

TABLE OF CONTENTS

Paragraph Number	Title	Page Number
Chapter 1		
Introduction		
1.1	Overview	1-1
1.2	Error Codes	1-4
1.3	Notation	1-4
1.4	Manual Organization	1-5
Chapter 2		
Installation Guide		
2.1	Introduction	2-1
2.1.1	Standard Installation On A Sun	2-1
2.1.2	Alternate Installation On A Sun	2-1
2.1.3	Standard Installation On A NeXT	2-2
2.1.4	Alternate Installation On A NeXT	2-2
2.1.5	Installation On An MS-DOS Machine (80386 or 80486)	2-3
2.2	Test Program	2-4
Chapter 3		
Control Program Options		
3.1	Overview	3-1
3.2	g96k Command Line Options	3-3
3.2.1	Preprocessor Phase Options	3-5
3.2.2	Compile Phase Options	3-18
3.2.3	Assemble Phase Options	3-28
3.2.4	Link Phase Options	3-28
Chapter 4		
About g96k		
4.1	Introduction	4-1
4.2	Identifiers	4-1
4.3	Predefined Preprocessor Macro Names	4-1
4.4	Data Types and Sizes	4-1

Table of Contents (Continued)

Paragraph Number	Title	Page Number
4.4.1	Integral Data Types	4-2
4.4.2	Floating-point Types	4-2
4.4.3	Pointer Types	4-4
4.5	Register Usage	4-4
4.6	Memory Usage	4-6
4.6.1	Activation Record	4-7
4.6.2	Global/Static Data	4-10
4.7	Compiler Naming Conventions	4-10
4.8	Subroutine Call Sequence	4-10
4.8.1	Caller Sequence	4-11
4.8.2	Callee Sequence	4-11
4.8.3	Return Sequence	4-11
4.9	Run-time Safety	4-11
4.9.1	Memory Allocation Checks	4-12
4.9.2	Run-time Stack Checks	4-12
4.10	Optimization Techniques Implemented	4-12
4.10.1	Register Promotion and Lifetime Analysis	4-12
4.10.2	Common Sub-expression Elimination	4-13
4.10.3	Constant Propagation/Folding	4-13
4.10.4	Dead Code Elimination	4-14
4.10.5	Tail Merging	4-14
4.10.6	Strength Reduction	4-14
4.10.7	Loop Invariant Code Motion	4-14
4.10.8	Hardware DO Loop Instruction	4-15
4.10.9	Loop Rotation	4-15
4.10.10	Jump Optimizations	4-15
4.10.11	Instruction Combination	4-15
4.10.12	Leaf Routine Detection	4-16
4.10.13	Function In-lining	4-16

Chapter 5 Mixing C and Assembly Language

5.1	Overview	5-1
5.2	In-line Assembly Code	5-1
5.2.1	Instruction Template	5-3
5.2.2	Output/Input Operands	5-4
5.2.3	Explicit Register Saving	5-6
5.2.4	In-line Assembly Code Examples	5-7
5.2.5	Controlling Labels Generated by the Compiler	5-11
5.2.5.1	Calling Assembly Subroutines	5-12
5.2.5.2	Calling C Subroutines from Assembly Code	5-13

Table of Contents (Continued)

Paragraph Number	Title	Page Number
5.2.5.3	Referencing Assembly Global Variables from C	5-14
5.2.5.4	Referencing Global C Variables from Assembly Language . . .	5-15
5.2.6	Specifying Registers for Variables.	5-17
5.2.7	Optimizer Effects on Code.	5-17
5.3	#pragma Directive	5-17
5.4	Out-of-line Assembly Code	5-20
5.4.1	General Template	5-21
5.4.1.1	Prologue	5-21
5.4.1.2	Save all registers.	5-22
5.4.1.3	Main Program	5-22
5.4.1.4	Restore all registers.	5-23
5.4.1.5	Epilogue.	5-24
5.4.1.6	Out-of-line Assembly Code Example.	5-24
5.4.2	Global C and Static Variables in C	5-28
5.4.3	Using Run-time Stack for Local Data	5-29
5.4.4	Calling C Routines.	5-30
5.4.5	Optimization Technique.	5-31

Chapter 6 Software-Hardware Integration

6.1	Overview	6-1
6.2	Run-Time Environment Specification Files	6-1
6.3	crt0 File	6-2
6.3.1	Bootstrapping the C program	6-2
6.3.2	Memory Configuration and Management	6-4
6.3.3	Interrupt Vectors	6-6
6.3.4	Miscellaneous Code	6-9
6.4	Signal File	6-9
6.4.1	Signal()	6-10
6.4.2	Raise()	6-11
6.5	Setjmp File.	6-12

Appendix A Programming Support

A.1	Standard ANSI Header Files	A-1
A.2	ANSI C Library Sub-routines	A-1
A.2.1	Hosted	A-2
A.2.2	Non-Hosted	A-2
A.3	Forcing Library Routines Out-of-line	A-2
A.4	Function Descriptions	A-3

Table of Contents (Continued)

Paragraph Number	Title	Page Number
---------------------	-------	----------------

Appendix B DSP96002 Instruction Set and Assembler Directive Summary

B.1	Overview	B-1
B.2	Instruction Set	B-1
B.2.1	Arithmetic Instructions	B-1
B.2.2	Logical Instructions	B-3
B.2.3	Bit Manipulation Instructions	B-3
B.2.4	Loop Instructions	B-4
B.2.5	Move Instructions	B-4
B.2.6	Program Control Instructions	B-4
B.3	Directive Summary	B-5
B.3.1	Assembly Control	B-5
B.3.2	Symbol Definition	B-6
B.3.3	Data Definition/Storage Allocation	B-6
B.3.4	Listing Control and Options	B-7
B.3.5	Object File Control	B-7
B.3.6	Macros and Conditional Assembly	B-7
B.3.7	Structured Programming	B-7

Appendix C Utilities

asm96000	— Motorola DSP96000 COFF Assembler	C-2
cldinfo	— Memory size information from Motorola DSP COFF object file	C-6
cldlod	— Motorola COFF to LOD Format converter	C-7
cofdmp	— Motorola DSP COFF File Dump Utility	C-8
dsplib	— Motorola DSP COFF Librarian	C-9
dsplnk	— Motorola DSP COFF Linker	C-11
gdb96	— GNU Source-level Debugger for DSP96002	C-15
run96sim	— Motorola DSP96002 Simulator Based Execution Device.	C-17
srec	— Motorola DSP S-Record Conversion Utility	C-18

Appendix D Additional Support

Motorola DSP Product Support	D-2
DSP96002CLASx Assembler/Simulator	D-2
Macro Cross Assembler Features:	D-2
Simulator Features:	D-3
DSP96000ADSx Application Development System	D-3

Table of Contents (Continued)

Paragraph Number	Title	Page Number
	DSP96000ADS Application Development System Hardware Features:	D-3
	DSP96000ADSx Application Development System Software Features:	D-4
	Dr. BuB Electronic Bulletin Board	D-4
	Motorola DSP News	D-5
	Motorola Field Application Engineers	D-5
	Design Helpline – 1-800-521-6274	D-5
	DSP Applications Helpline – (512) 891-3230	D-5
	DSP Marketing Information – (512) 891-2030	D-5
	DSP Third-Party Support Information – (512) 891-3098.	D-5
	DSP University Support – (512) 891-3098	D-5
	DSP Training Courses – (602) 897-3665	D-6
	Reference Books and Manuals.	D-6
	General DSP:	D-6
	Digital Audio and Filters:	D-7
	C Programming Language:	D-8
	Controls:	D-9
	Graphics:	D-9
	Image Processing:	D-10
	Motorola DSP Manuals:	D-11
	Numerical Methods:	D-11
	Pattern Recognition:	D-11
	Speech:	D-12
	Telecommunications:	D-12

Appendix E GNU Debugger (GDB)

Introduction	E-1
Commands Not Implemented	E-1
DSP96002 Differences / Specific Requirements	E-1
Summary of GDB	E-3
GNU GENERAL PUBLIC LICENSE	E-4
Preamble	E-4
TERMS AND CONDITIONS	E-5
NO WARRANTY	E-7
END OF TERMS AND CONDITIONS	E-8

E-1 Input and Output Conventions

E-2 Specifying GDB's Files

Table of Contents (Continued)

Paragraph Number	Title	Page Number
2.1.	Specifying Files with Arguments	E-13
2.2.	Specifying Files with Commands.	E-13

E-3 Compiling Your Program for Debugging

E-4 Running Your Program Under GDB

4.1.	Your Program's Arguments	E-20
4.2.	Your Program's Environment	E-20
4.3.	Your Program's Working Directory	E-21
4.4.	Your Program's Input and Output	E-21
4.5.	Debugging an Already-Running Process	E-22
4.6.	Killing the Child Process	E-22

E-5 Stopping and Continuing

5.1.	Signals.	E-25
5.2.	Breakpoints	E-27
5.2.1.	Setting Breakpoints	E-27
5.2.2.	Deleting Breakpoints	E-28
5.2.3.	Disabling Breakpoints	E-29
5.2.4.	Break Conditions	E-30
5.2.5.	Commands Executed on Breaking	E-32
5.2.6.	"Cannot Insert Breakpoints" Error	E-33
5.3.	Continuing	E-34
5.4.	Stepping	E-34

E-6 Examining the Stack

6.1.	Stack Frames	E-37
6.2.	Backtraces.	E-38
6.3.	Selecting a Frame	E-39
6.4.	Information on a Frame	E-40

E-7 Examining Source Files

7.1.	Printing Source Lines	E-43
------	---------------------------------	------

Table of Contents (Continued)

Paragraph Number	Title	Page Number
---------------------	-------	----------------

7.2.	Searching Source Files	E-45
7.3.	Specifying Source Directories	E-45

E-8 Examining Data

8.1.	Expressions	E-47
8.2.	Program Variables	E-48
8.3.	Artificial Arrays	E-48
8.4.	Format options	E-49
8.5.	Output formats	E-50
8.5.1.	Examining Memory	E-51
8.6.	Automatic Display	E-53
8.7.	Value History	E-55
8.8.	Convenience Variables	E-56
8.9.	Registers	E-57
8.9.1.	Examples	E-58

E-9 Examining the Symbol Table

E-10 Altering Execution

10.1.	Assignment to Variables	E-61
10.2.	Continuing at a Different Address	E-62
10.3.	Giving the Program a Signal	E-63
10.4.	Returning from a Function	E-63

E-11 Canned Sequences of Commands

11.1.	User-Defined Commands	E-65
11.2.	Command Files	E-66
11.3.	Commands for Controlled Output	E-66

E-12 Options and Arguments for GDB

12.1.	Mode Options	E-69
12.2.	File-specifying Options	E-69
12.3.	Other Arguments	E-70

Table of Contents (Continued)

Paragraph Number	Title	Page Number
	E-13	
	Using GDB under GNU Emacs	
	E-14	
	Remote Kernel Debugging	
14.1.	Commands for Remote Debugging	E-73

Chapter 1

Introduction

1.1 Overview

The DSP96KCC GNU based C cross-compiler is the latest high-level language development system for the Motorola DSP96002 of digital signal processors (DSP). It includes:

- an integrated control program — **g96k**
- an ANSI compliant C language preprocessor — **cpp**
- an ANSI optimizing C compiler — **gcc96-cc1**
- a DSP96000 common object file format (COFF) assembler — **asm96000**
- a COFF linker — **dsplnk**
- a COFF librarian — **dsplib**
- a source level debugger for C — **gdb96**

This integrated software system runs on a variety of machines and operating systems, including the IBM PC™ (80386 family and above — 386sx, 486, etc.), Sun workstations, and NeXT™ computers.

The compiler implements the full C language as defined in American National Standard for Information Systems - Programming Language - C, ANSI X3.159-1989. It accepts one or more C language source files as input and generates a corresponding number of assembly language source files which are suitable as inputs to the DSP96000 COFF assembler. The compiler automatically implements numerous optimizations which simplifies implementing fast and efficient DSP algorithms.

The C language preprocessor is an implementation of the ANSI standard which includes support for arbitrary text file inclusion, macro definition and expansion, and conditional compilation. The preprocessor exists as a separate program and may be used as a general-purpose macro preprocessor.

The compiler control program, **g96k**, is the standard compiler interface which is used to control the sequence of operations required to compile a program. This control program allows the user to select a wide variety of control options which affect the four compilation phases — preprocessing, compiling, assembling, and linking. The control

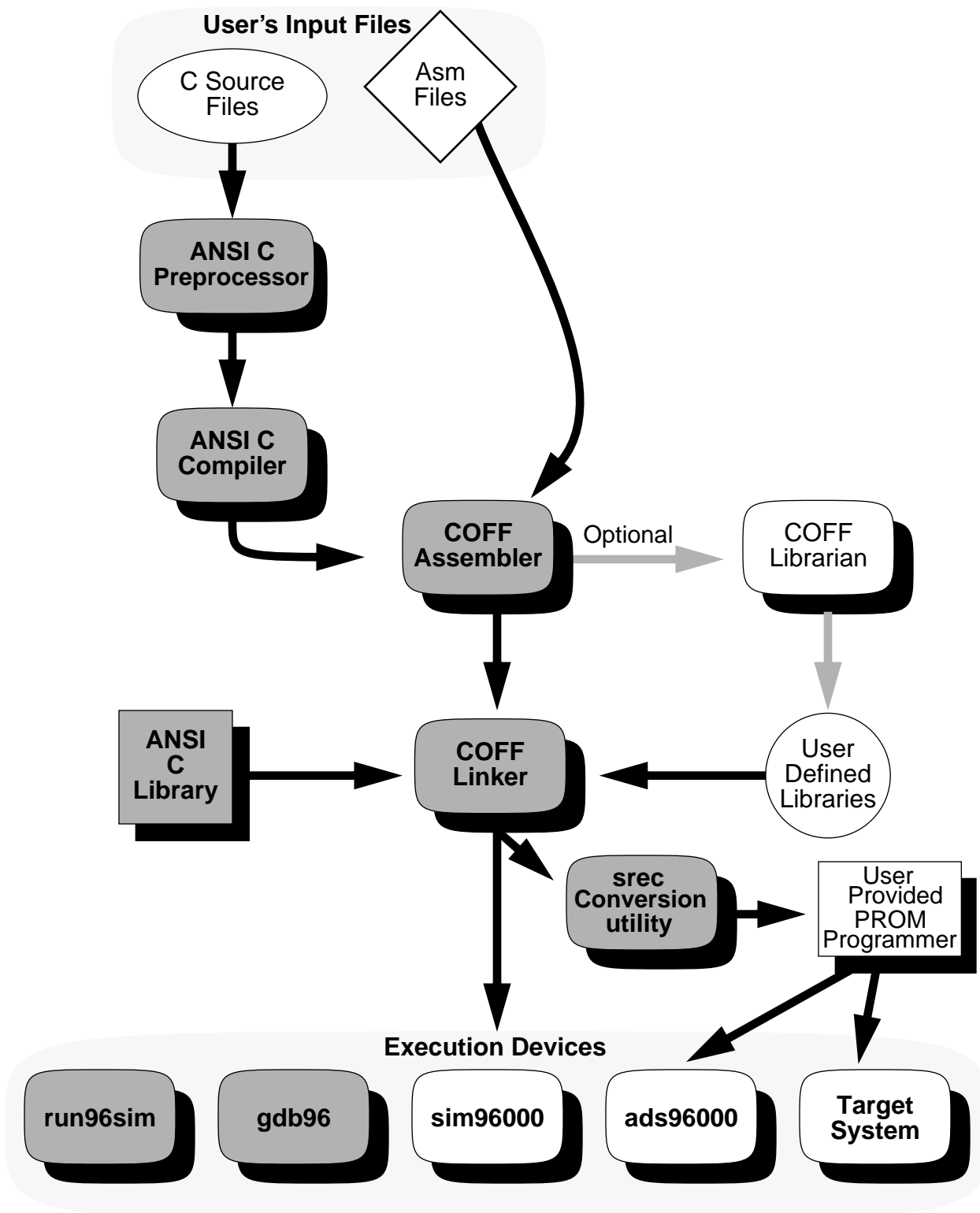


Figure 1-1. Motorola Software Development System

- The shaded box or ovals are provided with **DSP96KCC** package.

program accepts a list of command line options and filenames invoking each routine in the source translation process in turn.

1. Given a list of C source files from the user (see Figure 1-1) and options to apply to those files, the control program runs each file through the preprocessor and the compiler. The compiler creates individual assembly language source files for each C source file provided on the command line.
2. The control program then sends these compiled files to the assembler in addition to any assembly language files provided by the user which have been listed in the **g96k** command line.
3. The assembler output is routed to the linker for final processing. The linker tries to resolve all unresolved link-time symbols with the standard (or explicitly requested) C libraries. The COFF linker output (which is an object file) may then be directed to any of several execution devices. Notice that the assembler can also be used to create library files which can be included in a user defined library.
4. The execution devices shown in Figure 1-1 are:
 - a. **run96sim** which allows the DSP96002 code (in COFF format) to be executed on the host computer,
 - b. **gdb96** which provides debugging tools for trouble-shooting in the COFF format code,
 - c. **sim96000** is a complete DSP96002 simulator that will execute the assembled code (in COFF format or .lod file format) and allow examination of registers and memory,
 - d. **ads96000** is the development system that can then be used to load and execute the assembled code (in .lod file format) on the ADS development system, and
 - e. the target system shown is the user's custom designed DSP system. Depending on the design, code can be loaded into a DSP96002 from a block of memory or through the host port. The DSP96002 can use code that is preprogrammed in a ROM and down loaded.

Note: The three execution devices in the white ovals are not part of the C compiler software. The COFF linker output can be used by these execution devices directly. The conversion utility **srec** (see Figure 1-1) can be used to convert the executable file from the COFF Linker to a suitable format for PROM burning. These PROMs can then be used on the ADS development system or the user's target system. The PROM programmer, ADS development system and user's target system are not part of the DSP96KCC compiler.

The DSP96002 represents a departure from more conventional architectures on which other implementations of the C language are based. Also, the nature of DSP applications

dictates that a greater measure of control be provided to the programmer in specifying the constraints of the run-time environment. For these reasons, the components of the development system include options for handling separate memory spaces, stack initialization, chip operating modes and other issues.

The purpose of this manual is to:

1. provide detailed installation procedures for both UNIX based systems and MS-DOS based systems. This manual explains how to install and operate the DSP96KCC ANSI C compiler development system in a hosted environment.
2. provide an overview of the compiler operation. It also includes information on combining C modules with assembly language programs and advanced topics pertaining to compiler run-time characteristics.
3. provide reference information on compiler options, ANSI library routines, and utilities.

This manual assumes a familiarity with the C programming language, with the host machine and operating system environment. It also assumes that the programmer understands how to create and edit C language source files on the host system. For more information on the C language and other DSP96002 development tools, see the references listed in Appendix D.

1.2 Error Codes

The error messages generated by the compiler are intended to be complete without additional explanation. Since the compiler produces a detailed description of the problem rather than an error code, these error messages have not been reproduced in this manual.

1.3 Notation

The following notation will be used in this text.

1. A prompt used in this manual indicating that a user entered command follows is:
 `%`
2. A UNIX directory name is labeled as:
 `/usr/directory`
3. The content of a UNIX directory is examined by entering:
 `% ls`
4. The content of a UNIX file is examined by entering:
 `% cat filename`
5. An MS-DOS directory name is labeled as:
 `\usr\directory`

6. The content of an MS-DOS directory is examined by entering:
 % dir
7. The program “hello” is executed by the command line:
 % hello
8. The notation [x,y,l] indicates that x, y or l can be selected; e.g.
 crt096[x,y,l].cln
 means
 crt096x.cln
 or
 crt096y.cln
 or
 crt096l.cln

1.4 Manual Organization

Installation details are provided in Chapter 2, the compiler operation is described in Chapters 3-6 and reference information is in Chapter 3 and Appendices A-E. The content of each chapter and appendix is described below.

Chapter 2, Installation Guide, describes the installation and organization of the DSP96KCC software. It details how to set up an operating environment on the host system by defining global environment variables and includes a step-by-step installation procedure.

Chapter 3, Compiler Operation, discusses the four passes of the compilation process with particular attention to the functions of the compiler control program **g96k**. This chapter includes a list of the compiler invocation options along with example command lines for different memory and program configurations.

Chapter 4, About G96k, provides information on the compiler run-time environment, including explanations of compiler register and memory usage, stack frame architecture, stack overflow checking, and defining/referencing of absolute memory locations. Additionally, this chapter covers implementation issues such as data type sizes.

Chapter 5, Mixing C and Assembly Language, discusses the methods for using assembly language in conjunction with C language programs. It covers the inclusion of assembly language within a C source file and also describes (1) linking assembly language modules with C modules and (2) linking C modules with assembly language modules.

Chapter 6, Software-Hardware Integration, describes (1) modifying a program's run-time environment, (2) writing software to handle interrupts, and (3) the **setjmp/longjmp** ANSI library routines.

Appendix A, Programming Support, provides a complete description and brief example for every ANSI library subroutine distributed with the C compiler.

Appendix B, DSP96002 Instruction Set and Assembler Directive Summary, provides

a brief overview of the assembly language instructions and assembler directives.

Appendix C, Utilities, provides DSP96002 Assembler manual pages for each of the supporting programming utilities provided with the compiler.

Appendix D, Additional Support, provides a description of software, hardware, support telephone numbers, and suggested reading for the C language and various areas of DSP.

Appendix E, Debugger, provides a reproduction of the GNU debugger manual.

Chapter 2

Installation Guide

2.1 Introduction

There are two installation procedures described for each type of computer. The first procedure uses the default location for the files. The second procedure allows the user to select the directory that the compiler's files will be located in.

The various parts of the compiler reside in a directory tree named **dsp**. The default location for the **dsp** directory tree is **/usr/local** on UNIX systems and **c:** on MS-DOS systems. If these default locations are acceptable, then perform the standard installation; if they are not acceptable, then perform the alternate installation which allows to install the software on a directory called **<compiler's dsp directory tree>**.

2.1.1 Standard Installation On A SUN

1. Insert the supplied cartridge tape into the tape drive.
2. Login as **root**.
3. Enter the command: **cd /usr/local**.
4. Enter the command: **tar xvf /dev/rst0**. If the package was in floppy diskette instead of tape, then use the command: **bar xvf /dev/rfd0**. If the tape drive and floppy drive must be accessed via a different device file than **rst0** and **rfd0**, then use whatever method works on the system.
5. Logout.
6. Make sure that all users add **/usr/local/dsp/bin** to their path to point to the control program **g96k** and other programs in the DSP96KCC distribution package.

2.1.2 Alternate Installation On A SUN

1. Insert the supplied cartridge tape into the tape drive.

2. Login as **root**, or as yourself.
3. Inside the shell, use the command **cd** to go to the directory where the compiler's **dsp** directory tree is to reside. For this example, assume that the compiler is to be installed in **/usr/mydir**.
4. Make sure that you have write permission in the directory.
5. Enter the command: **tar xvf /dev/rst0**. If the package was in floppy diskette instead of tape, then use the command: **bar xvf /dev/rfd0**. If the tape drive and floppy drive must be accessed via a different device file than **rst0** and **rfd0**, then use whatever works on the system.
6. Make sure that every user adds **<compiler's dsp directory tree>/dsp/bin** to their path. In this example, the path **/usr/mydir/dsp/bin** would be added to everyone's path.
7. Make sure that every user sets the environment variable **DSPLOC** to the path leading to the **dsp** directory tree which is the directory **<compiler's dsp directory tree>**. In this example, **DSPLOC** would be set to **/usr/mydir**.

2.1.3 Standard Installation On A NeXT

1. Login as **root**.
2. Insert the supplied disk labeled **Disk 1** into the floppy drive.
3. Using the **File Viewer**, launch **/NextApps/Installer.app**.
4. From the **Installer** menu, choose **Package, Open**. The **Installer** will provide a file viewer labeled **Open**. Use it to select **/Disk #1/g96k.pkg**.
5. An **Installer** control panel will appear: choose the **Install** icon.
6. The **Installer** will provide a file viewer labeled **Install In:**, with **/usr/local** as the default location. Choose the default location.
7. Proceed with the installation; Insert subsequent disks when asked.
8. Make sure that all users add to their path **/usr/local/dsp/bin** to point to the control program **g96k** and other programs in the **DSP96KCC** distribution package.

2.1.4 Alternate Installation On A NeXT

1. Login as **root**, or as yourself. Make sure that you have write permission in the directory in which the compiler is to be installed.
2. Insert the supplied disk labeled **Disk 1** into the floppy drive.
3. Using the **File Viewer**, launch **/NextApps/Installer.app**.
4. From the **Installer** menu, choose **Package, Open**. The **Installer** will provide a file viewer labeled **Open**. Use it to select **/Disk #1/g96k.pkg**.
5. An **Installer** control panel will appear: choose the **Install** icon.
6. The **Installer** will provide a file viewer labeled **Install In:**, with **/usr/local** as the default location. Choose the directory where the compiler's directory tree, **dsp**, is to reside. Again, be sure that you have write permission in the chosen directory.
7. Proceed with the installation; insert subsequent disks when asked.
8. Make sure that every user adds the directory **<compiler's dsp directory tree>/dsp/bin** to their path. For example, if the compiler is installed in the directory **/usr/mydir**, the path **/usr/mydir/dsp/bin** should be added to everyone's path.
9. Make sure that every user sets the environment variable **DSPLOC** to the path leading to the **dsp** directory tree which is the directory **<compiler's dsp directory tree>**. In the above example, **DSPLOC** would be set to **/usr/mydir**.

2.1.5 Installation On An MS-DOS Machine (80386 or 80486)

1. Insert the supplied floppy labeled **Disk 1** into floppy drive **A:**.
2. Change to floppy drive **A**, with the command **A:**.
3. Run the install program, **install.exe**. This installation program will ask questions about the computer being used and about where the compiler's directory tree, **dsp**, is to reside.
4. Add all new lines of code specified by **install** into the **autoexec.bat** and **config.sys** files. The only difference between the standard and alternate installation procedure on the PC is whether or not the default output drive or default location is selected. If the defaults are not selected, an environment variable named **DSPLOC** must be added to the **autoexec.bat** file. The **install** program will provide directions. The lines to be added to the **autoexec.bat** file are as follows:

- a. **TERMCAP** must be set to the current location of **<compiler's dsp directory tree>\dsp\etc\termcap**:

SET TERMCAP=<compiler's dsp directory tree>\dsp\etc\termcap

if the directory <compiler's dsp directory tree> is d:\usr\mydir, then

SET TERMCAP=d:\dsp\mydir\dsp\etc\termcap

- b. **TERM** must be set to indicate the type of display to be used (probably **nansi-mono**):

SET TERM=NANSI-MONO

- c. **DSPLOC** need only be set if the default output drive or the default location is not chosen. If **DSPLOC** must be set, it must be set to the location of the dsp directory tree. For example, if one installed the compiler's directory tree **dsp** in the directory **d:\usr\mydir**, then **DSPLOC** would need to be set as **d:\usr\mydir**:

SET DSPLOC=<compiler's dsp directory tree>

if the directory <compiler's dsp directory tree> is d:\usr\mydir, then

SET DSPLOC=d:\usr\mydir

4. Make sure that **<compiler's dsp directory tree>\dsp\bin** is included in the **path** instruction. This is needed by **command.com** if it is to find **g96k**. If the default drive and path were chosen, then the path **c:\dsp\bin** would need to be added. If, for example the compiler's dsp directory tree was installed in **f:\usr\mydir**, then **f:\usr\mydir\dsp\bin** would need to be added.

PATH ...;c:\dsp\bin;...

If the directory <compiler's dsp directory tree> is f:\usr\mydir, then

PATH ...;f:\usr\mydir\dsp\bin;...

5. Make sure that the DOS extended memory manager for g96k is not interfered with by other DOS extended memory managers. The compiler g96k uses its own DOS extended memory manager called dos4gw.exe, and this memory manager may not work if some other memory manger is installed. It is recommended that all DOS extended memory managers be removed and the system be reboot without a memory resident memory manager. The DOS extended memory manager dos4gw.exe is called during the compiler's execution and requires at least 2M bytes of RAM. This manager uses hard drives for the swap space for the memory man-

agement. The default location and size of the swap space are C drive and 16M bytes.

6. The DOS environment DOS4GVM controls parameters for configuration of the DOS extended memory manager and has the format:

[option[#value]] [option[#value]]

The possible parameters for the option are:

MINMEM	The minimum amount of RAM managed by the memory manager. Default value is 512KB.
MAXMEM	The maximum amount of RAM managed by the memory manager. Default value is 4MB.
SWAPNAME	The name of the swap file used by the memory manager for swap space. The default is DOS4GVM.SWP on the current drive.
DELETESWAP	Specifies that the swap file should be deleted after memory management.
VIRTUALSIZE	The virtual memory space size. Default is 16MB.

The parameter values should be entered as a numeric value in kilobytes.

As an example, the following line in the autoexec.bat file will enable an 8MB swap file with automatic deletion of the swap file:

```
SET DOS4GVM=DELETESWAP VIRTUALSIZE#8192
```

The following line will use F drive for the swap space instead of the current drive.

```
SET DOS4GVM=DELETESWAP SWAPNAME#F:\BIG.SWP
```

7. There is READ.ME file included in the package and recommended to read for the information of any changes in the installation or relevant subject.

2.2 Test Program

The following test program is intended to be a very simple check to see if the installation has been completed correctly. The program should be put in a file named `example.c`. The control program, **g96k**, compiles the program in file “`hello.c`” and generates the output file “`a.cld`”. Do not enter the % as it is simply a prompt indicating that this line should be entered from the keyboard. The command **run96sim** executes the program in the file “`a.cld`” and the result is to print “`hello world.`” on the computer screen.

Example Program

```
#include <stdio.h>
main()
{
    printf("hello world.\n");
}
```

Commands to Compile and Execute the Example Program

```
% g96k hello.c
% run96sim a.cld
```

Result Printed on the Computer Screen

```
hello                                world.
```

Example 2-1. Test Program

Chapter 3

Control Program Options

3.1 Overview

Program **g96k** is the control program for Motorola's optimizing C compiler for the DSP96002 of digital signal processors. The program **g96k** automates control of the four C compiler phases – preprocessing, compiling, assembling, and linking. The program **g96k** utilizes a command syntax similar to those adopted by UNIX utilities. The **g96k** syntax is:

`g96k [options] file ...`

where:

1. [options] is one or more of the options found in this chapter. One difference between **g96k** and UNIX-style utilities is that the combination of multiple single character options is not allowed. For example, “**-O -g**” instructs the compiler to generate an optimized executable with **gdb96** source level debugger information, whereas “**-Og**”, which is acceptable to UNIX-style compilers is unacceptable to **g96k**.
2. “file ...” is the file(s) to be processed. Program **g96k** accepts input filenames suffixed with “**.c**” (ANSI C source files), “**.i**” (preprocessed C source files), “**.asm**” (DSP96002 assembly code), and “**.cln**” (COFF link files). The control program processes each file according to this suffix. The **g96k** output is controlled by the specific set of command line options provided. For instance, if no command line arguments are provided, the compiler will generate a COFF load file “**a.cld**”. If the **-c** option is invoked, the compiler will generate a COFF link file suffixed with “**.cln**”. A complete description of the command line options, with examples, is provided in Section 3.2.

Note: It is strongly recommended that **g96k** always be used to invoke the C compiler utilities rather than individually executing them.

A standard directory search list is consulted by **g96k** for:

1. Each of the four executables,
 - a. **cpp** — the C compiler preprocessor,
 - b. **gcc96-cc1** — the C compiler/optimizer,
 - c. **asm96000** — the DSP96002 assembler,
 - d. **dsplnk** — the DSP96002 linker.
2. Start-up file, **crt096*.cln**. Where * is a single character (x, y or l; i.e., x data memory, y data memory or long data memory) to denote the memory model.
3. ANSI C library, **lib96c*.clb**. Where * is a single character (x, y or l) to denote the memory model.

This standard directory search list for **UNIX** systems is:

1. /usr/local/dsp/bin/
2. /usr/local/dsp/lib/
3. /lib/
4. /usr/lib/
5. ./

The standard **MS-DOS** directory search list for the path set up in Section 2.3 is:

1. c:\dsp\bin
2. c:\dsp\lib
3. c:
4. c:\dos
5. other directories in the **path** name

Additionally, **g96k** searches first for programs, start-up files, and libraries prefixed with "gcc96-".

Table 3-1 lists all the user selectable options used by **g96k**. They are grouped to show what program uses each option. All of these options are described in detail later in this chapter; however, these lists provide an overview of what options are available. Notice that there is a -v option listed under both **g96k** Command Options and Preprocessor Phase Options. This is actually the same option but it is used by these two programs in different ways (see Section 3.2 and Section 3.2.1).

Under compile phase options, there is a group of -f options; these are the machine independent optimization options whereas the -m options below are the optimization options specific to the DSP96002. Although these various methods of optimization are all effective, they may have side effects which are undesirable in specific cases, e.g. an optimization option may increase code speed at the cost of increased memory usage. It is often preferable to trade memory space for speed, but in cases where the extra memory space

Table 3-1. DSP96KCC Options

g96k Command Line Options	Compile Phase Options
-Bdirectory	-fno-opt
-bPREFIX	-fno-peephole
-o FILE	-fno-strength-reduce
-v	-fno-defer-pop
	-fforce-addr
Preprocessor Phase Options	-finline-functions
-C	-fcaller-saves
-DMACRO	-fkeep-inline-functions
-DMACRO=DEFN	-fwritable-strings
-E	-fcond-mismatch
-IDIR	-fvolatile
-I-	-ffixed-REG
-i FILE	-g
-M	-O
-MM	-mno-dsp-optimization
-nostdinc	-mno-do-loop-generation
-pedantic	-mno-biv-plus-linv-promotion
-P	-mstack_check
-v	-mx-memory
-UMACRO	-my-memory
-Wcomment	-ml-memory
-Wtrigraphs	-pedantic
	-Q
Assemble Phase Options	-S
-asm string	-w
-c	-W
	-Wimplicit
Link Phase Options	-Wreturn-type
-crt file	-Wunused
-j string	-Wswitch
-LIBRARY	-Wall
-r MAPFILE	-Wshadow
	-Wid-clash-LEN
	-Wpointer-arith
	-Wcast-qual
	-Wwrite-strings

is not available, this particular optimization could not be used.

The various compiler phases will report errors; however, the user has the option to turn off all warnings using -w and can enable additional warnings individually or as a group using -Wall. Note that -Wall enables all but a few of the additional warnings. The warnings which are not enabled by **-Wall** are those listed below **-Wall** in Table 3-1.

3.2 g96k Command Line Options

The default options are:

1. use strict ANSI syntax,
2. perform all machine dependent and independent optimizations,

3. use trigraphs,
4. locate data in the Y data memory space.

-Bdirectory

Add directory to the standard search list and have it searched first. This can also be accomplished by defining the environment variable **G96_EXEC_PREFIX**. Note that only **one** additional directory can be specified and that the **-B** option will override the environment variable.

Example 3-1. To test a new version of the ANSI C library, lib96cy.clb, which is installed as

/usr/local/dsp/new_libs/lib96cy.clb use:

```
% g96k -B/usr/local/dsp/new_libs/ file.c -o file.cld
```

Example 3-2. Using the **G96_EXEC_PREFIX** environment variable to have the same effect as Example 3.1, include in the .cshrc or .login files:

```
setenv G96_EXEC_PREFIX /usr/local/dsp/new_libs/
```

and then execute:

```
% g96k file.c -o file.cld
```

Example 3-3. To test a new version of the DSP96002 C preprocessor before permanent installation, install a new **cpp** program as **/tmp/new/cpp** and then execute:

```
% g96k -B/tmp/new/ testfile.c
```

-bPREFIX

Direct **g96k** to search for compilation phases, start-up files and libraries whose names are prefixed with the word **PREFIX**. Note that only **one** additional prefix can be specified. This is very similar to the **-B** option.

Example 3-4. Test a new version of the ANSI C library, lib96cy.clb, installed as **/usr/local/dsp/lib/new-lib96cy.clb**.

```
% g96k -bnew- file.c -o file.cld
```

Example 3-5. Test a new version of the DSP96KCC preprocessor before permanent installation.

Install the new **cpp** program as **/usr/local/dsp/bin/new-cpp** and

```
% g96k -bnew- testfile.c
```


-o FILE

Select FILE as the output file. This applies to all phases of the compiler. When the **-o** flag is not in use, the following file names are used by the compiler as the default output file names depending upon the compiler option as follows:

-E (preprocess only)	stdout
-S (compile only)	foo.asm
-c (no linkage)	foo.cln
complete process	a.cld

where stdout is “standard output” and prints to the console.

Example 3-6. Only generate a preprocessed file (do not invoke the compiler, assembler or linker) and put the results in file.i.

```
% g96k -E file.c -o file.i
```

Example 3-7. Compile file.c and generate the executable output file, fft.cld. If an output name is not given, the default file name is a.cld.

```
% g96k file.c -o fft.cld
```

-v

Verbose mode. The compiler control program announces to **stderr** all commands that it attempts to execute for each phase of the compilation process. This command is also used by the preprocessor to print the software version information. If the **-E** option is selected, **-v** will only enable the verbose mode, otherwise it will enable the verbose mode and print the version information.

3.2.1 Preprocessor Phase Options

The options listed below control the C preprocessor, which is run on each C source file before actual compilation. Some options described only make sense when used in combination with the **-E** option, as the requested preprocessor output may be unsuitable for actual compilation. The default option is to use ANSI C syntax. For example, if the **-IDIR** option is not specified then ANSI specifies that the current working directory will be searched first for user defined include files.

-C

Tell the preprocessor **not** to discard comments. This option is only valid when used in conjunction with the **-E** option.

Example 3-8. This example preprocesses a simple program, `foo.c`, without discarding comments.

```
% ls
  foo.c

% cat foo.c
/*
 * This COMMENT won't be deleted.
 */
main()
{
    printf("Hello, DSP96002\n");
}

% g96k -E -C foo.c
# 1 "foo.c"
/*
 * This COMMENT won't be deleted.
 */
main()
{
    printf("Hello, DSP96002\n");
}
```

-DMACRO

Define the preprocessor macro MACRO with a constant value of **1**. This is equivalent to making MACRO a constant set to one.

Example 3-9. Compile and run a simple program, dsp.c, and enable or disable a printed message depending on the macro definition given at the command line.

```
% ls
    dsp.c

% cat dsp.c
#include <stdio.h>
main()
{
    #ifdef sim96000
        printf("message: sim96000.\n");
    #else
        printf("message: ads96000.\n");
    #endif
}

% g96k -Dsim96000 dsp.c
% ls
    a.cld dsp.c

% run96sim a.cld
    message: sim96000.

% g96k dsp.c
% ls
    a.cld dsp.c
% run96sim a.cld
    message: ads96000.
```

-DMACRO=DEFN

Define preprocessor macro MACRO as DEFN.

Example 3-10. The program dsp.c uses the macro FROM_COMMAND_LINE which prints a message to the standard output using a message code given on the command line.

```
% ls
    dsp.c

% cat dsp.c
#include <stdio.h>
main()
{
    printf("message code: %d.\n", FROM_COMMAND_LINE);
}

% g96k -DFROM_COMMAND_LINE=96002 dsp.c
% ls
    a.cld dsp.c

% a.cld
    message code: 96002.
```

-E

The input source file will only be preprocessed through **cpp** and the output results will be sent to the standard output. See the **-o** option to save the output into a named file.

Example 3-11. This example shows how to preprocess the C source program foo.c and send the results to the standard output.

```
% ls
    foo.c

% cat foo.c
#define DELAY 1000
main()
{
    int cnt = DELAY;
    while(cnt--);
}

% g96k -E foo.c
# 1 "foo.c"

main()
{
    int cnt = 1000 ;
    while(cnt--);
}
```

Example 3-12. The **cpp** output can be saved into file “foo.i” by using the -o option.

```
% ls
foo.c

% cat foo.c
#define DELAY 1000
main()
{
    int cnt = DELAY;
    while(cnt--);
}

% g96k -E foo.c -o foo.i
% ls
foo.c foo.i

% cat foo.i
# 1 "foo.c"

main()
{
    int cnt = 1000 ;
    while(cnt--);
}
```

-IDIR

The control line of the C source program of the form

```
#include <filename>
```

will cause the replacement of that line by the entire contents of the file filename. This is normally referred to as **file inclusion**. The named file is searched for in a sequence of implementation-dependent directories[†]. The standard include directory for this compiler is /usr/local/dsp/include. Similarly, a control line of the form

```
#include "filename"
```

searches first in the current working directory, and if that fails, then searches as if the control line were `#include <filename>`.

The option **-IDIR** directs the C compiler to include the directory DIR in addition to the standard include directory. For the file inclusion `<filename>`, the compiler searches first in the DIR directory and if that fails, then searches /usr/local/dsp/include. For the file inclusion `"filename"`, the compiler searches first in the DIR directory and if that fails, then searches the current working directory, and if that fails also, then searches /usr/local/dsp/include.

Example 3-13. A delay program foo.c uses delay constant DELAY which is defined in the include file, myinclude.h. The program uses the control line `#include "myinclude.h"` to include the definition of the constant DELAY. Without any option the include file should be located in the currently working directory since it is not in the standard include directory. Assuming that the include file `"myinclude.h"` is desired to be in the directory `./inc`, the following sequence of the commands explains how the `-I` option is used to include the file myinclude.h in the `./inc` directory with the control line `#include "myinclude.h"` in the foo.c program.

```
% ls
foo.c inc/

% ls inc
myinclude.h

% cat foo.c
#include "myinclude.h" /* this is the control line to include it */
main()
{
    int cnt;
    cnt = DELAY;
```

[†]the current implementation of the C compiler only uses the standard directory /usr/local/dsp/includes for the directory.

```

        while(cnt--);
    }

% cat inc/myinclude.h
    #define DELAY 100

% g96k -I./inc foo.c
% ls
    a.cld foo.c inc/

```

-I-

This option is always used in conjunction with the **-IDIR** option and limits the file search to look for file inclusions **#include "filename"**, whereas **-IDIR** alone directs C compiler to search the directory DIR for both file inclusion **<filename>** and **"filename"**. Any directories specified with **-I** options before the **-I-** option are searched only for the case of **#include "filename"**; they are not searched for **#include <filename>**.

If additional directories are specified with **-I** options after the **-I-** option, these directories are searched for both **#include "filename"** and **#include <filename>** directives.

As an example, the sequence of the options

```
-IDIRA -I- -IDIRB
```

directs C compiler to use both the directories DIRA and DIRB for the file inclusion **"filename"** and DIRB only for file inclusion **<filename>**.

NOTE

The **-I-** option inhibits the use of the current directory as the first search directory for **#include "filename"**. There is no way to override this effect of **-I-**. However, the directory which is current when the compiler is invoked can be searched by using **-I**. This is different from the preprocessor's default search list, but it is often satisfactory. **-I-** does not inhibit the use of the standard system directories for header files. Thus, **-I-** and **-nostdinc** are independent.

Example 3-14. A test program file.c is used to test a file operation fopen(). The file include <stdio.h> is used as if it is in the standard include directory. The file is desired to be developed or debugged, and it is located in the user working directory ./mysys. This example shows how to use -IDIR and -I- combination to test file inclusion <filename>. Notice that the **-I./inc -I- -I./mysys** option specifies the inc directory only for the file inclusion "cnt.h" and mysys directory for the file inclusion <stdio.h>.

```
% ls -R
file.c    inc/      mysys/

inc:
cnt.h

mysys:
stdio.h

% cat file.c
#include <stdio.h>
#include "cnt.h"
main()
{
    int delay = COUNT;
    FILE *fp;
    fp = fopen("myfile", "w");
    while(--delay);
}
% cat inc/cnt.h
#define COUNT 25

% cat mysys/stdio.h
FILE *fopen(char *, char *); /* new function to develop */

% g96k -I./inc -I- -I./mysys file.c
% ls
a.cld    file.c    inc/      mysys/
```

-i FILE

Process FILE as an input, discarding the resulting output, before processing the regular input file. Because the output generated from FILE is discarded, the only effect of **-i FILE** is to make the macros defined in FILE available for use in the main input.

Example 3-15. The program `greeting.c` prints a simple message using the macro `MESSAGE`. The file `macros.c` contains the macro definition, i.e. the actual message. The only role of the file `macros.c` is to provide the macro definitions and will not affect any other code or data segments.

```
% ls
  macros.c  greeting.c

% cat macros.c
#define MESSAGE "Hello, world."

% cat greeting.c
#include <stdio.h>
main()
{
    printf("Greeting: %s\n", MESSAGE);
}

% g96k -i macros.c greeting.c
% run96sim a.cld
  Greeting: Hello, world.
```

-M

Cause the preprocessor to output the makefile rules to describe the dependencies of each source file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the files needed to generate the object target file. This rule may be a single line or may be continued with '\-newline if it is long. **-M** implies **-E** with makefile rules.

Example 3-16. The program `big.c`, which prints the larger of two integers, uses the macro `greater(x,y)` which is defined in the file `greater.h`. A command line output using the `-M` option can be used for makefile utilities. For more information on how to use this dependency check the make utility information in any UNIX utility manual.

```
% ls
  big.c    greater.h

% cat big.c
#include <stdio.h>
#include "greater.h"
main()
```

```
{
    printf("big:%d\n", greater(10,20));
}

% cat greater.h
#define greater(x,y) ((x)>(y)?(x):(y))
% g96k -M big.c
big.o : big.c /usr/local/dsp/include/stdio.h \
    /usr/local/dsp/include/stdarg.h greater.h
```

-MM

Like **-M** but the output mentions only the header files described in the **#include "FILE"** directive. System header files included with **#include <FILE>** are omitted. **-MM** implies **-E** with makefile rules.

Example 3-17. The program big.c, which prints the larger of two integers, uses the macro greater(x,y) defined in the file greater.h. The -MM option is used to generate a makefile rule. Notice that the rule that generates an output file appended by ".o" can be modified to generate ".cld" which is required for the Motorola Cross C Compiler.

```
% ls
big.c    greater.h

% cat big.c
#include <stdio.h>
#include "greater.h"
main()
{
    printf("big:%d\n", greater(10,20));
}

% cat greater.h
#define greater(x,y) ((x)>(y)?(x):(y))

% g96k -MM big.c
big.o : big.c greater.h

% ls
big.c    greater.h makefile  text

% cat makefile
a.cld : big.o
      g96k big.o
big.o : big.c greater.h
      g96k -c -o big.o big.c
```

```
% make
g96k -c -o big.o big.c
g96k big.o
```

```
% run96sim a.cld
big:20
```

-nostdinc

Do not search the standard system directories for header files. Only the directories specified with **-I** options (and the current directory, if appropriate) are searched.

Using both **-nostdinc** and **-I** options, all directories from the search path except those specified can be eliminated.

Example 3-18. A test program, test.c, is used to test a new version of the function printf() which has a new header file inc/stdio.h. The directive #include <stdio.h> causes the program to use stdio.h; however, it would normally find it in the standard search directory, /usr/local/dsp/include/. Using the -nostdinc option prevents the standard search directory from being searched and allows the -I option to point to the correct directory.

```
% ls
inc/  test.c

% ls inc
stdio.h

% cat test.c
#include <stdio.h>
main()
{
    printf("Hello, there.\n");
}

% cat inc/stdio.h
void printf(char *);

% g96k -nostdinc -I./inc -E test.c
# 1 "test.c"
# 1 "./inc/stdio.h" 1
void printf(char *);
# 1 "test.c" 2

main()
{
    printf("Hello, there.\n");
}
```

-pedantic

The **-pedantic** option is used by both the preprocessor and the compiler (see **-pedantic** in the [Compile Phase Options](#) section for an explanation of this option).

-P

Only run the source file through the C preprocessor. This option sends the output to a file with a **.i** suffix. This output file does not include **#** line information in the output file.

-v

Output preprocessor version information. The primary purpose of this command is to display the software version. This information is needed when calling the Motorola DSP helpline for assistance (see Appendix D — Additional Support for the helpline phone number). Although information pertaining to the internal flags and switch settings is included in this information, it is not intended for use by the programmer and may be misleading. This option is also used by the command program to initiate the verbose mode of operation.

Example 3-19. The **-v** option is selected using the control program **g96k**. The version numbers for **g96k**, **cpp** and **gcc96-cc1** are printed. This information is showing the commands that the control program invokes along with the selected options. In this case it is showing the default options plus the **-v** option. However, the user should not invoke these programs independently but should always use the control program to invoke them. This option is intended to be used to supply information which requesting applications assistance.

```
% ls
foo.c
```

```
% g96k -v foo.c†
g96k version Motorola Version: x.xx -- GNU x.xx.x
/usr/local/dsp/bin/cpp -v -undef -D__L_MEMORY -trigraphs -$ -
D__STRICT_ANSI__ -D__DSP96K__ -D__OPTIMIZE__ foo.c
/usr/tmp/cca00527.cpp
```

```
GNU CPP version x.xx.x
/usr/local/dsp/bin/gcc96-cc1 /usr/tmp/cca00527.cpp -ansi -fstrength-reduce
-quiet -dumpbase foo.c -O -version -o /usr/tmp/cca00527.asm
```

```
GNU C version x.xx.x Motorola DSP96000/1 Motorola Version: x.xx compiled
by GNU C version x.xx.x.
```

[†]The message may vary due to the continuous improvement of the compiler.

default target switches: -mdsp -mlinv-plus-biv-promotion -mdo-loop-generation -my-memory -mcall-overhead-reduction -mrep -mreload-cleanup -mnormalization-reduction

```
/usr/local/dsp/bin/asm96000 -c -B foo.cln -- /usr/tmp/cca00527.asm
```

```
/usr/local/dsp/bin/dsplnk -c -B a.cld -- /usr/local/dsp/lib/crt096y.cln foo.cln -L  
/usr/local/dsp/lib/lib96cy.clb
```

```
% ls  
a.cld foo.c
```

-UMACRO

Undefine macro MACRO.

Example 3-20. An application program, test.c, is being tested and some portions of the code need to be debugged. The flag DEBUG may be turned on or off through the command line with the -D and -U options respectively. This flag can then be used inside the program to enable/disable debugging features within the program.

```
% ls  
debug.c  
  
% cat debug.c  
#include <stdio.h>  
main()  
{  
#ifdef DEBUG  
printf("debug: a message.\n");  
#endif  
printf("normal operation.\n");  
}  
  
% g96k -UDEBUG debug.c  
% run96sim a.cld  
normal operation.
```

-Wcomment

Warn the user whenever the comment start sequence /* appears within a comment.

Example 3-21. A comment is enclosed with `/*` and `*/` and therefore is ignored by the preprocessor. Any number of leading `/*`'s are permitted within the comment and will not be reported; however, a warning message can be enabled by using the `-Wcomment` option.

```
% ls
  foo.c

% cat foo.c
/* foo.c */
main() { /* begin */
  int d = 1000; /* /* delay */
  while(d--); /* /* main /* loop */
} /* end */

% g96k -Wcomment foo.c
foo.c:3: warning: '/*' within comment
foo.c:4: warning: '/*' within comment
foo.c:4: warning: '/*' within comment

% g96k foo.c
```

-Wtrigraphs

Warn if any trigraphs are encountered (Trigraphs are sequences of three characters which are replaced by a single character. These trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO 646:1983, which is a subset of the seven-bit ASCII code set.).

3.2.2 Compile Phase Options

The default options are:

1. perform all machine dependent and independent optimizations,
2. do not generate debugging information,
3. locate data only in the L data memory space.

-fno-opt

Disable all optimizations.

-fno-peephole

Disable the peephole portion of optimization.

-fno-strength-reduce

Disable the optimizations of loop strength reduction and elimination of iteration variables as well as DSP96002 specific looping optimizations (**DO** instruction usage, etc.).

-fno-defer-pop

By default, the compiler will try to defer (delay) restoring the stack pointer upon the return of a function call. The purpose of deferring restoration of the stack pointer is to reduce code size and decrease execution time; however, the stack penetration may increase (see the DSP96002 User's Manual for information on stack overflow).

Examples of function calls that will **not** incur deferred pops whether or not the **-fno-defer-pop** option is specified are:

- calls as function arguments
- calls in conditional expressions
- calls inside a statement expression

-fforce-addr

Force memory address constants to be copied into registers before doing arithmetic on them. The code generated with this option may be better or it may be worse depending on the source code. This option forces memory addresses into registers which, in turn, may be handled as common sub-expressions.

-finline-functions

Attempt to insert all simple functions in-line into their callers. The compiler heuristically decides which functions are simple enough to merit this form of integration.

If all calls to a given function are inserted, and the function is declared **static**, then the function is no longer needed as a separate function and normally is not output as a separate function in assembly code.

-fcaller-saves

Enable values to be allocated in registers that will be overwritten by function calls by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

-fkeep-inline-functions

Output a separate run-time callable version of the function even if all calls to a given function are integrated and the function is declared **static**.

-fwritable-strings

Store string constants in the writable data segment without making them unique. This is for compatibility with old programs which assume they can write into string constants. Writing into string constants is poor technique; constants should be constant.

Compile Phase Options

-fcond-mismatch

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

-fvolatile

Consider all memory references through pointers to be volatile. Refer to Section 5.2.7 for volatile variables.

-ffixed-REG

Treat the register named REG as a fixed register; generated code should never refer to it (except perhaps as a stack or frame pointer). Legal values for REG are:

r1, r2, r3, r4, r5, r7

This flag should be used very sparingly as it can have devastating results on the code generated.

Example 3-22. Reserve **r4** for later special purpose.

```
% 96k -O -ffixed-r4 file.c -o file.cld
```

Caution

C code that utilizes library code can produce non-deterministic results, as the libraries have been written to utilize the complete set of registers.

-g

Produce COFF debugging information for **gdb96**, the GDB cross debugger for the DSP96002.

A key feature afforded by the use of the GNU C compiler (**g96k**) teamed with the GNU source level debugger (gdb96) is that the programmer is allowed to generate optimized code with debug information (**select options -g -O**) making it possible for the programmer to debug optimized code directly. Due to the optimizations performed, it is possible that variables will not be defined (unused variable elimination), statements may not be executed (dead code elimination), and code may be executed early (code motion). This is a partial list of the oddities that may be encountered when debugging optimized code. However, the improved code performance due to optimization normally outweighs the problems encountered.

Example 3-23. The program `foo.c` has a bug. In this application the line `i *= 2` should be `i += 2`. In order to use the symbolic debugger, `gdb96`, to locate this problem, the `-g` option is used when `foo.c` is compiled. This causes the compiler to generate additional information used by the debugger.

```
% ls
  foo.c

% cat foo.c
main()
{
    int i = 100;
    i *= 2;
}

% g96k -g foo.c
% ls
  a.cld foo.c
```

-O

Perform machine dependent and independent optimizations. This is the **default** mode of the compiler.

Invoking the compiler with the optimizer may cause compile times to increase and require more system memory.

Invoking the compiler without the optimizer should be done only when the programmer requires additional flexibility while debugging code. An example of such flexibility includes the ability to assign new values to declared C variables. Additionally, non-optimized code takes register usage clues from the storage class specifier **register**, something not done with the optimizer invoked.

Disabling the optimizer is done via **-f** options listed above.

-mno-dsp-optimization

Disables all Motorola optimizer enhancements.

-mno-do-loop-generation

Disable **DO** instruction usage by optimizer.

-mno-biv-plus-linv-promotion

Disable the promotion of address expressions to address registers within loops. This optimization transforms array base address plus induction variable expressions into auto-increment/decrement style memory references.

Compile Phase Options

-mstack_check

Generate extra run-time code to check for run-time stack collision with the heap. This option causes run-time execution times to increase dramatically.

-mx-memory

Direct the compiler to locate data in the X data memory space. Memory modes cannot be mixed, i.e. only one of **-mx-memory**, **-my-memory** or **-ml-memory** may be selected.

Example 3-24. An application is programmed to utilize only the DSP96002 X data memory space and therefore must be compiled using the **-mx-memory** option.

```
% ls
  x.c

% cat x.c
void function(int a, int b);
int X;
main()
{
  int arg1,arg2;
  function(arg1, arg2);
}
void function(int a, int b)
{
  X = a + b;
}

% g96k -S -mx-memory x.c
% ls
  x.asm x.c
```

-my-memory

Direct the compiler to locate data in the Y data memory space. This is the default memory mode. Memory modes cannot be mixed, i.e. only one of **-mx-memory**, **-my-memory** or **-ml-memory** may be selected.

-ml-memory

Direct the compiler to locate data in the L data memory space. This has 2 side effects.

1. A performance increase for 48-bit data code (**long, float, and double**).
2. This requires that the X and Y memory spaces be evenly populated.

Memory modes cannot be mixed, i.e. only one of **-mx-memory**, **-my-memory** or **-ml-memory** may be selected.

-pedantic

Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

Without this option, certain GNU extensions and traditional C features are supported. With this option, they are rejected. Valid ANSI standard C programs will compile properly with or without this option.

-pedantic does not cause warning messages for use of the alternate keywords whose names begin and end with “__”.

-Q

Direct the compiler to execute in verbose mode.

-S

Compile to DSP96002 assembly code with the original C source lines as comments but do not assemble. The assembly language output is placed into a file suffixed **.asm**.

Example 3-25. Generate an optimized assembly language file (test.asm) of the C source program (test.c).

```
% ls
    test.c

% cat test.c
#include <stdio.h>
main()
{
    int i = 100;
    printf("value:%d\n", i++);
}

% g96k -S test.c
% ls
    test.asm test.c
```

Example 3-26. Generate an optimized assembly language file test.asm.

```
% g96k -O -S test.c
```

-w

Inhibit all warning messages.

-W

Print extra warning messages for the following events:

- An automatic variable is used without first being initialized.

This warning is possible only when the optimizer is invoked during compilation (default). The optimizer generates the data flow information required for reporting.

This warning will only occur for variables that are candidates for register promotion. Therefore, they do not occur for a variable that is declared **volatile**, whose address is taken, or whose size is other than 1 or 2 words (integral and float data types). Warnings will not occur for structures, unions or arrays, even when they are in registers.

There may be no warning about a variable that is used only to compute a value that is never used because such computations may be deleted by data flow analysis before the warnings are printed.

Spurious warnings may be avoided by declaring functions that do not return as **volatile**.

- A nonvolatile automatic variable may be changed by a call to **longjmp**.

This warning also requires that the optimizer be invoked.

The compiler sees only the calls to **setjmp**. It cannot know where **longjmp** will be called; in fact, a signal handler could call it at any point in the code. As a result, a warning may be issued even when there is no problem because **longjmp** cannot be called at the place which would cause a problem.

A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
    if (a > 0)
        return a;
}
```

Spurious warnings can occur because GNU CC does not realize that certain functions (including 'abort' and 'longjmp') will never return.

- An expression-statement contains no side effects.

Example 3-27. Extra warning messages are wanted to help find potential problems in a test function, `foo()`, which is programmed to return a value only if $a > 0$.

```
% ls
  foo.c

% cat foo.c
int foo(int);
main()
{
    int i;
    foo(i);
}
int foo(a)
{
    if(a > 0)
        return a;
}

% g96k -W foo.c
foo.c: In function main:
foo.c:4: warning: 'i' may be used uninitialized in this function
foo.c: In function foo:
foo.c:11: warning: this function may return with or without a value
```

-Wimplicit

Warn whenever a function is implicitly declared.

Example 3-28. The function `foo()` is declared implicitly in the program `foo.c`, the `-Wimplicit` option will generate a warning message for this situation.

```
% ls
  foo.c

% cat foo.c
main()
{
    foo();
}
int foo(){}

% g96k -Wimplicit foo.c
foo.c: In function main:
foo.c:3: warning: implicit declaration of function 'foo'

% ls
  a.cld foo.c
```

-Wreturn-type

Warn whenever a function is defined with a return-type that defaults to **int**. Also warn about any **return** statement with no return-value in a function whose return-type is not **void**.

Example 3-29. The function `foo()` is declared as a function that should return an integer but in this case does not return an integer. The `-Wreturn-type` option generates a warning message in this situation.

```
% ls
  foo.c

% cat foo.c
  int foo(), main();
  int main()
  {
    return foo();
  }
  int foo(){}

% g96k -Wreturn-type foo.c
foo.c: In function foo:
foo.c:6: warning: control reaches end of non-void function

% ls
  a.cld foo.c
```

-Wunused

Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined and whenever a statement computes a result that is explicitly not used.

Example 3-30. The file `foo.c` contains an undefined static function, unused local variable, and a dead statement. The `-Wunused` option will issue warnings to indicate these situations.

```
% ls
  foo.c

% cat foo.c
  static int foo();
  main()
  {
    int x;
    2+3;
  }
```

```
% g96k -Wunused foo.c
foo.c: In function main:
foo.c:5: warning: statement with no effect
foo.c:4: warning: unused variable 'x'
foo.c: At top level:
foo.c:1: warning: 'foo' declared but never defined

% ls
a.cld foo.c
```

-Wswitch

Warn whenever a **switch** statement has an enumeration type of index and lacks a **case** for one or more of the named codes of that enumeration. (The presence of a **default** label prevents this warning.) **case** labels outside the enumeration range also provoke warnings when this option is used.

-Wall

All of the above **-W** options combined. The remaining **-W** options described below are **not** implied by **-Wall** because certain kinds of useful macros are almost impossible to write without causing those warnings.

-Wshadow

Warn whenever a local variable shadows another local variable.

-Wid-clash-LEN

Warn whenever two distinct identifiers match in the first **LEN** characters. This may help prepare a program that will compile with certain obsolete compilers.

-Wpointer-arith

Warn about anything that depends on the **sizeof** a function type or of **void**. GNU C assigns these types a size of 1, for convenience in calculations with **void *** pointers and pointers to functions.

-Wcast-qual

Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a **const char *** is cast to an ordinary **char ***.

-Wwrite-strings

Give string constants the type **const char[LENGTH]** so that copying the address of one into a non-**const char *** pointer will generate a warning. These warnings help at compile time to find code that can try to write into a string constant, but only if **const** in declarations and prototypes have been used carefully.

3.2.3 Assemble Phase Options

This group of assemble phase options is the sub-set of the available assembler options that are compiler oriented (see the Motorola DSP96000 Macro Assembler Reference Manual for a complete option list). The default option is to add to the standard search list the directory that the C compiler writes its output into and then search that directory first.

-asm string

Pass the argument string directly to **asm96000**, the DSP96002 assembler.

Example 3-31. Pass a single option to the assembler.

```
% g96k -asm '-v' file.c
```

Example 3-32. Pass multiple options to the assembler.

```
% g96k -asm '-v -OS,CRE' file.c
```

You may have to use double quotes instead of single quotes around the string for the PC version of the compiler.

-c

Compile and/or assemble the source files, suppressing the link phase. This option generates corresponding output files suffixed “**.cln**”. Affected input files are suffixed with “**.c**” and “**.asm**”.

3.2.4 Link Phase Options

The options listed below control the link phase of the compilation process. This group of link phase options is the sub-set of the available linker options that are compiler oriented (see the Motorola DSP96000 Linker/Librarian Reference Manual for a complete option list). The **-crt** and **-l** options locate the file provided as an argument by searching a standard list of directories, see Section 3.1 for this directory list. The default option is to add the C compiler output directory into the standard search list and search that directory first.

-crt file

Replace the default start-up file (crt096.cln) with file. **g96k** searches the standard list of directories to find the start-up file. In addition, any directory defined using the **-B** option or the **G96_EXEC_PREFIX** environment variable will be searched. For additional information, see Chapter 6.3.

-j string

Pass the argument string directly to **dspInk**, the DSP96002 linker.

Example 3-33. Pass a single option to the linker.

```
% g96k -j '-v' file.c
```

Example 3-34. Pass multiple options to the linker.

```
% g96k -j '-v -i' file.c
```

You may have to use double quotes instead of single quotes around the string for the PC version of the compiler.

-lLIBRARY

Search the standard list of directories for a library file named **libLIBRARY.clb**. The linker automatically expands LIBRARY from the option command into **libLIBRARY.clb** and uses this file as if it had been specified precisely by name.

Example 3-35. Compile the source code using the special dsp application library. Searching the standard list of directories for a library named **libdspaps.clb**.

```
% g96k -O file.c -ldspaps
```

-r CTLFILE

Search the standard list of directories for the memory control file CTLFILE to be passed as an argument to the DSP96002 relocatable linker. This control file will be used as a table to locate object files sections to be linked. For more detailed information, see the **-R** options and the section on “Memory Control File” in the Motorola Linker/Librarian Reference Manual.

Chapter 4

About g96k

4.1 Introduction

The DSP96002 digital signal processor is designed to execute DSP oriented calculations as fast as possible and, as a result, it has an unconventional architecture for an ANSI C compiler implementation. Because of this unconventional architecture, there are characteristics of the compiler and the code generated by the compiler that the programmer must understand in order to take full advantage of the DSP96KCC programming environment. The following sections provide important information on data types, storage classes, memory and register usage in addition to other topics which will be useful to the DSP96002 application developer programming in C.

4.2 Identifiers

An identifier is defined as a sequence of letters, digits and underscore characters ('_'). The first character must be a letter or underscore. DSP96KCC identifier length limits are listed in Table 4-1.

Table 4-1 Identifier Length Limits

Identifier Storage Class	Length
Global/Static (External Linkage)	255
Auto	unlimited

4.3 Predefined Preprocessor Macro Names

DSP96KCC supports and expands all ANSI defined macros and three additional non-ANSI predefined macro names. Table 4-2 lists the macros and their explanation.

4.4 Data Types and Sizes

Due to the word orientation of the DSP96002 (32-bit words), all data types are aligned on word boundaries. This has several side effects, one of which is that **sizeof** (char) is equal to **sizeof**(int).

Table 4-2 Predefined Macro List and Explanation

MACRO	ANSI	Explanation
<code>__LINE__</code>	YES	The line number of the current source line (a decimal constant).
<code>__FILE__</code>	YES	The name of the source file (a character string).
<code>__DATE__</code>	YES	The compilation date (a character string of the form "Mmm dd yyyy" e.g., Jul 22 1991).
<code>__TIME__</code>	YES	The compilation time (a character string of the form "hh:mm:ss").
<code>__STDC__</code>	YES	Decimal constant 1, indicates ANSI conformation.
<code>__DSP96K__</code>	NO	Decimal constant 1, indicates that code is being generated for the DSP96002 Digital Signal Processor.
<code>__VERSION__</code>	NO	The version number of the compiler (a character string of the form "d.dd.d").
<code>__INCLUDE_LEVEL__</code>	NO	Decimal constant, indicates the current depth of file inclusion.

4.4.1 Integral Data Types

The type **char**, **unsigned char**, **signed char**, **short int**, **int**, **long int** and the enumerated types comprise the integral data types. The type sizes and ranges are defined in Table 4-3.

Table 4-3 Integral Data Type Sizes and Ranges

Data Type	Size (words)	Min value	Max value
char	1	-2147483648	2147483647
unsigned char	1	0	0xFFFFFFFF
short	1	-2147483648	2147483647
unsigned short	1	0	0xFFFFFFFF
int	1	-2147483648	2147483647
unsigned int	1	0	0xFFFFFFFF
long	1	-2147483648	2147483647
unsigned long	1	0	0xFFFFFFFF

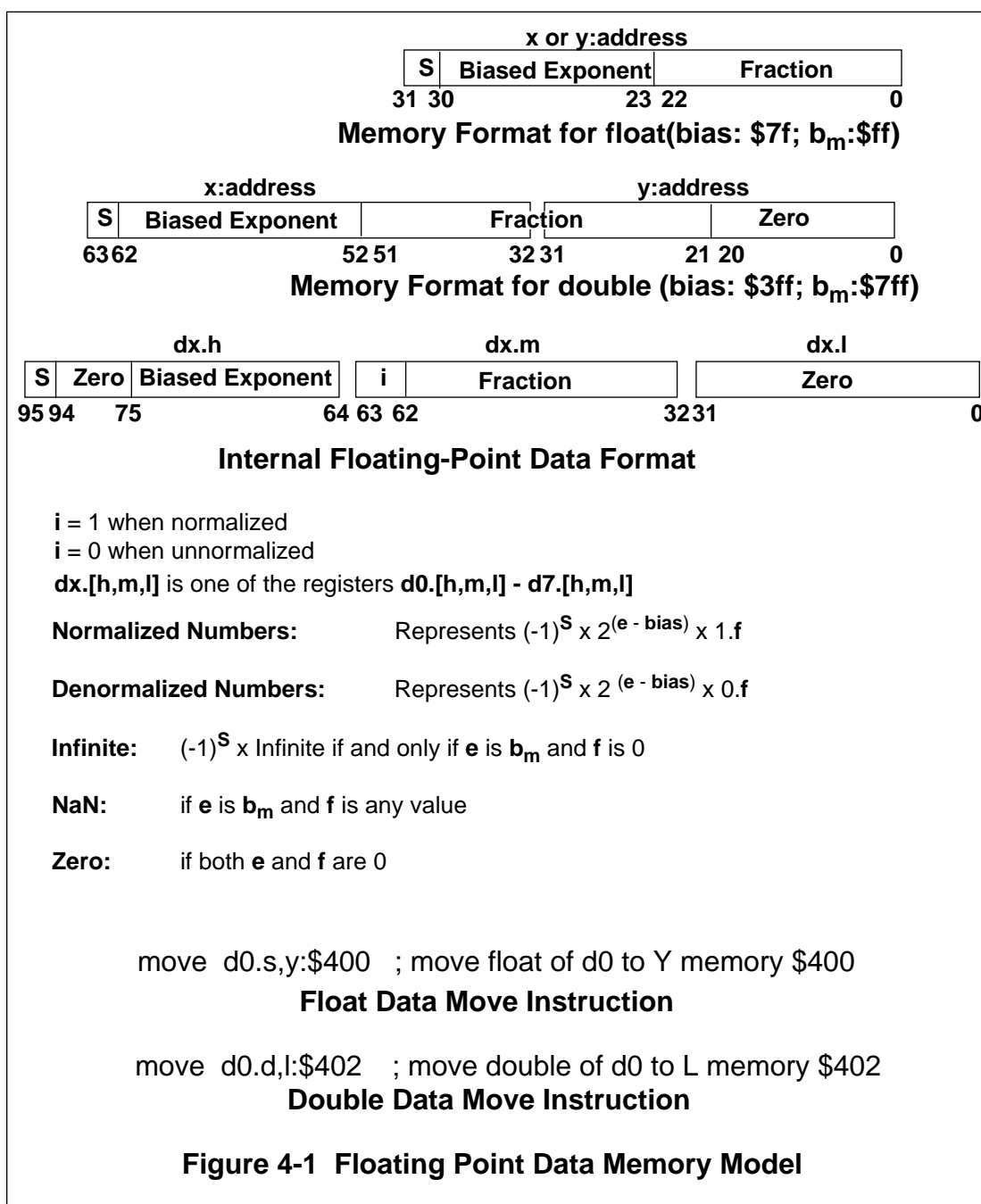
4.4.2 Floating-point Types

In DSP96KCC, the C data type **float** is implemented as single precision complying with the IEEE Standard 754. The C data type **double** is implemented as single extended precision defined in DSP96002 architecture (see Chapter 5 and Appendix C of DSP96002 User's Manual). Their sizes and ranges are listed in Table 4-4.

Table 4-4 Floating-point Data Type Sizes and Ranges

Data Type	Size (words)	Min value	Max value
float	1	1.175494351e-38	3.402823466e+38
double	2	2.225074e-308	1.79769e+308

The floating point operation is done through the 96 bit registers d0 - d7 (concatenation of dx.h, dx.m, and dx.l). The result of the floating point operation can be stored in the memory using the move instruction of DSP96002.



The float data can be stored in the 32 bit memory (either X or Y memory) as shown in the Figure 4-2, and the single precision value (float) of the 96 bit register is referenced with the register extension “.s”. Similarly, the double data can be stored in the 64 bit L memory (concatenation of X and Y memory space) as shown in the Figure 4-2. The single extended precision data (double) of the 96 bit register can be accessed with the extension “.d”. An example to store the float in the register d0 to the memory location y:\$400 is

```
move      d0.s,y:$400      ; move float in d0 register
```

and to store the double in the register d0 to the memory location l:\$402 is

```
move      d0.d,l:$402      ; move double in d0 register
```

When the data movement of the single extended precision data of the 96 bit register to the 64 bit memory occurs, the actual 44 bit portion of the 96 bit register will be saved in the L memory and the rest of the bits will be set zero.

4.4.3 Pointer Types

The size of the pointers in DSP96KCC is 32-bit wide and its range is listed in the Table 4-5.

Table 4-5 Pointer Size and Range

Data Type	Size (words)	Min value	Max value
pointers	1	0	0xFFFFFFFF

4.5 Register Usage

The DSP96002 digital signal processor register set is shown in Table 4-6. DSP96KCC uses all of the registers listed in Table 4-6 with the exception of the m_n address modifier registers.

Table 4-6 DSP96002 Processor Registers

Data ALU	
d0.l-d9.l-	Integer registers (32-bits)
d0.s-d9.s-	Single precision float registers (32-bits)
d0.d-d9.d-	Single extended precision float registers (C double)
d0.[m,h]	
	- Integer scratch registers when not in use by float registers (32-bits)
d9.[m,h]	
Address ALU	
r _n	- Address Registers r0-r7 (32-bits)
n _n	- Address Offset Registers n0-n7 (32-bits)
m _n	- Address Modifier Registers m0-m7 (32-bits)

Caution

The **m_n address modifier registers** are not used by DSP96KCC. While assembly code can access and use these registers, the programmer **must** restore them to their previous state (\$FFFFFFFF) before returning control to DSP96KCC. Failing to do so will cause **unpredictable errors** when DSP96KCC accesses these registers.

Each register in the set performs a specific function, thus requiring the programmer to preserve any register that is to be directly used in in-line and out-of-line assembly language code (see Chapter 5, Mixing C and Assembly Language). Table 4-7 outlines the compiler's usage of each register.

Table 4-7 DSP96KCCRegisters and Usage

Register	Usage
r0	Frame Pointer
r6	Stack Pointer
r7	Structure return address. When subroutine does not return a structure, r7 is allocated by the optimizer for register promotion.
r1 - r5, r7	Register promotion by the optimizer
n0 - n7	Code generator temporary
m0 - m7	Used by compiler; keep this as \$FFFFFFF
d0.l	32-bit function return value, or param1
d0.s	float function return value, or param1
d0.d	double function return value, or param1
d1.l	param2 (32 bit value; see Figure 4-3)
d1.s	param2 (float value; see Figure 4-3)
d1.d	param2 (double; see Figure 4-3)
d2.[lmh] - d7.[lmh]	32-bit register promotion by optimizer
d2.s - d7.s	float type register promotion by optimizer
d2.d - d7.d	double type register promotion by optimizer
d8.[lmh], d9.[lmh]	Code generator temporary

4.6 Memory Usage

The DSP96002 memory can be partitioned in several ways to provide high-speed operation and additional off-chip memory expansion. Program and data memory are separate and the data memory is divided into two separate 32-bit wide memory spaces, **X** and **Y** which can be combined to form a third 64-bit wide memory space, **L**.

DSP96KCC generates code that can utilize X or Y or L data memory. By default, the L memory model is selected. Each separate memory model is supported by separate run-time libraries that have been written and compiled/assembled to handle each memory space. See Chapter 3, Control Program Options, for examples of the mx-memory, my-memory and ml-memory command-line options for changing target memory spaces.

By default, the compiler expects that each memory space is fully populated and several global C variables are defined (see Chapter 6 — Software-Hardware Integration for information about customizing the memory configuration). Figure 4-2 and Figure 4-4 illustrate the default program and data memory configuration.

The L memory model is recommended for best performance when long, double data types are extensively used.

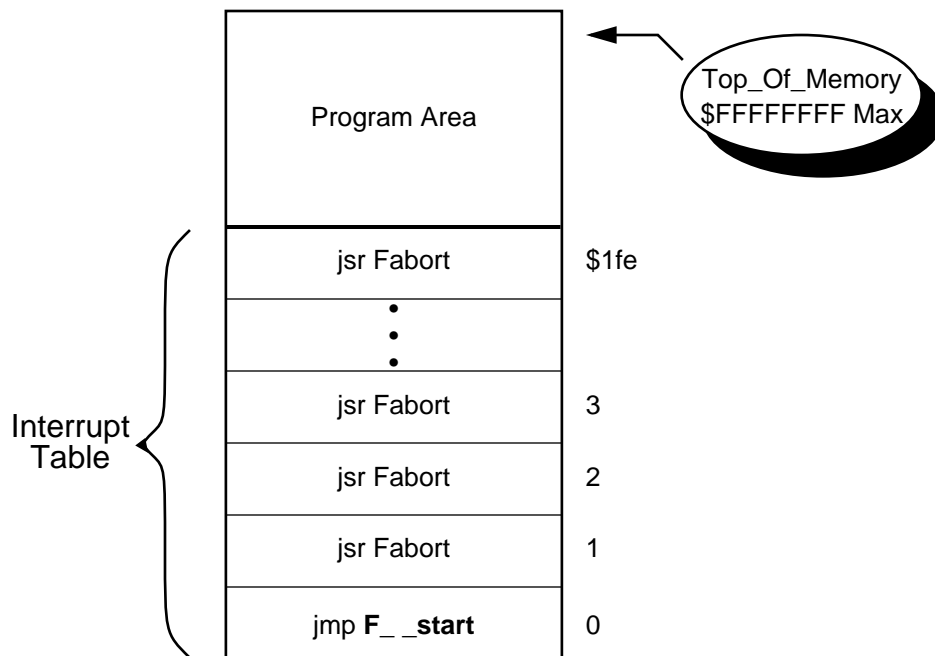


Figure 4-2 Default Program Memory Configuration

4.6.1 Activation Record

An activation record is the run-time representation of a C subroutine. A typical DSP96KCC activation record consists of the following elements and is illustrated in Figure 4-3:

1. Parameter data space. Information passed to C subroutines is stored in a parameter data space which is similar to the local data space (see Figure 4-3). However, the data is in reverse order and each parameter is referenced via a negative offset from the frame pointer. Actual parameters are pushed onto the activation record in reverse order by the calling subroutine. If the size of the first two parameter is one word, then they will be located in the registers d0 and d1.
2. Old frame pointer. The old frame pointer provides a dynamic link to the calling subroutine's activation record. Once the called subroutine has completed execution, the frame pointer will be updated with this value.
3. Return address — which is pushed on the DSP's system stack high (**ssh**) register. This is the return address to the calling subroutine. The return address is not saved for subroutines that have been determined to be a **leaf**. A leaf subroutine is one that makes no subroutine calls.
4. Local data space. The location of C variables that have a lifetime that extends

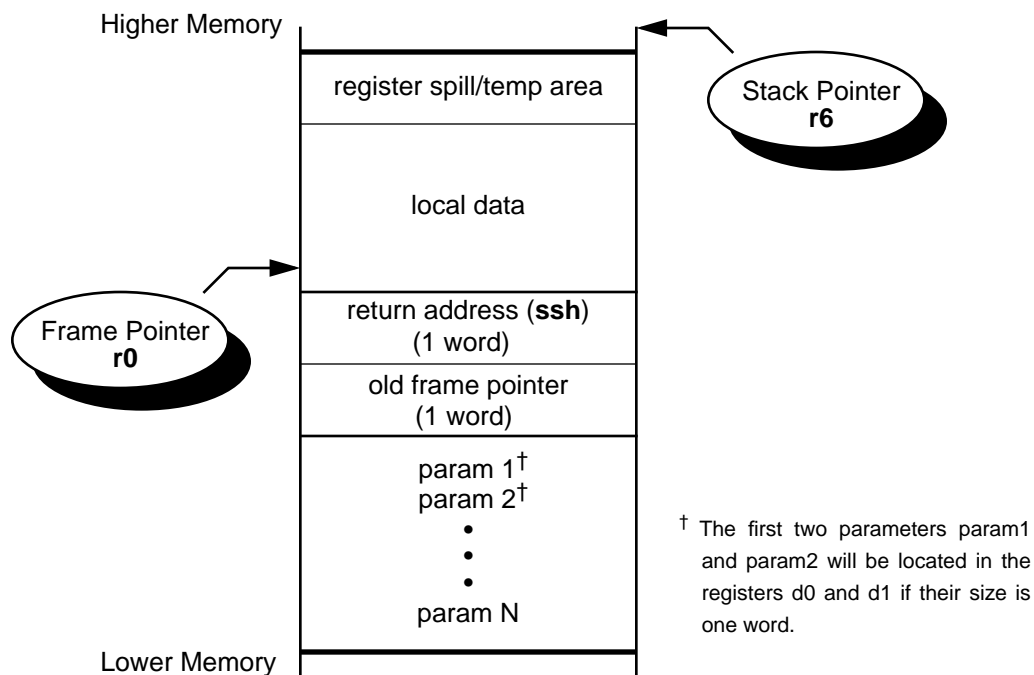


Figure 4-3 Activation Record

only as long as the subroutine is active and that could not be explicitly promoted to register storage class by the optimizer.

Note: The frame pointer (**r0**) points to the first element in the local data space.

5. Register spill and compiler temporary space. This area is utilized by the compiler to store intermediate results and preserve registers.

Note: The Stack Pointer (**r6**) points to the **next** available data memory location.

Each subroutine called puts a new copy of the subroutine activation record in the run-time stack and returning from the subroutine removes the activation record. The run-time stack is described in Figure 4-4, Default Data Memory Configuration. The variables shown in the bottom of the “X or Y memory option selected” memory are controlled by the crt0 file. When the L memory option is selected, the heap, run-time stack, global/static data and data that is more than 32 bits in length will occupy one word in L memory i.e., 64-bit memory. 16-bit and 32-bit data memory will occupy only one word in Y memory. DSIZE is set by the linker and points to the top address of the global and static data. DSIZE is used by the crt0 file as the default initial stack pointer.

The dynamic run-time stack growth is illustrated in Figure 4-5. In this example, there is one activation record as execution of the sample C code begins. This activation record is

X or Y Memory Option Selected

L (X:Y) Memory Option Selected

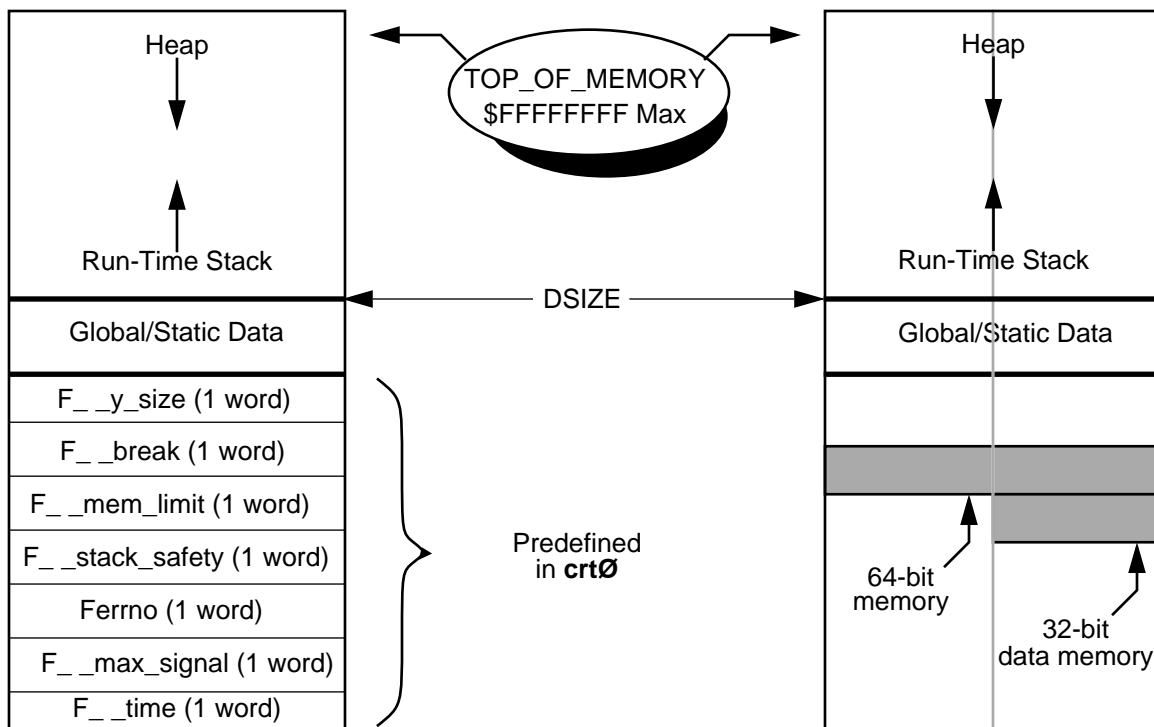


Figure 4-4 Default Data Memory Configuration

```
main()
{
    func_1();
}
```

Sample C code

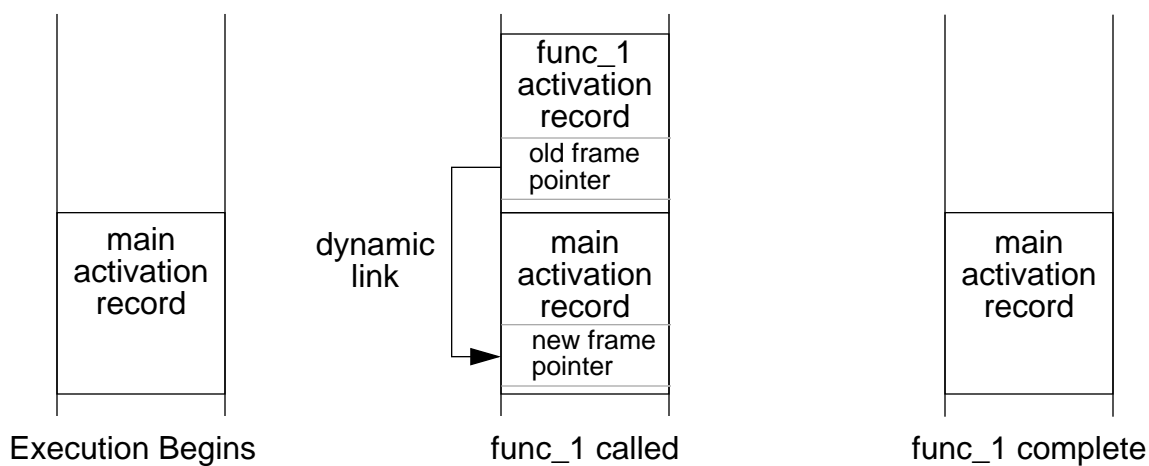


Figure 4-5 Run-time Stack Growth

pushed onto the stack and a new activation record is built with a dynamic link to the old

frame pointer. When the function returns, the function's activation record is popped and the original activation record is restored to its original place on the stack.

4.6.2 Global/Static Data

By default, global and static data elements are located below the run-time stack and each element is referenced by a unique label that is known at compile-time (see Chapter 6, Software-Hardware Integration for additional information).

4.7 Compiler Naming Conventions

The compiler uses five different internal label formats and a special section naming format. These six separate formats simplify the procedures to combine hand written assembly language and C language statements. Use of these formats also makes compiler generated assembly language listings easier to read. It is strongly recommended that the programmer avoid using labels with these formats.

L#	Local labels. Generally the targets of conditional and unconditional jumps. Where # is a decimal digit.
LC#	String Constant labels. The data memory location for C string constants, such as “ string constant ”.
F<identifier>	Global C variables, global subroutines, static C variables and static subroutines. A static C variable or subroutine is one which is not visible to any C code outside the file in which it has been declared, thus making it possible to reuse variable names across file boundaries. Where identifier is the variable or subroutine name.
F__<identifier>#	Variables static to a function.
ASM_APP_#	In-line assembly code delimiters. Required to allow the programmer to define and use local labels (labels beginning with an underscore character ‘_’).
<filename_c>	Section names. The contents of each assembly language file generated by the compiler are contained in a unique <u>section</u> . Where filename is the file name minus any ‘.’ extensions.

4.8 Subroutine Call Sequence

Each time a C language subroutine is called, a strict calling convention is followed. The subroutine calling sequence is broken down into three sub-sequences that are strictly defined. The three sub-sequences are caller, callee and return sequence.

Note: This calling convention must be followed when writing in-line or

out-of-line assembly language subroutines that call subroutines written in C.

4.8.1 Caller Sequence

The caller portion of the subroutine calling sequence is responsible for:

1. pushing arguments onto the activation record (in reverse order),
2. actual subroutine call (**jsr**),
3. stack pointer adjustment.

Additional caller sequence when the subroutine called will return a **structure**:

4. allocate space in the caller's activation record to store the return structure,
5. pass the address of the buffer allocated in the step 4 through address register **r7** (see Table 4-7).

4.8.2 Callee Sequence

During the initial portion of the subroutine calling sequence, the callee is responsible for:

1. saving return address (**ssh**) and the old frame pointer (**r0**),
2. updating frame and stack pointers,
3. saving the following registers, as required, **d0.[lmh] - d7.[lmh]**, **r1 - r5** and **r7**.

4.8.3 Return Sequence

During the final portion of the subroutine calling sequence, the callee is responsible for:

1. placing the return value in register **d0.l**.
2. testing the return value. This optimizes the case where function calls are arguments to conditional operators.

Additional callee sequence when the subroutine called will return a **structure**:

3. the return value is not passed in register **d0.l**. A copy of the return structure is placed into the space allocated in the caller's activation record and pointed to by address register **r7**.

4.9 Run-time Safety

DSP96KCC provides two methods for providing run-time memory utilization checks. The first method, memory allocation check, is automatic. The second method, run-time stack probes, is provided by selecting the command-line argument **-mstack_check**.

4.9.1 Memory Allocation Checks

Memory allocation checks are provided during every call to the run-time library routines **malloc**, **calloc** and **realloc**. This automatic run-time check determines when the **heap** is about to collide with the **run-time stack**. When this occurs, the library routine returns a NULL pointer and sets the global variable **errno** to **ENOMEM** defined in the `stdio.h` file. See Appendix A, Programming Support.

4.9.2 Run-time Stack Checks

If the programmer does not use **malloc**, **calloc** or **realloc** there is no automatic run-time collision check. A tool is provided to allow the programmer to check the code during debugging to see if a stack collision is likely to occur when the code is executing normally. By selecting the **-mstack_check** option on the command-line, the compiler is instructed to generate extra code to watch the stack and heap and detect when the run-time stack is about to collide with the heap. This may be important when writing code for embedded applications. When/if a stack collision occurs, the execution device, such as **run96sim**, will report that the collision occurred.

Note: All run-time libraries provided have been compiled/assembled without the stack checking option. Thus it is possible to have a run-time stack/heap collision during execution of library routines.

4.10 Optimization Techniques Implemented

This section provides a brief overview of the optimization techniques currently implemented by DSP96KCC. DSP96KCC implements most machine-independent C language optimization techniques as well as machine-specific optimizations. By default, the control program g96k enables all levels of optimization (see Chapter 3, Compiler Operation, for command-line options to disable all or part of the optimizer).

4.10.1 Register Promotion and Lifetime Analysis

The compiler automatically identifies all variables that are candidates for promotion to the register storage class. Using standard data flow techniques, lifetime analysis on variables is performed to determine the best candidates for promotion. When variable lifetimes do not overlap, more than one variable may be promoted to a single register.

This optimization is especially powerful when generating code for array references inside loops, for example:

```
for ( i = 0 ; i < 10 ; i ++ )
{
    array_1[ i ] = array_2[ i ];
}
```

becomes:

```
for ( i = 0 ; i < 10 ; i ++ )
{
    *aptr_1++ = *aptr_2++;
}
```

4.10.2 Common Sub-expression Elimination

A **Common Sub-Expression**, or CSE, is created when two or more statements compute the same value. When CSEs are detected during data flow analysis, the compiler eliminates all but one of the computations and propagates the result. A classic example of a CSE is the array element assignment from another array,

```
array_1[index + 1] = array_2[index + 1];
```

where the expression `index + 1` is the CSE.

This optimization is especially effective as CSEs become candidates for register promotion.

4.10.3 Constant Propagation/Folding

Propagation of constants is detected during data flow analysis and is simply the replacement of variable references with constant references when possible. For example,

```
a = 3;
/* block of C code with no references to a */
func_call( a + 709 );
```

becomes:

```
/* block of C code */
func_call( 3 + 709 );
```

Constant folding is the replacement of run-time computations with compile-time computations. This optimization works well with constant propagation, as it exposes many new opportunities for folding.

The example above would be further transformed to:

```
/* block of C code */
func_call( 712 );
```

4.10.4 Dead Code Elimination

During data flow analysis, the compiler detects when the results of C expressions are never used. When this is detected, the offending C statements are eliminated from code generation. This includes (1) the initialization of variables that are not referenced in the subroutine and (2) the situation where the variables next reference is not an assignment.

To guarantee code generation, a **volatile** type specifier can be used to declare the left side of the expression.

```
main()
{
    int volatile i = 0, j = 1;
}
```

The example above generates code to initialize variables i and j even though they are not used anywhere else. Without the key word **volatile**, the optimizing C compiler will eliminate the two local variables because they are not referenced anywhere in the main function.

4.10.5 Tail Merging

When two or more conditional blocks of code have similar ending sequences, the sections of code are rewritten to generate similar code only once.

This is a space saving optimization technique. For example:

<pre>if (a > b) { b = a; func(a); } else func(a);</pre>	becomes	<pre>if (a > b) { b = a; } func(a);</pre>
--	----------------	---

4.10.6 Strength Reduction

Strength reduction replaces expensive operators with less expensive operators. This optimization method is very machine specific. For instance, a popular strength reduction for many compilers is to replace a multiplication by a constant with a series of shifts, additions and subtractions. The exact opposite is the case on the DSP96002, however, since a series of left shifts is replaced with a single multiply by a constant power of 2.

4.10.7 Loop Invariant Code Motion

Loop Invariant Code Motion is a method in which all C expressions that yield the same

result each time through the loop are moved outside of the loop and are executed once prior to entering the loop.

4.10.8 Hardware DO Loop Instruction

The DSP96002 provides a method in hardware to perform zero overhead looping via the **DO** instruction. DSP96KCC will exchange the standard compare/conditional jump sequence with a single **DO** instruction (this is called do loop promotion) when the following conditions are met:

1. The body of the loop may contain no subroutine calls.
2. The loop must be entered from the top, i.e., no **goto** entries.
3. No conditional exits from the loop are allowed.
4. The loop's induction variable must not be altered in the body of the loop and may only be incremented or decremented in the **for** statement as shown in this example:

```
for ( i = a ; i < b ; i++ )
```

4.10.9 Loop Rotation

Loop rotation is the elimination of the code generated to check the loop's entry conditions. When a loop fails to qualify for do loop promotion i.e., it does not meet the four conditions listed above, it will qualify for loop rotation if the length of the loop is known at compile-time, for example:

```
for ( i = 0 ; i < 10 ; i++ )
```

The loop created with this **for** statement will always be executed at least one time. Therefore, the "is i < 10?" test does not have to be run the first time through the loop and as a result, can be eliminated during the first pass through the loop only. If the result of the first test cannot be predetermined then it cannot be eliminated. In the example below, the number of loops is a variable (and therefore cannot be predetermined) that may equal zero.

```
for ( i = 0 ; i < j ; i++ )
```

4.10.10 Jump Optimizations

All cases of jumps (conditional and unconditional) to unconditional jumps are eliminated to create a single jump and target label.

4.10.11 Instruction Combination

Instruction combination replaces several operators with a single, less expensive

operator. This optimization method is very machine specific.

Sequences that are commonly combined by the optimizer include:

- a memory reference combined with a pointer adjustment becomes an autoincrement or autodecrement addressing mode. This is very powerful when combined with register promotion and do loop promotion. For example,

```
for ( i = 0 ; i < 10 ; i ++ )
{
    array_1[ i ] = array_2[ i ];
}
```

the **for** loop becomes a **do** instruction, the array references are promoted to address registers and the induction variable is eliminated with array pointer advancement done via the autoincrement addressing mode.

4.10.12 Leaf Routine Detection

A leaf routine is a subroutine that makes no further subroutine calls. When the compiler identifies such routines, the prologue and epilogue code are optimized (no save and restore of the **ssh**).

4.10.13 Function In-lining

When explicitly requested via the command-line option **-finline-function**, the compiler will replace calls to subroutines with an in-line copy of the subroutine if the subroutine meets these requirements:

1. the subroutine must be a non-volatile leaf function
2. the subroutine must be in the same module
3. the definition must precede use of the subroutine.

Function in-lining eliminates the overhead associated with subroutine calls and provides additional opportunities for the optimizer and register allocator to further increase code quality. Function in-lining can also be performed explicitly by the programmer by utilizing the additional non-ANSI function type specifier **__inline**. By default, many run-time libraries are in-lined by the compiler.

Note: The function in-lining method can cause program memory requirements to grow significantly. See Appendix A, Programming Support, for instructions on disabling library routine in-lining.

Chapter 5

Mixing C and Assembly Language

5.1 Overview

In cases where the DSP96002 programmer requires direct access to the hardware or greater performance at the inner-loop of an algorithm, C programs can be mixed with assembly programs. This chapter describes two methods for combining C and assembly language source code. The first is the in-line method which is a convenient way to put assembly code into the compiler output via the non-ANSI C directive `__asm()`. The second is the out-of-line method, a generic method of combining C and assembly language source **files**.

Caution

Before mixing C and assembly, read and understand Chapter 4, **About g96k**, and the **DSP96002 User's Manual**. Attempting to program for this DSP without knowledge of the chip and how the compiler utilizes registers, memory, defines labels, etc. may generate unsatisfactory results. However, with an understanding of the DSP architecture and how this implementation of C uses it, programming should be straightforward.

Due to the difficulty in isolating the individual points to be illustrated, the examples for in-line assembly code are grouped together after the section describing in-line assembly code. An example for out-of-line assembly code are grouped together after the section describing out-of-line assembly code.

Note: Labels which begin with a double underline (e.g., `__asm()`) in this manual have a space between the double underlines to visually separate them. Do not separate the leading double underlines with a space when coding them (i.e., code `__asm()` as `__asm()`).

5.2 In-line Assembly Code

Each instruction of a C program must be a C instruction. In-line assembly code is assembly code that is inserted inside a C instruction in a C code source file. Since assembly

code is generated from this C statement directly, the C statement looks like assembly code in the C source and is referred to as in-line assembly code. All assembly code to be generated is visible at the C source-level and it is often very convenient and simple to intermix assembly programs with a C source program in this fashion.

Typically, in-line assembly code is used when:

1. inserting a small amount of assembly code directly into the compiler output i.e., inner loop kernels.
2. writing fast assembly language routines to be called by C subroutines. This eliminates the need to manage data referencing, register saving and allocation, and function call/return overhead.
3. C callable subroutines written in assembly language must be used as **in-line** functions (see Section 4.11.13, Function In-lining).

The key word `__asm` is introduced as an extension to the ANSI C language standard and this key word is used like a function call to specify the in-line assembly code generation rule discussed below.

The in-line assembly statement syntax is:

```
__asm ("instruction_template": output_operands: input_operands: reg_save);
```

where:

1. **instruction_template** is a template used to describe an assembly code instruction that is to be inserted into the compiler output. It may require arguments listed in the `output_operands` and `input_operands` to get more detailed information. It may contain a substring called operand expansion string (OES) which starts with a single character (%). Description of OES and `instruction_template` interpretation is described in Section 5.2.1.
2. **output_operands** are optional operands used to provide specific output information to the template. Each `output_operand` is separated by a comma and should begin with the character '='. As an example, the `output_operand` **"=a" (cptr)** means "address register of the C variable, cptr". See Section 5.2.2 for more details.
3. **input_operands** are optional operands to provide specific input information to the template. Each `input_operand` is separated by a comma and may include a C variable. As an example, the `input_operand` **"a" (cptr)** means "address register of the C variable, cptr". Full descriptions of the input and output operands can be found in Section 5.2.2.

4. **reg_save** specifies registers that are to be explicitly saved through the `__asm()` statement. The registers to be saved must be named as strings such as 'd0.l' and 'd1.l'. Each register is separated by a comma (see Section 5.2.3 for additional information).

5.2.1 Instruction Template

The first argument of the `__asm()` extension is the instruction template or **assembler instruction template** which is mandatory and describes the line or lines of assembly language source code to be placed into the compiler output (see Example 5-2 in Section 5.2.4). This template is not parsed for assembly language syntax violations and is simply written to the compiler output. As a result, the compiler will not detect assembly-time errors. These errors can be detected by assembling the in-line assembly code as a stand-alone program for this purpose.

More than one assembly instruction can be put into a single instruction template by using the line separator '\n'. The line separator, or newline, can be utilized as in a normal C statement. The line continuation character, '\', is particularly useful when an instruction template that contains a long assembly instruction line is too long to fit in one line (see Example 5-13 and Example 5-14 in Section 5.2.4). Other special C language characters such as '\t', '\f', etc. can also be used in the instruction template.

In many situations, it is desirable to use C variables or expressions directly in the instruction template. Since all memory and register accesses are accomplished through variables, manipulating memory and registers directly using assembly code requires knowledge of their locations. Without optimizations, the current value of a variable will be maintained in memory at a specific address. However, an optimizing C compiler such as DSP96KCC may retain a variable in a register and perform operations on that variable without updating the memory location corresponding to the variable between operations. This enhances the performance of the compiled code but makes accessing variables in memory risky. In order to guarantee that the correct value of a variable is returned when it is referenced, a mechanism called operand expansion string (OES) is provided.

The OES allows a variable to be securely accessed even though its current value is in a register. The operand expansion string is a substring of the instruction template and begins with the character '%'. This string is usually two or four characters long and provides the compiler with special information about a C variable or expression. It must reference only one C variable or expression which must be listed in the operand list (see Section 5.2.2). The OES is parsed by the compiler and gives sufficient information to allow the variable to be correctly referenced by the assembly language instruction in the instruction template. Most examples in Section 5.2.4 include an OES.

The OES syntax is:

% [modifier] operand_id_number

where:

1. **modifier** is a single optional character which specifies a particular operand. The available modifier is 'z'.

z — append “.l”, “.s”, or “.d” based on the type of the operand specified the operand_id_number. If the type is integer, then the suffix “.l” will be appended to the register allocated for the operand, e.g. d0.l - d7.l. If the type is float register, then the suffix “.s” will be appended, e.g. d0.s - d7.s. If the type is double register, the suffix “.d” will be appended, e.g. d0.d - d7.d.
2. **operand_id_number** specifies the operand location in the operand descriptor list (see Example 5-3 in Section 5.2.4). The operand descriptor list is a concatenation of the output operands and input operands (see Section 5.2.2). The first operand is zero and there can be up to 31 more operands in the list. More than one instruction template can be used if more than 32 operands are needed.

In-line assembly code can also be used to insert a label directly into the compiler output (see sections 5.2.5.1). A label without any white spaces in the in-line assembly code instruction template will guarantee that the same template label will be in the compiler output (see Example 5-15). Care should be taken not to use labels that the C compiler generates. Using the same labels as the C compiler will cause a duplicate label error (see Section 4.7, Compiler Naming Conventions).

5.2.2 Output/Input Operands

The operand list is a concatenation of **output** and **input operands** which the OES can access via the operand_id_number (see Section 5.2.1). Output or input operands consist of operands separated by a comma (','). Each operand should be associated with a **C expression** and its **operand constraint** described below.

A colon (:) is used to separate the assembler instruction template from the first output operand. A second colon separates the final output operand from the first input operand. A third colon can be used to separate the input operands from the optional field **reg_save**. Two consecutive colons are required when only input operands are needed in the operand list.

The operand syntax is:

“[=]operand_constraint” (C_expression)

where:

1. **=** differentiates input and output operands. All output operands must use this character.
2. **operand constraint** is a single character that describes the type of resource (memory or register) that an operand is to be loaded into or read from. Each operand constraint has an optional set of modifiers that may be applied in the instruction template.
3. **C_expression** is any valid C expression defined by the ANSI C standard. The C expression can be either l-value or r-value. Any output operand should use the l-value to specify the memory location to store the data.

The available **operand constraints** are “a”, “d”, “f” and “m”. All of these constraints originate from the DSP96002 architecture; therefore, a full understanding of these constraints requires that the programmer understand this architecture. The constraints are:

- a** — One of the **Address Registers** (rn, where n = 0 through 7; see Section 5 of the DSP96002 User's Manual) will be allocated to the C expression (see Example 5-4). The C expression will be promoted to this register. Typically the C expression should be a pointer to be assigned to an address register.
- d** — One of the 32 bit integer registers (d0.l - d7.l, d0.m - d7.m, and d0.h - d7.h; see Example 5-10).
- f** — One of the float or double registers (d0.s - d7.s, d0.d - d7.d; see Example 5-11).
- m** — A memory location will be allocated to the C expression (see Example 5-12). The DSP96002 has four memory spaces: X, Y, L and P but the C compiler may only use the X, Y or L memory spaces for this expression.
- i** — An immediate value of the operand (see Example 5-13).
- number** — Inherit all memory or register characteristics of the operand indicated by the **operand id number** (see Example 5-3). This constraint is usually used for read/write operands which are declared by using the same C variable as both the input and output operand.

The operand is sometimes referred to as a read-only operand if it is used only as input (see Example 5-5) and it is called a write-only operand if it is used only as an output (see Example 5-6). In most of cases, the operand is used as both an input and an output operand (see Example 5-7 and Example 5-8). In these cases the operand should be described as both. Since output operands should be listed first, the operand id number is determined when the input operand is declared. The id number will be used as the operand constraint of the input operand.

Table 5-1 lists the operand constraints and their related modifier.

Table 5-1 Operand Constraints/Modifiers Associations

Operand Constraint		Modifier
a	- Address Register (rn)	none
d	- Integer Register (d0-d7)	%z - append ".l" suffix
f	- Floating-point Register (d0 - d7)	%z - append ".l" suffix for integer ".s" for float ".d" for double
m	- Memory Location	none
i	- Immediate Value	none
number	- OES id number (either a, d, f, or m)%z or none	

5.2.3 Explicit Register Saving

It is possible to manually perform register allocation, in fact this may simplify the process of converting an existing body of DSP96002 assembly language subroutines to in-line assembly code. The programmer need only identify each register explicitly referenced in the assembly code and list them in the **reg_save** argument region (see Section 5.2). This guarantees that the content of each register is preserved across `__asm()` calls. Use of registers **r0** and **r6** is prohibited in the assembly code because they are reserved for the C compiler during run-time where **r0** is the frame pointer and **r6** is the stack pointer. Registers Mn are used by the compiler as temporary registers and Nn are not used by the C compiler at all. As a result, these registers do not need to be saved unless the programmer uses them in assembly code. If they are used in assembly code, they should only be used as local variables.

Explicit register saving is done through specifying the registers to be saved. A string is used to specify each register. The valid register names are listed in Table 5-2.

Table 5-2 reg_save Names

Register Type	Valid Names
Integer Registers	"d0.l"- "d7.l"
Float Register	"d0.s"- "d7.s"
address	"r0" - "r7"

A final example for in-line assembly code constructs a program performing an FIR filter operation which is based on the Dr. BuB FIR filter program "fir.asm". The program data

structure is the same as the Dr. BuB program and data is passed through the C variable “data”. Pointers “sptr” and “cptr” are used to point to static variables.

Example 5-1 reg_save: This in-line assembly code is converted from the Dr. BuB FIR filter program using reg_save. This example uses 4 tap FIR filter case.

```
extern float sbuf[];
__asm("\nFdbuf      dsm      4");

float fir(float data, float *coef)
{
    float result;
    static float *sval = &sbuf[0];
    __asm volatile
        ("move    #3,m0\n\
         fclr     d1      m0,m4\n\
         move     %z4,x:(%2)\n\
         move     %2,r0\n\
         move     %3,r4\n\
         fclr     d0      x:(r0)-,d4.s y:(r4)+,d6.s\n\
         rep      #4\n\
         fmpy     d4,d6,d1 fadd.s d1,d0 x:(r0)-,d4.s y:(r4)+,d6.s\n\
         fadd.s   d1,d0    (r0)+\n\
         move     d0.s,%z0\n\
         move     (r0)+\n\
         move     r0,%1\n\
         move     #-1,m0\n\
         move     #-1,m4"
         : "=f"(result), "=a" (sval)
         : "1"(sval), "a"(coef), "f"(data)
         : "r0", "r4", "d0", "d1", "d4", "d6");
    return result;
}
```

5.2.4 In-line Assembly Code Examples

The examples in this section illustrate the practical application of the __asm() extension. The main purpose of this section is to show how to write in-line assembly code. Since these examples are intended to illustrate the information presented earlier in this chapter, references to the appropriate subjects have been included.

Example 5-2 illustrates the use of the in-line assembly code “instruction_template”. Since this in-line assembly code directly clears register d0.l, the programmer should check to be sure that the contents of d0.l are not needed.

Example 5-2 Instruction_template: One line of in-line-assembly code clears register d0.l.

```
__asm(“clr d0.l”);
```

A pseudo operand will be used to illustrate use of the OES operand id number. The pseudo operand functions as an input or output operand. Example 5-3 uses five pseudo operands: V, W, X, Y and Z each of which is referenced by operand ids 0, 1, 2, 3 and 4, respectively. The pseudo operands are used as in the OES “%0”, “%1”, “%2”, “%3” and “%4”. Table 5-3 shows which operands in Example 5-3 are input or output operands.

Example 5-3 Instruction template with operand_id: In order to illustrate how to use output or input operands, pseudo operands V, W, X, Y, and Z are used. The operand_id listed in this example can be used as part of an instruction_template.

```
__asm(“instruction_template” : V, W, X : Y, Z );
```

Table 5-3 Output and Input Operands for Example 5-3

operand id	pseudo operand	operand type	OES
0	V	output	%0
1	W	output	%1
2	X	output	%2
3	Y	input	%3
4	Z	input	%4

Example 5-4 Input Expression / Output Expression: This in-line assembly code uses the pseudo assembly mnemonic “asm_instruction” and refers to two C expressions: output_expression and input_expression. This example illustrates how to interpret the **operand constraint** (see Section 5.2.2) and operand id (see Section 5.2.1 and Example 5-3). The example implies that the C expression output_expression is expanded with constraint f and is an output of the assembly code instruction asm_instruction. Similarly, the C expression input_expression is expanded with constraint d and used as an input to the assembly code instruction asm_instruction.

```
__asm(“asm_instruction %1,%0” : “=f” (output_expression) : “d” (input_expression));
```

Example 5-5 Read-Only Operand: This in-line assembly code uses the pseudo assembly mnemonic “asm_instruction” and uses

input_expression as a read-only operand.

```
__asm("asm_instruction %0" :: "f" (input_expression));
```

Example 5-6 Write-Only Operand: This in-line assembly code uses the pseudo assembly mnemonic "asm_instruction" and uses output_expression as a write-only operand.

```
__asm("asm_instruction %0" : "=d"(output_expression));
```

Example 5-7 Read-Write Operand: An addition is programmed using in-line assembly code and the C expression result is used as a read-write operand. The variable, foo, is used as a read only operand. Notice that operand **number** (i.e. '0') was used instead of the operand constraint 'd' because it should be used as both input and output. The second operand of the instruction ADD is source operand as well as destination operand (see the DSP96002 User's Manual — Appendix A for the syntax of the ADD instruction).

```
int foo, result;  
__asm("add %1,%0" : "=d" (result) : "d" (foo), "0" (result));
```

Example 5-8 Read-Write Operand: The same result will be obtained as in Example 5-7. Notice how the operand id is changed according to the placement of the C variables.

```
int foo, result;  
__asm("add %2,%0" : "=d" (result) : "0" (result), "d" (foo));
```

Example 5-9 through 5-12 illustrates the use of OES Operand Constraints (see Section 5.2.2).

Example 5-9 Operand Constraint a: The following in-line assembly code is used to increment an integer pointer. The assembly code generated assumes that the C compiler allocates the address register r4 for the C pointer cptr.

In-line assembly Code:

```
int *cptr;  
__asm("move (%0)+" : "=a" (cptr));
```

Assembly Code Generated:

```
move (r4)+
```

Example 5-10 Operand Constraint d: The following in-line assembly code is used to move the C integer value to the internal I/O memory

space of x:\$FFFFFFFF (refer to Section 7 of DSP96002 User's Manual). The assembly code generated assumes that the register d3.l is allocated for the integer. This in-line assembly code is especially useful when it is desired to access the internal I/O memory or to use the DMA controller from C programming environment.

In-line assembly Code:

```
int i;  
__asm("movep %z0,x:$FFFFFFFF" : "=d"(i));
```

Assembly Code Generated:

```
movep d3.l,x:$FFFFFFFF
```

Example 5-11 Operand Constraint f: The following in-line assembly code is used to move the C float value to X memory location \$EFFF. The assembly code generated assumes that the register allocated for the float variable foo is d4.

In-line Assembly Code:

```
float foo;  
__asm("move %z0,x:$EFFF" : "=f" (foo));
```

Assembly Code Generated:

```
move d4.s,x:$EFFF
```

Example 5-12 Operand Constraint m: The operand constraint **m** can be used to generate the following assembly code. The assembly code generated assumes that the memory location of the variable foo is \$700.

In-line Assembly Code:

```
int foo;  
__asm("move #$FFFF,y:%0" : "=m" (foo));
```

Assembly Code Generated:

```
move #$FFFF,y:$700
```

In the following examples, OES are not used because C variables are not referenced.

Example 5-13 Multiple Instruction — Single-Line: An in-line assembly program which places a value (e.g. \$709) in register d0.l and negates the result is written in one line. This one line will generate

two lines of assembly code in the C compiler output.

```
__asm("move #$709,d0.\n neg d0.l");
```

Example 5-14 Multiple Instruction — Multiple-Line: The two lines of in-line assembly code in this example have the same effect as the one line in Example 5-13. Notice that using two lines increases the in-line assembly code readability. The line continuation character ‘\’ used at the end of the in-line assembly codes in the first line makes this possible.

```
__asm("move #$709,d0.l\n\n\n neg d0.l");
```

Example 5-15 Instruction Template Label: The following in-line assembly code which generates the label “foo” uses a return character “\n” to insure that there is no white space in front of the label.

```
__asm("\nfoo");
```

Example 5-16 Operand Constraint i : The following in-line assembly code can be used to select the port for X, Y, and P memory space. This example selects port A for both X and P memory spaces and port B for Y memory space (see the DSP96002 User’s Manual).

```
#define PSEL 0x00ff00
__asm("movep %0,x:$ffffffc" :: "i" (PSEL));
```

Example 5-17 Host Interface : The following in-line assembly code can be used to utilize the host interface of the DSP96002.

```
__asm("HTRX equ $ffffffe7");
__asm("movep d0.l,x:HTRX"); /* write data to Host Interface */
__asm("movep x:HTRX,d0.l"); /* read data from Host Interface */
```

5.2.5 Controlling Labels Generated by the Compiler

Using the `__asm()` keyword, it is possible for the programmer to override the compilers label generation conventions for subroutines and global variables. This may be useful for:

1. calling assembly language subroutines,
2. calling c subroutines from assembly code,
3. referencing assembly language global variables from C,
4. referencing global C variables from assembly code.

5.2.5.1 Calling Assembly Subroutines

Calling a subroutine or function requires using a label that points to the subroutine or function. Most assembly language routines do not have labels which comply with the C compiler label convention described in Section 4.7. The `__asm` directive can be used to change a “C” label to a label assigned to an assembly routine so that routines initially written in assembly language can be accessed from the C program.

To illustrate how to use the `__asm` directive, Example 5-18 reads the data at X memory location \$100 and Y memory location “Y+2”. For test purposes, the Y memory space is filled with the integer sequence 0 through 9 and the X memory space is left uninitialized. The `printf()` statement prints the data returned from the functions `ValOfX(100)` and `ValOfY(Y+2)`. These two functions are constructed in assembly code and reside in another file. The two assembly subroutines, `ReadX` and `ReadY`, are written using the out-of-line assembly technique described in Section 5.3. The listings for `ReadX` and `ReadY` are shown in Example 5-32 in Section 5.3.5.

By using the statement

```
extern int ValOfX() __asm(“ReadX”), ValOfY() __asm(“ReadY”);
```

all C compiler generated function labels for `ValOfX()` and `ValOfY()` are replaced by the labels `ReadX` and `ReadY`, respectively.

Example 5-18 Calling assembly from C. This C program (called `prg516.c`) can be used to examine the data in X or Y memory by calling assembly routines “`ReadX`” or “`ReadY`”. Notice that the assembly code for `ReadX` and `ReadY` is listed in Example 5-32 of Section 5.3.5.

```
% cat prg516.c
/*
 * prg516.c
 */
#include <stdio.h>
extern int ValOfX() __asm(“ReadX”), ValOfY() __asm(“ReadY”);
unsigned Y[] = {0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9};
main()
{
    printf("<%x><%x>\n", ValOfX(100), ValOfY(Y+2));
}
```

The following two command lines test Example 5-18.

```
% g96k -mx-memory prg516.c memread.asm
% run96sim a.cld
```

5.2.5.2 Calling C Subroutines from Assembly Code

Any C program can be called from an assembly program to test the assembly program data or utilize built-in standard C libraries such as memory allocation routines. Calling a C subroutine from assembly code requires using the C subroutine calling convention (see Section 4.8 and Section 5.3.4) and matching the C function labels. The in-line-assembly directive, `__asm()`, can be used as shown in Example 5-19 to change the C program labels.

Example 5-19 Calling C from assembly. This C subroutine (called `c_print.c`) uses the standard C library routine, `printf()`, to print the input argument as a string.

```
% cat c_print.c
/*
 * c_print.c
 */
#include <stdio.h>
int C_printf() __asm("print");
C_printf(char *msg)
{
    printf("%s\n", msg);
}
```

Example 5-20 Calling C from assembly. This assembly program (called `greeting.asm`) prints the message "greeting: hello, there" on the screen. It uses the C subroutine, `print()`, to print this message. Notice that the assembly program name is `Fmain` because the control program, `g96k`, uses the default start-up file `crt0.cld` and `crt0.cld` uses `Fmain` to start up the main program (see Chapter 6 in this manual).

```

;;!cc
; greeting.asm
;
        section greeting
        org      l:
LC0      dc      "greeting: hello,there.", $00
        org      p:
Fmain    global  Fmain
        move     ssh,y:(r6)+
        move     #LC0,d0.l ; pass the argument LC0 to the C function.
        jsr      print
        move     (r6)-
        move     y:(r6),ssh
        rts
        endsec

```

The following two command lines are used to test two programs: `c_print.c` and `greeting.asm`.

```

% g96k greeting.asm c_print.c
% run96sim a.cld
greeting: hello, there.

```

5.2.5.3 Referencing Assembly Global Variables from C

The data in assembly language programs must be accessible to C programs to take full advantage of the DSP96002 architecture since the C language cannot access all of the DSP96002 features directly. One way to access this data is through global data which can be defined in assembly language and accessed in the C program environment. This feature is particularly useful to allocate modulo buffers for C variables. Detailed information on modulo buffers can be found in the DSP96002 User's Manual, Section 5 — Address Generation Unit and Address Modes.

Example 5-21 Generate data with assembly language. The data file, `sqtbl.asm`, is generated in assembly language and consists of the squares of the numbers 0 - 8. Notice that directives `BSC`, `DC`, `DS`, `DSM` and `DSR` (see Section 6.3 of the Motorola DSP96000 Macro Assembler Reference Manual) can be used depending on the application.


```

% cat sqtbl.asm
;;!cc
; sqtbl.asm
;;
        section data
        global table
        org y:
table    dc 0,1,4,9,16,25,36,49,64 ; squares of 0 - 8.
        endsec

```

Example 5-22 Access data with C. This test program (called prg520.c) prints the value of 5^2 on the screen.

```

% cat prg520.c
/*
 * prg520.c
 */
#include <stdio.h>
extern int SQUARE[] __asm("table");
main()
{
    printf("square of %d is %d\n", 5, SQUARE[5]);
}

```

The following two command lines for Example 5-22 test the two programs sqtbl.asm and prg520.c.

```

% g96k prg520.c sqtbl.asm
% run96sim a.cld
square of 5 is 25

```

5.2.5.4 Referencing Global C Variables from Assembly Language

Global data in a C program is accessible to assembly language programs using DSP96KCC. This feature is particularly useful when the data to be processed by an assembly language program is generated by the C program. Example 5-23 provides coefficients that are used in Example 5-24.

Example 5-23 Generate data with C. Data file, data.c, is generated by a C language program and contains the coefficients of an average filter which takes the average of the last four input data.

```

% cat data.c
/*
 * data.c
 */
int Cwaddr[] __asm("waddr");
int Ccaddr[] __asm("caddr");
int NTAP __asm("N_1");

int Cwaddr[4];
int Ccaddr[] = { 0.25, 0.25, 0.25, 0.25};
int NTAP = 4;

```

Example 5-24 4-tap FIR filter — Access data with assembly language: This FIR filter reads an input sample from memory location y:input and writes the filtered output to memory location y:output. The input data array is stored in X memory starting at waddr and the FIR coefficients are stored in Y memory starting at caddr. Notice that the memory space for waddr and caddr is allocated in the C routine described in Example 5-23.

```

% cat fir.asm
; fir.asm
;;
input equ $ffffff ; may need to change
output equ $ffffffe ; may need to change
org p:
move #waddr,r0 ; use the C buffer waddr in Example 5-23
move #caddr,r4 ; use the C buffer caddr in Example 5-23
move y:N_1,m0
fclr d1 m0,m4

movep y:input,x:(r0)

fclr d0 x:(r0)-,d4.s y:(r4)+,d6.s
rep m0
fmpy d4,d6,d1 fadd.s d1,d0 x:(r0)-,d4.s y:(r4)+,d6.s
fadd.s d1,d0 (r0)+
movep d0.s,y:output

end

```

```

% g96k -S data.c
% asm96000 -c fir.asm
% asm96000 -c data.asm
% dsplnk fir.cln data.cln
% cldlod fir.cld > fir.lod
% sim96000

```

5.2.6 Specifying Registers for Variables

DSP96KCC allows the programmer to identify a specific register for local and global variables, but due to the limited number of registers available, this may not have a positive effect on run-time performance. With this in mind, this feature should be used sparingly.

Both global and local variables are candidates for promotion to specific registers and syntactically they look the same:

```
register int *ptr __asm("r5");
```

By specifying a specific register for a local or global variable, the programmer is reserving the register for the variable's entire scope (global for the entire program, local for the function in which they are declared). This means that the compiler will not use the register for any other purpose and the register **will not be saved and restored** by the C function call.

5.2.7 Optimizer Effects on Code

All in-line assembly code is visible to the optimizer and as such it is possible that the optimizer will convert it into a new form or eliminate it entirely if it is determined to be unreachable or dead. In order to guarantee that code is **not removed** by the optimizer, the ANSI keyword **volatile** must be used.

```
__asm volatile ( ... );
```

5.3 #pragma Directive

The purpose of this section is to explain the proper techniques for manipulating the assembler's run-time and load-time counters while programming in the C language.

Currently the Motorola DSP assemblers allow the programmer to specify both a run-time location and a load-time location for any object; however, there is no corresponding concept within C. The generic **#pragma** facility is used to add this capability rather than extending the C language. Users now have complete freedom in specifying both the run-time and load-time counters for any static or global object. These directives may be used with either code or data.

This flexibility is achieved by allowing the user to modify any of eight counter strings maintained by the compiler — two for each memory space: x, y, l and p. When an object is defined, the current values of those counter strings are bound to that object.

The syntax for the pragma directive is

```
#pragma counter_string argument
      C function or data storage definition
#pragma counter_string
```

where

1. the two #pragma statements must encase the entire definition.
2. counter_string in the first #pragma specifies which phase (run or load time) and memory space is to be affected. It can be **x_run**, **y_run**, **l_run**, **p_run**, **x_load**, **y_load**, **l_load**, or **p_load**. The memory model used in the C compiler should match the memory model specified by counter_string. Note that the p_run and p_load counter_strings will always have an effect.
3. the argument in the first #pragma is the string that will be passed as either the runtime or load-time argument to the org assembler directive. This address, which is optional, is of the form x:address_value where x is the counter associated with the **x**, **y**, **l**, or **p** memory, and address_value is the value to be initially assigned to that counter. As an example, p:\$300 might be used for the counter string **p_load**.
4. the C function or data storage definition is a declaration that reserves storage.
5. The second counter_string should be the same as the first counter_string and will return the memory specification to the default setting.

If and only if the memory space of the counter string in the #pragma directive matches the memory model of the C compiler, then the compiler will insert an assembly org statement of the form:

```
(1)  org      a:runtime_address,b:loadtime_address
      or
(2)  org      a:runtime_address
```

where “a” is the run time counter and runtime_address is the optional initial value for that counter, as specified in the “argument” to #pragma.

“b” is the load time counter and loadtime_address is the optional initial value for that counter, as specified in the “argument” to #pragma.

Example two illustrates that the load time counter is optional. See the section on

the ORG statement in the Motorola DSP96000 Assembler Manual for a complete description and list of options.

Notice that the pragma directive run-time counter string will only affect the **run**-time address and the pragma directive load-time counter string will only affect the **load**-time address.

As a simple example, **assuming that I-memory model (or default memory model)** is used (see Chapter 3 to change the memory model to be used), the following C segment

```
#pragma I_load x:$100
int coeff [5] = {0.0, 2.3, 1.0, 3.3, 4.0};
#pragma I_load
```

produces the following assembly language code:

```
                org l;,x:$100
Fcoeff
                global  Fcoeff
                dc      0.0
                dc      2.3
                dc      1.0
                dc      3.3
                dc      4.0
```

Notice that the second #pragma directive will remove the effect of the first memory specification, i.e., #pragma I_load x:\$100, and the rest of the code generated by the C compiler will be in the default memory area (see Chapter 3 in order to change this default memory model).

The above example code will be loaded in the X memory location \$100, and it should be copied to the L memory space upon system start-up. When burning a PROM, only one memory space is desired to be used, as an example, P memory space, so that only one PROM is enough for both data and program. In such case, both the data and the program will be burned in the PROM and the data should be moved to the data memory space (either, X, Y or L memory space depending on the program) upon system start-up.

Let's assume that the coefficients of the above example need to be in the program space when burning the PROM. Then the following C segment

```
#pragma I_load p:$100
int coeff[5] = {0.0, 2.3, 1.0, 3.3, 4.0};
#pragma I_load
```

produces the following assembly language code:

```

                                org l:,p:$100
Fcoeff
                                global  Fcoeff
                                dc      0.0
                                dc      2.3
                                dc      1.0
                                dc      3.3
                                dc      4.0
```

The above assembly code will be loaded into the P memory space at p:\$100 for the PROM burning, and it should be copied to the L memory space before the actual program is executed. Manipulating the assembler's run-time and load-time counters requires a thorough understanding of the underlying assumptions about memory layout, which are made by the compiler (see Chapter 6). Incorrect use of this feature may cause compile-time, link-time and even run-time errors.

5.4 Out-of-line Assembly Code

Out-of-line assembly code is assembly code written in a separate source file that is called from a C program. Separating the assembly code and C code in this way provides a powerful and flexible interface to the DSP96002 architecture. This out-of-line method may be used to convert existing assembly subroutines to C callable routines, write new subroutines completely in assembly language or access both data spaces (X and Y). The key advantage of out-of-line assembly code is that it provides a complete assembly programming environment for the DSP96002 whereas the in-line assembly code must follow the C programming environment rules.

Writing out-of-line assembly code requires a complete understanding of the C Cross Compiler and the DSP96002 architecture. For out-of-line assembly code to be callable from a C program, the following five basic elements should be included in the assembly source file in sequence.

1. C subroutine entry code (prologue code)
2. Save all registers to be used
3. Main program
4. Restore all registers used
5. C subroutine exit code (epilogue code)

Each of these steps can be programmed as a macro; however, using generic macros can unnecessarily increase the size of the resulting assembly code program. The DSP's ability to move data in parallel with arithmetic operations provides opportunities to significantly optimize the assembly code by combining steps within a macro and by combining the code of two or more macros.

In order to illustrate the steps listed above, the out-of-line assembly code template is described first and each element of the template is then explained in detail. After reviewing the five elements, some optimization techniques are discussed.

5.4.1 General Template

The following template is a generic form used to make the C function "foo". There are five distinct elements in the template which are numbered as above. The actual code for the prologue and epilogue is shown but the "Save all registers to be used", "Main Program", and "Restore all registers used" are listed as comments because their actual code depends on the main program.

```

global      Ffoo      ; prologue:
Ffoo        ; set up entry point (C function address),
            move      r0,y:(r6)+ ; save old frame pointer (r0),
            lea       (r6)+,r0   ; update the frame pointer (r0),
            move      ssh,y:(r6)+ ; save the return address.
;; Save all registers to be used
;; Main Program
;; Restore all registers used

            ; epilogue:
            move      (r6)-      ; have stack pointer point to the return address
            move      y:(r6)-,ssh ; restore it, then updates stack pointer
            move      y:(r6),r0  ; restore the frame pointer (r0),
            tst       d0         ; update SR by testing d0.l.
            rts

```

5.4.1.1 Prologue

The first two lines of the prologue make the assembly program visible to the C program so that the subroutine or function is callable from the C program. In this case, any one of the following C statements can be used to access out-of-line assembly code.

```

foo();

x = foo();

```

```
x = foo(arg1, arg2, arg3);
```

The first function call assumes that the C function does not use any arguments and does not return any values. The second only returns a value which is the same data type as the variable x. The last call assumes that the C function uses the three arguments: arg1, arg2 and arg3 and then returns the value x.

The rest of the prologue saves the old frame pointer, updates the current frame pointer and saves the return address. The return address is saved when the JSR instruction pushes the hardware stack which saves the program counter in the high 16 bits of the system stack i.e., SSH. This section of the prologue provides bookkeeping for the C compiler activation record. Detailed information on the structure of the activation record can be found in Section 4.6.1. After this prologue, the r0 register is the frame pointer and r6 is the stack pointer.

5.4.1.2 Save all registers

All registers used in the main program should be saved before the main program changes them. This step is the second element of the template — “Save all registers to be used”.

In order to save the registers, r6 is used as a stack pointer. The stack grows upward and the current stack pointer (r6) points to the next element above the top of stack. The following statement saves one register to the top of stack and sets the stack pointer to the next available stack location.

```
move    d2.l,y:(r6)+
```

Two registers, d3.s and d4.s, can be saved with the following statements:

```
move    d3.s,y:(r6)+  
move    d4.s,y:(r6)+
```

Since saving and restoring the registers are the subroutine's responsibility, the order of saving the registers should be in accordance with their restoration. The restore process should be exactly the reverse order of the register saving sequence.

5.4.1.3 Main Program

The main C function program accesses the parameters passed, executes the operations on the parameters and returns a value. Passing parameters is done through the activation record which has been pushed onto the stack. The activation record frame pointer is pointed to by r0 and the first element of the parameter list is placed 3 locations below the frame pointer (see Section 4.6.1). The first two parameters may be located in the registers d0.l and d1.l, or in the parameter list depending on the size of the two parameters. If

the size is one word, then the two parameters will be passed through the registers d0.l and d1.l. If the size is greater than one word, then the parameters will be located in the parameter list. When the register d0.l and d1.l are used as the first and the second parameters, the rest of the parameter will be in the parameter list of the activation record.

For example, assuming the first four parameters are one word long, then the following statement should be used to move the first single word parameter to register r3.

```

move    d0.l,r3                ; the first parameter is in the
                                ; register d0, store it to r3.

```

The following statements move the second, third and fourth parameters to registers r4, r5, and d2.l respectively.

```

move    d1.l,r4                ; the second parameter
move    y:(r0+-3),r5           ; the third parameter
move    y:(r0+-4),d2.l         ; the fourth parameter

```

5.4.1.4 Restore all registers

The stack pointer, r6, is needed to restore the registers. Since the current stack pointer points to the next element above the top of stack, the stack pointer should be decremented before accessing the stack. The following code will restore one register.

```

move    (r6)-                  ; decrement
move    y:(r6),d2.l            ; restore

```

The restoring procedure can be simplified if more than one register is to be restored. Restoring registers d2.l, d3.s, d4.s and d5.l can be done by the statements below which are essentially four repetitions of the three statements above.

```

move    (r6)-
move    y:(r6),d2.l
move    (r6)-
move    y:(r6),d3.s
move    (r6)-
move    y:(r6),d4.s
move    (r6)-
move    y:(r6),d5.l

```

However, combining three statements

```

move    y:(r6),d2.l
move    (r6)-

```

into one statement

```

move    y:(r6)-,d2.l

```

saves three instructions. As a result, the eight statements in the restoring procedure listed above can be reduced to the following five statements:

```

move    (r6)-
move    y:(r6)-,d2.l
move    y:(r6)-,d3.s
move    y:(r6)-,d4.s
move    y:(r6),d5.l

```

After the main program, a return value can be passed to the caller. Any 32-bit floating value must be returned through register d0.s, 32-bit or 16-bit values must be returned through register d0.l. If the return value is larger than 32 bits, then the compiler places the proper amount of buffer space in the activation record and register r7 becomes the pointer to this buffer space. It is the main program's responsibility to copy any return values from the buffer whose address resides in register r7.

5.4.1.5 Epilogue

The out-of-line template epilogue is the reverse of the prologue. The epilogue restores the return address and updates the frame pointer, r0. Notice that the stack pointer, r6, should be decremented before each move operation. In addition, register d0 is tested to update the SR register. This testing is a part of the C compiler code generation feature and should be included (see Section 5.3.5 for optimization).

5.4.1.6 Out-of-line Assembly Code Example

The following example illustrates using the general template as well as the fact that the DSP96002 performs fractional arithmetic. Assume that section mod1 contains two C callable subroutines, mac01() and mac02() and section mod2 contains a single C callable subroutine, mactwo(). The two functions mac01() and mac02() take one argument each and mactwo() takes two arguments. These functions perform multiplication according to the following formulas and return their results:

mac01(arg)	calculates	$\text{arg} * 0.01$
mac02(arg)	calculates	$\text{arg} * 0.02$
mactwo(arg1, arg2)	calculates	$\text{arg1} * \text{arg2}$

The following function declarations (ANSI standard) can be used to declare functions which are known to perform floating point arithmetic.

```

float mac01(float);
float mac02(float);
float mactwo(float, float);

```

These three functions can be called as follows:

```
float result;  
result = mac01(2.345); /* 0.01 * 2.345 */  
result = mactwo(0.01, 2.345); /* 0.01 * 2.345 */
```

Example 5-25 General Template for Out-of-line Assembly Code: The two sections of this out-of-line assembly code are mod1 and mod2. The first section implements:

1. mac01(arg) which takes a floating-point argument and returns 0.01 times the argument and
2. mac02(arg) which takes a floating-point argument and returns 0.02 times the argument.

The second section implements mactwo(arg) which takes two floating-point arguments and returns $\text{arg1} * \text{arg2}$.

; section mod1:

```
; float mac01(float arg);  
; takes a floating-point argument and returns 0.01 * argument.  
; float mac02(float arg);  
; takes a floating-point argument and returns 0.02 * argument.
```

```

        section      mod1
        global      Fmac01          ; prologue
Fmac01
        move        r0,y:(r6)+
        lea         (r6)+,r0
        move        ssh,y:(r6)+

                                ; save all the registers to be used:
        move        d1.s,y:(r6)+    ; save register d1.s
        move        d2.s,y:(r6)+    ; save register d2.s

        move        d0.s,d2.s       ; the first argument
        move        #0.01,d1.s      ; operand 0.01
        fmpy.s      d1,d2,d0         ; main program: calculates multiplication of
                                ; d1.s and d2.s

                                ; restore all the registers used:
        move        (r6)-           ; adjust stack pointer
        move        y:(r6)-,d2.s    ; restore register d2.s
        move        y:(r6),d1.s     ; restore register d1.s

        move        (r6)-           ; epilogue
        move        y:(r6)-,ssh
        move        y:(r6),r0
        tst         d0
        rts

        global      Fmac02          ; prologue
Fmac02
        move        r0,y:(r6)+
        lea         (r6)+,r0
        move        ssh,y:(r6)+

                                ; save all the registers to be used:
        move        d1.s,y:(r6)+    ; save register d1.s
        move        d2.s,y:(r6)+    ; save register d2.s

```

```

move    d0.s,d2.s      ; the first argument
move    #0.02,d1.s     ; operand 0.02
fmpy.s  d1,d2,d0        ; main program: calculates multiplication of
                        ; d1.s and d2.s

```

```

                        ; restore all the registers used:
move    (r6)-          ; adjust the stack pointer
move    y:(r6)-,d2.s    ; restore register d2.s
move    y:(r6),d1.s     ; restore register d1.s

```

```

move    (r6)-          ; epilogue
move    y:(r6)-,ssh
move    y:(r6),r0
tst     d0
rts
endsec

```

; section mod2

```

; float mactwo( float arg1, float arg2);
; takes two floating-point arguments and returns arg1 * arg2.sectionmod2

```

```

global      Fmactwo

```

Fmactwo

```

move    r0,y:(r6)+      ; prologue
lea     (r6)+,r0
move    ssh,y:(r6)+

```

```

                        ; save all the registers to be used:
move    d1.s,y:(r6)+    ; save register d1.s
move    d2.s,y:(r6)+    ; save register d2.s

```

```

move    d0.s,d2.s      ; the first argument
                        ; the second argument is in d1.s
fmpy.s  d1,d2,d0        ; main program:
                        ; calculates multiplication of d1.s and d2.s

```

```

                                ; restore all the registers used:
move        (r6)-              ; adjust the stack pointer
move        y:(r6)-,d2.s       ; restore register d2.s
move        y:(r6),d1.s        ; restore register d1.s

move        (r6)-              ; epilogue
move        y:(r6)-,ssh
move        y:(r6),r0
tst         d0
rts
endsec

```

The DSP96KCC control program, g96k, should be used to assemble the out-of-line assembly code. The two sections of this code, mod1 and mod2, can be put in the same file or in separate files but in each case the following line should be in the **first line of each file**:

```
;;!cc
```

This line activates special options in the assembler to facilitate generating optimized assembly code by the C compiler.

The following command lines and source files provide a test for the case where each program is in a separate file.

```

% cat main.c
#include <stdio.h>
float mac01(float), mac02(float), mactwo(float,float);
main(){
    printf("%f, %f\n", mac01(2.345), mactwo(0.01, 2.345));
    printf("%f, %f\n", mac02(2.345), mactwo(0.02, 2.345));
}
% cat mod1.asm
;;!cc
;;a copy of section mod1 will be displayed
% cat mod2.asm
;;!cc
;;a copy of section mod2 will be displayed

```

```
% g96k main.c mod1.asm mod2.asm
% run96sim a.cld
0.02345, 0.02345
0.04690, 0.04690
```

5.4.2 Global C and Static Variables in C

The global C variables are accessed using labels generated by the C compiler. Any variables that are static to an assembly language subroutine will be accessed the same way. These variables are placed into memory at compile-time and are referenced symbolically according to the labels automatically generated by the compiler. However, it is possible to override the default labels by using the `__asm()` keyword as explained in Section 5.2.5 Controlling Labels Generated by Compiler.

For example, using the default labeling convention, the global integer, **Ginteger** which can be declared within the C statement “extern int Ginteger;” is loaded into the integer register d0.l in assembly code as follows:

```
move        y:FGinteger,d0.l
```

When declaring C global variables in an assembly language file, the programmer must be careful to follow the label generating convention or use the `__asm()` keyword to report to the compiler that the labeling convention has been changed. In both cases, the assembler directive **global** is used to export the labels to the C files. **DO NOT** use the XDEF/XREF pair of directives. **NOTE** that it is the programmers responsibility to allocate space for the global variables declared in this manner. In the example below, this is done with the assembler directive **dc**. Also, ANSI C requires that all global variables be initialized to zero if they are not explicitly initialized.

Example 5-26 Global Label in Assembly Language. This example shows assembly code that defines a global integer (named FGinteger) which is normally accessed as Ginteger in the C environment and FGinteger in the assembly programming environment.

```
FGinteger  dc      $0
           global  FGinteger
```

Example 5-27 Global Variable Declaration. This is the C code equivalent to Example 5-26 which defines the global integer Ginteger.

```
extern     int      Ginteger;
```

Example 5-28 Changing a Global Label. This example shows C code that generates a global integer (Ginteger) which is accessed as

Ginteger in both the C environment and the assembly programming environment.

```
extern int Ginteger __asm("Ginteger");
```

Which will appear in assembly language code as:

```
Ginteger    dc      $0
            global  Ginteger
```

5.4.3 Using Run-time Stack for Local Data

The run-time stack may be used when the programmer requires a temporary data space for automatic style variables — i.e., local variables in subroutines. Using the run-time stack requires additional steps in the prologue and epilogue sections. It is the subroutine's responsibility to automatically allocate and deallocate the stack at run-time.

In the prologue, an extra step is required to save the run-time stack space. Keeping in mind that the stack pointer must always point to the next available stack location, the stack space is allocated by advancing the stack pointer by the amount of space required. One way to allocate this space is shown in the Example 5-29.

Example 5-29 Run-time stack allocation: This code segment can be inserted in the general template prologue for out-of-line assembly code. Notice that “size” in the move statement below should be replaced with the appropriate constant.

```
move    size,n6      ; the stack size needed
nop                                           ; wait until n6 has been updated
move    (r6)+n6      ; allocate the run-time stack for locals
```

Referencing the data space can then be accomplished using negative offsets from the stack pointer or via initialized address registers. There are many alternatives to these methods but they are all similar.

In the epilogue, an extra step is required to restore the stack pointer — i.e., deallocate the run-time local stack. This is simply the reverse of the allocation process in the prologue.

Example 5-30 Run-time stack deallocation: This code segment can be inserted in the general template epilogue for out-of-line assembly code. Notice that “size” in the move statement below should be replaced with the appropriate constant.

```
move    size,n6      ; the stack size used before
nop                                           ; wait until n6 has been updated
move    (r6)-n6      ; deallocate the run-time stack
```


There are many ways to do this, one simple optimization would be to advance the **n6** load instruction in the program to eliminate the nop.

5.4.4 Calling C Routines

C routines are routines that are callable by a C program and may be written in either C or assembly language. When writing assembly language subroutines, it may be necessary to call library routines that have been provided or that have been written by the programmer — e.g., a call to **sin()** or **printf()**. In order to do this, the programmer must follow 3 steps:

1. Push arguments onto the run-time stack in **reverse** order.
2. Make the subroutine call.
3. Restore the stack pointer.

The following example assumes that the four parameters to be passed to the C function `foo()` are located in registers `d1.l`, `d2.l`, `d3.l`, and `d4.l`, respectively. The first four lines of assembly code push the arguments onto the run-time stack in reverse order. The **jsr** statement makes the subroutine call and the last two statements restore the stack pointer.

Example 5-31 Calling C Routines. The C function, `foo()`, is called from the following assembly code. Function `foo()` is declared as `int foo(int, int, int, int);`

```
; Notice that the first two parameters will be stored in the two registers
; d0.l and d1.l since they are of size 32 bit.
      move    d2.l,y:(r6)+      ; pushing arguments onto
      move    d3.l,y:(r6)+      ; run-time stack in reverse order
      move    d4.l,y:(r6)+
      jsr      Ffoo              ; subroutine call
      move    (r6+-3)           ; the stack size restored
```

5.4.5 Optimization Technique

The general template for out-of-line assembly code provides a clean template to build C callable functions. However, the DSP96002 microprocessor chips have powerful features such as multiple instruction execution (multiply and accumulate) and parallel data move operations that may allow additional optimization. After constructing the out-of-line assembly code from the general template, some hand-optimization can be performed by combining several assembly statements.

Information about these optimization techniques can be obtained from the DSP96002 User's Manual. Some optimization techniques which are related to the C compiler are

discussed in this section but additional optimization can be achieved using the architectural features described in the user's manual.

The r0 register (which is updated by the out-of-line assembly code template prologue) is required only when parameters are passed to the C function. If there are no parameters to be passed, then the following two assembly code lines are not required and can be omitted:

```

move    r0,y:(r6)+
lea     (r6)+,r0

```

The return address (SSH) was saved in the out-of-line assembly code prologue but it is only required when a C function calls another C function. A C function is called a leaf function if it does not call any other C function. In leaf functions, the return address is not needed and does not have to be saved.

Similarly the epilogue can be optimized in the same way as the prologue. For any leaf function or non parameter C function, the epilogue size can be reduced by eliminating:

```

move    (r6)-          ; for a leaf C function
move    y:(r6)-,ssh

```

or

```

move    y:(r6),r0      ; for a non-parameter C function

```

The test statement “**tst d0**” in the epilogue can be eliminated if the function does not return any values. The test statement may be required due to the C compiler's optimization features since it provides condition flags for an **if** statement in a function call. For example, if the out-of-line assembly function `foo()` is used in the statement `if(foo()) { }`, then the C compiler will not generate code to test the return value when a **jcc** statement is issued. This is primarily because the C compiler uses the condition flags which were executed at the end of the epilogue of `foo()`.

The “restoring all registers” and prologue can be combined to remove one extra move instruction (`move (r6)-`). The resulting combined code is:

```

move    (r6)-
; restore registers
move    y:(r6)-,ssh    ; epilogue
move    y:(r6),r0

```

A variety of optimizations can be achieved by combining the **move** instructions and main program code to utilize parallel moves (see Section 5.3.1.4 and Example 5-32 which point out possible optimizations in the comments). These and other DSP96002 specific

optimizations can dramatically improve the quality of the application specific library routines. A careful review of the DSP96002 User's Manual will be worthwhile for efficient library development.

Example 5-32 ReadX / ReadY Routines. This out-of-line assembly program (called memread.asm) reads the contents of X or Y memory and returns the data to the caller. The ReadX subroutine returns the X memory space contents pointed to by the input argument and the ReadY subroutine returns the Y memory content.

```
;;!cc
; memread.asm
;;

        section memread
        org    p:
ReadX   global ReadX                                ; prologue
                                                ;
        move   r1,y:(r6)+                        ; save register r1
        move   d0.l,r1                            ; the first parameter
        move   x:(r1),d0.l                        ; main program

        move   (r6)-                              ; restore register & epi-
logue
        move   y:(r6),r1                          ;
        tst    d0                                  ;
        rts                                         ;

ReadY   global ReadY                                ; prologue
                                                ;
        move   r1,y:(r6)+                        ; save register r1
        move   d0.l,r1                            ; the first parameter
        move   y:(r1),d0.l                        ; main program

        move   (r6)-                              ; restore register & epi-
logue
        move   y:(r6),r1                          ;
        tst    d0                                  ;
        rts                                         ;
        endsec
```


Chapter 6

Software-Hardware Integration

6.1 Overview

This chapter explains how the run-time environment may be changed and provides examples of these changes with their effects. The run-time environment provided with the compiler assumes, as a default, that the simulator is the target execution device. Several aspects of the default run-time environment must be altered in order to adapt the compiler to work with a custom hardware configuration.

The files which are alterable are discussed and classified according to effect. Aspects of the run-time environment such as: bootstrapping, interrupts and memory management are addressed individually.

6.2 Run-Time Environment Specification Files

The run-time environment is specified by three assembly language files: **crt096[x,y,l].asm**, **signal96[x,y,l].asm** and **setjmp96[x,y,l].asm** where x, y, or l denote the memory model (see chapters 2 and 4). Since the files provide a run-time environment for the C code, they may need to be modified if the run-time environment is to be customized.

The **crt0** file contains parameters which specify the C bootstrap code, memory configuration, memory management, interrupt vectors and other miscellaneous code. This file must be modified to match the software and hardware configuration because the memory configuration and interrupt vectors are determined by the hardware. The information in this manual on the crt0 file applies to DSP96KCC Version G1.0. The crt0 file may be different for other versions of this compiler.

The signal file, which is equivalent to a hardware interrupt, is implemented in the C environment. The signal file contains the code and data structures used to implement the **signal()** and **raise()** library functions. Changing this file is not recommended unless necessary since any change to this file requires detailed knowledge of the DSP96002 interrupt mechanism plus the **signal** and **raise** functions. This file is closely tied to the signal.h file.

The **setjmp** file contains code which implements the functions **setjmp()** and **longjmp()**. This file will probably never need to be modified unless the signal file is changed; however, if either the **setjmp** file or **setjmp.h** are modified, the code in both files must be made consistent. The source code for **setjmp()** and **longjmp()** is provided with DSP96KCC to allow modification should the signal mechanism need to be changed.

The operation of **setjmp()** and **longjmp()** is described in Section 6.5 and detailed implementation information can be obtained from the files provided with DSP96KCC.

6.3 crt0 File

The following subsections describe the various aspects under control of the **crt0** file.

6.3.1 Bootstrapping the C program

The processor enters a C program through the C bootstrap code in the **crt0** file. The C bootstrap code in **crt0** provides the C environment required to execute a C program. This environment includes a global data area, static data area, stack area, heap area, function calling mechanism, etc. This environment must be established before C programs will execute correctly.

The following steps are normally taken to initialize the processor to execute C code:

1. Jump from the chip reset vector to the C bootstrap code in **crt0** residing at **F_start**. Remember that the mode select pins on the chip control the chip operating mode when leaving reset which, in turn, controls the reset vector address (see the Motorola DSP96002 User's Manual for more details).
2. Configure all hardware registers needed (i.e., **omr**, host port, etc.). This is also a proper place to initialize any non-C related data structures or peripheral hardware such as the **bcra**, **bcrb**, **OnCE**, and **DMA**.
3. Load the Stack Pointer, **r6**, with a pointer to the base of the stack. Remember that the stack grows up (the value in the stack pointer gets greater as data is pushed). The value of **DSIZE** is generated by the linker and is the first address above the statically allocated data (C global and static variables). By default, this value is used as the initial stack pointer.
4. **Optional:** Initialize the frame pointer, **r0**. This may be of use to some debuggers when stack back traces are performed. Program execution will not be affected even if this initialization is omitted.
5. Call **main()** with instruction **jsr Fmain**. Notice that the label is **Fmain** and there are no parameters passed to the main function. The normal C compiler start-up passes two arguments but **g96k** does not pass any arguments because DSP96KCC does not support a hosted environment.

6. Label the end address of the C bootstrap code as **F__crt0_end**. This label is the program termination address that ends program execution. The two C functions, **abort()** and **exit()**, jump to this label in order to terminate the program. This label may be followed by a **stop** instruction and/or code that indicates program termination to another program such as a monitor.

Note: Labels which begin with a double underline (e.g., **__crt0_end**) in this manual have a space between the double underlines to visually separate them. **Do not separate the leading double underlines with a space when coding them** (i.e., code **__crt0_end** as **__crt0_end**).

Example 6-1 DSP96002 Operation Mode Change: Mode 4 has a reset vector of \$FFFF0000 which must contain a **jmp** to the C program bootstrap code. Adding the following code segment to the **crt0** file will change the bootstrap mode to Mode 4 (section 9 of DSP96002 User's Manual).

```
section mode4_reset
org p:$FFFF0000
jmp F__start      ; jump to the C start-up program.
endsec
```

Example 6-2 Hardware was designed to have a 256 byte ROM monitor located in the program memory space starting at \$00000000 and ending at \$FF. Program RAM starts at location p:\$100. The following changes to the **crt0** file will change the beginning location of the C bootstrap code to the first available RAM location (p:\$100). The DS statement allocates program space starting at p:\$00000000 and lets the ROM be located at address p:\$00000000. The org statement places the C bootstrap code at memory location p:\$100.

Change this portion of the **crt0** file:

```
org      p:
F__start

to:

org p:$00000000
ds $100
org p:$00000100
F__start
```

6.3.2 Memory Configuration and Management

The DSP96002 supports three memory spaces: program memory (p memory), x and y data memories. The DSP96002 design philosophy is to support concurrent access to these memory spaces to accelerate DSP operations. The C programming language does not support separate memory spaces but instead treats memory as a single memory space. As a result, there is a difference between hardware memory organization and C memory utilization.

DSP96KCC provides three different memory models to minimize these differences. They are the X memory model, Y memory model and L memory model. Selection of the appropriate memory model is made using the -m option of the control program, g96k (see Section 3.2.2). The memory configuration for each memory model is discussed later in this section. The **crt0** file can be modified to configure memory for any hardware design.

There are four data segments in the C programming environment. These are the program segment, global-static data segment, stack data segment and heap data segment. The program segment is located in program memory. The three data segments can be located in X memory, Y memory or L memory space (which is a concatenation of X and Y memory) depending on the memory model used.

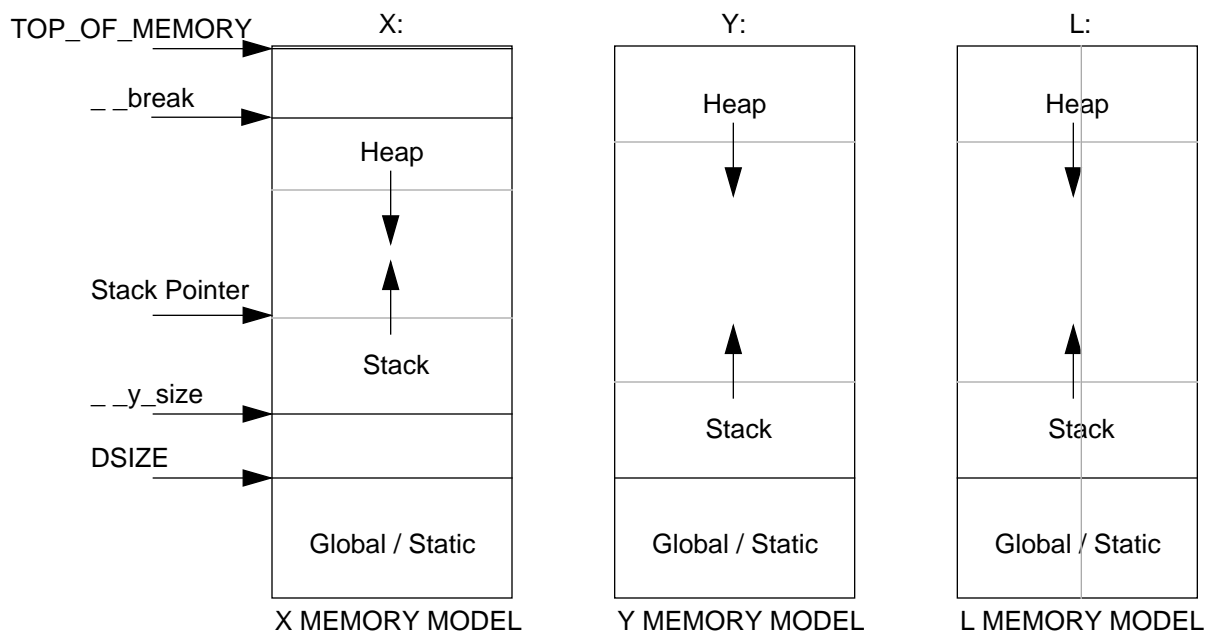


Figure 6-1 C Environment Memory Configuration

As indicated in Figure 6-1, global and static data reside at the bottom of the available data memory and the top address of the global and static data area, which is called DSIZE, is

set by the linker. The constant TOP_OF_MEMORY is defined to indicate the top of the entire available memory.

The two data segments, heap and stack, are located as shown Figure 6-1. The stack is located so that it can grow up and the heap is located so that it can grow down. There are two locations used to indicate the initial values for the heap pointer and stack pointer. These locations are `__y_size` and `__break` and are initialized in the `crt0` file as `DSIZE` and `TOP_OF_MEMORY`, respectively.

In summary, two variables `__y_size` and `__break` and the constant `TOP_OF_MEMORY` are used to configure the data segment. The program segment is configured using the `org` statement in the `crt0` file. The variable, `__y_size`, should be initialized with the initial stack pointer and the variable, `__break`, should be initialized with the initial heap pointer.

Caution

The stack and heap regions must not contain on-chip peripheral memory or the static or global data regions. **Also**, no region may be reconfigured after the C main function is called. Variables `__y_size` and `__break` should not be altered by an arbitrary function since they are utilized by system level libraries such as `malloc` and `free`.

Example 6-3 Fast Stack: In this example, the stack is required to reside in an 256k SRAM starting at L:\$4000. The following program reserves the stack space using **org** and **ds** statements and sets the initial stack pointer to the SRAM stack area.

Add this section to the `crt0` file:

```
section fast_ram
org      L:$4000
ds       $40000
endsec
```

Change the following line of C bootstrap code in the `crt0` file:

```
move y:F__y_size,r6
```

to:

```
move #L$4000,r6
```

Example 6-4 Fast Heap: The heap is required to reside in an 8k SRAM starting at L:\$4000. The following program reserves the heap space using **org** and **ds** statements and sets the initial heap pointer to the SRAM heap area.

Add this section to the `crt0` file:

```

section fast_ram
org    L:$4000
ds     $2000
endsec

```

Change the following line of C bootstrap code in the **crt0** file:

```
TOP_OF_MEMORY    equ    $7FFFFFFF
```

to:

```
TOP_OF_MEMORY    equ    $50000FFF
```

Sometimes hardware configurations map more than one memory space into a single physical memory. Other implementations partially populate various address spaces leaving holes. Some may have different regions with fast memory and slow memory. All of these special cases can usually be handled by modifying the **crt0** file.

When multiple memory spaces are mapped into a single physical memory, the memory must be partitioned among the memory spaces. A way to restrict the linker from overlapping these memory spaces is needed. For example, suppose that both the Y and P spaces are mapped into the same 64k physical RAM and need to be partitioned with the low 48k for program memory and the high 16k for data memory.

The linker can be restricted from allocating across holes in physical memory by using the **org** and **ds** directives to confiscate those addresses. Note that the linker may not automatically take advantage of memory which is present between holes. It may be required to manually place assembly language data structures to utilize this memory.

6.3.3 Interrupt Vectors

The interrupt vector locations for the DSP96002 (a.k.a. “interrupt source” in Section 8 of the DSP96002 User’s Manual) contain one or two instructions each to be executed when the interrupt assigned to that location occurs. There are 512 interrupt vectors available, all of which should be initialized with some value to avoid undefined behavior resulting from an unexpected interrupt.

The **crt0** file contains code to initialize these interrupt vectors. By default, all vectors are initialized with the instruction **jsr Fabort**. The first element of the vector table, which is the hardware RESET interrupt, is initialized with the instruction **jmp F__start**. The purpose of the C function **abort()**, which is labeled as **Fabort** in the assembly environment, is to stop program execution and leave the chip in the STOP mode. The **F__start** label is the program address of the C program bootstrap code which calls the C **main()** function after execution.

Interrupt vectors that are to be used must be reprogrammed to point to the interrupt service routines instead of the **abort()** function. Initialization of the interrupt vectors in the **crt0** file reduces the size of the resulting application program and may increase its speed.

The following **crt0** code segment (called section reset) is the default interrupt vector table initialization.

```
section reset
org    p:$0
jmp    F__start
org    p:$2
dup    $7F
jsr    Fabort
endm
org    p:$100
rts
nop
org    p:$100
dup    $7F
jsr    Fabort
endm
endsec
```

The interrupt vector table can be changed to point to user-provided interrupt service routines instead of the **abort()** routine in this portion of **crt0**. Example 6-5 illustrates how to initialize pointers to these user-provided interrupt service routines.

Example 6-5 User-defined Interrupt Vector Table: Assume the hardware supports all interrupts and each interrupt service routine is located at the address labeled interruptXXX (where XXX is the value of the interrupt vector). The following code initializes the interrupt vector table. Each service routine starting at interruptXXX can be programmed in assembly language as shown in Example 6-6.

```

section reset
org      p:$0
jmp      F__start
jsr      interrupt002
jsr      interrupt004
jsr      interrupt006
jsr      interrupt008
jsr      interrupt010

...
jsr      interrupt1FE
endsec

```

Example 6-6 Interrupt Service Routine: This service routine updates the global variable `Fdma_cnt` after each DMA data transfer. The value of the counter `dma_cnt` indicates the number of the DMA data transfers performed or the number of the data blocks finished by the DMA. The DMA data transfer should be initiated by programming the DMA data control registers in order for this interrupt service routine to be invoked every time the DMA data transfer has been completed. See Section 7.5.2 of the DSP96002 User's Manual for the information on how to set up the DMA controller. This program is based on the Y or L memory model of the C compiler.

```

;;!cc

section interrupt
org      p:
global   interrupt010

interrupt010
move     (r6)+          ; adjust C stack pointer
move     d1.l,y:(r6)    ; save the d1.l register
move     y:Fdma_cnt,d1.l ; retrieve the variable dma_cnt
inc      d1             ; increment the variable
move     d1.l,y:Fdma_cnt ; save the result
move     y:(r6)-,d1.l   ; restore the d1.l register
rti      ; return from interrupt service
endsec

```

Notice that fast interrupts can also be programmed by modifying the **crt0** file in the same way as for the long interrupts (see Chapter 8 of the DSP96002 User's Manual for more information on fast and long interrupts).

6.3.4 Miscellaneous Code

There are other data structures and code related to the run-time environment in the `crt0` file. They are:

1. Error code variable, **errno**, is a global integer used to record failure codes in C library calls. This error code variable will exist if C library calls are used. This variable can be utilized as a debugging aid in order to check which error code is returned from the C function calls.
2. Heap-stack checking window variable, **__stack_safety**, is a global variable declared in the `crt0` file that is used by the heap allocation routines, **malloc()**, **calloc()** and **realloc()** to avoid heap-stack growth collisions. If the distance between the bottom of the heap and the top of the stack is less than the value contained in **__stack_safety**, then the heap allocation routine will return an error code indicating that no more memory is available. The value may be set as required by the application since the window **__stack_safety** is declared as a global variable.
3. Memory limit variable, **__mem_limit**, is a global variable declared in the `crt0` file and used by the library routine **brk()** to disallow any meaningless memory requests. This variable should have the same value as **__break** upon entry into **main()**. For information on how to use the function **brk()**, refer to Appendix B in this manual.
4. Dynamic time variable, **__time**, is a global variable declared in the `crt0` file and used as a **volatile** timer counter by the simulator. This variable is updated by the DSP96000 simulator (run96sim) every clock cycle. Examining this variable allows the programmer to determine program execution time. This variable is only used by the simulator and can be omitted when the program is to be executed by hardware.
5. Printf stub function, **__printf_end()**, is defined in `crt0` and is called when a message is printed on the screen. Upon calling this function, the d0 register will contain the address of the string to be printed. A monitor program can use this function to support screen output of the **printf()** function.

All of these variables, constants, functions and pointers are related to the run-time environments that are used by C library functions and must be properly set.

6.4 Signal File

The hardware level interrupt mechanism (see section 8.3 of the DSP96002 User's Manual) is more efficient than the **signal()** function. However, in many cases interrupts can be handled in the C environment and it is often more preferable to do so. There are two func-

tions, **signal()** and **raise()**, used to support programming interrupt service routines in C. These functions are not associated with the **crt0** file. Although they are more complicated than the simple hardware interrupt vector table discussed in Section 6.3.3 (see section 8.2 of the DSP96002 User's Manual) they provide very handy tools for the C programmer. A thorough knowledge of the **signal()** function and the C environment is needed in order to modify the **signal()** function. This section describes how the **signal()** function is currently implemented.

6.4.1 Signal()

The **signal()** function is passed two arguments:

1. A **signal number** — On the DSP96002 processor, the signal number corresponds directly to the interrupt vector address. Notice that the signal number is not an integer sequence but is an even number.
2. A **function pointer** — The function pointer passed is assumed to belong to a C function either generated by this compiler or by assembly code.

Signal() performs the following three steps when binding the specified signal number and function:

1. The instruction **jsr F_ _c_sig_goto_dispatch+<signal number>** is placed into the interrupt table location specified by the signal number.
2. The function pointer passed is entered into the table **_ _c_sig_handlers**, which is used to store pointers to C signal handlers, indexed by the signal number.
3. The old signal handler address is returned.

Once the signal number and specified function are bound, the instruction

```
jsr    F_ _c_sig_goto_dispatch+<signal number>
```

is executed upon receiving the interrupt, where the **F_ _c_sig_goto_dispatch** variable is the starting address of a table of

```
jsr    F_ _c_sig_dispatch
```

instructions and each **jsr** instruction points to an interrupt service routine. The pseudo-function **_ _c_sig_dispatch()** is used to calculate the actual C interrupt routine.

All registers are saved before the **_ _c_sig_dispatch()** function calls the C signal handler. Pseudo function **_ _c_sig_dispatch()** then calculates the signal number using the return address program counter of the ssh. Since the signal number is the same as the interrupt vector address, each entry of the **_ _c_sig_goto_dispatch** table corresponds to an inter-

rupt vector. The pseudo function uses the signal number to fetch the actual C signal handler from the `__c_sig_handlers` table which is the C function pointer table.

Once the C signal handler is fetched from the `__c_sig_handler` table, its entry is replaced with the default signal handler **SIG_DFL**. This replacement is in compliance with the ANSI standard and forces the next signal service to abort. In most situations, this feature is not needed because any given interrupt will always invoke the same interrupt service routine. Resetting **signal()** after each C service routine or modifying this file so that it does not replace the table entry with **SIG_DFL** will change the interrupt service scheme. Modification of the signal file is only recommended when optimization of the service time is critical to the application.

Upon return from the C signal handler, all the registers are restored. Finally, the **rti** instruction is executed to return to the code that was executing when the interrupt occurred. Notice two factors in this scheme,

1. **all registers** are saved and restored before and after the C signal handler and
2. the **rti** instruction is executed by the `__c_sig_dispatch()` function.

Caution

The signal handler must not contain the **rti** instruction at the end of the program regardless which language is used to program the interrupt. The signal handler does not need to save or restore any context or registers. The function `__c_sig_dispatch()` will not act like a normal C function because it never returns to its caller. Instead, it will return to the code that was executing when the interrupt happened by executing the **rti** instruction.

Assembly language interrupt handlers can coexist with C signal handlers. The code in the signal file will not alter any interrupt vector except the one specified by the signal number passed to the **signal()** function (see the first of the three steps above). The C signal interface could be used with an assembly routine but would be unnecessarily slow. To use an assembly language interrupt handler, alter the vector (e.g., `interrupt008`) with a **jsr** to it (e.g., `jsr interrupt008`) or use a fast interrupt routine.

6.4.2 Raise()

The **raise()** function is used to simulate an interrupt. The code in **raise()** simply calls the entry in `__c_sig_goto_dispatch` that is matched to the interrupt vector specified by the signal number passed.

The ANSI standard signal handlers **SIG_DFL**, **SIG_ERR** and **SIG_IGN** are implemented by the hand-coded functions `__sig_dfl()`, `__sig_err()` and `__sig_ign()`, respectively.

1. **SIG_DFL** notes that the interrupt happened by incrementing `__sig_drop_count` and then returns.
2. **SIG_ERR** calls **abort()** and never returns.
3. **SIG_IGN** returns without any effect (i.e., ignore).

The mechanisms used to implement the C signal interface may be altered to fit a particular hardware application. Any series of alterations applied to the signal file must leave an implementation conforming to the ANSI standard X3.159 for C. Alteration of the signal file is done at one's own risk and is not generally advised. Again, the contents of the signal file must remain consistent with the include file **signal.h**.

6.5 Setjmp File

The functions **setjmp()** and **longjmp()** are implemented in the **setjmp** file. The **setjmp()** function stores the current process status (i.e., the current execution context) in a buffer that is passed. The **longjmp()** function is used to restore the process status to the execution context which was saved by **setjmp()**.

Saving the current execution context is done by saving the stack pointer, the return program counter value, the frame pointer and all of the callee-save registers into the buffer. The buffer that is passed to **setjmp()** should have enough space for the saving process. The structure **jmp_buf** defined in **setjmp.h** allocates the buffer space needed for the operation. The function **setjmp()** always returns a zero.

The function **longjmp()** takes two arguments, an environment buffer and a return value. It restores all registers from the buffer passed, including the frame pointer and stack pointer. It then places the return value passed into register **d0.l** and sets the **ccr** to reflect the return value just stored in register **d0.l**. The function **longjmp()** discards the return program counter on the hardware stack and jumps to the address pointed to by the program counter stored in the buffer.

This file must conform to the include files **setjmp.h** and **longjmp.h**. Since these two algorithms are very straightforward, modification of the file may be not needed. If modification is absolutely necessary, then the ANSI standard of the functions **setjmp()** and **longjmp()** should be followed.

Appendix A

Programming Support

A.1 Standard ANSI Header Files

Each function provided in the ANSI C library **lib96c[x,y,l].clb** is declared with the appropriate prototype in a header, whose contents are made available by the **#include** preprocessing directive. In addition to function prototypes, the header also defines all type and MACRO definitions required for ANSI conformance.

The ANSI standard header files provided are:

<assert.h>	<locale.h>	<stddef.h>
<ctype.h>	<math.h>	<stdio.h>
<errno.h>	<setjmp.h>	<stdlib.h>
<float.h>	<signal.h>	<string.h>
<limits.h>	<stdarg.h>	<time.h>

In general, header files may be included in any order and as often as desired with no negative side effects. The one exception occurs when including **<assert.h>**. The definition of the assert macro depends on the definition of the macro **NDEBUG**.

The contents of the standard header files provided are **exactly** as described by the ANSI document X3.159-1989 dated December 14, 1989. However, there are several **hosted** library sub-routines that are currently unsupported [see section A.2.3, Missing]. The unsupported routine prototypes are included for completeness and follow the banner,

```
/***** UNSUPPORTED HOSTED ANSI ROUTINES *****/
```

A.2 ANSI C Library Sub-routines

There are three library images provided with each compiler,

- lib96cx.clb — **X** memory model
- lib96cy.clb — **Y** memory model
- lib96cl.clb — **L** memory model

Each library contains all of the ANSI defined sub-routines compiled and/or assembled for the specific memory model. The C and assembly language source for each library routine is distributed free of charge with the compiler.

A.2.1 Hosted

For the first release, **printf()** is the only hosted library routine supported. Support is provided via the execution devices **run96sim** and **run96ads**. Future releases will include the remaining routines such as **open()**, **scanf()**, etc.

For a description of run96sim and run96ads, see Appendix C.

A.2.2 Non-Hosted

For the first release, **all** non-hosted ANSI defined library routines in both object and source code are provided.

A.3 Forcing Library Routines Out-of-line

For performance reasons, several run-time library routines have been created to be used in-line by the compiler. The compiler in-lines a sub-routine by replacing the occurrence of the sub-routine call with the body of the sub-routine itself. This provides execution time benefits:

1. Eliminates sub-routine call overhead. This is a substantial portion of the run-time for small library routines.
2. Exposes more optimization opportunities to the optimizer.

The following library routines are automatically in-lined by the compiler when their header file is included:

header: **ctype.h**

- | | | | |
|-----------|-----------|------------|-----------|
| • isalnum | • isalpha | • iscntrl | • isdigit |
| • isgraph | • islower | • isprint | • ispunct |
| • isspace | • isupper | • isxdigit | • tolower |
| • toupper | | | |

header: **math.h**

- | | | | |
|--------|--------|---------|--------|
| • ceil | • fabs | • floor | • fmod |
|--------|--------|---------|--------|

header: **stdlib.h**

- | | |
|-------|--------|
| • abs | • labs |
|-------|--------|

header: **string.h**

- | | |
|----------|----------|
| • strcmp | • strcpy |
|----------|----------|

When it is necessary to disable this feature, possibly for debugging or decreasing program size, simply do one of the following.

1. Add the following line to each C module (or once to a common header file)

```
#undef ROUTINE_NAME
```

where ROUTINE_NAME is the library routine that must be forced out-of-line.
For example, to force the library routine **ceil** out-of-line:

```
#undef ceil
```

2. Use the command-line option **-U**, see Chapter 3, Control Program Options. This will force the library routine to be called **for this compilation**. If the code is re-compiled, the **-U** option must be used again.

```
% g96k -Uceil file.c
```

3. Do **not** include the header file.

A.4 Function Descriptions

The following section describes each function in complete detail. The synopsis provides the syntax of the function and the options section discusses each option in detail. Many function descriptions also include references to related functions and an example of how to use the function. The list shown below provides an abbreviated description of each function.

abort	— Force abnormal program termination.
abs	— Absolute value of integer.
acos	— Arc cosine.
asin	— Arc sine.
atan	— Arc tangent.
atan2	— Arc tangent of angle defined by point y/x.
atexit	— Register a function for execution at normal program termination.
atof	— String to floating point.
atoi	— String to integer.
atol	— String to long integer.
bsearch	— Perform binary search.
calloc	— Dynamically allocate zero-initialized storage for objects.
ceil	— Ceiling function.
cos	— Cosine.
cosh	— Hyperbolic cosine.
div	— integer division with remainder.
ldiv	— Long integer division with remainder.
exit	— Terminate program normally.

exp	— Exponential, e^x .
fabs	— Absolute value of a double.
floor	— Floor function.
fmod	— Floating point remainder.
free	— Free storage allocated by calloc, malloc, and realloc.
frexp	— Break a floating point number into mantissa and exponent.
isalnum	— Test for alphanumeric character.
isalpha	— Test for alphabetic character.
iscntrl	— Test for control character.
isdigit	— Test for numeric character.
isgraph	— Test for printing character, excluding space and tab.
islower	— Test for lower-case alphabetic characters.
isprint	— Test for printing character, excluding '\t'.
ispunct	— Test for punctuation character.
isspace	— Test for white-space character.
isupper	— Test for upper-case alphabetic character.
isxdigit	— Test for hexadecimal numeric character.
labs	— Absolute value of a long integer.
ldexp	— Multiply floating point number by 2.
ldiv	— Long integer division with remainder.
log	— Natural logarithm, base e.
log10	— Base ten logarithm.
longjmp	— Execute a non-local jump.
malloc	— Dynamically allocate uninitialized storage.
mblen	— Length of a multibyte character.
mbstowcs	— Convert multibyte string to wide character string.
mbtowc	— Convert a multibyte character to a wide character.
memchr	— Find a character in a memory area.
memcmp	— Compare portion of two memory areas.
memcpy	— Copy from one area to another.
memmove	— Copy storage.
memset	— Initialize memory area.
modf	— Break a double into its integral and fractional parts.
perror	— Print error message.
pow	— Raise a double to a power.
putchar	— Write a character to standard output.
puts	— Write a string to standard output.
qsort	— Quick sort.
raise	— Raise a signal.
rand	— Pseudo- random number generator.

realloc	— Change size of dynamically allocated storage area.
setjmp	— Save a reference of the current calling environment for later use by longjmp.
signal	— Set up signal handler.
sin	— Sine.
sinh	— Hyperbolic Sine.
sprintf	— Print to a string.
sqrt	— Square root.
srand	— Seed the pseudo-random number generator.
strcat	— Concatenate two strings.
strchr	— Find first occurrence of a character in a string.
strcmp	— Compare two strings.
strcoll	— Compare two strings based on current locale.
strcpy	— Copy one string into another.
strcspn	— Compute the length of the prefix of one string consisting entirely of characters from another.
strerror	— Map error code into an error message string.
strlen	— Determine length of a string.
strncat	— Concatenate a portion of one string to another.
strncmp	— Compare a portions of two strings.
strncpy	— Copy a portion of one string into another.
strpbrk	— Find the first occurrence of a character from one string in another.
strrchr	— Find the last occurrence of a character in a string.
strspn	— Compute the length of the prefix of a string contained in another string.
strrchr	— Find the last occurrence of a character from one string in another.
strstr	— Find the first occurrence of one string in another.
strtod	— String to double.
strtok	— Break string into tokens.
strtol	— String to long integer.
strtoul	— String to unsigned long integer.
strxfrm	— Transform a string into locale-independent form.
tan	— Tangent.
tanh	— Hyperbolic tangent.
tolower	— Convert uppercase character to lowercase.
toupper	— Convert lowercase character to uppercase.
wcstombs	— Convert wchar_t array to multibyte string.
wctomb	— Convert wchar_t character to multibyte character.

NAME

abort — Force abnormal program termination.

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

DESCRIPTION

The **abort** function causes the program to terminate abnormally unless the signal **SIGABRT** is being caught and the signal handler does not return. The unsuccessful termination value, **-1**, is returned to the host environment (gdb96, exec96). The **abort** function will not return to its caller.

SEE ALSO

exit — Terminate a program normally.
signal — Set up a signal handler.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf("-- make abort call --\n");
    abort();
    printf("this line not reached\n");
}
```

prints to standard output:
-- make abort call --

NAME

abs — Absolute value of integer.

SYNOPSIS

```
#include <stdlib.h>
int abs(int j);
```

DESCRIPTION

The **abs** function returns the absolute value of **j**.

When the header file **stdlib.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

SEE ALSO

fabs — Absolute value of a double.
labs — Absolute value of a long.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    int neg = -709;
    printf("-- abs(%d) == %d --\n", neg,abs( neg ));
}
```

prints to standard output:

```
-- abs(-709) == 709 --
```

NAME

acos — Arc cosine.

SYNOPSIS

```
#include <math.h>
double acos( double x );
```

DESCRIPTION

The **acos** function computes the principle value of the arc cosine of **x** in the range [0.0, π], where **x** is in radians. If **x** is not in the range [-1, +1], a domain error occurs, **errno** is set to **EDOM** and a value of 0.0 is returned.

SEE ALSO

cos — Cosine.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double point;

    for ( point = -0.8 ; point < 1.0 ; point += 0.2 )
    {
        printf( "%f ", acos( point ) );
        if ( point >= 0.0 ) printf( "\n" );
    }
}
```

prints to standard output:

```
2.498092 2.214297 1.982313 1.772154 1.570795
1.369438 1.159279 0.927295 0.643501
```


NAME

asin — Arc sine.

SYNOPSIS

```
#include <math.h>

double asin( double x );
```

DESCRIPTION

The **asin** function computes the principle value of the arc sine of **x** in the range $[-\pi/2, +\pi/2]$, where **x** is in radians. If **x** is not in the range $[-1, +1]$, a domain error occurs, **errno** is set to **EDOM** and a value of 0.0 is returned.

SEE ALSO

sin — Sine.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double point;

    for ( point = -0.8 ; point < 1.0 ; point += 0.2 )
    {
        printf( "%f ", asin( point ) );
        if ( point >= 0.0 ) printf( "\n" );
    }
}

prints to standard output:
-0.927295 -0.643501 -0.411517 -0.201358 -0.000000
0.201358 0.411517 0.643501 0.927295
```

NAME

atan — Arc tangent.

SYNOPSIS

```
#include <math.h>
double atan( double x );
```

DESCRIPTION

The **atan** function computes the principle value of the arc tangent of **x** in the range $[-\pi/2, +\pi/2]$, where **x** is in radians.

SEE ALSO

tan — Tangent.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double point;

    for ( point = -0.8 ; point < 1.0 ; point += 0.2 )
    {
        printf( "%f ", atan( point ) );
        if ( point >= 0.0 ) printf( "\n" );
    }
}

prints to standard output:
-0.674741 -0.540420 -0.380506 -0.197396 -0.000000
0.197396 0.380506 0.540420 0.674741
```

NAME

atan2 — Arc tangent of angle defined by point y/x.

SYNOPSIS

```
#include <math.h>

double atan2( double y, double x );
```

DESCRIPTION

The **atan2** function computes the principle value of the arc tangent of **y/x** using the signs of both arguments to determine the quadrant of the return value. If both arguments are zero, errno is set to **EDOM** and 0.0 is returned.

<u>argument range</u>	<u>output range</u>
$y \geq 0.0, x \geq 0.0$	$[0.0, \pi/2]$
$y \geq 0.0, x < 0.0$	$[\pi/2, \pi]$
$y < 0.0, x < 0.0$	$[-\pi, -\pi/2]$
$y < 0.0, x \geq 0.0$	$[-\pi/2, 0.0]$

SEE ALSO

atan — Arc tangent.
tan — Tangent.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf("atan2(7.09,7.09) == %f\n", atan2(7.09,7.09));
    printf("atan2(-7.09,7.09) == %f\n", atan2(-7.09,7.09));
    printf("atan2(7.09,-7.09) == %f\n", atan2(7.09,-7.09));
    printf("atan2(-7.09,-7.09) == %f\n",atan2(-7.09,-7.09));
}
```

prints to standard output:

```
atan2( 7.09, 7.09 ) == .785398
atan2( -7.09, 7.09 ) == -.785398
atan2( 7.09, -7.09 ) == 2.356194
atan2( -7.09, -7.09 ) == -2.356194
```

NAME

atexit — Register a function for execution at normal program termination.

SYNOPSIS

```
#include <stdlib.h>

int atexit( void (*func)(void) );
```

DESCRIPTION

The **atexit** registers a function **func** that will be called at normal program execution. The registered function is called without arguments and returns nothing.

A total of **32** functions may be registered and will be called in the order in which they were registered. The **atexit** function returns zero if registration succeeds and a non-zero value for failure.

SEE ALSO

exit — Terminate a program normally.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    atexit( func_1 );
    atexit( func_2 );
    printf( "-- testing atexit --\n" );
}

void func_1( void )
{
    printf ( "first function called\n" );
}

void func_2( void )
{
    printf ( "second function called\n" );
}

prints to standard output:
-- testing atexit --
first function called
second function called
```

NAME

atof — String to floating point.

SYNOPSIS

```
#include <stdlib.h>

double atof( const char* nptr );
```

DESCRIPTION

The **atof** function converts the string pointed to by **nptr** to a double. If the result can not be represented, the behavior is undefined. This is exactly equivalent to:

```
strtod( nptr, (char **) NULL );
```

SEE ALSO

atoi — String to integer.
atol — String to long integer.
strtod — String to double.
strtol — String to long integer.
strtoul — String to unsigned long integer.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf( "atof( \"7.09\" ) == %f\n", atof( "7.09" ) );
}
```

prints to standard output:
atof("7.09") == 7.090000

NAME

atoi — String to integer.

SYNOPSIS

```
#include <stdlib.h>
int atoi( const char* nptr );
```

DESCRIPTION

The **atoi** function converts the string pointed to by **nptr** to an integer. If the result can not be represented, the behavior is undefined. This is exactly equivalent to:

```
(int) strtol( nptr, (char **) NULL, 10 );
```

SEE ALSO

atof — String to double.
atol — String to long integer.
strtod — String to double.
strtol — String to long integer.
strtoul — String to unsigned long integer.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf( "atoi( \"709\" ) == %d\n", atoi( "709" ) );
}
```

prints to standard output:

```
atoi( "709" ) == 709
```

NAME

atol — String to long integer.

SYNOPSIS

```
#include <stdlib.h>

long atol( const char* nptr );
```

DESCRIPTION

The **atol** function converts the string pointed to by **nptr** to a long integer. If the result can not be represented, the behavior is undefined. This is exactly equivalent to:

```
strtol( nptr, (char **) NULL, 10 );
```

SEE ALSO

atof — String to double.
atoi — String to long integer.
strtod — String to double.
strtol — String to long integer.
strtoul — String to unsigned long integer.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf( "atol( \"709\" ) == %ld\n", atol( "709" ) );
}
```

prints to standard output:

```
atol( "709" ) == 709
```

NAME

bsearch — Perform binary search.

SYNOPSIS

```
#include <stdlib.h>

void bsearch(      const void* key, const void* base,
                  size_t nmemb, size_t size,
                  int (*compare) (const void*, const void* ) );
```

DESCRIPTION

The **bsearch** function searches an array on **nmemb** objects (the initial element of which is pointed to by **base**) for an element that matches the object pointed to by **key**. The size of each element is specified by **size**.

The contents of the array must be in ascending order according to a user supplied comparison function, **compare**. The **compare** function is called with two arguments that must point to the **key** object and to an array member, in that order. Also, **compare** must return an integer that is less than, equal to, or greater than zero for the **key** object to be respectively considered less than, equal to or greater than the array element.

SEE ALSO

qsort — Perform quick sort.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

char* stuff[6] =
{
    "fred", "flintstone", "driving", "on", "bald", "feet"
};

static int compare( char** key, char** aelement )
{
    return( strcmp( (char*) key, (char*) aelement ) );
}

void main()
{
    char* p;
    char* key = "bald";
    p = bsearch( &key, stuff, 6, sizeof(char*), compare );
    if ( p )
    {
        printf( "YES, fred flintstone drives on bald feet\n" );
    }

    else
    {
        printf( "fred brings alice the meat\n" );
    }
}
```

prints to standard output:

YES, fred flintstone drives on bald feet

NAME

calloc — Dynamically allocate zero-initialized storage for objects.

SYNOPSIS

```
#include <stdlib.h>
```

```
void* calloc( size_t nmemb, size_t size );
```

DESCRIPTION

The **calloc** function allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all bits equal zero. If space can not be allocated, **calloc** returns a NULL pointer.

SEE ALSO

alloca	—	Allocate storage in the stack frame.
free	—	Free dynamically allocated storage.
malloc	—	Dynamically allocate uninitialized storage.
realloc	—	Alter size of previously dynamically allocated storage.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

int* iptr;

/* allocate space for 709 integers */
void main()
{
    iptr= (int*) calloc( 709, sizeof(int) );

    if ( iptr != NULL )
    {
        /* check first entry for zero initialization */
        if ( *iptr != 0 )
        {
            printf( "error: calloc failed to initialize\n" );
        }

        else
        {
            printf( "success: calloc ok\n" );
        }
    }

    else
    {
        printf( "error: calloc failed\n" );
    }
}
```

prints to standard output:
success: calloc ok

NAME

ceil — Ceiling function.

SYNOPSIS

```
#include <math.h>
double ceil( double x );
```

DESCRIPTION

The **ceil** function returns the smallest integer greater than or equal to **x**.

When the header file **math.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

SEE ALSO

floor — Floor function.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf( "ceil( 7.09 ) == %f\n", ceil( 7.09 ) );
}
```

prints to standard output:

```
ceil( 7.09 ) == 8.000000
```

NAME

cos — Cosine.

SYNOPSIS

```
#include <math.h>
double cos( double x );
```

DESCRIPTION

The **cos** function computes and returns the cosine of **x**, measured in radians.

SEE ALSO

acos — Arc cosine of an angle.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "cos( 45.0 ) == %f\n", cos( 45.0 ) );
}

prints to standard output:
cos( 45.0 ) == 0.525322
```

NAME

cosh — Hyperbolic cosine.

SYNOPSIS

```
#include <math.h>

double cosh( double x );
```

DESCRIPTION

The **cosh** function computes and returns the hyperbolic cosine of **x**. If the value of **x** is too large, a range error occurs, setting **errno** to **ERANGE** and causes **cosh** to return **HUGE_VAL**.

SEE ALSO

sinh — Hyperbolic sin.
tanh — Hyperbolic tangent.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "cosh( 3.1415 ) == %f\n", cosh( 3.1415 ) );
}
```

prints to standard output:
cosh(3.1415) == 11.590883

NAME

div — integer division with remainder.

SYNOPSIS

```
#include <stdlib.h>

div_t div( int numer, int denom );
```

DESCRIPTION

The **div** function computes the quotient and remainder of the division of the numerator **numer** by the denominator **denom** and returns them in a structure of type **div_t**. If the result can not be represented, the behavior is undefined.

SEE ALSO

ldiv — Long integer division with remainder.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    div_t result;
    int    numer = 709, denom = 96;

    result = div( numer, denom );
    printf( "quotient == %d\t", result.quot );
    printf( "remainder == %d\n", result.rem);
}
```

prints to standard output:
quotient == 12remainder == 37

NAME

exit — Terminate program normally.

SYNOPSIS

```
#include <stdlib.h>

void exit( int status );
```

DESCRIPTION

The **exit** function causes normal program termination to occur. Any functions registered with the function **atexit** are called in the order in which they were registered. **status** is returned to the environment (gdb96, exec96). If more than one call is made to **exit**, the result is undefined.

SEE ALSO

abort — Cause a program to terminate abnormally.
atexit — Register functions to be called at normal termination.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    printf( "-- exit test --\n" );
    exit ( 0 ); /* return with 0 status */
    printf( "Error: exit made this unreachable\n" );
}
```

prints to standard output:

```
-- exit test --
```


NAME

exp — Exponential, e^x .

SYNOPSIS

```
#include <math.h>

double exp( double x );
```

DESCRIPTION

The **exp** function computes and returns e^x . If the value of **x** is too large, a range error occurs with **errno** being set to **ERANGE** and **exp** returns **HUGE_VAL**. If the value of **x** is too small, a range error will also occur with **errno** being set to **ERANGE** and **exp** returns 0.0.

SEE ALSO

ldexp — Multiplying a number by a power of two.
pow — Raising a number to a power.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "exp( 7.09 ) == %f\n", exp( 7.09 ) );
}
```

prints to standard output:

```
exp( 7.09 ) == 1199.907686
```

NAME

fabs — Absolute value of a double.

SYNOPSIS

```
#include <math.h>

double fabs( double x );
```

DESCRIPTION

The **fabs** function computes and returns the absolute value of double **x**.

When the header file **math.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

SEE ALSO

abs — Absolute value of an integer.
labs — Absolute value of a long integer.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double pos, neg = -7.09;

    pos = abs( neg );
    printf( "-- absolute value of %d == %d --\n", neg, pos );
}
```

prints to standard output:

-- absolute value of -7.09 == 7.09 --

NAME

floor — Floor function.

SYNOPSIS

```
#include <math.h>
double floor( double x );
```

DESCRIPTION

The **floor** function returns the largest integer not greater than **x**.

When the header file **math.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

SEE ALSO

ceil — Ceiling function.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "floor( 7.09 ) == %f\n", floor( 7.09 ) );
}
```

prints to standard output:

```
floor( 7.09 ) == 7.000000
```

NAME

fmod — Floating point remainder.

SYNOPSIS

```
#include <math.h>

double fmod( double x, double y );
```

DESCRIPTION

The **fmod** function computes and returns the floating point remainder r of x/y . The remainder, r , has the same sign as x and $x == i * y + r$, where $|r| < |y|$. If x and y can not be represented, the result is undefined.

When the header file **math.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "fmod( -10.0, 3.0 ) == %f\n", fmod( -10.0, 3.0 ) );
}
```

prints to standard output:

```
fmod( -10.0, 3.0 ) == -1.00000000
```

NAME

free — Free storage allocated by **calloc**, **malloc**, and **realloc**.

SYNOPSIS

```
#include <stdlib.h>

void free( void* ptr);
```

DESCRIPTION

The **free** function causes the space pointed to by **ptr** to be deallocated. Once deallocated it is available to be used again by future dynamic memory allocation requests. If **ptr** is NULL, **free** returns immediately. If the space pointed to by **ptr** has already been deallocated by a previous call to **free** or **realloc**, the behavior is undefined.

SEE ALSO

alloca — Allocate storage in the stack frame.
calloc — Dynamically allocate zero-initialized storage.
malloc — Dynamically allocate uninitialized storage.
realloc — Alter size of previously dynamically allocated storage.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char* alloc;
    if ( ( alloc = (char*) malloc( 709 ) ) == NULL )
    {
        printf( "malloc error\n" );
        exit ( -1 );
    }

    /* free 709 words of memory */
    free( alloc );
}
```

NAME

frexp — Break a floating point number into mantissa and exponent.

SYNOPSIS

```
#include <math.h>

double frexp( double value, int* exp );
```

DESCRIPTION

The **frexp** function breaks a floating point number into a normalized fraction (the mantissa) and an integral power of 2 (the exponent). The **frexp** function returns the mantissa and stores the exponent in the integer object pointed to by **exp**. The mantissa is returned with a magnitude in the range $[1/2, 1]$ or zero, such that value equals the mantissa times 2^{exp} . If value is zero, both the exponent and mantissa are zero.

SEE ALSO

modf — Decomposing a double into mantissa and exponent.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    int exp;
    double mant;

    mantissa = frexp( 70.9, &exp );
    printf( "mantissa == %f\texponent == %d\n", mant, exp );
}
```

prints to standard output:

mantissa == 0.553906exponent == 7

NAME

isalnum — Test for alphanumeric character.

SYNOPSIS

```
#include <ctype.h>
int isalnum( int c );
```

DESCRIPTION

The **isalnum** function returns a nonzero value for any **alphabetic** or **numeric** character; zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isalnum( 'c' ) ) && ( isalnum( '1' ) ) )
    {
        printf( "c, 1 -- alpha and numeric\n" );
    }

    if ( ! ( isalnum( '@' ) ) )
    {
        printf( "@ -- neither alpha nor numeric\n" );
    }
}
```

prints to standard output:

c, 1 -- alpha and numeric

@ -- neither alpha nor numeric

NAME

isalpha — Test for alphabetic character.

SYNOPSIS

```
#include <ctype.h>
int isalpha( int c );
```

DESCRIPTION

The **isalpha** function returns a nonzero value for any **alphabetic** character; zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isalpha( 'c' ) ) )
    {
        printf( "c -- alpha\n" );
    }

    if ( ! ( isalpha( '@' ) ) && ! ( isalpha( '1' ) ) )
    {
        printf( "@, 1 -- non alpha\n" );
    }
}
```

prints to standard output:

c -- alpha

~,1 -- non alpha

NAME

isctrl — Test for control character.

SYNOPSIS

```
#include <ctype.h>
int isctrl( int c );
```

DESCRIPTION

The **isctrl** function returns a nonzero value for any **control** character; zero is returned in the false case. A control character is any character that is **NOT** a letter, digit, punctuation, or ' ' (the space character). This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    /* check the "beep" character */
    if ( ( isctrl( 0x07 ) ) )
    {
        printf( "\"beep\" (0x07) -- control character\n" );
    }

    if ( ! ( isctrl( '@' ) ) )
    {
        printf( "@ -- not control character\n" );
    }
}
```

prints to standard output:

```
"beep" (0x07) -- control character
@ -- not control character
```

NAME

isdigit — Test for numeric character.

SYNOPSIS

```
#include <ctype.h>
int isdigit( int c );
```

DESCRIPTION

The **isdigit** function returns a nonzero value for any **decimal** character; zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isdigit( '1' ) ) )
    {
        printf( "1 -- is a decimal character\n" );
    }

    if ( ! ( isdigit( 'f' ) ) )
    {
        printf( "f -- not a decimal character\n" );
    }
}
```

prints to standard output:

1 -- is a decimal character

f -- not a decimal character

NAME

isgraph — Test for printing character, excluding space and tab.

SYNOPSIS

```
#include <ctype.h>

int isgraph( int c );
```

DESCRIPTION

The **isgraph** function returns a nonzero value for any **printable** character, excluding space (' ') and tab ('\t'); zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    /* check the "beep" character */
    if ( ! ( isgraph( ' ' ) ) )
    {
        printf( "space -- not \"graph\" character\n" );
    }

    if ( ( isdigit( 'f' ) ) )
    {
        printf( "f -- \"graph\" character\n" );
    }
}
```

prints to standard output:

space -- not "graph" character

f -- "graph" character

NAME

islower — Test for lower-case alphabetic characters.

SYNOPSIS

```
#include <ctype.h>
int islower( int c );
```

DESCRIPTION

The **islower** function returns a nonzero value for any **lower-case alphabetic** character; zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( islower( 'a' ) ) )
    {
        printf( "a -- lower case character\n" );
    }

    if ( ! ( islower( 'F' ) ) )
    {
        printf( "F -- not a lower case character\n" );
    }
}
```

prints to standard output:

a -- lower case character

F -- not a lower case character

NAME

isprint — Test for printing character, excluding '\t'.

SYNOPSIS

```
#include <ctype.h>
int isprint( int c );
```

DESCRIPTION

The **isprint** function returns a nonzero value for any **printable** character, excluding the tab character ('\t'); zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isprint( ' ' ) ) )
    {
        printf( "space -- \"print\" character\n" );
    }

    if ( ! ( isprint( '\t' ) ) )
    {
        printf( "tab -- not \"print\" character\n" );
    }
}
```

prints to standard output:
space -- "print" character
tab -- not "print" character

NAME

ispunct — Test for punctuation character.

SYNOPSIS

```
#include <ctype.h>
int ispunct( int c );
```

DESCRIPTION

The **ispunct** function returns a nonzero value for any **punctuation** character; zero is returned in the false case. A punctuation character is one that is printable, not a digit, not a letter, and not a space (' ' or '\t'). This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( ispunct( ' ' ) ) )
    {
        printf( "space -- \"print\" character\n" );
    }

    if ( ! ( ispunct( '\t' ) ) )
    {
        printf( "tab -- not \"print\" character\n" );
    }
}
```

prints to standard output:

space -- "print" character

tab -- not "print" character

NAME

isspace — Test for white-space character.

SYNOPSIS

```
#include <ctype.h>
int isspace( int c );
```

DESCRIPTION

The **isspace** function returns a nonzero value for any standard **white-space** character; zero is returned in the false case. The standard white-space characters are space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isspace( ' ' ) ) )
    {
        printf( "space -- white-space character\n" );
    }

    if ( ! ( isspace( '@' ) ) )
    {
        printf( "@ -- not white-space character\n" );
    }
}
```

prints to standard output:

```
space -- white-space character
@ -- not white-space character
```

NAME

isupper — Test for upper-case alphabetic character.

SYNOPSIS

```
#include <ctype.h>

int isupper( int c );
```

DESCRIPTION

The **isupper** function returns a nonzero value for any **upper-case alphabetic** character; zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isupper( 'F' ) ) )
    {
        printf( "F -- upper-case character\n" );
    }

    if ( ! ( isupper( 'f' ) ) )
    {
        printf( "f -- not an upper-case character\n" );
    }
}
```

prints to standard output:

F -- upper-case character

f -- not an upper-case character

NAME

isxdigit — Test for hexadecimal numeric character.

SYNOPSIS

```
#include <ctype.h>
int isxdigit( int c );
```

DESCRIPTION

The **isxdigit** function returns a nonzero value for any **hexadecimal digit**; zero is returned in the false case. This function is provided both as an in-line and out-of-line function.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    if ( ( isupper( 'F' ) ) )
    {
        printf( "F -- hexadecimal character\n" );
    }

    if ( ! ( isupper( 'G' ) ) )
    {
        printf( "G -- not a hexadecimal character\n" );
    }
}
```

prints to standard output:

F -- hexadecimal character

G -- not a hexadecimal character

NAME

labs — Absolute value of a long integer.

SYNOPSIS

```
#include <stdlib.h>

long int labs( long int j );
```

DESCRIPTION

The **labs** function returns the absolute value of the long integer **j**.

SEE ALSO

abs — Absolute value of an integer.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    long int j = -19089709l;

    printf( "labs( -19089709l ) == %ld\n", labs( j ) );
}
```

prints to standard output:
labs(-19089709l) == 19089709

NAME

ldexp — Multiply floating point number by 2.

SYNOPSIS

```
#include <math.h>

double ldexp( double x, int exp );
```

DESCRIPTION

The **ldexp** function returns $x * 2^{\text{exp}}$. If the result exceeds **HUGE_VAL**, **errno** is set to **ERANGE** and the value **HUGE_VAL** is returned with the same sign as **x**.

SEE ALSO

exp — Raising e to a power.
pow — Raising a floating point number to a power.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "ldexp(7.09,4) == %f\n", ldexp(7.09,4) );
}
```

prints to standard output:
ldexp(7.09,4) == 113.440000

NAME

ldiv — Long integer division with remainder.

SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv( long int numer, long int denom );
```

DESCRIPTION

The **ldiv** function computes the quotient and remainder of **numer / denom** and returns the result in a structure of type **ldiv_t**. If the result cannot be represented, the result is undefined.

SEE ALSO

div — Integer division with remainder.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    ldiv_t result;
    long  numer = 709, denom = 56;

    result = ldiv( numer, denom );
    printf( "quotient == %ld\t", result.quot );
    printf( "remainder == %ld\n", result.rem);
}
```

prints to standard output:
quotient == 12remainder == 37

NAME

log — Natural logarithm, base e.

SYNOPSIS

```
#include <math.h>
double log( double x );
```

DESCRIPTION

The **log** function computes the natural logarithm of **x**. If the value of **x** is less than zero, **errno** is set to **EDOM** and the value **HUGE_VAL** is returned. If **x** is equal to zero, **errno** is set to **ERANGE** and the value **HUGE_VAL** is returned.

SEE ALSO

exp — Raising e to a power.
log10 — Base 10 logarithm.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "log( 7.09 ) == %f\n", log( 7.09 ) );
}
```

prints to standard output:
log(7.09) == 1.958685

NAME

log10 — Base ten logarithm.

SYNOPSIS

```
#include <math.h>

double log10( double x );
```

DESCRIPTION

The **log10** function computes the natural logarithm of **x**. If the value of **x** is less than zero, **errno** is set to **EDOM** and the value **HUGE_VAL** is returned. If **x** is equal to zero, **errno** is set to **ERANGE** and the value **HUGE_VAL** is returned.

SEE ALSO

exp — Raising e to a power.
log — Natural logarithm.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "log10( 7.09 ) == %f\n", log10( 7.09 ) );
}
```

prints to standard output:
log10(7.09) == 0.850646

NAME

longjmp — Execute a non-local jump.

SYNOPSIS

```
#include <setjmp.h>
void longjmp( jmp_buf env, int val );
```

DESCRIPTION

The **longjmp** function restores the calling environment referenced by **env** which must have been initialized by a previous call to **setjmp**. If there has been no invocation of **setjmp**, or if the function containing the call to **setjmp** has terminated execution before the call to **longjmp**, the behavior is undefined.

Upon completion of **longjmp**, program execution continues as if the corresponding call to **setjmp** had returned with a value **val**; if **val** is zero, 1 is returned.

Global and volatile-type variables have values as of the time **longjmp** was called; all **register** and non-volatile **automatic** variables will have indeterminate values.

For more information, see Chapter 6.

SEE ALSO

setjmp — Save a reference of the current calling environment for later use by longjmp.

EXAMPLE

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void func( void )
{
    longjmp( env, -709 );
}

main()
{
    if ( setjmp( env ) != 0 )
    {
        printf( "-- longjmp has been called --\n" );
    }
}
```

longjmp

```
        exit( 1 );
    }
    printf( "-- setjmp called --\n" );
    func();
}
```

prints to standard output:

```
-- setjmp called --
-- longjmp called --
```

longjmp

NAME

malloc — Dynamically allocate uninitialized storage.

SYNOPSIS

```
#include <stdlib.h>
void* malloc( size_t size );
```

DESCRIPTION

The **malloc** function returns a pointer to the lowest word of a **size** word block of storage; when **size** exceeds the amount of memory available, **malloc** returns NULL.

SEE ALSO

alloca — Allocate storage in the stack frame.
calloc — Dynamically allocate zero-initialized storage.
free — Free dynamically allocated memory.
realloc — Alter size of dynamically allocated storage.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char *char_array;

    if((char_array=(char*) malloc(709*sizeof(char))) == NULL)
    {
        printf( "error: not enough memory\n" );
        exit( 1 );
    }
    else
    {
        printf("-- space for 709 chars allocated OK --\n" );
    }
}
```

prints to standard output:
-- space for 709 chars allocated OK --

NAME

mblen — Length of a multibyte character.

SYNOPSIS

```
#include <stdlib.h>
```

```
int mblen( const char* s, size_t n );
```

DESCRIPTION

The **mblen** function determines the number of characters in the multibyte character pointed to by **s**. The **mblen** function is equivalent to

```
mbtowc( (wchar_t*) 0, s, n );
```

If **s** is a NULL pointer, **mblen** returns a zero. If **s** is not NULL, **mblen** returns

1. zero if **s** points to a NULL character,
2. the number of characters that comprise the multibyte character, or
3. -1 if an invalid multi-byte character is formed.

In no case will the return value exceed **n** or the **MB_CUR_MAX** macro.

SEE ALSO

mbtowc — Convert multibyte characters into wide characters.

wctomb — Convert wide characters into multibyte characters.

SPECIAL NOTE

The DSP96002 does not provide byte addressing, thus characters always require an entire word of memory each. One way to better utilize data memory (with a run-time cost) is to combine the ANSI data type **wchar_t** and the special ANSI **multibyte** and **wide character** library routines.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

char* gstr = NULL;

void main()
{
    int max = MB_CUR_MAX;
    char* strnull = gstr;
    char* str1 = "709";

    printf("mblen(strnull,5)==%d\n", mblen( strnull,5));
    printf("mblen(str1, max ) == %d\n", mblen(str1,max));
    printf("mblen(\"abcdef\",5) == %d\n", mblen("abcedf",5));
    printf("mblen(\"abcdef\",2) == %d\n", mblen("abcedf",2));
}
```

prints to standard output:

```
mblen( strnull, 5 ) == 3
mblen( str1, 5 ) == 3
mblen( "abcdef", 5 ) == 3
mblen( "abcdef", 2 ) == 2
```

NAME

mbstowcs— Convert multibyte string to wide character string.

SYNOPSIS

```
#include <stdlib.h>
```

```
int mbstowcs( wchar_t* pwcs, const char* s, size_t n );
```

DESCRIPTION

The **mbstowcs** function converts the character string pointed to by **s** into a wide character string pointed to by **pwcs**. Each character of the multibyte string is converted as if by the **mbtowc** function. At most, **n** characters will be converted and stored in the wide character string. Multibyte characters that follow a NULL character will not be examined or converted. If **s** and **pwcs** overlap, the behavior is undefined.

If an invalid character is encountered, **mbstowcs** returns (size_t) -1. Otherwise, **mbstowcs** returns the number of characters converted, not including the terminating NULL character.

SEE ALSO

wcstombs— Convert wide character strings into multibyte strings.

SPECIAL NOTE

The DSP96002 does not provide byte addressing, thus characters always require an entire word of memory each. One way to better utilize data memory (with a run-time cost) is to combine the ANSI data type **wchar_t** and the special ANSI **multibyte** and **wide character** library routines.

EXAMPLE

```

#include <stdio.h>
#include <stdlib.h>

wchar_t warray[10];

void main()
{
    char array = "abcdedgh";
    char* ptr = array;
    int convert;

    convert = mbstowcs( warray, array, 10 );

    printf( "unpacked array looks like:\n" );
    while ( *ptr != 0 )
    {
        printf( "%0.6x ", *ptr++ );
    }
    printf( "\n\n" );

    printf( "%d chars packed, packed array looks like:\n" );
    ptr = warray;
    while ( *ptr != 0 )
    {
        printf( "%0.6x ", *ptr++ );
    }
    printf( "\n" );
}

prints to standard output:
unpacked array looks like:
000061 000062 000063 000064 000065 000066 000067 000068

8 chars packed, packed array looks like:
616263 646966 676800

```

NAME

mbtowc — Convert a multibyte character to a wide character.

SYNOPSIS

```
#include <stdlib.h>
```

```
int mbtowc( wchar_t* pwc, const char* s, size_t n );
```

DESCRIPTION

The **mbtowc** function examines the multibyte (i.e., multi-character) string pointed to by **s** and converts it into a wide character (**wchar_t**). At most, **n** and never more than **MB_CUR_MAX** characters from **s** will be examined and converted.

If **s** is a NULL pointer, **mbtowc** returns zero. If **s** is not NULL, **mbtowc** returns

1. zero if **s** points to a NULL character,
2. the number of characters that comprise the multibyte character, or
3. -1 if an invalid multibyte character is formed.

In no case will the return value exceed **n** or the **MB_CUR_MAX** macro.

SEE ALSO

mblen — Determine the length of a multibyte character.
mbstowcs — Convert a multibyte string into a wide character string.
wctomb — Convert a wide character into a multibyte character.

SPECIAL NOTE

The DSP96002 does not provide byte addressing, thus characters always require an entire word of memory each. One way to better utilize data memory (with a run-time cost) is to combine the ANSI data type **wchar_t** and the special ANSI **multibyte** and **wide character** library routines.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    wchar_t wide = 0;
    char* mbstr = "abcde";
    int convert;

    convert = mbtowc( (wchar_t*) NULL, mbstr, 2 );
    printf("%d chars packed. wide == %0.6x\n", convert, wide);

    convert = mbtowc( &wide, mbstr, strlen( mbstr ) );
    printf("%d chars packed. wide == %0.6x\n", convert, wide);

    convert = mbtowc( &wide, mbstr, 2 );
    printf("%d chars packed. wide == %0.6x\n", convert, wide);
}
```

prints to standard output:

2 chars packed. wide == 000000

3 chars packed. wide == 616263

2 chars packed. wide == 006162

NAME

memchr — Find a character in a memory area.

SYNOPSIS

```
#include <string.h>

int memchr( const void* s, int c, size_t n );
```

DESCRIPTION

The **memchr** function finds the first occurrence of **c** (converted to an unsigned char) in the memory area pointed to by **s**. The terminating null character is considered to be part of the string. The **memchr** function returns a pointer to the located char or a NULL pointer if the character is not found.

SEE ALSO

- strchr** — Find the first occurrence of a character in a string.
- strcspn** — Compute the length of the prefix of a string not containing any characters contained in another string.
- strpbrk** — Find the first occurrence of a character from one string in another string.
- strrchr** — Find the last occurrence of a character in a string.
- strspn** — Compute the length of the prefix of a string contained in another string.

SPECIAL NOTE

Due to the word orientation of the **DSP96002**, there is no way to accurately implement an ANSI version of **memchr** for the **L** memory model.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* string = "fred flintstone driving on bald feet";
    char* result;

    /* locate the occurrence of 'b' */
    result = memchr( string, 'b', strlen( string ) );
    printf( "-- %s --\n", result );
}
```

prints to standard output:

-- bald feet --

NAME

memcmp — Compare portion of two memory areas.

SYNOPSIS

```
#include <string.h>

int memcmp( const void* s1, const void* s2, size_t n );
```

DESCRIPTION

The **memcmp** function compares the first **n** words of the object pointed to by **s1** with the first **n** words of the object pointed to by **s2**, comparison is lexicographical. The **memcmp** function returns zero if the two areas compared are equal, a value greater than zero if **s1** is greater, or a value less than zero if **s1** is smaller.

SEE ALSO

strncmp — Compare portion of two strings.

SPECIAL NOTE

Due to the word orientation of the **DSP96002**, there is no way to accurately implement an ANSI version of **memcmp** for the **L** memory model.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

struct test
{
    char cartoon[20];
    int value;
} g1 = { "flintstones", 709 },
  g2 = { "flintstones", 709 },
  g3 = { "jetsons", 709 };

void main()
{
    if ( memcmp( &g1, &g2, sizeof( struct test ) ) != 0 )
    {
        printf( "error: flintstones differ\n" );
    }
    else
    {
        printf( "-- flintstones are flintstones --\n" );
    }

    if ( memcmp( &g1, &g3, sizeof( struct test ) ) != 0 )
    {
        printf( "-- flintstones are not jetsons --\n" );
    }
    else
    {
        printf( "error: flintstones are NOT jetsons\n" );
    }
}

prints to standard output:
-- flintstones are flintstones --
-- flintstones are not jetsons --
```

NAME

memcpy — Copy from one area to another.

SYNOPSIS

```
#include <string.h>

int memcpy( void* s1, const void* s2, size_t n );
```

DESCRIPTION

The **memcpy** function copies **n** words from the area referenced by **s2** into the area specified by **s1**. If the source and destination areas overlap, the results are undefined. The **memcpy** function returns the value of **s1**.

SEE ALSO

strcpy — Copy one string to another.
strncpy — Copy a portion of one string to another.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

struct test
{
    char cartoon[20];
    int value;
} g1, g2 = { "flintstones", 709 };

void main()
{
    memcpy( &g1, &g2, sizeof( struct test ) );
    printf( "-- I watch the %s --\n", g1.cartoon );
}

prints to standard output:
-- I watch the flintstones --
```

NAME

memmove — Copy storage.

SYNOPSIS

```
#include <string.h>

int memmove( void* s1, const void* s2, size_t n );
```

DESCRIPTION

The **memmove** function copies **n** words from the area referenced by **s2** into the area specified by **s1**. The copy is done by first placing the **n** words into a temporary buffer and then moving the temporary buffer into the final location, this allows the source and destination areas to overlap.

SEE ALSO

memcpy — Copy one memory area to another.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

struct test
{
    char cartoon[20];
    int value;
} g1, g2 = {"flintstones", 709 };

void main()
{
    memmove( &g1, &g2, sizeof( struct test ) );
    printf( "-- I watch the %s --\n", g1.cartoon );
}

prints to standard output:
-- I watch the flintstones --
```

NAME

memset — Initialize memory area.

SYNOPSIS

```
#include <string.h>
int memset( void* s, int c, size_t n );
```

DESCRIPTION

The **memset** function copies the value **c** (converted to an **unsigned char**) into the first **n** words of the object pointed to by **s**.

SEE ALSO

memcpy — Copy one memory area to another.

SPECIAL NOTE

Due to the word orientation of the **DSP96002**, the **L** memory model version of **memset** may not act as expected when attempting to initialize **double** size elements (the **X** memory portion is unaffected).

EXAMPLE

```
#include <stdio.h>
#include <string.h>

struct test
{
    char  cartoon[20];
    int value;
}

void main()
{
    struct test local;

    /* auto struct local is initialized to all nines */
    memset( &local, 9, sizeof( struct test ) );

    /* random check */
    if ( local.cartoon[7] != 9 )
    {
        printf( "error: memset busted\n" );
    }
    else
    {
        printf( "-- memset OK --\n" );
    }
}

prints to standard output:
-- memset OK --
```

NAME

modf — Break a **double** into it's integral and fractional parts.

SYNOPSIS

```
#include <math.h>

double modf( double value, double* iptr );
```

DESCRIPTION

The **modf** function brakes **value** into its fractional and integral parts. The **modf** function returns the fractional portion of **value** and stores the integral portion in the double object pointed to by **iptr**.

SEE ALSO

frexp — Break a double into its mantissa and exponent

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double result;

    printf( "-- fractional == %f\t", modf( 7.09, &result );
    printf( "integral == %f --\n", result );
}
```

prints to standard output:

```
-- fractional == 0.090000integral == 7.000000 --
```


NAME

perror — Print error message.

SYNOPSIS

```
#include <stdio.h>
void perror( const char* s );
```

DESCRIPTION

The **perror** function prints out the string **s** followed by “: ” and the error message associated with **errno**.

SEE ALSO

strerror — Print out error message associated with **errno**.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double result;

    result = asin( 7.09 );

    if ( result == 0.0 && errno == EDOM )
    {
        perror( "asin perror test" );
    }
}
```

prints to standard output:
asin perror test:domain error

NAME

pow — Raise a double to a power.

SYNOPSIS

```
#include <math.h>

double pow( double x, double y );
```

DESCRIPTION

The **pow** function computes and returns x^y . If **x** is zero and **y** is less than zero, a domain error occurs setting **errno** to **EDOM** and returning 0.0. If $|x^y|$ is greater than **HUGE_VAL**, **errno** is set to **ERANGE** and **HUGE_VAL** is returned.

SEE ALSO

exp — Raising e to a power.
ldexp — Multiplying a number by a power of 2.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "-- pow( 2.0, 2.0 ) == %f --\n", pow( 2.0, 2.0 ) );
}
```

prints to standard output:

```
-- pow( 2.0, 2.0 ) == 4.000000 --
```

NAME

printf — Print to standard output.

SYNOPSIS

```
#include <stdio.h>

int printf( const char* format, ... );
```

DESCRIPTION

The **printf** function formats and writes a string to the standard output. Interpreting the format specifier **format** left to right.

The format specifier, **format**, consists of ordinary characters, escape sequences, and conversion specifications. The conversion specifications describe how arguments passed to **printf** are converted for output. All non-conversion specifying portions of **format** are sent directly to the standard output. If the number of arguments passed is less than specified by the **format** string, **printf** will write non-deterministic garbage to the standard output. If too many arguments are provided to **printf**, the extras will be evaluated but otherwise ignored.

A conversion specification is introduced by the character **%**, and has the following form:

%[flags][field width][.precision][size]**conversion character**

where flags, field width, precision, h, l, L are optional.

Flags are for justification of output and printing of signs, blanks, decimal points, octal and hexadecimal prefixes. Multiple flags may be utilized at once. The ANSI flags are:

- Left justify the result within the field. The default is right justified.
- + The result of a signed conversion will always have a sign (+ or -). The default case provides only for -.
- space** If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space character will be prefixed to the result. If the **space** and the **+** flags both appear, the **space** flag is ignored. The default mode is no **space**.
- #** The result is converted to an alternate form specified by the conversion character. For **o** conversion, it forces the first digit of the result to be a zero. For **x** (or **X**) conversion, the non-zero result will have 0x (0X) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal point character, even if no digits follow it. Additionally for **g** and **G**, trailing zeros will not be removed.

- 0** For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the **field width**; no space padding is performed. If the **0** and **-** flags both appear, the **0** flag will be ignored.

Each conversion takes place in character fields. The minimum size of the field can be specified with the field width. If the converted result contains fewer characters than specified by field width, the result will be left padded with spaces by default (see flags above). The field width takes the form of a decimal integer or an asterisk **"*"**. When the field width is an asterisk, the value is to be taken from an integer argument that precedes the argument to be converted.

Precision specifies the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, **X** conversions, the number of digits appear after the decimal point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in the **s** conversion. **The precision takes the form of a '.' followed by "*", or by an optional decimal integer; if only the period is specified, the precision is taken to be zero.** If precision appears with any other conversion character, the behavior is undefined.

Size specifies the argument size expected. There are three size specifiers defined by ANSI. The **h** specifies that the argument for the conversion characters **d**, **i**, **o**, **u**, **x**, or **X** will be unsigned short. The **l** specifies that the argument for the conversion characters **d**, **i**, **o**, **u**, **x**, or **X** will be long integer. The **L** specifies that the argument for the conversion characters **e**, **E**, **f**, **g**, or **G** will be long double.

There are 16 conversion characters; each is described below.

- d, i** The **int** argument is printed as a signed decimal number. The precision specifies the minimum number of digits to appear; if the value being printed can be represented in fewer digits, it is expanded with leading zeros. The default **precision** is 1. The result of printing a zero with **precision** zero is no characters (this is independent of padding specified by **field width**).
- o** The **unsigned int** argument is printed as an unsigned octal number. When used in association with the **#** flag, 0 will be prefixed to non-zero results. The **precision** specifies the minimum number of digits to appear; if the value can be represented in fewer digits, it will be expanded with leading zeros. The default **precision** is 1. The result of printing a zero with **precision** zero is no characters (this is independent of padding specified by **field width**).

- u** The **unsigned int** argument is printed as an unsigned decimal number. The **precision** specifies the minimum number of digits to appear; if the value can be represented in fewer digits, it will be expanded with leading zeros. The default **precision** is 1. The result of printing a zero with **precision** zero is no characters (this is independent of padding specified by **field width**).
- x, X** The **unsigned int** argument is printed as an unsigned hexadecimal number. Hexadecimal alpha characters (a,b,c,d,e,f) will be printed in lower case when **x** is used and in upper case when **X** is used. When used in association with the **#** flag, 0x will be prefixed to the result (0X in the **X** case). **Precision** specifies the minimum number of digits to appear; if the value can be represented in fewer digits, it will be expanded with leading zeros. The default **precision** is 1. The result of printing a zero with **precision** zero is no characters (this is independent of padding specified by **field width**).
- f** The **double** argument is printed out in decimal notation of the form [-]ddd.ddd, where **precision** specifies the number of digits to follow the decimal point. The default **precision** 6. When **precision** is 0 and the **#** flag is not specified, no decimal point character will be printed. A decimal digit will always follow at least one digit. The value printed is rounded to the appropriate number of digits.
- e, E** The **double** argument is printed out in the form [-] d.ddde±dd, where **precision** specifies the number of digits to follow the decimal point. The default **precision** 6. When **precision** is 0 and the **#** flag is not specified, no decimal point character will be printed. A decimal digit will always follow at least one digit. The exponent always contains at least two digits.
- g, G** The **double** argument is printed in the **f**, **e**, or **E** form. The **f** form is used unless the exponent to be printed is less than **-4** or greater than the **precision**. If precision is zero, the printed value consists of no characters (this is independent of padding specified by **field width**). Trailing zeros are removed from the fractional portion of the result; a decimal point character is printed only if it is followed by a digit.
- c** The **int** argument is printed as an unsigned character.

- s** The argument is a pointer to a character string (**(char*)**). Characters from the string are printed up to (but not including) a terminating null character or until **precision** characters have been printed. If **precision** is not explicitly specified or is greater than the length of the string, the string will be printed until the null character is encountered.
- p** The argument is a pointer to the void data type (**(void*)**). The value of the of the pointer is printed out as a hexadecimal digit.
- n** The argument is a pointer to an integer (**(int*)**) which is the number of characters printed so far by the current call to **printf**.
- %** Print the percent character, **%**. Note that the complete specifier is **%%**.

On successful completion, **printf** returns the an integer equal to the number of characters printed. On failure, **printf** returns an integer less than 0.

SEE ALSO

- scanf** — Read values from standard input.
- sscanf** — Read values from a string.
- sprintf** — Multiplying a number by a power of 2.

EXAMPLE

```
#include <stdio.h>

char* lib_name = "printf";

void main()
{
    int i = 709;
    double d = 7.09;

    printf("Show several %s examples\n", lib_name );
    printf("\tintegers:\n" );
    printf("\t\ttoctal == %o\n", i );
    printf("\t\ttoctal == %#.9o ", i );
    printf("(force leading 0 and zero pad)\n");
    printf("\t\tdecimal == %d\n", i );
    printf("\t\tdecimal == % d (force leading blank)\n", i );
    printf("\t\tthe x == %x\n", i );
    printf("\t\tthe x == %#X (force leading 0X)\n", i );
    printf("\tfloating point:\n" );
    printf("\t\tdouble == %f\n", d );
    printf("\t\tdouble == %e\n", d );
}
```

prints to standard output:

Show several printf examples

integers:

octal == 1305

octal == 000001305 (force leading 0 and zero pad)

decimal == 709

decimal == 709 (force leading blank)

hex == 2c5

hex == 0X2C5 (force leading 0X)

floating point:

```
double == 7.090000
```

```
double == 7.090000e+00
```

NAME

putchar — Write a character to standard output.

SYNOPSIS

```
#include <stdio.h>

int putchar( int c );
```

DESCRIPTION

The **putchar** function prints a character to standard output.

SEE ALSO

getchar — Get a character from standard input.
gets — Get a line of text from standard input.
puts — Write a string to standard output.

EXAMPLE

```
#include <stdio.h>

char* str = "bald feet\n";

void main()
{
    while ( *str != '\0' )
    {
        putchar ( *str++ );
    }
}
```

prints to standard output:
bald feet

NAME

puts — Write a string to standard output.

SYNOPSIS

```
#include <stdio.h>

int puts( const char* s );
```

DESCRIPTION

The **puts** function prints a string to standard output, appending a newline character. The **puts** function returns a zero if operation is successful and a non-zero value on failure.

SEE ALSO

getchar — Get a character from standard input.
gets — Get a line of text from standard input.
putchar — Write a char to standard output.

EXAMPLE

```
#include <stdio.h>
```

```
char* str = "bald feet";
```

```
void main()
{
    puts ( str );
}
```

prints to standard output:
bald feet

NAME

qsort — Quick sort.

SYNOPSIS

```
#include <stdlib.h>

void qsort( void* base, size_t nmemb, size_t size,
            int (*compar) (const void*, const void* ) );
```

DESCRIPTION

The **qsort** function sorts an array of **nmemb** objects of size **size**, pointed to by **base**.

The array is sorted in ascending order according to a comparison function pointed to by **compar** which is called with two pointers to the array members. The **compar** function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second argument.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* stuff[] = {
    "fred", "flintstone", "driving", "bald", "on", "feet"
};

static int compare( const char** a1, const char** a2 )
{
    return( strcmp( *a1, *a2 ) );
}

main()
{
    int i;

    qsort(stuff, (size_t)6, (size_t)sizeof(char*), compare);

    for ( i = 0 ; i < 6 ; i++ )
    {
        printf( "%s\t", stuff[i] );
    }

    printf( "\n" );
}
```

prints to standard output:

bald driving feet flintstone fred on

NAME

raise — Raise a signal.

SYNOPSIS

```
#include <signal.h>

int raise( int sig );
```

DESCRIPTION

The **raise** function sends the signal **sig** to the executing program and returns 0 if successful or non-zero if unsuccessful. See **signal.h** for list of available signals and their default actions.

For more information, see Chapter 6.

SEE ALSO

signal — Set up a signal handler.

EXAMPLE

```
#include <stdio.h>

void main()
{
    int onintr();

    signal (SIGINT, onintr);
    raise( SIGINT );
}

onintr()
{
    printf( "caught SIGINT, see ya ...\n" );
    exit( -9 );
}
```

prints to standard output:
caught SIGINT, see ya ...

NAME

rand — Pseudo- random number generator.

SYNOPSIS

```
#include <stdlib.h>

int rand( void );
```

DESCRIPTION

The **rand** function computes and returns a sequence of pseudo-random integers in the range of 0 to 32767.

SEE ALSO

srand — Seed the pseudo-random number generator.

EXAMPLE

```
#include <stdio.h>

void main()
{
    /* seed the random number sequence */
    srand( 1638 );

    /* spew out random numbers in the range 0 to 709 */
    for ( ;; )
    {
        printf( "%d", ( rand() ) % 709 );
    }
}
```

prints to standard output:

545 174 307 657 220 267 66 47 651 268 284 338 35 615 403 etc.

NAME

realloc — Change size of dynamically allocated storage area.

SYNOPSIS

```
#include <stdlib.h>

int realloc( void* ptr, size_t size );
```

DESCRIPTION

The **realloc** function changes the size of the storage area pointed to by **ptr** to **size**. The contents of the storage area are unchanged. If the new storage area is larger, the value of the new area is indeterminate. If **ptr** is null, **realloc** acts like **malloc**. If **ptr** was not dynamically allocated or the area was previously deallocated by a call to **free**, the behavior is undefined.

If **realloc** is unable to allocate the new size storage area, a null pointer is returned and the original storage area is unchanged.

SEE ALSO

alloca	—	Allocate storage in the stack frame.
calloc	—	Dynamically allocate zero-initialized storage.
free	—	Free dynamically allocated storage.
malloc	—	Dynamically allocate uninitialized storage.

EXAMPLE

```
#include <stdio.h>

void main()
{
    char* str;

    if ( ( str = (char*) malloc( (size_t) 15 ) ) == NULL )
    {
        error( "malloc failed" );
        exit (-8);
    }

    strcpy( str, "short string" );
    printf( "%s\n", string );

    /* allocate space for 40 character string */
    if ((str = (char*)realloc(str, 40*sizeof(char)) )== NULL )
    {
        perror( "realloc test" );
        exit ( -9 );
    }

    strcat( str, " becomes a long string" );
    printf( "%s\n", str );
}
```

prints to standard output:

short string

short string becomes a long string

NAME

setjmp — Save a reference of the current calling environment for later use by **longjmp**.

SYNOPSIS

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

DESCRIPTION

The **setjmp** function saves its calling environment in **env** for later use by **longjmp**. If the return is direct from **setjmp**, the value zero is returned. If the return is from the **longjmp** function, the value returned is non-zero.

For more information, see Chapter 6.

SEE ALSO

longjmp — Execute a non-local jump.

EXAMPLE

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void func( void )
{
    longjmp( env, -709 );
}

void main()
{
    if ( setjmp( env ) != 0 )
    {
        printf( "-- longjmp has been called --\n" );
        exit( 1 );
    }
    printf( "-- setjmp called --\n" );
    func();
}
```

prints to standard output:

```
-- setjmp called --
-- longjmp called --
```

NAME

signal — Set up signal handler.

SYNOPSIS

```
#include <setjmp.h>
void (*signal( int sig, void (*func)(int) ) ) (int);
```

DESCRIPTION

The **signal** function chooses one of three ways in which to handle the receipt of signal **sig**.

1. If the value of **func** is the macro **SIG_DFL**, default handling for the signal will occur.
2. If the value of **func** is the macro **SIG_IGN**, the signal is ignored.
3. Otherwise, **func** is a pointer to a function that will be called on the receipt of signal **sig**.

When a signal occurs, the signal handler for **sig** is reset to **SIG_DFL**; this is equivalent to making the call **signal(sig, SIG_DFL)**. The function **func** terminates by executing the **return** statement or by calling the **abort**, **exit**, or **longjmp** function. If the function **func** terminates with a **return** statement, execution continues at the point the signal was caught. Note that if the value of **sig** was **SIGFPE**, the behavior is undefined.

Also note that in order to continue catching signal **sig**, the signal handler must reissue the signal call.

For more information, see Chapter 6.

SEE ALSO

raise — Raise a signal.

EXAMPLE

```
#include <stdio.h>

void main()
{
    int onintr();

    signal (SIGINT, onintr);
    raise( SIGINT );
}

onintr()
{
    printf( "caught SIGINT, see ya ...\n" );
    exit( -9 );
}
```

prints to standard output:
caught SIGINT, see ya ...

NAME

sin — Sine.

SYNOPSIS

```
#include <math.h>
double sin( double x );
```

DESCRIPTION

The **sin** function computes and returns the sine of **x**, measured in radians.

SEE ALSO

asin — The arc sine of an angle.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "sin( 45.0 ) == %f\n", sin( 45.0 ) );
}

prints to standard output:
sin( 45.0 ) == 0.850904
```

NAME

sinh — Hyperbolic Sine.

SYNOPSIS

```
#include <math.h>

double sinh( double x );
```

DESCRIPTION

The **sinh** function computes and returns the hyperbolic sine of **x**, measured in radians. When the value of **x** is too large, **errno** will be set to **ERANGE** and the return value will be **HUGE_VAL** with the sign of **x**.

SEE ALSO

cosh — Hyperbolic cosine of an angle.
tanh — Hyperbolic tangent of an angle.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "sinh( 3.1415 ) == %f\n", sinh( 3.1415 ) );
}
```

prints to standard output:
sinh(3.1415) == 11.547661

NAME

sprintf — Print to a string.

SYNOPSIS

```
#include <stdio.h>
```

```
int sprintf( char* s, const char* format, ... );
```

DESCRIPTION

The **sprintf** function is equivalent to **printf** except that **s** specifies a string that the generated output is printed to rather than standard output. A null character is written at the end of the string. The **sprintf** function returns the number of characters written to the string.

SEE ALSO

printf — Print to a standard output.

EXAMPLE

```
#include <stdio.h>

void main()
{
    char buffer[296];
    char* bptr = buffer;
    char* str = "strings";
    int i = 709, count;
    double d = 7.09;

    bptr += sprintf( bptr,"testing sprintf with:\n" );
    sprintf( bptr, "\tstrings\t(%s)\n%n", str, &count );
    bptr += count;
    bptr += sprintf( bptr, "\thex digits\t%x\n", i );
    bptr += sprintf( bptr, "\tfloating point\t%f\n", d );

    puts( buffer );
}
```

prints to standard output:

testing sprintf with:

strings(strings)

hex digits2c5

floating point7.090000

NAME

sqrt — Square root.

SYNOPSIS

```
#include <math.h>
double sqrt( double x );
```

DESCRIPTION

The **sqrt** function computes and returns the nonnegative square root of **x**. If **x** is less than zero, **errno** is set to **EDOM** and 0.0 is returned.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    double = 50.2681;

    printf( "sqrt( 50.2681 ) == %.2f\n", sqrt( d ) );
}
```

prints to standard output:

```
sqrt( 50.2681 ) == 7.09
```


NAME

srand — Seed the pseudo-random number generator.

SYNOPSIS

```
#include <stdlib.h>

void srand( unsigned int seed );
```

DESCRIPTION

The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by **rand**. When **srand** is called with the same argument, the sequence of pseudo-random numbers will be repeated. If **srand** is not called, the default seed is 1.

SEE ALSO

rand — Generate a pseudo-random number sequence.

EXAMPLE

```
#include <stdio.h>

void main()
{
    /* seed the random number sequence */
    srand( 1638 );

    /* spew out random numbers in the range 0 to 709 */
    for ( ;; )
    {
        printf( "%d", ( rand() ) % 709 );

    }
}
```

prints to standard output:

545 174 307 657 220 267 66 47 651 268 284 338 35 615 403 etc.

NAME

strcat — Concatenate two strings.

SYNOPSIS

```
#include <string.h>

char* strcat( char* s1, const char* s2 );
```

DESCRIPTION

The **strcat** function appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The first character of the second string is written over the first string's terminating character. The **strcat** function returns a pointer to **s1**.

SEE ALSO

strncat — Concatenate n characters from one string to another.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char bigstr[80] = "string 1";
    char smallstr[20] = " string 2";

    printf("concatenate (%s) and (%s)\n", bigstr, smallstr);
    (void) strcat( bigstr, smallstr );
    puts( bigstr );
}
```

prints to standard output:
concatenate (string 1) and (string 2)
string 1 string 2

NAME

strchr — Find first occurrence of a character in a string.

SYNOPSIS

```
#include <string.h>

char* strchr( const char* s, int c );
```

DESCRIPTION

The **strchr** function locates the first occurrence of **c** (converted to a char) in the string pointed to by **s**. The terminating null character is considered part of the string. The **strchr** function returns a pointer to the located character or a null pointer if the character is not found in the string.

SEE ALSO

memchr — Find a character in a memory area.
strcspn — Compute the length of the prefix of a string not containing any characters contained in another string.
strpbrk — Find the first occurrence of a character from one string in another string.
strrchr — Find the last occurrence of a character in a string.
strspn — Compute the length of the prefix of a string contained in another string.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* string = "fred flintstone driving on bald feet";
    char* found;

    found = strchr( string, 'b' );
    puts( found );
}
```

prints to standard output:
bald feet

NAME

strcmp — Compare two strings.

SYNOPSIS

```
#include <string.h>

int strcmp( const char* s1, const char* s2 );
```

DESCRIPTION

The **strcmp** function compares the string pointed to by **s1** to the string pointed to by **s2**. If string **s1** is lexicographically greater than, equal to, or less than **s2**; an integer respectively greater than, equal to, or less than zero will be returned. The comparison of two strings of unequal length in which the longer string contains the smaller string yields the results that the longer string compares greater than.

i.e. `strcmp("xxx", "xxxxyz") < 0` **or** `strcmp("xxxxyz", "xxx") > 0`

When the header file `string.h` is included, the default case will be in-line [see section A.3, Forcing Library Routines Out-of-line].

SEE ALSO

memcmp — Compare two memory areas.
strcoll — Compare two strings based on current locale.
strncmp — Compare portions of two strings.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    if ( strcmp( "xxx", "xxxxyz" ) < 0 )
    {
        puts( "xxx is less than xxxyz" );
    }
    else
    {
        puts( "xxx is greater than xxxyz" );
    }

    if ( strcmp( "xxxxyz", "xxx" ) < 0 )
    {
        puts( "xxxxyz is less than xxx" );
    }
    else
    {
        puts( "xxxxyz is greater than xxx" );
    }

    if ( strcmp( "xxxxyz", "xxxxyz" ) == 0 )
    {
        puts( "xxxxyz is equal to xxxyz" );
    }
}
```

prints to standard output:

xxx is less than xxxyz

xxxxyz is greater than xxx

xxxxyz is equal to xxxyz

NAME

strcoll — Compare two strings based on current locale.

SYNOPSIS

```
#include <string.h>

int strcoll( const char* s1, const char* s2 );
```

DESCRIPTION

The **strcoll** function compares the string pointed to by **s1** to the string pointed to by **s2**, both strings are interpreted using the **LC_COLLATE** category of the current locale. If string **s1** is lexicographically greater than, equal to, or less than **s2**, an integer greater than, equal to, or less than zero will be returned. The comparison of two strings of unequal length in which the longer string contains the smaller string yields the result that the longer string compares greater than.

For **DSP96KCC**, **strcoll** functions exactly like **strcmp**.

SEE ALSO

strxfrm — Transform a string into locale-independent form.
strcmp — Compare two strings.

NAME

strcpy — Copy one string into another.

SYNOPSIS

```
#include <string.h>

int strcpy( char* s1, const char* s2 );
```

DESCRIPTION

The **strcpy** function copies the characters of string **s2**, including the terminating character, into the string pointed to by **s1**. If the strings overlap, the behavior is undefined. The value of **s1** is returned.

When the header file **string.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

SEE ALSO

memcpy — Copy one memory area to another.
memset — Initialize a memory area.
strncpy — Copy a portion of one string to another.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char string[80];
    char* sptr;

    strcpy( string, "-- no bald feet for george jetson --" );
    puts( sptr );
}
```

prints to standard output:
-- no bald feet for george jetson --

NAME

strcspn — Compute the length of the prefix of one string consisting entirely of characters from another.

SYNOPSIS

```
#include <string.h>

int strcspn( const char* s1, const char* s2 );
```

DESCRIPTION

The **strcspn** function computes and returns the length of the prefix of the string pointed to by **s1** that consists entirely of characters not found in the string pointed to by **s2**.

SEE ALSO

memchr — Find first occurrence of a character in a memory area.
strchr — Find first occurrence of a character in a string.
strpbrk — Find first occurrence of any character from one string in another string.
strrchr — Find last occurrence of a character in a string.
strspn — Compute the length of the prefix of one string that consists only of characters from another string.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    int i;

    i = strcspn( "azbyfghjki", "fkjeughtrg" );
    printf( "-- prefix length == %d --\n", i );
}
```

prints to standard output:

-- prefix length == 4 --

NAME

strerror — Map error code into an error message string.

SYNOPSIS

```
#include <string.h>
char* strerror( int errnum );
```

DESCRIPTION

The **strerror** function maps **errnum** to an error message string. A pointer to the string is returned. The string returned **should not be modified** by the programmer.

SEE ALSO

perror — Print error message.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    int i = 0;
    char* s;

    while ( ( s = strerror( i ++ ) ) != NULL )
    {
        printf( "\tmessage %d:\t%s\n", i, s );
    }
}
```

prints to standard output:

```
message 1:domain error
message 2:range error
message 3:out of heap memory
message 4:bad format for conversion string
```

NAME

strlen — Determine length of a string.

SYNOPSIS

```
#include <string.h>
size_t strlen( const char* s );
```

DESCRIPTION

The **strlen** function computes and returns the number of characters **proceeding** the terminating character.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* s = "is your name michael diamond?";

    printf( "strlen( \"%s\" ) == %d\n", s, strlen( s ) );
}
```

prints to standard output:

```
strlen( "is your name michael diamond?" ) == 29
```

NAME

strncat — Concatenate a portion of one string to another.

SYNOPSIS

```
#include <string.h>

char* strncat( char* s1, const char* s2, size_t n );
```

DESCRIPTION

The **strncat** function appends, at most, **n** characters from the string pointed by **s2** to the end of the string pointed to by **s1**. The first character of the second string is written over the first strings terminating character and a new terminating character is appended. The **strncat** function returns a pointer to **s1**. If **s1** does not have **n** words allocated past the terminating character, the behavior is undefined.

SEE ALSO

strcat — Concatenate one string to another.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char bstr[80] = "string 1";
    char sstr[20] = " string 2";

    printf("paste 5 chars of (%s) on to (%s)\n", sstr, bstr);
    (void) strncat( bstr, sstr, 5 );
    puts( bigstr );
}
```

prints to standard output:
paste 5 chars of (string 2) on to (string 1)
string 1 stri

NAME

strncmp — Compare a portions of two strings.

SYNOPSIS

```
#include <string.h>
```

```
int strncmp( const char* s1, const char* s2, size_t n );
```

DESCRIPTION

The **strncmp** function compares **n** characters of the string pointed to by **s2** with the string pointed to by **s1**. If string **s1** is lexicographically greater than, equal to, or less than **s2**; an integer respectively greater than, equal to, or less than zero will be returned. This is similar to **strcmp**.

SEE ALSO

strcmp — Compare two strings.

EXAMPLE

```
#include <stdio.h>
```

```
#include <string.h>
```

```
void main()
```

```
{
```

```
    char bigstr[80] = "string 1";
```

```
    char smallstr[20] = "string 2";
```

```
    if ( strncmp( bigstr, smallstr, 5 ) == 0 )
```

```
    {
```

```
        printf( "-- strncmp ok --\n" );
```

```
    }
```

```
    else
```

```
    {
```

```
        printf( "?? strncmp error ??\n" );
```

```
    }
```

```
}
```

prints to standard output:

```
-- strncmp ok --
```

NAME

strncpy — Copy a portion of one string into another.

SYNOPSIS

```
#include <string.h>

char* strncpy( char* s1, const char* s2, size_t n );
```

DESCRIPTION

The **strncpy** function copies exactly **n** characters from a string pointed to by **s2** into a string pointed to by **s1**. If **strlen(s2)** is less than **n**, the string **s1** is null padded. If **strlen(s2)** is greater than or equal to **n**, no null termination character is copied to **s1**. The **s1** pointer is returned.

Note that the behavior of non null terminated strings is undefined.

SEE ALSO

memcpy — Copy one memory area to another.
strcpy — Copy one string to another.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char bigstr[80] = "string 1";
    char smallstr[20] = "spanky 2";

    ( void ) strncpy( bigstr, smallstr, 6 );
    puts( bigstr );
}
```

prints to standard output:
spanky 1

NAME

strpbrk — Find the first occurrence of a character from one string in another.

SYNOPSIS

```
#include <string.h>
char* strpbrk( char* s1, const char* s2 );
```

DESCRIPTION

The **strpbrk** function finds the first occurrence of **any** character in the string pointed to by **s2** in the string pointed to by **s1**. If a character is found, a pointer to the character is returned. If a character is not found, a null pointer is returned.

SEE ALSO

memchr — Find a character in a memory area.
strchr — Find the first occurrence of a character in a string.
strcspn — Compute the length of the prefix of a string not containing any characters contained in another string.
strrchr — Find the last occurrence of a character in a string.
strspn — Compute the length of the prefix of a string contained in another string.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* string = "abcde random characters fghijkl";
    char* fndstr = "klmnopqr";
    char* found;

    if ( ( found = strpbrk( string, fndstr ) ) != NULL )
    {
        puts( found );
    }

    else
    {
        puts( "can't find a character" );
    }
}
```

prints to standard output:
random characters fghijkl

NAME

strchr — Find the last occurrence of a character from one string in another.

SYNOPSIS

```
#include <string.h>

char* strchr( char* s1, const char* s2 );
```

DESCRIPTION

The **strchr** function locates the last occurrence of **c** (converted to a char) in the string pointed to by **s**. The terminating null character is considered part of the string. **strchr** returns a pointer to the located character, or a null pointer if the character is not found in the string.

SEE ALSO

memchr — Find a character in a memory area.
strchr — Find the first occurrence of a character in a string.
strcspn — Compute the length of the prefix of a string not containing any characters contained in another string.
strpbrk — Find the first occurrence of a character from one string in another string.
strspn — Compute the length of the prefix of a string contained in another string.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* string = "fred flintstone driving on bald feet";
    char* found;

    found = strchr( string, 'f' );
    puts( found );
}

prints to standard output:
feet
```


NAME

strstr — Find the first occurrence of one string in another.

SYNOPSIS

```
#include <string.h>

char* strstr( const char* s1, const char* s2 );
```

DESCRIPTION

The **strstr** function locates the first occurrence of the string pointed to by **s2** (excluding the termination character) in the string pointed to by **s1**. If the string **s2** is found, a pointer to it is returned. If the string **s2** is not found, a null pointer is returned. If **s2** has a length of zero, a pointer to **s1** is returned.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* string = "abcdef random characters ghijkl";
    char* fndstr = "random";
    char* found;

    if ( ( found = strstr( string, fndstr ) ) != NULL )
    {
        puts( found );
    }
    else
    {
        puts( "can't find the string" );
    }
}
```

prints to standard output:
random characters ghijkl

NAME

strtod — String to double.

SYNOPSIS

```
#include <stdlib.h>
```

```
double strtod( const char* nptr, char** endptr );
```

DESCRIPTION

The **strtod** function converts and returns the string pointed to by **nptr** to floating point number. First **strtod** decomposes **nptr** into three sections;

1. an initial, possibly empty, sequence of white space characters,
2. a subject in the form of a floating point constant; and
3. a final string of one or more unrecognized characters, including the terminating null character of the input string.

If the first unrecognized character is **not** NULL, a pointer to that character is stored into the object that **endptr** points to. If the string is empty or the subject contains no floating point constant description, zero is returned.

SEE ALSO

atof — String to double.
atoi — String to integer.
atol — String to long integer.
strtol — String to long integer.
strtoul — String to unsigned long integer.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char* string = "7.09strtod stopped";
    char* stopped;
    double result;

    result = strtod( string, &stopped );
    printf( "string == (%s)\n", string );
    printf( "result == %f\n", result );
    printf( "stop string == (%s)\n", stopped );
}
```

prints to standard output:

string == (7.09strtod stopped)

result == 7.089998

stop string == (strtod stopped)

NAME

strtok — Break string into tokens.

SYNOPSIS

```
#include <stdlib.h>
```

```
char* strtok( char* s1, const char* s2 );
```

DESCRIPTION

The **strtok** function breaks the string pointed to by **s1** into tokens and each token is delimited by characters from the string pointed to by **s2**. The first call in the sequence has **s1** as its first argument and is followed by calls with a null pointer as the first argument. The separator string, **s2**, may be different from call to call. If a token is not found, a null pointer is returned. If a token is found, a null terminated token is returned.

EXAMPLE

```

#include <stdio.h>
#include <string.h>

void main()
{
    char* str1 = "$%^this#is string\tnumber!one.";
    char str2[] = "?a???b,,,#c";
    char* token;

    while ( ( token = strtok( str1, "$%^#\t! " ) ) != NULL )
    {
        printf( "%s ", token );
        str1 = NULL;
    }

    printf( "\n" );

    token = strtok( str2, "?" ); printf( "%s ", token );
    token = strtok( NULL, "," ); printf( "%s ", token );
    token = strtok( NULL, "#," ); printf( "%s\n", token );

    if ( ( token = strtok( NULL, "?" ) ) != NULL )
    {
        printf( "error: strtok busted\n" );
    }
}

```

prints to standard output:
 this is string number one.
 a ??b c

NAME

strtol — String to long integer.

SYNOPSIS

```
#include <stdlib.h>
```

```
long int strtol( const char* nptr, char** endptr, int base );
```

DESCRIPTION

The **strtol** function converts and returns the string pointed to by **nptr** to a long integer. First **strtol** decomposes **nptr** into three sections;

1. an initial, possibly empty, sequence of white space characters,
2. a subject in the form of an integer constant; and
3. a final string of one or more unrecognized characters, including the

terminating null

character of the input string.

If the first unrecognized character is **not** NULL, a pointer to that character is stored in to the object that **endptr** points to. If the string is empty or the subject contains no floating-point constant description, zero is returned.

If **base** is between 2 and 36, the expected form of the long integer subject is a sequence of letters and digits with the radix specified by base. The letters **a** (or **A**) through **z** (or **Z**) are ascribed values 10 to 35; only letters whose value is less than **base** are valid. If **base** is 16, **0x** or **0X** may optionally proceed the long integer subject. If **base** is zero, the long integer subject determines its own base.

Leading 0x, or 0X base == 16

Leading 0 base == 8

otherwise base == 10

If the value of the return value is too large to be expressed by a long int, **errno** is set to **ERANGE** and **LONG_MAX** is returned.

SEE ALSO

atof — String to double.

atoi — String to integer.

atol — String to long integer.

strtod — String to double.

strtoul — String to unsigned long integer.

EXAMPLE

```
#include <stdio.h>
#include <string.h>

void main()
{
    char* hexstr = "0xabcdef709hexstr stopped";
    char* decstr = "709709709decstr stopped";
    char* octstr = "012341234octstr stopped";
    char* stopped;
    long result;

    printf( "result\t\t\tstop string\n" );
    result = strtol( hexstr, &stopped, 16 );
    printf( "%lx\t\t\t%s", result, stopped );

    result = strtol( decstr, &stopped, 10 );
    printf( "%ld\t\t\t%s", result, stopped );

    result = strtol( octstr, &stopped, 8 );
    printf( "%lo\t\t\t%s", result, stopped );
}
```

prints to standard output:

result	stop string
abcdef709	hexstr stopped
709709709	decstr stopped
12341234	octstr stopped

NAME

strtol — String to unsigned long integer.

SYNOPSIS

```
#include <stdlib.h>
```

```
unsigned long int strtol( const char* nptr, char** endptr, int base );
```

DESCRIPTION

The **strtol** function converts and returns the string pointed to by **nptr** to a long integer. First **strtol** decomposes **nptr** into three sections; an initial, possibly empty, sequence of white space characters, a subject in the form of an integer constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string.

If the first unrecognized character is **not** NULL, a pointer to that character is stored in to the object that **endptr** points to. If the string is empty or the subject contains no floating point constant description, zero is returned.

If **base** is between 2 and 36, the expected form of the long integer subject is a sequence of letters and digits with the radix specified by base. The letters **a** (or **A**) through **z** (or **Z**) are ascribed values 10 to 35; only letters whose value is less than **base** are valid. If **base** is 16, **0x** or **0X** may optionally proceed the long integer subject. If **base** is zero, the long integer subject determines its own base.

Leading 0x, or 0X	base == 16
Leading 0	base == 8
otherwise	base == 10

If the value of the return value is too large to be expressed by a long int, **errno** is set to **ERANGE**, and **ULONG_MAX** is returned.

SEE ALSO

atof	—	String to double.
atoi	—	String to integer.
atol	—	String to long integer.
strtod	—	String to double.
strtoul	—	String to long integer.

EXAMPLE

```

#include <stdio.h>
#include <string.h>

void main()
{
    char* hexstr = "0xabcdef709hexstr stopped";
    char* decstr = "709709709decstr stopped";
    char* octstr = "012341234octstr stopped";
    char* stopped;
    unsigned long result;

    printf( "result\t\t\tstop string\n" );
    result = strtol( hexstr, &stopped, 16 );
    printf( "%lu\t\t\t%s", result, stopped );

    result = strtol( decstr, &stopped, 10 );
    printf( "%lu\t\t\t%s", result, stopped );

    result = strtol( octstr, &stopped, 8 );
    printf( "%lu\t\t\t%s", result, stopped );
}

```

prints to standard output:

result	stop string
2883516169	hexstr stopped
709709709	decstr stopped
12341234	octstr stopped

NAME

strxfrm — Transform a string into locale-independent form.

SYNOPSIS

```
#include <string.h>
```

```
size_t strxfrm( char* s1, const char* s2, size_t n );
```

DESCRIPTION

The **strxfrm** function transforms the string pointed to by **s2** and places the resulting string in the array pointed to by **s1**. The transformation is such that if the **strcmp** function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings. No more than **n** characters are placed into **s1**, including the terminating null character. If **s1** and **s2** overlap, the behavior is undefined.

The **strxfrm** function returns the length of the transformed string excluding the terminating null character. If the value returned is **n** or more, the contents of **s1** are indeterminate.

SEE ALSO

strcoll — Compare two strings based on current locale.
strcmp — Compare two strings.

SPECIAL NOTE

DSP96KCC only supports the standard locale, so no transformation is done.

NAME

tan — Tangent.

SYNOPSIS

```
#include <math.h>
double tan( double x );
```

DESCRIPTION

The **tan** function computes and returns the tangent of **x**, where **x** is in radians.

SEE ALSO

atan — Compute the arc tangent.
atan2 — Compute the arc tangent of a point.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "tan( 45 ) == %f\n", tan( 45 ) );
}
```

prints to standard output:

tan(45) == 1.619775

NAME

tanh — Hyperbolic tangent.

SYNOPSIS

```
#include <math.h>

double tanh( double x );
```

DESCRIPTION

The **tanh** function computes and returns the hyperbolic tangent of **x**,

$$\tanh(x) == \sinh(x) / \cosh(x)$$

If the value of **x** is too large, **errno** is set to **ERANGE** and the value **HUGE_VAL** is returned with the sign of **x**.

SEE ALSO

cosh — Compute the hyperbolic cosine.
sinh — Compute the hyperbolic sine.

EXAMPLE

```
#include <stdio.h>
#include <math.h>

void main()
{
    printf( "tanh( 45 ) == %f\n", tanh( 45 ) );
}
```

prints to standard output:

```
tanh( 45 ) == 1.000000
```

NAME

tolower — Convert uppercase character to lowercase.

SYNOPSIS

```
#include <ctype.h>
int tolower( int c );
```

DESCRIPTION

The **tolower** function converts uppercase to lowercase. If **c** is an uppercase letter, return the corresponding lowercase letter; otherwise return **c**.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    printf( "tolower( 'A' ) == %c\n", tolower( 'A' ) );
    printf( "tolower( 'z' ) == %c\n", tolower( 'z' ) );
    printf( "tolower( '#' ) == %c\n", tolower( '#' ) );
}
```

prints to standard output:

```
tolower( 'A' ) == a
tolower( 'z' ) == z
tolower( '#' ) == #
```

NAME

toupper — Convert lowercase character to uppercase.

SYNOPSIS

```
#include <ctype.h>
int toupper( int c );
```

DESCRIPTION

The **toupper** function converts lowercase to uppercase. If **c** is a lowercase letter, return the corresponding uppercase letter; otherwise return **c**.

When the header file **ctype.h** is included, the default case will be **in-line** [see section A.3, Forcing Library Routines Out-of-line].

EXAMPLE

```
#include <stdio.h>
#include <ctype.h>

void main()
{
    printf( "toupper( 'A' ) == %c\n", toupper( 'A' ) );
    printf( "toupper( 'z' ) == %c\n", toupper( 'z' ) );
    printf( "toupper( '#' ) == %c\n", toupper( '#' ) );
}
```

prints to standard output:

```
toupper( 'A' ) == A
toupper( 'z' ) == Z
toupper( '#' ) == #
```

NAME

wcstombs — Convert wchar_t array to multibyte string.

SYNOPSIS

```
#include <stdlib.h>
```

```
size_t wcstombs( char* s, const wchar_t* pwcs, size_t n );
```

DESCRIPTION

The **wcstombs** function converts a wide character string pointed to by **pwcs** into the character string pointed to by **s**. Each character of the wide character string is converted into the corresponding multibyte character as if by the **wctomb** function. Conversion will stop when **n** total characters have been converted or a null character is encountered. If **s** and **pwcs** overlap, the behavior is undefined.

If an invalid character is encountered, **wcstombs** returns (size_t) -1. Otherwise, **wcstombs** returns the number of characters converted not including the terminating NULL character, if any.

SEE ALSO

mbtowcs — Convert a multibyte string to a wchar_t array.

SPECIAL NOTE

The DSP96002 does not provide byte addressing, thus characters always require an entire word of memory each. One way to better utilize data memory (with a run-time cost) is to combine the ANSI data type **wchar_t** and the special ANSI **multibyte** and **wide character** library routines.

EXAMPLE

```

#include <stdio.h>
#include <stdlib.h>

char array[16];
wchar_t wstr[10] = L"abcdefgh";

void main()
{
    char* ptr = (char*) wstr;
    int convert;

    convert = wcstombs( array, wstr, 10 );

    printf( "packed array:\n" );
    while ( *ptr != 0 )
    {
        printf( "%0.6x ", *ptr++ );
    }
    printf( "\n\n" );

    printf( "%d chars extracted, unpacked array:\n", convert );
    ptr = array;
    while ( *ptr != 0 )
    {
        printf( "%0.6x ", *ptr++ );
    }
    printf( "\n" );
}

prints to standard output:
packed array:
616263 646966 676800

8 chars extracted, unpacked array:
000061 000062 000063 000064 000065 000066 000067 000068

```

NAME

wctomb — Convert wchar_t character to multibyte character.

SYNOPSIS

```
#include <stdlib.h>
```

```
int wctomb( char* s, wchar_t wchar );
```

DESCRIPTION

The **wctomb** function examines and converts the wide character **wchar** into a string of characters pointed to by **s**. At most, **MB_CUR_MAX** characters will be stored in **s**.

If **s** is a NULL pointer, **wctomb** returns zero. If **s** is not NULL, **wctomb** returns the number of characters that comprise the converted multibyte character unless an invalid multibyte character is detected in which case a -1 will be returned.

SEE ALSO

mblen — Determine the length of a multibyte character.
mbstowcs — Convert a multibyte string into a wide character string.
mbtowc — Convert a multibyte character into a wide character.

SPECIAL NOTE

The DSP96002 does not provide byte addressing, thus characters always require an entire word of memory each. One way to better utilize data memory (with a run-time cost) is to combine the ANSI data type **wchar_t** and the special ANSI **multibyte** and **wide character** library routines.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

char mbararray[8];

void main()
{
    wchar_t wide = L'abc';
    char* ptr = array;
    int convert;

    convert = wctomb( array, wide );

    printf( "packed char looks like:\n" );
    printf( "%0.6x\n\n", wide );

    printf( "%d extracted chars looks like:\n", convert );
    while ( *ptr != 0 )
    {
        printf( "%0.6x ", *ptr++ );
    }
    printf( "\n" );
}
```

prints to standard output:

packed char looks like:

616263

3 extracted chars looks like:

000061 000062 000063

Appendix B

DSP96002 Instruction Set and Assembler Directive Summary

B.1 Overview

This appendix is intended to assist the C programmer in understanding the C compiler's output. For a more in-depth discussion of assembler and assembly language issues, see the DSP96002 Digital Signal Processor User's Manual or the DSP96002 Assembler User's Manual.

B.2 Instruction Set

Instructions can be grouped by function into six types:

1. Arithmetic instructions
2. Logical instructions
3. Bit manipulation instructions
4. Loop instructions
5. Move instructions
6. Program control instructions

B.2.1 Arithmetic Instructions

The instructions used for arithmetic operations are:

ABS	— Absolute value
ADD	— Add
ADDC	— Add with carry
ASL	— Arithmetic shift accumulator left
ASR	— Arithmetic shift accumulator right
BFIN	— Find leading one
CLR	— Clear accumulator

CMP	— Compare
CMPG	— Compare magnitude
DEC	— Decrement by one
EXT	— Sign extend half word
EXTB	— Sign extend byte
FABS	— Floating point absolute value
FADD	— Floating point add
FADDSUB	— Floating point add and subtract
FCLR	— Clear floating point register
FCMP	— Floating point compare
FCMPG	— Floating point graphics compare
FCMPM	— Floating point magnitude compare
FCOPYS	— Floating point copy sign
FGETMAN	— Extract mantissa
FINT	— Extract integer
FLOAT	— Convert integer to floating point
FLOATU	— Unsigned integer to floating point
FLOOR	— Extract integer
FMPY	— Floating point multiply
FMPY/FADD	— Floating point multiply and add
FMPY/FSUB	— Floating point multiply and subtract
FMPY/FADDSUB	— Floating point multiply and add/subtract
FNEG	— Floating point negate
FSCALE	— Scale floating point operand
FSEEDD	— Reciprocal approximation
FSEEDR	— Square root reciprocal approximation
FSUB	— Floating point subtract
FTST	— Test floating point operand
GETEXP	— Extract exponent
INC	— Increment by one
INT	— Convert floating point to integer
INTRZ	— Convert floating point to integer with round to zero
INTU	— Convert floating point to unsigned integer
INTURZ	— Convert floating point to unsigned integer with round to zero
JOIN	— Join two 16-bit integers

JOINB	— Join two 8-bit integers
MPYS	— Signed multiply
MPYU	— Unsigned multiply
NEG	— Negate
NEGC	— Negate with carry
RND	— Round accumulator
SETW	— Set operand
SPLIT	— Extract 16-bit integer
SPLITB	— Extract 8-bit integer
SUB	— Subtract
SUBC	— Subtract with carry
TST	— Test

B.2.2 Logical Instructions

The instructions used for logical operations are:

AND	— Logical AND
ANDC	— Logical AND with complement
ANDI	— AND immediate to control register
EOR	— Logical exclusive OR
LSL	— Logical shift accumulator left
LSR	— Logical shift accumulator right
NOT	— Logical complement on accumulator
OR	— Logical inclusive OR
ORC	— Logical inclusive OR with complement
ORI	— OR immediate with control register
ROL	— Rotate accumulator left
ROR	— Rotate accumulator right

B.2.3 Bit Manipulation Instructions

The instructions used for bit manipulation are:

BCHG	— Bit test and change
BCLR	— Bit test and clear
BRCLR	— Branch if bit clear
BRSET	— Branch if bit set
BSCLR	— Branch to subroutine if bit clear
BSET	— Bit test and set

BSR	— Branch to subroutine
BSSET	— Branch to subroutine if bit set
BTST	— Bit test on memory
JCLR	— Jump if bit clear
JSCLR	— Jump to subroutine if bit clear
JSET	— Jump if bit set
JSSET	— Jump to subroutine if bit set

B.2.4 Loop Instructions

The instructions used for loop operations are:

DO	— Start hardware loop
DOR	— Start PC-relative hardware loop
ENDDO	— Exit from hardware loop

B.2.5 Move Instructions

The instructions used for move operations are:

FTFR	— Transfer floating point data ALU register
LEA	— Load effective address
LRA	— Load PC-relative address
MOVE	— Move data
MOVEC	— Move control register
MOVEI	— Move immediate short
MOVEM	— Move program memory
MOVEP	— Move peripheral data
MOVES	— Move absolute short
MOVETA	— Move data registers and test address
TFR	— Transfer data ALU register

B.2.6 Program Control Instructions

The DSP96000 instructions used for program control are:

Bcc	— Branch conditionally
BRA	— Branch always
BScC	— Branch to subroutine conditionally
DEBUGcc	— Enter debug mode conditionally
FBcc	— Floating point branch conditionally

FBScc	— Floating point branch to subroutine conditionally
FDEBUGcc	— Enter debug mode conditionally
FJcc	— Floating point jump conditionally
FJScc	— Floating point jump to subroutine conditionally
FTRAPcc	— Floating point conditional software interrupt
ILLEGAL	— Illegal instruction interrupt
Jcc	— Jump conditionally
JMP	— Jump
JScc	— Jump to subroutine conditionally
JSR	— Jump to subroutine
NOP	— No operation
REP	— Repeat next instruction
RESET	— Reset on-chip peripheral devices
RTI	— Return from interrupt
RTR	— Return from subroutine with restore
RTS	— Return from subroutine
STOP	— Stop processing (low power stand-by)
WAIT	— Wait for interrupt (low power stand-by)

B.3 Directive Summary

Directives can be grouped by function into seven types:

1. Assembly control
2. Symbol definition
3. Data definition / storage allocation
4. Listing control and options
5. Object file control
6. Macros and conditional assembly
7. Structured programming

B.3.1 Assembly Control

The directives used for assembly control are:

COMMENT	— Comment line(s)
DEFINE	— Define substitution strings
END	— End of source program
FAIL	— Programmer generated error message

HIMEM	— Set high memory bounds
INCLUDE	— Include secondary file
LOMEM	— Set low memory bounds
MODE	— Change relocation mode
MSG	— Program generated message
ORG	— Initialize memory space and location counters
RADIX	— Change input radix for constants
RDIRECT	— Remove directive / mnemonic from assembler table
UNDEF	— Undefine DEFINE symbol
WARN	— Programmer generated warning message

B.3.2 Symbol Definition

The directives used to control symbol definition are:

ENDSEC	— End section
EQU	— Equate symbol to a value
GLOBAL	— Global section symbol declaration
LOCAL	— Local section symbol declaration
SECTION	— Start section
SET	— Set symbol to a value
XDEF	— External section symbol definition
XREF	— External section symbol reference

B.3.3 Data Definition/Storage Allocation

The directives used to control constant data definition and storage allocation are:

BADDR	— Set buffer address
BSB	— Block storage bit-reverse
BSC	— Block storage of a constant
BSM	— Block storage modulo
BUFFER	— Start buffer
DC	— Define constant
DS	— Define storage
DSM	— Define modulo storage
DSR	— Define reverse carry storage
ENDBUF	— End buffer

B.3.4 Listing Control and Options

The directives used to control the output listing are:

LIST	— List the assembly
LSTCOL	— Set listing field widths
NOLIST	— Stop assembly listing
OPT	— Assembler options
PAGE	— Top of page / define page size
PRCTL	— Send control string to printer
STITLE	— Initialize program sub—title
TABS	— Set listing tab stops
TITLE	— Initialize program title

B.3.5 Object File Control

The directives used to control the object file are:

COBJ	— Comment object code
IDENT	— Object code identification record
SYMOBJ	— Write symbol information to object file

B.3.6 Macros and Conditional Assembly

The directives used for macros and conditional assembly are:

DUP	— Duplicate a sequence of source lines
DUPA	— Duplicate sequence with arguments
DUPC	— Duplicate sequence with characters
ENDIF	— End of conditional assembly
ENDM	— End of macro definition
EXITM	— Exit macro
IF	— Conditional assembly directive
MACLIB	— Macro library
MACRO	— Macro definition
PMACRO	— Purge a macro definition

B.3.7 Structured Programming

The directives used for structured programming are:

.BREAK	— Exit from structured loop construct
.CONTINUE	— Continue next iteration of structured loop

.ELSE	— Perform following statements when .IF false
.ENDF	— End of .FOR loop
.ENDI	— End of .IF condition
.ENDL	— End of hardware loop
.ENDW	— End of .WHILE loop
.FOR	— Begin .FOR loop
.IF	— Begin .IF condition
.LOOP	— Begin hardware loop
.REPEAT	— Begin .REPEAT loop
.UNTIL	— End of .REPEAT loop
.WHILE	— Begin .WHILE loop

Appendix C

Utilities

There are nine utility programs available with the DSP96KCC Rev. G1.0 compiler. They are:

1. asm96000
2. cldinfo
3. cldlod
4. cofdmp
5. dsplib
6. dsplnk
7. gdb96
8. run96sim
9. srec

These programs are described in detail in the following pages.

NAME

asm96000 — Motorola DSP96000 COFF Assembler

SYNOPSIS

```
asm96000 [ -A ] [ -B [ <objfil> ] ] [ -D <symbol> <string> ] [ -F <argfil> ]  
[ -I <ipath> ] [ -L [ <lstfil> ] ] [ -M <mpath> ] [ -O <opt> [ , <opt> ... ] ]  
[ -R <rev> [ , <rev>... ] ] [ -V ] <files...>
```

DESCRIPTION

asm96000 is a program that processes source program statements written in DSP96002 assembly language, translating these source statements into object programs compatible with other DSP96002 software and hardware products.

files is a list of operating system compatible file names including optional pathnames. If no extension is supplied for a given file, the assembler will first attempt to open the file using the file name as supplied. If that is not successful, the assembler appends .asm to the file name and tries to open the file again. If no path is given for a particular file, the assembler will look for that file in the current directory. The list of files will be processed sequentially in the order given and all files will be used to generate the output listing and object file.

The assembler will redirect the output listing to the standard output if it is not redirected via the -L command line option described below. Error messages will always appear on the standard output regardless of any option settings. Note that some options (-B and -L) allow a hyphen as an optional argument which indicates that the corresponding output should be sent to the standard output stream. Unpredictable results may occur if, for example, the object file is explicitly routed to standard output while the listing file is allowed to default to the same output stream.

OPTIONS

Any of the following command line options may be specified. These can be in any order but must precede the list of source file names. Option letters may be entered in either upper or lower case.

Option arguments may immediately follow the option letter or may be separated from the option letter by blanks or tabs. However, an ambiguity arises if an option takes an optional argument. Consider the following command line:

asm96000 -b main io

In this example it is not clear whether the file main is a source file or is meant to be an argument to the -B option. If the ambiguity is not resolved, the assembler will assume that main is a source file and attempt to open it for reading. This may not be what the programmer intended.

There are several ways to avoid this ambiguity. If main is supposed to be an argument to the -B option, it can be placed immediately after the option letter, without intervening white space:

asm96000 -bmain io

If there are other options on the command line besides those that take optional arguments, the other options can be placed between the ambiguous option and the list of source file names:

asm96000 -b main -v io

Alternatively, two successive hyphens may be used to indicate the end of the option list:

asm96000 -b -- main io

In this case the assembler interprets main as a source file name and uses the default naming conventions for the -B option.

- A** Indicates that the assembler should operate in absolute mode, creating a load file (.cld) if the **-B** option is given. By default, the assembler produces a link file (.cln) which is subsequently processed by the Motorola DSP linker.

-B[<objfil>]

This option specifies that an object file is to be created for assembler output. **objfil** can be any legal operating system file name, including an optional pathname. A hyphen may also be used as an argument to indicate that the object file should be sent to the standard output.

If a path is not specified, the file will be created in the current directory. If no file name is supplied, the assembler will use the basename (file name without extension) of the first file name encountered in the source input file list. The resulting output file will have an extension of .cln unless the **-A** option is given in which case the file will have a .cld extension. If the **-B** option is

not given, then the assembler will not generate an object file. The **-B** option should be specified only once.

-D<symbol> <string>

This is equivalent to a source statement of the form:

DEFINE <symbol> <string>

string must be enclosed in quotes if it contains any embedded blanks. Note that if quotes are used, they must be passed to the assembler intact, e.g. some host command interpreters will strip quotes from around arguments. The **-D<symbol> <string>** sequence may be repeated as often as desired.

-F<argfil>

This option indicates that an external file should be read for further command arguments. It is useful in host environments where the command line length is restricted. **argfil** must be present on the command line, but can be any legal operating system file name including an optional pathname.

The file may contain any legal command line options, including the **-F** option itself. The arguments need be separated only by white space (spaces, tabs, or newlines). A semicolon (;) on a line following white space causes the rest of the line in the file to be treated as a comment.

-I<ipath>

When the assembler encounters include files, the current directory (or the directory specified in the INCLUDE directive) is first searched for the file. If it is not found and the **-I** option is supplied, the assembler prefixes the file name (and optional pathname) specified in the INCLUDE directive with **ipath** and searches the newly formed directory pathname for the file. The **-I<ipath>** sequence may be repeated as many times as desired. The directories will be searched in the order given on the command line.

-L[<lstfil>]

This option specifies that a listing file is to be created for the assembler output. **lstfil** can be any legal operating system file name including an optional pathname. A hyphen also may be used as an argument to indicate that the listing file should be sent to the standard output.

If a path is not specified, the file will be created in the current directory. If no file name is supplied, the assembler will use the basename (file name without extension) of the first file name encountered in the source input file list. The resulting output file will have an extension of .lst. If the **-L** option is not given, then the assembler will route the listing output to the standard output. The **-L** option should be specified only once.

-M<mpath>

This is equivalent to a source statement of the form:

MACLIB <mpath>

The **-M<mpath>** sequence may be repeated as many times as desired. The directories will be searched in the order specified on the command line.

-O<opt>[,<opt>...]

opt can be any of the options that are available with the assembler **OPT** directive. If multiple options are supplied, they must be separated by commas. The **-O<opt>** sequence may be repeated for as many options as desired.

-R<rev>[,<rev>...]

Run the assembler without the specified processor revision level enhancements. This is for backward compatibility so that the assembler will flag new constructions as illegal. **rev** can be any of the revision specifiers given below, but must be appropriate for the target processor.

Processor	Revision
DSP96000	C

If multiple revision levels are supplied, they must be separated by commas. The **-R<rev>** sequence may be repeated as many times as desired.

- V** Indicates that the assembler should be verbose during processing, displaying a progress report as it assembles the input files. The assembler will show the beginning of each pass and when files are opened and closed. The information is sent to the standard error output stream.

NAME

cldinfo — Memory size information from Motorola DSP COFF object file.

SYNOPSIS

cldinfo [file]

DESCRIPTION

cldinfo is a utility that reads an absolute or relocatable Common Object File Format (COFF) file and produces a formatted display of the program memory size, data memory size and the programs starting address.

file is an operating system compatible file name. Only a single file name may be supplied, and it must be explicit; there is no default extension for the input file.

NAME

cldlod — Motorola COFF to LOD Format converter

SYNOPSIS

cldlod [file]

DESCRIPTION

cldlod is a utility that converts a binary COFF object file into an ascii LOD file.

file is an operating system compatible file name. Only a single file name may be supplied, and it must be explicit; there is no default extension for the input file.

NAME

cofdmp — Motorola DSP COFF File Dump Utility

SYNOPSIS

cofdmp [-cfhlorstv] [file]

DESCRIPTION

cofdmp is a utility that reads an absolute or relocatable COFF file and produces a formatted display of the object file contents. The entire file or only selected portions may be processed depending on command line options. The program also can generate either codes or symbolic references to entities such as symbol type or storage class.

file is an operating system compatible file name. Only a single file name may be supplied, and it must be explicit; there is no default extension for the input file.

OPTIONS

Any of the following command line options may be given. Option letters may be entered in either upper or lower case. If no option is specified, the entire object file is dumped.

- c** Dump the object file string table. This information may not be available if the object file has been stripped.
- f** Dump the file header of the object file.
- h** Dump the object file section headers.
- l** Dump the object file line number information. This information may not be available if the object file has been stripped.
- o** Dump the object file optional header.
- r** Dump the object file relocation information. This information is available only in relocatable object files.
- s** Dump the object file raw data contents.
- t** Dump the object file symbol table.
- v** Dump the object file symbolically, expanding bit flag, symbol type, and storage class names.

NAME

dsplib — Motorola DSP COFF Librarian

SYNOPSIS

dsplib [**-a** | **-c** | **-d** | **-l** | **-r** | **-u** | **-v** | **-x**] [**-f<argfil>**] library [files...]

DESCRIPTION

dsplib is a utility that allows separate files to be grouped together into a single file. The resulting library file can then be used for linking by the Motorola DSP Cross Linker program or for general-purpose archival storage.

library is an operating system compatible file name (including optional pathname) indicating the library file to create or access. If no extension is supplied, the librarian will automatically append .clb to the file name. If no pathname is specified, the librarian will look for the library in the current directory.

files is a list of operating system compatible file names. For input operations the file names may also contain an optional pathname; the path is stripped when the file is written to the library. For output operations only the file name should be used to refer to library modules.

If no arguments are given on the command line, the librarian enters an interactive mode where multiple commands may be entered without exiting the program. The syntax for the interactive mode is command library [files...] where command is an action corresponding to one of the options listed below, library is the library name, and files is the optional (based on the action requested) list of files/modules upon which to operate. For example the command add foo bar.cln adds the module bar.cln to the library foo. Because interactive input is taken from the standard input channel of the host environment, it is possible to create a batch of librarian commands and feed them to the program for execution via redirection. For more information on interactive commands, invoke the librarian without any arguments and enter help.

OPTIONS

Only one of the following command line options may be given for each invocation of the librarian. Option letters may be entered in either upper or lower case. If no option is given, the librarian operates as if the **-U** option were specified.

- a** This option adds the modules in the file list to the named library. The library file must exist and the modules must not already be in the library.
- c** Create a new library file and add any specified modules to it. If the library file already exists, an error is issued.
- d** Delete named modules from the library. If the module is not in the library, an error is issued.
- f<argfil>**

This option indicates that an external file should be read for further command arguments. It is useful in host environments where the command line length is restricted. **argfil** must be present on the command line but can be any legal operating system file name, including an optional pathname.

argfil is a text file containing module names to be passed to the librarian. The names need be separated only by white space (spaces, tabs, or new-lines). A semicolon (;) on a line following white space causes the rest of the line to be treated as a comment.
- l** List library contents. This option lists the module name as contained in the library header, the module size (minus library overhead), and the date and time the file was stored into the library. The listing output is routed to standard output so that it may be redirected to a file if desired.
- r** This option replaces the named modules in the given library. The modules must already be present in the library file.
- u** This option updates the specified modules if they exist in the library; otherwise it adds them to the end of the library file.
- v** This option displays the librarian version number and copyright notice.
- x** Extract named modules from the library. The resulting files are given the name of the modules as stored in the library module header. All files are created in the current working directory.

NAME

dsplnk — Motorola DSP COFF Linker

SYNOPSIS

```
dsplnk [ -B [<lodfil> ] ] [ -F<argfil> ] [ -I ] [ -L<library> ] [ -M<mapfil> ] [ -N ]  
[ -O<mem>[<ctr>][<map>]:<origin> ] [ -P<lpath> ] [ -R<ctlfil> ] [ -Z ]  
[ -U<symbol> ] [ -V ] <files...>
```

DESCRIPTION

dsplnk is a program that processes relocatable link files produced by the DSP assemblers, generating an absolute load file which can be

1. loaded directly into the Motorola DSP simulator or
2. converted to Motorola S-record format for PROM burning.

files is a list of operating system compatible file names including optional pathnames. If no extension is supplied for a given file, the linker will first attempt to open the file using the file name as supplied. If that is not successful the linker appends .cln to the file name and tries to open the file again. If no pathname is supplied for a given file, the linker will look for that file in the current directory. The list of files will be processed sequentially in the order given and all files will be used to generate the load file and map listing.

Note that some options (**-B** and **-M**) allow a hyphen as an optional argument which indicates that the corresponding output should be sent to the standard output stream. Unpredictable results may occur if, for example, the object file is explicitly routed to standard output while the listing file is allowed to default to the same output stream.

OPTIONS

Any of the following command line options may be specified. These can be in any order but must precede the list of link file names (except for the **-L** option). Option letters may be specified in either upper or lower case.

Option arguments may immediately follow the option letter or may be separated from the option letter by blanks or tabs. However, an ambiguity arises if an option takes an optional argument. Consider the following command line:

```
dsplnk -b main io
```

In this example it is not clear whether the file `main` is a link file or is meant to be an argument to the **-B** option. If the ambiguity is not resolved, the linker will assume that `main` is a link file and attempt to open it for reading. This may not be what the programmer intended.

There are several ways to avoid this ambiguity. If `main` is supposed to be an argument to the **-B** option, it can be placed immediately after the option letter without intervening white space:

```
dsplnk -bmain io
```

If there are other options on the command line besides those that take optional arguments the other options can be placed between the ambiguous option and the list of link file names.

```
dsplnk -b main -v io
```

Alternatively, two successive hyphens may be used to indicate the end of the option list:

```
dsplnk -b -- main io
```

In this case the linker interprets `main` as a link file name and uses the default naming conventions for the **-B** option.

-B[<objfil>]

This option specifies a name for the object file generated by the linker. **objfil** can be any legal operating system file name including an optional pathname. A hyphen may also be used as an argument to indicate that the object file should be sent to the standard output.

If a pathname is not given, the file will be created in the current directory. If no file name is supplied or if the **-B** option is not given, the linker will use the basename (file name without extension) of the first file name encountered in the link input file list. The resulting output file will have an extension of `.cld`. The **-B** option should be specified only once.

-F<argfil>

This option indicates that an external file should be read for further command arguments. It is useful in host environments where the command line length is restricted. **argfil** must be present on the command line but can be any legal operating system file name, including an optional pathname.

The file may contain any legal command line options including the **-F** option itself. The arguments need be separated only by white space (spaces, tabs, or newlines). A semicolon (;) on a line following white space causes the rest of the line to be treated as a comment.

- I Under normal operation, the linker produces an absolute load file as output. If the **-I** option appears on the command line, the linker combines the input files into a single relocatable link file suitable for a subsequent linker pass. No absolute addresses are assigned and no errors are issued for unresolved external references.

-L<library>

The linker ordinarily processes a list of link files which each contain a single relocatable code module. If the **-L** option is encountered, the linker treats the accompanying pathname as a library file, and searches the file for any outstanding unresolved references.

If a module is found in the library that resolves an outstanding external reference, the module is read from the library and included in the load file output. The linker continues to search a library until all external references are resolved or no more references can be satisfied within the current library. The linker searches a library only once: when it is encountered on the command line. Therefore, the position of the **-L** option on the command line is significant.

-M[<mapfil>

This option specifies that a map file is to be created. **mapfil** can be any legal operating system file name including an optional pathname.

If a pathname is not given, the file will be created in the current directory. If no file name is supplied, the linker will use the basename (file name without extension) of the first file name encountered in the link input file list. The resulting output file will have an extension of **.map**. The linker will not generate a map file if the **-M** option is not specified. The **-M** option should be specified only once.

- N Indicates that the linker should ignore case in symbol names. Ordinarily the linker is sensitive to upper and lower case letters in symbol names. If the **-N** option is supplied the linker maps all symbol characters to lower case.

-O<mem>[<ctr>][<map>]:<origin>

By default, the linker generates instructions and data for the load file beginning at absolute location zero for all DSP memory spaces. This option allows the programmer to redefine the start address for any memory space and associated location counter.

mem is one of the single-character memory space identifiers (X, Y, L, and P). The letter may be upper or lower case. The optional **ctr** is a letter indicating the high (H) or low (L) location counters. If no counter is specified, the default counter is used. **map** is also optional and signifies the desired

physical mapping for all relocatable code in the given memory space. It may be I for internal memory, E for external memory, or B for bootstrap memory (valid only in P program memory space). If **map** is not supplied, then no explicit mapping is presumed.

The **origin** is a hexadecimal number signifying the new relocation address for the given memory space. The **-O** option may be specified as many times as needed on the command line.

-P<lpath>

When the linker encounters a library specification on the command line, the current directory (or the directory given in the library specification) is first searched for the library file. If it is not found and the **-P** option is supplied, the linker prefixes the file name (and optional pathname) provided in the library specification with *lpath* and searches the newly formed directory pathname for the file. The directories will be searched in the order given on the command line.

-R[<ctlfil>]

This option indicates that a memory control file is to be read to determine the absolute placement of sections in DSP memory. **ctlfil** can be any legal operating system file name including an optional pathname.

If a pathname is not given, an attempt will be made to open the file in the current directory. If no file name is supplied, the linker will use the base-name (file name without extension) of the first file name encountered in the link input file list, appending an extension of **.ctl**. If the **-R** option is not specified, then the linker will not use a memory map file. The **-R** option should be specified only once.

-U<symbol>

Causes symbol to be entered into the unresolved external reference table. This is useful when the initial or only link file is a library. Since there are no external references when the linker is invoked, the **-U** option may be used to force inclusion of a library module that resolves the undefined reference. The **-U** option may be specified as often as desired.

-V Indicates that the linker should be verbose during processing, displaying a progress report as it links the input files. The linker will show the beginning of each pass and when files are opened and closed. The information is sent to the standard error output stream.

-Z The linker preserves symbol information in the load file when the **-Z** option appears on the command line. This includes the symbol name, its memory space and mapping attributes, and its value.

NAME

gdb96 — GNU Source-level Debugger for DSP96002

SYNOPSIS

```
gdb96 [ -s SYMFILE ] [ -e EXECFILE ] [ -se FILE ] [ -c COREFILE ] [ -x FILE ]  
[ -d directory ] [ -nx ] [ -q ] [ -batch ] [ -fullname ] [ -help ] [ -tty TTY ]  
[ -cd DIR ] <name> <core>
```

DESCRIPTION

gdb96 is a source-level symbolic debugger for C programs created by Richard Stallman for the GNU Project and distributed by the Free Software Foundation. **gdb96** has something of the flavor of the UNIX source-level debugger **dbx** but has more features and power.

gdb96 is invoked with the shell command **gdb96**. Once started, it reads commands from the terminal until given the quit command. **name** is the name of the executable program, and **core**, if specified, is the name of the DSP96002 state file to be examined or have execution resumed.

gdb96 has been modified to run with both software and hardware DSP96002 execution devices.

OPTIONS

The following command-line options can be used to more precisely specify the files to be debugged. All the options and command line arguments given are processed in sequential order. The order makes a difference when the **-x** command is used.

-s SYMFILE

Read symbol table from file SYMFILE.

-e EXECFILE

Use EXECFILE as the executable file to execute when appropriate and for examining pure data in conjunction with a core dump.

-se FILE

Read symbol table from FILE and use it as the executable file.

-c COREFILE

Analyze the core dump COREFILE.

-x file

Execute GDB commands from file.

-d directory

Add directory to the path to search for source files.

The following additional command-line options can be used to affect certain aspects of the behavior of gdb96:

-nx Don't execute commands from the init files `.gdbinit`. Normally, the commands in these files are executed after all the command options and arguments have been processed.

-q Do not print the version number on start-up.

-batch

Run in batch mode. Exit with code 1 after processing all the command files specified with **-x** (and `.gdbinit`, if not inhibited). Exit also if, due to an error, gdb96 would otherwise attempt to read a command from the terminal.

-fullname

This option is used when Emacs runs GDB as a subprocess. It tells GDB to produce the full file name and line number each time a stack frame is displayed (which includes each time the program stops).

-help

Print command-line argument summary.

-tty TTY

Use **TTY** for input/output by the program being debugged.

-cd DIR

Change current directory to **DIR**.

SEE ALSO

Appendix E — GNU Debugger (GDB)

NAME

run96sim — Motorola DSP96002 Simulator Based Execution Device.

SYNOPSIS

```
run96sim [ -b BCR_VALUE ] [ -s STATE_FILE ] [ -t ] [ file ]
```

DESCRIPTION

run96sim is a COFF object file execution utility that provides operating system style hooks to support hosted ANSI run-time library routines such as **printf()**.

file is an operating system compatible file name. Only a single file name may be supplied and it must be explicit; there is no default extension for the input file.

OPTIONS**-b BCR_VALUE**

Use BCR_VALUE to set the DSP96002 bus control register. Default bcr value is 0.

-s STATE_FILE

Resume the execution of the DSP96002 state file STATE_FILE.

-t

Update global C variable `__time` at each clock tick. Used to benchmark code.

NAME

srec — Motorola DSP S-Record Conversion Utility

SYNOPSIS

srec [**-b** | **-w**] [**-m** | **-s**] [**-l**] [**-r**] <files ...>

DESCRIPTION

srec converts Motorola DSP .cld and .lod format files into Motorola S-record files. The S-record format was devised for the purpose of encoding programs or data files in a printable form for transportation between computer systems. Motorola S-record format is recognized by many PROM programming systems.

files is a list of operating system compatible file names. If no pathname is specified for a given file, srec will look for that file in the current directory. If the special character '-' is used as a file name srec will read from the standard input stream. The list of files will be processed sequentially in the order given.

OPTIONS

Only one of the following command line options may be given for each invocation of the librarian. Option letters may be entered in either upper or lower case.

- b** Use byte addressing when transferring load addresses to S-record addresses. This means that load file DATA record start addresses are multiplied by the number of bytes per target DSP word and subsequent S1/S3 record addresses are computed based on the data byte count. The **-b** and **-w** options are mutually exclusive.
- l** Use double-word addressing when transferring load addresses from L space to S-record addresses. This means that load file DATA records for L space data are moved unchanged and subsequent S1/S3 record addresses are computed based on the data word count divided by 2. This option should always be used when the source load file contains DATA records in L memory space.
- m** Split each DSP word into bytes and store the bytes in parallel S-records. The **-m** and **-s** options are mutually exclusive.
- r** Write bytes high to low, rather than low to high. This option has no effect when used with the **-m** option.

- s** Write data to a single file, putting memory space information into the address field of the S0 header record. The **-m** and **-s** options are mutually exclusive.
- w** Use word addressing when transferring load addresses to S-record addresses. This means that load file DATA record start addresses are moved unchanged and subsequent S1/S3 record addresses are computed based on the data word count.

srec

srec

Appendix D

Additional Support

User support from the conception of a design through completion is available from Motorola and third-party companies as shown in the following list:

	Motorola	Third Party
Design	Data Sheets Application Notes Application Bulletins Software Examples	Data Acquisition Packages Filter Design Packages Operating System Software Simulator
Prototyping	Assembler Linker C Compiler Simulator Application Development System (ADS) In-Circuit Emulator Cable for ADS	Logic Analyzer with DSP96002 ROM Packages Data Acquisition Cards DSP Development System Cards Operating System Software Debug Software
Design Verification	Application Development System (ADS) In-Circuit Emulator Simulator	Data Acquisition Packages Logic Analyzer with DSP96002 ROM Packages Data Acquisition Cards DSP Development System Cards Application-Specific Development Tools Debug Software

The following is a partial list of the support available for the DSP96002. Additional information can be obtained through Dr. BuB or the appropriate support telephone service.

Motorola DSP Product Support

- DSP96000CLASx Design-In Software Package which includes:
 - Relocatable Macro Assembler
 - Linker
 - Simulator (simulates single or multiple DSP96002s)
 - Librarian
- DSP96KCCx Full ANSI Compliant C Compiler
- DSP96002 Applications Development System (ADS)
- Support Integrated Circuits
- DSP Bulletin Board (Dr. BuB)
- Motorola DSP Newsletter
- Motorola Technical Service Engineers (TSEs)
See your local telephone directory for the Motorola Semiconductor Sector sales office telephone number.
- Design Helpline
- Applications Assistance
- Marketing Information
- Third-Party Support Information
- University Support Information

DSP96002CLASx Assembler/Simulator

The macro cross assembler and simulator run on:

1. IBM™ PCs and clones using an 80386 or upward compatible processor
2. Macintosh™ computers with a NU-BUS™ expansion port
3. SUN computer
4. NeXT™ computer

Macro Cross Assembler Features:

- Production of relocatable object modules compatible with linker program when in relocatable mode
- Production of absolute files compatible with simulator program when in absolute mode
- Supports full instruction set, memory spaces, and parallel data transfer fields of

the DSP96002

- Modular programming features: local labels, sections, and external definition/reference directives
- Nested macro processing capability with support for macro libraries
- Complex expression evaluation including boolean operators
- Built-in functions for data conversion, string comparison, and common transcendental math functions
- Directives to define circular and bit-reversed buffers
- Extensive error checking and reporting

Simulator Features:

- Simulation of DSP96002
- Simulation of multiple DSP96002
- Linkable object code modules:
 - Nondisplay simulator library
 - Display simulator library
- C language source code for:
 - Screen management functions
 - Terminal I/O functions
 - Simulation examples
- Single stepping through object programs
- Conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Instruction, clock cycle, and histogram counters
- Session and/or command logging for later reference
- ASCII input/output files for peripherals
- Help-line display and expanded on-line help for simulator commands
- Loading and saving of files to/from simulator memory
- Macro command definition and execution
- Display enable/disable of registers and memory
- Hexadecimal/decimal/binary calculator

DSP96000ADSx Application Development System

DSP96000ADS Application Development System Hardware Features:

- Full-speed 40 MHz operation
- Multiple application development module (ADM) support with programmable ADM

addresses

- 128Kx32 (expandable to 512Kx32) user-configurable RAM for DSP96002 code development
- 2Kx32 monitor ROM expandable to 64Kx32
- 96-pin Euro-card connector making all DSP96002 pins accessible
- ADM can be used in stand-alone configuration
- No external power supply needed when connected to a host platform

DSP96000ADSx Application Development System Software Features:

- Single/multiple stepping through DSP96002 object programs
- Up to 99 conditional or unconditional breakpoints
- Program patching using a single-line assembler/disassembler
- Session and/or command logging for later reference
- Loading and saving files to/from ADM memory
- Macro command definition and execution
- Display enable/disable of registers and memory
- Debug commands supporting multiple ADMs
- Hexadecimal/decimal/binary calculator
- Host operating system commands from within ADS user interface program
- Multiple OS I/O file access from DSP96002 object programs
- Fully compatible with the DSP96000CLASx design-in software package
- On-line help screens for each command and DSP96002 register

Dr. BuB Electronic Bulletin Board

Dr. BuB is an electronic bulletin board providing free source code for a large variety of topics that can be used to develop applications with Motorola DSP products. In addition, the latest product information and documentation (including information on new products and improvements on existing products) is posted. Questions concerning Motorola DSP products posted on Dr. BuB are answered promptly. Access to Dr. BuB is through the following phone numbers:

(212A – 300/1200 Baud) (512) 891-DSP1

(V.22 – 1200 Baud) (512) 891-DSP2

(V.22bis – 2400 Baud) (512) 891-DSP3

Format: 7 data bits, even parity, 1 stop bit

User ID=guest

Motorola DSP News

The Motorola DSP News is a quarterly newsletter providing information on new products, application briefs, questions and answers, DSP product information, third-party product news, etc. This newsletter is free and is available upon request by calling the marketing information phone number listed below.

Motorola Field Application Engineers

Information and assistance for DSP applications is available through the local Motorola field office. See your local telephone directory for telephone numbers or call (512)891-2030.

Design Helpline – 1-800-521-6274

This is the Motorola number for information pertaining to any Motorola product.

DSP Applications Helpline – (512) 891-3230

Design assistance for specific DSP applications is available by calling this number.

DSP Marketing Information – (512) 891-2030

Marketing information including brochures, application notes, manuals, price quotes, etc. for Motorola DSP-related products are available by calling this number.

DSP Third-Party Support Information – (512) 891-3098

Information concerning third-party manufacturers using and supporting Motorola DSP products is available by calling this number. Third-party support includes:

- Filter design software
- Logic analyzer support
- Boards for VME, IBM-PC/XT/AT, MACII, SPARC, HP300
- Development systems
- Data conversion cards
- Operating system software
- Debug software

Additional information is available on Dr. BuB and in DSP News.

DSP University Support – (512) 891-3098

Information concerning university support programs and university discounts for all Motorola DSP products is available by calling this number.

DSP Training Courses – (602) 897-3665

There is a course available for the DSP96002:

- Introduction to the DSP96002 (MTT34) which is a four-day instructor-led course and laboratory covering the details of the DSP96002 architecture and programming.

Additional information is available by writing:

Motorola SPS Training and Technical Operations
Mail Drop HW68
P. O. Box 21007
Phoenix, Arizona 85036

or by calling the number above. A technical training catalog is available which describes these courses and gives the current training schedule and prices.

Reference Books and Manuals

A list of DSP-related books is included here as an aid for the engineer who is new to the field of DSP. This is a partial list of DSP references intended to help the new user find useful information in some of the many areas of DSP applications. Many books could be included in several categories but are not repeated.

General DSP:

ADVANCED TOPICS IN SIGNAL PROCESSING
Jae S. Lim and Alan V. Oppenheim
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

APPLICATIONS OF DIGITAL SIGNAL PROCESSING
A. V. Oppenheim
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978

DISCRETE-TIME SIGNAL PROCESSING
A. V. Oppenheim and R. W. Schaffer
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989

DIGITAL PROCESSING OF SIGNALS THEORY AND PRACTICE
Maurice Bellanger
New York, NY: John Wiley and Sons, 1984

DIGITAL SIGNAL PROCESSING
Alan V. Oppenheim and Ronald W. Schaffer
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975

DIGITAL SIGNAL PROCESSING: A SYSTEM DESIGN APPROACH
David J. DeFatta, Joseph G. Lucas, and William S. Hodgkiss

New York, NY: John Wiley and Sons, 1988

FOUNDATIONS OF DIGITAL SIGNAL PROCESSING AND DATA ANALYSIS

J. A. Cadzow

New York, NY: MacMillan Publishing Company, 1987

HANDBOOK OF DIGITAL SIGNAL PROCESSING

D. F. Elliott

San Diego, CA: Academic Press, Inc., 1987

INTRODUCTION TO DIGITAL SIGNAL PROCESSING

John G. Proakis and Dimitris G. Manolakis

New York, NY: Macmillan Publishing Company, 1988

MULTIRATE DIGITAL SIGNAL PROCESSING

R. E. Crochiere and L. R. Rabiner

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1983

SIGNAL PROCESSING ALGORITHMS

S. Stearns and R. Davis

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

SIGNAL PROCESSING HANDBOOK

C.H. Chen

New York, NY: Marcel Dekker, Inc., 1988

SIGNAL PROCESSING – THE MODERN APPROACH

James V. Candy

New York, NY: McGraw-Hill Company, Inc., 1988

THEORY AND APPLICATION OF DIGITAL SIGNAL PROCESSING

Rabiner, Lawrence R., Gold and Bernard

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975

Digital Audio and Filters:

ADAPTIVE FILTER AND EQUALIZERS

B. Mulgrew and C. Cowan

Higham, MA: Kluwer Academic Publishers, 1988

ADAPTIVE SIGNAL PROCESSING

B. Widrow and S. D. Stearns

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985

ART OF DIGITAL AUDIO, THE

John Watkinson

Stoneham, MA: Focal Press, 1988

DESIGNING DIGITAL FILTERS

Charles S. Williams
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

DIGITAL AUDIO SIGNAL PROCESSING AN ANTHOLOGY
John Strawn
William Kaufmann, Inc., 1985

DIGITAL CODING OF WAVEFORMS
N. S. Jayant and Peter Noll
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL FILTERS: ANALYSIS AND DESIGN
Andreas Antoniou
New York, NY: McGraw-Hill Company, Inc., 1979

DIGITAL FILTERS AND SIGNAL PROCESSING
Leland B. Jackson
Higham, MA: Kluwer Academic Publishers, 1986

DIGITAL SIGNAL PROCESSING
Richard A. Roberts and Clifford T. Mullis
New York, NY: Addison-Welsey Publishing Company, Inc., 1987

INTRODUCTION TO DIGITAL SIGNAL PROCESSING
Roman Kuc
New York, NY: McGraw-Hill Company, Inc., 1988

INTRODUCTION TO ADAPTIVE FILTERS
Simon Haykin
New York, NY: MacMillan Publishing Company, 1984

MUSICAL APPLICATIONS OF MICROPROCESSORS (Second Edition)
H. Chamberlin
Hasbrouck Heights, NJ: Hayden Book Co., 1985

C Programming Language:

C: A REFERENCE MANUAL
Samuel P. Harbison and Guy L. Steele
Prentice-Hall Software Series, 1987.

PROGRAMMING LANGUAGE - C
American National Standards Institute,
ANSI Document X3.159-1989
American National Standards Institute, inc., 1990

THE C PROGRAMMING LANGUAGE
Brian W. Kernighan, and Dennis M. Ritchie
Prentice-Hall, Inc., 1978.

Controls:**ADAPTIVE CONTROL**

K. Astrom and B. Wittenmark

New York, NY: Addison-Welsey Publishing Company, Inc., 1989

ADAPTIVE FILTERING PREDICTION & CONTROL

G. Goodwin and K. Sin

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

AUTOMATIC CONTROL SYSTEMS

B. C. Kuo

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987

COMPUTER CONTROLLED SYSTEMS: THEORY & DESIGN

K. Astrom and B. Wittenmark

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL CONTROL SYSTEMS

B. C. Kuo

New York, NY: Holt, Reinholt, and Winston, Inc., 1980

DIGITAL CONTROL SYSTEM ANALYSIS & DESIGN

C. Phillips and H. Nagle

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

ISSUES IN THE IMPLEMENTATION OF DIGITAL FEEDBACK COMPENSATORS

P. Moroney

Cambridge, MA: The MIT Press, 1983

Graphics:**CGM AND CGI**

D. B. Arnold and P. R. Bono

New York, NY: Springer-Verlag, 1988

COMPUTER GRAPHICS (Second Edition)

D. Hearn and M. Pauline Baker

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986

FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS

J. D. Foley and A. Van Dam

Reading MA: Addison-Wesley Publishing Company Inc., 1984

GEOMETRIC MODELING

Michael E. Morteson

New York, NY: John Wiley and Sons, Inc.

GKS THEORY AND PRACTICE

P. R. Bono and I. Herman (Eds.)
New York, NY: Springer-Verlag, 1987

ILLUMINATION AND COLOR IN COMPUTER GENERATED IMAGERY
Roy Hall
New York, NY: Springer-Verlag

POSTSCRIPT LANGUAGE PROGRAM DESIGN
Glenn C. Reid - Adobe Systems, Inc.
Reading MA: Addison-Wesley Publishing Company, Inc., 1988

MICROCOMPUTER DISPLAYS, GRAPHICS, AND ANIMATION
Bruce A. Artwick
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985

PRINCIPLES OF INTERACTIVE COMPUTER GRAPHICS
William M. Newman and Roger F. Sproull
New York, NY: McGraw-Hill Company, Inc., 1979

PROCEDURAL ELEMENTS FOR COMPUTER GRAPHICS
David F. Rogers
New York, NY: McGraw-Hill Company, Inc., 1985

RENDERMAN INTERFACE, THE
Pixar
San Rafael, CA. 94901

Image Processing:

DIGITAL IMAGE PROCESSING
William K. Pratt
New York, NY: John Wiley and Sons, 1978

DIGITAL IMAGE PROCESSING (Second Edition)
Rafael C. Gonzales and Paul Wintz
Reading, MA: Addison-Wesley Publishing Company, Inc., 1977

DIGITAL IMAGE PROCESSING TECHNIQUES
M. P. Ekstrom
New York, NY: Academic Press, Inc., 1984

DIGITAL PICTURE PROCESSING
Azriel Rosenfeld and Avinash C. Kak
New York, NY: Academic Press, Inc., 1982

SCIENCE OF FRACTAL IMAGES, THE
M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H. O. Peitgen,
D. Saupe, and R. F. Voss
New York, NY: Springer-Verlag

Motorola DSP Manuals:

MOTOROLA DSP96000 LINKER/LIBRARIAN REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP96000 MACRO ASSEMBLER REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP96000 SIMULATOR REFERENCE MANUAL
Motorola, Inc., 1991.

MOTOROLA DSP96000 USER'S MANUAL
Motorola, Inc., 1990.

Numerical Methods:

ALGORITHMS (THE CONSTRUCTION, PROOF, AND ANALYSIS OF
PROGRAMS)
P. Berliout and P. Bizard
New York, NY: John Wiley and Sons, 1986

MATRIX COMPUTATIONS
G. H. Golub and C. F. Van Loan
John Hopkins Press, 1983

NUMERICAL RECIPES IN C - THE ART OF SCIENTIFIC PROGRAMMING
William H. Press, Brian P. Flannery,
Saul A. Teukolsky, and William T. Vetterling
Cambridge University Press, 1988

NUMBER THEORY IN SCIENCE AND COMMUNICATION
Manfred R. Schroeder
New York, NY: Springer-Verlag, 1986

Pattern Recognition:

PATTERN CLASSIFICATION AND SCENE ANALYSIS
R. O. Duda and P. E. Hart
New York, NY: John Wiley and Sons, 1973

CLASSIFICATION ALGORITHMS
Mike James
New York, NY: Wiley-Interscience, 1985
Spectral Analysis:

STATISTICAL SPECTRAL ANALYSIS, A NONPROBABILISTIC THEORY
William A. Gardner
Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS

E. Oran Brigham

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988

THE FAST FOURIER TRANSFORM AND ITS APPLICATIONS

R. N. Bracewell

New York, NY: McGraw-Hill Company, Inc., 1986

Speech:

ADAPTIVE FILTERS – STRUCTURES, ALGORITHMS, AND APPLICATIONS

Michael L. Honig and David G. Messerschmitt

Higham, MA: Kluwer Academic Publishers, 1984

DIGITAL CODING OF WAVEFORMS

N. S. Jayant and P. Noll

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984

DIGITAL PROCESSING OF SPEECH SIGNALS

Lawrence R. Rabiner and R. W. Schafer

Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978

LINEAR PREDICTION OF SPEECH

J. D. Markel and A. H. Gray, Jr.

New York, NY: Springer-Verlag, 1976

SPEECH ANALYSIS, SYNTHESIS, AND PERCEPTION

J. L. Flanagan

New York, NY: Springer-Verlag, 1972

SPEECH COMMUNICATION – HUMAN AND MACHINE

D. O'Shaughnessy

Reading, MA: Addison-Wesley Publishing Company, Inc., 1987

Telecommunications:

DIGITAL COMMUNICATION

Edward A. Lee and David G. Messerschmitt

Higham, MA: Kluwer Academic Publishers, 1988

DIGITAL COMMUNICATIONS

John G. Proakis

New York, NY: McGraw-Hill Publishing Co., 1983

Appendix E

GNU Debugger (GDB)

Introduction

Appendix E contains the GNU Debugger Manual. This preface to the GDB manual describes the differences between the standard gdb source level debugger and the gdb96 debugger that uses the Motorola DSP96002 simulator.

A short description of the way the simulator is embedded in gdb96 will help explain some of the differences. The DSP96002 device is simulated by the same non-display version of the simulator program that is supplied with the DSP96000 Development Software. The simulator is linked with the gdb program. When a program is executed on the simulated DSP96002, it is done via direct function calls to the simulator library functions rather than via ptrace calls to an inferior process as in other versions of gdb.

Commands Not Implemented

1. The attach and detach commands are not implemented because the simulator program is linked directly to the gdb96 program. As a result, there is no ability to attach or release a separately running simulation.
2. The core-file command is not implemented at this time.

All other commands work as documented in the GDB Manual for the gnu Source-Level Debugger, Third Edition, October 1989. Some enhancements, listed below, were necessary to accommodate operation using the DSP96002.

DSP96002 Differences / Specific Requirements

1. Since the DSP96002 has three memory spaces, it will sometimes be necessary to use one of the prefixes "p:", "x:", or "y:" in front of an address in order to examine or modify the specific memory space. The debugger symbol information already has the proper memory space designator accompanying each symbol address so symbolic names **do not** require a memory space prefix; however, addresses entered as numeric constants **require** the prefix for x or y memory. **P memory is the default.**

2. The gdb96 debugger operates with COFF format “.cld” files generated by the DSP96000 COFF Assembler and Linker programs. The gdb96 subdirectory “m96kinc” contains header files which define the COFF structures used by gdb96 as well as other device-specific header files.
3. The gdb96 evaluator and display routines have been enhanced to handle the 96 bit long values and special format floating-point values used by the DSP96002.
4. The simulation is able to halt at breakpoints without actually breaking the pipeline activity of the device and without actually inserting breakpoint code into the device memory. As a result, the simulator can maintain an accurate record of the device execution time regardless of the number of inserted breakpoints. The cycle count is maintained in the variable \$cyc.
5. The normal gdb program uses .gdbinit as the default initialization file at start-up whereas gdb96 uses .gdbinit96.

GDB Manual

The GNU Source-Level Debugger

Third Edition, GDB version 3.4
October 1989

Richard M. Stallman
Copyright © 1988, 1989 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled “GNU General Public License” is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled “GNU General Public License” may be included in a translation approved by the author instead of in the original English.

Summary of GDB

The purpose of a debugger such as GDB is to allow you to execute another program while examining what is going on inside it. We call the other program “your program” or “the program being debugged”.

GDB can do four kinds of things (plus other things in support of these):

1. Start the program, specifying anything that might affect its behavior.
2. Make the program stop on specified conditions.
3. Examine what has happened, when the program has stopped, so that you can see bugs happen.
4. Change things in the program, so you can correct the effects of one bug and go on to learn about another without having to re-compile first.

GDB can be used to debug programs written in C and C++. Pascal support is being implemented, and Fortran support will be added when a GNU Fortran compiler is written.

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Copyright ©1989 Free Software Foundation, Inc.
675 Mass Ave., Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software---to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as “you”.
2. You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.
3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
 - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
 - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
 - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
 - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
 - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
 - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
 - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

8. Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

1. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
2. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program’s name and a brief idea of what it does.

Copyright (C) 19yy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave., Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.

This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items---whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program ‘Gnomovision’ (a program to direct compilers to make passes at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

That’s all there is to it!

1.

Input and Output Conventions

GDB is invoked with the shell command 'gdb'. Once started, it reads commands from the terminal until you tell it to exit.

A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name. For example, the command **step** accepts an argument which is the number of times to step, as in **step 5**. You can also use the step command with no arguments. Some command names do not allow any arguments.

GDB command names may always be abbreviated if the abbreviation is unambiguous. Sometimes even ambiguous abbreviations are allowed; for example, 's' is specially defined as equivalent to step even though there are other commands whose names start with 's'. Possible command abbreviations are often stated in the documentation of the individual commands.

A blank line as input to GDB means to repeat the previous command verbatim. Certain commands do not allow themselves to be repeated this way; these are commands for which unintentional repetition might cause trouble and which you are unlikely to want to repeat. Certain others (list and 'x') act differently when repeated because that is more useful.

A line of input starting with '#' is a comment; it does nothing. This is useful mainly in command files (See section Command Files).

GDB indicates its readiness to read a command by printing a string called the prompt. This string is normally '(gdb)'. You can change the prompt string with the 'set prompt' command. For instance, when debugging GDB with GDB, it is useful to change the prompt in one of the GDBs so that you tell which one you are talking to.

set prompt newprompt

Directs GDB to use newprompt as its prompt string henceforth.

To exit GDB, use the quit command (abbreviated 'q'). Ctrl-c will not exit from GDB, but rather will terminate the action of any GDB command that is in progress and return to GDB command level. It is safe to type Ctrl-c at any time because GDB does not allow it to take effect until a time when it is safe.

Certain commands to GDB may produce large amounts of information output to the screen. To help you read all of it, GDB pauses and asks you for input at the end of each page of output. Type RET when you want to continue the output. Normally GDB knows the size of the screen from on the termcap data base together with the value of the TERM environment variable; if this is not correct, you can override it with the 'set screensize' command:

set screensize lpp**set screensize lpp cpl**

Specify a screen height of lpp lines and (optionally) a width of cpl characters. If you omit cpl, the width does not change.

If you specify a height of zero lines, GDB will not pause during output no matter how long the output is. This is useful if output is to a file or to an editor buffer.

Also, GDB may at times produce more information about its own workings than is of interest to the user. Some of these informational messages can be turned on and off with the 'set verbose' command:

set verbose off

Disables GDB's output of certain informational messages.

set verbose on

Re-enables GDB's output of certain informational messages.

Currently, the messages controlled by 'set verbose' are those which announce that the symbol table for a source file is being read (see section File Commands, in the description of the command 'symbol-file').

2.

Specifying GDB's Files

GDB needs to know the file name of the program to be debugged, both in order to read its symbol table and in order to start the program. To debug a core dump of a previous run, GDB must be told the file name of the core dump.

2.1. Specifying Files with Arguments

The usual way to specify the executable and core dump file names is with two command arguments given when you start GDB. The first argument is used as the file for execution and symbols, and the second argument (if any) is used as the core dump file name. Thus,

`gdb prog core`

specifies 'prog' as the executable program and 'core' as a core dump file to examine. (You do not need to have a core dump file if what you plan to do is debug the program interactively.)

See section Options, for full information on options and arguments for invoking GDB.

2.2. Specifying Files with Commands

Usually you specify the files for GDB to work with by giving arguments when you invoke GDB. But occasionally it is necessary to change to a different file during a GDB session. Or you may run GDB and forget to specify the files you want to use. In these situations the GDB commands to specify new files are useful.

`exec-file filename`

Specify that the program to be run is found in filename. If you do not specify a directory and the file is not found in GDB's working directory, GDB will use the environment variable PATH as a list of directories to search, just as the shell does when looking for a program to run.

symbol-file filename

Read symbol table information from file filename. PATH is searched when necessary. Most of the time you will use both the 'exec-file' and 'symbol-file' commands on the same file.

'symbol-file' with no argument clears out GDB's symbol table.

The 'symbol-file' command does not actually read the symbol table in full right away. Instead, it scans the symbol table quickly to find which source files and which symbols are present. The details are read later, one source file at a time, when they are needed.

The purpose of this two-stage reading strategy is to make GDB start up faster. For the most part, it is invisible except for occasional messages telling you that the symbol table details for a particular source file are being read. (The 'set verbose' command controls whether these messages are printed; see section User Interface).

However, you will sometimes see in backtraces lines for functions in source files whose data has not been read in; these lines omit some of the information, such as argument values, which cannot be printed without full details of the symbol table.

When the symbol table is stored in COFF format, 'symbol-file' does read the symbol table data in full right away. We haven't bothered to implement the two-stage strategy for COFF.

core-file filename

Specify the whereabouts of a core dump file to be used as the "contents of memory". Note that the core dump contains only the writable parts of memory; the read-only parts must come from the executable file.

'core-file' with no argument specifies that no core file is to be used.

Note that the core file is ignored when your program is actually running under GDB. So, if you have been running the program and you wish to debug a core file instead, you must kill the sub-process in which the program is running. To do this, use the 'kill' command (see section Kill Process).

add-file filename address

The 'add-file' command reads additional symbol table

information from the file filename. You would use this when that file has been dynamically loaded into the program that is running. address should be the memory address at which the file has been loaded; GDB cannot figure this out for itself.

The symbol table of the file filename is added to the symbol table originally read with the 'symbol-file' command. You can use the 'add-file' command any number of times; the new symbol data thus read keeps adding to the old. The 'symbol-file' command forgets all the symbol data GDB has read; that is the only time symbol data is forgotten in GDB.

info files

Print the names of the executable and core dump files currently in use by GDB, and the file from which symbols were loaded.

While all three file-specifying commands allow both absolute and relative file names as arguments, GDB always converts the file name to an absolute one and remembers it that way.

The 'symbol-file' command causes GDB to forget the contents of its convenience variables, the value history, and all breakpoints and auto-display expressions. This is because they may contain pointers to the internal data recording symbols and data types, which are part of the old symbol table data being discarded inside GDB.

3.

Compiling Your Program for Debugging

In order to debug a program effectively, you need to ask for debugging information when you compile it. This information in the object file describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the ‘-g’ option when you run the compiler.

The Unix C compiler is unable to handle the ‘-g’ and ‘-O’ options together. This means that you cannot ask for optimization if you ask for debugger information.

The GNU C compiler supports ‘-g’ with or without ‘-O’, making it possible to debug optimized code. We recommend that you always use ‘-g’ whenever you compile a program. You may think the program is correct, but there’s no sense in pushing your luck.

GDB no longer supports the debugging information produced by giving the GNU C compiler the ‘-gg’ option, so do not use this option.

4.

Running Your Program Under GDB

To start your program under GDB, use the 'run' command. The program must already have been specified using the 'exec-file' command or with an argument to GDB (see section Files); what 'run' does is create an inferior process, load the program into it, and set it in motion.

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do before starting the program. (You can change it after starting the program, but such changes do not affect the program unless you start it over again.) This information may be divided into three categories:

The arguments.

You specify the arguments to give the program as the arguments of the 'run' command.

The environment.

The program normally inherits its environment from GDB, but you can use the GDB commands 'set environment' and 'unset environment' to change parts of the environment that will be given to the program.

The working directory.

The program inherits its working directory from GDB. You can set GDB's working directory with the 'cd' command in GDB.

After the 'run' command, the debugger does nothing but wait for your program to stop. See section Stopping.

Note that once your program has been started by the 'run' command, you may evaluate expressions that involve calls to functions in the inferior. See section Expressions. If you wish to evaluate a function simply for its side effects, you may use the 'set' command. See section Assignment.

4.1. Your Program's Arguments

The arguments to your program are specified by the arguments of the 'run' command. They are passed to a shell, which expands wildcard characters and performs redirection of I/O, and thence to the program.

'run' with no arguments uses the same arguments used by the previous 'run'.

The command 'set args' can be used to specify the arguments to be used the next time the program is run. If 'set args' has no arguments, it means to use no arguments the next time the program is run. If you have run your program with arguments and want to run it again with no arguments, this is the only way to do so.

4.2. Your Program's Environment

The environment consists of a set of environment variables and their values. Environment variables conventionally record such things as your user name, your home directory, your terminal type, and your search path for programs to run. Usually you set up environment variables with the shell and they are inherited by all the other programs you run. When debugging, it can be useful to try running the program with different environments without having to start the debugger over again.

info environment varname

Print the value of environment variable varname to be given to your program when it is started. This command can be abbreviated 'i env varname'.

info environment

Print the names and values of all environment variables to be given to your program when it is started. This command can be abbreviated 'i env'.

set environment varname value

set environment varname = value

Sets environment variable varname to value, for your program only, not for GDB itself. value may be any string; the values of environment variables are just strings, and any interpretation is supplied by your program itself. The value parameter is optional; if it is eliminated, the variable is set to a null value. This command can be abbreviated as short as 'set e'.

For example, this command:

set env USER = foo

tells the program, when subsequently run, to assume it is being run on behalf of the user named 'foo'.

delete environment varname

unset environment varname

Remove variable varname from the environment to be passed to your program. This is different from 'set env varname =' because 'delete environment' leaves the variable with no value, which is distinguishable from an empty value. This command can be abbreviated 'd e'.

4.3. Your Program's Working Directory

Each time you start your program with 'run', it inherits its working directory from the current working directory of GDB. GDB's working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in GDB with the 'cd' command.

The GDB working directory also serves as a default for the commands that specify files for GDB to operate on. See section Files.

cd directory

Set GDB's working directory to directory.

pwd

Print GDB's working directory.

4.4. Your Program's Input and Output

By default, the program you run under GDB does input and output to the same terminal that GDB uses.

You can redirect the program's input and/or output using 'sh'-style redirection commands in the 'run' command. For example,

run > outfile

starts the program, diverting its output to the file 'outfile'.

Another way to specify where the program should do input and output is with the 'tty' command. This command accepts a file name as argument, and causes this file to be the default for future 'run' commands. It also resets the controlling terminal for the child pro-

cess, for future 'run' commands. For example,

tty /dev/ttyb

directs that processes started with subsequent 'run' commands default to do input and output on the terminal '/dev/ttyb' and have that as their controlling terminal.

An explicit redirection in 'run' overrides the 'tty' command's effect on input/output redirection, but not its effect on the controlling terminal.

When you use the 'tty' command or redirect input in the 'run' command, only the input for your program is affected. The input for GDB still comes from your terminal.

4.5. Debugging an Already-Running Process

Some operating systems allow GDB to debug an already running process that was started outside of GDB. To do this, you use the 'attach' command instead of the 'run' command.

The 'attach' command requires one argument, which is the process-id of the process you want to debug. (The usual way to find out the process-id of the process is with the ps utility.)

The first thing GDB does after arranging to debug the process is to stop it. You can examine and modify an attached process with all the GDB commands that ordinarily available when you start processes with 'run'. You can insert breakpoints; you can step and continue; you can modify storage. If you would rather the process continue running, you may use the 'continue' command after attaching GDB to the process.

When you have finished debugging the attached process, you can use the 'detach' command to release it from GDB's control. Detaching the process continues its execution. After the 'detach' command, that process and GDB become completely independent once more, and you are ready to 'attach' another process or start one with 'run'.

If you exit GDB or use the 'run' command while you have an attached process, you kill that process. You will be asked for confirmation if you try to do either of these things.

The 'attach' command is also used to debug a remote machine via a serial connection. See section Attach, for more info.

4.6. Killing the Child Process

kill Kill the child process in which the program being debugged is running under GDB.

This command is useful if you wish to debug a core dump instead. GDB ignores any core dump file if it is actually running the program, so the 'kill' command is the only sure way to make sure the core dump file is used once again.

It is also useful if you wish to run the program outside the debugger for once and then go back to debugging it.

The 'kill' command is also useful if you wish to re-compile and re-link the program, since on many systems it is impossible to modify an executable file which is running in a process. But, in this case, it is just as good to exit GDB, since you will need to read a new symbol table after the program is re-compiled if you wish to debug the new version, and restarting GDB is the easiest way to do that.

5.

Stopping and Continuing

When you run a program normally, it runs until it terminates. The principal purpose of using a debugger is so that you can stop it before that point; or so that if the program runs into trouble you can investigate and find out why.

5.1. Signals

A signal is an asynchronous event that can happen in a program. The operating system defines the possible kinds of signals, and gives each kind a name and a number. For example, SIGINT is the signal a program gets when you type Ctrl-c; SIGSEGV is the signal a program gets from referencing a place in memory far away from all the areas in use; SIGALRM occurs when the alarm clock timer goes off (which happens only if the program has requested an alarm).

Some signals, including SIGALRM, are a normal part of the functioning of the program. Others, such as SIGSEGV, indicate errors; these signals are fatal (kill the program immediately) if the program has not specified in advance some other way to handle the signal. SIGINT does not indicate an error in the program, but it is normally fatal so it can carry out the purpose of Ctrl-c: to kill the program.

GDB has the ability to detect any occurrence of a signal in the program running under GDB's control. You can tell GDB in advance what to do for each kind of signal.

Normally, GDB is set up to ignore non-erroneous signals like SIGALRM (so as not to interfere with their role in the functioning of the program) but to stop the program immediately whenever an error signal happens. You can change these settings with the 'handle' command. You must specify which signal you are talking about with its number.

info signal

Print a table of all the kinds of signals and how GDB has been told to handle each one. You can use this to see the signal numbers of all the defined types of signals.

handle signalnum keywords...

Change the way GDB handles signal signalnum. The keywords say what change to make.

To use the 'handle' command you must know the code number of the signal you are concerned with. To find the code number, type 'info signal' which prints a table of signal names and numbers.

The keywords allowed by the handle command can be abbreviated. Their full names are

stop

GDB should stop the program when this signal happens. This implies the 'print' keyword as well.

print

GDB should print a message when this signal happens.

nostop

GDB should not stop the program when this signal happens. It may still print a message telling you that the signal has come in.

noprint

GDB should not mention the occurrence of the signal at all. This implies the 'nostop' keyword as well.

pass

GDB should allow the program to see this signal; the program will be able to handle the signal, or may be terminated if the signal is fatal and not handled.

nopass

GDB should not allow the program to see this signal.

When a signal has been set to stop the program, the program cannot see the signal until you continue. It will see the signal then, if 'pass' is in effect for the signal in question at that time. In other words, after GDB reports a signal, you can use the 'handle' command with 'pass' or 'nopass' to control whether that signal will be seen by the program when you later continue it.

You can also use the 'signal' command to prevent the program from seeing a signal, or cause it to see a signal it normally would not see, or to give it any signal at any time. See section Signaling.

5.2. Breakpoints

A breakpoint makes your program stop whenever a certain point in the program is reached. You set breakpoints explicitly with GDB commands, specifying the place where the program should stop by line number, function name or exact address in the program. You can add various other conditions to control whether the program will stop.

Each breakpoint is assigned a number when it is created; these numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. Each breakpoint may be enabled or disabled; if disabled, it has no effect on the program until you enable it again.

The command ‘info break’ prints a list of all breakpoints set and not deleted, showing their numbers, where in the program they are, and any special features in use for them. Disabled breakpoints are included in the list, but marked as disabled. ‘info break’ with a breakpoint number as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the ‘x’ command are set to the address of the last breakpoint listed (see section Memory).

5.2.1. Setting Breakpoints

Breakpoints are set with the ‘break’ command (abbreviated ‘b’). You have several ways to say where the breakpoint should go.

break function

Set a breakpoint at entry to function function.

break +offset

break -offset

Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected frame.

break linenum

Set a breakpoint at line linenum in the current source file. That file is the last file whose source text was printed. This breakpoint will stop the program just before it executes any of the code on that line.

break filename:linenum

Set a breakpoint at line linenum in source file filename.

break filename:function

Set a breakpoint at entry to function function found in file filename. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

break *address

Set a breakpoint at address address. You can use this to set breakpoints in parts of the program which do not have debugging information or source files.

break

Set a breakpoint at the next instruction to be executed in the selected stack frame (see section Stack). In any selected frame but the innermost, this will cause the program to stop as soon as control returns to that frame. This is equivalent to a 'finish' command in the frame inside the selected frame. If this is done in the innermost frame, GDB will stop the next time it reaches the current location; this may be useful inside of loops.

GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when the program stopped.

break ... if cond

Set a breakpoint with condition cond; evaluate the expression cond each time the breakpoint is reached, and stop only if the value is nonzero. '...' stands for one of the possible arguments described above (or no argument) specifying where to break. See section Conditions, for more information on breakpoint conditions.

tbreak args

Set a breakpoint enabled only for one stop. args are the same as in the 'break' command, and the breakpoint is set in the same way, but the breakpoint is automatically disabled the first time it is hit. See section Disabling.

GDB allows you to set any number of breakpoints at the same place in the program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see section Conditions).

5.2.2. Deleting Breakpoints

It is often necessary to eliminate a breakpoint once it has done its job and you no longer want the program to stop there. This is called deleting the breakpoint. A breakpoint that has been deleted no longer exists in any sense; it is forgotten.

With the 'clear' command you can delete breakpoints according to where they are in the program. With the 'delete' command you can delete individual breakpoints by specifying their breakpoint numbers.

It is not necessary to delete a breakpoint to proceed past it. GDB automatically ignores breakpoints in the first instruction to be executed when you continue execution without changing the execution address.

clear

Delete any breakpoints at the next instruction to be executed in the selected stack frame (see section Selection). When the innermost frame is selected, this is a good way to delete a breakpoint that the program just stopped at.

clear function

clear filename:function

Delete any breakpoints set at entry to the function function.

clear linenum

clear filename:linenum

Delete any breakpoints set at or within the code of the specified line.

delete bnums...

Delete the breakpoints of the numbers specified as arguments.

5.2.3. Disabling Breakpoints

Rather than deleting a breakpoint, you might prefer to disable it. This makes the breakpoint inoperative as if it had been deleted, but remembers the information on the breakpoint so that you can enable it again later.

You disable and enable breakpoints with the 'enable' and 'disable' commands, specifying one or more breakpoint numbers as arguments. Use 'info break' to print a list of breakpoints if you don't know which breakpoint numbers to use.

A breakpoint can have any of four different states of enablement:

- Enabled. The breakpoint will stop the program. A breakpoint made with the 'break' command starts out in this state.
- Disabled. The breakpoint has no effect on the program.
- Enabled once. The breakpoint will stop the program, but when it does so it will become disabled. A breakpoint made with the 'tbreak' command starts out in this state.

- Enabled for deletion. The breakpoint will stop the program, but immediately after it does so it will be deleted permanently.

You change the state of enablement of a breakpoint with the following commands:

disable breakpoints bnums...

disable bnums...

Disable the specified breakpoints. A disabled breakpoint has no effect but is not forgotten. All options such as ignore-counts, conditions and commands are remembered in case the breakpoint is enabled again later.

enable breakpoints bnums...

enable bnums...

Enable the specified breakpoints. They become effective once again in stopping the program, until you specify otherwise.

enable breakpoints once bnums...

enable once bnums...

Enable the specified breakpoints temporarily. Each will be disabled again the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

enable breakpoints delete bnums...

enable delete bnums...

Enable the specified breakpoints to work once and then die. Each of the breakpoints will be deleted the next time it stops the program (unless you have used one of these commands to specify a different state before that time comes).

Aside from the automatic disablement or deletion of a breakpoint when it stops the program, which happens only in certain states, the state of enablement of a breakpoint changes only when one of the commands above is used.

5.2.4. Break Conditions

The simplest sort of breakpoint breaks every time the program reaches a specified place. You can also specify a condition for a breakpoint. A condition is just a boolean expression in your programming language. (See section Expressions). A breakpoint with a condition evaluates the expression each time the program reaches it, and the program stops only if the condition is true.

Break conditions may have side effects, and may even call functions in your program. These may sound like strange things to do, but their effects are completely predictable unless there is another enabled breakpoint at the same address. (In that case, GDB

might see the other breakpoint first and stop the program without checking the condition of this one.) Note that breakpoint commands are usually more convenient and flexible for the purpose of performing side effects when a breakpoint is reached (see section Break Commands).

Break conditions can be specified when a breakpoint is set, by using 'if' in the arguments to the 'break' command. See section Set Breaks. They can also be changed at any time with the 'condition' command:

condition bnum expression

Specify expression as the break condition for breakpoint number bnum. From now on, this breakpoint will stop the program only if the value of expression is true (nonzero, in C). expression is not evaluated at the time the 'condition' command is given. See section Expressions.

condition bnum

Remove the condition from breakpoint number bnum. It becomes an ordinary unconditional breakpoint.

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. This is so useful that there is a special way to do it, using the ignore count of the breakpoint. Every breakpoint has an ignore count, which is an integer. Most of the time, the ignore count is zero, and therefore has no effect. But if the program reaches a breakpoint whose ignore count is positive, then instead of stopping, it just decrements the ignore count by one and continues. As a result, if the ignore count value is n, the breakpoint will not stop the next n times it is reached.

ignore bnum count

Set the ignore count of breakpoint number bnum to count. The next count times the breakpoint is reached, it will not stop.

To make the breakpoint stop the next time it is reached, specify a count of zero.

cont count

Continue execution of the program, setting the ignore count of the breakpoint that the program stopped at to count minus one. Thus, the program will not stop at this breakpoint until the count'th time it is reached.

This command is allowed only when the program stopped due to a breakpoint. At other times, the argument to 'cont' is ignored.

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, the condition will start to be checked.

Note that you could achieve the effect of the ignore count with a condition such as '\$foo-<= 0' using a debugger convenience variable that is decremented each time. See section Convenience Vars.

5.2.5. Commands Executed on Breaking

You can give any breakpoint a series of commands to execute when the program stops due to that breakpoint. For example, you might want to print the values of certain expressions, or enable other breakpoints.

commands bnum

Specify commands for breakpoint number bnum. The commands themselves appear on the following lines. Type a line containing just 'end' to terminate the commands.

To remove all commands from a breakpoint, use the command 'commands' and follow it immediately by 'end'; that is, give no commands.

With no arguments, 'commands' refers to the last breakpoint set.

It is possible for breakpoint commands to start the program up again. Simply use the 'cont' command, or 'step', or any other command to resume execution. However, any remaining breakpoint commands are ignored. When the program stops again, GDB will act according to the cause of that stop.

If the first command specified is 'silent', the usual message about stopping at a breakpoint is not printed. This may be desirable for breakpoints that are to print a specific message and then continue. If the remaining commands too print nothing, you will see no sign that the breakpoint was reached at all. 'silent' is not really a command; it is meaningful only at the beginning of the commands for a breakpoint.

The commands 'echo' and 'output' that allow you to print precisely controlled output are often useful in silent breakpoints. See section Output.

For example, here is how you could use breakpoint commands to print the value of x at entry to foo whenever it is positive.

```
break foo if x>0
commands
silent
```

```

echo x is\040
output x
echo \n
cont
end

```

One application for breakpoint commands is to correct one bug so you can test another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. End with the 'cont' command so that the program does not stop, and start with the 'silent' command so that no output is produced. Here is an example:

```

break 403
commands
silent
set x = y + 4
cont
end

```

One deficiency in the operation of automatically continuing breakpoints under Unix appears when your program uses raw mode for the terminal. GDB switches back to its own terminal modes (not raw) before executing commands, and then must switch back to raw mode when your program is continued. This causes any pending terminal input to be lost.

In the GNU system, this will be fixed by changing the behavior of terminal modes.

Under Unix, when you have this problem, you might be able to get around it by putting your actions into the breakpoint condition instead of commands. For example

```
condition 5 (x = y + 4), 0
```

specifies a condition expression (See section Expressions) that will change x as needed, then always have the value 0 so the program will not stop. Loss of input is avoided here because break conditions are evaluated without changing the terminal modes. When you want to have nontrivial conditions for performing the side effects, the operators '&&', '||' and '?...:' may be useful.

5.2.6. "Cannot Insert Breakpoints" Error

Under some operating systems, breakpoints cannot be used in a program if any other process is running that program. Attempting to run or continue the program with a breakpoint in this case will cause GDB to stop it.

When this happens, you have three ways to proceed:

1. Remove or disable the breakpoints, then continue.
2. Suspend GDB, and copy the file containing the program to a new name. Resume GDB and use the 'exec-file' command to specify that GDB should run the program under that name. Then start the program again.
3. Re-link the program so that the text segment is non-sharable, using the linker option '-N'. The operating system limitation may not apply to non-sharable executables.

5.3. Continuing

After your program stops, most likely you will want it to run some more if the bug you are looking for has not happened yet.

cont

Continue running the program at the place where it stopped.

If the program stopped at a breakpoint, the place to continue running is the address of the breakpoint. You might expect that continuing would just stop at the same breakpoint immediately. In fact, 'cont' takes special care to prevent that from happening. You do not need to delete the breakpoint to proceed through it after stopping at it.

You can, however, specify an ignore-count for the breakpoint that the program stopped at, by means of an argument to the 'cont' command. See section Conditions.

If the program stopped because of a signal other than SIGINT or SIGTRAP, continuing will cause the program to see that signal. You may not want this to happen. For example, if the program stopped due to some sort of memory reference error, you might store correct values into the erroneous variables and continue, hoping to see more execution; but the program would probably terminate immediately as a result of the fatal signal once it sees the signal. To prevent this, you can continue with 'signal 0'. See section Signaling. You can also act in advance to prevent the program from seeing certain kinds of signals, using the 'handle' command (see section Signals).

5.4. Stepping

Stepping means setting your program in motion for a limited time, so that control will return automatically to the debugger after one line of code or one machine instruction. Breakpoints are active during stepping and the program will stop for them even if it has not gone as far as the stepping command specifies.

step

Continue running the program until control reaches a different line, then stop it and return control to the debugger. This command is abbreviated 's'.

This command may be given when control is within a function for which there is no debugging information. In that case, execution will proceed until control reaches a different function, or is about to return from this function. An argument repeats this action.

step count

Continue running as in 'step', but do so count times. If a breakpoint is reached or a signal not related to stepping occurs before count steps, stepping stops right away.

next

Similar to 'step', but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the 'next' command was given. This command is abbreviated 'n'.

An argument is a repeat count, as in 'step'.

'next' within a function without debugging information acts as does 'step', but any function calls appearing within the code of the function are executed without stopping.

finish

Continue running until just after the selected stack frame returns (or until there is some other reason to stop, such as a fatal signal or a breakpoint). Print value returned by the selected stack frame (if any).

Contrast this with the 'return' command (see section Returning).

until

This command is used to avoid single stepping through a loop more than once. It is like the 'next' command, except that when 'until' encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump.

This means that when you reach the end of a loop after single stepping though it, 'until' will cause the program to continue execution until the loop is exited. In contrast, a 'next' command at the end of a loop will simply step back to the beginning of the loop, which would force you to step through the next iteration.

'until' always stops the program if it attempts to exit the current stack frame.

'until' may produce somewhat counterintuitive results if the order of the source lines does not match the actual order of execution. For example, in a typical C for-loop, the third expression in the for-statement (the loop-step expression) is executed after the statements in the body of the loop, but is written before them. Therefore, the 'until' command would appear to step back to the beginning of the loop when it advances to this expression. However, it has not really done so, not in terms of the actual machine code.

Note that 'until' with no argument works by means of single instruction stepping, and hence is slower than 'until' with an argument.

until location

Continue running the program until either the specified location is reached, or the current (innermost) stack frame returns. This form of the command uses breakpoints, and hence is quicker than 'until' without an argument.

stepi si

Execute one machine instruction, then stop and return to the debugger.

It is often useful to do 'display/i \$pc' when stepping by machine instructions. This will cause the next instruction to be executed to be displayed automatically at each stop. See section Auto Display.

An argument is a repeat count, as in 'step'.

nexti ni

Execute one machine instruction, but if it is a subroutine call, proceed until the subroutine returns.

An argument is a repeat count, as in 'next'.

A typical technique for using stepping is to put a breakpoint (see section Breakpoints) at the beginning of the function or the section of the program in which a problem is believed to lie, and then step through the suspect area, examining the variables that are interesting, until the problem happens.

The 'cont' command can be used after stepping to resume execution until the next breakpoint or signal.

6.

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there.

Each time your program performs a function call, the information about where in the program the call was made from is saved in a block of data called a stack frame. The frame also contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of memory called the call stack.

When your program stops, the GDB commands for examining the stack allow you to see all of this information.

One of the stack frames is selected by GDB and many GDB commands refer implicitly to the selected frame. In particular, whenever you ask GDB for the value of a variable in the program, the value is found in the selected frame. There are special GDB commands to select whichever frame you are interested in.

When the program stops, GDB automatically selects the currently executing frame and describes it briefly as the ‘frame’ command does (see section Frame Info, Info).

6.1. Stack Frames

The call stack is divided up into contiguous pieces called stack frames, or frames for short; each frame is the data associated with one call to one function. The frame contains the arguments given to the function, the function’s local variables, and the address at which the function is executing.

When your program is started, the stack has only one frame, that of the function main. This is called the initial frame or the outermost frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the innermost frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes, each of which has its own address; each kind of computer has a convention for choosing one of those bytes whose address serves as the address of the frame. Usually this address is kept in a register called the frame pointer register while execution is going on in that frame.

GDB assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on upward. These numbers do not really exist in your program; they are to give you a way of talking about stack frames in GDB commands.

Many GDB commands refer implicitly to one stack frame. GDB records a stack frame that is called the selected stack frame; you can select any frame using one set of GDB commands, and then other commands will operate on that frame. When your program stops, GDB automatically selects the innermost frame.

Some functions can be compiled to run without a frame reserved for them on the stack. This is occasionally done with heavily used library functions to save the frame setup time. GDB has limited facilities for dealing with these function invocations; if the innermost function invocation has no stack frame, GDB will give it a virtual stack frame of 0 and correctly allow tracing of the function call chain. Results are undefined if a function invocation besides the innermost one is frameless.

6.2. Backtraces

A backtrace is a summary of how the program got where it is. It shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack.

backtrace bt

Print a backtrace of the entire stack: one line per frame for all frames in the stack.

You can stop the backtrace at any time by typing the system interrupt character, normally Control-C.

backtrace n bt n

Similar, but print only the innermost n frames.

backtrace -n

bt -n

Similar, but print only the outermost n frames.

The names ‘where’ and ‘info stack’ are additional aliases for ‘backtrace’.

Every line in the backtrace shows the frame number, the function name and the program counter value.

If the function is in a source file whose symbol table data has been fully read, the backtrace shows the source file name and line number, as well as the arguments to the function. (The program counter value is omitted if it is at the beginning of the code for that line number.)

If the source file’s symbol data has not been fully read, just scanned, this extra information is replaced with an ellipsis. You can force the symbol data for that frame’s source file to be read by selecting the frame. (See section Selection).

Here is an example of a backtrace. It was made with the command ‘bt 3’, so it shows the innermost three frames.

```
#0 rtx_equal_p (x=(rtx) 0x8e58c, y=(rtx) 0x1086c4) (/gp/rms/cc/rtlanal.c
line 337)
#1 0x246b0 in expand_call (...) (...)
#2 0x21cfc in expand_expr (...) (...)
(More stack frames follow...)
```

The functions `expand_call` and `expand_expr` are in a file whose symbol details have not been fully read. Full detail is available for the function `rtx_equal_p`, which is in the file ‘rtlanal.c’. Its arguments, named `x` and `y`, are shown with their typed values.

6.3. Selecting a Frame

Most commands for examining the stack and other data in the program work on whichever stack frame is selected at the moment. Here are the commands for selecting a stack frame; all of them finish by printing a brief description of the stack frame just selected.

frame n

Select frame number `n`. Recall that frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost one, and so on. The highest-numbered frame is `main`’s frame.

frame addr

Select the frame at address `addr`. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for GDB to assign num-

bers properly to all frames. In addition, this can be useful when the program has multiple stacks and switches between them.

up n

Select the frame *n* frames up from the frame previously selected. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers, to frames that have existed longer. *n* defaults to one.

down n

Select the frame *n* frames down from the frame previously selected. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers, to frames that were created more recently. *n* defaults to one.

All of these commands end by printing some information on the frame that has been selected: the frame number, the function name, the arguments, the source file and line number of execution in that frame, and the text of that source line. For example:

```
#3 main (argc=3, argv=??, env=??) at main.c, line 67
67 read_input_file (argv[i]);
```

After such a printout, the 'list' command with no arguments will print ten lines centered on the point of execution in the frame. See section List.

6.4. Information on a Frame

There are several other commands to print information about the selected stack frame.

frame

This command prints a brief description of the selected stack frame. It can be abbreviated 'f'. With an argument, this command is used to select a stack frame; with no argument, it does not change which frame is selected, but still prints the same information.

info frame

This command prints a verbose description of the selected stack frame, including the address of the frame, the addresses of the next frame in (called by this frame) and the next frame out (caller of this frame), the address of the frame's arguments, the program counter saved in it (the address of execution in the caller frame), and which registers were saved in the frame. The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.

info frame addr

Print a verbose description of the frame at address `addr`, without selecting that frame. The selected frame remains unchanged by this command.

info args

Print the arguments of the selected frame, each on a separate line.

info locals

Print the local variables of the selected frame, each on a separate line. These are all variables declared static or automatic within all program blocks that execution in this frame is currently inside of.

7.

Examining Source Files

GDB knows which source files your program was compiled from, and can print parts of their text. When your program stops, GDB spontaneously prints the line it stopped in. Likewise, when you select a stack frame (see section Selection), GDB prints the line which execution in that frame has stopped in. You can also print parts of source files by explicit command.

7.1. Printing Source Lines

To print lines from a source file, use the ‘list’ command (abbreviated ‘l’). There are several ways to specify what part of the file you want to print.

Here are the forms of the ‘list’ command most commonly used:

list linenum

Print ten lines centered around line number linenum in the current source file.

list function

Print ten lines centered around the beginning of function function.

list

Print ten more lines. If the last lines printed were printed with a ‘list’ command, this prints ten lines following the last lines printed; however, if the last line printed was a solitary line printed as part of displaying a stack frame (see section Stack), this prints ten lines centered around that line.

list -

Print ten lines just before the lines last printed.

Repeating a ‘list’ command with RET discards the argument, so it is equivalent to typing just ‘list’. This is more useful than listing the same lines again. An exception is made for an argument of ‘-’; that argument is preserved in repetition so that each repetition moves up in the file.

In general, the 'list' command expects you to supply zero, one or two linespecs. Linespecs specify source lines; there are several ways of writing them but the effect is always to specify some source line. Here is a complete description of the possible arguments for 'list':

list linespec

Print ten lines centered around the line specified by linespec.

list first,last

Print lines from first to last. Both arguments are linespecs.

list ,last

Print ten lines ending with last.

list first,

Print ten lines starting with first.

list +

Print ten lines just after the lines last printed.

list -

Print ten lines just before the lines last printed.

list

As described in the preceding table.

Here are the ways of specifying a single source line--all the kinds of linespec.

linenum

Specifies line linenum of the current source file. When a 'list' command has two linespecs, this refers to the same source file as the first linespec.

+offset

Specifies the line offset lines after the last line printed. When used as the second linespec in a 'list' command that has two, this specifies the line offset lines down from the first linespec.

-offset

Specifies the line offset lines before the last line printed.

filename:linenum

Specifies line linenum in the source file filename.

function

Specifies the line of the open-brace that begins the body of the function function.

filename:function

Specifies the line of the open-brace that begins the body of the function function in the file filename. The file name is needed with a function name only for disambiguation of identically named functions in different source files.

***address**

Specifies the line containing the program address address. address may be any expression.

One other command is used to map source lines to program addresses.

info line linenum

Print the starting and ending addresses of the compiled code for source line linenum.

The default examine address for the 'x' command is changed to the starting address of the line, so that 'x/i' is sufficient to begin examining the machine code (see section Memory). Also, this address is saved as the value of the convenience variable \$_ (see section Convenience Vars).

7.2. Searching Source Files

There are two commands for searching through the current source file for a regular expression.

The command 'forward-search regexp' checks each line, starting with the one following the last line listed, for a match for regexp. It lists the line that is found. You can abbreviate the command name as 'fo'.

The command 'reverse-search regexp' checks each line, starting with the one before the last line listed and going backward, for a match for regexp. It lists the line that is found. You can abbreviate this command with as little as 'rev'.

7.3. Specifying Source Directories

Executable programs do not record the directories of the source files from which they were compiled, just the names. GDB remembers a list of directories to search for source files; this is called the source path. Each time GDB wants a source file, it tries all the directories in the list, in the order they are present in the list, until it finds a file with the desired name. **Note that the executable search path is not used for this purpose. Neither is the current working directory, unless it happens to be in the source path.**

When you start GDB, its source path contains just the current working directory. To add other directories, use the 'directory' command.

directory dirnames...

Add directory dirname to the end of the source path. Several directory names may be given to this command, separated by whitespace or ':'.

directory

Reset the source path to just the current working directory of GDB. This requires confirmation.

Since this command deletes directories from the search path, it may change the directory in which a previously read source file will be discovered. To make this work correctly, this command also clears out the tables GDB maintains about the source files it has already found.

info directories

Print the source path: show which directories it contains.

Because the 'directory' command adds to the end of the source path, it does not affect any file that GDB has already found. If the source path contains directories that you do not want, and these directories contain misleading files with names matching your source files, the way to correct the situation is as follows:

1. Choose the directory you want at the beginning of the source path. Use the 'cd' command to make that the current working directory.
2. Use 'directory' with no argument to reset the source path to just that directory.
3. Use 'directory' with suitable arguments to add any other directories you want in the source path.

8. Examining Data

The usual way to examine data in your program is with the 'print' command (abbreviated 'p'). It evaluates and prints the value of any valid expression of the language the program is written in (for now, C). You type

print exp

where exp is any valid expression, and the value of exp is printed in a format appropriate to its data type.

A more low-level way of examining data is with the 'x' command. It examines data in memory at a specified address and prints it in a specified format.

8.1. Expressions

Many different GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is legal in an expression in GDB. This includes conditional expressions, function calls, casts and string constants. It unfortunately does not include symbols defined by preprocessor #define commands.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer so as to examine a structure at that address in memory.

GDB supports three kinds of operator in addition to those of programming languages:

- @ ' @ ' is a binary operator for treating parts of memory as arrays. See section Arrays, for more information.
- :: '::' allows you to specify a variable in terms of the file or function it is defined in. See section Variables.

{type} addr Refers to an object of type type stored at address addr in memory. addr may be any expression whose value is an integer or pointer (but parentheses are required around non-unary operators, just as in a cast). This construct is allowed regardless of what kind of data is officially supposed to reside at addr.

8.2. Program Variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see section Selection); they must either be global (or static) or be visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

```
foo (a)
int a;
{
    bar (a);
    {
        int b = test ();
        bar (b);
    }
}
```

the variable `a` is usable whenever the program is executing within the function `foo`, but the variable `b` is visible only while the program is executing inside the block in which `b` is declared.

As a special exception, you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name (if they are in different source files). In such a case, it is not defined which one you will get. If you wish, you can specify any one of them using the colon colon construct:

block::variable

Here `block` is the name of the source file whose variable you want.

8.3. Artificial Arrays

It is often useful to print out several successive objects of the same type in memory; a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

This can be done by constructing an artificial array with the binary operator '@'. The left operand of '@' should be the first element of the desired array, as an individual object. The right operand should be the length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of array with

```
p *array@len
```

The left operand of '@' must reside in memory. Array values made with '@' in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. (It would probably appear in an expression via the value history, after you had printed it out.)

8.4. Format options

GDB provides a few ways to control how arrays and structures are printed.

info format

Display the current settings for the format options.

set array-max number-of-elements

If GDB is printing a large array, it will stop printing after it has printed the number of elements set by the 'set array-max' command. This limit also applies to the display of strings.

set prettyprint on

Cause GDB to print structures in an indented format with one member per line, like this:

```
$1 = {
    next = 0x0,
    flags = {
        sweet = 1,
        sour = 1
    },
    meat = 0x54 "Pork"
}
```

set prettyprint off

Cause GDB to print structures in a compact format, like this:

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}
```

This is the default format.

set unionprint on

Tell GDB to print unions which are contained in structures. This is the default setting.

set unionprint off

Tell GDB not to print unions which are contained in structures.

For example, given the declarations

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly} Bug_forms;
struct thing {
    Species it;
    union {
        Tree_forms tree;
        Bug_forms bug;
    } form;
};
struct thing foo = {Tree, {Acorn}};
```

with 'set unionprint on' in effect 'p foo' would print

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with 'set unionprint off' in effect it would print

```
$1 = {it = Tree, form = {...}}
```

8.5. Output formats

GDB normally prints all values according to their data types. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or an instruction. These things can be done with output formats.

The simplest use of output formats is to say how to print a value already computed. This is done by starting the arguments of the 'print' command with a slash and a format letter. The format letters supported are:

- 'x' Regard the bits of the value as an integer, and print the integer in hexadecimal.
- 'd' Print as integer in signed decimal.
- 'u' Print as integer in unsigned decimal.
- 'o' Print as integer in octal.
- 'a' Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.

- 'c'** Regard as an integer and print it as a character constant.
- 'f'** Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see section Registers), type

p/x \$pc

Note that no space is required before the slash; this is because command names in GDB cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the 'print' command with just a format and no expression. For example, 'p/x' reprints the last value in hex.

8.5.1. Examining Memory

The command 'x' (for 'examine') can be used to examine memory without reference to the program's data types. The format in which you wish to examine memory is instead explicitly specified. The allowable formats are a superset of the formats described in the previous section.

'x' is followed by a slash and an output format specification, followed by an expression for an address. The expression need not have a pointer value (though it may); it is used as an integer, as the address of a byte of memory. See section Expressions for more information on expressions. For example, 'x/4xw \$sp' prints the four words of memory above the stack pointer in hexadecimal.

The output format in this case specifies both how big a unit of memory to examine and how to print the contents of that unit. It is done with one or two of the following letters:

These letters specify just the size of unit to examine:

- 'b'** Examine individual bytes.
- 'h'** Examine halfwords (two bytes each).
- 'w'** Examine words (four bytes each).

Many assemblers and cpu designers still use 'word' for a 16-bit quantity, as a holdover from specific predecessor machines of the 1970's that really did use two byte words. But more generally the term 'word' has always referred to the size of quantity that a machine normally operates on and stores in its registers. This is 32 bits for all the machines that GDB runs on.

- 'g'** Examine giant words (8 bytes).

These letters specify just the way to print the contents:

- 'x'** Print as integers in unsigned hexadecimal.
- 'd'** Print as integers in signed decimal.
- 'u'** Print as integers in unsigned decimal.
- 'o'** Print as integers in unsigned octal.
- 'a'** Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.
- 'c'** Print as character constants.
- 'f'** Print as floating point. This works only with sizes 'w' and 'g'.
- 's'** Print a null-terminated string of characters. The specified unit size is ignored; instead, the unit is however many bytes it takes to reach a null character (including the null character).
- 'i'** Print a machine instruction in assembler syntax (or nearly). The specified unit size is ignored; the number of bytes in an instruction varies depending on the type of machine, the opcode and the addressing modes used.

If either the manner of printing or the size of unit fails to be specified, the default is to use the same one that was used last. If you don't want to use any letters after the slash, you can omit the slash as well.

You can also omit the address to examine. Then the address used is just after the last unit examined. This is why string and instruction formats actually compute a unit size based on the data: so that the next string or instruction examined will start in the right place. The 'print' command sometimes sets the default address for the 'x' command; when the value printed resides in memory, the default is set to examine the same location. 'info line' also sets the default for 'x', to the address of the start of the machine code for the specified line and 'info breakpoints' sets it to the address of the last breakpoint listed.

When you use RET to repeat an 'x' command, it does not repeat exactly the same: the address specified previously (if any) is ignored, so that the repeated command examines the successive locations in memory rather than the same ones.

You can examine several consecutive units of memory with one command by writing a repeat-count after the slash (before the format letters, if any). The repeat count must be a decimal integer. It has the same effect as repeating the 'x' command that many times except that the output may be more compact with several units per line. For example,

```
x/10i $pc
```

prints ten instructions starting with the one to be executed next in the selected frame. After doing this, you could print another ten following instructions with

```
x/10
```

in which the format and address are allowed to default.

The addresses and contents printed by the 'x' command are not put in the value history because there is often too much of them and they would get in the way. Instead, GDB makes these values available for subsequent use in expressions as values of the convenience variables \$_ and \$__.

After an 'x' command, the last address examined is available for use in expressions in the convenience variable \$_. The contents of that address, as examined, are available in the convenience variable \$__.

If the 'x' command has a repeat count, the address and contents saved are from the last memory unit printed; this is not the same as the last address printed if several units were printed on the last line of output.

The specialized command 'disassemble' is also provided to dump a range of memory as machine instructions. The default memory range is the function surrounding the program counter of the selected frame. A single argument to this command is a program counter value; the function surrounding this value will be dumped. Two arguments specify a range of address (first inclusive, second exclusive) to be dumped.

8.6. Automatic Display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the automatic display list so that GDB will print its value each time the program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
```

```
3: bar[5] = (struct hack *) 0x3804
```

showing item numbers, expressions and their current values.

If the expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is printed only when execution is inside that lexical context. For example, if you give the command 'display name' while inside a function with an argument name, then this argument will be displayed whenever the program stops inside that function, but not when it stops elsewhere (since this argument doesn't exist elsewhere).

display exp

Add the expression exp to the list of expressions to display each time the program stops. See section Expressions.

display/fmt exp

For fmt specifying only a display format and not a size or count, add the expression exp to the auto-display list but arranges to display it each time in the specified format fmt.

display/fmt addr

For fmt 'i' or 's', or including a unit-size or a number of units, add the expression addr as a memory address to be examined each time the program stops. Examining means in effect doing 'x/fmt addr'. See section Memory.

undisplay dnums...**delete display dnums...**

Remove item numbers dnums from the list of expressions to display.

disable display dnums...

Disable the display of item numbers dnums. A disabled display item is not printed automatically, but is not forgotten. It may be re-enabled later.

enable display dnums...

Enable display of item numbers dnums. It becomes effective once again in auto display of its expression, until you specify otherwise.

display

Display the current values of the expressions on the list, just as is done when the program stops.

info display

Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

8.7. Value History

Every value printed by the 'print' command is saved for the entire session in GDB's value history so that you can refer to it in other expressions.

The values printed are given history numbers for you to refer to them by. These are successive integers starting with 1. 'print' shows you the history number assigned to a value by printing '\$num = ' before the value; here num is the history number.

To refer to any previous value, use '\$' followed by the value's history number. The output printed by 'print' is designed to remind you of this. Just \$ refers to the most recent value in the history, and \$\$ refers to the value before that.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

If you have a chain of structures where the component 'next' points to the next one, you can print the contents of the next one with this:

```
p *$.next
```

It might be useful to repeat this command many times by typing RET.

Note that the history records values, not expressions.

If the value of x is 4 and you type this command:

```
print x
```

```
set x=5
```

then the value recorded in the value history by the 'print' command remains 4 even though the value of x has changed.

info values

Print the last ten values in the value history, with their item numbers. This is like 'p \$\$9' repeated ten times, except that 'info values' does not change the history.

info values n

Print ten history values centered on history item number n.

info values +

Print ten history values just after the values last printed.

8.8. Convenience Variables

GDB provides convenience variables that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no effect on further execution of your program. That's why you can use them freely.

Convenience variables have names starting with '\$'. Any name starting with '\$' can be used for a convenience variable, unless it is one of the predefined set of register names (see section Registers).

You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. Example:

```
set $foo = *object_ptr
```

would save in \$foo the value contained in the object pointed to by object_ptr.

Using a convenience variable for the first time creates it; but its value is void until you assign a new value. You can alter the value with another assignment at any time.

Convenience variables have no fixed types. You can assign a convenience variable any type of value, even if it already has a value of a different type. The convenience variable as an expression has whatever type its current value has.

info convenience

Print a list of convenience variables used so far, and their values. Abbreviated 'i con'.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example:

```
set $i = 0
print bar[$i++]>contents
...repeat that command by typing RET.
```

Some convenience variables are created automatically by GDB and given values likely to be useful.

\$_ The variable \$_ is automatically set by the 'x' command to the last address examined (see section Memory). Other commands which provide a default address for 'x' to examine also set \$_ to that address; these commands include 'info line' and 'info breakpoint'.

\$__ The variable \$__ is automatically set by the 'x' command to the value found in the last address examined.

8.9. Registers

Machine register contents can be referred to in expressions as variables with names starting with '\$'. The names of registers are different for each machine; use 'info registers' to see the names used on your machine. The names \$pc and \$sp are used on all machines for the program counter register and the stack pointer. Often \$fp is used for a register that contains a pointer to the current stack frame, and \$ps is used for a register that contains the processor status. These standard register names may be available on your machine even though the info registers command displays them with a different name. For example, on the SPARC, info registers displays the processor status register as \$psr but you can also refer to it as \$ps.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered floating point. There is no way to refer to the contents of an ordinary register as floating point value (although you can print it as a floating point value with 'print/f \$regname').

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" format, but all C programs expect to work with "double" format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the 'info registers' command prints the data in both formats.

Register values are relative to the selected stack frame (see section Selection). This means that you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the real contents of all registers, you must select the innermost frame (with 'frame 0').

Some registers are never saved (typically those numbered zero or one) because they are used for returning function values; for these registers, relativization makes no difference.

info registers

Print the names and relativized values of all registers.

info registers regname

Print the relativized value of register regname. regname may be any register name valid on the machine you are using, with or without the initial '\$'.

8.9.1. Examples

You could print the program counter in hex with

p/x \$pc

or print the instruction to be executed next with

x/i \$pc

or add four to the stack pointer with

set \$sp += 4

The last is a way of removing one word from the stack, on machines where stacks grow downward in memory (most machines, nowadays). This assumes that the innermost stack frame is selected. Setting \$sp is not allowed when other stack frames are selected.

9.

Examining the Symbol Table

The commands described in this section allow you to make inquiries for information about the symbols (names of variables, functions and types) defined in your program. This information is found by GDB in the symbol table loaded by the 'symbol-file' command; it is inherent in the text of your program and does not change as the program executes.

whatis exp

Print the data type of expression exp. exp is not actually evaluated, and any side-effecting operations (such as assignments or function calls) inside it do not take place. See section Expressions.

whatis

Print the data type of \$, the last value in the value history.

info address symbol

Describe where the data for symbol is stored. For a register variable, this says which register it is kept in. For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.

Note the contrast with 'print &symbol', which does not work at all for a register variables, and for a stack local variable prints the exact address of the current instantiation of the variable.

ptype typename

Print a description of data type typename. typename may be the name of a type, or for C code it may have the form 'struct struct-tag', 'union union-tag' or 'enum enum-tag'.

info sources

Print the names of all source files in the program for which there is debugging information.

info functions

Print the names and data types of all defined functions.

info functions regexp

Print the names and data types of all defined functions whose names contain a match for regular expression regexp. Thus, 'info fun step' finds all functions whose names include 'step'; 'info fun ^step' finds those whose names start with 'step'.

info variables

Print the names and data types of all variables that are declared outside of functions (i.e., except for local variables).

info variables regexp

Print the names and data types of all variables (except for local variables) whose names contain a match for regular expression regexp.

info types

Print all data types that are defined in the program.

info types regexp

Print all data types that are defined in the program whose names contain a match for regular expression regexp.

printsyms filename

Write a complete dump of the debugger's symbol data into the file filename.

10.

Altering Execution

Once you think you have find an error in the program, you might want to find out for certain whether correcting the apparent error would lead to correct results in the rest of the run. You can find the answer by experiment, using the GDB features for altering execution of the program.

For example, you can store new values into variables or memory locations, give the program a signal, restart it at a different address, or even return prematurely from a function to its caller.

10.1. Assignment to Variables

To alter the value of a variable, evaluate an assignment expression. See section Expressions. For example,

```
print x=4
```

would store the value 4 into the variable x, and then print the value of the assignment expression (which is 4).

All the assignment operators of C are supported, including the incrementation operators ‘++’ and ‘--’, and combining assignments such as ‘+=’ and ‘<=>’.

If you are not interested in seeing the value of the assignment, use the ‘set’ command instead of the ‘print’ command. ‘set’ is really the same as ‘print’ except that the expression’s value is not printed and is not put in the value history (see section Value History). The expression is evaluated only for side effects.

Note that if the beginning of the argument string of the ‘set’ command appears identical to a ‘set’ sub-command, it may be necessary to use the ‘set variable’ command. This command is identical to ‘set’ except for its lack of sub-commands.

GDB allows more implicit conversions in assignments than C does; you can freely store an integer value into a pointer variable or vice versa, and any structure can be converted to any other structure that is the same length or shorter.

To store values into arbitrary places in memory, use the '{...}' construct to generate a value of specified type at a specified address (see section Expressions). For example, {int}0x83040 would refer to memory location 0x83040 as an integer (which implies a certain size and representation in memory), and

```
set {int}0x83040 = 4
```

would store the value 4 into that memory location.

10.2. Continuing at a Different Address

Ordinarily, when you continue the program, you do so at the place where it stopped, with the 'cont' command. You can instead continue at an address of your own choosing, with the following commands:

jump linenum

Resume execution at line number linenum. Execution may stop immediately if there is a breakpoint there.

The 'jump' command does not change the current stack frame, or the stack pointer, or the contents of any memory location or any register other than the program counter. If line linenum is in a different function from the one currently executing, the results may be bizarre if the two functions expect different patterns of arguments or of local variables. For this reason, the 'jump' command requests confirmation if the specified line is not in the function currently executing. However, even bizarre results are predictable based on careful study of the machine-language code of the program.

jump *address

Resume execution at the instruction at address address.

You can get much the same effect as the jump command by storing a new value into the register \$pc. The difference is that this does not start the program running; it only changes the address where it will run when it is continued. For example,

```
set $pc = 0x485
```

causes the next 'cont' command or stepping command to execute at address 0x485, rather than at the address where the program stopped. See section Stepping.

The most common occasion to use the 'jump' command is when you have stepped across a function call with next, and found that the return value is incorrect. If all the relevant data appeared correct before the function call, the error is probably in the function that just returned.

In general, your next step would now be to rerun the program and execute up to this function call, and then step into it to see where it goes astray. But this may be time consuming. If the function did not have significant side effects, you could get the same information by resuming execution just before the function call and stepping through it. To do this, first put a breakpoint on that function; then, use the 'jump' command to continue on the line with the function call.

10.3. Giving the Program a Signal

signal signalnum

Resume execution where the program stopped, but give it immediately the signal number signalnum.

Alternatively, if signalnum is zero, continue execution without giving a signal. This is useful when the program stopped on account of a signal and would ordinarily see the signal when resumed with the 'cont' command; 'signal 0' causes it to resume without a signal.

10.4. Returning from a Function

You can cancel execution of a function call with the 'return' command. This command has the effect of discarding the selected stack frame (and all frames within it), so that control moves to the caller of that function. You can think of this as making the discarded frame return prematurely.

First select the stack frame that you wish to return from (see section Selection). Then type the 'return' command. If you wish to specify the value to be returned, give that as an argument.

This pops the selected stack frame (and any other frames inside of it), leaving its caller as the innermost remaining frame. That frame becomes selected. The specified value is stored in the registers used for returning values of functions.

The 'return' command does not resume execution; it leaves the program stopped in the state that would exist if the function had just returned. Contrast this with the 'finish' command (see section Stepping), which resumes execution until the selected stack frame returns naturally.

11.

Canned Sequences of Commands

GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files.

11.1. User-Defined Commands

A user-defined command is a sequence of GDB commands to which you assign a new name as a command. This is done with the 'define' command.

define commandname

Define a command named `commandname`. If there is already a command by that name, you are asked to confirm that you want to redefine it.

The definition of the command is made up of other GDB command lines, which are given following the 'define' command. The end of these commands is marked by a line containing 'end'.

document commandname

Give documentation to the user-defined command `commandname`. The command `commandname` must already be defined. This command reads lines of documentation just as 'define' reads the lines of the command definition, ending with 'end'. After the 'document' command is finished, 'help' on command `commandname` will print the documentation you have specified.

You may use the 'document' command again to change the documentation of a command. Redefining the command with 'define' does not change the documentation.

User-defined commands do not take arguments. When they are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

Commands that would ask for confirmation if used interactively proceed without asking when used inside a user-defined command. Many GDB commands that normally print messages to say what they are doing omit the messages when used in user-defined command.

11.2. Command Files

A command file for GDB is a file of lines that are GDB commands. Comments (lines starting with '#') may also be included. An empty line in a command file does nothing; it does not mean to repeat the last command, as it would from the terminal.

When GDB starts, it automatically executes its init files, command files named '.gdbinit'. GDB reads the init file (if any) in your home directory and then the init file (if any) in the current working directory. (The init files are not executed if the '-nx' option is given.) You can also request the execution of a command file with the 'source' command:

source filename

Execute the command file filename.

The lines in a command file are executed sequentially. They are not printed as they are executed. An error in any command terminates execution of the command file.

Commands that would ask for confirmation if used interactively proceed without asking when used in a command file. Many GDB commands that normally print messages to say what they are doing omit the messages when used in a command file.

11.3. Commands for Controlled Output

During the execution of a command file or a user defined command, the only output that appears is what is explicitly printed by the commands of the definition. This section describes three commands useful for generating exactly the output you want.

echo text

Print text. Non-printing characters can be included in text using C escape sequences, such as '\n' to print a newline. No newline will be printed unless you specify one. In addition to the standard C escape sequences a backslash followed by a space stands for a space. This is useful for outputting a string with spaces at the beginning or the end, since leading and trailing spaces are trimmed from all arguments. Thus, to print "and foo = ", use the command "echo \ and foo = \ "

A backslash at the end of text can be used, as in C, to continue the command onto subsequent lines. For example,

```
echo This is some text\n\
```

```
which is continued\n\
```

```
onto several lines.\n
```

produces the same output as

```
echo This is some text\n echo which is continued\n echo onto several lines.\n
```

output expression

Print the value of expression and nothing but that value: no newlines, no '\$nn = '. The value is not entered in the value history either. See section Expressions for more information on expressions.

output/fmt expression

Print the value of expression in format fmt. See section Output formats, for more information.

printf string, expressions...

Print the values of the expressions under the control of string. The expressions are separated by commas and may be either numbers or pointers. Their values are printed as specified by string, exactly as if the program were to execute

```
printf (string, expressions...);
```

For example, you can print two values in hex like this:

```
printf "foo, bar-foo = 0x%x, 0x%x\n", foo, bar-foo
```

only backslash-escape sequences that you can use in the string are the simple ones that consist of backslash followed by a letter.

12.

Options and Arguments for GDB

When you invoke GDB, you can specify arguments telling it what files to operate on and what other things to do.

12.1. Mode Options

- '-nx'** Do not execute commands from the init files '.gdbinit'. Normally, the commands in these files are executed after all the command options and arguments have been processed. See section Command Files.
- '-q'** "Quiet". Do not print the usual introductory messages.
- '-batch'** Run in batch mode. Exit with code 0 after processing all the command files specified with '-x' (and '.gdbinit', if not inhibited). Exit with nonzero status if an error occurs in executing the GDB commands in the command files.
- '-fullname'** This option is used when Emacs runs GDB as a subprocess. It tells GDB to output the full file name and line number in a standard, recognizable fashion each time a stack frame is displayed (which includes each time the program stops). This recognizable format looks like two '\032' characters, followed by the file name, line number and character position separated by colons, and a newline. The Emacs-to-GDB interface program uses the two '\032' characters as a signal to display the source code for the frame.

12.2. File-specifying Options

All the options and command line arguments given are processed in sequential order. The order makes a difference when the '-x' option is used.

- '-s file'** Read symbol table from file file.
- '-e file'** Use file file as the executable file to execute when appropriate, and for examining pure data in conjunction with a core dump.

'-se file' Read symbol table from file file and use it as the executable file.

'-c file' Use file file as a core dump to examine.

'-x file' Execute GDB commands from file file.

'-d directory' Add directory to the path to search for source files.

12.3. Other Arguments

If there are arguments to GDB that are not options or associated with options, the first one specifies the symbol table and executable file name (as if it were preceded by '-se') and the second one specifies a core dump file name (as if it were preceded by '-c').

13.

Using GDB under GNU Emacs

A special interface allows you to use GNU Emacs to view (and edit) the source files for the program you are debugging with GDB.

To use this interface, use the command `M-x gdb` in Emacs. Give the executable file you want to debug as an argument. This command starts GDB as a subprocess of Emacs, with input and output through a newly created Emacs buffer.

Using GDB under Emacs is just like using GDB normally except for two things:

- All “terminal” input and output goes through the Emacs buffer. This applies both to GDB commands and their output, and to the input and output done by the program you are debugging.

This is useful because it means that you can copy the text of previous commands and input them again; you can even use parts of the output in this way.

All the facilities of Emacs’s Shell mode are available for this purpose.

- GDB displays source code through Emacs. Each time GDB displays a stack frame, Emacs automatically finds the source file for that frame and puts an arrow (`=>`) at the left margin of the current line.

Explicit GDB ‘list’ or search commands still produce output as usual, but you probably will have no reason to use them.

In the GDB I/O buffer, you can use these special Emacs commands:

M-s Execute to another source line, like the GDB ‘step’ command.

M-n Execute to next source line in this function, skipping all function calls, like the GDB ‘next’ command.

M-i Execute one instruction, like the GDB ‘stepi’ command.

C-c C-f

Execute until exit from the selected stack frame, like the GDB 'finish' command.

M-c Continue execution of the program, like the GDB 'cont' command.

M-u Go up the number of frames indicated by the numeric argument (see section Arguments, Numeric Arguments, emacs, The GNU Emacs Manual), like the GDB 'up' command.

M-d Go down the number of frames indicated by the numeric argument, like the GDB 'down' command.

In any source file, the Emacs command C-x SPC (gdb break) tells GDB to set a breakpoint on the source line point is on.

The source files displayed in Emacs are in ordinary Emacs buffers which are visiting the source files in the usual way. You can edit the files with these buffers if you wish; but keep in mind that GDB communicates with Emacs in terms of line numbers. If you add or delete lines from the text, the line numbers that GDB knows will cease to correspond properly to the code.

14.

Remote Kernel Debugging

If you are trying to debug a program running on a machine that can't run GDB in the usual way, it is often useful to use remote debugging. For example, you might be debugging an operating system kernel, or debugging a small system which does not have a general purpose operating system powerful enough to run a full-featured debugger. Currently GDB supports remote debugging over a serial connection.

The program to be debugged on the remote machine needs to contain a debugging device driver which talks to GDB over the serial line using the protocol described below. The same version of GDB that is used ordinarily can be used for this. Several sample remote debugging drivers are distributed with GDB; see the 'README' file in the GDB distribution for more information.

For details of the communication protocol, see the comments in the GDB source file 'remote.c'.

14.1. Commands for Remote Debugging

To start remote debugging, first run GDB and specify as an executable file the program that is running in the remote machine. This tells GDB how to find the program's symbols and the contents of its pure text. Then establish communication using the 'attach' command with a device name rather than a pid as an argument. For example:

```
attach /dev/ttyd
```

if the serial line is connected to the device named '/dev/ttyd'. This will stop the remote machine if it is not already stopped.

Now you can use all the usual commands to examine and change data and to step and continue the remote program.

To resume the remote program and stop debugging it, use the 'detach' command.

Occasionally it is useful to execute a shell command from within GDB. This can be done with the 'shell' command.

shell shell command string

Directs GDB to invoke an inferior shell to execute shell command string. The environment variable SHELL is used if it exists, otherwise GDB uses '/bin/sh'.

—Symbols—

#pragma	5-17
.cld	3-1
.cln	3-1
__asm()	5-2
__asm() multiple instructions	5-3
__asm() reg_save	5-6
__c_sig_dispatch	6-10, 6-11
__c_sig_goto_dispatch	6-10
__c_sig_handlers	6-10, 6-11
__DATE__	4-2
__DSP96K__	4-2
__FILE__	4-2
__INCLUDE_LEVEL__	4-2
__LINE__	4-2
__mem_limit	6-9
__printf_end	6-9
__sig_dfl	6-11
__sig_drop_count	6-12
__sig_err	6-11
__sig_ign	6-11
__stack_safety	6-9
__STDC__	4-2
__time	6-9
__TIME__	4-2
__VERSION__	4-2

—Numerics

80386	2-3
80387 math coprocessor	2-3
80486	2-3

—A—

abort	6-3, 6-7, 6-12, A-6
abs	A-7

acos	A-8
activation record	4-7
address ALU	4-5
address offset registers	4-5
address registers	4-5
ads96000	1-3
ANSI	1-1
ANSI C compiler	4-1
asin	A-9
-asm option	3-3, 3-28
asm96000	1-1, 3-2, C-2
assembling	1-1
atan	A-10
atan2	A-11
atexit	A-12
atof	A-13
atoi	A-14
atol	A-15

—B—

-B option	3-3, 3-4
-b option	3-3, 3-4
brk	6-9
bsearch	A-16

—C—

C bootstrap code	6-2
-C option	3-3, 3-6
-c option	3-3, 3-28
callee	4-10
caller	4-10
calling convention	4-10
calloc	6-9, A-18
ceil	A-20
char	4-2, 4-4
cldinfo	C-6

cldlod C-7
 cofdmp C-8
 COFF 3-1
 Common Sub-Expression 4-13
 compiler control program 1-1
 compiler naming convention 4-10
 compiler temporary space 4-8
 compiling 1-1
 constant folding 4-13
 control line 3-10
 control program 3-1
 cos A-21
 cosh A-22
 counter_string 5-18
 cpp 1-1, 3-2
 -crt option 3-3, 3-28
 crt0 file 6-1, 6-2
 CSE 4-13

—D—

-D option 3-3, 3-7
 data ALU 4-5
 data memory configuration 4-9
 data segments 6-4
 div A-23
 DMA controller 6-8
 DMA data transfer 6-8
 double 4-2, 4-3
 Dr. BuB FIR filter program 5-7
 DSIZE 6-2
 dsplib 1-1, C-9
 dspln 3-2
 dsplnk 1-1, C-11
 dynamic link 4-7
 dynamic time variable 6-9

—E—

-E option 3-3, 3-8
 emu 2-3
 errno 4-12, 6-9
 error code 6-9
 exit 6-3, A-24
 exp A-25

—F—

F__crt0_end 6-3
 fabs A-26
 -fcaller-saves option 3-3, 3-19
 -fcond-mismatch option 3-3, 3-20
 -ffixed-REG option 3-3, 3-20
 -fforce-addr option 3-3, 3-19
 file inclusion 3-10
 -finline-functions option 3-3, 3-19
 -fkeep-inline-functions option 3-3, 3-19
 float 4-2, 4-3
 floating point data memory model 4-3
 floor A-27
 fmod A-28
 -fno-defer-pop option 3-3, 3-19
 -fno-opt option 3-3, 3-18
 -fno-peephole option 3-3, 3-18
 -fno-strength-reduce option 3-3, 3-18
 frame pointer 4-6, 4-8, 6-2
 free A-29
 frexp A-30
 -fvolatile option 3-3, 3-20
 -fwritable-strings option 3-3, 3-19

—G—

-g option 3-3, 3-20
 G96_EXEC_PREFIX 3-4, 3-28

g96k 1-1, 1-3, 3-1
gcc96-cc1 1-1, 3-2
gdb96 1-1, 1-3, 3-1, 3-20, C-15
global assembler directive 5-29
global C variable 4-10
global data 5-14, 5-15
global-static data segment 6-4
GO32 2-3

—H—

heap data segment 6-4
heap-stack check 6-9
hello.c 2-4
Host Interface 5-11
host port 6-2

—I—

-l option 3-3, 3-10, 3-11
-i option 3-3, 3-12
identifier 4-1
identifier length limits 4-1
in-line assembly 5-1
in-line assembly code 5-1, 5-2
in-line assembly code delimiter 4-10
input_operands 5-2
instruction combination 4-15
instruction_template 5-2
int 4-2, 4-4
integral data type 4-2
interrupt source 6-6
interrupt vector 6-1, 6-6
isalnum A-31
isalpha A-32
iscntrl A-33
isdigit A-34
isgraph A-35

islower A-36
isprint A-37
ispunct A-38
isspace A-39
isupper A-40
isxdigit A-41

—J—

-j option 3-3, 3-28

—L—

-l option 3-3, 3-29
l_load 5-18
l_run 5-18
labs A-42
ldexp A-43
ldiv A-44
leaf routine 4-16
linking 1-1
local data space 4-7
local label 4-10
log A-45
log10 A-46
long 4-2, 4-4
longjmp 3-24, 6-2, 6-12, A-47
loop invariant code motion 4-14
loop rotation 4-15

—M—

-M option 3-3, 3-13
makefile rules 3-13
malloc 6-9, A-49
map files 3-29
MB_CUR_MAX A-122
mblen A-50
mbstowcs A-52

mbtowc A-54
 memchr A-56
 memcmp A-58
 memcpy A-60
 memmove A-61
 memory limit variable 6-9
 memset A-62
 mixing C and assembly 5-1
 -ml-memory option 3-3, 3-22, 3-23
 -MM option 3-3, 3-14
 -mno-biv-plus-linv-promotion option 3-3, 3-21
 -mno-do-loop-generation option . 3-3, 3-21
 -mno-dsp-optimization option . . 3-3, 3-21
 modf A-64
 modifier 5-4
 modifier registers 4-5
 -mstack_check option . . . 3-3, 3-22, 4-12
 -mx-memory option 3-3, 3-22, 3-23
 -my-memory option 3-3, 3-22, 3-23

—N—

-nostdinc option 3-3, 3-15

—O—

-O option 3-3, 3-21
 -o option 3-3, 3-5
 old frame pointer 4-7
 omr 6-2
 opearnd constraint a 5-5
 operand constraint d 5-5
 operand constraint f 5-5
 Operand Constraint i 5-11
 operand constraint m 5-5
 operand constraint number 5-5
 operand expansion string (OES) 5-3

operand_id_number 5-4
 optimization techniques 5-31

Option, Assemble

-asm string 3-28
 -c 3-28

Option, Command line

-Bdirectory 3-4
 -bPREFIX 3-4
 -o FILE 3-5
 -v 3-5

Option, Compile

-fcaller-saves 3-19
 -fcond-mismatch 3-20
 -ffixed-REG 3-20
 -fforce-addr 3-19
 -finline-functions 3-19
 -fkeep-inline-functions 3-19
 -fno-defer-pop 3-19
 -fno-opt 3-18
 -fno-peephole 3-18
 -fno-strength-reduce 3-18
 -fvolatile 3-20
 -fwritable-strings 3-19
 -g 3-20
 -ml-memory 3-22
 -mno-biv-plus-linv-promotion 3-21
 -mno-do-loop-generation 3-21
 -mno-dsp-optimization 3-21
 -mstack_check 3-22
 -mx-memory 3-22
 -my-memory 3-22
 -O 3-21
 -pedantic 3-23
 -Q 3-23
 -S 3-23
 -W 3-24

- w 3-24
- Wall 3-27
- Wcast-qual 3-27
- Wid-clash-LEN 3-27
- Wimplicit 3-25
- Wpointer-arith 3-27
- Wreturn-type 3-26
- Wshadow 3-27
- Wswitch 3-27
- Wunused 3-26
- Wwrite-strings 3-27
- Option, Link
 - crt file 3-28
 - j string 3-28
 - lLIBRARY 3-29
 - r MAPFILE 3-29
- Option, Preprocessor
 - C 3-6
 - DMACRO 3-7
 - DMACRO=DEFN 3-8
 - E 3-8
 - I- 3-11
 - i FILE 3-12
 - IDIR 3-10
 - M 3-13
 - MM 3-14
 - nostdinc 3-15
 - P 3-16
 - pedantic 3-16
 - UMACRO 3-17
 - v 3-16
 - Wcomment 3-17
 - Wtrigraphs 3-18
- out-of-line assembly code 5-20
- out-of-line calling C routines 5-30
- output_operands 5-2

—P—

- P option 3-3, 3-16
- p_load 5-18
- p_run 5-18
- parameter data space 4-7
- pedantic option 3-3, 3-16, 3-23
- perror A-65
- pointers 4-4
- pow A-66
- pragma 5-17
- predefined macro names 4-1
- preprocessing 1-1
- printf A-67
- printf stub function 6-9
- Program Memory Configuration 4-7
- program segment 6-4
- putchar A-72
- puts A-73

—Q—

- Q option 3-3, 3-23
- qsort A-74

—R—

- r option 3-3, 3-29
- raise 6-1, A-76
- raise() 6-11
- rand A-77
- read-only operand 5-5
- realloc 6-9, A-78
- reg_save 5-3
- return address 4-7
- return sequence 4-10
- run96sim 1-3, 4-12, C-17
- run-time environment 6-1

—S—

-S option 3-3, 3-23
section name 4-10
setjmp 3-24, 6-1, 6-12, A-80
short 4-2, 4-4
SIG_ERR 6-11
SIG_IGN 6-11
signal 6-1, 6-9, A-82
signal file 6-9
signal() 6-1
sim96000 1-3
sin A-84
single extended precision 4-2
single extended precision float 4-5
single precision 4-2
single precision float 4-5
sinh A-85
sizeof 4-1
sprintf A-86
sqrt A-88, A-89
srec C-18
stack data segment 6-4
stack pointer 4-6, 4-8, 6-2
standard directory search list 3-2
standard include directory 3-10
standard system directories 3-15
start-up files 3-2
static function 4-10
stop 6-3
strcat A-90
strchr A-91
strcmp A-92
strcoll A-94
strcpy A-95
strcspn A-96
strength reduction 4-14

strerror A-97
string constant label 4-10
strlen A-98
strncat A-99
strncmp A-100
strncpy A-101
strpbrk A-102, A-104
strstr A-105
strtod A-106
strtok A-108
strtol A-110
strtoul A-112
strxfrm A-114
subroutine call 5-12, 5-13
system header files 3-14

—T—

tan A-115
tanh A-116
template-epilogue 5-20, 5-24
template-main program 5-20, 5-22
template-prologue 5-20, 5-21
template-restore all registers 5-20, 5-23
template-save all registers 5-20, 5-22
tolower A-117
toupper A-118

—U—

-U option 3-3, 3-17
unsigned char 4-2, 4-4
unsigned int 4-2, 4-4
unsigned long 4-2, 4-4
unsigned short 4-2, 4-4

—V—

-v option 3-3, 3-5, 3-16

volatile 3-24, 4-14, 5-17

—W—

-W option 3-3, 3-17, 3-24
-w option 3-23
-Wall option 3-3, 3-27
-Wcast-qual option 3-3, 3-27
wcstombs A-120
wctomb A-122
-Wid-clash-LEN option 3-3, 3-27
-Wimplicit option 3-3, 3-25
word boundary 4-1
-Wpointer-arith option 3-3, 3-27
-Wreturn-type option 3-3, 3-26
write-only operand 5-5
-Wshadow option 3-3, 3-27
-Wswitch option 3-3, 3-27
-Wunused option 3-3, 3-26
-Wwrite-strings option 3-3, 3-27

—X—

x_load 5-18
x_run 5-18
XDEF assembler directive 5-29
XREF assembler directive 5-29

—Y—

y_load 5-18
y_run 5-18

