Freescale Digital Signal Processing
Development Software

DSP563CCC

Freescale DSP56300 Family Optimizing C Compiler User's Manual

DSP563CCUG

Specification and information herein are subject to change without notice.

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

***For Literature Requests Only*:**
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Table Of Contents

Table of Contents (Continued)

Chapter 1
Introduction

1.1 Overview

The DSP563CCC GNU based C cross-compiler is the latest high-level language development system for the Freescale DSP56300 family of digital signal processors (DSPs). It includes:

- an integrated control program — **g563c**
- an ANSI compliant C language preprocessor — **mcpp**
- an ANSI optimizing C compiler — **g563-cc1**
- an assembly language optimizer which does instruction scheduling and coalescing (combining ALU operations with MOVEs) — **alo563**
- a **DSP56300** common object file format (COFF) assembler — **asm56300**
- a COFF linker — **dsplnk**
- a COFF librarian — **dsplib**
- a simulator based command-line execution program — **run563**
- various object file manipulation tools — **cldinfo, cldlod, cofdmp, srec, strip**

This integrated software system runs on a variety of machines and operating systems, including the IBM PC™, Sun SPARC™ workstations, and HP 700 series workstations. The C compiler is included in and installed by default, along with the Symphony Studio Integrated Development Environment.

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 input to the assembler. The compiler automatically implements numerous optimizations.

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.

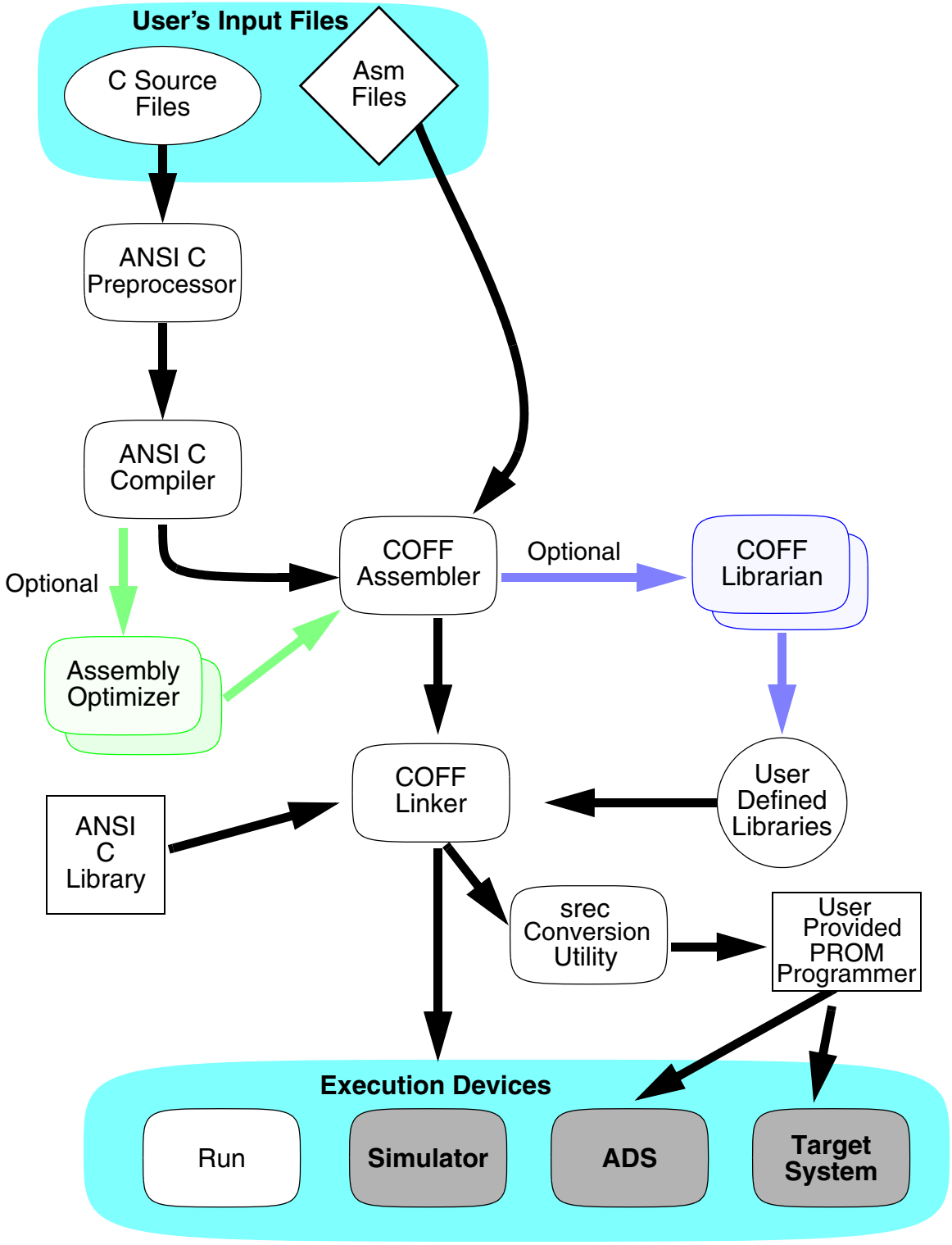

Figure 1-1. Freescale Software Development System

The compiler control program, **g563c**, is the standard compiler interface 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 program parses the command line options, and invokes the required sub-programs on the specified files.

Note: Object files are stored using the COFF format. COFF stands for *Common Object File Format.* Utilities such as **cldinfo** and **cldlod** may be used to gain visibility into object files.

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 C preprocessor and the C 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 the compiler output from these files to the assembler, in addition to any assembly language files specified by the user on the **g563c** command line.
3. The assembler output is routed to the linker for final processing. The linker resolves all unresolved link-time symbols with the standard (and any explicitly requested) C libraries. The COFF linker object file output 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. **run563** which allows the DSP56300 code (in COFF format) to be executed on the host computer's CPU,
   b. **sim56300** which is a complete DSP56300 simulator that can be used to execute the compiled application (in either COFF format or '.lod' file format) and allow examination of registers and memory,
   c. **ads56300** is the development system hardware that can then be used to load and execute the compiled application (in either COFF format or '.lod' file format) on the ADS development system, and
   d. the target system shown is the user's custom DSP system.

Note:The three execution devices in the shaded boxes 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 DSP563CCC compiler system.

The DSP56300 family represents a departure from more conventional architectures on which many other implementations of the C language are based (no byte-addressable memory, multiple memory spaces, unusual addressing modes, etc.). Also, the nature of DSP applications dictates that a great 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 stack initialization, chip operating modes and other issues.

This manual provides:

1. 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.
2. Reference information on compiler options, ANSI library routines, and utilities.

This manual assumes a familiarity with the C programming language, and with the host machine and operating environment. It also assumes that the programmer understands how to create and edit C language source files on the host system.

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 is indicated in this manual by:
   `C:\>`
2. An example of an MS-DOS directory name is:
   `\USR\DIRECTORY`
3. The contents of an MS-DOS directory are examined by entering:
   `C:\> DIR`
4. The contents of an MS-DOS file are displayed by entering:
   `C:\> TYPE FILE`
5. The program "HELLO.EXE" would be executed by the command line:
   `C:\> HELLO`

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-C. The contents of each chapter and each appendix are described below.

Chapter 1, *Introduction*, describes the overall organization of the DSP563CCC compiler system. It also details the structure of this document, and conventions followed herein.

Chapter 2, *Installation Guide*, describes the installation and organization of DSP563CCC. 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, *Control Program Options*, discusses the four passes of the compilation process with particular attention to the functions of the compiler control program **g563c.** This chapter includes a list of the compiler invocation options along with example command lines for different memory and program configurations.

Chapter 4, *About g563c,* 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 linking assembly language modules with C modules and linking C modules with assembly language modules.

Chapter 6, *Software-Hardware Integration*, describes how to modify a program's run-time environment, how to write software to handle interrupts, and the **setjmp**/**longjmp** ANSI library routines.

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

Appendix B, *Utilities*, provides documentation for each of the support utilities provided with the compiler.

Appendix C, *GNU General Public License*, explains your rights as to the redistribution of the GNU based programs **g563c**, **g563-cc1**, and **mcpp**.

Chapter 2
Control Program Options

2.1 Overview

Program **g563c** is the *control program* for Freescale's optimizing C compiler for the DSP56300 family of digital signal processors. The program **g563c** automates control of the four C compiler phases – preprocessing, compiling, assembling, and linking. The program **g563c** utilizes a command syntax similar to those adopted by typical UNIX utilities. The **g563c** syntax is:

```
g563c [options] files
```

where:

1. [options] is one or more of the options found in this chapter. One difference between **g563c** 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 source level debug information, whereas "**-Og**", which is acceptable to UNIX-style compilers is not acceptable to **g563c**.

2. "files …" are the files to be processed. Program **g563c** accepts input filenames suffixed with "**.c**" (ANSI C source files), "**.i**" (preprocessed C source files), "**.asm**" (DSP56300 assembly code), and "**.cln**" (COFF link files). The control program processes each file according to this suffix. The **g563c** output is controlled by the specific set of command line options provided. For instance, if no command line arguments are provided, the compiler will attempt to 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 2.2.

Note: It is strongly recommended that g563c always be used to invoke the C compiler utilities rather than individually executing them.

A standard directory search list is consulted by **g563c** for:

1. Each of the five executables,
   a. **mcpp** – the C preprocessor,
   b. **g563-cc1** – the C compiler/optimizer,
   c. **alo563** – the assembly language optimizer,
   d. **asm56300** – the DSP56300 assembler,
   e. **dsplnk** – the DSP56300 linker.
2. Start-up file, **crt0563[xyl].cln**.
3. ANSI C library, **lib563c[xyl].clb**.

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 2.2 is:

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

Note that if the environment variable **DSPLOC** is set, the value of **DSPLOC** will be substituted for 1 and 2 above.

Table 3-1 lists all the user selectable options used by **g563c.** 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 **g563c** Command Line Options and Preprocessor Phase Options. This is actually the same option but it is used by these two programs in different ways (see Section 2.2 and Section 2.2.1).

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 DSP56300. 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

Table 2-1 - Options

g563c Command Line Options

-B*directory*
-**b***PREFIX*
-o *FILE*
-v

Preprocessor Phase Options

-C
-D*MACRO*
-D*MACRO=DEFN*
-E
-I*DIR*
-I-
-i *FILE*
-M
-MM
-nostdinc
-pedantic
-v
-U*MACRO*
-W*comment*
-W*trigraphs*

Assemble Phase Options

-asm *string*
-c

Link Phase Options

-crt *file*
-j *string*
-l*LIBRARY*
-r *MAPFILE*

Compile Phase Options

-alo
-f*no-opt*
-f*no-peephole*
-f*no-strength-reduce*
-f*no-defer-pop*
-f*force-addr*
-f*inline-functions*
-f*caller-saves*
-f*keep-inline-functions*
-f*writable-strings*
-f*cond-mismatch*
-f*volatile*
-f*fixed-REG*
-g
-O
-m*conserve-p-mem*
-m*no-dsp*
-m*no-do-loop-generation*
-m*no-linv-plus-biv-promotion*
-m*p-mem-switchtable*
-m*x-memory*
-m*y-memory*
-m*l-memory*
-m*stack-check*
-pedantic
-Q
-S
-w
-W
-W*implicit*
-W*return-type*
-W*unused*
-W*switch*
-W*all*
-W*shadow*
-W*id-clash-LEN*
-W*pointer-arith*
-W*cast-qual*
-W*write-strings*

preferable to trade memory space for speed, but in cases where the extra memory space is not available, a particular optimization might be unwise.

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

2.2 g563c 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

-B*directory*

Add *directory* to the standard search list and have it searched first. This can also be accomplished by defining the environment variable **G563_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, lib563cy.clb, which is installed as **\dsp\new**\lib563cy.clb use:

```
C:\> g563c -B\dsp\new\ file.c -o file.cld
```

Example 3-2. Using the G563_EXEC_PREFIX environment variable to have the same effect as Example 3.1, include in the autoexec.bat file:

```
set G563_EXEC_PREFIX=C:\DSP\NEW\
```

and then execute:

```
C:\> g563c file.c -o file.cld
```

Example 2-3. To test a new version of the DSP56300 C preprocessor before permanent installation, install a new **mcpp** program as **c:\tmp\new\mcpp** and then execute:

```
C:\> g563c -Bc:\tmp\new\ testfile.c
```

-**b***PREFIX*

Direct **g563c** 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 2-4. Test a new version of the ANSI C library, lib563cy.clb, installed as c:\dsp\lib**\new-lib563cy.clb.**

```
C:\> g563c -bnew- file.c -o file.cld
```

Example 2-5. Test a new version of the **DSP563CCC** preprocessor before permanent installation.

Install the new **mcpp** program **dsp\bin\new-mcpp** and

```
C:\> g563c -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 options 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 2-6. Only generate a preprocessed file (do not invoke the compiler, assembler or linker) and put the results in file.i.

```
C:\> g563c -E file.c -o file.i
```

Example 2-7. 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.**

```
C:\> g563c 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.

2.2.1 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 (preprocess only), as the requested preprocessor output may be unsuitable for actual compilation. The default option is to use ANSI C syntax. For example, if the **-I***DIR* 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 2-8. This example preprocesses a simple program, foo.c, without discarding comments.

```
C:\> type foo.c
/*
 * This COMMENT won't be deleted.
 */
main()
{
  printf("Hello, DSP56301\n");
}

C:\> g563c -E -C foo.c
# 1 "foo.c"
/*
 * This COMMENT won't be deleted.
 */
main()
{
  printf("Hello, DSP56301\n");
}
```

-D*MACRO*

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

Example 2-9. 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.

```
C:\> type dsp.c
    #include <stdio.h>
    main()
    {
    #ifdef DSP56300
        printf("message: DSP56300.\n");
    #else
        printf("message: DSP56301.\n");
    #endif
    }

C:\> g563c -DDSP56300 dsp.c
C:\> dir
    a.cld  dsp.c

C:\> run56 a.cld
    message: DSP56300.

C:\> g563c dsp.c

C:\> ls
    a.cld  dsp.c

C:\> run56 a.cld
    message: DSP56301.
```

-D*MACRO=DEFN*

Define preprocessor macro *MACRO as DEFN.*

Example 2-10. 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.

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

C:\> g563c -DFROM_COMMAND_LINE=56300 dsp.c
C:\> dir
    a.cld  dsp.c

C:\> run563 a.cld
    message code: 56300.
```

-E

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

Example 2-11. This example shows how to preprocess the C source program foo.c and send the results to the standard output.

```
C:\> type foo.c
    #define DELAY 1000
    main()
    {
      int cnt = DELAY;
      while(cnt--);
    }
C:\> g563c -E foo.c
    # 1 "foo.c"

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

Example 2-12. The mcpp output can be saved into file "foo.i" by using the **-o** option.

```
C:\> type foo.c
   #define DELAY 1000
   main()
   {
     int cnt = DELAY;
     while(cnt--);
   }

C:\> g563c -E foo.c -o foo.i
C:\> dir
   foo.c  foo.i

C:\> type foo.i
   # 1 "foo.c"

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

-I*DIR*

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* on UNIX systems, and c:\dsp\include on MS-DOS systems. 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 **-I***DIR* 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* or c:\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* or c:\dsp\include.

Example 2-13. 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 current 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.

```
C:\> dir
   foo.c  inc/

C:\> dir inc
   myinclude.h

C:\> type foo.c
   #include "myinc.h" /* this is the control line to include it */
   main()
   {
     int cnt;
```

```
    cnt = DELAY;
    while(cnt--);
  }

C:\> type inc\myinc.h
  #define DELAY   100

C:\> g563c -I.\inc foo.cC:\> dir
  a.cld  foo.c  inc/
```

-I-

This option is always used in conjunction with the **-I***DIR* option and limits the file search to look for file inclusions #**include "filename"**, whereas -I*DIR* 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

-I*DIRA* -I- -I*DIRB*

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 2-14. A test program file.c is used to test a file operation fopen() which is, in this example, desired to be developed for a DSP56300 system. 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 -I*DIR* 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>.

```
C:\> dir
    file.c     inc/      mysys/

C:\> dir inc
    cnt.h

C:\> dir mysys
    stdio.h

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

C:\> type mysys\stdio.h
typedef    struct      FILE { /* FILE data structure to develop */
    char      name[10];
    char      buffer[1024];
 }  FILE;
 FILE *fopen(char *, char *); /* new function to develop */

C:\> g563c -I.\inc -I- -I.\mysys -E file.c
#   1   "file.c"
#   1   ".\mysys\stdio.h"      1
typedef    struct      FILE {
    char      name[10];
    char      buffer[1024];
```

```
}   FILE;
FILE*fopen(char*,char*);
#   1   ”file.c”    2
#   1   ”.\inc\cnt.h”      1
#   2   ”file.c”    2

main()
{
    int delay = 25;
    FILE *fp;
    fp = fopen (“myfile”, ”w”);
    while (--delay);
}
```

Notice that the file inclusion “cnt.h” is from the directory ./inc as shown in the line # 1 “.\inc\cnt.h” 1 and the file inclusion <stdio.h> is from the directory .\myinc as shown in the line # 1 “.\myinc\stdio.h” 1.

-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 2-15. 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.

```
C:\> dir
    macros.c   greeting.c

C:\> type macros.c
    #define  MESSAGE   "Hello, world."

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

C:\> g563c -i macros.c greeting.c
C:\> run563 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 2-16. 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.

```
C:\> dir
   big.c     greater.h

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

C:\> type greater.h
   #define   greater(x,y)   ((x)>(y)?(x):(y))
C:\> g563c -M big.c
   big.o : big.c \dsp\include\stdio.h     \dsp\include\ioprim.h  \dsp\in-
clude\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 2-17. 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 Freescale Cross C Compiler.

```
C:\> dir
   big.c     greater.h

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

C:\> type greater.h
   #define   greater(x,y)    ((x)>(y)?(x):(y))

C:\> g563c -MM big.c
   big.o : big.c greater.h

C:\> dir
   big.c     greater.h  makefile   text

C:\> type makefile
   a.cld : big.o
         g563c big.o
   big.o : big.c greater.h
         g563c -c -o big.o big.c
C:\> make
   g563c -c -o big.o big.c
   g563c big.o

C:\> run563 a.cld
   big:20
```

-nostdinc

Do not search the standard system directories for file inclusions. 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 2-18. A test program, test.c, is used to test a new version of the function printf() which is declared in 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, c:\ds\include or /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.

```
C:\> dir
    inc/    test.c

C:\> dir inc
    stdio.h

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

C:\> type inc\stdio.h
    void printf(char *);

C:\> g563c -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).

-v

Output preprocessor version information. The primary purpose of this command is to display the software version. This information is needed when calling the Freescale DSP helpline for assistance. 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 command is also used by the com-

mand

program to initiate the verbose mode of operation.

Example 2-19. The -v option is selected using the control program **g563c.** The version numbers for **g563c**,**mcpp** and **g563c-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.

```
C:\> dir
    foo.c

C:\> g563c -v foo.c
    g563c version Freescale Version: g1.24 -- GNU 1.37.1
c:\dsp\bin\mcpp -v -undef -D__Y_MEMORY -trigraphs -$ -D__STRICT_ANSI__
        -D__DSP563C__ -D__OPTIMIZE__ foo.c cca00527.cpp

    GNU CPP version 1.37.1
c:\dsp\bin\g563-cc1 cca00527.cpp -ansi -fstrength-reduce -quiet -dumpbase foo.c
        -O -version -o cca00527.asm

    GNU C version1.37.1 Freescale DSP563XX Freescale Version:g1.24 compiled
        by GNU C version 1.37.1.

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

c:\dsp\bin\asm56300 -c -B foo.cln -- cca00527.asm

c:\dsp\bin\dsplnk -c -B acld --c:\dsp\lib\crt0-y.cln foo.cln -L c:\dsp\lib\lib563cy.clb

C:\> dir
    a.cld  foo.c
```

-U*MACRO*

Undefine macro *MACRO*.

Example 2-20. 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.

```
C:\> dir
    debug.c

C:\> type debug.c
    #include <stdio.h>
    main()
    {
    #ifdef DEBUG
     printf("debug: a message.\n");
    #endif
     printf("normal operation.\n");
    }

C:\> g563c -UDEBUG debug.c
C:\> run563 a.cld
    normal operation.
```

-W*comment*

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

Example 2-21. 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.

```
C:\> dir
    foo.c

C:\> type foo.c
    /* foo.c */
    main() { /* begin */
      int d = 1000;  /* /* delay */
      while(d--);   /* /* main /* loop */
    }  /* end */

C:\> g563c -Wcomment foo.c
    foo.c:3: warning: '/*' within comment
    foo.c:4: warning: '/*' within comment
    foo.c:4: warning: '/*' within comment
C:\> g563c foo.c
```

-W*trigraphs*

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.).

2.2.2 Compile Phase Options

The default options are:

1. perform all machine dependent and independent optimizations
2. do *not* run the assembly language optimizer (**alo563**)
3. do not generate debugging information
4. locate data only in the Y data memory space.

-alo

Run the assembly language optimizer on the assembly language output of **g563-cc1**. This improves the utilization of parallel moves.

-f*no-opt*

Disable all optimizations.

-f*no-peephole*

Disable the peephole portion of optimization.

-f*no-strength-reduce*

Disable the optimizations of loop strength reduction and elimination of iteration variables as well as DSP56300 specific looping optimizations (**DO** instruction usage, etc.).

-f*no-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 *DSP56300 Family Manual* for information on stack overflow).

Examples of function calls that will **not** incur deferred pops whether or not the **-f***no-defer-pop* option is specified are:

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

-f*force-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.

-f*inline-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.

-f*caller-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.

-f*keep-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**.

-f*writable-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.

-f*cond-mismatch*

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

-f*volatile*

Consider all memory references through pointers to be volatile.

-f*fixed-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:

r0, r1, r2, r3, r4, r5

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

Example 2-22. Reserve r4 for later special purpose.

```
C:\> g563c -O -ffixed-r4 file.c -o file.cld
```

Caution

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

-g

Produce COFF debugging information.

A key feature afforded by the use of the GNU C compiler (**g563c**) teamed with the source level debugger 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 out weighs the problems encountered.

-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.

-m*conserve-p-mem*

Generate code that consumes less program memory at the expense of run time. Rather than generating a prologue and epilogue for each function, calls are made to the prolog and epilog routines included in the library. Similarly, rather than doing in-

line expansion of several operations (for instance modulus), a call to a library routine is emitted.

-m*no-dsp-optimization*
Disables all Freescale optimizer enhancements.

-m*no-do-loop-generation*
Disable **DO** instruction usage by optimizer.

-m*no-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.

-m*p-mem-switchtable*
Forces the compiler to locate all switch tables in P memory.

-m*x-memory*
Direct the compiler to locate data in the X data memory space. Memory modes cannot be mixed, i.e. *only one* of **-m***x-memory*, **-m***y-memory* or **-m***l-memory* may be selected.

Example 2-23. An application is programmed to utilize only the X data memory space and therefore must be compiled using the -mx-memory option.

```
C:\> ls
   x.c

C:\> type 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;
   }

C:\> g563c -S -mx-memory x.c
```

```
C:\> dir
   x.asm  x.c
```

-m*y-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 **-m***x-memory*, **-m***y-memory* or **-m***l-memory* **may be selected**.

-m*l-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 (**double** or **long**).
2. This requires that the X and Y memory spaces be evenly populated.

Memory modes cannot be mixed, i.e. **only one** of **-m***x-memory*, **-m***y-memory* or **-m***l-memory* **may be selected**.

-m*stack-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.

-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 DSP56300 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 2-24. Generate an optimized assembly language file (test.asm) of the C source program (test.c).

```
C:\> dir
   test.c

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

C:\> g563c -S test.c
C:\> dir
   test.asm  test.c
```

Example 2-25. Generate an optimized assembly language file *test.asm*.

```
C:\> g563c -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 non-volatile 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 2-26. 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.

```
C:\> dir
    foo.c

C:\> type foo.c
    int foo(int);
    main()
    {
      int i;
      foo(i);
    }
    int foo(a)
    {
      if(a > 0)
        return a;
    }

C:\> g563c -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
```

-W*implicit*

Warn whenever a function is implicitly declared.

Example 2-27. The function foo() is declared implicitly in the program foo.c, the -Wimplicit option will generate a warning message for this situation.

```
C:\> dir
    foo.c

C:\> type foo.c
    main()
    {
      foo();
    }
    int foo(){}
C:\> g563c -Wimplicit foo.c
    foo.c: In function main:
    foo.c:3: warning: implicit declaration of function ‘foo’
C:\> dir
    a.cld foo.c
```

-W*return-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 2-28. 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.

```
C:\> dir
    foo.c

C:\> type foo.c
    int foo(), main();
    int main()
    {
      return foo();
    }
    int foo(){}

C:\> g563c -Wreturn-type foo.c
    foo.c: In function foo:
    foo.c:6: warning: control reaches end of non-void function
```

```
C:\> dir
    a.cld foo.c
```

-W*unused*

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 2-29. 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.

```
C:\> dir
    foo.c

C:\> type foo.c
    static int foo();
    main()
    {
      int x;
      2+3;
    }
C:\> g563c -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

C:\> dir
    a.cld foo.c
```

-W*switch*

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.

-W*all*

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.

-W*shadow*

Warn whenever a local variable shadows another local variable.

-W*id-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.

-W*pointer-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.

-W*cast-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 ***.

-W*write-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.

2.2.3 Assemble Phase Options

This group of assemble phase options is the sub-set of the available assembler options that are compiler oriented (see the *Freescale DSP56300 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 **asm56300**, the DSP56300 assembler.

Pass a single command line option to the assembler.

```
C:\> g563c -asm -v file.c
```

Example 2-30. Pass multiple options to the assembler.

```
C:\> g563c -asm "-v -OS,CRE" file.c
```

-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"**.

2.2.4 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 *Freescale DSP56300 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 2.1 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 (**crt0563y.cln**) with *file*. **g563c** searches the standard list of directories to find the start-up file. In addition, any directory defined using the **-B** option or the **G563_EXEC_PREFIX** environment variable will be searched. For additional information, see Chapter 6.

Example 2-31. Compile the C program foo.c with the crt0 file crt.asm. Notice that the crt0 file crt.asm should be assembled before use since the option -crt takes .cln file not .asm file.

```
C:\> dir
  crt.asm    foo.c
C:\> g563c -c crt.asm
C:\>                                        dir

  crt.cln    crt.asm    foo.c
C:\> g563c -crt crt.cln foo.c
```

-j *string*

Pass the argument *string* directly to **dsplnk**, the DSP56300 linker.

Example 2-32. Pass a single option to the linker.

```
C:\> g563c -j -v file.c
```

Example 2-33. Pass multiple options to the linker.

```
C:\> g563c -j "-v -i" file.c
```

-l*LIBRARY*

Search the standard list of directories for a library file named **lib*LIBRARY*.clb**. The

linker automatically expands *LIBRARY* from the option command into **lib*LIBRARY*.clb** and uses this file as if it had been specified precisely by name.

Example 2-34. Compile the source code using the special dsp application library. Searching the standard list of directories for a library named **libdspaps.clb.**

```
C:\> g563c -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 DSP56300 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 *Freescale Linker/Librarian Reference Manual.*

Example 2-35. Compile the source code main.c and data.c with the memory configuration described in the control file map.ctl. Notice that the section main_c of the program main.c is located at the memory address p:$3000 and the section of data_c of the data data.c is located at the memory address y:$5000. See chapter 5 for detailed information on the in-line assembly code (_ _ asm(...)).

```
C:\> type map.ctl
section     main_c     p:$3000
section     data_c     y:$5000


C:\> type data.c
int data = 0x1; /* test value */


C:\> type main.c
extern int data;
main()
{
   int                                                    i;
```

Chapter 3
About g563c

3.1 Introduction

The DSP56300 digital signal processors are designed to execute DSP oriented calculations as fast as possible. As a by-product, they have an architecture that is somewhat unconventional for C programming. Because of this 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 DSP563CCC programming environment. *All programmers, whether they are familiar with DSP or not, should understand the DSP56300 family architecture before attempting to program it in C*. The following sections provide important information on data types, storage classes, memory and register usage, and other topics which will be useful to the DSP56300 application developer programming in C.

3.2 Identifiers

An identifier is defined as a sequence of letters, digits and underscore characters ('_'). The first character must be a letter or underscore. DSP563CCC identifier length limits are listed in Table 3-1.

Table 3-1 Identifier Length Limits

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

3.3 Predefined Preprocessor Macro Names

DSP563CCC supports and expands all ANSI defined macros and four additional non-ANSI predefined macro names. Table 3-2 lists the macros and their explanation.

3.4 Data Types and Sizes

Due to the word orientation of the DSP56300 family (24-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 3-2 Predefined Macro List and Explanation

MACRO	ANSI Required?	Explanation
_ _LINE_ _	YES	The line number of the current source line (a decimal constant).
_ _FILE_ _	YES	The name of the source file (a character string).
_ _DATE_ _	YES	The compilation date (a character string of the form "Mmm dd yyyy" e.g., Jul 22 1991).
_ _TIME_ _	YES	The compilation time (a character string of the form "hh:mm:ss").
_ _STDC_ _	YES	Decimal constant 1, indicates ANSI conformation.
_ _DSP563C_ _	NO	Decimal constant 1, indicates that code is being generated for the DSP56300.
_ _VERSION_ _	NO	The GNU version number of the compiler (a character string of the form "d.dd.d").
_ _INCLUDE_LEVEL_ _	NO	Decimal constant, indicates the current depth of file inclusion.
_ _MOTOROLA_DSP_ _	NO	Defined for all Freescale DSP C compilers.

3.4.1 Integral Data Types

The type **char**, **short int**, **int**, **long int** and the enumerated types comprise the *integral* data types. All but the enumerated types are available as unsigned types as well as signed by default. The type sizes and ranges are defined in Table 3-3. *Note that* **long int***s are stored in memory with the least significant word occupying the memory location with the smaller address.*

Table 3-3 Integral Data Type Sizes and Ranges

Data Type	Size (words)	Min value	Max value
char	1	-8388608	8388607
unsigned char	1	0	0xFFFFFF
short	1	-8388608	8388607
unsigned short	1	0	0xFFFFFF
int	1	-8388608	8388607
unsigned int	1	0	0xFFFFFF
long	2	-140737488355328	140737488355327
unsigned long	2	0	0xFFFFFFFFFFFF

3.4.2 Floating-point Types

In DSP563CCC, the C data types **float** and **double** are both implemented as **single** precision (see Table 3-4). DSP563CCC does **not** implement the IEEE STD 754-1985 standard format for binary floating-point arithmetic. A description of the format and a comparison with the IEEE standard follow.

Table 3-4 Floating-point Data Type Sizes and Ranges

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

3.4.3 Floating-point Format Description

Figure 3-1 illustrates the floating -point format used in DSP563CCC. Figure 3-1a shows that the exponent and mantissa occupy consecutive memory locations. Figure 3-1b is in number line format and shows the fractional nature of the mantissa and the fact that, due to the nature of a fractional arithmetic mantissa, the numbers between -0.5 and +0.5 (except for zero) are not needed and are therefore reserved. Figure 3-1c shows the range used by the exponent in this implementation. Notice how this compares with the IEEE implementation shown in Table 3-6. Figure 3-1d is a combined number line showing the range of numbers which can be represented in DSP563CCC. The mantissa $C00000 (-0.5) is not included as the smallest negative floating-point number because the normalization routine automatically detects the two leading ones and decrements the exponent which, if at $003FFF, will result in an underflow. Therefore, the smallest negative mantissa has been set to $BFFFFF (-0.916). Table 3-5 lists the specific floating-point format information for DSP563CCC and is a tabular version of the information in Figure 3-1.

3.4.4 Comparison of DSP563CCC and IEEE 754-1985 floating-point.

One major difference is the use of affine arithmetic in the IEEE standard versus the use of saturation arithmetic in the DSP563CCC format. Affine arithmetic gives separate identity to plus infinity, minus infinity, plus zero and minus zero. In operations involving these values, finite quantities remain finite and infinite quantities remain infinite. In contrast, DSP563CCC format gives special identity only to unsigned zero. This format performs saturation arithmetic such that any result out of the representable floating-point range is replaced with the closest floating-point representation. Since the dynamic range of this format is quite large, it is adequate for most applications.

The IEEE floating-point standard provides extensive error handling required by affine

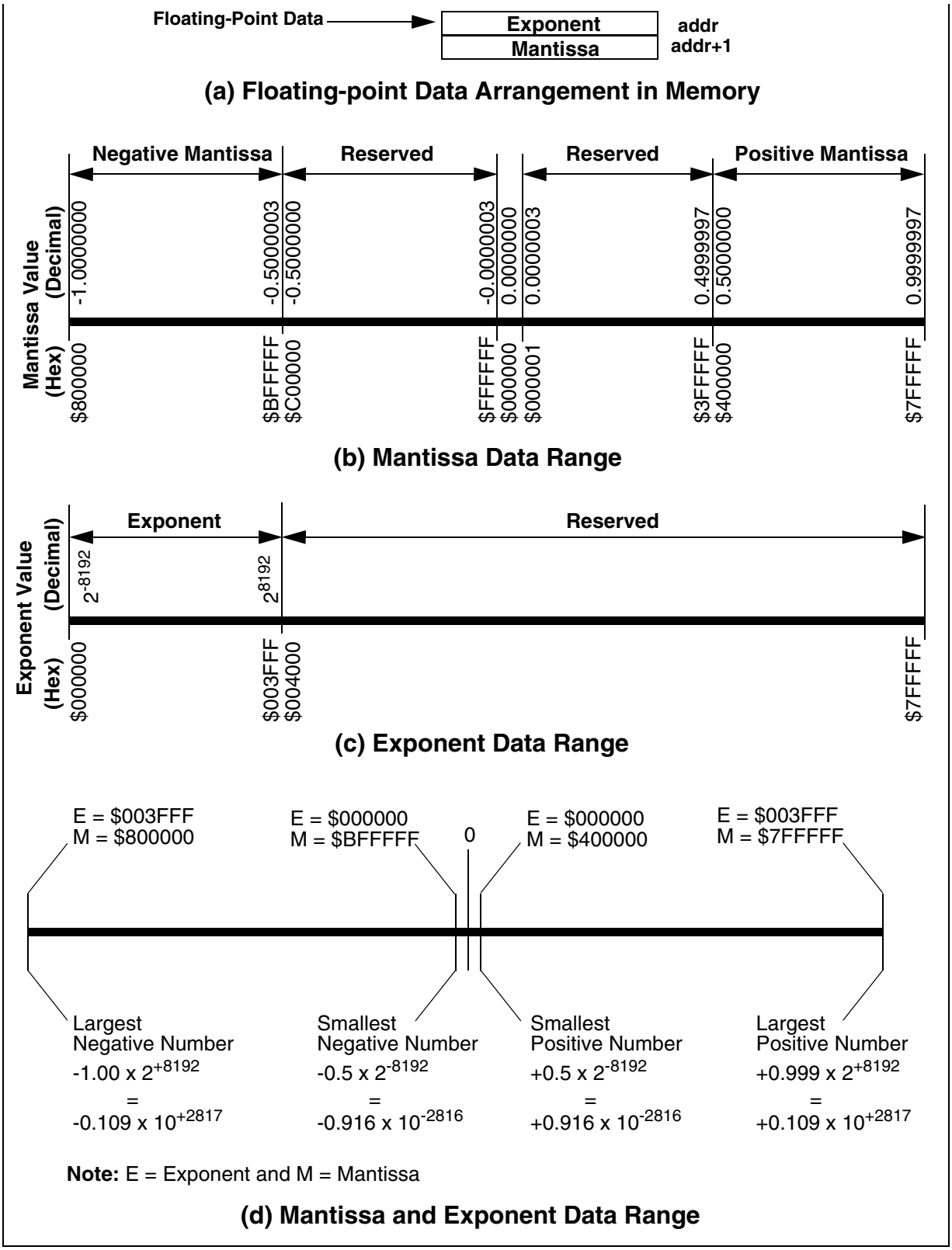

Figure 3-1 Mantissa and Exponent Data Range of C floating point

arithmetic, denormalized numbers, signaling Not-a-Numbers (NaNs) and quiet NaNs. It postpones the introduction of computational errors by using internal signaling and user traps to process each exception condition. Computational errors will be introduced by the application program if the calculation is completed instead of aborting the program. The DSP563CCC format introduces computation errors when an exception occurs in order to maintain real-time execution. An error flag (L bit in CCR) is set to inform the application program that an exception has occurred. This bit will remain set until reset by the application program. The user can then eliminate the exception by algorithm modifications.

Table 3-5 Floating-point Format Description

IEEE Format Characteristic	DSP563CCC Value
Decimal Value	$m * 2^{(e - ebias)}$
Mantissa	24-bit two's complement, normalized fractional mantissa. This gives a precision of approximately 7 decimal digits. A hidden leading 1 is not implemented in this format (see Figure 3-1).
Largest Positive Mantissa	$7FFFFF
Smallest Positive Mantissa	$400000
Floating-point Zero Mantissa	$000000
Smallest Negative Mantissa	$BFFFFF
Largest Negative Mantissa	$800000
Reserved Mantissas	$000001 through $3FFFFF and $C00000 through $FFFFFF
exponent	14-bit exponent (unsigned integer, biased by ebias = $1FFF). Stored as a 24-bit unsigned integer with 10 leading zeros. The 14-bit exponent used by DSP563CCC provides a larger dynamic range than IEEE double precision format.
Largest exponent (biased)	$003FFF = 2^{+8192}
Smallest exponent (biased)	$000000 = 2^{-8192}
Reserved exponents	$004000 through $FFFFFF

Notes:

1. No distinct exponents are reserved for plus infinity, minus infinity, Not-a-Number (IEEE NaN), minus zero or denormalized numbers as is done in IEEE format.
2. All reserved mantissas are illegal since they represent denormalized mantissas.
3. If the 15th bit is set, exponent overflow has occurred.
4. If the 16th bit is set, exponent underflow has occurred.

Table 3-6 Comparison of DSP563CCC and IEEE 754-1985

CHARACTERISTIC	DSP563CCC FORMAT	IEEE FORMAT
Mantissa Precision	23 bits	24 bits
Hidden Leading One	No	Yes
Mantissa Format	24-bit Two's Complement Fraction	23-bit Unsigned Magnitude Fraction
Exponent Width	16 bits (14 bits used)	8 bits 11 bits
Maximum Exponent	+8192	+127 (8 bit case) +1023 (11 bit case)
Minimum Exponent	-8191	-127 (8 bit case) -1022 (11 bit case)
Exponent Bias	+8191	+127 (8 bit case) +1023 (11 bit case)
Format Width	48 bits	32 bits (8 bit case) 64 bits (11 bit case)
Rounding	Round to Nearest	Round to Nearest Round to +/- Infinity Round to Zero
Infinity Arithmetic	Saturation Limiting	Affine Operations
Denormalized Numbers	No (Forced to Zero)	Yes (with Min Exp)
Exceptions	Divide by Zero Overflow Negative Square Root	Invalid Operation Divide by Zero Overflow Underflow Inexact Arithmetic

3.4.5 Pointer Types

With DSP563CCC, all pointers are 24**-bits** in size. When computing addresses with **long** arithmetic, only the **least significant 24-bits** are relevant.

3.5 Register Usage

The DSP56300 family digital signal processor register set is shown in Table 3-7. DSP563CCC uses all of the registers listed in Table 3-7 with the **exception** of the $\mathbf{m_n}$address modifier registers, which are assumed to be set to the linear addressing mode.

Table 3-7 DSP56300 Family Processor Registers

Data ALU		
$\mathbf{x_n}$	-	Input Registers **x1**, **x0** (24-bits)
$\mathbf{y_n}$	-	Input Registers **y1**, **y0** (24-bits)
$\mathbf{a_n}$	-	Accumulator Registers **a2** (8-bits), **a1**, **a0** (24-bits)
$\mathbf{b_n}$	-	Accumulator Registers **b2** (8-bits), **b1**, **b0** (24-bits)
x	-	Input Register **x** (**x1:x0**, 48-bits)
y	-	Input Register **y** (**y1:y0**, 48-bits)
a10	-	Input Register **a10** (**a1:a0**, 48-bits)
b10	-	Input Register **b10** (**b1:b0**, 48-bits)
a	-	Accumulator **a** (**a2:a1:a0**, 56-bits)
b	-	Accumulator **b** (**b2:b1:b0**, 56-bits)
Address ALU		
$\mathbf{r_n}$	-	Address Registers **r0**-**r7** (24-bits)
$\mathbf{n_n}$	-	Address Offset Registers **n0**-**n7** (24-bits)
$\mathbf{m_n}$	-	Address Modifier Registers **m0**-**m7** (24-bits)

Caution

The ***m_n address modifier registers*** *are not used directly by DSP563CCC. Some of these registers are implied whenever any address registers are referenced either in C library or in C. While assembly code can access and use these registers, the programmer* ***must*** *restore them to their previous state ($FFFFFF) before returning control to DSP563CCC. Failing to do so will cause* ***unpredictable errors*** *when compiled code uses the address registers.Again, the C compiler assumes that the modifier registers have been initialized for linear addressing.*

The programmer is required to preserve any registers that are directly used in *in-line* and in *out-of-line* assembly language code (see Chapter 5, *Mixing C and Assembly Language*). Table 3-8 outlines the compiler's usage of each register.

Table 3-8 DSP563CCC registers and Usage

Register	Usage
r6	Stack Pointer
r0 - r5, r7	Register promotion by the optimizer
n0 - n7	Code generator temporary
m0 - m7	Used by compiler; keep this as $FFFF
a	48-bit function return value. float, double, or long
	Also used to pass first parameter to function.
a1	24-bit return value. Integer or pointer
b	48-bit register promotion by optimizer
	Also used to pass second parameter to function.
x, y	48-bit register promotion by optimizer
x1, x0, y1, y0	24-bit register promotion by optimizer

3.6 Memory Usage

The DSP56300 memory can be partitioned in several ways to provide high-speed operation and additional off-chip memory expansion. Program and data memory are *separate*.

By default, the compiler expects that all memory spaces are fully populated and that several global C variables are defined in the **crt0** file (see Chapter 6 — *Software-Hardware Integration* for information about customizing the memory configuration). Figure 4-1 and Figure 4-3 illustrate the default program and data memory configuration.

3.6.1 Activation Record

An activation record is where a C subroutine stores its local data, saved registers and return address, etc. A typical DSP563CCC activation record consists of the following elements and is illustrated in Figure 4-2

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 4-2). However, the data is in reverse order and each parameter is referenced via a *negative* offset from the stack pointer. Actual parameters are pushed onto the activation record in reverse order by the calling subroutine.
2. Return address — which is pushed on the DSP's system stack high (**ssh**)

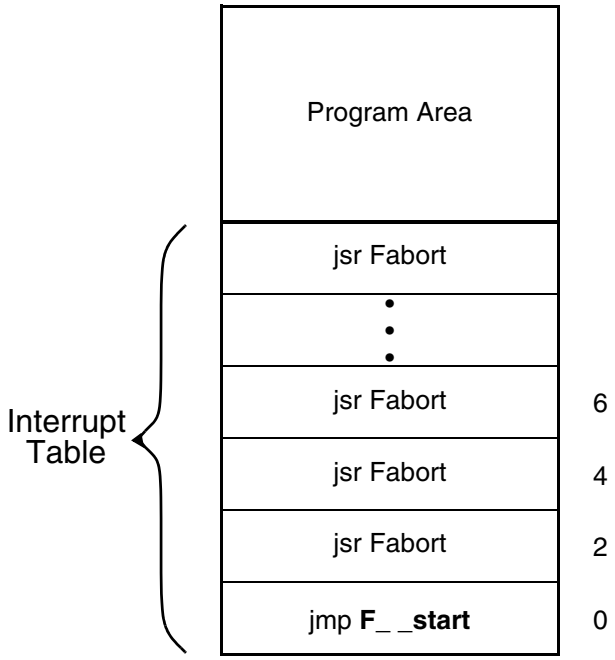

Figure 4-1 Default Program Memory Configuration

register. This is the return address to the calling subroutine. The return

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

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) generally points to the *next available* data memory location, but during the epilogue portion of a function, and during the execution of some of the code in the library, it's possible for the stack pointer to point to the *last used* location. For this reason, in ISRs that save items to the stack, the stack pointer should be incremented before stores to the stack are done, and decremented back to its initial position at the end of the ISR.

Each subroutine called generates a new subroutine activation record on the run-time stack. When it returns, the subroutine removes the activation record. The run-time stack is described in Figure 4-3, *Default Data Memory Configuration*. The variables in the crt0 file may be changed or relocated by the user. These variables are needed for the C

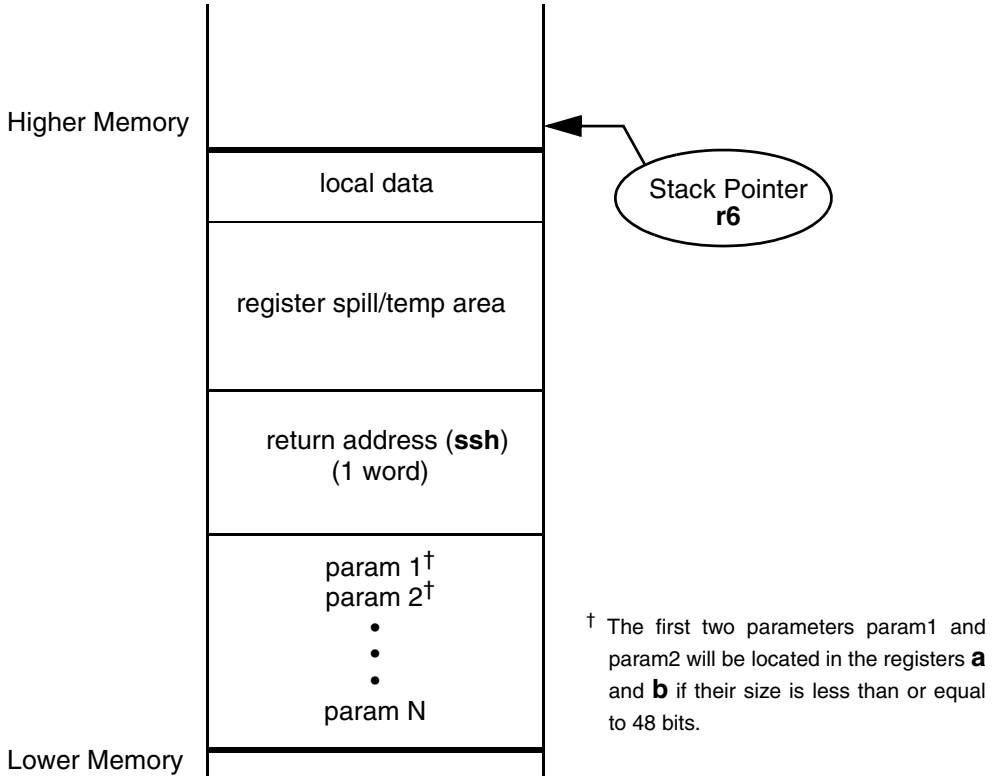

Figure 4-2 Typical Activation Record

run-time environment. In general, the linker expects that they will exist somewhere in memory, but it doesn't really care where. **DSIZE** is set by the linker and points to the top address of the global and static data. **DSIZE** is used in the **crt0** file as the default initial stack pointer.

Dynamic run-time stack growth is illustrated in Figure 3-1. In this example, there is one activation record as execution of the sample C code begins. This activation record is pushed onto the stack and a new activation record is built. When the function returns, the callee and caller work together to clear the callee's activation record from the stack. The register **r6** is used both as a stack pointer and as a frame pointer; references to local data are made with offsets of **r6**, and **r6** marks the next free location on the run-time stack. This means that a caller's activation record may be restored by simply restoring the value of **r6**.

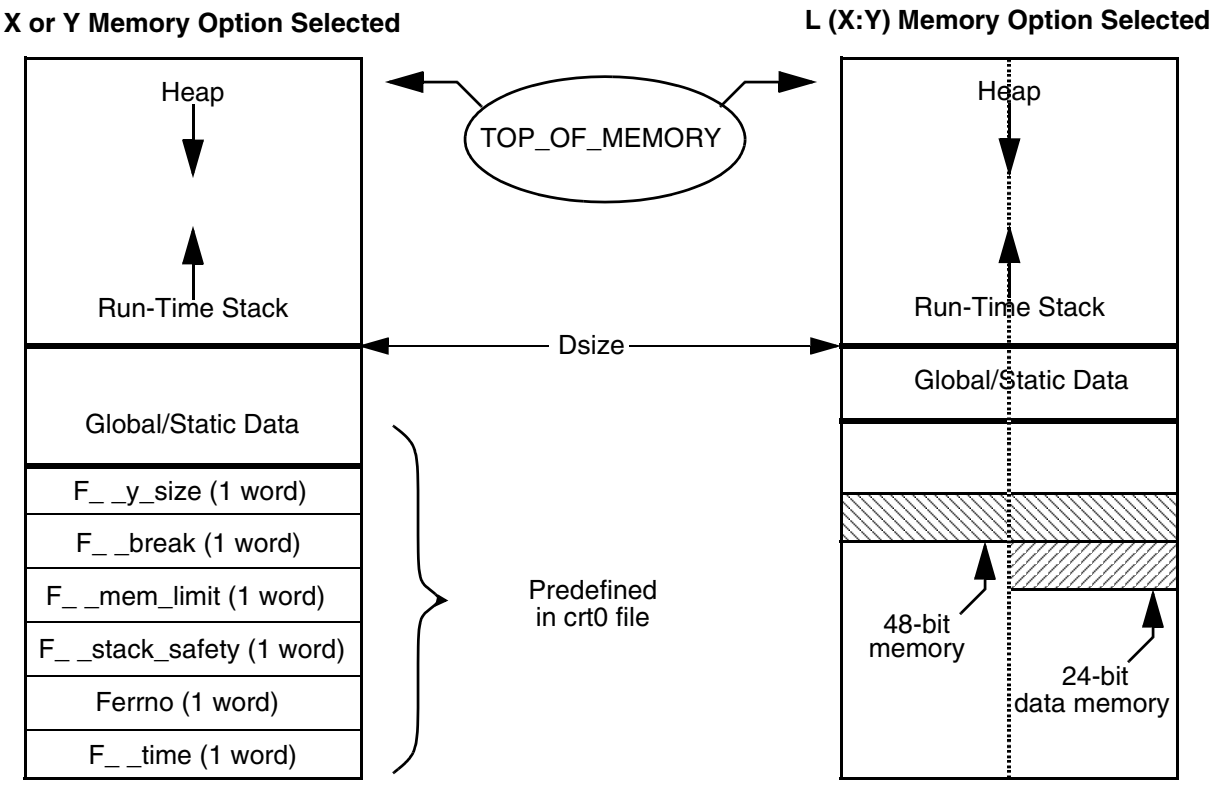

Figure 4-3 Default Data Memory Configuration

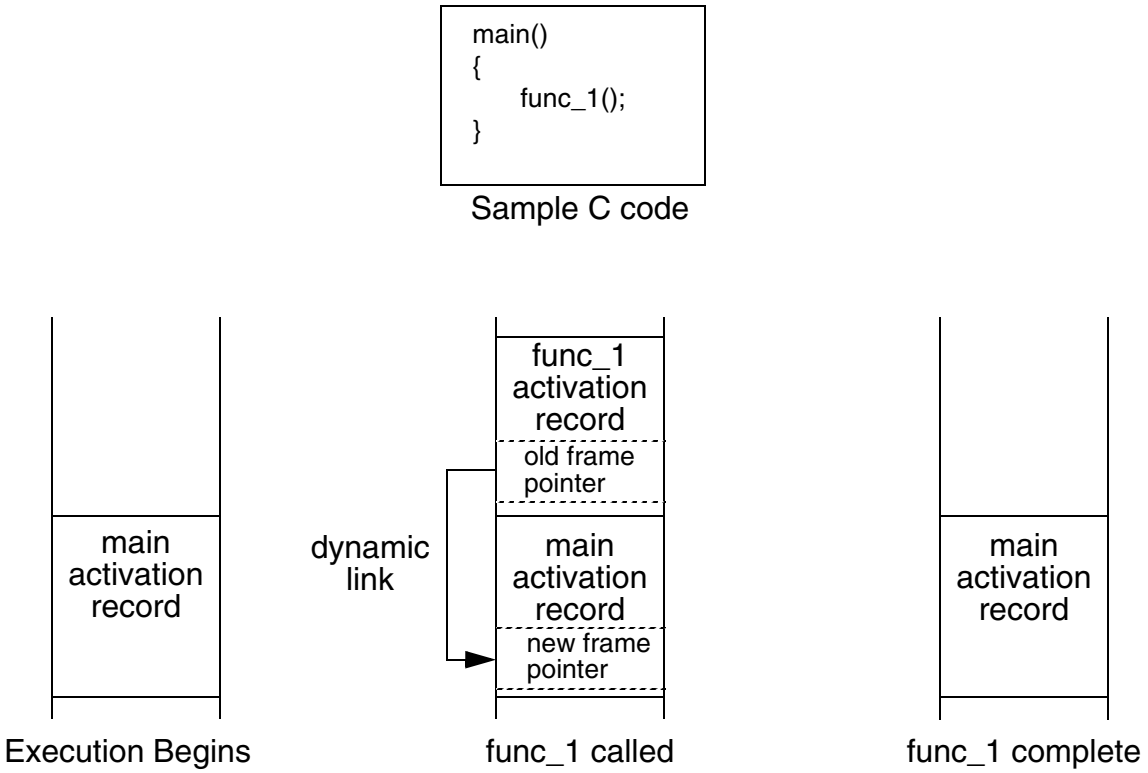

Figure 3-1 Run-time Stack Growth

3.6.2 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).

3.7 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 **"**hello world\n**"**.
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 **section**. Where *filename* is the file name minus any '**.**' extensions.

3.8 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.

3.8.1 Caller Sequence

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

1. save the following caller-save registers: x1, x0, r0, n0, r1, n1, r4, n4, r5, n5, a, and b.
2. pushing arguments onto the activation record (in reverse order),
3. actual subroutine call (**jsr**),
4. stack pointer adjustment.
5. restore the caller-save registers.

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

6. allocate space in the caller's activation record to store the return **struct**,
7. pass the return value *address* in register **r7**.

3.8.2 Callee Sequence

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

1. saving return address (**ssh**)
2. updating frame / stack pointer,
3. saving the following registers, as required: y1, y0, r2, n2, r3, n3, r7 and n7**.**

3.8.3 Return Sequence

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

1. placing the return value in accumulator a.
2. testing the return value. This optimizes the case where function calls are arguments to conditional operators. The return value need not be tested if the function is returning void, or a struct.

Additional *callee* sequence when the subroutine called will return a **struct**

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

3.9 Software Support for Arithmetic Routines

The DSP56300 family provides full hardware support for all 24-bit integer arithmetic operations, and partial support for 48-bit *integer* operations. Support for all float/double and a portion of the 48-bit long is provided via special software library routines. These special library routines *do not* pass arguments to the routines according to the normal subroutine calling convention for performance reasons.

3.10 Run-time Safety

DSP563CCC provides two methods for providing run-time memory utilization checks. The first method, heap memory allocation checking, is automatic. The second method, run-time stack probing, is provided by selecting the command-line argument **-m***stack-check*.

3.10.1 Memory Allocation Checks

Heap memory allocation checks are provided during every call to the run-time library routines **malloc**, **calloc** and **realloc**. These automatic run-time checks determine 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**.

3.10.2 Run-time Stack Checks

By selecting the **-m***stack-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.

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. The user is free to rebuild the library routines with **-m***stack-check* as needed.

3.11 Optimization Techniques Implemented

This section provides a brief overview of the optimization techniques currently included in DSP563CCC. Many machine-independent optimization techniques are used by DSP563CCC, along with some machine-specific optimizations as well. By default, the control program **g563c** enables all levels of optimization (see chapter 3, *Compiler Operation*, for command-line options to disable all or part of the optimizer) except post-pass instruction scheduling.

3.11.1 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 is performed to determine the best variables for promotion. When variable lifetimes do not overlap, more than one variable may be promoted to a single register.

3.11.2 Common Sub-expression Elimination

A **C***ommon* **S***ub*-**E***xpression*, 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 when CSEs become candidates for register promotion.

3.11.3 Constant Propagation

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.

3.11.4 Dead Code Elimination

During data flow analysis, the compiler detects when the results of C expressions are never used. When this is detected, the useless C statements are eliminated. This includes both the initialization of variables that are never referenced in the subroutine and back to back assignments.

To guarantee code generation for statements that have hidden effects, a **volatile** type

specifier can be used when declaring variables and functions.

```
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 function main.

3.11.5 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:

```
if ( a > b )
{
    b = a;
    func( a );
}
else
    func( a );
```

becomes

```
if ( a > b )
{
    b = a;
}

func( a );
```

3.11.6 Strength Reduction

Strength reduction replaces expensive operators with less expensive operators. This optimization method can be 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 DSP56300, however since a series of left shifts may be replaced with a single multiply by a constant power of 2.

3.11.7 Loop Invariant Code Motion

Loop Invariant Code Motion is a method in which all C expressions that yield the same result in each iteration of the loop are moved outside of the loop and are executed once prior to entering the loop.

3.11.8 Hardware DO Loop Instruction

The DSP56300 family architecture provides a method in hardware to perform zero

overhead looping via the **do** instruction DSP56300 may exchange the standard increment/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 contains *no* subroutine calls,
2. The loop is entered from the top, i.e., no **goto label** entries.
3. *No* conditional exits from the loop are allowed.
4. The loop's induction variable is only altered in the body of the loop once per iteration. Please note that this includes any modifications to the induction variable within the actual **for** or **while** statement.

3.11.9 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 ++**)

3.11.10 Jump Optimizations

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

3.11.11 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:

1. integer add/multiply becomes a **mac** instruction,
2. integer subtract/multiply becomes a **mac** instruction,

3. a memory reference combined with a pointer adjustment becomes an **autoincrement** or **autodectrement** 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.

3.11.12 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**).

3.11.13 Function In-lining

When explicitly requested via the command-line option **-f***inline-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.

3.11.14 Instruction Scheduling / Microcode Compaction

The command line switch **-alo** causes an assembly language optimizer (**alo563**) to be run, using the assembly code emitted by the compiler as input. This optimizer attempts to compact multiple operations into a single instruction word, while simultaneously avoiding

the pipeline hazards exposed by the address generation unit. Because this optimizer mixes together instructions from different C language statements, debugging code compiled with **-alo** may be more difficult.

Chapter 4
Mixing C and Assembly Language

4.1 Overview

In cases where the DSP56300 programmer requires direct access to the hardware or greater performance in the inner-loop of an algorithm, C can be mixed with assembly in the same program. 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 generated object **files**.

Caution

Before mixing C and assembly, read and understand Chapter 4, ***About g563c****, and the* ***DSP56300 Family Manual****. Attempting to write programs 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.*

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 ()).

4.2 In-line Assembly Code

In-line assembly code is assembly code that is inserted inside a C statement in a C 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 of the assembly code to be generated is visible at the C source-level and it is often convenient to intermix assembly code 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, small 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 code.

The keyword _ _**asm** is introduced as an extension to the ANSI C language standard. This keyword is used in a fashion similar to a function call in order to specify in-line assembly code generation.

The in-line assembly statement syntax is:

_ _**asm** (instruction_template: output_operands: input _operands**:** reg_save)**;**

where:

1. **instruction_template** is a string used to describe a sequence of assembly code instructions that are to be inserted into the compiler output stream. It may specify arguments, which are listed in output_operands and input_operands. It does this via a substring called an operand expansion string (OES). An OES starts with a '%'. OES and instruction_template interpretation is described in Section 4.2.1.

2. **output_operands** are optional operands used to provide specific output information to the compiler. Each output_operand string is separated by a comma and should begin with the character '='. As an example, the output_operand **"=A" (cptr)** means "the C variable **cptr** will get its value from this output operand, which will be in an address register". See Section 4.2.2 for more details.

3. **input_operands** are optional operands to provide specific input information to the compiler. Each input_operand is separated by a comma and may include a C variable. As an example, the input_operand **"A" (cptr)** means "the value of this input operand will be taken from the C variable **cptr**, and placed in an address register". Again, full descriptions of the input and output operands can be found in Section 4.2.2.

4. **reg_save** specifies registers that are to be explicitly reserved for the _ _asm () statement. The registers to be saved must be named as strings such as**"r1"**. Each register is separated by a comma (see Section 4.2.3 for additional information) The compiler assumes that all data residing in the reg_save registers will be invalidated by the _ _ asm () statement.

4.2.1 Instruction Template

The first *argument* of the _ _**asm()** extension is the instruction template or assembler instruction template. This instruction template is mandatory, and describes the line or lines of assembly language to be placed into the compiler's assembly language output (see Example 4-1. in Section 4.2.4). 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 will be caught by the assembler.

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 sequence, a '\' followed by an immediate newline, is particularly useful when an instruction template contains an assembly instruction that is too long to fit in one source line (see Example 4-18. in Section 4.2.4). Other C language character constants such as '\t', '\f', etc. can also be used in the instruction template.

In many situations, it is desirable to use the values of C variables and/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 DSP563CCC 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 location is unknown. The operand expansion string is a substring of the instruction template and begins with the character '%'. This string is usually two or three characters long and provides the compiler with special information about an operand, and how its reference should be printed. An OES must reference only one C variable or expression, which in turn must be listed in either one or both operand lists (see Section 4.2.2). 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 4.2.4 include an OES.

The OES syntax is:

% [modifier] operand_id_number

where:

1. **modifier** is a single optional character which specifies a particular class of operand. The available modifiers are '**j**', '**e**', '**h**', '**k**', '**g**', '**i**', '**f**', '**p**' and '**q**'.

   j — an offset register (**nx**) associated with the corresponding address registers (**rx**). Since the offset register is paired with the address register, the allocated offset register has the same index as the address register (see Example 4-3. in Section 4.2.4).

   e — **a1** or **b1**, upper word of the destination registers **a** or **b** (see Example 4-4. in Section 4.2.4).

   h — **a0** or **b0**, lower word of the destination registers **a** or **b** (see Example 4-5. in Section 4.2.4).

   k — **a2** or **b2**, extension register of the destination register, **a** or **b** (see Example 4-6. in Section 4.2.4).

   g — Select the 24--bit portion of the 48-bit ALU register (**x** or **y**) that is not occupied by data pointed to by the operand id — e.g., if the operand id points to **x0** then **x1** is selected and similarly **x1**→**x0**, **y0**→**y1**, **y1**→**y0** (see Example 4-7. and Example 4-8. in Section 4.2.4).

   i — strip the **0** or **1** from the allocated register name i.e., **x0**→**x**, **b1**→**b** (see Example 4-9. in Section 4.2.4).

   f — insert the memory space identifier (**x** or **y**) for the memory location (see Example 4-10. in Section 4.2.4).

2. **operand_id_number** specifies the operand location in the operand descriptor list (see Example 4-2. in Section 4.2.4). The operand descriptor list is a concatenation of the output operands and input operands (see Section 4.2.2). The first operand is labeled 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. 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 4-19.). 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 3.7, Compiler Naming Conventions).

4.2.2 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 4.2.1). 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, leaving us with the empty list of output operands.

The operand syntax is:

“[=]operand_constraint” (C_expression)

where:

1. **=** differentiates input and output operands. All output operands must use this character first.
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**”, “**R**”, “**S**”, “**N**”, “**r**”, “**i**” and “**m**”. All of these constraints originate from the DSP56300 architecture: a full understanding of these constraints requires that the programmer understand said architecture. The constraints are:

A —One of the **Address Registers** (**rx**, where **x** = **0** through **7**; see the *DSP56300 Family Manual*) will be allocated, (see Example 4-3.) and the C expression will be promoted to this register. Typically the C expression should be a pointer to be assigned to an address register. The OES modifier, **j**, can only be associated with operand constraint **A** (see Section 4.2.1).

N —One of the **Offset Registers** (**nx**, where **x** = **0** through **7**; see the *DSP56300 Family Manual*) will be allocated, (see Example 4-3.) and the C expression

will be promoted to this register. Registers allocated to “**N**” and “**A**” are not

guaranteed to be paired (i.e. **r3** and **n3**) by the compiler.

D —One of the 56-bit accumulators (**a** or **b** which are referred to as **Destination Registers**; see Section 4 of the *DSP56300 Family Manual*) will be allocated (see Example 4-4. through Example 4-6.), and the C expression will be promot-

ed to this register. The OES modifiers ‘**e**’, ‘**h**’ and ‘**k**’ can be associated with operand constraint **D** (see Section 4.2.1).

R —One of the Input Registers (**x0** or **y0** which are also called **Source Registers;** see Section 4 of the *DSP56300 Family Manual*) will be allocated to the C expression (see Example 4-9.). The C expression will be promoted to this register. The OES modifiers ‘**g**’ and ‘**i**’ can only be associated with operand constraint R (see Section 4.2.1).

r —One of the General Registers (**a**, **b**, **x0**, **x1**, **y0**, **y1**, **r0-r5**, **r7**, **n0-n5, n7**). This operand constraint is useful when one wants a scratch register that won’t be used in an instruction other than a **move**.

S —One of the Input Registers (**x0**, **x1**, **y0** or **y1** which are also called **Source Registers;** see Section 4 of the *DSP56300 Family Manual)* will be allocated to the C expression (see through Example 4-15.). The C expression will be promoted to this register. The OES modifiers ‘**g**’ and ‘**i**’ can only be associated with operand constraint **S** (see Section 4.2.1).

i —An immediate constant; a constant is generated in the form of #constant if no modifier is specified.

m —The C expression will be referenced in memory (see Example 4-10.). The DSP56300 has three memory spaces: **y:, x:** and **p:**, but the C compiler will only use the **y** memory space for this expression. The OES modifier ‘**f**’ can only be associated with operand constraint **m** (see Section 4.2.1).

number —Inherit all memory or register characteristics of the operand indicated by the operand id number (see Example 4-2.). 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 an input (see Example 4-12.). It is called a write-only operand if it is used only as an output (see Example 4-13.). In most cases, the operand is used as both an input and an output operand (see Example 4-14. and Example 4-15.). In these cases the operand must be described as both. Since output operands should be listed first, the operand id number is determined when the output operand is declared. The id number will be used as the operand constraint of the associated input operand.

4.2.3 Explicit Register Saving

It is possible to manually perform register allocation. This may simplify the process of

converting an existing body of DSP56300 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 4.2). This guarantees that the compiler won't expect values to be preserved in these registers across _ _asm() calls. Modification of the register **r6** is prohibited in the assembly code because it is reserved for the C compiler during run-time, where **r6** is the stack pointer. **n** registers are used by the compiler as temporary registers and **m** registers are assumed to be set for linear addressing. 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. If an **m** register is to be modified, then its original value must be restored by the programmer.

Explicit register saving is done by specifying the registers to be saved. A string is used to specify each register.

4.2.4 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 4-1. illustrates the use of the in-line assembly code instruction_template. Since this in-line assembly code directly clears register **a**, the programmer should check to be sure that the contents of **a** are not needed. The correct way of doing this would be to include "**a**" in the reg_save section of the _ _ **asm ()** statement.

Example 4-1. **Instruction_template:** The following are a few examples of how to utilize the instruction template with in-line assembly code. This feature allows the generation of any valid assembly instruction and it is probably the most frequently used feature with in-line assembly coding.

```
_ _asm("clr    a"); /* clears the register A */
_ _asm("move   #$10, a2"); /* load the register A2 with the hex value 10 */
_ _asm("\nABC equ  $ffc4"); /* equate the symbol ABC to $ffc4   */
```

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 4-2. 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 4-9 shows which operands in Example 4-2. are input or output operands.

Example 4-2. **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);

Examples Example 4-3. through Example 4-11. illustrate the use of OES modifier (see Section 4.2.1).

Table 4-9 Output and Input Operands for Example 4-2.

Example 4-3. **OES modifier j:** The following in-line assembly code is used to generate executable assembly code. Notice that the actual register selection is totally dependent on the C compiler but the register selected (**r3** in this example) is guaranteed to be related to the C expression used (in this case cptr, see Section 4.2).

In-line Assembly code:

```
char *cptr;
_ _asm(“move (%0)+%j0”::“A”(cptr));
```

Assembly Code Generated:

```
move (r3)+n3
```

Example 4-4. **OES modifier e:** The modifier **e** can be used to generate the assembly code below because **a1** is the upper part of register **a**.

In-line Assembly code:

```
int foo;
_ _asm(“move #$ffff,%e0”:“=D”(foo));
```

Assembly Code Generated:

```
move #$ffff,a1
```

Example 4-5. **OES modifier h:** The **h** modifier can be used to generate the following assembly code because **a0** is the lower part of register **a**.

In-line Assembly code:

```
int foo;
_ _asm(“move #$ffff,%h0”:“=D”(foo))
```

Assembly Code Generated:

```
move #$ffff,a0
```

Example 4-6. **OES modifier k:** The **k** modifier can be used to generate the following assembly code because **a2** is the extension portion of register **a**.

In-line Assembly code:

```
int foo;
_ _asm(“move #$ff,%k0”:“=D” (foo));
```

Assembly Code Generated:

```
move #$ff,a2
```

Example 4-7. **OES modifier g:** Swap the most significant 24-bit portion and the least significant 24-bit portion of 24-bit registers **x** and **y** to allow the **OR** instruction to operate on an entire 24-bit register.

```
/*
* The following assembly code could be generated (note that the
  * optimizer may vary the code actually generated).
                move x1,a1
                move x0,x1
                move a1,x0
  *
  * The variable foo can be allocated to either x0, x1, y0, or y1
  * by using the operand constraint S. The swap operation can
  * be applied to the register allocated to the variable foo by
  * using the following in-line assembly code.
  *
  */
main()
{
  int foo;
  _ _asm volatile ("move %g0,a1" : : "S" (foo));
  _ _asm volatile ("move %0,%g0" : "=S" (foo) : "0" (foo));
  _ _asm volatile ("move a1,%0"  : "=S" (foo));
}
```

Example 4-8. **OES modifier g:** A bit checker program looks to see if any bit in the 48-bit registers **x** or **y** is set. The example code looks to see whether the variable **foo**, which is placed in either the **x** or **y** register, is zero or contains a set bit. The result is stored in the register **a1**. If register **a1** is not zero then foo has one or more set bits.

```
/*
```

```
 * The variable foo can be allocated to either the x or yregister by using
 *  the operand constraint S. The or instruction only operates on 24-bit
 * registers so that to OR the x register with another register, x1 must
 *  be ored separately from x0. The same applies for the y register.
 */
main()
{
  long volatile foo;

  _ _asm volatile ("clr a");
  _ _asm volatile ("or %0,a" :: "S"(foo));
  _ _asm volatile ("or %g0,a" :: "S" (foo));
}
```

Example 4-9. **OES modifier i:** The modifier can be used to generate the following assembly code because **x** is a register without a **0** or **1** portion.

In-line Assembly code:

```
int foo;
_ _asm("move l:<$0,%i0" : "=R"(foo));
```

Assembly Code Generated:

```
move l:<$0, x
```

Example 4-10. **OES modifier f:** The f modifier can be used to generate the following assembly code. Assuming that the memory location of the variable "foo" is 233, then the memory space indicator "**y:**"will be automatically generated by the f modifier.

In-line Assembly code:

```
int foo;
_ _asm("move #$ff,%f0": "=m" (foo));
```

Assembly Code Generated:

```
move #$ff,y:233
```

Example 4-11. **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 4.2.2) and operand id (see Section 4.2.1 and Example 4-2.). The example implies that the C expression output_expression is expanded with constraint D and is an output of the assembly code instruction asm_instruction. Similarly, the C expression input_expression is expanded with constraint **S** and used as an input to the assembly code instruction asm_instruction.

```
_ _asm("asm_instruction %1,%0" : "=D" (output_expression) : "S" (input_expression));
```

Example 4-12. **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" :: "S" (input_expression));
```

Example 4-13. **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 4-14. **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 constraint '**0**' was used to reference the **add** instructions second source operand which is also the destination operand (see the *DSP56300Family Manual* — Appendix A for the syntax of the **add** instruction).

```
int foo, result;
_ _asm("add %1,%0" : "=D" (result) : "S" (foo), "0" (result));
```

Example 4-15. **Read-Write Operand:** The same result will be obtained as in Example 4-14. 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), "S" (foo));
```

Example 4-16. **Multiple Instruction — Single-Line:** An in-line assembly program which places a value (e.g. $709) in register a 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,a\n neg a" : : "a" );
```

Example 4-17. **Multiple use of _ _asm().** This trivial example and Example 4-18. are done in-line with the compiler performing all register allocation and all operands are referenced via C expressions. The method used to write this in-line assembly program is to use an _ _asm() statement for each assembly language instruction.

```
int read_n_add ( int data, int* ptr_a, int* ptr_b )
{
        int tmp_a, tmp_b;
        _ _asm ( "move x:(%1),%0" : "=S" (tmp_a) : "A" (ptr_a) );
        _ _asm ( "move x:(%1),%0" : "=S" (tmp_b) : "A" (ptr_b) );
        _ _asm ( "add  %1,%0" : "=D" (data) : "S" (tmp_a), "0" (data) );
        _ _asm ( "add  %1,%0" : "=D" (data) : "S" (tmp_b), "0" (data) );

         return data;
  }
```

Example 4-18. **Line Separation.** This in-line program is functionally identical to Example 4-17. except that line separation is used to insert the entire assembly language program. Notice how much easier it is to read the program.

```
int read_n_add ( int data, int* ptr_a, int* ptr_b )
{
        int tmp_a, tmp_b;
        _ _asm ( "\
        move x:(%3),%1                  \n\
        move x:(%4),%2                  \n\
        add  %1,%0                      \n\
        add  %2,%0" : "=D" (data), "=S", "=S" : "A" (ptr_a), "A" (ptr_b));

         return data;
  }
```

Example 4-19. **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");
```

4.2.5 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. *In general, this practice is discouraged*. 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.

4.2.5.1 Calling Assembly Subroutines

Calling a subroutine or function requires using a label that points to the subroutine or function. The C compiler uses a predetermined labeling convention (see Section 3.7). In order to call assembly subroutines labeled in an arbitrary fashion, _ _asm() can be used to overwrite the C convention label with an arbitrary label.

To illustrate how to use the _ _asm directive for this purpose, Example 4-20. reads the data at x memory location $100 and y memory location “X+2”. For test purposes, the y memory space is filled with the integer sequence 0 through 9. The printf() statement prints the data returned from the function calls ValOfX(100) and ValOfX(X+2). This function was written in assembly code and resides in another file.

By using the statement

```
extern int ValOfX() _ _asm(“ReadX”);
```

all C compiler generated function labels for ValOfX() are replaced by the label ReadX.

Example 4-20. Calling assembly from C. This C program (called test.c) can be used to examine the data in y memory by calling the assembly routine “ReadX”.

C:\> type test.c

```
#include <stdio.h>
extern int ValOfX() _ _asm(“ReadX”);
unsigned X[] = {0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8,0x9};
main()
{
   printf(“<%x><%x>\n”, ValOfX(100), ValOfX(X+2));
}
```

The following two command lines test Example 4-20..

```
C:\> g563c test.c memread.asm
C:\> run563 a.cld
```

4.2.5.2 Calling C Subroutines from Assembly Code

Any C function can be called from an assembly program to test the assembly program data or utilize built-in standard C libraries such as floating-point operations. Calling a C subroutine from assembly code requires using the C subroutine calling convention (see Section 3.8 and Section 4.4.4) and matching the C function labels. The in-line-assembly directive, **_ _asm()**, can be used as shown in Example 4-21. to change the C program labels.

Example 4-21. **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.

```
C:\> type c_print.c

#include <stdio.h>
int C_printf() _ _asm("print");
C_printf(char *msg)
{
printf("%s\n", msg);
 }
```

Example 4-22. **Calling C from assembly.** This assembly program (called greeting.asm) prints the message "greeting: hello, there" on the screen. It uses the C subroutine **printf ()**, to print this message. Notice that the assembly program name is **Fmain** because the control program, **g563c**, uses the default start-up file **crt0563y.cln**.

crt0563y.cln uses **Fmain** to start up the main program

```
C:\> type greeting.asm
        section   greeting
        org       y:
 LC0    dc        “greeting: hello,there.”,$00
        org       p:
        global    Fmain
Fmain   move      ssh,y:(r6)+
        move      #LC0,a
        jsr       Fprintit
        move      y:-(r6),ssh
        rts
        endsec
```

The following two MS-DOS command lines can be used to test the program:

```
C:\> g563c greeting.asm c_print.c
C:\> run563 a.cld

greeting: hello, there.
```

4.2.5.3 Referencing Assembly Global Variables from C

The data in assembly language programs must be accessible to C programs to take full advantage of the DSP56300 family architecture since the C language cannot access all of the DSP56300 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. Detailed information on modulo buffers can be found in the *DSP56300 Family Manual.*

Example 4-23. Generate data with assembly language. The data file, sqtbl.asm, is generated in assembly language and consists of a

Example 4-24. series of squares.

C:\> type sqtbl.asm

```
section data
            global table
            org     y:
 table      dc 0,1,4,9,16,25,36,49,64
            endsec
```

Example 4-25. Access data with C. This test program (called test.c) prints the value of 5^2 on the screen.

C:\> type test.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 4-25. test the two programs sqtbl.asm and test.c.

C:\> g563c test.c sqtbl.asm
C:\> run563 a.cld

4.2.5.4 Referencing Global C Variables from Assembly Language

One DSP563CCC feature is that global data in a C program is available to assembly language programs. This feature is particularly useful when the data to be processed by an assembly language program is generated by the C program.

Example 4-26. Generate data with C. data.c contains the coefficients of an average filter which takes the average of the last four input data.

C:\> type data.c

```
int Cwaddr[]   _ _asm("cwaddr");
int Ccaddr[]   _ _asm("caddr");
int NTAP       _ _asm("N_1");

int Cwaddr[4];
int Ccaddr[]  = {0x2000, 0x2000, 0x2000, 0x2000 };
int NTAP = 4;
```

4.2.6 Specifying Registers for Variables

DSP563CCC 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("r3");
```

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.

4.2.7 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 ( … );
```

4.3 #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 Freescale 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 x and p. When an object is or defined, the current values of those counter strings are bound to that object.

Syntax for the pragma directive is

#pragma counter_string argument

C function or data storage definition

#pragma counter_string

where

5. the two #pragma statements must encase the entire definition.
6. *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**.
7. 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 x 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**.
8. the C function or data storage definition is a declaration that reserves storage.
9. 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**.

The following two examples illustrate that the load time counter is optional. See the section on the ORG statement in the *Freescale DSP 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, the following C segment:

```
#pragma x_load p:$100
int coeff[5] = {0x19999a, 0x200000, 0x266666, 0x2ccccd, 0x333333};
#pragma x_load
```

produces the following assembly language code:

```
        global  Fcoeff
        org     x:,p:$100
Fcoeff
        dc 1677722
        dc 2097152
        dc 2516582
        dc 2936013
        dc 3355443
```

Notice that the second **#pragma** directive will remove the effect of the first memory specification, i.e., **#pragma x_load p:$100**; any following data definitions would have x memory load time locations, as is the default.

The above example code will be loaded at p memory location $100, and it should be copied to the x memory space upon system start-up. When burning a PROM, often 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 upon system start-up.

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

```
#pragma x_load p:$100
int coeff[5] = {0x19999a, 0x200000, 0x266666, 0x2ccccd, 0x333333};
#pragma x_load
```

produces the following assembly language code:

```
                global  Fcoeff
                org     x:,p:$100
Fcoeff
                dc 1677722
                dc 2097152
                dc 2516582
                dc 2936013
                dc 3355443
```

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 x 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**.**

4.4 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 DSP56300 family architecture. This out-of-line method may be used to convert existing assembly subroutines, or new subroutines completely in assembly language may be written. The key advantage of out-of-line assembly code is that it provides a complete assembly programming environment for the DSP56300 family 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 DSP56300 family 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. Function core
4. Restore all registers used
5. C subroutine exit code (epilogue code)

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.

4.4.1 General Template

The following template is a generic form used to make the C function “foo”. 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 the actual code depends on the function.

```
        global      Ffoo          ; prologue:
Ffoo                              ; sets up entry point (C function address).


        move        #k,n6         ; k is the amount of local space needed.
        move        ssh,y:(r6)+   ; save the return address.
        move        (r6)+n6       ; allocate local stack space of size k.

; Save all registers to be used
; Function body.
; Restore all registers used

        move        #k+1,n6
        move        (r6)-n6       ; deallocate local stack space, set ccr flags.
        tst         a
        move        y:(r6),ssh    ; get the return address.
        rts
```

4.4.1.1 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 return address and allocates any needed stack space for the function. Increasing the stack pointer value will protect local data from corruption by interrupt routines. The return address is saved when the **jsr** instruction pushes the

program counter onto the high 16 bits of the system stack (**ssh**).

4.4.1.2 Save all registers

All registers used in the function should be saved before the function alters 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 (except during the epilogue portion of a function, and during some of the routines in the library, where it can sometimes point to the last used element). The following statement saves one register to the top of stack and sets the stack pointer to the next available stack location.

```
move    r1,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.

4.4.1.3 Main Program

A typical C function accesses the parameters passed, executes using the parameters and returns a value. Passing parameters is done by pushing them onto the stack. When a function is called, the first parameter is directly underneath the stack pointer (see Section 3.6.1). The parameters are pushed by the caller in reverse order. For example, the following statement should be used to move the first single word parameter to register **r3**.

An exception to the above rule is when either the first or both the first and second parameter to a function will fit in accumulators. If the first parameter to a function is an int, long, float, pointer or double, then it will be passed in a. If the second parameter is also of one of those types, then it will be passed in b. If not, then the first and/or second paramter will be passed in the same manner as all subsequent parameters: on the stack.

```
move    y:(r6-z-2),r3                    ; z is local stack area size.
```

Assuming the first three parameters on the stack are one word long, the following statements move the second and third parameters on the stack to registers **r1** and **r2,** respectively.

```
move    y:(r6-z-3),r1
move    y:(r6-z-4),r2
```

4.4.1.4 Restore all registers

The stack pointer is needed to restore the registers. The following code will restore one

register. At this point in the function's execution, the stack pointer points to the location above the last saved register, hence the pre-decrement.

```
move    y:-(r6),r1                    ; restore
```

The restoring procedure can be simplified if more than one register is to be restored. Restoring registers **r1, r0** and **r3** can be done by the statements below.

```
move    (r6)-
move    y:(r6)-,r1
move    y:(r6)-,r0
move    y:(r6),r3
```

After the function has finished, a return value can be passed to the caller. Any 48-bit or 24-bit value must be returned through register **a**. If the return value is larger than 48-bits, then the compiler allocates the proper amount of buffer space and register **r7** becomes the pointer to this buffer space *upon callee execution*. It is the callee's responsibility to copy any return values from the buffer whose address resides in register **r7**. This is the method used for returning **struct**s.

4.4.1.5 Epilogue

The out-of-line template epilogue is the reverse of the prologue. The epilogue restores the stack pointer and the return address. In addition, register **a** is tested to update the **ccr** flags. This testing is a part of the C compiler code generation feature and should be included in functions that return values in the **a** register (see Section 4.4.5 for optimization).

4.4.2 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 4.2.5.

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 input register **x0** in assembly code as follows:

```
move    x:FGinteger,x0
```

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 programmer's 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 4-27. **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.

```
            org     y:
            global  FGinteger
FGinteger
            dc      $0
```

Example 4-28. **Global Variable Declaration.** This is the C code equivalent to Example 4-27. which defines the global integer **Ginteger.**

```
int         Ginteger;
```

Example 4-29. **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.

```
int Ginteger _ _asm("Ginteger");
```

Which will appear in assembly language code as:

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

4.4.3 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 should 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 4-30..

Example 4-30. **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 for pipeline delay.
move    (r2)+n2         ; 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 4-31. **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

Example 4-32. replaced with the appropriate constant.

```
move    #-size,n6       ; the stack size used before
nop                     ; wait until n6is available.
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.

4.4.4 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.
   NOTE: If either the first argument (or both the first and second argument) may be passed in an accumulator, it (they) must be passed that in an accumulator rather than on the stack.
2. Make the subroutine call.
3. Restore the stack pointer.

4.4.5 Optimization Techniques

The general template for out-of-line assembly code provides a clean template to build C callable functions. However, the DSP56300 family 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 *DSP56300 Family 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 return address (**ssh**) was saved in the out-of-line assembly code prologue but it is only required when a function calls another function. A function is called a leaf function if it does not call any other C function. In leaf functions, the return address does not have to be saved because the hardware stack will not overflow on subsequent **jsr** instructions.

The test statement "**tst a**" in the epilogue can be eliminated if the function does not return any value. 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 **jne** instruction is issued. This is primarily because the C compiler uses the condition flags which were generated at the end of the epilogue of **foo()**.

A variety of optimizations can be achieved by combining the **move** instructions and code to utilize parallel moves. These and other DSP56300 specific optimizations can dramatically improve the quality of the application specific library routines. A careful review of the *DSP56300 Family Manual* will be worthwhile for efficient library development.

Chapter 5
Software-Hardware Integration

5.1 Overview

This chapter explains how the run-time environment may be changed and provides examples of some changes and 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.

5.2 Run-Time Environment Specification Files

The run-time environment is specified by three assembly language files: **crt0563[xyl].asm**, **signal563[xyl].asm** and **setjmp563[xyl].asm,** where x, y, or l denote the memory model. These files may need to be modified if the run-time environment is to be customized.

The **crt0** file contains the C bootstrap code, parameters that specify memory configuration, memory management, interrupt vectors, and other miscellaneous code. This file must be modified to match the software and hardware configuration, as both the memory configuration and interrupt vectors are determined by hardware. The information in this manual on the **crt0** file applies to DSP563CCC.

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 DSP56300 family interrupt mechanism in addition to the semantics of 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 kept consistent. The source code for **setjmp()** and **longjmp()** is provided with DSP563CCC to allow modification, should the signal mechanism need to be changed.

The operation of **setjmp()** and **longjmp()** is described in Section 5.5 and detailed implementation information can be obtained from the files provided with DSP563CCC.

5.3 The crt0 File

The following subsections describe the various functions of the **crt0** file.

5.3.1 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 / static data area, stack area, heap area, etc. This environment must be established before C programs can execute correctly.

The following bootstrapping steps are normally taken before the processor starts to execute C code:

1. Jump from the chip reset vector to the C bootstrap code labeled at **F_ _start** in **ctr0**. 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 *Freescale DSP56300 Family 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.

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. Call **main()** with instruction **jsr Fmain**. Notice that the label is **Fmain** and that there are no parameters passed to the main function. Typical C compilers passes two or three arguments to **main()**, however **g563c** does not pass any arguments as DSP563CCC does not support a particular hosted environment.

The bootstrap code is followed with the label **F_ _crt0_end**. This label is used by **run563** to detect program termination.

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 5-1 **DSP56300 Operation Mode Change:** Mode 2 has a reset vector of $E000 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 Mode2.

```
section mode2_reset
org p:$e000
jmp F_ _start              ; jump to the C start-up program.
endsec
```

Example 5-2 Hardware was designed to have a 256 byte ROM monitor located in the program memory space starting at $0000 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:$0000 and lets the ROM be located at address p:$0000. 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:$0
            ds $100                    ; reserve space
            org p:$100
F_ _start
```

5.3.2 Memory Configuration and Management

The DSP56300 family supports three memory spaces: program memory (p memory), y data memory, and x data memory. 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.

Global and static data reside at the bottom of the available data memory. 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 dynamic data segments, heap and stack, are located at the top and bottom of memory, respectively. 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 segments. The program segment is configured using the org statements in the **crt0** file. The variable **_ _y_size** should be initialized with the desired initial stack pointer and the variable **_ _break** should be initialized with the desired 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 5-3 **Fast Stack:** In this example, it is desired that the stack reside in an 8k SRAM starting at x:$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     x:$4000
ds      $2000
endsec
```

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

```
move y:F_ __size,r6
```

to:

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

Example 5-4 **Fast Heap:** It is desired that the heap reside in an 8k SRAM starting at x:$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     x:$4000
ds      $2000
endsec
```

Change the following line in the **crt0** file:

```
TOP_OF_MEMORY     equ     $ff0000
```

to:

```
TOP_OF_MEMORY     equ     $4000
```

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. A way to restrict the linker from overlapping these memory spaces is needed. For example, suppose that both the x and pspaces 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 data structures in order to utilize this memory.

5.3.3 Interrupt Vectors

The interrupt vector locations for the DSP56300 family (a.k.a. “interrupt source” in the *DSP56300 Family Manual*) contain one or two instructions each to be executed when the interrupt assigned to that location occurs. There are 128 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 but one of the vectors are initialized with the instruction **jsr Fabort**. The first element of the vector table, which is the hardware reset vector, 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 due to an error. The **F_ _start** labels the program address of the bootstrap code that calls **main()**.

Interrupt vectors that are to be used must be reprogrammed to point to the interrupt service routines instead of the **abort()** function.

Thefollowing **crt0** code segment is the default interrupt vector table initialization.

```
section reset
org     p:$0
jmp     F_start
org     p:$2
dup     127
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 5-5 illustrates how to initialize pointers to these user-provided interrupt service routines.

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

```
section reset
org     p:$0
jmp     F_ _start
jsr     interrupt02
jsr     interrupt04
jsr     interrupt06
jsr     interrupt08
…
jsr     interrupt
endsec
```

Example 5-6

Interrupt Service Routine: This service routine updates the global variable **F_ _time** at each hypothetical timer interrupt.

```
section interrupt
org         p:
global      interrupt
interrupt
```

```
move        (r6)+              ; secure the stack pointer
                               ; (refer to section 5.4.1.4)
move        r1,:(r6)           ; save the r1 register
move        y:F_ _time,r1      ;retrieve the variable _ _time
move        (r1)+              ; increment the variable
move        r1,y:F_ _time      ; save the result
move        y:(r6) -,r1        ; restore the r1 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 the *DSP56300 Family Manual* for more information on fast and long interrupts).

5.3.4 Miscellaneous Code

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

1. The error code variable, **errno**, is a global integer used to record failure codes in C library calls. This error code variable is needed if standard ANSI C library calls are used. This variable can be utilized as a debugging aid in order to check which error code is returned fromC 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 DSP563XX simulator (**run563**) 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. Host I/O stub functions, _ _**send()** and _ _ **receive(),** are defined in **crt0** and are called by the standard I/O library functions. The provided **crt0** file only has stub functions, as **run563** watches these addresses and performs all I/O directly.

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.

5.4 Signal File

The hardware level interrupt mechanism (see the *DSP56300 Family Manual*) is more efficient than the **signal()** function. However, in many cases interrupts can be handled in the C environment and it is often preferable to do so. There are two functions, **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 5.3.3 (see the *DSP56300 Family 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.

5.4.1 Signal()

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

1. A **signal number** — On the DSP56300 processor, the signal number corresponds directly to the interrupt vector address. Notice that the signal numbers are even numbers.

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. The pointed-to function is assumed to follow the compiler's calling conventions with respect to registers saved, etc.

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 interrupt 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. Re-running **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., interrupt 08) with a **jsr** to it (e.g., **jsr interrupt 08**)or use a fast interrupt routine.

5.4.2 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.**

5.5 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 accumulator **a** and sets the **ccr** to reflect the return value just stored in accumulator **a**. The function **longjmp()** discards the return pro-

gram 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.

5.6 Host-Supported I/O (printf (), et al)

The library provided with DSP563CCC includes a full implementation of the ANSI C standard input and output routines. These routines already work with **run563**, and can easily be embedded in custom applications. Anywhere that formatted I/O is desired, these library routines can be included to simplify development. The entire suite of routines is based upon a simple communication protocol between the DSP and a host resident I/O driver, so porting the entire system to custom hardware is trivial.

5.6.1 DSP functions __send () and __receive ()

All standard I/O functions, no matter how complicated, are built upon two simple communication functions, **__send ()**, and **__receive ()**. **__send ()** sends a message to the I/O driver code residing on the host. **__receive ()** retrieves a message from that same driver. Implementing these two functions is all that need be done on the DSP in order to support standard I/O on custom hardware.

It is assumed that some sort of hardware communication channel exists between the host and the DSP. **__send ()** and **__receive ()** implement a simple message passing mechanism on top of such a channel. **__send ()** accepts two arguments: the address of the buffer to send, and the number of words to draw from that buffer. **__receive ()** accepts the address of a buffer in which to place the received message as its single argument. All of the interactions between the host and DSP are driven by the library code running on the DSP; because the DSP is in control, it knows the size of return messages from the host, rendering a count argument to the function **__receive ()** superfluous. **__send ()** and **__receive ()** are as simple as they seem; the complexity of the standard I/O package is embedded in the host-side driver and the **lib563[xyl].clb** library routines.

5.6.2 The Host-Side I/O Driver

The application running on the host must have the provided I/O driver embedded in it. The driver is written in C, and uses typically available library routines such as **open (), close ()**, **read ()**, and **write ()** to perform the actions requested by the DSP. The I/O code is written as an event-driven state machine, so that the host side application can perform concurrently with the DSP when the DSP is not requesting I/O activity. In fact, the I/O driver on the host may be interrupt driven.

The host-side driver consists of the code in **dsp/etc/hostio.h** and **dsp/etc/hostio.c**. The meat of the package consists of two functions, **init_host_io_structures()** and **process_pending_host_io ()**, and two buffers **hio_send** and **hio_receive**. The function **init_host_io_structures ()** is called to initialize host-side driver data structures before DSP execution is commenced. The buffer **hio_send** is used to send messages to the DSP, and **hio_receive** is used to receive messages from the DSP. The function **process_pending_host_io ()** considers the current state of the buffers and its own internal state, and then performs any required buffer modification or host I/O.

5.6.3 Communication between the Host and DSP

The messages passed between the DSP and the host's I/O driver are defined in the file **dsp/include/ioprim.h**. All sequences of communication are initiated by the DSP as a direct result of a call to a standard I/O function. Each standard I/O call may initiate a series of messages between the DSP and the host, with the host eventually returning a message containing the completion status of the original request. The file **dsp/include/ioprim.h** is included by both the host-side I/O driver, and the standard I/O library code; it defines the constant definitions used in the aforementioned messages. A typical series of events and messages that comprise a standard I/O call might look like this:

1) The application running on the DSP makes a call to **fopen ()**.

2) The library code in **lib563[xyl].clb** calls **__send ()**, with a buffer that contains the code **DSP_OPEN**, the flags, the mode, and the string length of the path.

3) The host receives the message into the buffer **hio_receive**, sets its valid flag, and calls **process_pending_host_io ()**. The state machine inside **process_pending_host_io ()** notes that it is now in the middle of an open file request, records the values from the first message, and then returns. At this point, code written by the application developer must check the valid flag of the buffer hio_send; in this case, the buffer **hio_send** has not been marked valid.

4) The library code in **lib563[xyl].clb** calls **__send ()** again, this time sending the path.

5) Again, the host receives the message into the buffer **hio_receive**, and calls **process_pending_host_io ()** after setting the buffer's valid flag.

6) **process_pending_host_io ()** uses the information from the two messages to perform the file open. It then builds an operation status message, places it in the buffer **hio_send**, and sets that buffer's valid flag. **process_pending_host_io ()** resets its internal state and returns.

7) The host checks the buffer valid flag on **hio_send**, sees that it is true, and transmits the message to the DSP.

8) The library code running on the DSP finishes the **fopen ()** call and returns.

On the host side of the interface, the application writer must write the code that exchanges data with the DSP, the code that calls **process_pending_host_io ()**, and the code that checks buffer valid flags. On the DSP side of the interface, the application writer must write the routines **__send ()** and **__receive ()**. The communication between the DSP and the host is always initiated by the DSP and always follows a predetermined pattern, depending on the initial message. Because this communication is so simple, the code that calls **process_pending_host_io ()** can also be quite simple.

Example 5-7 is a *hypothetical* non-reentrant interrupt handler written in C. It uses two functions, **peek ()** and **poke ()**, to access some sort of hardware communication device connected to the DSP. The functions **peek ()** and **poke ()** aren't provided; they're simply an abstraction for host-side hardware access. This code assumes that the DSP sends the size of a message directly before sending a message. **CHECK_BUFFER_SIZE** is a macro defined in **dsp/etc/hostio.h**. It should always be used to ensure that the buffer **hio_receive** is large enough to handle the incoming message. Finally, this example assumes that the function **signal ()** is available to register interrupt handlers.

This code doesn't have to be implemented in an interrupt driven manner; periodic polling could be used as well. The critical issues are that the communication must be reliable, and that the system must not deadlock; the latter is easy to ensure, given the simple nature of the communication protocol.

Example 5-7 Sample Host-Side Glue Code

```
void interrupt_driven_io ( )
{
        int i;

        /* get the size of the message from the DSP. */
        int size = peek ( IN_PORT );

        /* make sure that hio_receive is large enough. */
        CHECK_BUFFER_SIZE ( & hio_receive, size );

        /* read the message via hardware, into the buffer. */
        for ( i = 0; i < size; ++ i )
        {
```

```
        hio_receive.buffer[i] = peek ( IN_PORT );
    }

    /* mark the buffer as valid, perform any requested I/O. */
    hio_receive.valid_p = TRUE;
    process_pending_host_io ( );

    /* if the driver wants to send to the DSP, then do so now. */
    if ( hio_send.valid_p )
    {
        for ( i = 0; i < hio_send.length; ++ i )
        {
            poke ( OUT_PORT, hio_send.buffer[i] );
        }
        hio_send.valid_p = FALSE;
    }

    /* re-register this handler for future DSP message interrupts. */
    signal ( SIG_IN_PORT, interrupt_driven_io );
}
```

Appendix A
Library Support

A.1 Standard ANSI Header Files

Each function provided in the ANSI C library **lib563c[xyl].clb** is declared with the appropriate prototype in a header file, whose contents may be included by using the **#include** preprocessing directive with angular brackets. 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. ANSI C Library Functions

The library contains all of the ANSI defined subroutines. The C and assembly language source for each library routine is distributed free of charge with the compiler.

A.1.1 Hosted vs. Non-Hosted Library Routines

Some of the standard ANSI defined routines perform I/O. Programs that use these functions will not encounter problems when run using **run563**. Extra work may be needed to port hosted I/O routines to custom hardware configurations. Non-hosted routines will not encounter any problems on custom hardware.

For a description of **run563**, see Appendix B.

A.2 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 subroutine by replacing the occurrence of the subroutine call with the body of the subroutine itself. This provides execution time benefits:

1. Eliminates subroutine call overhead. This is a substantial portion of the run-time for some 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.

```
C:\> g563c -Uceil file.c
```

A.3 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 following list 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 upon 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.
clearerr — Clear error indicators associated with a stream.
cos — Cosine.
cosh — Hyperbolic cosine.
div — integer division with remainder.
exit — Terminate program normally.
exp — Exponential, e^x.

fabs — Absolute value of a double.
fclose — Close a stream.
feof — Test the end-of-file indicator of a stream.
ferror — Test the error indicator of a stream.
fflush — Flush all output pending on a stream.
fgetc — Read a character from a stream.
fgetpos — Retrieve the current value of the file position indicator of a stream.
fgets — Read a string from a stream.
floor — Floor function.
fmod — Floating point remainder.
fopen — Open a named file on the disk, to be accessed via a stream.
fprintf — Write formatted output to a stream.
fputc — Write a character to a stream.
fputs — Write a string to a stream.
fread — Read unformatted input from a stream.
free — Free storage allocated by calloc, malloc, and realloc.
freopen — Open a named file on the disk, to be accessed via a stream.
frexp — Break a floating point number into mantissa and exponent.
fscanf — Read formatted input from a stream.
fseek — Set a stream's file position indicator.
fsetpos — Set a stream's file position indicator.
ftell — Retrieve the current value of a stream's file position indicator.
fwrite — Write unformatted output to a stream.
getc — Read a character from a stream (this may be a macro).
getchar — Read a character from the stream stdin (this may be a macro).
gets — Read a string from the stream stdin.
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 a power of two.
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 from one area to another (source and destination may overlap).
memset — Initialize memory area.
modf — Break a double into it's integral and fractional parts.
perror — Print error message indicated by errno.
pow — Raise a double to a power.
printf — Write formatted output to the stream stdout.
putc — Write a character to a stream (this may be a macro).
putchar — Write a character to the stream stdout (this may be a macro).
puts — Write a string to the stream stdout.
qsort — Quick sort.
raise — Raise a signal.
rand — Pseudo- random number generator.
realloc — Change size of dynamically allocated storage area.
remove — Remove a file from the disk.
rename — Rename a file on the disk.
rewind — Reset the file position indicator of a stream to the beginning of the file on the disk.
scanf — Read formatted input from the stream stdin.
setjmp — Save a reference of the current calling environment for later use by longjmp.
setbuf — Associate a buffer with a stream.
setvbuf — Associate a buffer with a stream, while also specifying the buffering mode and buffer size.
signal — Set up signal handler.
sin — Sine.
sinh — Hyperbolic Sine.
sprintf — Write formatted output to a string.
sqrt — Square root.
srand — Seed the pseudo-random number generator.
sscanf — Read formatted input from a string.
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 a string not contained in a second string.
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 a second string.
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.
tmpfile — Create a temporary binary file on the disk to be referenced via a stream.
tmpnam — Generate a unique, valid file name.
tolower — Convert uppercase character to lowercase.
toupper — Convert lowercase character to uppercase.
ungetc — Push a character back onto a specified input stream.
vfprintf — Write formatted output to a stream, using a va_list.
vprintf — Write formatted output to the stream stdout, using a va_list.
vsprintf — Write formatted output to a string, using a va_list.
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 (**run563**). 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 principal 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:

```
2.498090 2.214290 1.982310 1.772150 1.570790 1.369430
1.159270
0.927295
0.643501
0.000345
```

NAME

asin — Arc sine.

SYNOPSIS

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

DESCRIPTION

The **asin** function computes the principal value of the arc sine of **x** in the range [-π/2, +π/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 -0.411516 -0.201357 -0.000000 0.201357
0.411516
0.643501
0.927295
1.570450
```

NAME

atan — Arc tangent.

SYNOPSIS

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

DESCRIPTION

The **atan** function computes the principal value of the arc tangent of **x** in the range [-π/2, +π/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:

```
-0.674740 -0.540419 -0.380506 -0.197395 -0.000000 0.197395
0.380506
0.540419
0.674740
0.785398
```

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 principal 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.

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

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 ) == 0.785398
atan2( -7.09, 7.09 ) == -0.785398
atan2( 7.09, -7.09 ) == 2.356190
```

atan2(-7.09, -7.09) == -2.356190

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 reverse order of their registration. 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 func_1 ( void )
{
        printf ( "first function called\n" );
}

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

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

prints to standard output:

```
-- testing atexit --
second function called
first 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 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" ) == 7.089990
```

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 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 of **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] =
{
        "bald", "driving", "feet", "flintstone", "fred", "on"
};

int compare ( const void *key, const void *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( "NO, sam the butcher 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

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 <math.h>

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

prints to standard output:

```
ceil( 7.09 ) == 8.000000
```

NAME

clearerr — Clear any error indicators associated with a specified stream.

SYNOPSIS

```
#include <stdio.h>
void clearerr ( FILE *stream );
```

DESCRIPTION

The **clearerr** function clears the end-of-file and error indicators for the specified stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        FILE *stream = tmpfile (); /* initially empty. */
        clearerr ( stream );
        printf ( “end-of-file indicator is: %d\n”, feof ( stream ));
}
```

prints to standard output:

```
end-of-file indicator is: 0
```

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 ) == 11.590800
```

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 = 56;

        result = div( numer, denom );
        printf( "quotient == %d\t", result.quot );
        printf( "remainder == %d\n", result.rem);
}
```

prints to standard output:

```
quotient == 12 remainder == 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 where registered. Status is returned to the environment (**run563**). 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 program 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** returning **HUGE_VAL**. If the value of **x** is too small, a range error will also occur with errno being set to **ERANGE** and exp returning 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 ) == 1199.900000
```

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 **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 = fabs( neg );
        printf( "-- absolute value of %f == %f --\n", neg, pos );
}
```

prints to standard output:

```
-- absolute value of -7.090000 == 7.090000 --
```

NAME

fclose — Close a stream.

SYNOPSIS

```
#include <stdio.h>
int fclose ( FILE * );
```

DESCRIPTION

The function **fclose** flushes all output on the specified stream, and disassociates the stream from the file on the host. The function **fclose** returns **EOF** if there are any problems, otherwise 0.

SEE ALSO

fprintf — Used to write formatted output to a stream.

EXAMPLE

```
#include <stdio.h>

void main()
{
        fprintf ( stdout, “see me second” );
        fprintf ( stderr, “see me first\n” );
        fclose ( stdout );
}
```

prints to combined standard error and standard output:

```
see me first
see me second
```

Note that **stdout** is by default line buffered, while **stderr** is not. The call to **fclose** causes the pending output on **stdout** to be flushed.

NAME

feof — Test the end-of-file indicator of a specified stream.

SYNOPSIS

```
#include <stdio.h>
int feof ( FILE * );
```

DESCRIPTION

The function **feof** function tests the end-of-file indicator associated with the specified stream. It returns non-zero if and only if the end-of-file indicator is set.

SEE ALSO

fopen — Used to associate a stream with a file on the host's disk.

fseek — Used to alter the file position indicator associated with a stream.

fprintf — Used to write formatted output to a stream.

EXAMPLE

```
#include <stdio.h>

void main()
{
        FILE *somefile = fopen ( "somefile", "rb+" );
        (void) fseek ( somefile, 0L, SEEK_END );
        (void) fgetc ( somefile );
        fprintf ( stdout, "are we at the file's end? %s\n",
                        feof ( somefile ) ? "yes" :  "no" );
}
```

prints to standard output:

```
are we at the file's end? yes
```

NAME

ferror — Test the error indicator of a stream.

SYNOPSIS

```
#include <stdio.h>
int ferror ( FILE * );
```

DESCRIPTION

The function **ferror** function tests the error indicator associated with the specified stream. It returns non-zero if and only if the error indicator is set. **ferror** should be used following one or more stream I/O function calls.

NAME

fflush — Flush all pending output associated with a stream.

SYNOPSIS

```
#include <stdio.h>
void fflush ( FILE* );
```

DESCRIPTION

The function **fflush** causes any pending output associated with the specified stream to be written to the output device.

SEE ALSO

fprintf — Used to write formatted output to a stream.

EXAMPLE

```
#include <stdio.h>

void main()
{
        fprintf ( stdout, “see me second” );
        fprintf ( stderr, “see me first\n” );
        fflush ( stdout );
}
```

prints to combined standard error and standard output:

```
see me first
see me second
```

Note that **stdout** is by default line buffered, while **stderr** is not. The call to **fflush** causes the pending output on **stdout** to be flushed.

NAME

fgetc — Read a character from the specified stream.

SYNOPSIS

```
#include <stdio.h>
int fgetc ( FILE *stream );
```

DESCRIPTION

The function **fgetc** will retrieve the next input character from the specified stream. If the stream is associated with a file on the disk, then the file position indicator is advanced. On error, **fgetc** returns **EOF**.

SEE ALSO

fputc — Write a character to a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        char value = (char) fgetc ( stdin );

        while ( EOF != value )
        {
                fputc ( value, stdout );
                value = (char) fgetc ( stdin );
        }
}
```

will echo all characters from standard input to standard output until the input is exhausted.

NAME

fgetpos — Get the value of the file position indicator associated with a stream.

SYNOPSIS

```
#include <stdio.h>
int fgetpos ( FILE *stream, fpos_t *pos );
```

DESCRIPTION

The function **fgetpos** fetches the value of the file position indicator associated with **stream** and stores it in the object pointed to by **pos**. **fgetpos** returns zero if it was successful. The value of the file position indicator is meaningless except as an argument to the function fsetpos.

SEE ALSO

fopen — Open a file on the host's disk and associate it with a stream.

fseek — Used to alter the file position indicator associated with a stream.

fsetpos — Set the value of the file position indicator associated with a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        FILE *preexisting = fopen ( “already.here”, “r” );
        fpos_t pos;

        (void) fgetpos ( preexisting, & pos );
        (void) fseek ( preexisting, 0L, SEEK_END );
        (void) fsetpos ( preexisting, & pos );
}
```

will open a hypothetical pre-existing file on the disk, record the initial position in pos, seek to the end of the file, and finally restore the initial value of the file position indicator.

NAME

fgets — Read a string from the specified stream.

SYNOPSIS

```
#include <stdio.h>
char *fgets ( char *s, int n, FILE *stream );
```

DESCRIPTION

The function **fgets** will read at most **n** -1 characters from the specified stream. **fgets** will not read past a newline character. The characters are stored in memory starting at the location pointed to by **s**. **fgets** returns **s** if it was successful, **NULL** otherwise. **fgets** will update the file position indicator for any characters read.

SEE ALSO

fputs — Write a string to a stream.

rewind — Reset the file position indicator associated with a stream to the beginning of the file.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        FILE *disk_file = fopen ( “newfile”, “a+” );
        char one_line[64];

        fputs ( “read this line\n” “but not this line.\n”, disk_file );
        rewind ( disk_file );
        fgets ( one_line, 64, disk_file );
        fputs ( one_line, stdout );
}
```

will open a new file on the disk named “newfile”. Two lines will be written to the new file, and the first line will be read back using fgets. The retrieved line is then printed to standard output as follows:

```
read this line
```

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 ) == -1.000000
```

NAME

fopen — Open a named file on the host's disk.

SYNOPSIS

```
#include <stdio.h>
FILE* fopen ( const char *filename, const char *mode );
```

DESCRIPTION

The **fopen** function attempts to open a named file for access via a stream. If **fopen** is able to open the specified file, then the new stream associated with that file is returned. If fopen fails, then it returns **NULL**. The **mode** argument indicates the type of the stream, and how the stream may be accessed:

"r"	—	text type, read only,
"w"	—	text type, write only,
"a"	—	text type, append only (write after end of current file),
"rb"	—	binary type, read only,
"wb"	—	binary type, write only,
"ab"	—	binary type, append only (write after end of current file),
"r+"	—	text type, read and write,
"w+"	—	text type, read and write (any pre-existing file is destroyed),
"a+"	—	text type, append only (read/write after end of current file),
"rb+"	—	binary type, read and write,
"wb+"	—	binary type, read and write (any pre-existing file is destroyed),
"ab+"	—	binary type, append only (read/write after end of current file).

Note that opening a file that does not exist will fail if **r** is the first character in the **mode** string. When opened, the stream is initially line buffered.

SEE ALSO

fputs	—	Write a string to a stream.
fgets	—	Read a string from a stream.
fprintf	—	Used to write formatted output to a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        FILE *stream = fopen ( “file.new”, “w” );
        char data[64];

        fprintf ( stream, “verify this\n” );
        fclose ( stream );

        stream = fopen ( “file.new”, “r” );
        fgets ( data, 64, stream );
        fputs ( data, stdout );
}
```

This example opens an new text file, writes to the associated stream, and closes that stream. It then reopens the file and displays the first line of that file on standard output:

```
verify this
```

NAME

fprintf — Write formatted output to a stream.

SYNOPSIS

```
#include <stdio.h>
int fprintf ( FILE *stream, const char *format, ... );
```

DESCRIPTION

The function **fprintf** functions exactly like the function **printf**, except that the output is directed to the specified stream rather than being automatically directed to standard output. Please use the description of argument values in the description of the **printf** function.

SEE ALSO

printf — Used to write formatted output to a standard output.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        fprintf ( stdout, “hello world\n” );
}
```

Will cause the following output to be printed to standard output:

```
hello world
```

NAME

fputc — Write a single character to a stream.

SYNOPSIS

```
#include <stdio.h>
int fputc ( int c, FILE *stream );
```

DESCRIPTION

The function **fputc** writes the character **c** to the specified stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        fputc ( (int) 'S', stdout );
        fputc ( (int) 'h', stdout );
        fputc ( (int) 'a', stdout );
        fputc ( (int) 'd', stdout );
        fputc ( (int) 'r', stdout );
        fputc ( (int) 'a', stdout );
        fputc ( (int) 'c', stdout );
        fputc ( (int) 'k', stdout );
        fputc ( (int) '\n', stdout );
}
```

Will cause the following output to be printed to standard output:

```
Shadrack
```

NAME

fputs — Write a string to a stream.

SYNOPSIS

```
#include <stdio.h>
int fputs ( const char *s, FILE *stream );
```

DESCRIPTION

The function **fputc** writes the string **s** to the specified stream. The trailing '\0' in s is not written to the stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        fputs ( "hand me down pumas\n", stdout );
}
```

Will cause the following output to be printed to standard output:

```
hand me down pumas
```

NAME

fread — Read data directly from a stream.

SYNOPSIS

```
#include <stdio.h>
size_t fread ( void *ptr, size_t size, size_t nmemb, FILE *stream );
```

DESCRIPTION

The function fread reads raw data from the specified stream. The data is stored in memory starting with the location pointed to by **ptr**. The quantity of data is **size** * **nmemb**. **fread** returns the number of elements successfully read.

SEE ALSO

printf — Used to write formatted output to a standard output.

fopen — Open a file and associate it with a stream.

EXAMPLE

Assume that the disk file “professor” has as its contents the following string, including the trailing ‘\0’:

“What’s another word for pirate treasure?”

The following C program uses fread:

```
#include <stdio.h>

void main ()
{
        FILE *booty = fopen ( “professor”, “r” );
        char buffer[64];

        fread ( buffer, 63, sizeof ( char ), booty );
        fprintf ( stdout, buffer );
}
```

and will cause the following output to be printed to standard output:

What’s another word for pirate treasure?”

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

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

freopen — Open a named file on the disk, to be accessed via the specified stream.

SYNOPSIS

```
#include <stdio.h>
FILE *freopen ( const char *filename, const char *mode, FILE *stream );
```

DESCRIPTION

The function **freopen** opens the named disk file in the same manner as **fopen**. However, **freopen** associates that file with the specified stream. If successful, **freopen** returns its argument **stream**, and if unsuccessful, it returns **NULL**. The range of acceptable values for the **mode** argument are the same as those for **fopen**.

SEE ALSO

printf — Used to write formatted output to standard output.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        freopen ( “diskfile”, “w”, stdout );
        printf ( “hello world\n” );
}
```

This example redirects standard output to a file on the disk via **freopen**. The “hello world” output will not appear on the normal standard output device, but rather in the file “diskfile”.

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 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 $2^{*\mathbf{exp}}$. 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 <math.h>

void main()
{
        int exp;
        double mant;

        mant = frexp( 70.9, &exp );
        printf( "mantissa == %f\texponent == %d\n", mant, exp );
}
```

prints to standard output:

```
mantissa == 0.553906 exponent == 7
```

NAME

fscanf — Read formatted input from a stream.

SYNOPSIS

```
#include <stdio.h>
int fscanf ( FILE *stream, const char *format, ... );
```

DESCRIPTION

The function **fscanf** reads input from the specified stream. It uses the string **format** as a guide for interpreting the input and storing values in memory. Subsequent arguments to **fscanf** are used as pointers to objects in memory that will receive the input values read from the stream.

The format string is composed of directives. These are parsed from left to right from the format string, and indicate how the input from the specified stream should be processed.

If **fscanf** fails to apply a directive, then it returns. Directives are composed of either white-space characters or normal characters. White space character sequences indicate that **fscanf** should read input from the specified stream up to the first non-white-space character. Directives that consist of non-white-space characters are processed in the following manner: input is read from the specified stream until the input character is not a member of the set of characters comprising the directive. Finally, directives can be conversion specifications; these directives begin with the '%' character. They describe how **fscanf** should parse input, and how **fscanf** should synthesize a value to be stored in memory.

Conversion specifications are processed as follows. First, the stream is read until all white-space characters have been exhausted, unless '[', 'c', or 'n' is part of the conversion specification. Second, a value is derived from the input stream according to the conversion specifier. A conversion specifier may be one of the following:

'd' — match a signed, decimal integer.

'i' — match a signed integer, whose base is determined in the same manner as a C integer constant.

'o' — match an octal integer.

'u' — match an unsigned, decimal integer.

'x' — match a signed, hexadecimal integer. (**'X'** is also valid).

'e','f','g' — match a floating-point number. (**'E'**,**'F'** and **'G'** are also valid).

's' — match a sequence of non-white-space characters, essentially scan a token string.

'[' — match a non-empty sequence of characters from a set of expected characters, which are bounded by a following **']'**.

'c' — match a sequence of characters, as specified by the field width. As a default, scan only one character.

'n' — don't match anything, just store the number of characters read from the input stream during this call to **fscanf**.

'%' — match a '**%**' character.

fscanf returns **EOF** if an input failure is detected before any conversions take place. Otherwise it returns the number of assignments made. Note that an optional assignment suppression character '*' may follow the initial '%'. This character will cause **fscanf** to discard the converted value without advancing along the list of object pointers.

SEE ALSO

scanf — Read formatted input from standard input.

sscanf — Read formatted input from a string.

EXAMPLES

(a) The following program will, assuming that the input pending on standard input is "my 98", store the three characters 'm', 'y', '\0' will be stored in word[], and 98 will be stored in number.

```
#include <stdio.h>

void main ()
{
        char word[8];
        int number;
        fscanf ( stdin, " %s %d", word, & number );
}
```

(b) The following program will, assuming that the input pending on standard input is "yall come", store the following five characters in the array word: 'y', 'a', 'l', 'l', '\0'.

```
#include <stdio.h>

void main ()
{
        char word[8];
        fscanf ( stdin, "%[lay]", word );
}
```

NAME

fseek — Set the file position indicator associated with a stream.

SYNOPSIS

```
#include <stdio.h>
int fseek ( FILE *stream, long int offset, int whence );
```

DESCRIPTION

The function **fseek** will set the file position indicator associated with the specified stream, according to the values of offset plus an initial value indicated by **whence**. The return value is -1 for an improper seek, 0 otherwise. Initial values derived of **whence** values are as follows:

SEEK_SET — The initial value is the beginning of the file.
SEEK_CUR — The initial value is the current position in the file.
SEEK_END — The initial value is the end of the file.

The value of **offset** must either be zero or the value returned by a call to **ftell**.

SEE ALSO

fopen — Open a disk file and associate it with a stream.
fgetc — Read a single character from a stream.
fclose — Close a stream.

EXAMPLES

The following function will read the last character in a text file specified by the parameter **name**.

```
#include <stdio.h>

char last_in_file ( char *name )
{
        FILE *tmp = fopen ( name, “r” );
        char return_value;

        (void) fseek ( tmp, -1L, SEEK_END );
        return_value = (char) fgetc ( tmp );
        fclose ( tmp );
        return return_value;
}
```

NAME

fsetpos — Set the file position indicator associated with a stream.

SYNOPSIS

```
#include <stdio.h>
int fsetpos ( FILE *stream, const fpos_t *pos );
```

DESCRIPTION

The function **fsetpos** will change the file position indicator associated with the specified stream. The value of **pos** must be the return value from a prior call to **fgetpos**. Note that a successful call to **fsetpos** will undo any effect of an immediately preceding **ungetc** on the same stream. If it is successful, **fsetpos** returns zero.

SEE ALSO

fgetpos — Obtain the file position indicator value associated with a stream.

EXAMPLES

```
#include <stdio.h>

void main ()
{
        FILE *preexisting = fopen ( “already.here”, “r” );
        fpos_t pos;

        (void) fgetpos ( preexisting, & pos );
        (void) fseek ( preexisting, 0L, SEEK_END );
        (void) fsetpos ( preexisting, & pos );
}
```

will open a hypothetical pre-existing file on the disk, record the initial position in pos, seek to the end of the file, and finally restore the initial value of the file position indicator.

NAME

ftell — Get the file position indicator associated with a stream.

SYNOPSIS

```
#include <stdio.h>
long int ftell ( FILE *stream );
```

DESCRIPTION

The function **ftell** will return the value of the file position indicator for the specified stream. The return value is usable only by the function fseek. **ftell** returns -1L.

SEE ALSO

fopen — Open a file and associate it with a stream.
fread — Read data from a stream.
putchar — Send character data to standard output.
fseek — Change the file position indicator associated with a stream.

EXAMPLE

```
#include <stdio.h>

main ()
{
        FILE *stream = fopen ( “file.abc”, “r” );
        long int beginning = ftell ( stream );
        char send;

        (void) fread ( & send, sizeof ( char ), 1, stream );
        putchar ( send );
        fseek ( stream, beginning, SEEK_SET );
        (void) fread ( & send, sizeof ( char ), 1, stream );
        putchar ( send );
}
```

will read and print the first character in the file “file.abc” twice. **ftell** is used to reset the file position indicator to the beginning of the file.

NAME

fwrite — Write data directly to a stream.

SYNOPSIS

```
#include <stdio.h>
size_t fwrite ( const void *ptr, size_t size, size_t nmemb, FILE *stream );
```

DESCRIPTION

The function **fwrite** writes raw data to the specified stream. The data is drawn from memory starting with the location pointed to by **ptr**. The quantity of data is **size** * **nmemb**. **fwrite** returns the number of elements successfully written.

EXAMPLE

```
#include <stdio.h>

main ()
{
        char message[] = “hand me down pumas”;

        fwrite ( message, sizeof ( char ), strlen ( message ), stdout );
}
```

will write the message “hand me down pumas” onto standard output.

NAME

getc — Read a character from the specified stream.

SYNOPSIS

```
#include <stdout.h>
int fgetc ( FILE *stream );
```

DESCRIPTION

getc is equivalent to **fgetc**, except that **getc** may be implemented as a macro. If such is the case, the argument **stream** may be evaluated more than once. This only becomes a problem if evaluation of the argument has side effects.

SEE ALSO

fgetc — Read a character from a stream.

fputc — Write a character to a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        char value = (char) getc ( stdin );

        while ( EOF != value )
        {
                fputc ( value, stdout );
                value = (char) getc ( stdin );
        }
}
```

will echo all characters from standard input to standard output until the input is exhausted.

NAME

getchar — Read a character from standard input.

SYNOPSIS

```
#include <stdio.h>
int getchar ( void );
```

DESCRIPTION

The function **getchar** reads the next character from standard input. If there is no pending input, **getchar** returns **EOF**, otherwise the read character is cast to type **int** and returned.

SEE ALSO

fputc — Write a character to a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        char value = (char) getchar ();

        while ( EOF != value )
        {
                fputc ( value, stdout );
                value = (char) getchar ( );
        }
}
```

will echo all characters from standard input to standard output until the input is exhausted.

NAME

gets — Read a string from standard input.

SYNOPSIS

```
#include <stdio.h>
char *gets ( char *s );
```

DESCRIPTION

The function **fgets** will read characters from standard input, and place them sequentially in memory, starting with the location pointed to by **s**. **gets** will not read past a newline character, or past the end of the file. The characters are stored in memory starting at the location pointed to by **s**. **gets** returns **s** if it was successful, **NULL** otherwise. **gets** will update the file position indicator for any characters read. If **gets** encounters the end of file on its first read of standard input, **NULL** is returned. **gets** does not append a '\0' to the array of characters read.

SEE ALSO

fgets — Read a string from a stream.

puts — Read a string from a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        char line_buffer[BUFSIZ];

        puts ( gets ( line_buffer ));
}
```

will echo a newline-terminated line of input from standard input onto standard output. Note that **gets** doesn't read past the newline; **puts** supplies one by definition.

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 all other cases. 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 all other cases. 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

iscntrl — Test for control character.

SYNOPSIS

```
#include <ctype.h>
int iscntrl( int c );
```

DESCRIPTION

The **iscntrl** function returns a nonzero value for any control character; zero is returned in all other cases. 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 ( ( iscntrl( 0x07 ) ) )
        {
                printf( "\"beep\" (0x07) -- control character\n" );
        }

        if ( ! ( iscntrl( '@' ) ) )
        {
                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 all other cases. 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 ( ( isgraph ( '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 all other cases. 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 all other cases. 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 all other cases. 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( "comma -- \"punct\" character\n" );
        }

        if ( ! ( ispunct( '\t' ) ) )
        {
                printf( "tab -- not \"punct\" character\n" );
        }
}
```

prints to standard output:

```
comma -- "punct" character
tab -- not "punct" 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 all other cases. 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 all other cases. 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 all other cases. 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 ( ( isxdigit ( 'F' ) ) )
        {
                printf( "F -- hexadecimal character\n" );
        }

        if ( ! ( isxdigit ( '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 a power of two.

SYNOPSIS

```
#include <math.h>
double ldexp( double x, int exp );
```

DESCRIPTION

The **ldexp** function returns **x** * $2^{\mathbf{exp}}$. 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

Please see the include file **stdlib.h** for the definition of **ldiv_t**.

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

void main()
{
                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 == 12 remainder == 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 ) == 1.958680
```

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 returned 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.

All global and **volatile** variables have defined values as of the point in time that **longjmp** was called; all **register** and non-volatile automatic variables will have undefined 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" );
                exit( 1 );
        }
        printf( "-- setjmp called --\n" );
        func();
}
```

prints to standard output:

```
-- setjmp called --
-- longjmp has been called --
```

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 block of storage space that is **size** words in size. If **size** exceeds the amount of memory available, **malloc** returns **NULL**.

SEE ALSO

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 DSP56300 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 use 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 ) == 0
mblen( str1, max ) == 2
mblen( "abcdef", 5 ) == 2
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 DSP56300 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 use 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 = "abcdefgh";
        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", 8 );
        ptr = warray;
        while ( *ptr != 0 )
        {
                printf( "%0.6x \n", *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:

006162
006364
006564
006768
000075
006e70
006163
006b65
006420
006172
000001
0015eb

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 DSP56300 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 use 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
2 chars packed. wide == 006162
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.

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**. The 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.

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.

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 breaks **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 == 0.090000 integral == 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>
#include <errno.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 $\mathbf{x}^{\mathbf{y}}$. If **x** is zero and **y** is less than zero, a domain error occurs setting **errno** to **EDOM** and returning 0.0. If $|\mathbf{x}^{\mathbf{y}}|$ 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.ddd***e**±*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 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 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\toctal == %o\n", i );
        printf("\t\toctal == %#.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\thex     == %x\n", i );
        printf("\t\thex     == %#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

putc — Write a single character to a stream.

SYNOPSIS

```
#include <stdio.h>
int putc ( int c, FILE *stream );
```

DESCRIPTION

The function **putc** writes the character **c** to the specified stream. It is identical to the function **fputc**, except that **putc** may be implemented as a macro. This means that arguments to **putc** may be evaluated more than once. This is only a problem for function arguments that have side effects when evaluated.

SEE ALSO

fputc — Write a single character to a stream.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        putc ( (int) 'S', stdout );
        putc ( (int) 'h', stdout );
        putc ( (int) 'a', stdout );
        putc ( (int) 'd', stdout );
        putc ( (int) 'r', stdout );
        putc ( (int) 'a', stdout );
        putc ( (int) 'c', stdout );
        putc ( (int) 'k', stdout );
        putc ( (int) '\n', stdout );
}
```

Will cause the following output to be printed to standard output:

```
Shadrack
```

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

gets — Get a line of text from standard input.

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

gets — Get a line of text from standard input.

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>
#include <signal.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\n", ( rand() ) % 709 );
        }
}
```

prints to standard output:

```
569
303
194
224
58
30
...
```

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 a new size, **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, **NULL** is returned and the original storage area is unchanged.

SEE ALSO

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 )
        {
                perror( "malloc failed" );
                exit (-8);
        }

        strcpy( str, "short string" );
        printf( "%s\n", str );

        /* 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

remove — Remove a file from the disk.

SYNOPSIS

```
#include <stdio.h>
int remove ( char *filename );
```

DESCRIPTION

The function **remove** will eliminate the file associated with the specified **filename**. The effect of this call on open files may vary from host to host, and is considered undefined.

EXAMPLE

```
#include <stdio.h>

void main()
{
        remove ( “foo.exe” );
}
```

will remove the file “foo.exe” on the disk, if such a file exists.

NAME

rename — Rename a file on the disk.

SYNOPSIS

```
#include <stdio.h>
int rename ( const char *old, const char *new );
```

DESCRIPTION

The function **rename** disassociates the a disk file from the name **old**, and associates it with the name **new**. The behavior of this call is undefined if there already exists a file associated with the name **new**. **rename** returns zero if it is successful. If it fails, the file remains associated with the old name, and is not altered in any way.

EXAMPLE

```
#include <stdio.h>

void main()
{
        rename ( “old.exe”, “new.exe” );
}
```

will rename the file “old.exe” to “new.exe”, provided that “old.exe” actually exists on the disk. Note that “old.exe” will cease to exist.

NAME

rewind — Reset the file position indicator to the beginning of the file.

SYNOPSIS

```
#include <stdio.h>
void rewind ( FILE *stream );
```

DESCRIPTION

The function **rewind** will reset the file position indicator associated with the specified stream. Any pending error is also cleared.

SEE ALSO

fgetpos — Obtain the file position indicator value associated with a stream.

fsetpos — Set the file position indicator value associated with a stream.

EXAMPLES

```
#include <stdio.h>

void main ()
{
        FILE *preexisting = fopen ( “already.here”, “r” );

        putchar ( fgetc ( preexisting ));
        rewind ( preexisting );
        putchar ( fgetc ( preexisting ));
}
```

will print the first character in the file “already.here” onto standard output twice.

NAME

scanf — Read formatted input from standard input.

SYNOPSIS

```
#include <stdio.h>
int scanf (char *format, ... );
```

DESCRIPTION

The function **scanf** is equivalent to the **fscanf** function, except that input is always read from standard input. Please use the description of argument values in the description of the **fscanf** function.

SEE ALSO

fscanf — Read formatted input from a stream.

EXAMPLES

See the manual entry for **fscanf** for examples. The only difference between **scanf** and **fscanf** is that **scanf** does not require a **FILE*** argument; **stdin** is implied.

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 has been called --
```

NAME

setbuf — Alter stream buffering.

SYNOPSIS

```
#include <stdio.h>
void setbuf ( FILE *stream,  char *buf );
```

DESCRIPTION

If **buf** is **NULL**, the specified stream will be unbuffered. If buf is non-**NULL**, then the stream will be fully buffered with a buffer of size **BUFSIZ**. Note that **setbuf** must be used only before any other operations are performed on the specified stream, and that the stream argument must be associated with an opened file. Calling **setbuf** is equivalent to calling **setvbuf** using **_IOBUF** for the **mode** argument, and **BUFSIZ** for the **size** argument.

SEE ALSO

setvbuf — Read formatted input from a stream.

NAME

setvbuf — Alter stream buffering.

SYNOPSIS

```
#include <stdio.h>
int setvbuf ( FILE *stream,  char *buf, int mode, size_t size );
```

DESCRIPTION

The function **setvbuf** is used to alter the way a specified stream is buffered. It must only be used before any other operation is performed on the specified stream. The argument **mode** determines the buffering policy:

_IOFBF — Use the full size of the buffer in the most efficient way.

_IOLBF — Use a line buffering policy: flush on newlines.

_IONBF — Do not buffer the stream at all.

The argument **size** specified the buffer size for this stream. The pointer **buf**, if non-**NULL**, may be used for stream buffering. If **buf** is **NULL**, then **setvbuf** will allocate any needed buffer.

SEE ALSO

setbuf — A restricted form of **setvbuf**.

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 the 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>
#include <signal.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 ) == 0.850903
```

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 ) == 11.547600
```

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[256];
        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 digits 2c5
        floating point 7.090000
```

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 d = 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\n", ( rand() ) % 709 );
        }
}
```

prints to standard output:

```
569
303
194
224
58
30
...
```

NAME

sscanf — Read formatted input from a string.

SYNOPSIS

```
#include <stdio.h>
int sscanf ( const char *s, const char *format, ... );
```

DESCRIPTION

The function **sscanf** reads formatted input from the string argument **s**, according to the format string **format**. The operation of **sscanf** is identical to fscanf except that input is read from a string.

SEE ALSO

fscanf — Read formatted input from a string.

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 strings terminating character. The **strcat** function returns the pointer **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", "xxxyz") < 0 **or** strcmp("xxxyz", "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", "xxxyz" ) < 0 )
        {
                puts( "xxx is less than xxxyz" );
        }
        else
        {
                puts( "xxx is greater than xxxyz" );
        }

        if ( strcmp( "xxxyz", "xxx" ) < 0 )
        {
                puts( "xxxyz is less than xxx" );
        }
        else
        {
                puts( "xxxyz is greater than xxx" );
        }

        if ( strcmp( "xxxyz", "xxxyz" ) == 0 )
        {
                puts( "xxxyz is equal to xxxyz" );
        }
}
```

prints to standard output:

```
xxx is less than xxxyz
xxxyz is greater than xxx
xxxyz 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 **DSP563CCC**, **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];

        strcpy( string, "-- no bald feet for george jetson --" );
        puts( string );
}
```

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 not in 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;

        for ( i = 1; i < 5; ++ i )
        {
                printf ( “message %d:%s\n”, i, strerror( i ));
        }
}
```

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 *preceeding* 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 — **C**oncatenate 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( bstr );
}
```

prints to standard output:

```
paste 5 chars of (string 2) on to ( string 1)
string 1 stri
```

NAME

strncmp — Compare a portion 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 — **C**ompare 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 — **C**opy 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

strrchr — Find the last occurrence of a character from one string in another.

SYNOPSIS

```
#include <string.h>
char* strpbrk( char* s1, const char* s2 );
```

DESCRIPTION

The **strrchr** 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. **strrchr** returns a pointer to the located character, or **NULL**, 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 = strrchr( string, 'f' );
        puts( found );
}
```

prints to standard output:

```
feet
```

NAME

strspn — Find the maximal initial substring that is composed from a specified set of characters.

SYNOPSIS

```
#include <string.h>
size_t strstr ( const char* s1, const char* s2 );
```

DESCRIPTION

The **strspn** function computes a maximal initial substring from **s1**. This substring will only contain characters from the set of characters contained in the string **s2**. The return value of **strspn** is the length of the computed substring.

EXAMPLE

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

void main ()
{
        char* string = "bow wow, yippie yay yippie yay";
        char* ok_set = "wob ";

        printf ( “%*s\n”, (int) strspn ( string, ok_set ), string );
}
```

prints to standard output:

```
bow wow
```

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, **NULL** 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 == 7.089990
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 preceed 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 <stdlib.h>

void main()
{
        char* hexstr = "0x3cdef709hexstr stopped";
        char* decstr = "709709709decstr stopped";
        char* octstr = "012341234octstr stopped";
        char* stopped;
        long result;

        printf( "result\t\tstop string\n" );
        result = strtol( hexstr, &stopped, 16 );
        printf( "%lx\t\t%s\n", result, stopped );

        result = strtol( decstr, &stopped, 10 );
        printf( "%ld\t\t%s\n", result, stopped );

        result = strtol( octstr, &stopped, 8 );
        printf( "%lo\t\t%s\n", result, stopped );
}
```

prints to standard output:

```
result          stop string
3cdef709        hexstr stopped
709709709       decstr stopped
12341234        octstr stopped
```

NAME

strtoul — String to unsigned long integer.

SYNOPSIS

```
#include <stdlib.h>
unsigned long int strtoul( const char* nptr, char** endptr, int base );
```

DESCRIPTION

The **strtoul** function converts and returns the string pointed to by **nptr** to a long integer. First **strtoul** 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 preceed 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.
strtol — String to long integer.

EXAMPLE

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

void main()
{
        char* hexstr = "0xbcdef709hexstr stopped";
        char* decstr = "709709709decstr stopped";
        char* octstr = "012341234octstr stopped";
        char* stopped;
        unsigned long result;

        printf( "result\t\tstop string\n" );
        result = strtoul( hexstr, &stopped, 16 );
        printf( "%lu\t\t%s\n", result, stopped );

        result = strtoul( decstr, &stopped, 10 );
        printf( "%lu\t\t%s\n", result, stopped );

        result = strtoul( octstr, &stopped, 8 );
        printf( "%lu\t\t%s\n", result, stopped );
}
```

prints to standard output:

```
result          stop string
3168728841      hexstr stopped
709709709       decstr stopped
2736796         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

DSP563CCC 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 ) == 1.619770
```

NAME

tanh — Hyperbolic tangent.

SYNOPSIS

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

DESCRIPTION

The **tanh** function computes and returns the hyperbolic tanget 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

tmpfile — Create a temporary binary file.

SYNOPSIS

```
#include <stdio.h>
FILE *tmpfile (  void );
```

DESCRIPTION

The function tmpfile will create a temporary file on the disk. The file will be automatically removed when the program terminates. The file will be opened with the mode “**wb+**”. If tmpfile fails, it returns a **NULL** pointer.

SEE ALSO

tmpnam — Generate a valid temporary file name.

NAME

tmpnam — Create a temporary file name.

SYNOPSIS

```
#include <stdio.h>
char *tmpnam (  char *s );
```

DESCRIPTION

The function **tmpnam** will create a string that could be used as a unique temporary file name. This function may be called as many as **TMP_MAX** times. Each time it will return a different string. If the argument **s** is **NULL**, then **tmpnam** will return an internal static buffer that may be clobbered by subsequent calls. If s is non-**NULL**, then it must point to a writable buffer of at least **L_tmpnam** characters.

SEE ALSO

tmpfile — Create a temporary binary file.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        char buffer[L_tmpnam];

        (void) fopen ( tmpnam ( buffer ), “w+” );
}
```

will create a temporary *text* file on the disk. Note that unlike when **tmpfile** is called, one must remove any files created using **fopen** and **tmpnam**.

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

ungetc — Push a character back onto an input stream.

SYNOPSIS

```
#include <stdio.h>
int ungetc ( int c, FILE *stream );
```

DESCRIPTION

The function **ungetc** converts the argument **c** to an **unsigned char**, and pushes it back onto the specified input stream. Pushed characters will be read back in reverse order by any functions reading from said stream. If a call is made to a file positioning function, such as **fseek**, all pushed characters will be lost. Only one call to **ungetc** before a read from the stream is allowed. **EOF** cannot be pushed. **ungetc** returns **EOF** upon failure, while the converted value is returned upon success.

SEE ALSO

tmpfile — Create a temporary file.

EXAMPLE

```
#include <stdio.h>

void main ()
{
        char peek = getchar ();
        putchar ( peek );
        ungetc ( peek, stdin );
        putchar ( getchar ());
}
```

will print the first character from standard input twice on standard output.

NAME

vfprintf — Write formatted output to a stream using a va_list.

SYNOPSIS

```
#include <stdio.h>
int vfprintf ( FILE *stream, const char *format, va_list arg );
```

DESCRIPTION

The function **vfprintf** is exactly the same as the function **fprintf** except that an existing **va_list** is used in place of a series of arguments. The macro **va_start** must have been invoked on the argument arg before the call to **vfprintf** is made. **vfprintf** returns the number of characters printed. On error, **vfprintf** returns a negative value.

SEE ALSO

fprintf — Write formatted output to a stream.

EXAMPLE

```
#include <stdio.h>

int printf ( const char *format, ... )
{
        va_list ap;
        int result;

        va_start ( ap, format );
        result = vfprintf ( stdout, format, ap );
        va_end ( ap );
        return result;
}
```

is essentially the library function **printf**.

NAME

vprintf — Write formatted output to standard output using a va_list.

SYNOPSIS

```
#include <stdio.h>
int vprintf ( const char *format, va_list arg );
```

DESCRIPTION

The function **vprintf** is exactly the same as the function **printf** except that an existing **va_list** is used in place of a series of arguments. The macro **va_start** must have been invoked on the argument arg before the call to **vprintf** is made. **vprintf** returns the number of characters printed. On error, **vprintf** returns a negative value.

SEE ALSO

printf — Write formatted output to standard output.

EXAMPLE

```
#include <stdio.h>

int printf ( const char *format, ... )
{
        va_list ap;
        int result;

        va_start ( ap, format );
        result = vprintf ( format, ap );
        va_end ( ap );
        return result;
}
```

is essentially the library function **printf**.

NAME

vsprintf — Write formatted output to a string using a va_list.

SYNOPSIS

```
#include <stdio.h>
int vsprintf ( char *s, const char *format, va_list arg );
```

DESCRIPTION

The function **vsprintf** is exactly the same as the function **printf** except that an existing **va_list** is used in place of a series of arguments. The macro **va_start** must have been invoked on the argument arg before the call to **vsprintf** is made. **vsprintf** returns the number of characters printed. On error, **vsprintf** returns a negative value.

SEE ALSO

sprintf — Write formatted output to a string.

EXAMPLE

```
#include <stdio.h>

int sprintf ( char *s, const char *format, ... )
{
        va_list ap;
        int result;

        va_start ( ap, format );
        result = vsprintf ( s, format, ap );
        va_end ( ap );
        return result;
}
```

is essentially the library function **sprintf**.

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 DSP56300 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 use 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[] = L"abcdefgh";

void main()
{
        char* ptr = (char*) wstr;
        int convert;

        convert = wcstombs( array, wstr, 10 );

        printf( "packed array:\n" );
        while ( *ptr != 0 )
        {
                printf( "%0.4x ", *ptr++ );
        }
        printf( "\n\n" );

        printf("%d chars extracted, unpacked array:\n", convert );
        ptr = array;
        while ( *ptr != 0 )
        {
                printf( "%0.4x ", *ptr++ );
        }
        printf( "\n" );
}
```

prints to standard output:

```
packed array:
6162 6364 6566 6768

8 chars extracted, unpacked array:
0061 0062 0063 0064 0065 0066 0067 0068
```

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 **NULL**, **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 -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 DSP56300 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 use the ANSI data type **wchar_t** and the special ANSI **multibyte** and **wide character** library routines.

EXAMPLE

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

char mbarray[8];

void main()
{
        wchar_t wide = L'ab';
        char* ptr = mbarray;
        int convert;

        convert = wctomb( mbarray, wide );

        printf( "packed char looks like:\n" );
        printf( "%0.4x\n\n", wide );

        printf( "%d extracted chars looks like:\n", convert );
        while ( *ptr != 0 )
        {
                printf( "%0.4x ", *ptr++ );
        }
        printf( "\n" );
}
```

prints to standard output:

```
packed char looks like:
6162

2 extracted chars looks like:
0061 0062
```

Appendix B
Utilities

There are several utility programs available with the DSP563CCC compiler. They are:

1. asm56300
2. cldinfo
3. cldlod
4. cofdmp
5. dsplib
6. dsplnk
7. run563
8. srec

These programs are described in detail in the following pages.

NAME

asm56300 — Freescale DSP56300 Family Assembler

SYNOPSIS

asm56300 [**-A**] [**-B** [<*objfil*>]] [**-D** <*symbol*> <*string*>] [**-F** <*argfil*>][**-G**] [**-I** <*ipath*>] [**-L** [<*lstfil*>]] [**-M** <*mpath*>] [**-O** <*opt*> [, <*opt*> ...]][**-R** <*rev*> [, <*rev*>...]] [**-V**] <files...>

DESCRIPTION

asm56300 is a program that processes source program statements written in DSP56300 assembly language, translating these source statements into object programs compatible with other DSP56300 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:

asm56300 -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:

asm56300 -b*main 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:

asm56300 -b *main* **-v** *io*

Alternatively, two successive hyphens may be used to indicate the end of the option list:

asm56300 -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 Freescale 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.

-G

Send source file line number information to the object file. This option is valid only in conjunction with -B command line option. The generated line number information can be used by debuggers to provide source-level debugging.

-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.

-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 Freescale 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 the name of a Freescale DSP COFF format object file. Only a single file name may be supplied.

NAME

cldlod — Freescale COFF to LOD Format converter

SYNOPSIS

cldlod cldfile > lodfile

DESCRIPTION

cldlod is a utility that converts a binary COFF object file into an ascii LOD file.

cldfile is an operating system compatible filename which contains COFF information. Only a single file name may be supplied, and it must be explicit; there is no default extension for the input file.

lodfile is an LOD file.

NAME

cofdmp —Freescale DSP COFF File Dump Utility

SYNOPSIS

cofdmp [**-cfhlorstv**] [-d file] files

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.

-d Dump to output file.

-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 — Freescale 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 Freescale 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 newlines). 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 — Freescale 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**] [**-X**<opt>[, <opt>...]] [**-Z**] **<**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 Freescale DSP simulator or
2. converted to Freescale 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 -b*main 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, then 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 basename (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.

-X<opt>[,<opt>,...,<opt>]
The -X option directs the linker to perform differently than the standard operation of the linker.The options are described below with their different op-

erations performed. All options may be preceded by **NO** to reverse their meaning. The **-X**<opt> sequence can be repeated for as many options as desired.

Option	Meaning
XC	Relative terms from different sections used in an expression cause an error.
RSV	Reserve special target processor memory areas.
AEC	Check form of address expressions.
RO	Allow region overlap.
ESO	Do not allocate memory below ordered sections.
ASC	Enable absolute section bounds checking.

-Z The linker strips source file line number and symbol information from the input file. Symbol information normally is retained for debugging purposes. This option has no effect if incremental linking is being done (see the **-I** option).

NAME

run563 — Freescale DSP563XX Simulator Based Execution Device.

SYNOPSIS

run563 [**-b** BCR_VALUE] [**-s** STATE_FILE] [**t**] [file]

DESCRIPTION

run563 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 DSP563XX bus control register. Default **bcr** value is $212421.

-c Store the simulator state to a file if the execution of run563 is terminated prematurely (for instace by killing the process).

-d DEVICE
Use the string specified to select an alternate execution device.

-l Be loud and let the user know what is occuring (loading, executing, etc.). Current status is printed to stderr.

-t Update global C variable **__time** at each clock tick. Used to benchmark code. Report the number of instruction cycles that elapsed upon program termination.

-x Indicates the input file was generated from the x memory model of C compiler. The C compiler may use x, y or l memory model for the compilation (refer to the option -m of the C compiler), and this option directs the **run563** program to use the X memory space for data buffer. Without this option, y memory model or L memory model is assumed.

-v Display the version number of the program.

NAME

srec — Freescale DSP S-Record Conversion Utility

SYNOPSIS

srec [**-b** | **-w**] [**-m** | **-s**] [**-l**] [**-r**] <files ...>

DESCRIPTION

srec converts Freescale DSP *.cld* and *.lod* format files into Freescale 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. Freescale 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

-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.

Index

—D—

—E—

—F—

—G—

—H—

—I—

—J—

—L—

—M—

—N—

—O—

—P—

—Q—

—R—

—S—

—T—

—U—

—V—

—W—

—X—

—Y—