntax error in the source code.
31	Use integer constant expression to specify array bounds. You must specify the bounds of an array with a constant expression that can be reduced to an integer. Correct the syntax error in the source code.
32	<identifier>...undefined for "goto" statement. A goto statement references a label that is not defined. Correct the goto statement or define the label.
33	WARNING: <identifier> is declared, but not referenced. The identifier in the message text is declared but not referenced in the program. This is an ERROR WARNING MESSAGE.
34	<identifier>...struct or union referenced before declaration. A struct or union variable is referenced before the struct or union is declared. Define the struct or union.
35	Cannot initialize a function. You placed an equal sign after a function declaration specifying initialization. Correct the syntax error in the source code.

Table C-1. (continued)

Error	Meaning
36	Initializing variable with "extern". You can initialize variables with the extern keyword. This is allowed in Digital Research C, but not described by Kernighan and Ritchie. This is an ERROR WARNING MESSAGE.
37	Array dimensions are extended automatically. The syntax (array_name[<num>] = {...}) is used, but more data items are supplied than stated in num. The size of the array is extended. Correct the size declaration for the array.
38	Switch expression cannot be floating-point. The expression in a switch statement must be nonfloating. Correct the semantic error in the source code.
39	Cannot read switch statement. "case <const>" ignored. The compiler read the construct "case" but did not process a switch statement. The "case" construct is ignored. Correct the syntax error in the source code. Possibly missing a left brace after the switch.
40	Constant in "case" construct cannot be floating-point. The constant expression after the keyword case is not a standard C language construct and does not work in Digital Research C. Correct the semantic error in the source code.

Table C-1. (continued)

Error	Meaning
41	Cannot read switch statement. "default:" ignored. The compiler read a default: construct but did not process a switch statement. The default: construct is ignored. Correct the syntax error in the source code. Probably missing a left brace after the switch.
42	Break location undefined. No loop or switch. The compiler read a break statement but did not process a for/while loop. Probably a syntax error. Correct the source program.
43	Continue location undefined. No loop or switch. The compiler read a continue statement but did not process a for/while loop. Probably a syntax error. Correct the source program.
44	Identifier not specified in "goto" statement. An identifier is required after a goto statement. The compiler did not find the identifier. Correct the syntax error in the source code.
45	Same statement label used more than once. You used the same label more than once in a function. Correct the semantic error in the source code.
46	<identifier>...defined more than once. The same variable or function name is defined more than once in the same compilation. Correct the error in the source program.

Table C-1. (continued)

Error	Meaning
47	<identifier>...Undefined identifier. The program references an identifier before it is defined. Correct the error in the source program.
48	Unexpected end-of-file (EOF) on input file. A misplaced end-of-file is detected on the input file. Probably mismatched braces. Correct the error in the source program.
49	Comma or semicolon is missing. Syntax error. Correct the error in the source program.
50	Right brace } is missing. Syntax error. Correct the error in the source program.
51	Left brace { is missing. Syntax error. Correct the error in the source program.
52	Right parenthesis) is missing. Syntax error. Correct the error in the source program.
53	Right square bracket] is missing in array declaration. Syntax error in array declaration. Correct the error in the source program.

Table C-1. (continued)

Error	Meaning
54	Function parameters cannot have parameters. The compiler read a parameter declaration for an identifier that already exists as a parameter to a function. Correct the error in the source program.
55	Right parenthesis) is missing. Syntax error. Correct the error in the source program.
56	Do not list parameters in the function declaration. You cannot list identifiers between the parentheses in a function declaration such as int f(). List the identifiers in the function body definition. Correct the error in the source program.
57	Comma or semicolon is missing. Syntax error. Correct the error in the source program.
58	Right brace } is missing. Syntax error. Correct the error in the source program.
59	Comma or semicolon is missing. Syntax error. Correct the error in the source program.
60	Semicolon is missing. Syntax error. Correct the error in the source program.

Table C-1. (continued)

Error	Meaning
61	Too many initializers. Right brace } is missing. You specified more initial values than variable locations. Correct the error in the source program.
62	Left parenthesis (is missing. Syntax error. Correct the error in the source program.
63	Keyword "while" is missing in "do...while" construct. Probably mismatched braces. Correct the error in the source program.
64	Colon is missing. Syntax error. Correct the error in the source program.
65	Internal compiler error. Bad constant load. Internal compiler error. Contact the Digital Research Technical Support Center if this message displays isolated from any other error messages.
66	Internal compiler error. Unknown pointer size. Internal compiler error. Contact the Digital Research Technical Support Center if this message displays isolated from any other messages.
67	Use operators ++ and -- on int/char/long/short only. You used ++ and -- operators on function pointers. Correct the semantic error in the source code.

Table C-1. (continued)

Error	Meaning
68	MESSAGE SPACE RESERVED
69	MESSAGE SPACE RESERVED
70	MESSAGE SPACE RESERVED
71	Cannot return certain types of expressions. The compiler read a return statement but can not return certain types of expressions, such as whole structures. Correct the semantic error in the source program.
72	Internal compiler error. Internal compiler error. Contact the Digital Research Technical Support Center.
73	Use constant expression to initialize static and extern variables. You can initialize statically allocated variables with a constant expression only. A statically allocated variable is one declared either static or extern. You can initialize automatic variables with nonconstant expressions. Correct the semantic error in the source program.
74	MESSAGE SPACE RESERVED
75	MESSAGE SPACE RESERVED
76	Variable is not large enough to hold a pointer. You specified a variable that is not large enough to hold a pointer. For example, a char variable was specified to hold an array name. Correct the semantic error in the source program.

Table C-1. (continued)

Error	Meaning
77	Variable too large to hold initial value. You specified a variable that is not large enough to hold the initial value. Correct the semantic error in the source program.
78	Offsets into other segments not implemented. Internal compiler error. Contact the Digital Research Technical Support Center.
79	With pointers, only use operators: + - ++ -- You specified incorrect operators for use with pointers. Correct the semantic error in the source program.
80	MESSAGE SPACE RESERVED
81	Invalid parameter expression. Internal compiler error. Contact the Digital Research Technical Support Center if this message displays isolated from any other error messages.
82	MESSAGE SPACE RESERVED
83	WARNING: Indirection for non-pointers is not portable. Integers can be indirected successfully in Digital Research C (small model only) and PDP-11 C. This is an ERROR WARNING MESSAGE.
84	Cannot add arrays or structures. Do not use + operator with arrays. The program attempts to add arrays. Correct the semantic error in the source program.

Table C-1. (continued)

Error	Meaning
85	MESSAGE SPACE RESERVED
86	Use only += or -= operators with pointers. You cannot use certain assignment operators, such as /=, in an expression that involves a pointer. Correct the semantic error in the source program.
87	Colon is missing. Syntax error. Correct the error in the source program.
88	Cannot add pointers. Do not use + operator with pointers. The addition operator, +, is used in an expression with two pointers. You cannot add pointers. Subtraction is acceptable. Correct the semantic error in the source program.
89	Incorrect expression syntax. Syntax error. Correct the error in the source program. Often mismatched parentheses.
90	Comma or right parenthesis expected in parameter list. Syntax error. Correct the error in the source program.
91	Expression is missing before [operator. An lvalue expression of type array or pointer is required on the left of the [operator. Correct the semantic error in the source program.

Table C-1. (continued)

Error	Meaning
92	An lvalue is required before [operator. An expression of the wrong type exists on the left of the [operator. The expression must be an array or pointer. Correct the semantic error in the source program.
93	Array or pointer required on left of [operator. See error 92.
94	Array or pointer required. Cannot subscript. See error 92.
95	WARNING: Subscript is truncated to short int. In the 8086 implementation of C a single aggregate data structure cannot exceed 64K bytes. A long int is used as a subscript and the compiler discards the upper word. This is an ERROR WARNING MESSAGE.
96	Right square bracket] is missing. Syntax error. Correct the error in the source program.
97	Identifier missing on right of . operator. A required identifier that names a struct or union member is missing after the period operator. Correct the syntax error in the source program.
98	Expression missing on left of . operator A required expression describing a struct or union is missing before the period operator. Correct the syntax error in the source program.

Table C-1. (continued)

Error	Meaning
99	Left operand for . operator must be a struct. The operand to the left of a period operator is not a struct or union. Correct the semantic error in the source code.
100	WARNING: Non-local structure field assumed. The struct or union member used with the period operator is not defined as being in the same structure described by the left operand. This is an ERROR WARNING MESSAGE.
101	Identifier missing on right of -> operator. A required identifier that names a struct or union member is missing after the -> operator. Correct the syntax error in the source program.
102	Expression missing on left of -> operator. A required expression describing a struct or union is missing before the -> operator. Correct the syntax error in the source program.
103	Left operand of -> operator must be a pointer. The operand to the left of a -> operator is not a pointer. Correct the semantic error in the source program.
104	WARNING: Non-local structure field assumed. The struct or union member used with the period operator is not defined as being in the same structure described by the left operand. This is an ERROR WARNING MESSAGE.

Table C-1. (continued)

Error	Meaning
105 Division by the constant 0.	The compiler, in an attempt to optimize constant expressions, read an expression with a zero on the right of a divide operator. Correct the semantic error in the source program.
106 Operand types do not match. Cannot coerce to compatible types.	Conflicting types for operands in an expression. The operands cannot be coerced automatically to compatible types. Specify compatible types for operands.
107 Cannot coerce operand type to double.	The compiler attempts to coerce the type of an expression before performing an operation but can not. For example, double_var * pointer. Correct the semantic error in the source program.
108 Cannot coerce operand type to long.	See error 107. Correct the semantic error in the source program.
109 Cannot coerce operand type to unsigned.	See error 107. Correct the semantic error in the source program.
110 WARNING: Indirection for non-pointers is not portable.	Integers can be indirected successfully in Digital Research C (small model only) and PDP-11 C. This is an ERROR WARNING message.

Table C-1. (continued)

Error	Meaning
111 WARNING:	& operator (address of) used redundantly. An array is specified without a subscript expression, and the ampersand operator is used redundantly. The compiler ignores the &. The expression is, by definition, the address of the array. This is an ERROR WARNING MESSAGE.
112 The & operator (address of) requires an lvalue.	The & operator cannot accept the address of an rvalue expression. Correct the semantic error in the source program.
113 An lvalue is required with ++ or -- operator.	The ++ and -- operators require an lvalue. The compiler did not find one. Correct the semantic error in the source program.
114 Incorrect operand type for ++ or -- operator.	You cannot use the ++ and -- operators on arrays or structures. Correct the semantic error in the source program.
115 Data type not specified for expression after sizeof operator.	When an expression is not found after the sizeof operator the compiler assumes a type declaration but finds none. Correct the semantic error in the source program.
116 An lvalue is required with ++ or -- operator.	The ++ and -- operators require an lvalue but the compiler cannot find one. Correct the semantic error in the source program.

Table C-1. (continued)

Error	Meaning
117 Incorrect operand type for ++ or -- operator.	The ++ and -- operators require an operand of the appropriate type. Correct the semantic error in the source program.
118 MESSAGE SPACE RESERVED	
119 Output file write error. Disk is probably full.	Write function returns NULL indicating that available disk space for the output file is exhausted. Compilation is stopped. Use a disk that has more space.
120 WARNING: Not enough registers available for variables.	You specified too many register variables in the program. There are not enough registers to hold all the variables. The register storage class is automatically converted to auto. This warning displays only if you specify the -v3, -v4, or -v5 option switches.
121 WARNING: Additional registers available for variables.	Additional registers are available to hold register variables that you specified in the program. This warning displays only if you specify the -v3, -v4, or -v5 option switches.

End of Appendix C

Appendix D
Variations among Compilers

Digital Research C is designed to be compatible with the UNIX Version 7 operating system. This appendix presents the major variations among Digital Research C for the 8086/88, Digital Research C for the 68000, and the Kernighan and Ritchie description of UNIX C.

- Digital Research C for the 8086 does not support the #line preprocessor command.
- The library functions abort() and signal() are available in the 68000 version of C but not the 8086.
- The =op form of the standard op= operators is not implemented on the 8086 version of C. It is implemented on the 68000 version.
- Kernighan and Ritchie explain in The C Programming Language that you should define a global variable once and only once without the keyword extern but that you should specify the extern keyword for all other references to that global variable. However, Kernighan's and Ritchie's programming convention is not compatible with UNIX or most other C compilers. Instead, UNIX and most other C compilers let you define a global variable in as many places as you like, with or without the extern keyword, provided that the definitions are identical. Digital Research C follows this latter procedure.
- Digital Research C does not support the UNIX function listed in Section 3.

End of Appendix D

Appendix E
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.

E.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.

E.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.

E.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.

E.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.

E.2 Required 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 uppercase and lowercase characters.

- Some compilers sign-extend char 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 to 0xffffffff.

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

E.2.1 Variable and Constant Names

Local variable names should be unique in the first eight characters. Global variable names and procedure names should be unique in the first seven characters. All variable and procedure names should be completely lowercase and should not start with underscore characters.

Usually, names defined with a #define statement should be entirely uppercase. The only exceptions are functions defined as macros, such as getc and isascii. These names must be unique to 16 characters. You can use #define to get around the seven and eight character restrictions on global and local names by redefining long names as unique short names.

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

E.2.2 Variable Types

PORTAB.H contains a set of variable type declaration keywords (Table E-1) and storage class declaration keywords (Table E-2) that you can use to ensure consistent internal representation of data types across different processors.

Declaration keywords in PORTAB.H are macro definitions specified with #define. Using standard type specifiers can be unsafe in programs designed to be portable because of variations in internal representation among different compilers. For example, an integer declared with the keyword int might be 16-bits long on one processor and 32-bits on a different processor. However, an integer declared with the macro WORD is 16-bits on any processor. The standard I/O file STDIO.H already includes PORTAB.H. Therefore, if your program does not include STDIO.H, you must include PORTAB.H explicitly to use the macros shown in Tables 3-1 and 3-2.

Table E-1. Variable Type Macro 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)
DEFAULT	int	(16 bits)
VOID	void (function return)	

Table E-2. Storage Class Macro Definitions

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

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

E.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 example

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

the printf statement uses a long hex constant for masking. This solves the problem for all compilers. Specifying the one's complement often yields ~0xff instead of 0xff00.

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

Commas that separate arguments in functions are not operators. Evaluation order is not guaranteed. For example, the following function call might perform differently for different compilers.

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

E.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 cause 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.

E.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.

E.2.6 Initialized and Uninitialized Data

Usually, C programs have three sections: code (program instructions), initialized data, and 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.

E.2.7 Recommended Module Layout

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

- At the beginning of the file, place a comment describing the 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 no value returns.
  - 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 F for a sample program.

E.3 Coding Suggestions

The following suggestions increase program portability and make programs easier to maintain.

- Keep source code within an 80-character margin for easier screen editing.
- Use a standard indention technique, such as the following:
  - Begin statements in a procedure one tab stop (column eight) from the left margin.
  - Indent statements controlled by an if, else, while, do, or for one tab stop. If you require multiple nested indentions, use two spaces for each nesting level. Avoid going more than five levels deep.
  - Place the brackets surrounding each compound statement on a separate line, aligned with the indention of the controlling statement. For example,

```
for(i=0;i<MAXNUM;i++)
{
    j = compute(i);
    if (j > UPPER)
            j = UPPER;
    output(j);
}
```

  - Place a null statement controlled by an if, else, while, for, or do on a separate line, indented for readability.
- To document your code, insert plenty of comments. If your code is particularly abstruse, inserting comments helps clarify it.
- Put all maintenance documentation in the source code itself. If you do not, the documentation will not be updated when the code changes.
- Use blank lines, form-feeds, and white space to improve readability.

End of Appendix E

Appendix F
Sample C Modules

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          */
/*              _prntl          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.                                        */
/*                                                                          */
/*                                                                          */
/****************************************************************************/


/*
 *      Include files:
 */
#include        <stdio.h>
```

Listing F-1. _Printf Module

```
/*
 *      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            */
                                        /*****************************/
```

Listing F-1. (continued)

```
/****************************************************************************
*
*               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,argl);
*
*       Where:
*
*               fd              Is the file or string pointer.
*               func            Is the function to handle output.
*               fmt             Is the address of the format string.
*               argl            Is the address of the first arg.
*
*
*       Returns:
*
*               Number of characters output
*
*       Bugs:
*
*       It is assumed that args are contiguous starting at "argl", and that
*       all are the same size (LONG), except for floating point.
*
*
****************************************************************************/
_printf(fd,f,fmt,al)                            /****************************/
        LONG    fd;                             /*                          */
        LONG    (*f)();                         /* Function pointer         */
        BYTE    *fmt;                           /* -> Format string         */
        LONG    *al;                            /* -> 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       */
                                                /****************************/
```

Listing F-2. Printf Internal Routine

```
                                                /***************************/
        ccount = 0;                             /* Initially no characters */
        _ptrbf = buf;                           /* Set buffer pointer      */
        adx = al;                               /* Copy address variable   */
        __ptrst = fd;                           /* Copy file descriptor    */
        __fmt = fmt;                            /* Copy format address     */
                                                /***************************/
        if(*__fmt == 'L' || *__fmt == 'l')      /* Skip long output        */
                __fmt++;                        /*      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            */
                                                /***************************/
```

Listing F-2. (continued)

```
  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);             /* Count it                */
       ccount++;                     /*                         */
     }                               /***************************/
```

Listing F-2. (continued)

```
           while (n--)                          /* Output Converted        */
           {                                    /*                         */
                (*f)(fd,*s++);                  /*                   Data  */
                ccount++;                       /* Count it                */
           }                                    /*                         */
                                                /***************************/
           while (m-- > 0)                      /* If left adjust,         */
           {                                    /*                         */
                (*f)(fd,padchar);               /*                 Pad     */
                ccount++;                       /* Count padded characters */
           }                                    /***************************/
           _ptrbf = buf;                        /* Reset buffer pointer    */
          }                                     /* End else                */
        }                                       /* End while               */
        if((*f) == _putstr)                     /* If string output,       */
                (*f)(fd,'0');                   /* Drop in terminator char */
                                                /***************************/
        return(ccount);                         /* Return appropriate value*/
}                                               /* End _printf             */
                                                /***************************/
```

Listing F-2. (continued)

```
/****************************************************************************/
/*                                                                          */
/*                      _ 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)&0x7) || 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               */
                                                /****************************/
```

Listing F-3. _Prnt8 Procedure

```
/*****************************************************************************/
/*                                                                           */
/*                      _ 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 & 0x0fffffff0);       /* If <> 0, print first     */
        d = n&0xf;                               /* Take low four bits       */
        *_ptrbf++ =  d > 9 ? 'A'+d-10 : '0' + d;/* ASCIIfy into buffer       */
}                                                /****************************/
```

Listing F-4. _Prntx Function

```
/*****************************************************************************/
/*                                                                           */
/*                        __C o n v   F u n c t i o n                        */
/*                        ------------------------------                     */
/*                                                                           */
/*      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);                               /****************************/
}
```

Listing F-5. __Conv Function

```
/****************************************************************************/
/*                                                                          */
/*                    _ 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                  */
}                                               /****************************/
```

Listing F-6. _Putstr Function

```
/******************************************************************************/
/*                                                                            */
/*                      _ P r t l   F u n c t i o n                           */
/*                      ---------------------------                           */
/*                                                                            */
/*      Function "_prtl" converts a LONG binary quantity to decimal ASCII    */
/*      at the buffer pointed to by "_ptrbf".                                 */
/*                                                                            */
/*      Calling Sequence:                                                     */
/*                                                                            */
/*              _prtl(n);                                                     */
/*                                                                            */
/*      where "n" is the value to be converted.                               */
/*                                                                            */
/*      Returns:                                                              */
/*                                                                            */
/*              (none)                                                        */
/*                                                                            */
/******************************************************************************/
VOID _prtl(n)                                     /*                          */
        REG BYTE        n;                        /* Conversion input         */
{                                                 /****************************/
        REG LONG        digs[10];                 /* store digits here        */
        REG LONG        *dpt;                     /* Points to last digit     */
                                                  /****************************/
        dpt = digs;                               /* Initialize digit pointer */
                                                  /****************************/
        if (n < 0)                                /* Fix                      */
                                                  /*      up                  */
                                                  /*            sign          */
{*_ptrbf++ = '-'; n = -n;}                        /*                  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!*/
        }                                         /*                          */
}                                                 /****************************/
```

Listing F-7. _Prtl Function

End of Appendix F

Index

NOTES

NOTES

NOTES

This page added for scan notes.

- -scanned @ 150dpi 6.62in w by 8.5 in h in color, original document.
- -omitted blank pages.
- -omitted DRI reprint of 'The C Programming Language' by K&R, which is the first half of the original documentation but a separate manual.
- -since DRI C follows this version, it is pre-ansi C.
- -each page scanned into a separate JPEG files, 188 total, front cover to rear.

This effort is in a memorial to:

Tim Olmstead WB5PFJ

'Faith is the substance of things hoped for.'

Reader Comment Card

We welcome your comments and suggestions. They help us provide you with better product documentation.

Date ____________

1. What sections of this manual are especially helpful?

____

____

____

____

2. What suggestions do you have for improving this manual? What information is missing or incomplete? Where are examples needed?

____

____

____

____

3. Did you find errors in this manual? (Specify section and page number.)

____

____

____

____

C Language Programmer's Guide
for the CP/M-86® Family of Operating Systems
2nd Edition Oct. 1983
3049-2023-002

From: ____________________

____

____

____

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 182 / PACIFIC GROVE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL RESEARCH®
Attn: Publications Production
P.O. BOX 579
PACIFIC GROVE, CA 93950-9987