TMS320C6000 Optimizing Compiler User's Guide

Literature Number: SPRU187L
May 2004

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

The *TMS320C6000 Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

- Compiler
- Assembly optimizer
- Standalone simulator
- Library-build utility
- C++ name demangler

The TMS320C6000™ C/C++ compiler accepts C and C++ code conforming to the International Organization for Standardization (ISO) standards for these languages, and produces assembly language source code for the TMS320C6x device. The compiler supports the 1989 version of the C language.

This user's guide discusses the characteristics of the C/C++ compiler. It assumes that you already know how to write C programs. *The C Programming Language* (second edition), by Brian W. Kernighan and Dennis M. Ritchie, describes C based on the ISO C standard. You can use the Kernighan and Ritchie (hereafter referred to as K&R) book as a supplement to this manual. References to K&R C (as opposed to ISO C) in this manual refer to the C language as defined in the first edition of Kernighan and Ritchie's *The C Programming Language*.

Before you use the information about the C/C++ compiler in this user's guide, you should install the C/C++ compiler tools.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample of C code:

```
#include <stdio.h>

main()

{
   printf("hello, world\n");
}
```

- In syntax descriptions, the instruction, command, or directive is in a **boldface** typeface and parameters are in *italics*. Portions of a syntax that are in bold must be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered. Syntax that is entered on a command line is centered in a bounded box:

  cl6x [*options*] [*filenames*] [*−z* [*link_options*] [*object files*]]

  Syntax used in a text file is left justified in a bounded box:

  inline *return-type function-name* (*parameter declarations*) **{***function***}**

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. This is an example of a command that has an optional parameter:

  load6x [*options*] *filename.out*

  The load6x command has two parameters. The first parameter, *options*, is optional. The second parameter, *filename.out*, is required.

- Braces ({ and }) indicate that you must choose one of the parameters within the braces; you do not enter the braces themselves. This is an example of a command with braces that are not included in the actual syntax but indicate that you must specify either the −c or −cr option:

  cl6x −z {**−c** | **−cr**} *filenames* [**−o** *name.out*] **−l** *libraryname*

- The TMS320C6200 core is referred to as C6200. The TMS320C6400 core is referred to as 6400. The TMS320C6700 core is referred to as 6700. TMS320C6000 and C6000 can refer to either C6200, 6400 or C6700.

Related Documentation From Texas Instruments

The following books describe the TMS320C6000 and related support tools. To obtain any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477−8924. When ordering, identify the book by its title and literature number (located on the title page):

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the C6000 generation of devices.

Code Composer User's Guide (literature number SPRU328) explains how to use the Code Composer development environment to build and debug embedded real-time DSP applications.

TMS320C6000 DSP/BIOS User's Guide (literature number SPRU303) describes how to use TMS320C6000™ DSP/BIOS tools and APIs to analyze embedded real-time DSP applications.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the C6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201 and TMS320C6701 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port interface (HPI), multichannel buffered serial ports (McBSPs), direct memory access (DMA), enhanced DMA (EDMA), expansion bus, clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000™ DSPs and includes application program examples.

TMS320C6000 Technical Brief (literature number SPRU197) gives an introduction to the C6000 platform of digital signal processors, development tools, and third-party support.

Related Documentation

You can use the following books to supplement this user's guide:

ANSI X3.159−1989, Programming Language − C (Alternate version of the 1989 C Standard), American National Standards Institute

C: A Reference Manual (fourth edition), by Samuel P. Harbison, and Guy L. Steele Jr., published by Prentice Hall, Englewood Cliffs, New Jersey

International Standard ISO 14882 (1998) - Programming Languages - C++ (The C standard)

ISO/IEC 14882−1998, International Standard − Programming Languages − C++ (The C++ Standard), International Organization for Standardization

ISO/IEC 9899:1989, International Standard − Programming Languages − C (The 1989 C Standard), International Organization for Standardization

ISO/IEC 9899:1999, International Standard − Programming Languages − C (The C Standard), International Organization for Standardization

Programming Embedded Systems in C and C++, by Michael Barr, Andy Oram (Editor), published by O'Reilly & Associates; ISBN: 1565923545, February 1999

Programming in C, Steve G. Kochan, Hayden Book Company

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjame Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

The C++ Programming Language (second edition), Bjame Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

Trademarks

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Windows and Windows NT are registered trademarks of Microsoft Corporation.

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments Incorporated. Trademarks of Texas Instruments include: TI, XDS, Code Composer, Code Composer Studio, TMS320, TMS320C6000 and 320 Hotline On-line.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction

The TMS320C6000 is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembly optimizer, an assembler, a linker, and assorted utilities.

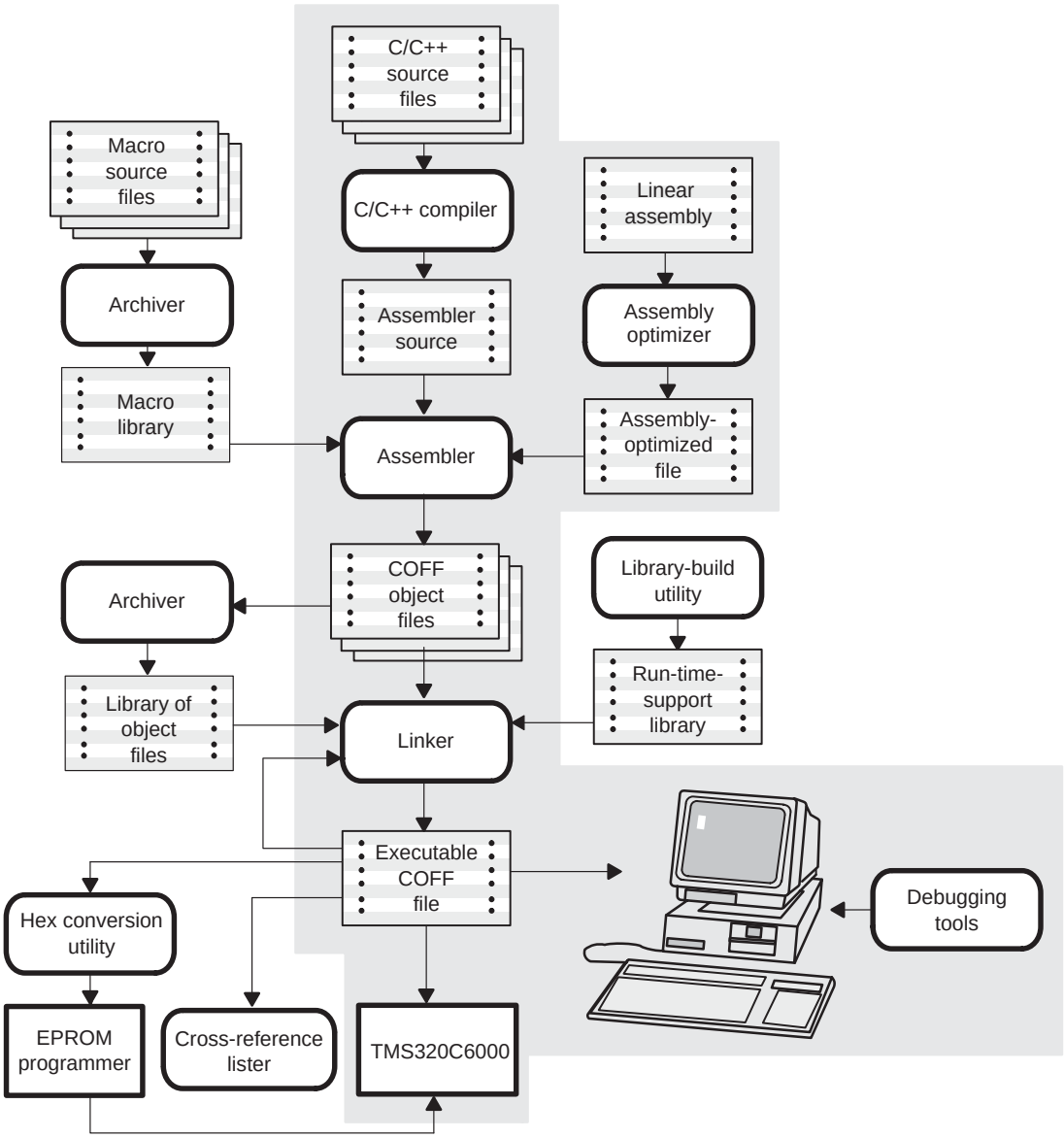
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembly optimizer is discussed in Chapter 4. The assembler and linker are discussed in detail in the *TMS320C6000 Assembly Language Tools User's Guide*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1−1 illustrates the C6000 software development flow. The shaded portion of the figure highlights the most common path of software development for C language programs. The other portions are peripheral functions that enhance the development process.

Figure 1−1. TMS320C6000 Software Development Flow



The following list describes the tools that are shown in Figure 1−1:

- The **assembly optimizer** allows you to write linear assembly code without being concerned with the pipeline structure or with assigning registers. It accepts assembly code that has not been register-allocated and is unscheduled. The assembly optimizer assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly that takes advantage of software pipelining. See Chapter 4, *Using the Assembly Optimizer*, for information about invoking the assembly optimizer, writing linear assembly code (.sa files), specifying functional units, and using assembly optimizer directives.

- The **C/C++ compiler** accepts C/C++ source code and produces C6000 assembly language source code. A **compiler**, **optimizer**, and an **interlist feature** are parts of the compiler:

  - The compiler enables you to compile, assemble, and link source modules in one step. If any input file has a .sa extension, the compiler program invokes the assembly optimizer.
  - The optimizer modifies code to improve the efficiency of C programs.
  - The interlist feature interweaves C/C++ source statements with assembly language output.

  See Chapter 2, *Using the C/C++ Compiler*, for information about how to invoke the C/C++ compiler, the optimizer, and the interlist feature using the compiler program.

- The **assembler** translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). The *TMS320C6000 Assembly Language Tools User's Guide* explains how to use the assembler.

- The **linker** combines object files into a single executable object module. As it creates the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. See Chapter 5, *Linking C/C++ Code*, for information about invoking the linker. See the *TMS320C6000 Assembly Language Tools User's Guide* for a complete description of the linker.

- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object modules. The *TMS320C6000 Assembly Language Tools User's Guide* explains how to use the archiver.

- You can use the **library-build utility** to build your own customized run-time-support library (see Chapter 10, *Library-Build Utility*). Standard run-time-support library functions for C and C++ are provided as source code in rts.src. The object code for the run-time-support functions is compiled for little-endian mode versus big-endian mode into standard libraries as follows:

  - For little-endian C and C++ code—rts6200.lib, rts6400.lib, and rts6700.lib
  - For big-endian C and C++ code—rts6200e.lib, rts6400e.lib, and rts6700e.lib

  The **run-time-support libraries** contain the ISO standard run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the C6000 compiler. See Chapter 8, *Run-Time Environment.*

- The **hex conversion utility** converts a COFF object file into TI-Tagged, ASCII-hex, Intel, Motorola-S, or Tektronix object format. You can download the converted file to an EPROM programmer. The *TMS320C6000 Assembly Language Tools User's Guide* explains how to use the hex conversion utility.

- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *TMS320C6000 Assembly Language Tools User's Guide* explains how to use the cross-reference utility.

- The main product of this development process is a module that can be executed in a **TMS320C6000** device. You can use one of several debugging tools to refine and correct your code. Available products include:

  - An instruction-accurate and clock-accurate software simulator
  - An XDS emulator

- The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

  For information about these debugging tools, see the *TMS320C6000 Code Composer Studio Tutorial* and the *Code Composer Studio User's Guide*.

1.2 C/C++ Compiler Overview

The C6000 C/C++ compiler is a full-featured optimizing compiler that translates standard ISO C programs into C6000 assembly language source. The following subsections describe the key features of the compiler.

1.2.1 ISO Standard

The following features pertain to ISO standards:

- **ISO-standard C**

  The C6000 C/C++ compiler fully conforms to the ISO C standard as defined by the ISO specification and described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The ISO C standard includes extensions to C that provide maximum portability and increased capability.

- **ISO-standard C++**

  The C6000 C/C++ compiler supports C++ as defined by the ISO C++ Standard and described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM). The compiler also supports embedded C++.

  For a description of unsupported C++ features, see section 7.2, *Characteristics of TMS320C6000 C++*, on page 7-5.

- **ISO-standard run-time support**

  The compiler tools come with a complete run-time library. All library functions conform to the ISO C/C++ library standard. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. The library includes the ISO C subset as well as those components necessary for language support. For more information, see Chapter 8, *Run-Time Environment.*

1.2.2 Output Files

The following features pertain to output files created by the compiler:

- **Assembly source output**

  The compiler generates assembly language source files that you can inspect easily, enabling you to see the code generated from the C/C++ source files.

- **COFF object files**

  Common object file format (COFF) allows you to define your system's memory map at link time. This maximizes performance by enabling you to link C/C++ code and data objects into specific memory areas. COFF also supports source-level debugging.

- **EPROM programmer data files**

  For stand-alone embedded applications, the compiler has the ability to place all code and initialization data into ROM, allowing C/C++ code to run from reset. The COFF files output by the compiler can be converted to EPROM programmer data files by using the hex conversion utility, as described in the *TMS320C6000 Assembly Language Tools User's Guide*.

1.2.3 Compiler Interface

The following features pertain to interfacing with the compiler:

- **Compiler program**

  The compiler tools include a compiler program that you use to compile, assembly optimize, assemble, and link programs in a single step. For more information, see section 2.1, *About the compiler Program,* on page 2-2.

- **Flexible assembly language interface**

  The compiler has straightforward calling conventions, so you can write assembly and C functions that call each other. For more information, see Chapter 8, *Run-Time Environment*.

1.2.4 Compiler Operation

The following features pertain to the operation of the compiler:

- **Integrated preprocessor**

  The C/C++ preprocessor is integrated with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available. For more information, see section 2.5, *Controlling the Preprocessor*, on page 2-26.

- **Optimization**

  The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C/C++ source. General optimizations can be applied to any C/C++ code, and C6000-specific optimizations take advantage of the features specific to the C6000 architecture. For more information about the C/C++ compiler's optimization techniques, see Chapter 3, *Optimizing Your Code*.

1.2.5 Utilities

The following features pertain to the compiler utilities:

- **Source interlist feature**

  The compiler tools include a utility that interlists your original C/C++ source statements into the assembly language output of the compiler. This utility provides you with a method for inspecting the assembly code generated for each C/C++ statement. For more information, see section 2.13, *Using the Interlist Feature,* on page 2-46.

- **Library-build utility**

  The library-build utility (mk6x) lets you custom-build object libraries from source for any combination of run-time models or target CPUs. For more information, see Chapter 10, *Library-Build Utility*.

- **Stand-alone simulator**

  The stand-alone simulator (load6x) loads and runs an executable COFF .out file. When used with the C I/O libraries, the stand-alone simulator supports all C I/O functions with standard output to the screen. For more information, see Chapter 6, *Using the Stand-Alone Simulator*.

- **C++ name demangler**

  The C++ name demangler (dem6x) is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code. For more information, see Chapter 11, *C++ Name Demangler*.

1.3 Code Composer Studio and the Compiler

Code Composer Studio provides a graphical interface for using the code generation tools.

A Code Composer Studio project manages the information needed to build a target program or library. A project records:

- ❑ Filenames of source code and object libraries
- ❑ Compiler, assembler, and linker options
- ❑ Include file dependencies

When you build a project with Code Composer Studio, the appropriate code generation tools are invoked to compile, assemble, and/or link your program.

Compiler, assembler, and linker options can be specified within Code Composer Studio's Build Options dialog. Nearly all command line options are represented within this dialog. Options that are not represented can be specified by typing the option directly into the editable text box that appears at the top of the dialog.

The information in this book describes how to use the code generation tools from the command line interface. For information on using Code Composer Studio, consult the *Code Composer Studio User's Guide*. For information on setting code generation tool options within Code Composer Studio, consult the *Code Generation Tools Help*.

Chapter 2

Using the C/C++ Compiler

The compiler translates your source program in code that the TMS320C6x can execute. Source code must be compiled, assembled, and linked to create an executable object file. All of these steps are executed at once by using the compiler, cl6x.

This chapter provides a complete description of how to use the cl6x to compile, assemble, and link your programs. Also included, are descriptions of the preprocessor inline function expansion features and interlist utility.

Topic **Page**

2.1 About the Compiler

The compiler cl6x allows you to compile, assemble, and optionally link in one step. The compiler performs the following steps on one or more source modules:

- The **compiler**, which includes the parser and optimizer, accepts C/C++ source code and produces C6x assembly language source code.

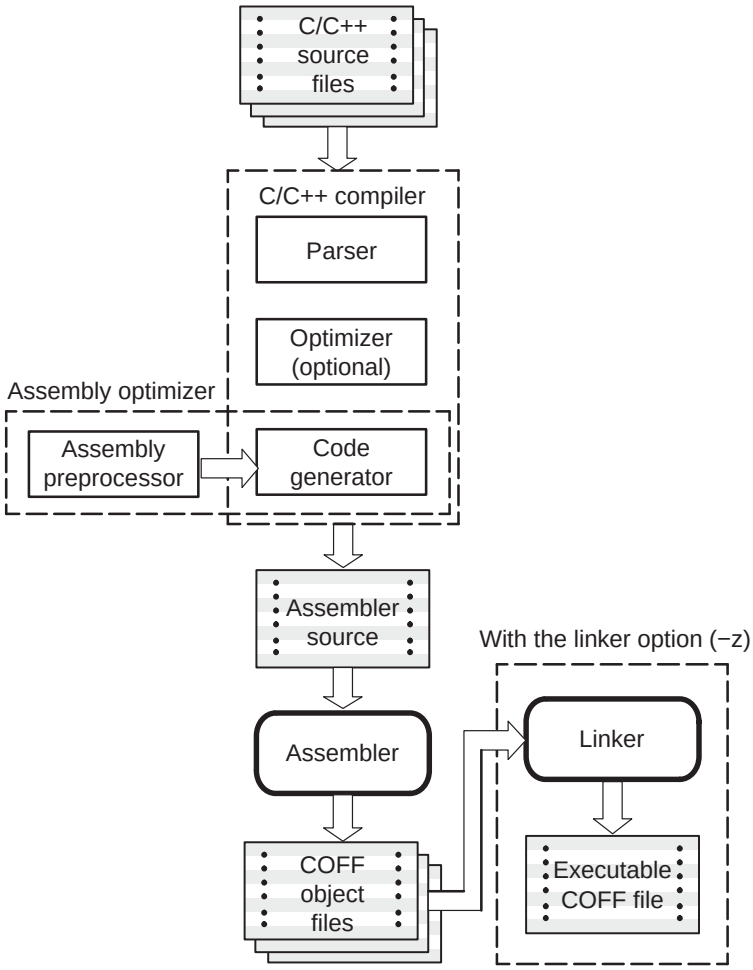
  You can compile C and C++ files in a single command. The compiler uses the conventions for filename extensions to distinguish between different file types. See section 2.3.4, *Specifying Filenames*, for more information.

- The **assembler** generates a COFF object file.

- The **linker** combines your object files to create an executable object file. The link step is optional, so you can compile and assemble many modules independently and link the later. See Chapter 5, *Linking C/C++ Code,* for information about linking the files in a separate step.

By default, the compiler does not perform the link step. You can invoke the linker by using the -z compiler option. Figure 2−1 illustrates the path the compiler takes with and without using the linker.

Figure 2−1. C/C++ Compiler



For a complete description of the assembler and the linker, see the *TMS320C6000 Assembly Language Tools User's Guide*.

2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

cl6x [*options*] [*filenames*] [**–z** [*link_options*] [*object files*]]

cl6x	Command that runs the compiler and the assembler
options	Options that affect the way the compiler processes input files. The options are listed in Table 2−1 on page 2-6.
filenames	One or more C/C++ source files, assembly language source files, linear assembly files, or object files
–z	Option that invokes the linker. See Chapter 5, *Linking C/C++ Code,* for more information about invoking the linker.
link_options	Options that control the linking process
object files	Name of the additional object files for the linking process

The arguments to cl6x are of three types:

- Compiler options
- Linker options
- Files

The -z linker option is the signal that linking is to be performed. If the -z linker option is used, any compiler options must precede the -z linker option, and any other linker options must follow the -z linker option.

Source code filenames must be placed before the -z linker option. Additional object file filenames can be placed after the -z linker option. Otherwise, options and filenames can be placed in any order.

For example, if you want to compile two files named symtab.c and file.c, assemble a third file named seek.asm, assembly optimize a fourth file named find.sa, and link to create an executable program called myprogram.out, you will enter:

```
cl6x –q symtab.c file.c seek.asm find.sa -z –llnk.cmd -lrts6200.lib -o myprogram.out
```

As cl6x encounters each source file, it prints the C/C++ filenames and assembly language filenames in square brackets ([ ]), and linear assembly filenames in braces ({ }). Entering the example command produces these messages:

```
[symtab.c]
[file.c]
[seek.asm]
{find.sa}
<Linking>
```

2.3 Changing the Compiler's Behavior With Options

Options control the operation of both the compiler and the programs it runs. This section provides a description of option conventions and an option summary table. It also provides detailed descriptions of the most frequently used options, including options used for type-checking and assembling.

The following apply to the compiler options:

- Options are preceded by one or two hyphens.
- Options are case sensitive.
- Options are either single letters or sequences of characters.
- Individual options cannot be combined.
- An option with a *required* parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to undefine a constant can be expressed as −U=*name*. Although not recommended, you can separate the option and the parameter with or without a space, as in −U *name* or −U*name*.
- An option with an *optional* parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to specify the maximum amount of optimization can be expressed as −O=3. Although not recommended, you can specify the parameter directly after the option, as in −O3. No space is allowed between the option and the optional parameter, so −O 3 is not accepted.
- Files and options except the −z option can occur in any order. The −z option must follow all other compiler options and precede any linker options.

You can define default options for the compiler by using the C_OPTION or C6X_C_OPTION environment variable. For a detailed description of these environment variables, see section 2.4, *Setting Default Compiler Options (C_OPTION and C6X_C_OPTION)*, on page 2-25.

Table 2−1 summarizes all options (including linker options). Use the page references in the table for more complete descriptions of the options.

For an online summary of the options, enter **cl6x** with no parameters on the command line.

Table 2−1. Compiler Options Summary

(a) Options that control the compiler

Option	Effect	Page
−@*filename*	Interprets contents of a file as an extension to the command line. Multiple −@ instances may be used.	2-15
−c	Disables linking (negates −z)	2-15, 5-4
−D*name*[=*def*]	Predefines *name*	2-15
−h	Help	2-16
−I*directory*	Defines #include search path	2-16, 2-28
−k	Keeps the assembly language (.asm) file	2-16
−n	Compiles or assembly optimizes only	2-16
−q	Suppresses progress messages (quiet)	2-16
−s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	2-17
−ss	Interlists C source and assembly statements	2-17, 3-30
−U*name*	Undefines *name*	2-17
−−verbose	Displays a banner and function progress information	−−
−z	Enables linking	2-17

Table 2−1. Compiler Options Summary (Continued)

(b) Options that control symbolic debugging and profiling

Option	Effect	Page
−g	Enables symbolic debugging (equivalent to −−symdebug:dwarf)	2-18
−−profile:breakpt	Enables breakpoint-based profiling	2-18
−−symdebug:coff	Enables symbolic debugging using the alternate STABS debugging format	2-18, 3-33
−−symdebug:dwarf	Enables symbolic debugging using the DWARF debugging format (equivalent to −g)	2-18
−−symdebug:none	Disables all symbolic debugging	2-18
−−symdebug:skeletal	Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	2-18

(c) Options that change the default file extensions

Option	Effect	Page
−ea[.]*extension*	Sets a default extension for assembly source files	2-21
−ec[.]*extension*	Sets a default extension for C source files	2-21
−el[.]*extension*	Sets a default extension for linear assembly source files	2-21
−eo[.]*extension*	Sets a default extension for object files	2-21
−ep[.]*extension*	Sets a default extension for C++ source files	2-21
−es[.]*extension*	Sets a default extension for listing files	2-21

Table 2−1. Compiler Options Summary (Continued)

(d) Options that specify files

Option	Effect	Page
−fa*filename*	Identifies *filename* as an assembly source file regardless of its extension. By default, the compiler and assembler treat .asm files as assembly source files.	2-20
−fc*filename*	Identifies *filename* as a C source file regardless of its extension. By default, the compiler treats .c files as C source files.	2-20
−fg	Processes all source files with a C extension as C++ source files.	2-20
−fl*filename*	Identifies *filename* as a linear assembly source file regardless of its extension. By default, the compiler and assembly optimizer treat .sa files as linear assembly source files.	2-20
−fo*filename*	Identifies *filename* as an object code file regardless of its extension. By default, the compiler and linker treat .obj files as object code files.	2-20
−fp*filename*	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc and .cxx files as a C++ files.	2-20

(e) Options that specify directories

Option	Effect	Page
−fb*directory*	Specifies an absolute listing file directory	2-22
−ff*directory*	Specifies an assembly listing file and cross-reference listing file directory	2-22
−fr*directory*	Specifies an object file directory	2-22
−fs*directory*	Specifies an assembly file directory	2-22
−ft*directory*	Specifies a temporary file directory	2-22

Table 2−1. Compiler Options Summary (Continued)

(f) Options that are machine-specific

Option	Effect	Page
−−consultant	Generates compiler Consultant Advice	2-15
−ma	Indicates that a specific aliasing technique is used	3-25
−mb	Compiles C6400 code compatible with array alignment restrictions of version 4.0 tools or C6200/C6700 object code	2-45
−mc	Prevents reordering of associative floating-point operations	3-28
−me	Produces object code in big-endian format.	2-16
−speculate_loads*n*	Allows speculative execution of loads with bounded address ranges	3-14
−mi*n*	Specifies an interrupt threshold value	2-43
−ml*n*	Changes near and far assumptions on four levels (−ml0, −ml1, −and ml2, and −ml3)	2-16, 7-11
−mo	Turns on function subsections	5-13
−mr*n*	Make calls to run-time-support functions near (−mr0) or far (−mr1)	7-12
−ms*n*	Controls code size on four levels (−ms0, −ms1, −ms2, and −ms3)	3-17
−mt	Allows the compiler to make certain assumptions about aliasing and loops	3-26, 4-43
−mu	Turns off software pipelining	3-5
−mv*n*	Selects target version	2-17
−mw	Produce verbose software pipelining report	3-5

Table 2−1. Compiler Options Summary (Continued)

(g) Options that control parsing

Option	Effect	Page
−pe	Enables embedded C++ mode	7-38
−pi	Disables definition-controlled inlining (but −O3 optimizations still perform automatic inlining)	2-40
−pk	Allows K&R compatibility	7-36
−pl	Generates a raw listing file	2-36
−pm	Combines source files to perform program-level optimization	3-20
−pr	Enables relaxed mode; ignores strict ISO violations	7-38
−ps	Enables strict ISO mode (for C/C++, not K&R C)	7-38
−px	Generates a cross-reference listing file	2-35
−rtti	Enables run time type information (RTTI)	7-5

(h) Parser options that control preprocessing

Option	Effect	Page
−ppa	Continues compilation after preprocessing	2-29
−ppc	Performs preprocessing only. Writes preprocessed output, keeping the comments, to a file with the same name as the input but with a .pp extension	2-30
−ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility	2-30
−ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	2-30
−ppl	Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension	2-30
−ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension	2-29

Table 2−1. Compiler Options Summary (Continued)

(i) Parser options that control diagnostics

Option	Effect	Page
−pdel *num*	Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.)	2-33
−pden	Displays a diagnostic's identifiers along with its text	2-33
−pdf	Generates a diagnostics information file	2-33
−pdr	Issues remarks (nonserious warnings)	2-33
−pds *num*	Suppresses the diagnostic identified by *num*	2-33
−pdse *num*	Categorizes the diagnostic identified by *num* as an error	2-33
−pdsr *num*	Categorizes the diagnostic identified by *num* as a remark	2-33
−pdsw *num*	Categorizes the diagnostic identified by *num* as a warning	2-33
−pdv	Provides verbose diagnostics that display the original source with line-wrap	2-34
−pdw	Suppresses warning diagnostics (errors are still issued)	2-34

Table 2−1. Compiler Options Summary (Continued)

(j) Options that control optimization

Option	Effect	Page
−O0	Optimizes register usage	3-2
−O1	Uses −O0 optimizations and optimizes locally	3-2
−O2 or −O	Uses −O1 optimizations and optimizes globally	3-3
−O3	Uses −O2 optimizations and optimizes the file	3-3
−oi*size*	Sets automatic inlining size (−O3 only). If *size* is not specified, the default is 1.	3-29
−ol0 or −oL0	Informs the optimizer that your file alters a standard library function	3-18
−ol1 or −oL1	Informs the optimizer that your file declares a standard library function	3-18
−ol2 or −oL2	Informs the optimizer that your file does not declare or alter library functions. Overrides the −ol0 and −ol1 options (default).	3-18
−on0	Disables the optimization information file	3-19
−on1	Produces an optimization information file	3-19
−on2	Produces a verbose optimization information file	3-19
−op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	3-21
−op1	Specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code	3-21
−op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default)	3-21
−op3	Specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code	3-21
−os	Interlists optimizer comments with assembly statements	3-30

The machine-specific −ma, −mh*n*, −mi*n*, −ms*n*, and −mt options, see Table 2−1(f), also effect optimization.

Table 2−1. Compiler Options Summary (Continued)

(k) Options that control the assembler

Option	Effect	Page
−aa	Enables absolute listing	2-23
−ac	Makes case insignificant in assembly source files	2-23
−ad*name*	Sets the *name* symbol.	2-23
−ahc*filename*	Copies the specified file for the assembly module	2-23
−ahi*filename*	Includes the specified file for the assembly module	2-23
−al	Generates an assembly listing file	2-23
−apd	Performs preprocessing; lists only assembly dependencies	2-23
−api	Performs preprocessing; lists only included #include files	2-23
−as	Puts labels in the symbol table	2-24
−au*name*	Undefines the predefined constant *name*	2-24
−ax	Generates the cross-reference file	2-24

Table 2−1. Compiler Options Summary (Continued)

(l) Options that control the linker

Options	Effect	Page
−a	Generates absolute executable output	5-5
−abs	Produce absolute listing file	5-5
−ar	Generates relocatable executable output	5-5
−b	Disables merge of symbolic debugging information.	5-5
−c	Autoinitializes variables at run-time	5-5, 8-51
−cr	Initializes variables at loadtime	5-5, 8-51
−e *global_symbol*	Defines entry point	5-5
−f *fill_value*	Defines fill value	5-5
−g *global_symbol*	Keeps a *global_symbol* global (overrides −h)	5-5
−h	Makes global symbols static	5-5
−heap *size*	Sets heap size (bytes)	5-6
−I *directory*	Defines library search path	5-6
−j	Disable conditional linking	5-6
−l *libraryname*	Supplies library or command filename	5-6
−m *filename*	Names the map file	5-6
−o *name.out*	Names the output file	5-6
−q	Suppresses progress messages (quiet)	5-6
−priority	Satisfies each unresolved reference by the first library that contains a definition for that symbol	5-6
−r	Generates relocatable non-expendable output module	5-6
−s	Strips symbol table information and line number entries from the output module	5-6
−stack *size*	Sets stack size (bytes)	5-5
−U *symbol*	Creates unresolved external symbol	5-7
−w	Displays a message when an undefined output section is created	5-7
−x	Forces rereading of libraries	5-7

2.3.1 Frequently Used Options

Following are detailed descriptions of options that you will probably use frequently:

−@*filename* Appends the contents of a file to the command line. You can use this option to avoid limitations on command line length or C style comments imposed by the host operating system. Use a # or ; at the beginning of a line in the command file to include comments. You can also include comments by delimiting them with /* and */.

You can use the −@ option multiple times to specify multiple files. For instance, the following indicates that file3 should be compiled as source and file1 and file2 are −@ files:

```
cl6x –@ file1 –@ file2 file3
```

−c Suppresses the linker and overrides the −z option, which specifies linking. Use this option when you have −z specified in the C_OPTION or C6X_C_OPTION environment variable and you do not want to link. For more information, see section 5.1.3, *Disabling the Linker (−c Compiler Option)*, on page 5-4.

−−consultant Generates compile time loop information through the Compiler Consultant Advice tool. See the *TMS320C6000 Code Composer Studio Online Help* for more information about the Compiler Consultant Advice tool.

−D*name[=def]* Predefines the constant *name* for the preprocessor. This is equivalent to inserting #define *name def* at the top of each C source file. If the optional [=*def*] is omitted, the *name* is set to 1.If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows, use −Dname="\"string def\"". For example, `–Dcar="\"sedan\""`
- For UNIX, use −Dname='"string def"'. For example, `–Dcar='"sedan"'`
- For Code Composer Studio, enter the definition in a file and include that file with the −@ option.

−h Help

−I*directory* (uppercase i) Adds *directory* to the list of directories that the compiler searches for #include files. Each pathname must be preceded by the −I option. If you do not specify a directory name, the preprocessor ignores the −I option. For more information, see section 2.5.2.1, *Changing the #include File Search Path With the −I Option*, on page 2-28.

−k Retains the assembly language output from the compiler or assembly optimizer. Normally, the compiler deletes the output assembly language file after assembly is complete.

−me Produces code in big-endian format. By default, little-endian code is produced.

−ml*n* Generates large-memory model code on four levels (−ml0, −ml1, −ml2, and −ml3):

- **−ml0** defaults aggregate data (structs and arrays) to far
- **−ml1** defaults all function calls to far
- **−ml2** defaults all aggregate data and calls to far
- **−ml3** defaults all data and calls to far

If no level is specified, all data and functions default to near. Near *data* is accessed via the data page pointer more efficiently while near *calls* are executed more efficiently using a PC relative branch.

Use these options if you have too much static and extern data to fit within a 15-bit scaled offset from the beginning of the .bss section, or if you have calls where the called function is more than ±1M words away from the call site. The linker issues an error message when these situations occur. See section 7.4.4, *The near and far Keywords*, on page 7-11, and section 8.1.5, *Memory Models*, on page 8-6, for more information.

−mv *num* Selects the target CPU version (For more information about the −mv option, see page 2-17.)

−mw Produce verbose software pipelining report

−n Compiles or assembly optimizes only. The specified source files are compiled or assembly optimized but not assembled or linked. This option overrides −z. The output is assembly language output from the compiler.

−q Suppresses banners and progress information from all the tools. Only source filenames and error messages are output.

−s Invokes the interlist feature, which interweaves optimizer comments *or* C/C++ source with assembly source. If the optimizer is invoked (−O*n* option), optimizer comments are interlisted with the assembly language output of the compiler. If the optimizer is not invoked, C/C++ source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C/C++ statement. The −s option implies the −k option. The −s option can have a negative performance and/or code size impact.

−ss Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. The interlisted C statements may appear to be out of sequence. You can use the interlist feature with the optimizer by combining the −os and −ss options. For more information, see section 2.13, *Using the Interlist feature,* on page 2-46.The −ss option can have a negative performance and/or code size impact.

−U*name* Undefines the predefined constant *name*. This option overrides any −D options for the specified constant.

−z Runs the linker on the specified object files. The −z option and its parameters follow all other options on the command line. All arguments that follow −z are passed to the linker. For more information, see section 5.1, *Invoking the Linker Through the Compiler*, on page 5-2.

2.3.2 Selecting Target CPU Version (−mv Option)

Select the target CPU version using the last four digits of the TMS320C6000 part number. This selection controls the use of target-specific instructions and alignment, such as −mv6701 or −mv6412. Alternatively, you can also specify the family of the part, for example, −mv6400 or −mv6700. If this option is not used, the compiler generates code for the C6200 parts. If the −mv option is not specified, the code generated runs on all C6000 parts; however, the compiler does not take advantage of target-specific instructions or alignment.

2.3.3 Symbolic Debugging and Profiling Options

−g or **−−symdebug:dwarf**
Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The −g option disables many code generator optimizations, because they disrupt the debugger. You can use the −g option with the −o option to maximize the amount of optimization that is compatible with debugging (see section 3.12, *Debugging Optimized Code*, on page 3-33.

For more information on the DWARF debug format, see the *DWARF Debugging Information Format Specification*, 1992−1993, UNIX International, Inc.

−−profile:breakpt
Disables optimizations that would cause incorrect behavior when using a breakpoint-based profiler.

−symdebug:coff
Enables symbolic debugging using the alternate STABS debugging format. This may be necessary to allow debugging with older debuggers or custom tools, which do not read the DWARF format.

−symdebug:none
Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.

−−symdebug:skeletal
Generates as much symbolic debugging information as possible without hindering optimization. Generally, this consists of global-scope information only. This option reflects the default behavior of the compiler.

See section 2.3.9 on page 2-24 for a list of deprecated symbolic debugging options.

2.3.4 Specifying Filenames

The input files that you specify on the command line can be C source files, C++ source files, assembly source files, linear assembly files, or object files. The compiler uses filename extensions to determine the file type.

Extension	File Type
.c	C source
.C	Depends on operating system
.cpp, .cxx, .cc	C++ source
.sa	Linear assembly
.asm, .abs, or .s* (extension begins with s)	Assembly source
.obj	Object

Note: Case Sensitivity in Filename Extensions

Case sensitivity in filename extensions is determined by your operating system. If your operating system is not case sensitive, a file with a .C extension is interpreted as a C file. If your operating system is case sensitive, a file with a .C extension is interpreted as a C++ file.

The conventions for filename extensions allow you to compile C and C++ files and optimize and assemble assembly files with a single command.

For information about how you can alter the way that the compiler interprets individual filenames, see section 2.3.5 on page 2-20. For information about how you can alter the way that the compiler interprets and names the extensions of assembly source and object files, see section 2.3.7 on page 2-22.

You can use wildcard characters to compile or assemble multiple files. Wildcard specifications vary by system; use the appropriate form listed in your operating system manual. For example, to compile all of the files in a directory with the extension .cpp, enter the following:

```
cl6x *.cpp
```

Note: No Default Extension for Source Files is Assume

If you list a filename called *example* on the command line, the compiler assumes that the entire filename is *example* not *example.c*. No default extensions are added onto files that do not contain an extension.

2.3.5 Changing How the Compiler Program Interprets Filenames (−fa, −fc, −fg, −fl, −fo, and −fp Options)

You can use options to change how the compiler interprets your filenames. If the extensions that you use are different from those recognized by the compiler, you can use the −fa, −fc, −fl, −fo, and −fp options to specify the type of file. You can insert an optional space between the option and the filename. Select the appropriate option for the type of file you want to specify:

−fa*filename*	for an assembly language source file
−fc*filename*	for a C source file
−fl*filename*	for a linear assembly file
−fo*filename*	for an object file
−fp*filename*	for a C++ source file

For example, if you have a C source file called file.s and an assembly language source file called assy, use the −fa and −fc options to force the correct interpretation:

```
cl6x –fc file.s –fa assy
```

You cannot use the −fa, −fc, −fl, and −fo options with wildcard specifications.

The −fg option causes the compiler to process C files as C++ files. By default, the compiler treats files with a .c extension as C files. See section 2.3.4, Specifying Filenames, on page 2-19, for more information about filename extension conventions.

2.3.6 Changing How the Compiler Program Interprets and Names Extensions (−ea, −ec, −el, −eo, -ep, and −es Options)

You can use options to change how the compiler program interprets filename extensions and names the extensions of the files that it creates. The −ea, −el, and −eo options must precede the filenames they apply to on the command line. You can use wildcard specifications with these options. An extension can be up to nine characters in length. Select the appropriate option for the type of extension you want to specify:

−ea[**.**] *new extension*	for an assembly language file
−ec[**.**] *new extension*	for a C source file
−el[.] *new extension*	for a linear assembly source file
−eo[**.**] *new extension*	for an object file
−ep[**.**] *new extension*	for a C++ source file
−es[**.**] *new extension*	sets default extension for listing files

The following example assembles the file fit.rrr and creates an object file named fit.o:

```
cl6x –ea .rrr –eo .o fit.rrr
```

The period (.) in the extension and the space between the option and the extension are optional. You can also write the example above as:

```
cl6x –earrr –eoo fit.rrr
```

2.3.7 Specifying Directories (−fb, −ff, −fr, −fs, and −ft Options)

By default, the compiler program places the object, assembly, and temporary files that it creates into the current directory. If you want the compiler program to place these files in different directories, use the following options:

−fb*directory* Specifies the destination directory for absolute listing files. The default is to use the same directory as the object file directory. To specify an absolute listing file directory, type the directory's pathname on the command line after the −fb option:

```
cl6x –fb d:\abso_list
```

−ff*directory* Specifies the destination directory for assembly listing files and cross-reference listing files. The default is to use the same directory as the object file directory. To specify an assembly/cross-reference listing file directory, type the directory's pathname on the command line after the −ff option:

```
cl6x –ff d:\listing
```

−fr*directory* Specifies a directory for object files. To specify an object file directory, type the directory's pathname on the command line after the −fr option:

```
cl6x –fr d:\object
```

−fs*directory* Specifies a directory for assembly files. To specify an assembly file directory, type the directory's pathname on the command line after the −fs option:

```
cl6x –fs d:\assembly
```

−ft*directory* Specifies a directory for temporary intermediate files. To specify a temporary directory, type the directory's pathname on the command line after the −ft option:

```
cl6x –ft c:\temp
```

2.3.8 Options That Control the Assembler

Following are assembler options that you can use with the compiler:

−aa Invokes the assembler with the −a assembler option, which creates an absolute listing. An absolute listing shows the absolute addresses of the object code.

−ac makes case insignificant in the assembly language source files. For example, −c makes the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (this is the default).

−ad*name* Predefines the constant *name* for the assembler. If the optional [=*def*] is omitted, the *name* is set to 1.If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows, use −ad*name*="\"string def\"". For example, `-adcar="\"sedan\""`
- For UNIX, use −ad*name*="'string def"'. For example, `-adcar='"sedan"'`
- For Code Composer Studio, enter the definition in a file and include that file with the −@ option.

−ahc*filename* Invokes the assembler with the −hc assembler option to tell the assembler to copy the specified file for the assembly module. The file is inserted before source file statements. The copied file appears in the assembly listing files.

−ahi*filename* Invokes the assembler with the −hi assembler option to tell the assembler to include the specified file for the assembly module. The file is included before source file statements. The included file does not appear in the assembly listing files.

−al Invokes the assembler with the −l (lowercase L) assembler option to produce an assembly listing file.

−apd Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .ppa extension.

−api Performs preprocessing for assembly files, but instead of writing preprocessed output, writes a list of files included with the #include directive. The list is written to a file with the same name as the source file but with a .ppa extension.

−as Invokes the assembler with the −s assembler option to put labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

−au*name* Undefines the predefined constant *name*. This option overrides any −ad options for the specified constant.

−ax Invokes the assembler with the −x assembler option to produce a symbolic cross-reference in the listing file.

For more information about assembler options, see the *TMS320C6000 Assembly Language Tools User's Guide.*

2.3.9 Deprecated Options

Several compiler options have been deprecated. The compiler continues to accept these options, but they are not recommended for use. Future releases of the tools will not support these options. Table 2−2 lists the deprecated options and the options that have replaced them.

Table 2−2. Compiler Backwards-Compatibility Options Summary

Old Option	Effect	New Option
−gp	Allows function-level profiling of optimized code	−g
−gt	Enables symbolic debugging using the alternate STABS debugging format	−−symdebug:coff
−gw	Enables symbolic debugging using the DWARF debugging format	−−symdebug:dwarf or −g

Additionally, the −−symdebug:profile_coff option has been added to enable function-level profiling of optimized code with symbolic debugging using the STABS debugging format (the −−symdebug:coff or −gt option).

2.4 Setting Default Compiler Options (C_OPTION and C_C6X_OPTION)

You might find it useful to set the compiler, assembler, and linker default options using the C_OPTION or C6X_C_OPTION environment variable. If you do this, the compiler uses the default options and/or input filenames that you name with C6X_C_OPTION or C_OPTION every time you run the compiler.

Setting the default options with these environment variables is useful when you want to run the compiler consecutive times with the same set of options and/or input files. After the compiler reads the command line and the input filenames, it looks for the C6X_C_OPTION environment variable first and then reads and processes it. If it does not find the C6X_C_OPTION, it reads the C_OPTION environment variable and processes it.

The table below shows how to set the C_OPTION environment variable. Select the command for your operating system:

Operating System	Enter
UNIX (Bourne shell)	**C_OPTION="***option*$_1$ [*option*$_2$ . . .]**"** **export C_OPTION**
Windows™	**set C_OPTION=***option*$_1$[;*option*$_2$ . . .]

Environment variable options are specified in the same way and have the same meaning as they do on the command line. For example, if you want to always run quietly (the −q option), enable C/C++ source interlisting (the −s option), and link (the −z option) for Windows, set up the C_OPTION environment variable as follows:

```
set C_OPTION=-q -s -z
```

In the following examples, each time you run the compiler, it runs the linker. Any options following −z on the command line or in C_OPTION are passed to the linker. This enables you to use the C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the command line. If you have set −z in the environment variable and want to compile only, use the compiler −c option. These additional examples assume C_OPTION is set as shown above:

```
cl6x   *c                  ; compiles and links
cl6x   -c *.c              ; only compiles
cl6x   *.c -z lnk.cmd      ; compiles and links using a
                           ; command file
cl6x   -c *.c -z lnk.cmd   ; only compiles (-c overrides -z)
```

For more information about compiler options, see section 2.3, *Changing the Compiler's Behavior With Options*, on page 2-5. For more information about linker options, see section 5.2, *Linker Options*, on page 5-5.

2.5 Controlling the Preprocessor

This section describes specific features that control the C6000 preprocessor, which is part of the parser. A general description of C preprocessing is in section A12 of K&R. The C6000 C/C++ compiler includes standard C/C++ preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles:

- ❏ Macro definitions and expansions
- ❏ #include files
- ❏ Conditional compilation
- ❏ Various other preprocessor directives (specified in the source file as lines beginning with the # character)

The preprocessor produces self-explanatory error messages. The line number and the filename where the error occurred are printed along with a diagnostic message.

2.5.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 2−3.

Table 2−3. Predefined Macro Names

Macro Name	Description
_TMS320C6X	Always defined
_TMS320C6200	Defined if target is C6200
_TMS320C6400	Defined if target is C6400
_TMS320C6700	Defined if target is C6700
_BIG_ENDIAN	Defined if big-endian mode is selected (the −me option is used); otherwise, it is undefined
_INLINE	Expands to 1 if optimization is used; undefined otherwise. Regardless of any optimization, always undefined when −pi is used.
_LARGE_MODEL	Defined if large-model mode is selected (the −ml option is used); otherwise, it is undefined
_LARGE_MODEL_OPTION	Set to the large-model specified by −mln; otherwise, it is undefined.
_LITTLE_ENDIAN	Defined if little-endian mode is selected (the −me option is not used); otherwise, it is undefined

† Specified by the ISO standard

Table 2−3. Predefined Macro Names (Continued)

Macro Name	Description
_SMALL_MODEL	Defined if small-model mode is selected (the −ml option is not used); otherwise, it is undefined
_ _LINE_ _†	Expands to the current line number
_ _FILE_ _†	Expands to the current source filename
_ _DATE_ _†	Expands to the compilation date in the form *mmm dd yyyy*
_ _TIME_ _†	Expands to the compilation time in the form *hh:mm:ss*
_ _TI_COMPILER_VERSION_ _	Defined to a 3− (or more) digit integer that consists of a major version number and a 2-digit minor version number. The number does not contain a decimal. For example, version 5.14 is represented as 514.
_ _STDC_ _†	Defined to indicate that compiler conforms to ISO C Standard. See section 7.1, *Characteristics of TMS320C6000 C*, on page 7-2, for exceptions to ISO C conformance.

† Specified by the ISO standard

You can use the names listed in Table 2−3 in the same manner as any other defined name. For example,

```
printf ( "%s %s" , __TIME__ , __DATE__);
```

translates to a line such as:

```
printf ("%s %s" , "13:58:17", "Jan 14 1997");
```

2.5.2 The Search Path for #include Files

The #include preprocessor directive tells the compiler to read source statements from another file. When specifying the file, you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, partial path information, or a filename with no path information.

- If you enclose the filename in double quotes (" "), the compiler searches for the file in the following directories in this order:
  1) The directory that contains the current source file. The current source file refers to the file that is being compiled when the compiler encounters the #include directive.
  2) Directories named with the `−I` option
  3) Directories set with the C6X_C_DIR or C_DIR environment variable

- If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:
  1) Directories named with the −I option
  2) Directories set with the C6X_C_DIR then C_DIR environment variables

See section 2.5.2.1, *Changing the #include File Search Path (−I Option*) for information on using the −I option. See the code generation tools CD-ROM insert for information on the C_DIR environment variable.

2.5.2.1 Changing the #include File Search Path (−I Option)

The −I option names an alternate directory that contains #include files. The format of the −I option is:

−I=*directory1* [**−I**=*directory2* ...]

There is no limit to the number of −I options per invocation of the compiler; each −I option names one *directory*. In C source, you can use the #include directive without specifying any directory information for the file; instead, you can specify the directory information with the −I option. For example, assume that a file called source.c is in the current directory. The file source.c contains the following directive statement:

```
#include "alt.h"
```

Assume that the complete pathname for alt.h is:

UNIX	/6xtools/files/alt.h
Windows	c:\6xtools\files\alt.h

The table below shows how to invoke the compiler. Select the command for your operating system:

Operating System	Enter
UNIX	**cl6x** −I=6xtools/files source.c
Windows	**cl6x** −Ic:\6xtools\files source.c

Note: Specifying Path Information in Angle Brackets

If you specify the path information in angle brackets, the compiler applies that information *relative* to the path information specified with −I options and the C_DIR or C6X_C_DIR environment variable.

For example, if you set up C_DIR with the following command:

```
C_DIR "/usr/include;/usr/ucb"; export C_DIR
```

or invoke the compiler with the following command:

```
cl6x –I=/usr/include file.c
```

and file.c contains this line:

```
#include <sys/proc.h>
```

the result is that the included file is in the following path:

```
/usr/include/sys/proc.h
```

2.5.3 Generating a Preprocessed Listing File (−ppo Option)

The −ppo option allows you to generate a preprocessed version of your source file, with an extension of .pp. The compiler's preprocessing functions perform the following operations on the source file:

- Each source line ending in a backslash (\) is joined with the following line.
- Trigraph sequences are expanded.
- Comments are removed.
- #include files are copied into the file.
- Macro definitions are processed.
- All macros are expanded.
- All other preprocessing directives, including #line directives and conditional compilation, are expanded.

2.5.4 Continuing Compilation After Preprocessing (−ppa Option)

If you are preprocessing, the preprocessor performs preprocessing only. By default, it does not compile your source code. If you want to override this feature and continue to compile after your source code is preprocessed, use the −ppa option along with the other preprocessing options. For example, use −ppa with −ppo to perform preprocessing, write preprocessed output to a file with a .pp extension, and then compile your source code.

2.5.5 Generating a Preprocessed Listing File With Comments (−ppc Option)

The −ppc option performs all of the preprocessing functions except removing comments and generates a preprocessed version of your source file with a .pp extension. Use the −ppc option instead of the −ppo option if you want to keep the comments.

2.5.6 Generating a Preprocessed Listing File With Line-Control Information (−ppl Option)

By default, the preprocessed output file contains no preprocessor directives. If you want to include the #line directives, use the −ppl option. The −ppl option performs preprocessing only and writes preprocessed output with line-control information (#line directives) to a file with the same name as the source file but with a .pp extension.

2.5.7 Generating Preprocessed Output for a Make Utility (−ppd Option)

The −ppd option performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. The list is written to a file with the same name as the source file but with a .pp extension.

2.5.8 Generating a List of Files Included With the #include Directive (−ppi Option)

The −ppi option performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive. The list is written to a file with the same name as the source file but with a .pp extension.

2.6 Understanding Diagnostic Messages

One of the compiler's primary functions is to report diagnostics for the source program. When the compiler detects a suspect condition, it displays a message in the following format:

"*file.c***", line** *n***:** *diagnostic severity***:** *diagnostic message*

"*file.c***"**	The name of the file involved
line *n***:**	The line number where the diagnostic applies
diagnostic severity	The severity of the diagnostic message (a description of each severity category follows)
diagnostic message	The text that describes the problem

Diagnostic messages have an associated severity, as follows:

- A **fatal error** indicates a problem of such severity that the compilation cannot continue. Examples of problems that can cause a fatal error include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- An **error** indicates a violation of the syntax or semantic rules of the C/C++ language. Compilation continues, but object code is not generated.
- A **warning** indicates something that is valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- A **remark** is less serious than a warning. It indicates something that is valid and probably intended, but may need to be checked. Compilation continues and object code is generated (if no errors are detected). By default, remarks are not issued. Use the −pdr compiler option to enable remarks.

Diagnostics are written to standard error with a form like the following example:

```
"test.c", line 5: error: a break statement may only be used
          within a loop or switch
    break;
    ^
```

By default, the source line is omitted. Use the −pdv compiler option to enable the display of the source line and the error position. The above example makes use of this option.

The message identifies the file and line involved in the diagnostic, and the source line itself (with the position indicated by the ^ character) follows the message. If several diagnostics apply to one source line, each diagnostic has the form shown; the text of the source line is displayed several times, with an appropriate position indicated each time.

Long messages are wrapped to additional lines, when necessary.

You can use a command-line option (−pden) to request that the diagnostic's numeric identifier be included in the diagnostic message. When displayed, the diagnostic identifier also indicates whether the diagnostic can have its severity overridden on the command line. If the severity can be overridden, the diagnostic identifier includes the suffix −D (for *discretionary*); otherwise, no suffix is present. For example:

```
"Test_name.c",line 7: error #64-D: declaration does not
          declare anything
    struct {};
    ^

"Test_name.c",line 9: error #77: this declaration has no
          storage class or type specifier
    xxxxx;
    ^
```

Because an error is determined to be discretionary based on the error severity associated with a specific context, an error can be discretionary in some cases and not in others. All warnings and remarks are discretionary.

For some messages, a list of entities (functions, local variables, source files, etc.) is useful; the entities are listed following the initial error message:

```
"test.c", line 4: error: more than one instance of overloaded
          function "f" matches the argument list:
            function "f(int)"
            function "f(float)"
            argument types are: (double)
   f(1.5);
   ^
```

In some cases, additional context information is provided. Specifically, the context information is useful when the front end issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
"test.c", line 7: error: "A::A()" is inaccessible
   B x;
   ^

          detected during implicit generation of "B::B()" at
          line 7
```

Without the context information, it is difficult to determine to what the error refers.

2.6.1 Controlling Diagnostics

The C/C++ compiler provides diagnostic options that allow you to modify how the parser interprets your code. These options control diagnostics:

−pdel *num* Sets the error limit to *num*, which can be any decimal value. The compiler abandons compiling after this number of errors. (The default is 100.)

−pden Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (−pds, −pdse, −pdsr, and −pdsw).

This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix −D; otherwise, no suffix is present. See section 2.6, *Understanding Diagnostic Messages*, on page 2-31 for more information.

−pdf Produces a diagnostics information file with the same name as the corresponding source file with an *.err* extension

−pdr Issues remarks (nonserious warnings), which are suppressed by default

−pds *num* Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the −pden option first in a separate compile. Then use −pds *num* to suppress the diagnostic. You can suppress only discretionary diagnostics.

−pdse *num* Categorizes the diagnostic identified by *num* as an error. To determine the numeric identifier of a diagnostic message, use the −pden option first in a separate compile. Then use −pdse *num* to recategorize the diagnostic as an error. You can alter the severity of discretionary diagnostics only.

−pdsr*num* Categorizes the diagnostic identified by *num* as a remark. To determine the numeric identifier of a diagnostic message, use the −pden option first in a separate compile. Then use −pdsr *num* to recategorize the diagnostic as a remark. You can alter the severity of discretionary diagnostics only.

−pdsw *num* Categorizes the diagnostic identified by *num* as a warning. To determine the numeric identifier of a diagnostic message, use the −pden option first in a separate compile. Then use −pdsw *num* to recategorize the diagnostic as a warning. You can alter the severity of discretionary diagnostics only.

−pdv Provides verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line

−pdw Suppresses warning diagnostics (errors are still issued)

2.6.2 How You Can Use Diagnostic Suppression Options

The following example demonstrates how you can control diagnostic messages issued by the compiler.

Consider the following code segment:

```
int one();
int i;
int main()
{
   switch (i){
   case  1;
         return one ();
         break;
   default:
         return 0;
         break;
   }
}
```

If you invoke the compiler with the −q option, this is the result:

```
"err.c", line 9: warning: statement is unreachable
"err.c", line 12: warning: statement is unreachable
```

Because it is standard programming practice to include break statements at the end of each case arm to avoid the fall-through condition, these warnings can be ignored. Using the −pden option, you can find out the diagnostic identifier for these warnings. Here is the result:

```
[err.c]
"err.c", line 9: warning #111-D: statement is unreachable
"err.c", line 12: warning #111-D: statement is unreachable
```

Next, you can use the diagnostic identifier of 111 as the argument to the −pdsr option to treat this warning as a remark. This compilation now produces no diagnostic messages (because remarks are disabled by default).

Although this type of control is useful, it can also be extremely dangerous. The compiler often emits messages that indicate a less than obvious problem. Be careful to analyze all diagnostics emitted before using the suppression options.

2.7 Other Messages

Other error messages that are unrelated to the source, such as incorrect command-line syntax or inability to find specified files, are usually fatal. They are identified by the symbol >> preceding the message.

2.8 Generating Cross-Reference Listing Information (−px Option)

The −px option generates a cross-reference listing file that contains reference information for each identifier in the source file. (The −px option is separate from −ax, which is an assembler rather than a compiler option.) The cross-reference listing file has the same name as the source file with a *.crl* extension.

The information in the cross-reference listing file is displayed in the following format:

sym-id name X filename line number column number

sym-id — An integer uniquely assigned to each identifier

name — The identifier name

X — One of the following values:

X Value	Meaning
D	Definition
d	Declaration (not a definition)
M	Modification
A	Address taken
U	Used
C	Changed (used and modified in a single operation)
R	Any other kind of reference
E	Error; reference is indeterminate

filename — The source file

line number — The line number in the source file

column number — The column number in the source file

2.9 Generating a Raw Listing File (−pl Option)

The −pl option generates a raw listing file that can help you understand how the compiler is preprocessing your source file. Whereas the preprocessed listing file (generated with the −ppo, −ppc, −ppl, and −ppf preprocessor options) shows a preprocessed version of your source file, a raw listing file provides a comparison between the original source line and the preprocessed output. The raw listing file has the same name as the corresponding source file with a *.rl* extension.

The raw listing file contains the following information:

- ❏ Each original source line
- ❏ Transitions into and out of include files
- ❏ Diagnostics
- ❏ Preprocessed source line if nontrivial processing was performed (comment removal is considered trivial; other preprocessing is nontrivial)

Each source line in the raw listing file begins with one of the identifiers listed in Table 2−4.

Table 2−4. Raw Listing File Identifiers

Identifier	Definition
N	Normal line of source
X	Expanded line of source. It appears immediately following the normal line of source if nontrivial preprocessing occurs.
S	Skipped source line (false #if clause)
L	Change in source position, given in the following format: L *line number filename key* Where *line number* is the line number in the source file. The *key* is present only when the change is due to entry/exit of an include file. Possible values of *key* are as follows: 1 = entry into an include file 2 = exit from an include file

The −pl option also includes as defined in Table 2−5.

Table 2−5. Raw Listing File Diagnostic Identifiers

Diagnostic identifier	Definition
E	Error
F	Fatal
R	Remark
W	Warning

Diagnostic raw listing information is displayed in the following format:

S filename line number column number diagnostic

S	One of the identifiers in Table 2−5 that indicates the severity of the diagnostic
filename	The source file
line number	The line number in the source file
column number	The column number in the source file
diagnostic	The message text for the diagnostic

Diagnostics after the end of file are indicated as the last line of the file with a column number of 0. When diagnostic message text requires more than one line, each subsequent line contains the same file, line, and column information but uses a lowercase version of the diagnostic identifier. For more information about diagnostic messages, see section 2.6, *Understanding Diagnostic Messages*, on page 2-31.

2.10 Using Inline Function Expansion

When an inline function is called, the C/C++ source code for the function is inserted at the point of the call. This is known as inline function expansion. Inline function expansion is advantageous in short functions for the following reasons:

- ❏ It saves the overhead of a function call.
- ❏ Once inlined, the optimizer is free to optimize the function in context with the surrounding code.

There are several types of inline function expansion:

- ❏ Inlining with intrinsic operators (intrinsics are always inlined)
- ❏ Automatic inlining
- ❏ Definition-controlled inlining with the unguarded inline keyword
- ❏ Definition-controlled inlining with the guarded inline keyword

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. If your code size seems too large, see section 3.4, *Reducing Code Size (−ms Option)*, on page 3-17.

2.10.1 Inlining Intrinsic Operators

There are many intrinsic operators for the C6000. All of them are automatically inlined by the compiler. The inlining happens automatically whether or not you use the optimizer.

For details about intrinsics, and a list of the intrinsics, see section 8.5.2, *Using Intrinsics to Access Assembly Language Statements*, on page 8-26.

2.10.2 Automatic Inlining

When compiling C/C++ source code with the −O3 option, inline function expansion is performed on small functions . For more information, see section 3.10, *Automatic Inline Expansion (−oi Option)*, on page 3-29.

2.10.3 Unguarded Definition-Controlled Inlining

The inline keyword specifies that a function is expanded inline at the point at which it is called rather than by using standard calling procedures. The compiler performs inline expansion of functions declared with the inline keyword.

You must invoke the optimizer with any −O option (−O0, −O1, −O2, or −O3) to turn on definition-controlled inlining. Automatic inlining is also turned on when using −O3.

The following example shows usage of the inline keyword, where the function call is replaced by the code in the called function.

Example 2−1. Using the inline keyword

```
inline float volume_sphere(float r)
{
   return 4.0/3.0 * PI * r * r * r;
}
int foo(...)
{
   ...
   volume = volume_sphere(radius);
   ...
}
```

The −pi option turns off definition-controlled inlining. This option is useful when you need a certain level of optimization but do not want definition-controlled inlining.

2.10.4 Guarded Inlining and the _INLINE Preprocessor Symbol

When declaring a function in a header file as static inline, additional procedures should be followed to avoid a potential code size increase when inlining is turned off with −pi or the optimizer is not run.

In order to prevent a static inline function in a header file from causing an increase in code size when inlining gets turned off, use the following procedure. This allows external-linkage when inlining is turned off; thus, only one function definition will exist throughout the object files.

- Prototype a static inline version of the function. Then, prototype an alternative, nonstatic, externally-linked version of the function. Conditionally preprocess these two prototypes with the _INLINE preprocessor symbol, as shown in Example 2−2.
- Create an identical version of the function definition in a .c or .cpp file, as shown in Example 2−2.

In Example 2−2 there are two definitions of the strlen function. The first, in the header file, is an inline definition. This definition is enabled and the prototype is declared as static inline only if _INLINE is true (_INLINE is automatically defined for you when the optimizer is used and −pi is not specified).

The second definition, for the library, ensures that the callable version of strlen exists when inlining is disabled. Since this is not an inline function, the _INLINE preprocessor symbol is undefined (#undef) before string.h is included to generate a noninline version of strlen's prototype.

Example 2−2. How the Run-Time-Support Library Uses the _INLINE Preprocessor Symbol

(a) string.h

```
/*****************************************************************************/
/* string.h  vx.xx                                                           */
/* Copyright (c) 1993-1999 Texas Instruments Incorporated                    */
/* Excerpted ...                                                             */
/*****************************************************************************/

#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern _CODE_ACCESS
#endif

_IDECL size_t  strlen(const char *_string);

#ifdef _INLINE

/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;

   do n++; while (*++s);
   return n;
}

#endif
```

(b) strlen.c

```
/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
#undef _INLINE

#include <string.h>

{

_CODE_ACCESS size_t strlen(cont char * string)

   size_t      n = (size_t)-1;
   const char *s = string - 1;

   do n++; while (*++s);
   return n;
}
```

2.10.5 Inlining Restrictions

There are several restrictions on what functions can be inlined for both automatic inlining and definition-controlled inlining. Functions with local static variables or a variable number of arguments are not inlined, with the exception of functions declared as static inline. In functions declared as static inline, expansion occurs despite the presence of local static variables. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. Furthermore, inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this).

A function may be disqualified from inlining if it:

- ❏ Returns a struct or union
- ❏ Has a struct or union parameter
- ❏ Has a volatile parameter
- ❏ Has a variable length argument list
- ❏ Declares a struct, union, or enum type
- ❏ Contains a static variable
- ❏ Contains a volatile variable
- ❏ Is recursive
- ❏ Contains a pragma
- ❏ Has too large of a stack (too many local variables)

2.11 Interrupt Flexibility Options (−mi Option)

On the C6000 architecture, interrupts cannot be taken in the delay slots of a branch. In some instances the compiler can generate code that cannot be interrupted for a potentially large number of cycles. For a given real-time system, there may be a hard limit on how long interrupts can be disabled.

The −mi*n* option specifies an interrupt threshold value *n*. The threshold value specifies the maximum number of cycles that the compiler can disable interrupts. If the *n* is omitted, the compiler assumes that the code is never interrupted. In Code Composer Studio, to specify that the code is never interrupted, select the Interrupt Threshold check box and leave the text box blank in the Build Options dialog box on the Compiler tab, Advanced category.

If the -mi*n* option is not specified, then interrupts are only explicitly disabled around software pipelined loops. When using the −mi*n* option, the compiler analyzes the loop structure and loop counter to determine the maximum number of cycles it takes to execute a loop. If it can determine that the maximum number of cycles is less than the threshold value, the compiler generates the fastest/optimal version of the loop. If the loop is smaller than 6 cycles, interrupts are not able to occur because the loop is always executing inside the delay slots of a branch. Otherwise, the compiler generates a loop that can be interrupted (and still generate correct results—single assignment code), which in most cases degrades the performance of the loop.

The −mi*n* option does not comprehend the effects of the memory system. When determining the maximum number of execution cycles for a loop, the compiler does not compute the effects of using slow off-chip memory or memory bank conflicts. It is recommended that a conservative threshold value is used to adjust for the effects of the memory system.

See section 7.7.7, *The FUNC_INTERRUPT_THRESHOLD Pragma*, on page 7-24 or the *TMS320C6000 Programmer's Guide* for more information.

Note: RTS Library Files Are Not Built With −mi Option

The run-time-support library files provided with the compiler are not built with the interrupt flexibility option. Please refer to the readme file to see how the run-time-support library files were built for your release. See Chapter 10, *Library-Build Utility*, to build your own run-time-support library files with the interrupt flexibility option.

Note: Special Cases With the −mi Option

The −mi0 option generates the same code to disable interrupts around software-pipelined loops as when the −mi option is not used.

The −mi option (the threshold value is omitted) means that no code is added to disable interrupts around software pipelined loops. Also, loop performance does not degrade because the compiler is not trying to make the loop interruptible by ensuring that there is at least one cycle in the loop kernel that is not in the delay slot of a branch instruction.

2.12 Linking C6400 Code With C6200/C6700/Older C6400 Object Code

In order to facilitate certain packed-data optimizations, the alignment of top-level arrays for the C6400 family was changed from 4 bytes to 8 bytes. (For C6200 and C6700 code, the alignment for top-level arrays is always 4 bytes.)

If you are linking C6400 with C6200/6700 code or older C6400 code, you may need to take steps to ensure compatibility. The following lists the potential alignment conflicts and possible solutions.

Potential alignment conflicts occur when:

- Linking new C6400 code with any C6400 code already compiled with the 4.0 tools.
- Linking new C6400 code with code already compiled with any version of the tools for the C6200 or C6700 family.

Solutions (pick one):

- Recompile the entire application with the −mv6400 switch. This solution, if possible, is recommended because it can lead to better performance.
- Compile the new code with the −mb option. The −mb switch changes the alignment of top-level arrays to 4 bytes when the −mv6400 switch is used.

2.13 Using Interlist

The compiler tools include a feature that interlists C/C++ source statements into the assembly language output of the compiler. The interlist feature enables you to inspect the assembly code generated for each C statement. The interlist behaves differently, depending on whether or not the optimizer is used, and depending on which options you specify.

The easiest way to invoke the interlist feature is to use the −ss option. To compile and run the interlist on a program called function.c, enter:

```
cl6x -ss function
```

The −ss option prevents the compiler from deleting the interlisted assembly language output file. The output assembly file, function.asm, is assembled normally.

When you invoke the interlist feature without the optimizer, the interlist runs as a separate pass between the code generator and the assembler. It reads both the assembly and C/C++ source files, merges them, and writes the C/C++ statements into the assembly file as comments.

Using the −ss option can cause performance and/or code size degradation.

Example 2−3 shows a typical interlisted assembly file.

Example 2−3. An Interlisted Assembly Language File

```
_main:
           STW     .D2     B3,*SP--(12)
           STW     .D2     A10,*+SP(8)
;-----------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;-----------------------------------------------------------------------------
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
;-----------------------------------------------------------------------------
;   6 | return 0;
;-----------------------------------------------------------------------------
           ZERO    .L1     A10
           MV      .L1     A10,A4
           LDW     .D2     *+SP(8),A10
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

For more information about using the interlist feature with the optimizer, see section 3.11, *Using the Interlist feature With the Optimizer*, on page 3-30.

Chapter 3

Optimizing Your Code

The compiler tools can perform many optimizations that improve the execution speed and reduce the size of C and C++ programs by performing tasks such as simplifying loops, software pipelining, rearranging statements and expressions, and allocating variables into registers.

This chapter describes how to invoke different levels of optimization and describes which optimizations are performed at each level. This chapter also describes how you can use the Interlist feature when performing optimization and how you can profile or debug optimized code.

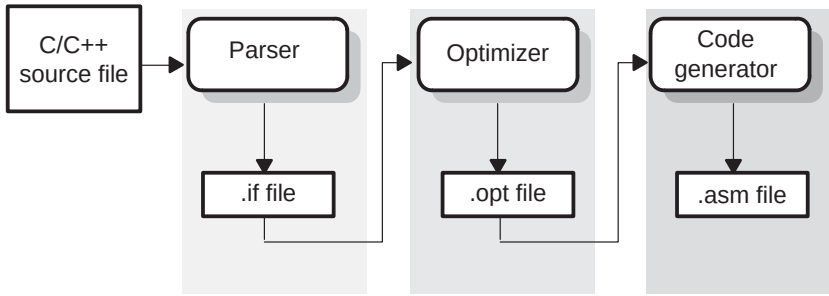
Topic **Page**

3.1 Invoking Optimization

The C/C++ compiler is able to perform various optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator. High-level optimizations must be used to achieve optimal code.

Figure 3−1 illustrates the execution flow of the compiler with the optimizer and code generator.

Figure 3−1. Compiling a C/C++ Program With Optimization



The easiest way to invoke optimization is to use the cl6x compiler program, specifying the −O*n* option on the cl6x command line. The *n* denotes the level of optimization (0, 1, 2, and 3), which controls the type and degree of optimization.

❑ **−O0**

- Performs control-flow-graph simplification
- Allocates variables to registers
- Performs loop rotation
- Eliminates unused code
- Simplifies expressions and statements
- Expands calls to functions declared inline

❑ **−O1**

Performs all −O0 optimizations, plus:

- Performs local copy/constant propagation
- Removes unused assignments
- Eliminates local common expressions

❑ **−O2**

Performs all −O1 optimizations, plus:

- Performs software pipelining (see section 3.2 on page 3-4)
- Performs loop optimizations
- Eliminates global common subexpressions
- Eliminates global unused assignments
- Converts array references in loops to incremented pointer form
- Performs loop unrolling

The optimizer uses −O2 as the default if you use −O without an optimization level.

❑ **−O3**

Performs all −O2 optimizations, plus:

- Removes all functions that are never called
- Simplifies functions with return values that are never used
- Inlines calls to small functions
- Reorders function declarations so that the attributes of called functions are known when the caller is optimized
- Propagates arguments into function bodies when all calls pass the same value in the same argument position
- Identifies file-level variable characteristics

If you use −O3, see section 3.5, *Performing File-Level Optimization (−O3 Option*, on page 3-18 for more information.

The levels of optimizations described above are performed by the stand-alone optimization pass. The code generator performs several additional optimizations, particularly processor-specific optimizations. It does so regardless of whether you invoke the optimizer. These optimizations are always enabled, although they are more effective when the optimizer is used.

Note: Do Not Lower the Optimization Level to Control Code Size

When trying to reduce code size, do not lower the level of optimization. Instead, use the −ms option to control the code size/performance tradeoff. Higher −O levels combined with high −ms levels generally result in the smallest code size. For more information, see section 3.4, *Reducing Code Size (−ms Option)*, on page 3-17.

Note: The −O*n* Option Applies to the Assembly Optimizer

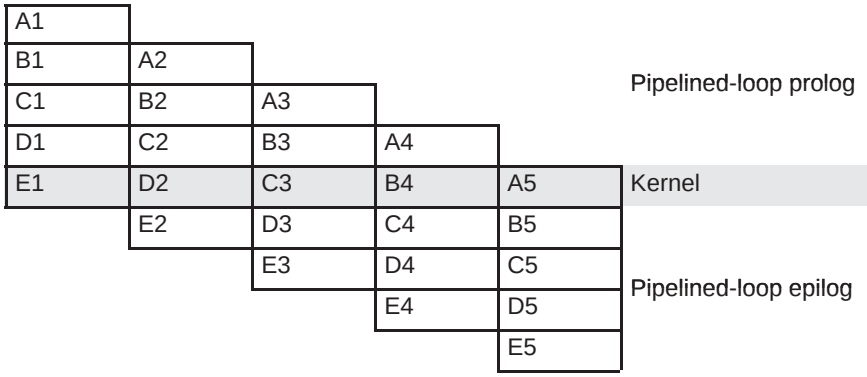
The −O*n* option should also be used with the assembly optimizer. Although the assembly optimizer does not perform all the optimizations described here, key optimizations such as software pipelining and loop unrolling require the −O option to be specified.

3.2 Optimizing Software Pipelining

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. The compiler always attempts to software pipeline. In general, code size and performance are better when you use the −O2 or −O3 option. (See section 3.1, *Invoking Optimization*.) You should also use the −ms option to reduce code size.

Figure 3−2 illustrates a software pipelined loop. The stages of the loop are represented by A, B, C, D, and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the loop *kernel*. In the loop kernel, all five stages execute in parallel. The area above the kernel is known as the *pipelined loop prolog*, and the area below the kernel is known as the *pipelined loop epilog*.

Figure 3−2. Software-Pipelined Loop



The assembly optimizer also software pipelines loops. For more information about the assembly optimizer, see Chapter 4. For more information about software-pipelining, see the *TMS320C6000 Programmer's Guide*.

3.2.1 Turn Off Software Pipelining (−mu Option)

At optimization levels −O2 and −O3, the compiler attempts to software pipeline your loops. You might not want your loops to be software-pipelined for debugging reasons. Software-pipelined loops are sometimes difficult to debug because the code is not presented serially. The −mu option affects both compiled C/C++ code and assembly optimized code.

Note: Software Pipelining Can Significantly Increase Code Size

To reduce code size, use the −ms2 or −ms3 option on non-performance critical code, rather than the −mu option. These code size options not only disable software pipelining, they enable code size reduction optimizations.

3.2.2 Software Pipelining Information

The compiler embeds software pipelined loop information in the .asm file. This information is used to optimize C/C++ code or linear assembly code.

The software pipelining information appears as a comment in the .asm file before a loop and for the assembly optimizer the information is displayed as the tool is running. Example 3−1 illustrates the information that is generated for each loop.

The −mw option adds additional information displaying the register usage at each cycle of the loop kernel and displays the instruction ordering of a single iteration of the software pipelined loop.

Note: More Details on Software Pipelining Information

Refer to Chapter 2 or Appendix A of the *TMS320C6000 Programmer's Guide* for details on all the information and messages that can appear in the Software Pipelining Information comment block before each loop.

Example 3−1. Software Pipelining Information

```
;*----------------------------------------------------------------------------*

;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count          : 2
;*      Known Maximum Trip Count          : 2
;*      Known Max Trip Count Factor       : 2
;*      Loop Carried Dependency Bound(^)  : 4
;*      Unpartitioned Resource Bound      : 4
;*      Partitioned Resource Bound(*)     : 5
;*      Resource Partition:
;*        A-side   B-side
;*      .L units         2        3
;*      .S units         4        4
;*      .D units         1        0
;*      .M units         0        0
;*      .X cross paths   1        3
;*      .T address paths 1        0
;*      Long read paths  0        0
;*      Long write paths 0        0
;*      Logical  ops (.LS)           0        1     (.L or .S unit)
;*      Addition ops (.LSD)          6        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS) 3        4
;*      Bound(.L .S .D .LS .LSD)     5*       4
;*

;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 2 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*----------------------------------------------------------------------------*
```

The terms defined below appear in the software pipelining information. For more information on each term, see the *TMS320C6000 Programmer's Guide*.

- **Loop unroll factor.** The number of times the loop was unrolled specifically to increase performance based on the resource bound constraint in a software pipelined loop.
- **Known minimum trip count.** The minimum number of times the loop will be executed.
- **Known maximum trip count.** The maximum number of times the loop will be executed.
- **Known max trip count factor.** Factor that would always evenly divide the loops trip count. This information can be used to possibly unroll the loop.
- **Loop label.** The label you specified for the loop in the linear assembly input file. This field is not present for C/C++ code.
- **Loop carried dependency bound.** The distance of the largest loop carry path. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration. Instructions that are part of the loop carry bound are marked with the ^ symbol.
- **Iteration interval** (ii). The number of cycles between the initiation of successive iterations of the loop. The smaller the iteration interval, the fewer cycles it takes to execute a loop.
- **Resource bound.** The most used resource constrains the minimum iteration interval. For example, if four instructions require a .D unit, they require at least two cycles to execute (4 instructions/2 parallel .D units).
- **Unpartitioned resource bound.** The best possible resource bound values before the instructions in the loop are partitioned to a particular side.
- **Partitioned resource bound (*).** The resource bound values after the instructions are partitioned.
- **Resource partition.** This table summarizes how the instructions have been partitioned. This information can be used to help assign functional units when writing linear assembly. Each table entry has values for the A-side and B-side registers. An asterisk is used to mark those entries that determine the resource bound value. The table entries represent the following terms:
  - **.L units** is the total number of instructions that require .L units.

- **.S units** is the total number of instructions that require .S units.
- **.D units** is the total number of instructions that require .D units.
- **.M units** is the total number of instructions that require .M units.
- **.X cross paths** is the total number of .X cross paths.
- **.T address paths** is the total number of address paths.
- **Long read path** is the total number of long read port paths.
- **Long write path** is the total number of long write port paths.
- **Logical ops (.LS)** is the total number of instructions that can use either the .L or .S unit.
- **Addition ops (.LSD)** is the total number of instructions that can use either the .L or .S or .D unit

❑ **Bound(.L .S .LS)** is the resource bound value as determined by the number of instructions that use the .L and .S units. It is calculated with the following formula:

Bound(.L .S .LS) = ceil((.L + .S + .LS) / 2)

❑ **Bound(.L .S .D .LS .LSD)** is the resource bound value as determined by the number of instructions that use the .D, .L and .S unit. It is calculated with the following formula:

Bound(.L .S .D .LS .SLED) = ceil((.L + .S + .D + .LS + .LSD) / 3)

❑ **Minimum required memory pad.** The number of bytes that are read if speculative execution is enabled. See section 3.2.3, *Collapsing Prologs and Epilogs for Improved Performance and Code Size*, on page 3-14, for more information.

3.2.2.1 Loop Disqualified for Software Pipelining Messages

The following messages appear if the loop is completely disqualified for software pipelining:

- **Bad loop structure**. This error is very rare and can stem from the following:
  - An asm statement inserted in the C code inner loop
  - Parallel instructions being used as input to the Linear Assembly Optimizer
  - Complex control flow such as GOTO statements and breaks
- **Loop contains a call**. Sometimes the compiler may not be able to inline a function call that is in a loop. Because the compiler could not inline the function call, the loop could not be software pipelined.
- **Too many instructions**. There are too many instructions in the loop to software pipeline.
- **Software pipelining disabled**. Software pipelining has been disabled by a command-line option. Pipelining is turned off when using the −mu option, not using the −O2 or −O3 option, or using the −ms2 or −ms3 option.
- **Uninitialized trip counter**. The trip counter may not have been set to an initial value.
- **Suppressed to prevent code expansion**. Software pipelining may be suppressed because of the −ms1 option. When the −ms1 option is used, software pipelining is disabled in less promising cases to reduce code size. To enable pipelining, use −ms0 or omit the −ms option altogether.
- **Loop carried dependency bound too large**. If the loop has complex loop control, try −mh according to the recommendations in section 3.2.3.2, *Selecting the Best Threshold Value*, on page 3-15.
- **Cannot identify trip counter**. The loop trip counter could not be identified or was used incorrectly in the loop body.

3.2.2.2 Pipeline Failure Messages

The following messages can appear when the compiler or assembly optimizer is processing a software pipeline and it fails:

- **Address increment is too large**. An address register's offset must be adjusted because the offset is out of range of the C6000's offset addressing mode. You must minimize address register offsets.
- **Cannot allocate machine registers**. A software pipeline schedule was found, but it cannot allocate machine registers for the schedule. You must simplify the loop.

  The register usage for the schedule found at the given ii is displayed. This information can be used when writing linear assembly to balance register pressure on both sides of the register file. For example:

```
ii = 11 Cannot allocate machine registers
Regs Live Always : 3/0  (A/B-side)
Max Regs Live : 20/14
Max Condo Regs Live : 2/1
```

  - **Regs Live Always.** The number of values that must be assigned a register for the duration of the whole loop body. This means that these values must always be allocated registers for any given schedule found for the loop.
  - **Max Regs Live.** Maximum number of values live at any given cycle in the loop that must be allocated to a register. This indicates the maximum number of registers required by the schedule found.
  - **Max Cond Regs Live.** Maximum number of registers live at any given cycle in the loop kernel that must be allocated to a condition register.

- **Cycle count too high − Not profitable**. With the schedule that the compiler found for the loop, it is more efficient to use a non-software-pipelined version.
- **Did not find schedule**. The compiler was unable to find a schedule for the software pipeline at the given ii (iteration interval). You should simplify the loop and/or eliminate loop carried dependencies.
- **Iterations in parallel > minimum or maximum trip count**. A software pipeline schedule was found, but the schedule has more iterations in parallel than the minimum or maximum loop trip count. You must enable redundant loops or communicate the trip information.
- **Speculative threshold exceeded**. It would be necessary to speculatively load beyond the threshold currently specified by the −mh option. You must increase the −mh threshold as recommended in the software-pipeline feedback located in the assembly file.

- **Register is live too long**. A register must have a value that exists (is live) for more than ii cycles. You may insert MV instructions to split register lifetimes that are too long.

  If the assembly optimizer is being used, the .sa file line numbers of the instructions that define and use the registers that are live too long are listed after this failure message.

  ```
  ii = 9  Register is live too long
          |10| −> |17|
  ```

  This means that the instruction that defines the register value is on line 10 and the instruction that uses the register value is on line 17 in the sa file.

- **Too many predicates live on one side**. The ’C6000 has predicate, or conditional, registers available for use with conditional instructions. There are 5 predicate registers on the ’C6200 and ’C6700, and 6 predicate registers on the ’C6400. There are two or three on the A side and three on the B side. Sometimes the particular partition and schedule combination requires more than these available registers.

- **Schedule found with N iterations in parallel**. A software pipeline schedule was found with N iterations executing in parallel.

- **Too many reads of one register**. The same register can be read a maximum of 4 times per cycle with the ’C6200 or ’C6700 core. The ’C6400 core can read the same register any number of times per cycle.

- **Trip variable used in loop − Can’t adjust trip count**. The loop trip counter has a use in the loop other than as a loop trip counter.

3.2.2.3 Investigative Feedback

The following messages can appear when the compiler or assembly optimizer detects that performance can be improved with the software pipeline:

- **Loop carried dependency bound is much larger than unpartitioned resource bound**. There may be a potential memory alias disambiguation problem. This means that there are two pointers that may or may not point to the same location, and thus, the compiler must assume they might. This can cause a dependency (often between the load of one pointer and the store of another) that does not really exist. For software pipelined loops, this can greatly degrade performance.
- **Two loops are generated, one not software pipelined**. If the trip count is too low, it is illegal to execute the software pipelined version of the loop. In this case, the compiler could not guarantee that the minimum trip count would be high enough to always safely execute the pipelined version. Hence, it generated a non-pipelined version as well. Code is generated, so that at run time, the appropriate version of the loop will be executed.
- **Uneven resources**. If the number of resources to do a particular operation is odd, unrolling the loop can be beneficial. If a loop requires 3 multiplies, then a minimum iteration interval of 2 cycles is required to execute this. If the loop was unrolled, 6 multiplies could be evenly partitioned across the A and B side, having a minimum ii of 3 cycles, giving improved performance.
- **Larger outer loop overhead in nested loop**. In cases where the inner loop count of a nested loop is relatively small, the time to execute the outer loop can start to become a large percentage of the total execution time. For cases where this significantly degrades overall loop performance, unrolling the inner loop may be desired.
- **There are memory bank conflicts**. In cases where the compiler generates two memory accesses in one cycle and those accesses are either 8 bytes apart on a 'C620x device, 16 bytes apart on a 'C670x device, or 32 bytes apart on a 'C640x device, *and* both accesses reside within the same memory block, a memory bank stall will occur. Memory bank conflicts can be completely avoided by either placing the two accesses in different memory blocks or by writing linear assembly and using the .mptr directive to control memory banks.
- **T address paths are resource bound**. T address paths defined the number of memory accesses that must be sent out on the address bus each loop iteration. If these are the resource bound for the loop, it is often possible to reduce the number of accesses by performing word accesses (LDW/STW) for any short accesses being performed.

3.2.2.4 Register Usage Table Generated by the −mw Option

The −mw option places additional software pipeline feedback in the generated assembly file. This information includes a single scheduled iteration view of the software pipelined loop.

If software pipelining succeeds for a given loop, and the −mw option was used during the compilation process, a register usage table is added to the software pipelining information comment block in the generated assembly code.

The numbers on each row represent the cycle number within the loop kernel.

Each column represents one register on the TMS320C6000 (A0−A15, B0−B15 for all C6000 processors and A16−A31, B16−B31 for the C6400). The registers are labeled in the first three rows of the register usage table and should be read columnwise.

An * in a table entry indicates that the register indicated by the column header is live on the kernel execute packet indicated by the cycle number labeling each row.

An example of the register usage table follows:

```
;*       Searching for software pipeline schedule at
;*         ii = 15 Schedule found with 2 iterations in parallel
;*
;*       Register Usage Table:
;*           +---------------------------------+
;*           |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*           |0000000000111111|0000000000111111|
;*           |0123456789012345|0123456789012345|
;*           |----------------+----------------|
;*         0:|***   ****      |*** ******      |
;*         1:|****  ****      |*** ******      |
;*         2:|****  ****      |*** ******      |
;*         3:| **  *****      |*** ******      |
;*         4:| **  *****      |*** ******      |
;*         5:| **  *****      |*** ******      |
;*         6:| **  *****      |**********      |
;*         7:|***  *****      |** *******      |
;*         8:|**** *****      |***********     |
;*         9:|**********      |** ********     |
;*        10:|***********     |** *********    |
;*        11:|***********     |** *********    |
;*        12:|**********      |***********     |
;*        13:|**** *****      |** ******* *    |
;*        14:|***  *****      |*** ****** *    |
;*           +---------------------------------+
;*
```

This example shows that on cycle 0 (first execute packet) of the loop kernel registers A0, A1, A2, A6, A7, A8, A9, B0, B1, B2, B4, B5, B6, B7, B8, and B9 are all live during this cycle.

3.2.3 Collapsing Prologs and Epilogs for Improved Performance and Code Size

When a loop is software pipelined, a prolog and epilog are generally required. The prolog is used to pipe up the loop and epilog is used to pipe down the loop.

In general, a loop must execute a minimum number of iterations before the software-pipelined version can be safely executed. If the minimum known trip count is too small, either a redundant loop is added or software pipelining is disabled. Collapsing the prolog and epilog of a loop can reduce the minimum trip count necessary to safely execute the pipelined loop.

Collapsing can also substantially reduce code size. Some of this code size growth is due to the redundant loop. The remainder is due to the prolog and epilog.

The prolog and epilog of a software-pipelined loop consists of up to *p*−1 stages of length *ii*, where *p* is the number of iterations that are executed in parallel during the steady state and *ii* is the cycle time for the pipelined loop body. During prolog and epilog collapsing the compiler tries to collapse as many stages as possible. However, overcollapsing can have a negative performance impact. Thus, by default, the compiler attempts to collapse as many stages as possible without sacrificing performance. When −ms0/−ms1 options are invoked, the compiler increasingly favors code size over performance.

3.2.3.1 Speculative Execution

When prologs and epilogs are collapsed, instructions might be speculatively executed, thereby causing loads to addresses beyond either end of the range explicitly read within the loop. By default, the compiler cannot speculate loads because this could cause an illegal memory location to be read. Sometimes, the compiler can predicate these loads to prevent over execution. However, this can increase register pressure and might decrease the total amount collapsing which can be performed.

When the −−speculate_loads*n* option is used, the speculative threshold is increased from the default of 0 to *n*. When the threshold is *n*, the compiler can allow a load to be speculatively executed as the memory location it reads will be no more than *n* bytes before or after some location explicitly read within the loop. If the *n* is omitted, the compiler assumes the speculative threshold is unlimited. To specify this in Code Composer Studio, select the Speculate Threshold check box and leave the text box blank in the Build Options dialog box on the Compiler tab, Advanced category.

Collapsing can usually reduce the minimum safe trip count. If the minimum known trip count is less than the minimum safe trip count, a redundant loop is required. Otherwise, pipelining must be suppressed. Both these values can be found in the comment block preceding a software pipelined loop.

```
;*     Known Minimum Trip Count : 1
....
;*     Minimum safe trip count  : 7
```

If the minimum safe trip count is greater than the minimum known trip count, use of −mh is highly recommended, not only for code size, but for performance.

When using −mh, you must ensure that potentially speculated loads will not cause illegal reads. This can be done by padding the data sections and/or stack, as needed, by the required memory pad in both directions. The required memory pad for a given software-pipelined loop is also provided in the comment block for that loop.

```
;*     Minimum required memory pad : 8 bytes
```

3.2.3.2 Selecting the Best Threshold Value

When a loop is software pipelined, the comment block preceding the loop provides the following information:

- Required memory pad for this loop
- The minimum value of *n* needed to achieve this software pipeline schedule and level of collapsing
- Suggestion for a larger value of *n* to use which might allow additional collapsing

This information shows up in the comment block as follows:

```
;* Minimum required memory pad : 5 bytes
;* Minimum threshold value     : -mh7
;*
;* For further improvement on this loop, try option -mh14
```

For safety, the example loop requires that array data referenced within this loop be preceded and followed by a pad of at least 5 bytes. This pad can consist of other program data. The pad will not be modified. In many cases, the threshold value (namely, the minimum value of the argument to −mh that is needed to achieve a particular schedule and level of collapsing) is the same as the pad. However, when it is not, the comment block will also include the minimum threshold value. In the case of this loop, the threshold value must be at least 7 to achieve this level of collapsing.

Another interesting question that arises is whether there is a larger threshold value which would facilitate additional collapsing? If there is, this information will also be provided. For example, in the above comment block, a threshold value of 14 might facilitate further collapsing.

3.3 Redundant Loops

Every loop iterates some number of times before the loop terminates. The number of iterations is called the *trip count*. The variable used to count each iteration is the *trip counter*. When the trip counter reaches a limit equal to the trip count, the loop terminates. The C6000 tools use the trip count to determine whether or not a loop can be pipelined. The structure of a software pipelined loop requires the execution of a minimum number of loop iterations (a minimum trip count) in order to fill or prime the pipeline.

The minimum trip count for a software pipelined loop is determined by the number of iterations executing in parallel. In Figure 3−2 on page 3-4, the minimum trip count is five. In the following example A, B, and C are instructions in a software pipeline, so the minimum trip count for this single-cycle software pipelined loop is three:

```
A
B  A
C  B  A     ← Three iterations in parallel = minimum trip count
   C  B
      C
```

When the C6000 tools cannot determine the trip count for a loop, then by default two loops and control logic are generated. The first loop is not pipelined, and it executes if the run-time trip count is less than the loop's minimum trip count. The second loop is the software pipelined loop, and it executes when the run-time trip count is greater than or equal to the minimum trip count. At any given time, one of the loops is a *redundant loop*. :

```
foo(N)  /* N is the trip count */
{
   for (i=0; i < N; i++) /* i is the trip counter */
}
```

After finding a software pipeline for the loop, the compiler transforms foo() as below, assuming the minimum trip count for the loop is 3. Two versions of the loop would be generated and the following comparison would be used to determine which version should be executed:

```
foo(N)
{
   if (N < 3)
   {
      for (i=0; i < N; i++)   /* Unpipelined version */
   }
   else
   }
      for (i=0; i < N; i++)   /* Pipelined version   */
   }
}
foo(50); /* Execute software pipelined loop          */
foo(2);  /* Execute loop (unpipelined)*/
```

You may be able to help the compiler avoid producing redundant loops with the use of −pm −O3 (see section 3.6 on page 3-20) or the use of the MUST_ITERATE pragma (see section 7.7.14 on page 7-28).

Note: Turning Off Redundant Loops

Specifying any −ms option turns off redundant loops.

3.4 Reducing Code Size (−ms Option)

When using the −O or −O*n* option, you are telling the compiler to optimize your code. The higher the value of *n*, the more effort the compiler invests in optimizing your code. However, you might still need to tell the compiler what your optimization priorities are.

By default, when −O2 or −O3 is specified, the compiler optimizes primarily for performance. (Under lower optimization levels, the priorities are compilation time and debugging ease.) You can adjust the priorities between performance and code size by using the code size flag −ms*n*. The −ms0, −ms1, −ms2 and −ms3 options increasingly favor code size over performance.

It is recommended that a code size flag not be used with the most performance-critical code. Using −ms0 or −ms1 is recommended for all but the most performance-critical code. Using −ms2 or −ms3 is recommended for seldom-executed code. Either −ms2 or −ms3 should be used also if you need the minimum code size. In all cases, it is generally recommended that the code size flags be combined with −O2 or −O3.

Note: Disabling Code-Size Optimizations or Reducing the Optimization Level

If you reduce optimization and/or do not use code size flags, you are disabling code-size optimizations and sacrificing performance.

Note: The −ms Option is Equivalent to −ms0

If you use −ms with no code size level number specified, the option level defaults to −ms0.

3.5 Performing File-Level Optimization (−O3 Option)

The −O3 option instructs the compiler to perform file-level optimization. You can use the −O3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimizations. The options listed in Table 3−1 work with −O3 to perform the indicated optimization:

Table 3−1. Options That You Can Use With −O3

If you ...	Use this option	Page
Have files that redeclare standard library functions	−ol*n*	3-18
Want to create an optimization information file	−on*n*	3-19
Want to compile multiple source files	−pm	3-20

Note: Do Not Lower the Optimization Level to Control Code Size

When trying to reduce code size, do not lower the level of optimization. In fact, you might see an increase in code size when lowering the level of optimization. Instead, use the −ms option to control the code size/performance tradeoff. For more information, see section 3.4, *Reducing Code Size (−ms Option)*, on page 3-17.

3.5.1 Controlling File-Level Optimization (−ol*n* Option)

When you invoke the compiler with the −O3 option, some of the optimizations use known properties of the standard library functions. If your file redeclares any of these standard library functions, these optimizations become ineffective. The −ol (lowercase L) option controls file-level optimizations. The number following the −ol denotes the level (0, 1, or 2). Use Table 3−2 to select the appropriate level to append to the −ol option.

Table 3−2. Selecting a Level for the −ol Option

If your source file...	Use this option
Declares a function with the same name as a standard library function	−ol0
Contains but does not alter functions declared in the standard library	−ol1
Does not alter standard library functions, but you used the −ol0 or −ol1 option in a command file or an environment variable. The −ol2 option restores the default behavior of the optimizer.	−ol2

3.5.2 Creating an Optimization Information File (−on*n* Option)

When you invoke the compiler with the −O3 option, you can use the −on option to create an optimization information file that you can read. The number following the −on denotes the level (0, 1, or 2). The resulting file has an .nfo extension. Use Table 3−3 to select the appropriate level to append to the −on option.

Table 3−3. Selecting a Level for the −on Option

If you...	Use this option
Do not want to produce an information file, but you used the −on1 or −on2 option in a command file or an environment variable. The −on0 option restores the default behavior of the optimizer.	−on0
Want to produce an optimization information file	−on1
Want to produce a verbose optimization information file	−on2

3.6 Performing Program-Level Optimization (−pm and −O3 Options)

You can specify program-level optimization by using the −pm option with the −O3 option. With program-level optimization, all of your source files are compiled into one intermediate file called a *module*. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

- If a particular argument in a function always has the same value, the compiler replaces the argument with the value and passes the value instead of the argument.
- If a return value of a function is never used, the compiler deletes the return code in the function.
- If a function is not called directly or indirectly by main, the compiler removes the function.

To see which program-level optimizations the compiler is applying, use the −on2 option to generate an information file. See section 3.5.2, *Creating an Optimization Information File (−on*n *Option)*, on page 3-19 for more information.

In Code Composer Studio, when the −pm option is used, C and C++ files that have the same options are compiled together. However, if any file has a file-specific option that is not selected as a project-wide option, that file is compiled separately. For example, if every C and C++ file in your project has a different set of file-specific options, each is compiled separately, even though program-level optimization has been specified. To compile all C and C++ files together, make sure the files do not have file-specific options. Be aware that compiling C and C++ files together may not be safe if previously you used a file-specific option such as −ma.

Note: Compiling Files With the −pm and −k Options

If you compile all files with the −pm and −k options, the compiler produces only one .asm file, not one for each corresponding source file.

3.6.1 Controlling Program-Level Optimization (−op*n* Option)

You can control program-level optimization, which you invoke with −pm −O3, by using the −op option. Specifically, the −op option indicates if functions in other modules can call a module's external functions or modify a module's external variables. The number following −op indicates the level you set for the module that you are allowing to be called or modified. The −O3 option combines this information with its own file-level analysis to decide whether to treat this module's external function and variable declarations as if they had been declared static. Use Table 3−4 to select the appropriate level to append to the −op option.

Table 3−4. Selecting a Level for the −op Option

If your module …	**Use this option**
Has functions that are called from other modules and global variables that are modified in other modules	−op0
Does not have functions that are called by other modules but has global variables that are modified in other modules	−op1
Does not have functions that are called by other modules or global variables that are modified in other modules	−op2
Has functions that are called from other modules but does not have global variables that are modified in other modules	−op3

In certain circumstances, the compiler reverts to a different −op level from the one you specified, or it might disable program-level optimization altogether. Table 3−5 lists the combinations of −op levels and conditions that cause the compiler to revert to other −op levels.

Table 3−5. Special Considerations When Using the −op Option

If your −op is...	Under these conditions...	Then the −op level...
Not specified	The −O3 optimization level was specified	Defaults to −op2
Not specified	The compiler sees calls to outside functions under the −O3 optimization level	Reverts to −op0
Not specified	Main is not defined	Reverts to −op0
−op1 or −op2	No function has main defined as an entry point and functions are not identified by the FUNC_EXT_CALLED pragma	Reverts to −op0
−op1 or −op2	No interrupt function is defined	Reverts to −op0
−op1 or −op2	Functions are identified by the FUNC_EXT_CALLED pragma	Remains −op1 or −op2
−op3	Any condition	Remains −op3

In some situations when you use −pm and −O3, you *must* use an −op option or the FUNC_EXT_CALLED pragma. See section 3.6.2, *Optimization Considerations When Mixing C and Assembly*, on page 3-22 for information about these situations.

3.6.2 Optimization Considerations When Mixing C/C++ and Assembly

If you have any assembly functions in your program, you need to exercise caution when using the −pm option. The compiler recognizes only the C/C++ source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C/C++ functions, the −pm option optimizes out those C/C++ functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see section 7.7.6, *The FUNC_EXT_CALLED Pragma*, on page 7-23) before any declaration or reference to a function that you want to keep.

Another approach you can take when you use assembly functions in your program is to use the −op*n* option with the −pm and −O3 options (see section 3.6.1, *Controlling Program-Level Optimization*, on page 3-21).

In general, you achieve the best results through judicious use of the FUNC_EXT_CALLED pragma in combination with −pm −O3 and −op1 or −op2.

If any of the following situations apply to your application, use the suggested solution:

Situation Your application consists of C/C++ source code that calls assembly functions. Those assembly functions do not call any C/C++ functions or modify any C/C++ variables.

Solution Compile with −pm −O3 −op2 to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables. See section 3.6.1 for information about the −op2 option.

If you compile with the −pm −O3 options only, the compiler reverts from the default optimization level (−op2) to −op0. The compiler uses −op0, because it presumes that the calls to the assembly language functions that have a definition in C/C++ may call other C/C++ functions or modify C/C++ variables.

Situation Your application consists of C/C++ source code that calls assembly functions. The assembly language functions do not call C/C++ functions, but they modify C/C++ variables.

Solution Try both of these solutions and choose the one that works best with your code:

- Compile with −pm −O3 −op1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with −pm −O3 −op2.

See section 3.6.1 on page 3-21 for information about the −op*n* option.

Situation Your application consists of C/C++ source code and assembly source code. The assembly functions are interrupt service routines that call C/C++ functions; the C/C++ functions that the assembly functions call are never called from C/C++. These C/C++ functions act like main: they function as entry points into C/C++.

Solution Add the volatile keyword to the C/C++ variables that may be modified by the interrupts. Then, you can optimize your code in one of these ways:

- You achieve the best optimization by applying the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compiling with −pm −O3 −op2. *Be sure that you use the pragma with all of the entry-point functions.* If you do not, the compiler might remove the entry-point functions that are not preceded by the FUNC_EXT_CALL pragma.
- Compile with −pm −O3 −op3. Because you do not use the FUNC_EXT_CALL pragma, you must use the −op3 option, which is less aggressive than the −op2 option, and your optimization may not be as effective.

Keep in mind that if you use −pm −O3 without additional options, the compiler removes the C functions that the assembly functions call. Use the FUNC_EXT_CALLED pragma to keep these functions.

3.7 Indicating Whether Certain Aliasing Techniques Are Used

Aliasing occurs when you can access a single object in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization, because any indirect reference can refer to another object. The compiler analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while preserving the correctness of the program. The compiler behaves conservatively.

The following sections describe some aliasing techniques that may be used in your code. These techniques are valid according to the ISO C standard and are accepted by the C6000 compiler; however, they prevent the optimizer from fully optimizing your code.

3.7.1 Use the −ma Option When Certain Aliases are Used

The compiler, when invoked with optimization, assumes that any variable whose address is passed as an argument to a function is not subsequently modified by an alias set up in the called function. Examples include:

- ❏ Returning the address from a function
- ❏ Assigning the address to a global variable

If you use aliases like this in your code, you must use the −ma option when you are optimizing your code. For example, if your code is similar to this, use the −ma option:

```
int *glob_ptr;

g()
{
    int  x = 1;
    int *p = f(&x);

    *p        = 5;    /* p aliases x    */
    *glob_ptr = 10;   /* glob_ptr aliases x */

    h(x);
}

int *f(int *arg)
{
    glob_ptr = arg;
    return arg;
}
```

3.7.2 Use the −mt Option to Indicate That These Techniques Are *Not* Used

The −mt option informs the compiler that it can make certain assumptions about how aliases are used in your code. These assumptions allow the compiler to improve optimization. The −mt option also specifies that loop-invariant counter increments and decrements are non-zero. Loop invariant means the value of an expression does not change within the loop.

- The −mt option indicates that your code does not use the aliasing technique described in section 3.7.1. If your code uses that technique, do *not* use the −mt option; however, you must compile with the −ma option.

  Do *not* use the −ma option with the −mt option. If you do, the −mt option overrides the −ma option.

- The −mt option indicates that a pointer to a character type does *not* alias (point to) an object of another type. That is, the special exception to the general aliasing rule for these types given in section 3.3 of the ISO specification is ignored. If you have code similar to the following example, do *not* use the −mt option:

```
{
    long  l;
    char *p = (char *) &l;

    p[2] = 5;
}
```

- The −mt option indicates that indirect references on two pointers, P and Q, are not aliases if P and Q are distinct parameters of the same function activated by the same call at run time. If you have code similar to the following example, do *not* use the −mt option:

```
g(int j)
{
    int a[20];

    f(&a, &a)          /* Bad */
    f(&a+42, &a+j)     /* Also Bad */
}

f(int *ptr1, int *ptr2)
{
    ...
}
```

- The −mt option indicates that each subscript expression in an array reference A[E1]..[En] evaluates to a nonnegative value that is less than the corresponding declared array bound. Do *not* use −mt if you have code similar to the following example:

```
static int ary[20][20];

int g()
{
    return f(5, –4); /* –4 is a negative index */
    return f(0, 96); /* 96 exceeds 20 as an index */
    return f(4, 16); /* This one is OK */
}

int f(int i, int j)
{
    return ary[i][j];
}
```

  In this example, ary[5][−4], ary[0][96], and ary[4][16] access the same memory location. Only the reference ary[4][16] is acceptable with the −mt option because both of its indices are within the bounds (0..19).

- The −mt option indicates that loop-invariant counter increments and decrements of loop counters are non-zero. Loop invariant means a value of an expression doesn't change within the loop.

If your code does *not* contain any of the aliasing techniques described above, you should use the −mt option to improve the optimization of your code. However, you must use discretion with the −mt option; unexpected results may occur if these aliasing techniques appear in your code and the −mt option is used.

3.7.3 Using the −mt Option With the Assembly Optimizer

The −mt option allows the assembly optimizer to assume there are no memory aliases in your linear assembly, i.e., no memory references ever depend on each other. However, the assembly optimizer still recognizes any memory dependences you point out with the .mdep directive. For more information about the .mdep directive, see page 4-21 and 4-44.

3.8 Prevent Reordering of Associative Floating-Point Operations

The compiler freely reorders associative floating-point operations. If you do not wish to have the compiler reorder associative floating point operations, use the −mc option. Specifying the −mc option may decrease performance.

3.9 Use Caution With asm Statements in Optimized Code

You must be extremely careful when using asm (inline assembly) statements in optimized code. The compiler rearranges code segments, uses registers freely, and can completely remove variables or expressions. Although the compiler never optimizes out an asm statement (except when it is unreachable), the surrounding environment where the assembly code is inserted can differ significantly from the original C/C++ source code.

It is usually safe to use asm statements to manipulate hardware controls such as interrupt masks, but asm statements that attempt to interface with the C/C++ environment or access C/C++ variables can have unexpected results. After compilation, check the assembly output to make sure your asm statements are correct and maintain the integrity of the program.

3.10 Automatic Inline Expansion (−oi Option)

When optimizing with the −O3 option, the compiler automatically inlines small functions. A command-line option, −ois*ize*, specifies the size threshold. Any function larger than the *size* threshold is not automatically inlined. You can use the −ois*ize* option in the following ways:

- If you set the *size* parameter to 0 (−oi0), automatic inline expansion is disabled.
- If you set the *size* parameter to a nonzero integer, the compiler uses this size threshold as a limit to the size of the functions it automatically inlines. The compiler multiplies the number of times the function is inlined (plus 1 if the function is externally visible and its declaration cannot be safely removed) by the size of the function.

The compiler inlines the function only if the result is less than the size parameter. The compiler measures the size of a function in arbitrary units; however, the optimizer information file (created with the −on1 or −on2 option) reports the size of each function in the same units that the −oi option uses.

The −ois*ize* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the −ois*ize* option, the compiler inlines very small functions. .

Note: −O3 Optimization and Inlining

In order to turn on automatic inlining, you must use the −O3 option. The −O3 option turns on other optimizations. If you desire the −O3 optimizations, but not automatic inlining, use −oi0 with the −O3 option.

Note: Inlining and Code Size

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. In order to prevent increases in code size because of inlining, use the −oi0 and −pi options. These options cause the compiler to inline intrinsics only. If your code size still seems too large, see section 3.4, *Reducing Code Size (−ms Option)*, on page 3-17.

3.11 Using the Interlist Feature With Optimization

You control the output of the interlist feature when compiling with optimization (the −O*n* option) with the −os and −ss options.

- The −os option interlists compiler comments with assembly source statements.
- The −ss and −os options together interlist the compiler comments and the original C/C++ source with the assembly code.

When you use the −os option with optimization, the interlist feature does *not* run as a separate pass. Instead, the compiler inserts comments into the code, indicating how the compiler has rearranged and optimized the code. These comments appear in the assembly language file as comments starting with ;**. The C/C++ source code is not interlisted, unless you use the −ss option also.

The interlist feature can affect optimized code because it might prevent some optimization from crossing C/C++ statement boundaries. Optimization makes normal source interlisting impractical, because the compiler extensively rearranges your program. Therefore, when you use the −os option, the compiler writes reconstructed C/C++ statements.

Example 3−2 shows the function from Example 2−3 on page 2-47 compiled with optimization (−O2) and the −os option. The assembly file contains compiler comments interlisted with assembly code.

Note: Impact on Performance and Code Size

The −ss option can have a negative effect on performance and code size.

Example 3−2. The Function From Example 2−3 Compiled With the −O2 and −os Options

```
_main:
;** 5  ----------------------    printf(”Hello, world\n”);
;** 6  ----------------------    return 0;
           STW     .D2     B3,*SP--(12)
   .line  3
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0

           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3

           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
   .line  4
           ZERO    .L1     A4
   .line  5
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
   .endfunc  7,000080400h,12
```

When you use the −ss and −os options with optimization, the compiler inserts its comments and the interlist feature runs before the assembler, merging the original C/C++ source into the assembly file.

Example 3−3 shows the function from Example 2−3 on page 2-47 compiled with the optimization (−O2) and the −ss and −os options. The assembly file contains compiler comments and C source interlisted with assembly code.

Example 3−3. The Function From Example 2−3 Compiled With the −O2, −os, and −ss Options

```
_main:
;** 5  ------------------------    printf(”Hello, world\n”);
;** 6  ------------------------    return 0;
           STW     .D2     B3,*SP--(12)
;-------------------------------------------------------------------------------
;   5 | printf(”Hello, world\n”);
;-------------------------------------------------------------------------------
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
;-------------------------------------------------------------------------------
;   6 | return 0;
;-------------------------------------------------------------------------------
           ZERO    .L1     A4
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

3.12 Debugging and Profiling Optimized Code

Debugging fully optimized code is not recommended, because the compiler's extensive rearrangement of code and the many-to-many allocation of variables to registers often make it difficult to correlate source code with object code. Profiling code that has been built with the −−symdebug:dwarf (or −g) option or the −−symdebug:coff option (STABS debug) is not recommended as well, because these options can significantly degrade performance. To remedy these problems, you can use the options described in the following sections to optimize your code in such a way that you can still debug or profile the code.

3.12.1 Debugging Optimized Code (−−symdebug:dwarf, −−symdebug:coff, and −O Options)

To debug optimized code, use the −O option in conjunction with one of the symbolic debugging options (−−symdebug:dwarf or −−symdebug:coff). The symbolic debugging options generate directives that are used by the C/C++ source-level debugger, but they disable many compiler optimizations. When you use the −O option (which invokes optimization) with the −−symdebug:dwarf or −−symdebug:coff option, you turn on the maximum amount of optimization that is compatible with debugging.

If you are having trouble debugging loops in your code, you can use the −mu option to turn off software pipelining. See section 3.2.1 on page 3-5 for more information.

Note: Symbolic Debugging Options Affect Performance and Code Size

Using the −−symdebug:dwarf or −symdebug:coff option can cause a significant performance and code size degradation of your code. Use these options for debugging only. Using −−symdebug:dwarf or −symdebug:coff when profiling is not recommended.

3.12.2 Profiling Optimized Code

To profile optimized code, use optimization (−O0 through −O3) without any debug option. By default, the compiler generates a minimal amount of debug information without affecting optimizations, code size, or performance.

If you have a breakpoint-based profiler, use the −−profile:breakpt option with the −O option. The −−profile:breakpt option disables optimizations that would cause incorrect behavior when using a breakpoint-based profiler.

If you need to profile code at a finer grain that the function level in Code Composer Studio, you can use the −−symdebug:dwarf or −symdebug:coff option, although this is not recommended. You might see a significant performance degradation because the compiler cannot use all optimizations with −−symdebug:dwarf or −symdebug:coff. It is recommended that outside of Code Composer Studio, you use the clock( ) function.

Note: Profile Points

In Code Composer Studio, when symbolic debugging is not used, profile points can only be set at the beginning and end of functions.

3.13 What Kind of Optimization Is Being Performed?

The TMS320C6000 C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/C++ programs and to reduce their size.

Following are the optimizations performed by the compiler:

Optimization	Page
Cost-based register allocation	3-36
Alias disambiguation	3-38
Branch optimizations and control-flow simplification	3-38
Data flow optimizations ❑ Copy propagation ❑ Common subexpression elimination ❑ Redundant assignment elimination	3-41
Expression simplification	3-41
Inline expansion of functions	3-42
Induction variable optimizations and strength reduction	3-43
Loop-invariant code motion	3-44
Loop rotation	3-44
Register variables	3-44
Register tracking/targeting	3-44
Software pipelining	3-45

3.13.1 Cost-Based Register Allocation

The compiler, when optimization is enabled, allocates registers to user variables and compiler temporary values according to their type, use, and frequency. Variables used within loops are weighted to have priority over others, and those variables whose uses do not overlap can be allocated to the same register.

Induction variable elimination and loop test replacement allow the compiler to recognize the loop as a simple counting loop and software pipeline, unroll, or eliminate the loop. Strength reduction turns the array references into efficient pointer references with autoincrements.

Example 3−4. Strength Reduction, Induction Variable Elimination, Register Variables, and Software Pipelining

(a) C source

```
int a[10];

main()
{
    int i;

    for (i=0; i<10; i++)
        a[i] = 0;
}
```

Example 3−4. Strength Reduction, Induction Variable Elimination, Register Variables and Software Pipelining (Continued)

(b) Compiler output

```
FP  .set   A15
DP  .set   B14
SP  .set   B15

;  opt6x -O2 j3_32.if j3_32.opt
   .sect  ".text"
   .global   _main

_main:
;**--------------------------------------------------------*
           MVK     .S1     _a,A0
           MVKH    .S1     _a,A0

           MV      .L2X    A0,B4
||         ZERO    .L1     A3
||         ZERO    .D2     B5
||         MVK     .S2     2,B0          ; |7|
;**--------------------------------------------------------*
L2:        ; PIPED LOOP PROLOG
   [ B0]   B       .S1     L3            ; |7|
   [ B0]   B       .S1     L3            ;@ |7|
   [ B0]   B       .S1     L3            ;@@ |7|

   [ B0]   B       .S1     L3            ;@@@ |7|
|| [ B0]   SUB     .L2     B0,2,B0       ;@@@@ |7|

           ADD     .S2     8,B4,B4       ; |8|
|| [ B0]   B       .S1     L3            ;@@@@ |7|
|| [ B0]   SUB     .L2     B0,2,B0       ;@@@@@ |7|
;**--------------------------------------------------------*
L3:        ; PIPED LOOP KERNEL
           STW     .D1T1   A3,*A0++(8)   ; |8|
||         STW     .D2T2   B5,*-B4(4)    ; |8|
||         ADD     .S2     8,B4,B4       ;@ |8|
|| [ B0]   B       .S1     L3            ;@@@@@ |7|
|| [ B0]   SUB     .L2     B0,2,B0       ;@@@@@@ |7|
;**--------------------------------------------------------*
L4:        ; PIPED LOOP EPILOG
;**--------------------------------------------------------*
           B       .S2     B3            ; |9|
           NOP             5
           ; BRANCH OCCURS ; |9|

   .global   _a
   .bss   _a,40,4
```

3.13.2 Alias Disambiguation

C and C++ programs generally use many pointer variables. Frequently, compilers are unable to determine whether or not two or more I values (lowercase L: symbols, pointer references, or structure references) refer to the same memory location. This aliasing of memory locations often prevents the compiler from retaining values in registers because it cannot be sure that the register and memory continue to hold the same values over time.

Alias disambiguation is a technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

3.13.3 Branch Optimizations and Control-Flow Simplification

The compiler analyzes the branching behavior of a program and rearranges the linear sequences of operations (basic blocks) to remove branches or redundant conditions. Unreachable code is deleted, branches to branches are bypassed, and conditional branches over unconditional branches are simplified to a single conditional branch.

When the value of a condition is determined at compile time (through copy propagation or other data flow analysis), the the compiler can delete a conditional branch. Switch case lists are analyzed in the same way as conditional branches and are sometimes eliminated entirely. Some simple control flow constructs are reduced to conditional instructions, totally eliminating the need for branches.

In Example 3−5, the switch statement and the state variable from this simple finite state machine example are optimized completely away, leaving a streamlined series of conditional branches.

Example 3−5. Control-Flow Simplification and Copy Propagation

(a) C source

```
fsm()
{
    enum { ALPHA, BETA, GAMMA, OMEGA } state = ALPHA;
    int *input;
    while (state != OMEGA)
        switch (state)
        {
            case ALPHA:  state = (*input++ == 0) ?  BETA:  GAMMA; break;
            case BETA:   state = (*input++ == 0) ?  GAMMA: ALPHA; break;
            case GAMMA:  state = (*input++ == 0) ?  GAMMA: OMEGA; break;
        }
}

main()
{
    fsm();
}
```

Example 3−5.Control Flow Simplification and Copy Propagation (Continued)

(b) Compiler output

```
FP  .set   A15
DP  .set   B14
SP  .set   B15

;   OPT6X.EXE -O3 fsm.if fsm.opt
    .sect  ".text"
    .global   _fsm

;******************************************************************************
;* FUNCTION NAME: _fsm                                                        *
;*                                                                            *
;*   Regs Modified     : B0,B4                                                *
;*   Regs Used         : B0,B3,B4                                             *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                    *
;******************************************************************************
_fsm:
;** --------------------------------------------------------------------------*
;** --------------------------------------------------------------------------*
L2:
            LDW     .D2T2   *B4++,B0     ; |8|
;** --------------------------------------------------------------------------*
L3:
            NOP             4
   [ B0]    B       .S1     L7           ; |8|
            NOP             4
   [ B0]    LDW     .D2T2   *B4++,B0     ; |10|
            ; BRANCH OCCURS ; |8|
;** --------------------------------------------------------------------------*
            LDW     .D2T2   *B4++,B0     ; |9|
            NOP             4
   [ B0]    B       .S1     L3           ; |9|
            NOP             4
   [ B0]    LDW     .D2T2   *B4++,B0     ; |8|
            ; BRANCH OCCURS ; |9|
;** --------------------------------------------------------------------------*
L5:
            LDW     .D2T2   *B4++,B0     ; |10|
;** --------------------------------------------------------------------------*
L6:
;** --------------------------------------------------------------------------*
L7:
            NOP             4
   [!B0]    B       .S1     L6           ; |10|
            NOP             4
   [!B0]    LDW     .D2T2   *B4++,B0     ; |10|
            ; BRANCH OCCURS ; |10|
;** --------------------------------------------------------------------------*
            B       .S2     B3           ; |12|
            NOP             5
            ; BRANCH OCCURS ; |12|
```

3.13.4 Data Flow Optimizations

Collectively, the following data flow optimizations replace expressions with less costly ones, detect and remove unnecessary assignments, and avoid operations that produce values that are already computed. The compiler with optimization enabled performs these data flow optimizations both locally (within basic blocks) and globally (across entire functions).

- **Copy propagation**

  Following an assignment to a variable, the compiler replaces references to the variable with its value. The value can be another variable, a constant, or a common subexpression. This can result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable (see Example 3−5 on page 3-39 and Example 3−6 on page 3-42).

- **Common subexpression elimination**

  When two or more expressions produce the same value, the compiler computes the value once, saves it, and reuses it.

- **Redundant assignment elimination**

  Often, copy propagation and common subexpression elimination optimizations result in unnecessary assignments to variables (variables with no subsequent reference before another assignment or before the end of the function). The compiler removes these dead assignments (see Example 3−6).

3.13.5 Expression Simplification

For optimal evaluation, the compiler simplifies expressions into equivalent forms, requiring fewer instructions or registers. Operations between constants are folded into single constants. For example, a = (b + 4) − (c + 1) becomes a = b − c + 3 (see Example 3−6).

In Example 3−6, the constant 3, assigned to a, is copy propagated to all uses of a; a becomes a dead variable and is eliminated. The sum of multiplying j by 3 plus multiplying j by 2 is simplified into b = j * 5. The assignments to a and b are eliminated and their values returned.

Example 3−6. Data Flow Optimizations and Expression Simplification

(a) C source

```
char simplify(char j)
{
   char a = 3;
   char b = (j*a) + (j*2);
   return b;
}
```

(b) Compiler output

```
FP  .set   A15
DP  .set   B14
SP  .set   B15

;   opt6x -O2 t1.if t1.opt
    .sect      ".text"
    .global    _simplify


_simplify:
    B      .S2    B3
    NOP           2
    MPY    .M1    5,A4,A0
    NOP           1
    EXT    .S1    A0,24,24,A4
    ; BRANCH OCCURS
```

3.13.6 Inline Expansion of Functions

The compiler replaces calls to small functions with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations (see Example 3−7).

In Example 3−7, the compiler finds the code for the C function plus( ) and replaces the call with the code.

Example 3−7. Inline Function Expansion

(a) C source

```
int plus (int x, int y)
{
   return x + y;
}
main ()
{
   int a = 3;
   int b = 4;
   int c = 5;

   return plus (a, plus (b, c));
}
```

(b) Compiler output

```
FP  .set   A15
DP  .set   B14
SP  .set   B15

;  opt6x -O3 t2.if t2.opt
   .sect     ".text"
   .global   _main;

_main:
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING plus()
;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<LEAVING plus()
;>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>ENTERING plus()
;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<LEAVING plus()
       B   .S2    B3
       NOP        4
       MVK .S1    12,A4
       ; BRANCH OCCURS
```

3.13.7 Induction Variables and Strength Reduction

Induction variables are variables whose value within a loop is directly related to the number of executions of the loop. Array indices and control variables for loops are often induction variables.

Strength reduction is the process of replacing inefficient expressions involving induction variables with more efficient expressions. For example, code that indexes into a sequence of array elements is replaced with code that increments a pointer through the array.

Induction variable analysis and strength reduction together often remove all references to your loop-control variable, allowing its elimination (see Example 3−4 on page 3-36).

3.13.8 Loop-Invariant Code Motion

This optimization identifies expressions within loops that always compute to the same value. The computation is moved in front of the loop, and each occurrence of the expression in the loop is replaced by a reference to the precomputed value.

3.13.9 Loop Rotation

The compiler evaluates loop conditionals at the bottom of loops, saving an extra branch out of the loop. In many cases, the initial entry conditional check and the branch are optimized out.

3.13.10 Register Variables

The compiler helps maximize the use of registers for storing local variables, parameters, and temporary values. Accessing variables stored in registers is more efficient than accessing variables in memory. Register variables are particularly effective for pointers (see Example 3−4 on page 3-36).

3.13.11 Register Tracking/Targeting

The compiler tracks the contents of registers to avoid reloading values if they are used again soon. Variables, constants, and structure references such as (a.b) are tracked through straight-line code. Register targeting also computes expressions directly into specific registers when required, as in the case of assigning to register variables or returning values from functions (see Example 3−8 on page 3-45).

Example 3−8. Register Tracking/Targeting

(a) C source

```
int x, y;

main()
{
   x += 1;
   y = x;
}
```

(b) Compiler output

```
FP  .set   A15
DP  .set   B14
SP  .set   B15

;   opt6x -O2 t3.if t3.opt
    .sect      ".text"
    .global    _main

_main:
           LDW .D2    *+B14(_x),B4
           NOP        1
           B   .S2    B3
           NOP        2
           ADD .L2    1,B4,B4
           STW .D2    B4,*+B14(_y)
           STW .D2    B4,*+B14(_x)
           ; BRANCH OCCURS

    .global    _x
    .bss       _x,4,4
    .global    _y
    .bss       _y,4,4
```

3.13.12 Software Pipelining

Software pipelining is a technique use to schedule from a loop so that multiple iterations of a loop execute in parallel. See section 3.2, *Optimizing Software Pipelining*, on page 3-4, for more information.

Chapter 4

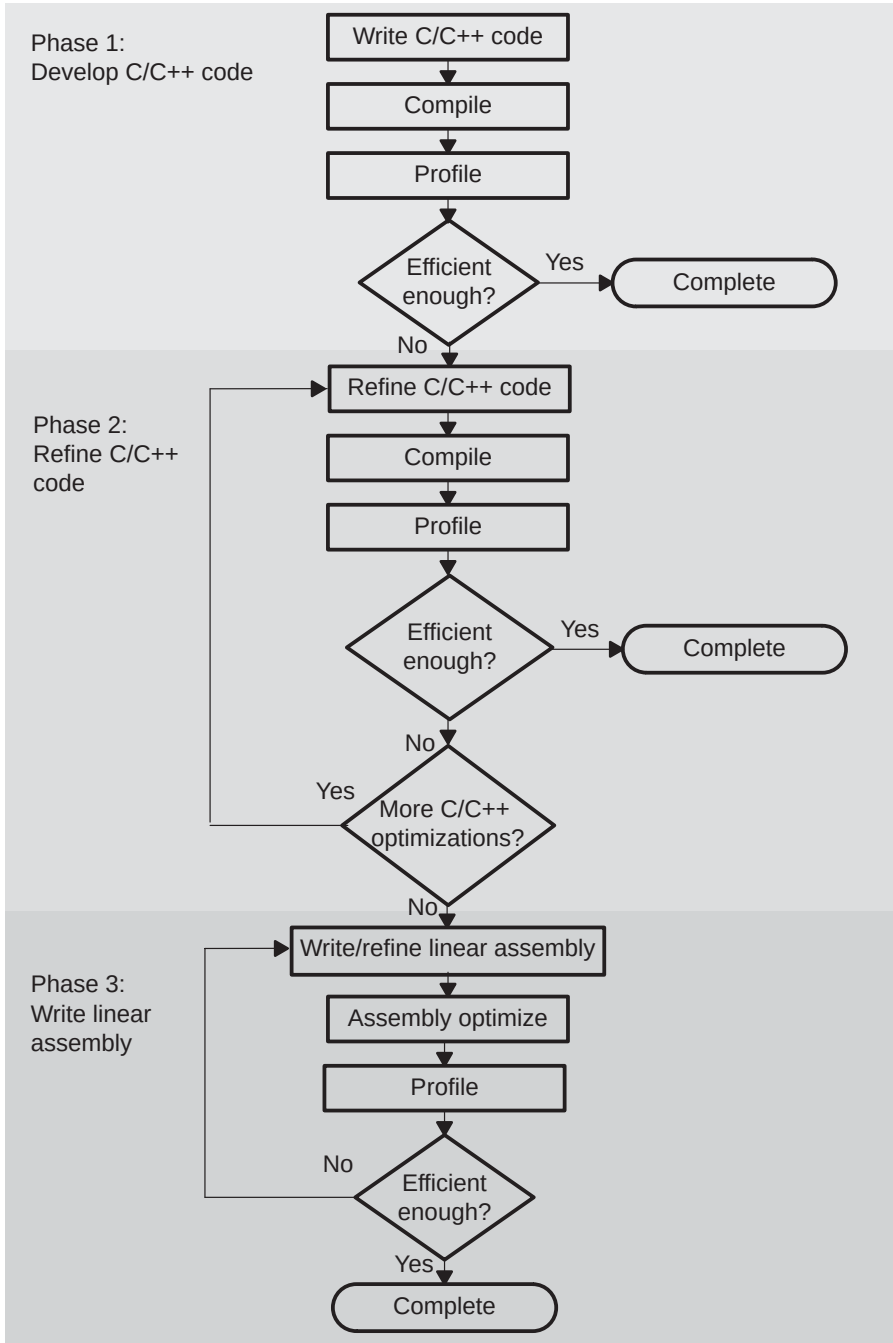
Using the Assembly Optimizer

The assembly optimizer allows you to write assembly code without being concerned with the pipeline structure of the C6000 or assigning registers. It accepts *linear assembly code*, which is assembly code that may have had register-allocation performed and is unscheduled. The assembly optimizer assigns registers and uses loop optimizations to turn linear assembly into highly parallel assembly.

Topic **Page**

4.1 Code Development Flow to Increase Performance

You can achieve the best performance from your C6000 code if you follow this flow when you are writing and debugging your code:



There are three phases of code development for the C6000:

- **Phase 1: write in C**

  You can develop your C/C++ code for phase 1 without any knowledge of the C6000. Use the stand-alone simulator with the −g option (see section 6.4, *Using the Profiling Capability of the Stand-Alone Simulator*, on page 6-8) to identify any inefficient areas in your C/C++ code. To improve the performance of your code, proceed to phase 2.

- **Phase 2: refine your C/C++ code**

  In phase 2, use the intrinsics and compiler options that are described in this book to improve your C/C++ code. Use the stand-alone simulator with the −g option to check the performance of your altered code. Refer to the *TMS320C6000 Programmer's Guide* for hints on refining C/C++ code. If your code is still not as efficient as you would like it to be, proceed to phase 3.

- **Phase 3: write linear assembly**

  In this phase, you extract the time-critical areas from your C/C++ code and rewrite the code in linear assembly. You can use the assembly optimizer to optimize this code. When you are writing your first pass of linear assembly, you should not be concerned with the pipeline structure or with assigning registers. Later, when you are refining your linear assembly code, you might want to add more details to your code, such as partitioning registers.

  Improving performance in this stage takes more time than in phase 2, so try to refine your code as much as possible before using phase 3. Then, you should have smaller sections of code to work on in this phase.

4.2 About the Assembly Optimizer

If you are not satisfied with the performance of your C/C++ code after you have used all of the C/C++ optimizations that are available, you can use the assembly optimizer to make it easier to write assembly code for the C6000.

The assembly optimizer performs several tasks including the following:

- ❏ Schedules instructions to maximize performance using the instruction-level parallelism of the C6000
- ❏ Ensures that the instructions conform to the C6000 latency requirements
- ❏ Allocates registers for your source code

Like the C/C++ compiler, the assembly optimizer performs software pipelining. *Software pipelining* is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. The code generation tools attempt to software pipeline your code with inputs from you and with information that it gathers from your program. For more information, see section 3.2, *Software Pipelining*, on page 3-4.

To invoke the assembly optimizer, use the compiler program (cl6x). The assembly optimizer is automatically invoked by the compiler program if one of your input files has a *.sa* extension. You can specify C/C++ source files along with your linear assembly files. For more information about the compiler program, see section 2.1, on page 2-2.

4.3 What You Need to Know to Write Linear Assembly

By using the C6000 profiling tools, you can identify the time-critical sections of your code that need to be rewritten as linear assembly. The source code that you write for the assembly optimizer is similar to assembly source code. However, linear assembly code does not need to be partitioned, scheduled, or register allocated. The intention is for you to let the assembly optimizer determine this information for you. When you are writing linear assembly code, you need to know about these items:

- ❏ **Assembly optimizer directives**

  Your linear assembly file can be a combination of assembly optimizer code and regular assembly source. Use the assembly optimizer directives to differentiate the assembly optimizer code from the regular assembly code and to provide the assembly optimizer with additional information about your code. The assembly optimizer directives are described in section 4.4, on page 4-13.

❑ Options that affect what the assembly optimizer does

The following compiler options affect the behavior of the assembly optimizer:

Option	Effect	Page
−el	Changes the default extension for assembly optimizer source files	2-21
−fl	Changes how assembly optimizer source files are identified	2-20
−k	Keeps the assembly language (.asm) file	2-16
−mi*n*	Specifies an interrupt threshold value	2-43
−ms*n*	Controls code size on four levels (−ms0, −ms1, −ms2, and −ms3)	3-17
−mt	Presumes no memory aliasing	3-27
−mu	Turns off software pipelining	3-5
−mv*n*	Select target version	2-17
−mw	Generates verbose software pipelining information	3-5
−n	Compiles or assembly optimizes only (does not assemble)	2-16
−o*n*	Increases level of optimization (−o0, −o1, −o2, and −o3)	3-2
−q	Suppresses progress messages	2-16
−−speculate_loads=*n*	Allows speculative execution of loads with bounded address ranges	3-14

❑ TMS320C6000 instructions

When you are writing your linear assembly, your code does *not* need to indicate the following:

- Pipeline latency
- Register usage
- Which unit is being used

As with other code generation tools, you might need to modify your linear assembly code until you are satisfied with its performance. When you do this, you will probably want to add more detail to your linear assembly. For example, you might want to partition or assign some registers.

Note: Do Not Use Scheduled Assembly Code as Source

The assembly optimizer assumes that the instructions in the input file are placed in the logical order in which you would like them to occur (that is, linear assembly code). Parallel instructions are illegal. On the other hand, the assembler assumes that you have placed instructions in a location that accounts for any delay slots due to pipeline latency. Therefore, it is not valid to use code written for the assembler (that is, scheduled assembly code), or assembly optimizer output, as input for the assembly optimizer.

- **Linear assembly source statement syntax**

  The linear assembly source programs consist of source statements that can contain assembly optimizer directives, assembly language instructions, and comments. See section 4.3.1 for more information on the elements of a source statement.

- **Specifying registers or register sides**

  Registers can be assigned explicitly to user symbols. Alternatively, symbols can be assigned to the A-side or B-side leaving the compiler to do the actual register allocation. See section 4.3.2 for information on specifying registers.

- **Specifying the functional unit**

  The functional unit specifier is optional in both regular assembly code and linear assembly code. Specifying the functional unit enables you to control which side of the register file is used for an instruction, which helps the assembly optimizer perform functional unit and register allocation. This method is obsolete, specifying registers is preferred. See section 4.3.3 for information on specifying the functional unit.

- **Source comments**

  The assembly optimizer attaches the comments on instructions from the input linear assembly to the output file. It attaches the 2-tuple <x, y> to the comments to specify which iteration and cycle of the loop an instruction is on in the software pipeline. The zero-based number x represents the iteration the instruction is on during the first execution of the kernel. The zero-based number y represents the cycle the instruction is scheduled on within a single iteration of the loop. See section 4.3.4, *Using Linear Assembly Source Comments*, on page 4-11 for an illustration of the use of source comments and the resulting assembly optimizer output.

4.3.1 Linear Assembly Source Statement Format

A source statement can contain five ordered fields (label, mnemonic, unit specifier, operand list, and comment). The general syntax for source statements is as follows:

[*label* [:]] [**[***register***]**] *mnemonic* [*unit specifier*] [*operand list*] [;*comment*]

label [:] Labels are optional for all assembly language instructions and for most (but not all) assembly optimizer directives. When used, a label must begin in column 1 of a source statement. A label can be followed by a colon.

[*register***]** Square brackets ([]) enclose conditional instructions. The machine-instruction mnemonic is executed based on the value of the register within the brackets; valid register names are A0 for C6400 only, A1, A2, B0, B1, B2, or symbolic.

mnemonic The mnemonic is a machine-instruction (such as ADDK, MVKH, B) or assembly optimizer directive (such as .proc, .trip)

unit specifier The optional unit specifier enables you to specify the functional unit operand. Only the specified unit side is used; other specifications are ignored. The preferred method is specifying register sides.

operand list The operand list is not required for all instructions or directives. The operands can be symbols, constants, or expressions and must be separated by commas.

comment Comments are optional. Comments that begin in column 1 must begin with a semicolon or an asterisk; comments that begin in any other column must begin with a semicolon.

The C6000 assembly optimizer reads up to 200 characters per line. Any characters beyond 200 are truncated. Keep the operational part of your source statements (that is, everything other than comments) less than 200 characters in length for correct assembly. Your comments can extend beyond the character limit, but the truncated portion is not included in the .asm file.

Follow these guidelines in writing linear assembly code:

- ❏ All statements must begin with a label, a blank, an asterisk, or a semicolon.
- ❏ Labels are optional; if used, they must begin in column 1.
- ❏ One or more blanks must separate each field. Tab characters are interpreted as blanks. You must separate the operand list from the preceding field with a blank.

- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.

- If you set up a conditional instruction, the register must be surrounded by square brackets.

- A mnemonic cannot begin in column 1 or it is interpreted as a label.

See the *TMS320C6000 Assembly Language Tools User's Guide* for information on the syntax of C6000 instructions, including conditional instructions, labels, and operands.

4.3.2 Register Specification for Linear Assembly

There are only two cross paths in the C6000. This limits the C6000 to one source read from each data path's opposite register file per cycle. The compiler must select a side for each instruction; this is called partitioning.

It is recommended that you do not initially partition the linear assembly source code by hand. This allow the compiler more freedom to partition and optimize your code. If the compiler does not find an optimal partition in a software pipelined loop, then you can partition enough instructions by hand to force optimal partitioning by using functional unit specifiers. If you add functional unit specifiers to your linear assembly source code, this gives the compiler more information on where these and subsequent instructions should go.

You select registers for partitioning with the .reg directive or a register side with the .rega and .regb directives (see page 4-28).

The **.reg** directive allows you to use descriptive names for values that are stored in registers. The assembly optimizer chooses a register for you such that its use agrees with the functional units chosen for the instructions that operate on the value. See page 4-26 for further details and examples of the .reg directive.

Registers can be directly partitioned through two directives. The **.rega** directive is used to constrain a symbolic name to A-side registers. The **.regb** directive is used to constrain a symbolic name to B-side registers. See page 4-28 for further details on the .rega and .regb directives.

Example 4−1 is a hand-coded linear assembly program that computes a dot product; compare to Example 4−2, which illustrates C code.

Example 4−1. Linear Assembly Code for Computing a Dot Product

```
_dotp: .cproc a_0, b_0

       .rega     a_4, tmp0, sum0, prod1, prod2
       .regb     b_4, tmp1, sum1, prod3, prod4
       .reg      cnt, sum
       .reg      val0, val1

       ADD    4, a_0, a_4
       ADD    4, b_0, b_4
       MVK    100, cnt
       ZERO   sum0
       ZERO   sum1

loop:  .trip   25
       LDW          *a_0++[2], val0      ; load a[0-1]
       LDW          *b_0++[2], val1      ; load b[0-1]
       MPY          val0, val1, prod1    ; a[0] * b[0]
       MPYH         val0, val1, prod2    ; a[1] * b[1]
       ADD          prod1, prod2, tmp0   ; sum0 += (a[0]*b[0]) +
       ADD          tmp0, sum0, sum0     ;         (a[1]*b[1])

       LDW          *a_4++[2], val0      ; load a[2-3]
       LDW          *b_4++[2], val1      ; load b[2-3]
       MPY          val0, val1, prod3    ; a[2] * b[2]
       MPYH         val0, val1, prod4    ; a[3] * b[3]
       ADD          prod3, prod4, tmp1   ; sum1 =+ (a[2]*b[2]) +
       ADD          tmp1, sum1, sum1     ;         (a[3]*b[3])

[cnt]  SUB    cnt, 4, cnt        ; cnt -= 4
[cnt]  B      loop               ; if (cnt!=0) goto loop

       ADD    sum0, sum1, sum    ; compute final result

       .return sum
       .endproc
```

Example 4−2 is refined C code for computing a dot product.

Example 4−2. C Code for Computing a Dot Product

```
int dotp(short a[], shortb[])
{
    int sum0 = 0;
    int sum1 = 0;

    int sum, i;

    for (i = 0; i < 100/4; i +=4)
    {
        sum0 += a[i]   * b[i];
        sum0 += a[i+1] * b[i+1];

        sum1 += a[i+2] * b[i+2];
        sum1 += a[i+3] *  [b[i+3];
    }

    return
}
```

The old method of partitioning registers indirectly by partitioning instructions can still be used. Side and functional unit specifiers can still be used on instructions. However, functional unit specifiers (.L/.S/.D/.M) are ignored. Side specifiers are translated into partitioning constraints on the corresponding symbolic names, if any. For example:

```
MV  .1     x, y     ; translated to .REGA y
LDW .D2T2 *u, v:w   ; translated to .REGB u, v, w
```

4.3.3 Functional Unit Specification for Linear Assembly

Specifying functional units has been deprecated by the ability to partition registers directly. (See section 4.3.2 on page 4-8 for details.) While you can use the unit specifier field in linear assembly, only the register side information is used by the compiler.

You specify a functional unit by following the assembler instruction with a period (.) and a functional unit specifier. One instruction can be assigned to each functional unit in a single instruction cycle. There are eight functional units, two of each functional type, and two address paths. The two of each functional type are differentiated by the data path each uses, A or B.

.D1 and **.D2**	Data/addition/subtraction operations
.L1 and **.L2**	Arithmetic logic unit (ALU)/compares/long data arithmetic
.M1 and **.M2**	Multiply operations
.S1 and **.S2**	Shift/ALU/branch/field operations
.T1 and **.T2**	Address paths

There are several ways to enter the unit specifier filed in linear assembly. Of these, only the specific register side information is recognized and used:

- ❑ You can specify the particular functional unit (for example, .D1).
- ❑ You can specify the .D1 or .D2 functional unit followed by T1 or T2 to specify that the nonmemory operand is on a specific register side. T1 specifies side A and T2 specifies side B. For example:

```
LDW  .D1T2    *A3[A4], B3
LDW  .D1T2    *src, dst
```

- ❑ You can specify only the functional type (for example, .M), and the assembly optimizer assigns the specific unit (for example, .M2).
- ❑ You can specify only the data path (for example, .1), and the assembly optimizer assigns the functional type (for example, .L1).

Whether you specify the functional unit or not, the assembly optimizer selects the functional unit based on the mnemonic field.

For more information on functional units, including which machine-instruction mnemonics require which functional type, see the *TMS320C6000 CPU and Instruction Set Reference Guide*.

4.3.4 Using Linear Assembly Source Comments

Your comments in linear assembly can begin in any column and extend to the end of the source line. A comment can contain any ASCII character, including blanks. Your xomments are printed in the linear assembly source listing, but they do not affect the linear assembly.

A source statement that contains only a comment is valid. If it begins in column 1, it can start with a semicolon (**;**) or an asterisk (*). Comments that begin anywhere else on the line must begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

The assembly optimizer schedules instructions; that is, it rearranges instructions. Stand-alone comments are moved to the top of a block of instructions. Comments at the end of an instruction statement remain in place with the instruction.

If you enter comments on instructions in your linear assembly input file, the assembly optimizer moves the comments to the output file along with additional information. It attaches a 2-tuple <x, y> to the comments to specify the iteration and cycle of the lop an instruction is on in the software pipeline. The zero-based number x represents the iteration the instruction is on during the first execution of the loop kernel. The zero-based number y represents the cycle that the instruction is scheduled on within a single iteration of the loop.

Example 4−3 shows code for a function called Lmac that contains comments.

Example 4−3. Lmac Function Code Showing Comments

```
Lmac:   .cproc   A4,B4

        .reg    t0,t1,p,i,sh:sl

        MVK     100,i
        ZERO    sh
        ZERO    sl

loop:   .trip   100

        LDH           *a4++, t0       ; t0 = a[i]
        LDH           *b4++, t1       ; t1 = b[i]
        MPY           t0,t1,p         ; prod = t0 * t1
        ADD           p,sh:sl,sh:sl   ; sum += prod
[i]     ADD           -1,i,i          ; --i
[i]     B             loop            ; if (i) goto loop

        .return sh:sl

        .endproc
```

4.3.5 Assembly File Retains Your Symbolic Register Names

In the output assembly file, register operands contain your symbolic name. This aids you in debugging your linear assembly files and in gluing snippets of linear assembly output into assembly files.

A .map directive (see page 4-20) at the beginning of an assembly function associates the symbolic name with the actual register. In other words, the symbolic name becomes an alias for the actual register.

When the compiler splits a user symbol into two symbols and each is mapped to distinct machine register, a suffix is appended to instances of the symbolic name to generate unique names so that each unique name is associated with one machine register.

For example, if the compiler associated the symbolic name y with A5 in some instructions and B6 in some others, the output assembly code might look like:

```
.MAP y/A5
.MAP y'/B6
...
ADD .S2X y, 4, y'  ; Equivalent to add A5, 4, B6
```

To disable this format with symbolic names and display assembly instructions with actual registers instead, compile with the −−machine_regs option.

4.4 Assembly Optimizer Directives

Assembly optimizer directives supply data for and control the assembly optimization process. The assembly optimizer optimizes linear assembly code that is contained within procedures; that is, code within the .proc and .endproc directives or within the .cproc and .endproc directives. If you do not use these directives in your linear assembly file, your code will not be optimized by the assembly optimizer. This section describes these directives and others that you can use with the assembly optimizer.

Table 4−1 summarizes the assembly optimizer directives. It provides the syntax for each directive, a description of each directive, any restrictions that you should keep in mind, and a page reference for more detail.

Table 4−1. Assembly Optimizer Directives Summary

Syntax	Description	Restrictions	Page
.call [*ret_reg* =] *func_name* **(***arg1*, *arg2***)**	Calls a function	Valid only within procedures	4-15
.circ *variable1***/***register1*[**,** *variable2***/** *register2*]	Declares circular addressing	Must manualy insert setup/teardown code for circular addressing	4-17
label **.cproc** [*variable*$_1$ [**,** *variable*$_2$**,** …]]	Start a C/C++ callable procedure	Must use with .endproc	4-17
.endproc	End a C/C++ callable procedure	Must use with .cproc	4-17
.endproc [*register*$_1$ [**,** *register*$_2$**,** …]]	End a procedure	Must use with .proc	4-24
.map *variable1***/***register1*[**,** *variable2***/** *register2*]	Assigns a symbol to a register	Must use an actual machine register	4-20
.mdep [*symbol1*[, *symbol2*]]	Indicates a memory dependence	Valid only within procedures	4-21
.mptr {*register*|*symbol}*, *base* [**+** *offset*] [**,** *stride*]	Avoid memory bank conflicts	Valid only within procedures; can use variables in the register parameter	4-21
.no_mdep	No memory aliases in the function	Valid only within procedures	4-23
.pref *variable***/***register*[**/***register...*], ...	Assigns a symbol to a register in a set	Must use actual machine registers	4-23
label **.proc** [*register*$_1$ [**,** *register*$_2$**,** …]]	Start a procedure	Must use with .endproc	4-24
.reg *variable*$_1$ [**,** *variable*$_2$**,**…]	Declare variables	Valid only within procedures	4-26

Table 4−1. Assembly Optimizer Directives Summary (Continued)

Syntax	Description	Restrictions	Page
.rega *variable*$_1$ [**,** *variable*$_2$**,**…]	Partition symbol to A-side register	Valid only within procedures	4-28
.regb *variable*$_1$ [**,** *variable*$_2$**,**…]	Partition symbol to B-side register	Valid only within procedures	4-28
.reserve [*register*$_1$ [**,** *register*$_2$**,** …]]	Prevents the compiler from allocating a register	Valid only within procedures	4-28
.return [*argument*]	Return a value to a procedure	Valid only within .cproc procedures	4-29
label **.trip** *min*	Specify trip count value	Valid only within procedures	4-30
.volatile *variable*$_1$ [**,** *variable*$_2$**,**…]	Designate memory reference volatile	Use −mi1 if reference may be modified during an interrupt	4-32

.call — *Calls a Function*

Syntax

.call [*ret_reg* =] *func_name* **(**[*arg1, arg2,...*]**)**

Description

Use the **.call** directive to call a function. Optionally, you can specify a register that is assigned the result of the call. The register can be a symbolic or machine register. The .call directive adheres to the same register and function calling conventions as the C/C++ compiler. For information, see section 8.3, *Register Conventions*, on page 8-17, and section 8.4, *Function Structure and Calling Conventions*, on page 8-19. There is no support for alternative register or function calling conventions.

You cannot call a function that has a variable number of arguments, such as printf. No error checking is performed to ensure the correct number and/or type of arguments is passed. You cannot pass or return structures through the .call directive.

Following is a description of the .call directive parameters:

ret_reg (Optional) Symbolic/machine register that is assigned the result of the call. If not specified, the assembly optimizer presumes the call overwrites the registers A5 and A4 with a result.

func_name The name of the function to call, or the name of the symbolic/machine register for indirect calls. A register pair is not allowed. The label of the called function must be defined in the file. If the code for the function is not in the file, the label must be defined with the .global or .ref directive. If you are calling a C/C++ function, you must use the appropriate linkname of that function. See section 7.8, *Generating Linknames*, on page 7-33 for more information.

arguments (Optional) Symbolic/machine registers passed as an argument. The arguments are passed in this order and cannot be a constant, memory reference, or other expression.

You can use the cl6x −ml*n* option to indicate whether a call is near or far. If the −ml*n* option is set to 0 or if no level is specified (default), the call is near. If the −ml*n* option is set to 1, 2, or 3, the call is far. To force a far call, you must explicitly load the address of the function in a register, and then issue an indirect call. For example:

```
MVK   func,reg
MVKH  func,reg
.call reg(op1)              ; forcing a far call
```

If you want to use * for indirection, you must abide by C/C++ syntax rules, and use the following alternate syntax:

.call [*ret_reg* =] (* *ireg*) **(**[*arg1, arg2,...*]**)**

For example:

```
.call  (*driver)(op1, op2) ; indirect call

.reg   driver
.call  driver(op1, op2)    ; also an indirect call
```

Here are other valid examples that use the .call syntax.

```
.call  fir(x, h, y)            ; void function

.call  minimal()               ; no arguments

.call  sum = vecsum(a, b)      ; returns an int

.call  hi:lo = _atol(string)   ; returns a long
```

Since you can use machine register names anywhere you can use symbolic registers, it may appear you can change the function calling convention. For example:

```
.call   A6 = compute()
```

It appears that the result is returned in A6 instead of A4. This is incorrect. Using machine registers does not override the calling convention. After returning from the *compute* function with the returned result in A4, a MV instruction transfers the result to A6.

Here is a complete .call example:

```
        .global _main
        .global _puts, _rand, _ltoa
        .sect   ".const"
string1:        .string "The random value returned is ", 0
string2:        .string " ", 10, 0  ; '10' == newline
        .bss    charbuf, 20
        .text
_main:  .cproc
        .reg    random_value, bufptr, ran_val_hi:ran_val_lo
        .call   random_value = _rand()               ; get a random value

        MVKL    string1, bufptr                      ; load address of string1
        MVKH    string1, bufptr
        .call   _puts(bufptr)                        ; print out string1
        MV      random_value, ran_val_lo
        SHR     ran_val_lo, 31, ran_val_hi           ; sign extend random value
        .call   _ltoa(ran_val_hi:ran_val_lo, bufptr) ; convert it to a string
        .call   _puts(bufptr)                        ; print out the random value
        MVKL    string2, bufptr                      ; load address of string2
        MVKH    string2, bufptr
        .call   _puts(bufptr)                        ; print out a newline
        .endproc
```

.circ — *Declare Circular Addressing*

Syntax

.circ *variable***/***register* [**,** *variable***/***register*, ...]

Description

The **.circ** directive assigns a variable name to a machine register and declares the variable as available for circular addressing. The compiler then assigns the variable to the register and ensures that all code transformations are safe in this situation. You must insert setup/teardown code for circular addressing.

variable — A valid symbol name to be assigned to the register. The variable is up to 128 characters long and must begin with a letter. Remaining characters of the variable can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

register — Name of the actual register to be assigned a variable.

The compiler assumes that it is safe to speculate any load using an explicitly declared circular addressing variable as the address pointer and may exploit this assumption to perform optimizations.

When a variable is declared with the .circ directive, it is not necessary to declare that variable with the .reg directive.

The .circ directive is equivalent to using .map (page 4-20) with a circular declaration.

Example

Here the symbolic name Ri is assigned to actual register Mi and Ri is declared as potentially being used for circular addressing.

```
.CIRC R1/M1, R2/M2 ...
```

.cproc/.endproc — *Define a C Callable Procedure*

Syntax

label **.cproc** [$variable_1$ [**,** $variable_2$**,** …]]
.endproc

Description

Use the **.cproc/.endproc** directive pair to delimit a section of your code that you want the assembly optimizer to optimize and treat as a C/C++ callable function. This section is called a procedure. The .cproc directive is similar to the .proc directive in that you use .cproc at the beginning of a section and .endproc at the end of a section. In this way, you can set off sections of your assembly code that you want to be optimized, like functions. The directives must be used in pairs; do not use .cproc without the corresponding .endproc. Specify a label with the .cproc directive. You can have multiple procedures in a linear assembly file.

The .cproc directive differs from the .proc directive in that the compiler treats the .cproc region as a C/C++ callable function. The assembly optimizer performs some operations automatically in a .cproc region in order to make the function conform to the C/C++ calling conventions and to C/C++ register usage conventions.

These operations include the following:

- When you use save-on-entry registers (A10 to A15 and B10 to B15), the assembly optimizer saves the registers on the stack and restores their original values at the end of the procedure.
- If the compiler cannot allocate machine registers to symbolic register names specified with the .reg directive (see page 4-26) it uses local temporary stack variables. With .cproc, the compiler manages the stack pointer and ensures that space is allocated on the stack for these variables.

For more information, see section 8.3, *Register Conventions*, on page 8-17 and section 8.4, *Function Structure and Calling Conventions*, on page 8-19.

Please see the .proc directive on page 4-25 for the types of instructions that cannot appear in a .cproc region.

Use the optional *variable* to represent function parameters. The variable entries are very similar to parameters declared in a C/C++ function. The arguments to the .cproc directive can be of the following types:

- **Machine-register names**. If you specify a machine-register name, its position in the argument list must correspond to the argument passing conventions for C. For example, the C/C++ compiler passes the first argument to a function in register A4. This means that the first argument in a .cproc directive must be A4 or a symbolic name. Up to ten arguments can be used with the .cproc directive.
- **Variable names**. If you specify a variable name, then the assembly optimizer ensures that either the variable name is allocated to the appropriate argument passing register or the argument passing register is copied to the register allocated for the variable name. For example, the first argument in a C/C++ call is passed in register A4, so if you specify the following .cproc directive:

  ```
  frame   .cproc arg1
  ```

  The assembly optimizer either allocates arg1 to A4, or arg1 is allocated to a different register (such as B7) and an MV A4, B7 is automatically generated.

- **Register pairs**. A register pair is specified as arghi:arglo and represents a 40-bit argument or a 64-bit type double argument for 'C6700. For example, the .cproc defined as follows:

```
_fcn:   .cproc  arg1,  arg2hi:arg2lo, arg3, B6, arg5, B9:B8

        ...
        .return res
        ...
        .endproc
```

  corresponds to a C function declared as:

```
int fcn(int arg1, long arg2, int arg3, int arg4, int arg5, long arg6);
```

  In this example, the fourth argument of .cproc is register B6. This is allowed since the fourth argument in the C/C++ calling conventions is passed in B6. The sixth argument of .cproc is the actual register pair B9:B8. This is allowed since the sixth argument in the C/C++ calling conventions is passed in B8 or B9:B8 for longs.

If you are calling a procedure from C++ source, you must use the appropriate linkname for the procedure label. Otherwise, you can force C naming conventions by using the extern C declaration. See section 7.8, *Generating Linknames*, on page 7-33, and section 8.5, *Interfacing C/C++ with Assembly Language*, on page 8-23, for more information.

When .endproc is used with a .cproc directive, it cannot have arguments. The *live out* set for a .cproc region is determined by any .return directives that appear in the .cproc region. (A value is *live out* if it has been defined before or within the procedure and is used as an output from the procedure.) Returning a value from a .cproc region is handled by the .return directive. The return branch is automatically generated in a .cproc region. See page 4-29 for information on the .return directive.

Only code within procedures is optimized. The assembly optimizer copies any code that is outside of procedures to the output file and does not modify it. See page 4-25 for a list of instruction types that cannot be used in .cproc regions.

Example Here is an example in which .cproc and .endproc are used:

```
_if_then: .cproc   a, cword, mask, theta

        .reg    cond, if, ai, sum, cntr

        MVK         32,cntr             ; cntr = 32
        ZERO        sum                 ; sum = 0

LOOP:
        AND         cword,mask,cond     ; cond = codeword & mask
 [cond] MVK         1,cond              ; !(!(cond))
        CMPEQ       theta,cond,if       ; (theta == !(!(cond)))
        LDH         *a++,ai             ; a[i]
 [if]   ADD         sum,ai,sum          ; sum += a[i]
 [!if]  SUB         sum,ai,sum          ; sum -= a[i]
        SHL         mask,1,mask         ; mask = mask << 1
 [cntr] ADD         -1,cntr,cntr        ; decrement counter
 [cntr] B           LOOP                ; for LOOP

        .return sum

        .endproc
```

.map — *Assign a Variable to a Register*

Syntax **.map** *variable*/*register* [**,** *variable*/*register*, ...]

Description The **.map** directive assigns variable names to machine registers. Symbols/variables are stored in the substitution symbol table. The association between symbolic names and actual registers is wiped out at the beginning and end of each linear assembly function. The .map directive can be used in assembly and linear assembly files.

variable A valid symbol name to be assigned to the register. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the variable can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

register Name of the actual register to be assigned a variable.

When a variable is declared with the .map directive, it is not necessary to declare that variable with the .reg directive.

Example Here the .map directive is used to assign x to register A6 and y to register B7. The variables are used with a move statement.

```
        .MAP x/A6, y/B7

   MV  x, y     ; Equivalent to MV A6, B7
```

.mdep — *Indicates a Memory Dependence*

Syntax

.mdep *symbol1*, *symbol2*

Description

The **.mdep** directive identifies a specific memory dependence.

Following is a description of the .mdep directive parameters:

symbol The symbol parameter is the name of the memory reference.

The symbol used to name a memory reference has the same syntax restrictions as any assembly symbol. (For more information about symbols, see the *TMS320C6000 Assembly Language Tools User's Guide*.) It is in the same space as the symbolic registers. You cannot use the same name for a symbolic register and annotating a memory reference.

The .mdep directive tells the assembly optimizer that there is a dependence between two memory references.

The .mdep directive is valid only within procedures; that is, within occurrences of the .proc and .endproc directive pair or the .cproc and .endproc directive pair.

Example

Here is an example in which .mdep is used to indicate a dependence between two memory references.

```
.mdep ld1, st1

LDW    *p1++ {ld1}, inp1  ;name memory reference "ld1"
;other code ...
STW    outp2, *p2++ {st1} ;name memory reference "st1"
```

.mptr — *Avoid Memory Bank Conflicts*

Syntax

.mptr {*register* | *symbol*}, *base* [**+** *offset*] [**,** *stride*]

Description

The **.mptr** directive associates a register with the information that allows the assembly optimizer to determine automatically whether two memory operations have a memory bank conflict. If the assembly optimizer determines that two memory operations have a memory bank conflict, then it does not schedule them in parallel.

A memory bank conflict occurs when two accesses to a single memory bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle while the second value is read from memory. For more information on memory bank conflicts, including how to use the .mptr directive to prevent them, see section 4.5 on page 4-33.

Following are descriptions of the .mptr directive parameters:

register|*symbol*	The name of the register or the symbol used to name a specific memory reference.
base	A symbol that associates related memory accesses
offset	The offset in bytes from the starting base symbol. The offset is an optional parameter and defaults to 0.
stride	The register loop increment in bytes. The stride is an optional parameter and defaults to 0.

The .mptr directive tells the assembly optimizer that when the *register* or *symbol* name is used as a memory pointer in an LD(B/BU)(H/HU)(W) or ST(B/H/W) instruction, it is initialized to point to *base* + *offset* and is incremented by *stride* each time through the loop.

The .mptr directive is valid within procedures only; that is, within occurrences of the .proc and .endproc directive pair or the .cproc and .endproc directive pair.

The symbols used for base symbol names are in a name space separate from all other labels. This means that a symbolic register or assembly label can have the same name as a memory bank base name. For example:

```
      .mptr  Darray,Darray
```

Example Here is an example in which .mptr is used to avoid memory bank conflicts.

```
_blkcp:   .cproc i

          .reg   ptr1, ptr2, tmp1, tmp2

          MVK    0x0, ptr1              ; ptr1 = address 0
          MVK    0x8, ptr2              ; ptr2 = address 8

loop:     .trip  50

          .mptr  ptr1, a+0, 4
          .mptr  foo, a+8, 4

                                        ; potential conflict
          LDW    *ptr1++, tmp1          ; load  *0, bank 0
          STW    tmp1, *ptr2++{foo}     ; store *8, bank 0

   [i]    ADD    -1,i,i                 ; i--
   [i]    B      loop                   ; if (!0) goto loop

          .endproc
```

.no_mdep — *No Memory Aliases in the Function*

Syntax

.no_mdep

Description

The **.no_mdep** directive tells the assembly optimizer that no memory dependences occur within that function, with the exception of any dependences pointed to with the .mdep directive.

Example

Here is an example in which .no_mdep is used.

```
fn:    .cproc    dst, src, cnt
       .no_mdep  ;no memory aliasing in this function
       ...
       .endproc
```

.pref — *Assign a Variable to a Register in a Set*

Syntax

.pref *variable***/***register1*[**/***register2...*]

Description

The **.pref** directive communicates a preference to assign a variable to one of a list of registers. Symbols/variables are stored in the substitution symbol table. The association between symbolic names and actual registers is wiped out at the beginning and end of each linear assembly function.

variable — A valid symbol name to be assigned to the register. The substitution symbol is up to 128 characters long and must begin with a letter. Remaining characters of the variable can be a combination of alphanumeric characters, the underscore (_), and the dollar sign ($).

register — List of actual registers to be assigned a variable.

There is no guarantee that the variable will be assigned to any register in the specified group. The compiler may ignore the preference.

When a variable is declared with the .pref directive, it is not necessary to declare that variable with the .reg directive.

Example

Here x is given a preference to be assigned to either A6 or B7. However, It would be correct for the compiler to assign x to B3 (for example) instead.

```
.PREF x/A6/B7 ; Preference to assign x to either A6 or B7
```

.proc/.endproc — *Define a Procedure*

Syntax

label **.proc** [*register*$_1$ [**,** *register*$_2$**,** …]]
.endproc [*register*$_1$ [**,** *register*$_2$**,** …]]

Description

Use the **.proc/.endproc** directive pair to delimit a section of your code that you want the assembly optimizer to optimize. This section is called a procedure. Use .proc at the beginning of the section and .endproc at the end of the section. In this way, you can set off sections of your assembly code that you want to be optimized, like functions. The directives must be used in pairs; do not use .proc without the corresponding .endproc. Specify a label with the .proc directive. You can have multiple procedures in a linear assembly file.

Use the optional *register* parameter in the .proc directive to indicate which registers are live in, and use the optional register parameter of the .endproc directive to indicate which registers are live out for each procedure. The *register* can be an actual register or a symbolic name. For example:

```
   .PROC x, A5, y, B7
...
   .ENDPROC y
```

A value is *live in* if it has been defined before the procedure and is used as an input to the procedure. A value is *live out* if it has been defined before or within the procedure and is used as an output from the procedure. If you do not specify any registers with the .endproc directive, it is assumed that no registers are live out.

Only code within procedures is optimized. The assembly optimizer copies any code that is outside of procedures to the output file and does not modify it.

Example

Here is a block move example in which .proc and .endproc are used:

```
move   .proc A4, B4, B0
       .no_mdep

loop:
       LDW    *B4++, A1
       MV     A1, B1
       STW    B1, *A4++
       ADD    -4, B0, B0
 [B0]  B      loop
       .endproc
```

The following types of instructions are not allowed in .proc or .cproc (see page 4-17 and 4-24) regions:

- Instructions that reference the stack pointer (register B15) are not allowed in a .proc or .cproc region. Stack space can be allocated by the assembly optimizer in a .proc or .cproc region for storage of temporary values. To allocate this storage area, the stack pointer is decremented on entry to the region and incremented on exit from the region. Since the stack pointer can change value on entry to the region, the assembly optimizer does not allow code that references the stack pointer register.

- Indirect branches are not allowed in a .proc or .cproc region so that the .proc or .cproc region exit protocols cannot be bypassed. Here is an example of an indirect branch:

```
   B   B4 <=  illegal
```

- Direct branches to labels not defined in the .proc or .cproc region are not allowed so that the .proc or .cproc region exit protocols cannot be bypassed. Here is an example of a direct branch outside of a .proc region:

```
   .proc
  ...
   B outside    <= illegal
   .endproc
outside:
```

- Direct branches to the label associated with a .proc directive are not allowed. If you require a branch back to the start of the linear assembly function, then use the .call directive. Here is an example of a direct branch to the label of a .proc directive:

```
_func: .proc
       ...
       B _func   <= illegal
       ...
       .endproc
```

- An .if/.endif loop must be entirely inside or outside of a proc or .cproc region. It is not alllowed to have part of an .if/.endif loop inside of a .proc or .cproc region and the other part of the .if/.endif loop outside of the .proc or .cproc region. Here are two examples of legal .if/.endif loops. The first loop is outside a .cproc region, the second loop is inside a .proc region:

```
.if
.cproc
...
.endproc
.endif

.proc
.if
....
.endif
.endproc
```

  Here are two examples of .if/.endif loops that are partly inside and partly outside of a .cproc or .proc region:

```
.if
.cproc
.endif
.endproc

.proc
.if
...
.else
.endproc
.endif
```

.reg — *Declare Variables*

Syntax

.reg *variable*$_1$ [**,** *variable*$_2$**,**…]

Description

The **.reg** directive allows you to use descriptive names for values that are stored in registers. The assembly optimizer chooses a register for you such that its use agrees with the functional units chosen for the instructions that operate on the value.

The .reg directive is valid within procedures only; that is, within occurrences of the .proc and .endproc directive pair or the .cproc and .endproc directive pair.

Declaring register pairs explicitly is optional. The assembly optimizer will derive when registers are used as pairs. Here is an example of declaring a register pair:

```
.reg  A6, A7
```

Example 1 This example uses the same code as the block move example on page 4-24 but the .reg directive is used:

```
move   .cproc dst, src, cnt

       .reg tmp1, tmp2

loop:
       LDW    *src++, tmp1
       MV     tmp1, tmp2
       STW    tmp2, *dst++
       ADD    -4, cnt, cnt
[cnt]  B      loop

       .endproc
```

Notice how this example differs from the .proc example on page 4-24: symbolic registers declared with .reg are allocated as machine registers.

Example 2 The code in the following example is invalid, because you cannot use a variable defined by the .reg directive with the .proc directive:

```
move   .proc dst, src, cnt ; WRONG: You cannot use a
       .reg dst, src, cnt      ; variable with .proc
```

This example could be corrected as follows:

```
move   .cproc dst, src, cnt
```

Example 3 The code in the following example is invalid, because a variable defined by the .reg directive cannot be used outside of the defined procedure:

```
move   .proc A4
       .reg tmp

       LDW    *A4++, tmp
       MV     tmp, B5

       .endproc

       MV tmp, B6    ; WRONG: tmp is invalid outside of
                     ; the procedure
```

.rega/.regb — *Partition Registers Directly*

Syntax

.rega *variable*$_1$ [**,** *variable*$_2$**,**…]
.regb *variable*$_1$ [**,** *variable*$_2$**,**…]

Description

Registers can be directly partitioned through two directives. The **.rega** directive is used to constrain a variable name to A-side registers. The **.regb** directive is used to constrain a variable name to B-side registers. For example:

```
.REGA y
.REGB u, v, w

MV   x, y
LDW *u, v:w
```

The .rega and .regb directives are valid within procedures only; that is, within occurrences of the .proc and .endproc directive pair or the .cproc and .endproc directive pair.

When a variable is declared with the .rega or .regb directive, it is not necessary to declare that variable with the .reg directive.

The old method of partitioning registers indirectly by partitioning instructions can still be used. Side and functional unit specifiers can still be used on instructions. However, functional unit specifiers (.L/.S/.D/.M) and crosspath information are ignored. Side specifiers are translated into partitioning constraints on the corresponding variable names, if any. For example:

```
MV  .1X     z, y     ; translated to .REGA y
LDW .D2T2 *u, v:w    ; translated to .REGB u, v, w
```

.reserve — *Reserve a Register*

Syntax

.reserve [*register*$_1$ **[,** *register*$_2$**,** …]]

Description

The **.reserve** directive prevents the assembly optimizer from using the specified *register* in a .proc or .cproc region.

If a .reserved register is explicitly assigned in a .proc or .cproc region, then the assembly optimizer can also use that register. For example, the variable tmp1 can be allocated to register A7, even though it is in the .reserve list, since A7 was explicitly defined in the ADD instruction:

```
.cproc
.reserve  a7
.reg      tmp1

....
ADD       a6, b4, a7
....
.endproc
```

Note: Reserving Registers A4 and A5

When inside of a .cproc region that contains a .call statement, A4 and A5 cannot be specified in a .reserve statement. The calling convention mandates that A4 and A5 are used as the return registers for a .call statement.

Example 1

The .reserve in this example guarantees that the assembly optimizer does not use A10 to A13 or B10 to B13 for the variables tmp1 to tmp5:

```
test   .proc  a4, b4
   .reg   tmp1, tmp2, tmp3, tmp4, tmp5
   .reserve a10, a11, a12, a13, b10, b11, b12, b13

   .....

   .endproc a4
```

Example 2

The assembly optimizer may generate less efficient code if the available register pool is overly restricted. In addition, it is possible that the available register pool is constrained such that allocation is not possible and an error message is generated. For example, the following code generates an error since all of the conditional registers have been reserved, but a conditional register is required for the variable tmp:

```
   .cproc ...
   .reserve a1,a2,b0,b1,b2
   .reg tmp

   ....
[tmp]  ....
   ....

   .endproc
```

.return — *Return a Value to a C Callable Procedure*

Syntax

.return [*argument*]

Description

The **.return** directive function is equivalent to the return statement in C/C++ code. It places the optional argument in the appropriate register for a return value as per the C/C++ calling conventions (see section 8.4 on page 8-19).

The optional *argument* can have the following meanings:

- Zero arguments implies a .cproc region that has no return value, similar to a void function in C/C++ code.
- An argument implies a .cproc region that has a 32-bit return value, similar to an int function in C/C++ code.

- A register pair of the format hi:lo implies a .cproc region that has a 40-bit long, a 64-bit long long, or a 64-bit type double return value; similar to a long/long long/double function in C/C++ code.

Arguments to the .return directive can be either symbolic register names or machine-register names.

All return statements in a .cproc region must be consistent in the type of the return value. It is not legal to mix a .return arg with a .return hi:lo in the same .cproc region.

The .return directive is unconditional. To perform a conditional .return, simply use a conditional branch around a .return. The assembly optimizer removes the branch and generates the appropriate conditional code. For example, to return if condition cc is true, code the return as:

```
[!cc] B  around
   .return
around:
```

Example

This example uses a symbolic register name, tmp, and a machine-register, A5, as .return arguments:

```
   .cproc ...
   .reg tmp

   ...
   .return tmp  <= legal symbolic name
   ...
   .return a5   <= legal actual name
```

.trip — *Specify Trip Count Values*

Syntax

label **.trip** *minimum value*, [*maximum value*[, *factor*]]

Description

The **.trip** directive specifies the value of the trip count. The *trip count* indicates how many times a loop iterates. The .trip directive is valid within procedures only. Following are descriptions of the .trip directive parameters:

label	The label represents the beginning of the loop. This is a required parameter.
minimum value	The minimum number of times that the loop can iterate. This is a required parameter. The default is 1.
maximum value	The maximum number of times that the loop can iterate. The maximum value is an optional parameter.

factor The factor used, along with *minimum value* and *maximum value*, to determine the number of times that the loop can iterate. In the following example, the loop executes some multiple of 8, between 8 and 48, times:

```
loop:  .trip 8, 48, 8
```

A *factor* of 2 states that your loop always executes an even number of times allowing the compiler to unroll once; this can result in a performance increase.

The factor is optional when the maximum value is specified.

You are not required to specify a .trip directive with every loop; however, you should use .trip if you know that a loop iterates some number of times. This generally means that redundant loops are not generated (unless the minimum value is really small) saving code size and execution time.

If you know that a loop always executes the same number of times whenever it is called, define maximum value (where maximum value equals minimum value) as well. The compiler may now be able to unroll your loop thereby increasing performance.

When you are compiling with the interrupt flexibility option (−mi*n*), using a .trip maximum value allows the compiler to determine the maximum number of cycles that the loop can execute. Then, the compiler compares that value to the threshold value given by the −mi option. See section 2.11, *Interrupt Flexibility Options (−mi Option)*, on page 2-43 for more information.

If the assembly optimizer cannot ensure that the trip count is large enough to pipeline a loop for maximum performance, a pipelined version and an unpipelined version of the same loop are generated. This makes one of the loops a *redundant loop*. The pipelined or the unpipelined loop is executed based on a comparison between the trip count and the number of iterations of the loop that can execute in parallel. If the trip count is greater or equal to the number of parallel iterations, the pipelined loop is executed; otherwise, the unpipelined loop is executed. For more information about redundant loops, see section 3.3 on page 3-16.

Example The .trip directive states that the loop will execute 16, 24, 32, 40 or 48 times when the w_vecsum routine is called.

```
w_vecsum:    .cproc  ptr_a, ptr_b, ptr_c, weight, cnt
             .reg    ai, bi, prod, scaled_prod, ci
             .no_mdep

loop:        .trip 16, 48, 8
             ldh     *ptr_a++, ai
             ldh     *ptr_b++, bi
             mpy     weight, ai, prod
             shr     prod, 15, scaled_prod
             add     scaled_prod, bi, ci
             sth     ci, *ptr_c++
  [cnt]      sub     cnt, 1, cnt
  [cnt]      b       loop
             .endproc
```

.volatile *Declare Memory References as Volatile*

Syntax **.volatile** *symbol*$_1$ [**,** *symbol*$_2$**,**…]

Description The **.volatile** directive allows you to designate memory references as volatile. Volatile loads and stores are not deleted. Volatile loads and stores are not reordered with respect to other volatile loads and stores.

If the .volatile directive references a memory location that may be modified during an interrupt, compile with the −mi1 option to ensure interruptibility of all code referencing the volatile memory location.

Example The st and ld memory references are designated as volatile.

```
.volatile st, ld

STW  W, *X{st}          ; volatile store
STW  U, *V
LDW *Y{ld}, Z           ; volatile load
```

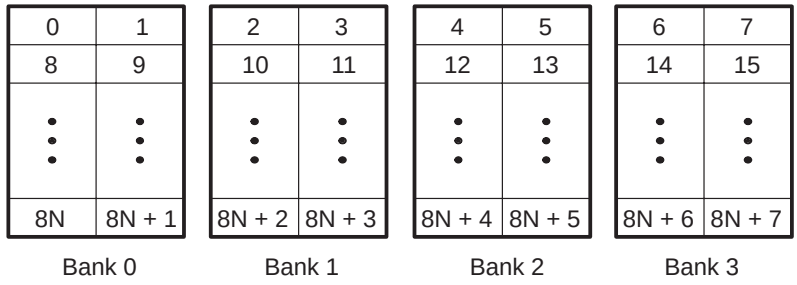
4.5 Avoiding Memory Bank Conflicts With the Assembly Optimizer

The internal memory of the C6000 family varies from device to device. See the appropriate device data sheet to determine the memory spaces in your particular device. This section discusses how to write code to avoid memory bank conflicts.

Most C6000 devices use an interleaved memory bank scheme, as shown in Figure 4−1. Each number in the diagram represents a byte address. A load byte (LDB) instruction from address 0 loads byte 0 in bank 0. A load halfword (LDH) from address 0 loads the halfword value in bytes 0 and 1, which are also in bank 0. A load word (LDW) from address 0 loads bytes 0 through 3 in banks 0 and 1.

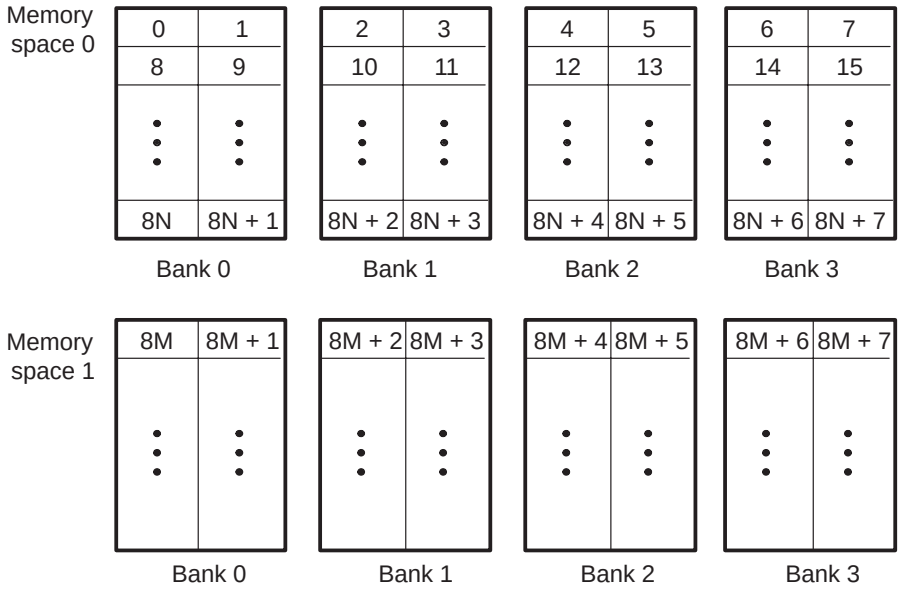
Because each bank is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle while the second value is read from memory. Two memory operations per cycle are allowed without any stall, as long as they do not access the same bank.

Figure 4−1. 4-Bank Interleaved Memory



For devices that have more than one memory space (Figure 4−2), an access to bank 0 in one memory space does not interfere with an access to bank 0 in another memory space, and no pipeline stall occurs.

Figure 4−2. 4-Bank Interleaved Memory With Two Memory Spaces



4.5.1 Preventing Memory Bank Conflicts

The assembly optimizer uses the assumptions that memory operations do not have bank conflicts. If it determines that two memory operations have a bank conflict on any loop iteration it does *not* schedule the operations in parallel. The assembly optimizer checks for memory bank conflicts only for those loops that it is trying to software pipeline.

The information required for memory bank analysis indicates a base, an offset, a stride, a width, and an iteration delta. The width is implicitly determined by the type of memory access (byte, halfword, word, or double word for the ’C6400 and ’C6700). The iteration delta is determined by the assembly optimizer as it constructs the schedule for the software pipeline. The base, offset, and stride are supplied by the load and store instructions and/or by the .mptr directive.

An LD(B/BU)(H/HU)(W) or ST(B/H/W) operation in linear assembly can have memory bank information associated with it implicitly, by using the .mptr directive. The .mptr directive associates a register with the information that allows the assembly optimizer to determine automatically whether two memory operations have a bank conflict. If the assembly optimizer determines that two memory operations have a memory bank conflict, then it does not schedule them in parallel. The syntax is:

.mptr *register***,** *base***+***offset***,** *stride*

For example:

```
.mptr a_0,a+0,16
.mptr a_4,a+4,16

LDW *a_0++[4], val1  ; base=a, offset=0, stride=16
LDW *a_4++[4], val2  ; base=a, offset=4, stride=16

.mptr dptr,D+0,8

LDH *dptr++, d0      ; base=D, offset=0, stride=8
LDH *dptr++, d1      ; base=D, offset=2, stride=8
LDH *dptr++, d2      ; base=D, offset=4, stride=8
LDH *dptr++, d3      ; base=D, offset=6, stride=8
```

In this example, the offset for dptr is updated after every memory access. The offset is updated only when the pointer is modified by a constant. This occurs for the pre/post increment/decrement addressing modes.

See page 4-21 for information about the .mptr directive.

Example 4−4 shows loads and stores extracted from a loop that is being software pipelined.

Example 4−4. Load and Store Instructions That Specify Memory Bank Information

```
        .mptr   Ain,IN,-16
        .mptr   Bin,IN-4,-16

        .mptr   Aco,COEF,16
        .mptr   Bco,COEF+4,16

        .mptr   Aout,optr+0,4
        .mptr   Bout,optr+2,4

        LDW           *Ain--[2],Ain12           ; IN(k-i) & IN(k-i+1)
        LDW           *Bin--[2],Bin23           ; IN(k-i-2) & IN(k-i-1)
        LDW           *Ain--[2],Ain34           ; IN(k-i-4) & IN(k-i-3)
        LDW           *Bin--[2],Bin56           ; IN(k-i-6) & IN(k-i-5)

        LDW           *Bco++[2],Bco12           ; COEF(i) & COEF(i+1)
        LDW           *Aco++[2],Aco23           ; COEF(i+2) & COEF(i+3)
        LDW           *Bco++[2],Bin34           ; COEF(i+4) & COEF(i+5)
        LDW           *Aco++[2],Ain56           ; COEF(i+6) & COEF(i+7)

        STH           Assum,*Aout++[2]          ; *oPtr++ = (r >> 15)
        STH           Bssum,*Bout++[2]          ; *oPtr++ = (i >> 15)
```

4.5.2 A Dot Product Example That Avoids Memory Bank Conflicts

The C code in Example 4−5 implements a dot product function. The inner loop is unrolled once to take advantage of the C6000's ability to operate on two 16-bit data items in a single 32-bit register. LDW instructions are used to load two consecutive short values. The linear assembly instructions in Example 4−6 implement the dotp loop kernel. Example 4−7 shows the loop kernel determined by the assembly optimizer.

For this loop kernel, there are two restrictions associated with the arrays a[] and b[]:

- Because LDW is being used, the arrays must be be aligned to start on word boundaries.
- To avoid a memory bank conflict, one array must start in bank 0 and the other array in bank 2. If they start in the same bank, then a memory bank conflict occurs every cycle and the loop computes a result every two cycles instead of every cycle, due to a memory bank stall. For example:

  Bank conflict:

```
     MVK   0, A0
  || MVK   8, B0
     LDW   *A0, A1
  || LDW   *B0, B1
```

  No bank conflict:

```
     MVK   0, A0
  || MVK   4, B0
     LDW   *A0, A1
  || LDW   *B0, B1
```

Example 4−5. C Code for Dot Product

```
int dotp(short a[], short b[])
{
    int sum0 = 0, sum1 = 0, sum, i;

    for (i = 0; i < 100/2; i+= 2)
    {
        sum0 += a[i] * b[i];
        sum1 += a[i + 1] * b[i + 1];
    }
    return sum0 + sum1;
}
```

Example 4−6. Linear Assembly for Dot Product

```
_dotp:  .cproc  a, b
        .reg    sum0, sum1, i
        .reg    val1, val2, prod1, prod2

        MVK     50,i ; i = 100/2
        ZERO    sum0 ; multiply result = 0
        ZERO    sum1 ; multiply result = 0

loop:   .trip 50
        LDW     *a++,val1         ; load a[0-1]  bank0
        LDW     *b++,val2         ; load b[0-1]  bank2
        MPY     val1,val2,prod1   ; a[0] * b[0]
        MPYH    val1,val2,prod2   ; a[1] * b[1]
        ADD     prod1,sum0,sum0   ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1   ; sum1 += a[1] * b[1]

    [i] ADD     -1,i,i            ; i--
    [i] B       loop              ; if (!i) goto loop

        ADD     sum0,sum1,A4      ; compute final result
        .return A4
        .endproc
```

Example 4−7. Dot Product Software-Pipelined Kernel

```
L2:    ; PIPED LOOP KERNEL

           ADD    .L2     B7,B4,B4           ; |14| <0,7>  sum0 += a[0]*b[0]
||         ADD    .L1     A5,A0,A0           ; |15| <0,7>  sum1 += a[1]*b[1]
||         MPY    .M2X    B6,A4,B7           ; |12| <2,5>  a[0] * b[0]
||         MPYH   .M1X    B6,A4,A5           ; |13| <2,5>  a[1] * b[1]
|| [ B0]  B      .S1     L2                 ; |18| <5,2>  if (!i) goto loop
|| [ B0]  ADD    .S2     0xffffffff,B0,B0   ; |17| <6,1>  i--
||         LDW    .D2T2   *B5++,B6           ; |10| <7,0>  load a[0-1] bank0
||         LDW    .D1T1   *A3++,A4           ; |11| <7,0>  load b[0-1] bank2
```

It is not always possible to control fully how arrays and other memory objects are aligned. This is especially true when a pointer is passed into a function and that pointer may have different alignments each time the function is called. A solution to this problem is to write a dot product routine that cannot have memory hits. This would eliminate the need for the arrays to use different memory banks.

If the dot product loop kernel is unrolled once, then four LDW instructions execute in the loop kernel. Assuming that nothing is known about the bank alignment of arrays a and b (except that they are word aligned), the only safe assumptions that can be made about the array accesses are that a[0−1] cannot conflict with a[2−3] and that b[0−1] cannot conflict with b[2−3]. Example 4−8 shows the unrolled loop kernel.

Example 4−8. Dot Product From Example 4−6 Unrolled to Prevent Memory Bank Conflicts

```
_dotp2: .cproc   a_0, b_0
        .reg    a_4, b_4, sum0, sum1, i
        .reg    val1, val2, prod1, prod2

        ADD     4,a_0,a_4
        ADD     4,b_0,b_4
        MVK     25,i    ; i = 100/4
        ZERO    sum0    ; multiply result = 0
        ZERO    sum1    ; multiply result = 0

        .mptr   a_0,a+0,8
        .mptr   a_4,a+4,8
        .mptr   b_0,b+0,8
        .mptr   b_4,b+4,8

loop:   .trip   25
        LDW     *a_0++[2],val1  ; load a[0-1] bankx
        LDW     *b_0++[2],val2  ; load b[0-1] banky
        MPY     val1,val2,prod1 ; a[0] * b[0]
        MPYH    val1,val2,prod2 ; a[1] * b[1]
        ADD     prod1,sum0,sum0 ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1 ; sum1 += a[1] * b[1]

        LDW     *a_4++[2],val1  ; load a[2-3] bankx+2
        LDW     *b_4++[2],val2  ; load b[2-3] banky+2
        MPY     val1,val2,prod1 ; a[2] * b[2]
        MPYH    val1,val2,prod2 ; a[3] * b[3]
        ADD     prod1,sum0,sum0 ; sum0 += a[2] * b[2]
        ADD     prod2,sum1,sum1 ; sum1 += a[3] * b[3]

    [i] ADD     -1,i,i          ; i--
    [i] B       loop            ; if (!0) goto loop

        ADD     sum0,sum1,A4    ; compute final result
        .return A4
        .endproc
```

The goal is to find a software pipeline in which the following instructions are in parallel:

```
   LDW *a0++[2],val1 ; load a[0-1]  bankx
|| LDW *a2++[2],val2 ; load a[2-3]  bankx+2

   LDW *b0++[2],val1 ; load b[0-1]  banky
|| LDW *b2++[2],val2 ; load b[2-3]  banky+2
```

Example 4−9. Unrolled Dot Product Kernel From Example 4−7

```
L2:    ; PIPED LOOP KERNEL

   [ B1]  SUB    .S2    B1,1,B1       ; <0,8>
||        ADD    .L2    B9,B5,B9      ; |21| <0,8>  ^  sum0 += a[0] * b[0]
||        ADD    .L1    A6,A0,A0      ; |22| <0,8>  ^  sum1 += a[1] * b[1]
||        MPY    .M2X   B8,A4,B9      ; |19| <1,6>  a[0] * b[0]
||        MPYH   .M1X   B8,A4,A6      ; |20| <1,6>  a[1] * b[1]
|| [ B0]  B      .S1    L2            ; |32| <2,4>  if (!i) goto loop
|| [ B1]  LDW    .D1T1  *A3++(8),A4   ; |24| <3,2>  load a[2-3] bankx+2
|| [ A1]  LDW    .D2T2  *B6++(8),B8   ; |17| <4,0>  load a[0-1] bankx

   [ A1]  SUB    .S1    A1,1,A1       ; <0,9>
||        ADD    .L2    B5,B9,B5      ; |28| <0,9>  ^  sum0 += a[2] * b[2]
||        ADD    .L1    A6,A0,A0      ; |29| <0,9>  ^  sum1 += a[3] * b[3]
||        MPY    .M2X   A4,B7,B5      ; |26| <1,7>  a[2] * b[2]
||        MPYH   .M1X   A4,B7,A6      ; |27| <1,7>  a[3] * b[3]
|| [ B0]  ADD    .S2    -1,B0,B0      ; |31| <3,3>  i--
|| [ A1]  LDW    .D2T2  *B4++(8),B7   ; |25| <4,1>  load b[2-3] banky+2
|| [ A1]  LDW    .D1T1  *A5++(8),A4   ; |18| <4,1>  load b[0-1] banky
```

Without the .mptr directives in Example 4−8, the loads of a[0−1] and b[0−1] are scheduled in parallel, and the loads of a[2−3] and b[2−3] are scheduled in parallel. This results in a 50% chance that a memory conflict will occur on every cycle. However, the loop kernel shown in Example 4−9 can never have a memory bank conflict.

In Example 4−6, if .mptr directives had been used to specify that a and b point to different bases, then the assembly optimizer would never find a schedule for a 1-cycle loop kernel, because there would always be a memory bank conflict. However, it would find a schedule for a 2-cycle loop kernel.

4.5.3 Memory Bank Conflicts for Indexed Pointers

When determining memory bank conflicts for indexed memory accesses, it is sometimes necessary to specify that a pair of memory accesses always conflict, or that they never conflict. This can be accomplished by using the .mptr directive with a stride of 0.

A stride of 0 indicates that there is a constant relation between the memory accesses regardless of the iteration delta. Essentially, only the base, offset, and width are used by the assembly optimizer to determine a memory bank conflict. Recall that the stride is optional and defaults to 0.

In Example 4−10, the .mptr directive is used to specify which memory accesses conflict and which never conflict.

Example 4−10. Using .mptr for Indexed Pointers

```
.mptr  a,RS
.mptr  b,RS

.mptr  c,XY
.mptr  d,XY+2

LDW     *a++[i0a],A0  ; a and b always conflict with each other
LDW     *b++[i0b],B0  ;

STH     A1,*c++[i1a]    ; c and d never conflict with each other
STH     B2,*d++[i1b]    ;
```

4.5.4 Memory Bank Conflict Algorithm

The assembly optimizer uses the following process to determine if two memory access instructions might have a memory bank conflict:

1) If either access does not have memory bank information, then they do not conflict.

2) If both accesses do not have the same base, then they conflict.

3) The offset, stride, access width, and iteration delta are used to determine if a memory bank conflict will occur. The assembly optimizer uses a straightforward analysis of the access patterns and determines if they ever access the same relative bank. The stride and offset values are always expressed in bytes.

   The iteration delta is the difference in the loop iterations of the memory references being scheduled in the software pipeline. For example, given three instructions A, B, C and a software pipeline with a single-cycle kernel, then A and C have an iteration delta of 2:

```
A
B   A
C   B   A
    C   B
        C
```

4.6 Memory Alias Disambiguation

Memory aliasing occurs when two instructions can access the same memory location. Such memory references are called ambiguous. Memory alias disambiguation is the process of determining when such ambiguity is not possible. When you cannot determine whether two memory references are ambiguous, you presume they are ambiguous. This is the same as saying the two instructions have a memory dependence between them.

Dependences between instructions constrain the instruction schedule, including the software pipeline schedule. In general, the fewer the dependences, the greater freedom you have in choosing a schedule and the better the final schedule performs.

4.6.1 How the Assembly Optimizer Handles Memory References (Default)

The assembly optimizer assumes all memory references are always aliased; they always depend on one another. This presumption is safe for all possible input. This gives you complete control over how possible memory aliases are to be handled.

In some cases, this presumption is overly conservative. In such cases, the extra instruction dependences, due to the presumed memory aliases, can cause the assembly optimizer to emit instruction schedules that have less parallelism and do not perform well. To handle these cases, the assembly optimizer provides one option and two directives.

4.6.2 Using the −mt Option to Handle Memory References

In the assembly optimizer, the −mt option means no memory references ever depend on each other. The −mt option does not mean the same thing to the compiler. The compiler interprets the −mt switch to indicate several specific cases of memory aliasing are guaranteed not to occur. For more information about using the −mt option, see section 3.7.2, page 3-26.

4.6.3 Using the .no_mdep Directive

You can specify the .no_mdep directive anywhere in a .(c)proc function. Whenever it is used, you guarantee that no memory dependences occur within that function.

Note: Memory Dependency Exception

For both of these methods, −mt and .no_mdep, the assembly optimizer recognizes any memory dependences the user points out with the .mdep directive.

4.6.4 Using the .mdep Directive to Identify Specific Memory Dependences

You can use the .mdep directive to identify specific memory dependences by annotating each memory reference with a name, and using those names with the .mdep directive to indicate the actual dependence. Annotating a memory reference requires adding information right next to the memory reference in the assembly stream. Include the following Immediately after a memory reference:

{ *symbol* }

The symbol has the same syntax restrictions as any assembly symbol. (For more information about symbols, see the *TMS320C6000 Assembly Language Tools User's Guide.)* It is in the same name space as the symbolic registers. You cannot use the same name for a symbolic register and annotating a memory reference.

Example 4−11. Annotating a Memory Reference

```
LDW    *p1++ {ld1}, inp1  ;name memory reference "ld1"
;other code ...
STW    outp2, *p2++ {st1} ;name memory reference "st1"
```

The directive to indicate a specific memory dependence in the previous example is as follows:

```
.mdep ld1, st1
```

This means that whenever ld1 accesses memory at location X, some later time in code execution st1 may also access location X. This is equivalent to adding a dependence between these two instructions. In terms of the software pipeline, these two instructions must remain in the same order. The ld1 reference must always occur before the st1 reference; the instructions cannot even be scheduled in parallel.

It is important to note the directional sense of the directive from ld1 to st1. The opposite, from st1 to ld1, is not implied. In terms of the software pipeline, while every ld1 must occur before every st1, it is still legal to schedule the ld1 from iteration n+1 before the st1 from iteration n.

Example 4−12 is a picture of the software pipeline with the instructions from two different iterations in different columns. In the actual instruction sequence, instructions on the same horizontal line are in parallel.

Example 4−12. Software Pipeline Using .mdep ld1, st1

```
      iteration n         iteration n+1
      -------------       -------------
      LDW { ld1 }

      ...                 LDW { ld1 }

      STW { st1 }         ...

                          STW { st1 }
```

If that schedule does not work because the iteration n st1 might write a value the iteration n+1 ld1 should read, then you must note a dependence relationship from st1 to ld1.

```
.mdep   st1, ld1
```

Both directives together force the software pipeline shown in Example 4−13.

Example 4−13. Software Pipeline Using .mdep st1, ld1 and .mdep ld1, st1

```
      iteration n         iteration n+1
      -------------       -------------
      LDW { ld1 }

      ...

      STW { st1 }

                          LDW { ld1 }

                          ...

                          STW { st1 }
```

Indexed addressing, *+base[index], is a good example of an addressing mode where you typically do not know anything about the relative sequence of the memory accesses, except they sometimes access the same location. To correctly model this case, you need to note the dependence relation in both directions, and you need to use both directives.

```
.mdep   ld1, st1
.mdep   st1, ld1
```

4.6.5 Memory Alias Examples

Following are memory alias examples that use the .mdep and .no_mdep directives.

❑ **Example 1**

The .mdep r1, r2 directive declares that LDW must be before STW. In this case, src and dst might point to the same array.

```
fn:    .cproc    dst, src, cnt
       .reg      tmp
       .no_mdep
       .mdep     r1, r2

       LDW       *src{r1}, tmp
       STW       cnt, *dst{r2}

       .return   tmp
       .endproc
```

❑ **Example 2**

Here, .mdep r2, r1 indicates that STW must occur before LDW. Since STW is after LDW in the code, the dependence relation is across loop iterations. The STW instruction writes a value that may be read by the LDW instruction on the next iteration. In this case, a 6-cycle recurrence is created.

```
fn:    .cproc    dst, src, cnt
       .reg      tmp
       .no_mdep
       .mdep     r2, r1

LOOP:  .trip     100
       LDW       *src++{r1}, tmp
       STW       tmp, *dst++{r2}
 [cnt] SUB       cnt, 1, cnt
 [cnt] B         LOOP

       .endproc
```

Note: Memory Dependence/Bank Conflict

Do not confuse the topic of memory alias disambiguation with the handling of memory bank conflicts. They may seem similar because they each deal with memory references and the effect of those memory references on the instruction schedule. Alias disambiguation is a correctness issue, bank conflicts are a performance issue. A memory dependence has a much broader impact on the instruction schedule than a bank conflict. It is best to keep these two topics separate.

Chapter 5

Linking C/C++ Code

The C/C++ compiler and assembly language tools provide two methods for linking your programs:

- You can compile individual modules and link them together. This method is especially useful when you have multiple source files.
- You can compile and link in one step. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/C++ code, including the run-time-support libraries, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see the *TMS320C6000 Assembly Language Tools User's Guide*.

Topic **Page**

5.1 Invoking the Linker Through the Compiler (-z Option)

This section explains how to invoke the linker after you have compiled and assembled your programs: as a separate step or as part of the compile step.

5.1.1 Invoking the Linker as a Separate Step

This is the general syntax for linking C/C++ programs as a separate step:

cl6x −z {**−c**|**−cr**} *filenames* [*options*] [**−o** *name*.out] **−l** *library* [*lnk.cmd*]

cl6x -z — The command that invokes the linker.

−c | **−cr** — Options that tell the linker to use special conventions defined by the C/C++ environment. When you use cl6x −z, you must use **−c** or **−cr**. The −c option uses automatic variable initialization at run time; the −cr option uses variable initialization at load time.

filenames — Names of object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the −o option to name the output file.

options — Options affect how the linker handles your object files. Linker options can only appear after the **−z** option on the command line, but may otherwise be in any order. (Options are discussed in section 5.2 on page 5-5.)

−o *name.out* — Names the output file.

−l *library* — (lowercase L) Identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions, or linker command files. If you are linking C/C++ code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter.

lnk.cmd — Contains options, filenames, directives, or commands for the linker.

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For information, see the *TMS320C6000 Assembly Language Tools User's Guide*.

You can link a C/C++ program consisting of modules prog1.obj, prog2.obj, and prog3.obj, with an executable filename of prog.out with the command:

```
cl6x –z –c prog1 prog2 prog3 –o prog.out –l rts6200.lib
```

5.1.2 Invoking the Linker as Part of the Compile Step

This is the general syntax for linking C/C++ programs as part of the compile step:

cl6x *filenames* [*options*] **–z** {**–c**|**–cr**} *filenames* [*options*] [**–o** *name*.out] –l *library* [*lnk.cmd*]

The **–z** option divides the command line into the compiler options (the options before –z) and the linker options (the options following –z). The –z option must follow all source files and compiler options on the command line.

All arguments that follow –z on the command line are passed to the linker. These arguments can be linker command files, additional object files, linker options, or libraries. These arguments are the same as described in section 5.1.1, *Invoking the Linker As a Separate Step*.

All arguments that precede –z on the command line are compiler arguments. These arguments can be C/C++ source files, assembly files, linear assembly files, or compiler options. These arguments are described in section 2.2, *Invoking the C/C++ Compiler*, on page 2-4.

You can compile and link a C/C++ program consisting of modules prog1.c, prog2.c, and prog3.c, with an executable filename of prog.out with the command:

```
cl6x prog1.c prog2.c prog3.c –z –c –o prog.out –l rts6200.lib
```

Note: Order of Processing Arguments in the Linker

The order in which the linker processes arguments is important. The compiler passes arguments to the linker in the following order:

1) Object filenames from the command line
2) Arguments following the –z option on the command line
3) Arguments following the –z option from the C_OPTION or C6X_C_OPTION environment variable

5.1.3 Disabling the Linker (−c Compiler Option)

You can override the −z option by using the −c compiler option. The −c option is especially helpful if you specify the −z option in the C_OPTION or C6X_C_OPTION environment variable and want to selectively disable linking with the −c option on the command line.

The −c linker option has a different function than, and is independent of, the −c compiler option. By default, the compiler uses the −c linker option when you use the −z option. This tells the linker to use C/C++ linking conventions (autoinitialization of variables at run time). If you want to initialize variables at load time, use the −cr linker option following the −z option.

5.2 Linker Options

All command-line input following the −z option is passed to the linker as parameters and options. Following are the options that control the linker, along with detailed descriptions of their effects.

−a Produces an absolute, executable module. This is the default; if neither −a nor −r is specified, the linker acts as if −a is specified.

−abs Produces an absolute listing file.

−ar Produces a relocatable, executable object module. The output module contains the special linker symbols, an optional header, and all symbol references. The relocation information is retained.

−−args=*size* Allocates memory to be used by the loader to pass arguments from the command line of the loader to the program. The linker allocates *size* bytes in an uninitialized .args section. The __c_args__ symbol contains the address of the .args section.

−b Disables merge of symbolic debugging information. The linker keeps the duplicate entries of symbolic debugging information commonly generated when a C program is compiled for debugging.

−c Autoinitializes variables at run time. See section 8.8.4 on page 8-56, for more information.

−cr Initializes variables at load time. See section 8.8.5 on page 8-57, for more information.

−e *global_symbol* Defines a *global_symbol* that specifies the primary entry point for the output module

−f *fill_value* Sets the default fill value for null areas within output sections; *fill_value* is a 32-bit constant

−g *global_symbol* Defines *global_symbol* as global even if the global symbol has been made static with the −h linker option

−h Makes all global symbols static; global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

−heap *size*	Sets the heap size (for dynamic memory allocation) to *size* bytes and defines a global symbol that specifies the heap size. The default is 1K bytes.
−I *directory*	Alters the library-search algorithm to look in *directory* before looking in the default location. This option must appear before the −l linker option. The directory must follow operating system conventions. You can specify up to 128 −I options.
−j	Disables conditional linking that has been set up with the assembler .clink directive. By default, all sections are unconditionally linked.
−l *libraryname*	(lower case L) Names an archive library file or linker command filename as linker input. The *libraryname* is an archive library name and must follow operating system conventions.
−m *filename*	Produces a map or listing of the input and output sections, including null areas, and places the listing in *filename.* The filename must follow operating system conventions.
−o *filename*	Names the executable output module. The *filename* must follow operating system conventions. If the −o option is not used, the default filename is a.out.
−q	Requests a quiet run (suppresses the banner)
−priority	Satisfies each unresolved reference by the first library that contains a definition for that symbol
−r	Retains relocation entries in the output module.
−s	Creates a smaller output section by stripping symbol table information and line number entries from the output module.
−stack *size*	Sets the C/C++ system stack size to *size* bytes and defines a global symbol that specifies the stack size. The default is 1K bytes.

−−trampolines Generates a trampoline code section for each call that is linked out-of-range of its called destination. The trampoline code section contains a sequence of instructions that performs a transparent long branch to the original called address. Each calling instruction that is out-of-range from the called function is redirected to the trampoline.

−u *symbol* Places the unresolved external symbol *symbol* into the output module's symbol table. This forces the linker to search a library and include the member that defines the symbol.

−w Displays a message when the linker encounters one or more input sections that do not have a corresponding output section defined in the the SECTIONS directive.

−x Forces rereading of libraries. Linker continues to reread libraries until no more references can be resolved.

−−xml_link_info *file* Generates an XML link information file. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

For more information on linker options, see the *Linker Description* chapter in the *TMS320C6000Assembly Language Tools User's Guide.*

5.3 Controlling the Linking Process

Regardless of the method you choose for invoking the linker, special requirements apply when linking C/C++ programs. You must:

- ❏ Include the compiler's run-time-support library
- ❏ Specify the type of initialization
- ❏ Determine how you want to allocate your program into memory

This section discusses how these factors are controlled and provides an example of the standard default linker command file.

For more information about how to operate the linker, see the linker description in the *TMS320C6000 Assembly Language Tools User's Guide.*

5.3.1 Linking With Run-Time-Support Libraries

You must link all C/C++ programs with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. The run-time-support library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. You must use the −l linker option to specify which C6000 run-time-support library to use. The −l option also tells the linker to look at the `-I` options and then the C_DIR or C6X_C_DIR environment variable to find an archive path or object file. To use the −l linker option, type on the command line:

cl6x -z{**−c** | **−cr**} *filenames* **−l** *libraryname*

Generally, you should specify the library as the last name on the command line because the linker searches libraries for unresolved references in the order that files are specified on the command line. If any object files follow a library, references from those object files to that library are not resolved. You can use the −x option to force the linker to reread all libraries until references are resolved. Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

By default, if a library introduces an unresolved reference and multiple libraries have a definition for it, then the definition from the same library that introduced the unresolved reference is used. Use the -priority option if you want the linker to use the definition from the first library on the command line that contains the definition.

5.3.2 Run-Time Initialization

You must link all C/C++ programs with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. When a C/C++ program begins running, it must execute boot.obj first. The boot.obj module contains code and data to initialize the run-time environment; the linker automatically extracts boot.obj and links it when you use −c and include the appropriate run-time-support library in the link.

The archive libraries listed below contain C/C++ run-time-support functions:

rts6200.lib	rts6400.lib	rts6700.lib
rts6200e.lib	rts6400e.lib	rts6700e.lib

The boot.obj module contains code and data for initializing the run-time environment. The module performs the following tasks:

1) Sets up the stack and configuration registers
2) Processes the *.cinit* run-time initialization table and autoinitializes global variables (when using the −c option)
3) Calls all global constructors (*.pinit*)
4) Calls main
5) Calls exit when main returns

The run-time-support object libraries contain boot.obj. You can:

- ❏ Use the archiver to extract boot.obj from the library and then link the module in directly
- ❏ Include the appropriate run-time-support library as an input file (the linker automatically extracts boot.obj when you use the −c or −cr option)

A sample bootstrap routine is _c_int00, provided in boot.obj in the run-time support object libraries. The *entry point* is usually set to the starting address of the bootstrap routine.

Chapter 9 describes additional run-time-support functions that are included in the library. These functions include ISO C standard run-time support.

Note: The _c_int00 Symbol

One important function contained in the run-time support library is _c_int00. The symbol _c_int00 is the starting point in boot.obj; if you use the −c or −cr linker option, _c_int00 is automatically defined as the entry point for the program. If your program begins running from reset, you should set up the reset vector to branch to _c_int00 so that the processor executes boot.obj first.

5.3.3 Global Object Constructors

Global C++ variables having constructors and destructors require their constructors to be called during program initialization and their destructors to be called during program termination. The C/C++ compiler produces a table of constructors to be called at startup.

The table is contained in a named section called .pinit. The constructors are invoked in the order that they occur in the table.

Global constructors are called after initialization of other global variables and before main( ) is called. Global destructors are invoked during exit( ), similar to functions registered through atexit( ).

Section 8.8.3, Initialization Tables, on page 8-53 discusses the format of the .pinit table.

5.3.4 Specifying the Type of Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 8.8.3, *Initialization Tables*, on page 8-53 discusses the format of these tables. These tables are in a named section called *.cinit*. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the −c linker option (see section 8.8.4, *Autoinitialization of Variables at Run Time*, on page 8-56).
- Global variables are initialized at *load time*. Use the −cr linker option (see section 8.8.5, *Initialization of Variables at Load time*, on page 8-57).

When you link a C/C++ program, you must use either the −c or −cr linker option. These options tell the linker to select initialization at run time or load time.

When you compile and link programs, the −c linker option is the default. If used, the −c linker option must follow the −z option. (See section 5.1). The following list outlines the linking conventions used with −c or −cr:

- The symbol _c_int00 is defined as the program entry point; it identifies the beginning of the C/C++ boot routine in boot.obj. When you use −c or −cr, _c_int00 is automatically referenced, ensuring that boot.obj is automatically linked in from the run-time-support library.
- The *.cinit* output section is padded with a termination record so that the loader (load time initialization) or the boot routine (run-time initialization) knows when to stop reading the initialization tables.

- When using initialization at load time (the −cr linker option), the following occur:
  - The linker sets the symbol cinit to −1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.
- When autoinitializing at run time (−c linker option), the linker defines the symbol cinit as the starting address of the .cinit section. The boot routine uses this symbol as the starting point for autoinitialization.

5.3.5 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 5−1 summarizes the sections.

Table 5−1. Sections Created by the Compiler

(a) Initialized sections

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and contain string literals
.pinit	Table of constructors to be called at startup
.switch	Jump tables for large switch statements
.text	Executable code and constants

(b) Uninitialized sections

Name	Contents
.bss	Global and static variables
.far	Global and static variables declared far
.stack	Stack
.sysmem	Memory for malloc functions (heap)

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM. With the exception of .text, the initialized and uninitialized sections created by the compiler cannot be allocated into internal program memory. See section 8.1.1, on page 8-2 for a complete description of how the compiler uses these sections.

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the linker chapter in the *TMS320C6000 Assembly Language Tools User's Guide*.

5.3.6 A Sample Linker Command File

Example 5−1 shows a typical linker command file that links a C program. The command file in this example is named lnk.cmd and lists several linker options:

−c	Tells the linker to use autoinitialization at run time.
−heap	Tells the linker to set the C heap size at 0x2000 bytes.
−stack	Tells the linker to set the stack size to 0x0100 bytes.
−l	Tells the linker to use an archive library file, rts6200.lib, for input.

To link the program, use the following syntax:

cl6x −z *object_file(s)* **−o** *outfile* **−m** *mapfile* **lnk.cmd**

The MEMORY and possibly the SECTIONS directives, might require modification to work with your system. See the *TMS320C6000 Assembly Language Tools User's Guide* for more information on these directives.

Example 5−1. Sample Linker Command File

```
−c
−heap 0x2000
−stack 0x0100
−l rts6200.lib

MEMORY
{
    VECS:   o = 00000000h       l = 00400h /* reset & interrupt vectors     */
    PMEM:   o = 00000400h       l = 0FC00h /* intended for initialization   */
    BMEM:   o = 80000000h       l = 10000h /* .bss, .sysmem, .stack, .cinit */
}

SECTIONS
{
    vectors     >       VECS
    .text       >       PMEM
    .data       >       BMEM
    .stack      >       BMEM
    .bss        >       BMEM
    .sysmem     >       BMEM
    .cinit      >       BMEM
    .const      >       BMEM
    .cio        >       BMEM
    .far        >       BMEM
}
```

5.3.7 Using Function Subsections (−mo Compiler Option)

When the linker places code into an executable file, it allocates all the functions in a single source file as a group. This means that if any function in a file needs to be linked into an executable, then all the functions in the file are linked in. This can be undesirable if a file contains many functions and only a few are required for an executable.

This situation may exist in libraries where a single file contains multiple functions, but the application only needs a subset of those functions. An example is a library .obj file that contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. By default, both the signed and unsigned routines are linked in since they exist in the same .obj file.

The −mo compiler option remedies this problem by placing each function in a file in its own subsection. Thus, only the functions that are referenced in the application are linked into the final executable. This can result in an overall code size reduction.

However, be aware that using the −mo compiler option can result in overall code size growth if all or nearly all functions are being referenced. This is because any section containing code must be aligned to a 32-byte boundary to support the C6000 branching mechanism. When the −mo option is not used, all functions in a source file are usually placed in a common section which is aligned. When −mo is used, each function defined in a source file is placed in a unique section. Each of the unique sections requires alignment. If all the functions in the file are required for linking, code size may increase due to the additional alignment padding for the individual subsections.

Thus, the −mo compiler option is advantageous for use with libraries where normally only a limited number of the functions in a file are used in any one executable.

The alternative to the −mo option is to place each function in its own file.

Chapter 6

Using the Stand-Alone Simulator

The TMS320C6000 stand-alone simulator loads and runs an executable COFF .out file. When used with the C I/O libraries, the stand-alone simulator supports all C I/O functions with standard output to the screen.

The stand-alone simulator gives you a way to gather statistics about your program using the clock function. Additional benefits are that the stand-alone simulator can be used in a batch file and is included in the code generation tools.

This chapter describes how to invoke the stand-alone simulator. It also provides an example of C code and the stand-alone simulator results.

Topic **Page**

6.1 Invoking the Stand-Alone Simulator

This section shows how to invoke the stand-alone simulator to load and run an executable COFF .out file. This is the general syntax for invoking the stand-alone simulator:

load6x [*options*] *filename.out*

load6x	The command that invokes the stand-alone simulator.
options	Options affect how the stand-alone simulator acts and how it handles your .out file. Options can appear anywhere on the command line. (Options are discussed in section 6.2, *Stand-Alone Simulator Options*.)
filename.out	Names the .out file to be loaded into the stand-alone simulator. The .out file must be an executable COFF file.

The stand-alone simulator can run C6200, C6400, and C6700 files. No options are needed to specify either a floating-point or fixed-point .out file. The stand-alone simulator determines the target version by reading COFF flags in the .out file.

The banner generated upon invoking the stand-alone simulator defines the values (memory map, silicon revision, fast or slow version of load6x, etc.) used to load and run the .out file. Example 6−1 provides two instances of the banner.

Example 6−1. Sample Stand-Alone Simulator Banners

(a) The file clock.out invoked with no options

```
load6x clock.out
TMS320C6x Standalone Simulator Version X.X
Copyright (c) 1989-2000 by Texas Instruments Incorporated
OPTIONS -- C6xxx Simulator
OPTIONS -- REVISION 2
OPTIONS -- MAP 1 *** DEFAULT MEMORY MAPPING ***
NOTE    : For details on above options please refer to the readme.1st
Loading t.out
 174 Symbols loaded
Done
Interrupt to abort . . .
Hello, world
Time = 133 cycles
NORMAL COMPLETION: 9873 cycles
```

(b) The file closeout invoked with the −a option

```
load6x -a clock.out
TMS320C6x Standalone Simulator Version X.X
Copyright (c) 1989-2000 by Texas Instruments Incorporated
OPTIONS -- C6xxx Memory Hierarchy Modeling Simulator
OPTIONS -- REVISION 2
OPTIONS -- MAP 1 *** DEFAULT MEMORY MAPPING ***
WARNING : Ensure that map modes for linker.cmd file and load6x are same!!
NOTE    : For details on above options please refer to the readme.1st
Loading t.out
 174 Symbols loaded
Done
Interrupt to abort . . .
Hello, world
Time = 7593 cycles
NORMAL COMPLETION: 98705 cycles
```

6.2 Stand-Alone Simulator Options

Following are the options that control the stand-alone simulator, along with descriptions of their effects.

−a Enables data memory bank conflict checking

−b Initializes all memory in the .bss section (data) with 0s. The C language ensures that all uninitialized static storage class variables are initialized to 0 at the beginning of the program. Because the compiler does not set uninitialized variables, the −b option enables you to initialize these variables.

−d[**d**] Enables verbose mode. Prints internal status messages describing I/O at a low level. Use −dd for more verbose information.

−f *value* Initializes all memory in the .bss section (data) with a specified *value*. The *value* is a 32-bit constant (up to 8 hexadecimal digits). For example, `load6x –f 0xabcdabcd` will fill the .bss section with the hexadecimal value abcdabcd.

−g Enables profiling mode. Source files must be compiled with at least the default level of symbolic debug information for profiling to work on the stand-alone simulator. See section 6.4, *Using the Profiling Capability of the Stand-Alone Simulator (−g Option)*, on page 6-8 for more information.

−h Prints the list of available options for the stand-alone simulator.

−i Loads the outfile, even if it is unexecutable

−map *value* Selects the memory map. The `value` can be 0 for memory map 0 (internal program memory begins at 0x1400000) or 1 for memory map 1. Memory map 1 is used by default. If the -q option is not used, the load6x banner lists the selected memory map.

−o *xxx* Sets overall timeout to *xxx* minutes. The stand-alone simulator aborts if the loaded program is not finished after *xxx* minutes.

−q Requests a quiet run (suppresses the banner)

–r *xxx* Relocates all sections by *xxx* bytes during the load. For more information on relocation, see the linker chapter of the *TMS320C6000 Assembly Language Tools User's Guide.*

–rev *value* Selects the silicon revision to simulate. The *value* can be 2 for revision 2 or 3 for revision 3. Revision 2 silicon is simulated by default. See section 6.5, *Selecting Silicon Revision to Simulate (–rev Option)*, on page 6-9 for more information.

–s Time stamp output

–t *xxx* Sets timeout to *xxx* seconds. The stand-alone simulator aborts if no I/O event occurs for *xxx* seconds. I/O events include system calls.

–z Pauses after each internal I/O error. Does not pause for EOF

6.3 Passing Arguments to a Program Through the Loader

In general, for any command-line tool, you can pass arguments on the command line to the program, for example:

Example 6−2. Passing Options on the Command-Line

```
cl6x –a –b –c –d file.c
```

C provides a standard mechanism for communicating arguments to a program, through the argc and argv arguments to main. In C, you can declare the function main as taking two arguments as shown:

```
int main(int argc, char *argv[])
```

The command name is included, so there are six arguments in Example 6−2:

- cl6x
- −a
- −b
- −c
- −d
- file.c

The number of command-line arguments is stored in argc. An array of pointers to strings containing the arguments is stored in argv.

6.3.1 Determining Which Arguments Effect Which Program

The loader and the run-time environment allow you to pass command-line arguments to the program being loaded, just as if the target program was run directly from the command line. The syntax to pass arguments is:

load6x [*options*] *filename.out* [*options*]

Command-line options *before* the object filename are treated as arguments for the loader. The loader treats any command-line options *after* the object filename as command-line arguments to the loaded program. For example:

```
load6x –q –x my_program.out –a –b –c
```

In this example, −q and −x are arguments for load6x; −a, −b, and −c are arguments to be passed to the loaded program (on the target). So, in this case, when *my_program* is loaded, the boot code needs to create an argc and argv value and pass them to the main function. The argc value is 4, and argv is an array of pointers to the four strings: my_program.out, −a, −b, and −c.

6.3.2 Reserving Target Memory to Store the Arguments (−−args Linker Option)

The arguments from the host load6x program must be passed to the target system, so the boot code can pass the information to main. This requires target memory in which to store the strings and the array of pointers to those strings. The linker −−args=*size* option instructs the linker to allocate memory on the target so that the loader can use that memory to store all the contents of the argv array and the argc variable.

It is your responsibility to make the *size* big enough to accommodate all the arguments passed on the loader command line. For Example 6−2, on C6x there are four arguments and four strings. The C standard states that argv must always have an extra NULL pointer after all the legitimate arguments. For Example 6−2 this requires a total of 48 bytes:

Array of five pointers	20 bytes
Space for the strings my_program.out, −a, −b, and −c	24 bytes
Space for argc	4 bytes

If −−args is not specified, or if not enough space is allocated through −−args, the loader will print a warning message which includes the suggested size needed.

6.4 Using the Profiling Capability of the Stand-Alone Simulator

Invoking load6x with the -g option runs the standalone simulator in profiling mode. Source files must be compiled with the at least the default level of symbolic debug information. The profile results resemble the results given by the profiler in the Code Composer Studio debugger. The profile results are stored in a text file called by the same name as the .out file with the .vaa extension.

For example, to create a profile information file called file.vaa, enter the following:

```
load6x –g file.out
```

Example 6−3 runs three different versions of the dot product routines and prints out the result of each routine.

Example 6−3. Profiling Dot Product Routines

```
load6x –q –g t.out
val1 = 11480
val2 = 11480
val3 = 11480

<t.vaa>
Program Name:   /c6xcode/t.out
Start Address:  01409680 main, in line 46, "/c6xcode/t.c"
Stop Address:   014001c0   exit
Run Cycles:     17988
Profile Cycles: 17988
BP Hits:        61


Name       Count  Inclusive  Incl-Max  Exclusive  Excl-Max    Address  Size  Full Name
dot_prod1      1       1024      1024       1024      1024   01409c20   168  _dot_prod1
dot_prod2      1        842       842        842       842   01409b20   232  _dot_prod2
main           1      17980     17980        125       125   01409680   576  _main
dot_prod3      1         89        89         89        89   01498ce0   124  _dot_prod3
```

6.5 Selecting Silicon Revision to Simulate (−rev Option)

A new silicon revision option allows the standalone simulator to support both revisions 2 and 3 of C6000 silicon. By default, the standalone simulator simulates revision 2 silicon.

load6x -rev *value* `file`.out

The valid values are 2 to select revision 2 silicon and 3 to select revision 3 silicon. In revision 3 silicon, the internal data memory has been divided into two memory spaces (0x8000000−0x80007fff and 0x800800−0x800ffff) allowing accesses to the same bank of memory if you are accessing different halves. For example:

```
      MVK    .S2    0x80000000, B5
      MVKH   .S2    0x80000000, B5
      MVK    .S1    0x80008000, A5
      MVKH   .S1    0x80008000, A5
      LDW    .D2    *B5, B6
||    LDW    .D1    *A5, A6
```

In this example, the LDW instructions in parallel do not cause a memory bank conflict in revision 3 silicon, while it will in revision 2 silicon.

For an illustration of an interleaved memory with two memory spaces as for revision 3 silicon, see Figure 4−2 on page 4-34.

If the -q option is not used, the load6x banner lists the selected silicon revision.

6.6 Stand-Alone Simulator Example

A typical use of the stand-alone simulator is running code that includes the clock function to find the number of cycles required to run the code. Use printf statements to display your data to the screen. Example 6−4 shows an example of the C code for accomplishing this.

Example 6−4. C Code With Clock Function

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

main()
{
    clock_t start;
    clock_t overhead;
    clock_t elapsed;

    /* Calculate the overhead from calling clock() */

    start    = clock();
    overhead = clock() – start;

    /* Calculate the elapsed time */

    start   = clock();
    puts(”Hello, world”);
    elapsed = clock() – start – overhead;

    printf(”Time = %ld cycles\n”, (long)elapsed);
}
```

To compile and link the code in Example 6−4, enter the following text on the command line. The −z option invokes the linker, −l linker option names a linker command file, and the −o linker option names the output file.

```
cl6x clock.c -z -l lnk60.cmd -o clock.out
```

To run the stand-alone simulator on the resulting executable COFF file, enter:

```
load6x clock.out
```

Example 6−5. Stand-Alone Simulator Results After Compiling and Linking Example 6−4

```
TMS320C6x Standalone Simulator   Version x.xx
Copyright (c) 1989-2000 Texas Instruments Incorporated
Interrupt to abort . . .
Hello, world
Time = 3338 cycles
NORMAL COMPLETION: 11692 cycles
```

Chapter 7

TMS320C6000 C/C++ Language Implementation

The TMS320C6000 C/C++ compiler supports the C/C++ language standard that was developed by a committee of the American National Standards Institute (ISO) to standardize the C programming language.

The C++ language supported by the C6000 is defined by the ISO/IEC 14882−1998 standard with certain exceptions.

Topic **Page**

7.1 Characteristics of TMS320C6000 C

ISO C supersedes the de facto C standard that is described in the first edition of *The C Programming Language*, by Kernighan and Ritchie. The ISO standard is described in the American National Standard for Information Systems−Programming Language C X3.159−1989. The second edition of *The C Programming Language* is based on the ISO standard and is a reference. ISO C encompasses many of the language extensions provided by recent C compilers and formalizes many previously unspecified characteristics of the language.

The ISO standard identifies some features of the C language that are affected by characteristics of the target processor, run-time environment , or host environment. For reasons of efficiency or practicality, this set of features can differ among standard compilers. This section describes how these features are implemented for the C6000 C/C++ compiler.

The following list identifies all such cases and describes the behavior of the C6000 C/C++ compiler in each case. Each description also includes a reference to more information. Many of the references are to the formal ISO standard or to the second edition of *The C Programming Language* by Kernighan and Ritchie (K&R).

7.1.1 Identifiers and Constants

- All characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external.
  (ISO 3.1.2, K&R A2.3)

- The source (host) and execution (target) character sets are assumed to be ASCII. There are no multibyte characters.
  (ISO 2.2.1, K&R A12.1)

- Hex or octal escape sequences in character or string constants may have values up to 32 bits. (ISO 3.1.3.4, K&R A2.5.2)

- Character constants with multiple characters are encoded as the last character in the sequence. For example,

  ```
  'abc' == 'c'
  ```
  (ISO 3.1.3.4, K&R A2.5.2)

7.1.2 Data Types

- For information about the representation of data types, see section 7.3 on page 7-6. (ISO 3.1.2.5, K&R A4.2)
- The type size_t, which is the result of the *sizeof* operator, is unsigned int. (ISO 3.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is the result of pointer subtraction, is int. (ISO 3.3.6, K&R A7.7)

7.1.3 Conversions

- Float-to-integer conversions truncate toward 0. (ISO 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted. (ISO 3.3.4, K&R A6.6)

7.1.4 Expressions

- When two signed integers are divided and either is negative, the quotient is negative, and the sign of the remainder is the same as the sign of the numerator. The slash mark (/) is used to find the quotient and the percent symbol (%) is used to find the remainder. For example,

  ```
  10 / -3 == -3,    -10 / 3 == -3
  10 % -3 == 1,     -10 % 3 == -1
  ```
  (ISO 3.3.5, K&R A7.6)

  A signed modulus operation takes the sign of the dividend (the first operand).

- A right shift of a signed value is an arithmetic shift; that is, the sign is preserved. (ISO 3.3.7, K&R A7.8)

7.1.5 Declarations

- ❑ The *register* storage class is effective for all chars, shorts, ints, and pointer types. For more information, see section 7.5, *Register Variables*, on page 7-16. (ISO 3.5.1, K&R A2.1)

- ❑ Structure members are packed into words. (ISO 3.5.2.1, K&R A8.3)

- ❑ A bit field defined as an integer is signed. Bit fields are packed into words and do not cross word boundaries. For more information about bit-field packing, see section 8.2.2, *Bit Fields,* page 8-15. (ISO 3.5.2.1, K&R A8.3)

- ❑ The interrupt keyword can be applied only to void functions that have no arguments. For more information about the interrupt keyword, see section 7.4.3 on page 7-10.

7.1.6 Preprocessor

- ❑ The preprocessor ignores any unsupported #pragma directive. (ISO 3.8.6, K&R A12.8)

  The following pragmas *are* supported:

  - CODE_SECTION
  - DATA_ALIGN
  - DATA_MEM_BANK
  - DATA_SECTION
  - FUNC_CANNOT_INLINE
  - FUNC_EXT_CALLED
  - FUNC_INTERRUPT_THRESHOLD
  - FUNC_IS_PURE
  - FUNC_IS_SYSTEM
  - FUNC_NEVER_RETURNS
  - FUNC_NO_GLOBAL_ASG
  - FUNC_NO_IND_ASG
  - INTERRUPT
  - MUST_ITERATE
  - NMI_INTERRUPT
  - PROB_ITERATE
  - STRUCT_ALIGN
  - UNROLL

  For more information on pragmas, see section 7.7 on page 7-18.

7.2 Characteristics of TMS320C6000 C++

The TMS320C6000 compiler supports C++ as defined in the ISO/IEC 14882:1998 standard. The *exceptions* to the standard are as follows:

- Complete C++ standard library support is not included. C subset and basic language support is included.
- These C++ headers for C library facilities are not included:
  - <clocale>
  - <csignal>
  - <cwchar>
  - <cwctype>
- These C++ headers are the only C++ standard library header files included:
  - <new>
  - <typeinfo>
  - <ciso646>
- No support for bad_cast or bad_type_id is included in the typeinfo header.
- Exception handling is not supported.
- Run time type information (RTTI) is disabled by default. RTTI can be enabled with the −rtti compiler option.
- The reinterpret_cast type does not allow casting a pointer to member of one class to a pointer to member of a another class if the classes are unrelated.
- Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- Template parameters are not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.

7.3 Data Types

Table 7−1 lists the size, representation, and range of each scalar data type for the C6000 compiler. Many of the range values are available as standard macros in the header file limits.h. For more information, see section 9.3.6, *Limits (float.h and limits.h)*, on page 9-19.

Table 7−1. TMS320C6000 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	8 bits	ASCII	−128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	−32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	32 bits	2s complement	−2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	40 bits	2s complement	−549 755 813 888	549 755 813 887
unsigned long	40 bits	Binary	0	1 099 511 627 775
long long, signed long long	64 bits	2s complement	−9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	32 bits	2s complement	−2 147 483 648	2 147 483 647
float	32 bits	IEEE 32-bit	1.175 494e−38†	3.40 282 346e+38
double	64 bits	IEEE 64-bit	2.22 507 385e−308†	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e−308†	1.79 769 313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

† Figures are minimum precision.

7.4 Keywords

The C6000 C/C++ compiler supports the standard const, register, restrict, and volatile keywords. In addition, the C6000 C/C++ compiler extends the C/C++ language through the support of the cregister, interrupt, near, and far keywords.

7.4.1 The const Keyword

The TMS320C6000 C/C++ compiler supports the ISO standard keyword *const*. This keyword gives you greater optimization and control over allocation of storage for certain data objects. You can apply the const qualifier to the definition of any variable or array to ensure that its value is not altered.

If you define an object as far const, the .const section allocates storage for the object. The const data storage allocation rule has two exceptions:

- If the keyword volatile is also specified in the definition of an object (for example, volatile const int x). Volatile keywords are assumed to be allocated to RAM. (The program does not modify a const volatile object, but something external to the program might.)
- If the object has automatic storage (allocated on the stack).

In both cases, the storage for the object is the same as if the const keyword were not used.

The placement of the const keyword within a definition is important. For example, the first statement below defines a constant pointer p to a variable int. The second statement defines a variable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
far const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

7.4.2 The cregister Keyword

The C6000 compiler extends the C/C++ language by adding the cregister keyword to allow high level language access to control registers.

When you use the cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers for the C6000 (see Table 7−2). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

Table 7−2. Valid Control Registers

Register	Description
AMR	Addressing mode register
CSR	Control status register
FADCR	(C6700 only) Floating-point adder configuration register
FAUCR	(C6700 only) Floating-point auxiliary configuration register
FMCR	(C6700 only) Floating-point multiplier configuration register
GFPGFR	(C6400 only) Galois field polynomial generator function register
ICR	Interrupt clear register
IER	Interrupt enable register
IFR	Interrupt flag register
IRP	Interrupt return pointer
ISR	Interrupt set register
ISTP	Interrupt service table pointer
NRP	Nonmaskable interrupt return pointer

The cregister keyword can only be used in file scope. The cregister keyword is not allowed on any declaration within the boundaries of a function. It can only be used on objects of type integer or pointer. The cregister keyword is not allowed on objects of any floating-point type or on any structure or union objects.

The cregister keyword does not imply that the object is volatile. If the control register being referenced is volatile (that is, can be modified by some external control), then the object must be declared with the volatile keyword also.

To use the control registers in Table 7−2, you must declare each register as follows. The c6x.h include file defines all the control registers in this manner:

extern cregister volatile unsigned int *register***;**

Once you have declared the register, you can use the register name directly. Note that IFR is read only. See the *TMS320C6000 CPU and Instruction Set Reference Guide* for detailed information on the control registers.

See Example 7−1 for an example that declares and uses control registers.

Example 7−1. Define and Use Control Registers

```
extern cregister volatile unsigned int AMR;
extern cregister volatile unsigned int CSR;
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int ISR;
extern cregister volatile unsigned int ICR;
extern cregister volatile unsigned int IER;

extern cregister volatile unsigned int FADCR;
extern cregister volatile unsigned int FAUCR;
extern cregister volatile unsigned int FMCR;

main()
{
   printf("AMR = %x\n", AMR);
}
```

7.4.3 The interrupt Keyword

The C6000 compiler extends the C/C++ language by adding the interrupt keyword, which specifies that a function is treated as an interrupt function.

Functions that handle interrupts follow special register-saving rules and a special return sequence. When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

You can only use the interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables. For example:

```
interrupt void int_handler()
{
    unsigned int flags;

    ...
}
```

The name c_int00 is the C/C++ entry point. This name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main. Because it has no caller, c_int00 does not save any registers.

Use the alternate keyword, __interrupt, if you are writing code for strict ISO mode (using the −ps compiler option).

7.4.4 The near and far Keywords

The C6000 C/C++ compiler extends the C/C++ language with the near and far keywords to specify how global and static variables are accessed and how functions are called.

Syntactically, the near and far keywords are treated as storage class modifiers. They can appear before, after, or in between the storage class specifiers and types. With the exception of near and far, two storage class modifiers cannot be used together in a single declaration. The following examples are legal combinations of near and far with other storage class modifiers:

```
far static int x;
static near int x;
static int far x;
far int foo();
static far int foo();
```

7.4.4.1 Near and far data objects

Global and static data objects can be accessed in the following two ways:

near keyword The compiler assumes that the data item can be accessed relative to the data page pointer. For example:

```
LDW   *+dp(_address),a0
```

far keyword The compiler cannot access the data item via the dp. This can be required if the total amount of program data is larger than the offset allowed (32K) from the DP. For example:

```
MVKL  _address,a1
MVKH  _address,a1
LDW   *a1,a0
```

Once a variable has been defined to be far, all external references to this variable in other C files or headers must also contain the far keyword. This is also true of the near keyword. However, you will get compiler or linker errors when the far keyword is not used everywhere. Not using the near keyword everywhere only leads to slower data access times.

By default, the compiler generates small-memory model code, which means that every data object is handled as if it were declared near, unless it is actually declared far. If an object is declared near, it is loaded using relative offset addressing from the data page pointer (DP, which is B14). DP points to the beginning of the .bss section.

If you use the DATA_SECTION pragma, the object is indicated as a far variable, and this cannot be overridden. If you reference this object in another file, then you need to use *extern far* when declaring this object in the other source file. This ensures access to the variable, since the variable might not be in the .bss section. For details, see section 7.7.4, DATA_SECTION pragma, on page 7-22.

Note: Defining Global Variables in Assembly Code

If you also define a global variable in assembly code with the .usect directive (where the variable is not assigned in the .bss section) or you allocate a variable into separate section using a #pragma DATA_SECTION directive; and you want to reference that variable in C code, you must declare the variable as extern far. This ensures the compiler does not try to generate an illegal access of the variable by way of the data page pointer.

7.4.4.2 Near and far function calls

Function calls can be invoked in one of two ways:

near keyword The compiler assumes that destination of the call is within ± 1 M word of the caller. Here the compiler uses the PC-relative branch instruction.

```
B     _func
```

far keyword The compiler is told by the user that the call is not within ± 1 M word.

```
MVKL  _func,a1
MVKH  _func,a1
B     a1
```

By default, the compiler generates small-memory model code, which means that every function call is handled as if it were declared near, unless it is actually declared far.

7.4.4.3 Controlling How Run-Time-Support Functions Are Called (−mr Option)

The −mr*n* option controls how run-time-support functions are called:

−mr0 Run-time-support data and calls are near

−mr1 Run-time-support data and calls are far

By default, run-time-support functions are called with the same convention as ordinary functions you code yourself. If you do not use a −ml option to enable one of large-memory models, then these calls will be near. The −mr0 option causes calls to run-time-support functions to be near, regardless of the setting of the −ml option. The −mr0 option is for special situations, and typically is not needed. The −mr1 option causes calls to run-time-support functions to be far, regardless of the setting of the −ml option.

The −mr option only addresses how run-time-support functions are called. Calling functions with the far method does not mean those functions must be in off-chip memory. It simply means those functions can be placed at any distance from where they are called.

Be default, all run-time support data is defined as far.

7.4.4.4 Large model option (−ml)

The large model command line option changes the default near and far assumptions. The near and far modifiers always override the default.

The −ml*n* option generates large-memory model code on four levels (−ml0, −ml1, −ml2, and −ml3):

−ml/−ml0	Aggregate data (structs/arrays) default to far
−ml1	All calls default to far
−ml2	All aggregate data and calls default to far
−ml3	All calls and all data default to far

If no level is specified, all data and functions default to near. Near *data* is accessed via the data page pointer more efficiently while near *calls* are executed more efficiently using a PC relative branch.

Use these options if you have too much static and extern data to fit within a 15-bit scaled offset from the beginning of the .bss section, or if you have calls in which the called function is more than ± 1 M word away from the call site. The linker issues an error message when these situations occur.

If an object is declared far, its address is loaded into a register and the compiler does an indirect load of that register. For more information on the −ml*n* option, see page 2-16.

For more information on the differences in the large and small memory models, see section 8.1.5 on page 8-6.

7.4.5 The restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that may be applied to pointers, references, and arrays. Its use represents a guarantee by the programmer that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In Example 7−2, the restrict Keyword is used to tell the compiler that the function func1 is never called with the pointers a and b pointing to objects that overlap in memory. You are promising that accesses through a and b will never conflict; this means that a write through one pointer cannot affect a read from any other pointers. The precise semantics of the restrict Keyword are described in the 1999 version of the ISO C Standard.

Example 7−2. Use of the restrict type qualifier with pointers

```
void func1(int * restrict a, int * restrict b)
{
  /* func1's code here */
}
```

Example 7−3 illustrates using the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

Example 7−3. Use of the restrict type qualifier with arrays

```
void func2(int c[restrict], int d[restrict])
{
  int i;

  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

7.4.6 The volatile Keyword

The compiler analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C/C++ code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section (as opposed to a register). The compiler does not optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

In this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single-memory read. To correct this, define *ctrl as:

```
volatile unsigned int *ctrl;
```

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.

7.5 Register Variables and Parameters

The TMS320C6000 C/C++ compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the −o option.

- **Compiling with optimization**

  The compiler ignores any register definitions and allocates registers to variables and temporary values by using an algorithm that makes the most efficient use of registers.

- **Compiling without optimization**

  If you use the register keyword, you can suggest variables as candidates for allocation into registers. The compiler uses the same set of registers for allocating temporary expression results as it uses for allocating register variables.

  The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. This limit causes excessive movement of register contents to memory.

Any object with a scalar type (integral, floating point, or pointer) can be defined as a register variable. The register designator is ignored for objects of other types, such as arrays.

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. The compiler copies a register parameter to a register instead of the stack, which speeds access to the parameter within the function.

For more information about registers, see section 8.3, *Register Conventions*, on page 8-17.

7.6 The asm Statement

The TMS320C6000 C/C++ compiler can embed C6000 assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language—the *asm* statement. The asm statement provides access to hardware features that C/C++ cannot provide. The asm statement is syntactically like a call to a function named asm, with one string constant argument:

```
asm("assembler text");
```

The compiler copies the argument string directly into your output file. The assembler text must be enclosed in double quotes. All the usual character string escape codes retain their definitions. For example, you can insert a .byte directive that contains quotes as follows:

```
asm("STR: .byte \"abc\"");
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about the assembly language statements, see the *TMS320C6000 Assembly Language Tools User's Guide*.

The asm statements do not follow the syntactic restrictions of normal C/C++ statements. Each can appear as a statement or a declaration, even outside of blocks. This is useful for inserting directives at the very beginning of a compiled module.

Note: Avoid Disrupting the C/C++ Environment With asm Statements

Be careful not to disrupt the C/C++ environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C/C++ code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use optimization with asm statements. Although the compiler cannot remove asm statements, it can significantly rearrange the code order near them and cause undesired results.

7.7 Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code . The C6000 C/C++ compiler supports the following pragmas:

- CODE_SECTION
- DATA_ALIGN
- DATA_MEM_BANK
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_INTERRUPT_THRESHOLD
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- INTERRUPT
- MUST_ITERATE
- NMI_INTERRUPT
- PROB_ITERATE
- STRUCT_ALIGN
- UNROLL

Most of these pragmas apply to functions. Except for the DATA_MEM_BANK pragma, the arguments *func* and *symbol* cannot be defined or declared inside the body of a function. Pragmas that apply to functions must be specified outside the body of a function; and it must occur before any declaration, definition, or reference to the func or symbol argument. If you do not do this, the compiler issues a warning.

For the pragmas that apply to functions or symbols, the syntax for the pragmas differs between C and C++. In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. In C++, the name is omitted; the pragma applies to the declaration of the object or function that follows it.

7.7.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in a section named *section name*.

The syntax of the pragma in C is:

#pragma CODE_SECTION (*symbol*, **"***section name***");**

The syntax of the pragma in C++ is:

#pragma CODE_SECTION ("*section name***");**

The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

Example 7−4 demonstrates the use of the CODE_SECTION pragma.

Example 7−4. Using the CODE_SECTION Pragma

(a) C source file

```
#pragma CODE_SECTION(fn, "my_sect")

int fn(int x)
{
    return x;
}
```

(b) Generated assembly code

```
   .sect  "my_sect"
   .global   _fn

;******************************************************************************
;* FUNCTION NAME: _fn                                                         *
;*                                                                            *
;*   Regs Modified     : SP                                                   *
;*   Regs Used         : A4,B3,SP                                             *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
_fn:
;** --------------------------------------------------------------------------*
           RET     .S2     B3                ; |6|
           SUB     .D2     SP,8,SP           ; |4|
           STW     .D2T1   A4,*+SP(4)        ; |4|
           ADD     .S2     8,SP,SP           ; |6|
           NOP             2
           ; BRANCH OCCURS                   ; |6|
```

7.7.2 The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2.

The syntax of the pragma in C is:

#pragma DATA_ALIGN (*symbol***,** *constant***);**

The syntax of the pragma in C++ is:

#pragma DATA_ALIGN (*constant***);**

7.7.3 The DATA_MEM_BANK Pragma

The DATA_MEM_BANK pragma aligns a symbol or variable to a specified C6000 internal data memory bank boundary. The *constant* specifies a specific memory bank to start your variables on. (See Figure 4−1 on page 4-33 for a graphic representation of memory banks.) The value of *constant* depends on the C6000 device:

C6200	The C6200 devices contain four memory banks (0, 1, 2, and 3); *constant* can be 0 or 2.
C6400	The C6400 devices contain 8 memory banks; *constant* can be 0, 2, 4, or 6.
C6700	The C6700 devices contain 8 memory banks; *constant* can be 0, 2, 4, or 6.

The syntax of the pragma in C is:

#pragma DATA_MEM_BANK (*symbol***,** *constant***);**

The syntax of the pragma in C++ is:

#pragma DATA_MEM_BANK (*constant***);**

Both global and local variables can be aligned with the DATA_MEM_BANK pragma. The DATA_MEM_BANK pragma must reside inside the function that contains the local variable being aligned. The *symbol* can also be used as a parameter in the DATA_SECTION pragma.

When optimization is enabled, the tools may or may not use the stack to store the values of local variables.

The DATA_MEM_BANK pragma allows you to align data on any data memory bank that can hold data of the *symbol*'s type size. This is useful if you need to align data in a particular way to avoid memory bank conflicts in your hand-coded assembly code versus padding with zeros and having to account for the padding in your code.

This pragma increases the amount of space used in data memory by a small amount as padding is used to align data onto the correct bank.

For C6200, the code in Example 7−5 guarantees that array x begins at an address ending in 4 or c (in hexadecimal), and that array y begins at an address ending in 4 or c. The alignment for array y affects its stack placement. Array z is placed in the .z_sect section, and begins at an address ending in 0 or 8.

Example 7−5. Using the DATA_MEM_BANK Pragma

```
#pragma DATA_MEM_BANK (x, 2);
short x[100];

#pragma DATA_MEM_BANK (z, 0);
#pragma DATA_SECTION (z, ".z_sect");
short z[100];

void main()
{
    #pragma DATA_MEM_BANK (y, 2);
    short y[100];
    ...
}
```

7.7.4 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section named *section name*.

The syntax of the pragma in C is:

#pragma DATA_SECTION (*symbol*, **"***section name***");**

The syntax of the pragma in C++ is:

#pragma DATA_SECTION ("*section name***");**

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section. If you allocate a global variable using a DATA_SECTION pragma and you want to reference the variable in C code, you must declare the variable as extern far.

Example 7−6 demonstrates the use of the DATA_SECTION pragma.

Example 7−6. Using the DATA_SECTION Pragma

(a) C source file

```
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferA[512];
char bufferB[512];
```

(b) C++ source file

```
char bufferA[512];
#pragma DATA_SECTION("my_sect")
char bufferB[512];
```

(c) Assembly source file

```
        .global _bufferA
        .bss    _bufferA,512,4
        .global _bufferB
_bufferB:       .usect  "my_sect",512,4
```

7.7.5 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see section 2.10, *Using Inline Function Expansion*, on page 2-38.

The pragma must appear before any declaration or reference to the function that you want to keep.

The syntax of the pragma in C is:

```
#pragma FUNC_CANNOT_INLINE (func);
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_CANNOT_INLINE;
```

In C, the argument *func* is the name of the function that cannot be inlined. In C++, the pragma applies to the next function declared.

7.7.6 The FUNC_EXT_CALLED Pragma

When you use the −pm option, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C/C++ functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep these C functions or any other functions that these C/C++ functions call. These functions act as entry points into C/C++.

The pragma must appear before any declaration or reference to the function that you want to keep.

The syntax of the pragma in C is:

```
#pragma FUNC_EXT_CALLED (func);
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_EXT_CALLED;
```

In C, the argument *func* is the name of the function that you do not want removed. In C++, the pragma applies to the next function declared.

Except for _c_int00, which is the name reserved for the system reset interrupt for C/C++programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See section 3.6.2, *Optimization Considerations When Mixing C and Assembly*, on page 3-22.

7.7.7 The FUNC_INTERRUPT_THRESHOLD Pragma

The compiler allows interrupts to be disabled around software pipelined loops for threshold cycles within the function. This implements the −mi option for a single function (see section 2.11, *Interrupt Flexibility Options*, on page 2-43). The FUNC_INTERRUPT_THRESHOLD pragma always overrides the −mi*n* command line option. A threshold value less than 0 assumes that the function is never interrupted, which is equivalent to an interrupt threshold of infinity.

The syntax of the pragma in C is:

#pragma FUNC_INTERRUPT_THRESHOLD (*func***,** *threshold***);**

The syntax of the pragma in C++ is:

#pragma FUNC_INTERRUPT_THRESHOLD (*threshold***);**

The following examples demonstrate the use of different thresholds:

- `#pragma FUNC_INTERRUPT_THRESHOLD (foo, 2000)`

  The function foo( ) must be interruptible at least every 2,000 cycles.

- `#pragma FUNC_INTERRUPT_THRESHOLD (foo, 1)`

  The function foo( ) must always be interruptible.

- `#pragma FUNC_INTERRUPT_THRESHOLD (foo, –1)`

  The function foo( ) is never interrupted.

7.7.8 The FUNC_IS_PURE Pragma

The FUNC_IS_PURE pragma specifies to the compiler that the named function has no side effects. This allows the compiler to do the following:

- ❏ Delete the call to the function if the function's value is not needed
- ❏ Delete duplicate functions

The pragma must appear before any declaration or reference to the function.

The syntax of the pragma in C is:

```
#pragma FUNC_IS_PURE (func);
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_PURE;
```

In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

7.7.9 The FUNC_IS_SYSTEM Pragma

The FUNC_IS_SYSTEM pragma specifies to the compiler that the named function has the behavior defined by the ISO standard for a function with that name.

The pragma must appear before any declaration or reference to the function that you want to keep.

The syntax of the pragma in C is:

```
#pragma FUNC_IS_SYSTEM (func);
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_SYSTEM;
```

In C, the argument *func* is the name of the function to treat as an ISO standard function. In C++, the pragma applies to the next function declared.

7.7.10 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the compiler that the function never returns to its caller.

The pragma must appear before any declaration or reference to the function that you want to keep.

The syntax of the pragma in C is:

```
#pragma FUNC_NEVER_RETURNS (func);
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NEVER_RETURNS;
```

In C, the argument *func* is the name of the function that does not return. In C++, the pragma applies to the next function declared.

7.7.11 The FUNC_NO_GLOBAL_ASG Pragma

The FUNC_NO_GLOBAL_ASG pragma specifies to the compiler that the function makes no assignments to named global variables and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep.

The syntax of the pragma in C is:

```
#pragma FUNC_NO_GLOBAL_ASG (func);
```

The syntax of the pragma in C is:

```
#pragma FUNC_NO_GLOBAL_ASG;
```

In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

7.7.12 The FUNC_NO_IND_ASG Pragma

The FUNC_NO_IND_ASG pragma specifies to the compiler that the function makes no assignments through pointers and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep.

The syntax of the pragma in C is:

```
#pragma FUNC_NO_IND_ASG (func);
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_IND_ASG;
```

In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

7.7.13 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma INTERRUPT (func);
```

The syntax of the pragma in C++ is:

```
#pragma INTERRUPT;
```

The code for the function will return via the IRP (interrupt return pointer).

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

7.7.14 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. You guarantee that these properties are always true. Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a specific number of times. Anytime the UNROLL pragma is applied to a loop, MUST_ITERATE should be applied to the same loop. Here the MUST_ITERATE pragma's third argument, multiple, is the most important and should always be specified.

Furthermore, the MUST_ITERATE pragma should be applied to any other loops as often as possible. This is because the information provided via the pragma (especially the minimum number of iterations) aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). It also helps the compiler reduce code size.

No statements are allowed between the MUST_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and PROB_ITERATE, can appear between the MUST_ITERATE pragma and the loop.

7.7.14.1 The MUST_ITERATE Pragma Syntax

The syntax of the pragma for C and C++ is:

#pragma MUST_ITERATE (*min***,** *max***,** *multiple***);**

The arguments *min* and *max* are programmer-guaranteed minimum and maximum trip counts. The trip count is the number of times a loop iterates. The trip count of the loop must be evenly divisible by *multiple*. All arguments are optional. For example, if the trip count could be 5 or greater, you can specify the argument list as follows:

```
#pragma MUST_ITERATE(5);
```

However, if the trip count could be any nonzero multiple of 5, the pragma would look like this:

```
#pragma MUST_ITERATE(5, , 5); /* Note the blank field for max */
```

It is sometimes necessary for you to provide min and multiple in order for the compiler to perform unrolling. This is especially the case when the compiler cannot easily determine how many iterations the loop will perform (i.e. the loop has a complex exit condition).

When specifying a multiple via the MUST_ITERATE pragma, results of the program are undefined if the trip count is not evenly divisible by multiple. Also, results of the program are undefined if the trip count is less than the minimum or greater than the maximum specified.

If no min is specified, zero is used. If no max is specified, the largest possible number is used. If multiple MUST_ITERATE pragmas are specified for the same loop, the smallest max and largest min are used.

7.7.14.2 Using MUST_ITERATE to Expand Compiler Knowledge of Loops

Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a certain number of times. The example below tells the compiler that the loop is guaranteed to run exactly 10 times:

```
#pragma MUST_ITERATE(10,10);
for(i = 0; i < trip_count; i++)  { ...
```

In this example, the compiler attempts to generate a software pipelined loop even without the pragma. However, if MUST_ITERATE is not specified for a loop such as this, the compiler generates code to bypass the loop, to account for the possibility of 0 iterations. With the pragma specification, the compiler knows that the loop iterates at least once and can eliminate the loop-bypassing code.

MUST_ITERATE can specify a range for the trip count as well as a factor of the trip count. For example:

```
#pragma MUST_ITERATE(8, 48, 8);
for(i = 0; i < trip_count; i++)  { ...
```

This example tells the compiler that the loop executes between 8 and 48 times and that the trip_count variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The multiple argument allows the compiler to unroll the loop.

You should also consider using MUST_ITERATE for loops with complicated bounds. In the following example:

```
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

The compiler would have to generate a divide function call to determine, at run time, the exact number of iterations performed. The compiler will not do this. In this case, using MUST_ITERATE to specify that the loop always executes 8 times allows the compiler to attempt to generate a software pipelined loop:

```
#pragma MUST_ITERATE(8, 8);
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

7.7.15 The NMI_INTERRUPT Pragma

The NMI_INTERRUPT pragma enables you to handle non-maskable interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

#pragma NMI_INTERRUPT (*func***);**

The syntax of the pragma in C++ is:

#pragma NMI_INTERRUPT;

The code generated for the function will return via the NRP versus the IRP as for a function declared with the interrupt keyword or INTERRUPT pragma.

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (function) does not need to conform to a naming convention.

7.7.16 The PROB_ITERATE Pragma

The PROB_ITERATE pragma specifies to the compiler certain properties of a loop. You assert that these properties are true in the common case. The PROB_ITERATE pragma aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). PROB_ITERATE is useful only when the MUST_ITERATE pragma is not used or the PROB_ITERATE parameters are more constraining than the MUST_ITERATE parameters.

No statements are allowed between the PROB_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and MUST_ITERATE, may appear between the PROB_ITERATE pragma and the loop.

The syntax of the pragma for C and C++ is:

#pragma PROB_ITERATE (*min***,** *max***);**

Where min and max are the minimum and maximum trip counts of the loop in the common case. The trip count is the number of times a loop iterates. Both arguments are optional.

For example, PROB_ITERATE could be applied to a loop that executes for eight iterations in the majority of cases (but sometimes may execute more or less than eight iterations):

```
#pragma PROB_ITERATE(8, 8);
```

If only the minimum expected trip count is known (say it is 5), the pragma would look like this:

```
#pragma PROB_ITERATE(5);
```

If only the maximum expected trip count is known (say it is 10), the pragma would look like this:

```
#pragma PROB_ITERATE(, 10);  /* Note the blank field for min */
```

7.7.17 The STRUCT_ALIGN Pragma

The STRUCT_ALIGN pragma is similar to DATA_ALIGN, but it can be applied to a structure, union type, or typedef and is inherited by any symbol created from that type. The STRUCT_ALIGN pragma is supported only in C.

The syntax of the pragma is:

#pragma STRUCT_ALIGN (*type***,** *constant expression***);**

This pragma guarantees that the alignment of the named type or the base type of the named typedef is at least equal to that of the expression. (The alignment may be greater as required by the compiler.) The alignment must be a power of 2. The *type* must be a type or a typedef name. If a type, it must be either a structure tag or a union tag. If a typedef, its base type must be either a structure tag or a union tag.

Since ISO C declares that a typedef is simply an alias for a type (i.e. a struct) this pragma can be applied to the struct, the typedef of the struct, or any typedef derived from them, and affects all aliases of the base type.

This example aligns any st_tag structure variables on a page boundary:

```
typedef struct st_tag
{
   int   a;
   short b;
} st_typedef;

#pragma STRUCT_ALIGN (st_tag, 128);
```

Any use of STRUCT_ALIGN with a basic type (int, short, float) or a variable results in an error.

7.7.18 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. The UNROLL pragma is useful for helping the compiler utilize SIMD instructions on the C6400 family. It is also useful in cases where better utilization of software pipeline resources are needed over a non-unrolled loop.

The optimizer must be invoked (use −o1, −o2, or −o3) in order for pragma-specified loop unrolling to take place. The compiler has the option of ignoring this pragma.

No statements are allowed between the UNROLL pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as MUST_ITERATE and PROB_ITERATE, may appear between the UNROLL pragma and the loop.

The syntax of the pragma is for both C and C++:

#pragma UNROLL (*n*);

If possible, the compiler unrolls the loop so there are *n* copies of the original loop. The compiler only unrolls if it can determine that unrolling by a factor of *n* is safe. In order to increase the chances the loop is unrolled, the compiler needs to know certain properties:

- The loop iterates a multiple of *n* times. This information can be specified to the compiler via the multiple argument in the MUST_ITERATE pragma.
- The smallest possible number of iterations of the loop.
- The largest possible number of iterations of the loop.

The compiler can sometimes obtain this information itself by analyzing the code. However, sometimes the compiler can be overly conservative in its assumptions and therefore generates more code than is necessary when unrolling. This can also lead to not unrolling at all.

Furthermore, if the mechanism that determines when the loop should exit is complex, the compiler may not be able to determine these properties of the loop. In these cases, you must tell the compiler the properties of the loop by using the MUST_ITERATE pragma.

Specifying #pragma UNROLL(1); asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

If multiple UNROLL pragmas are specified for the same loop, it is undefined which unroll pragma is used, if any.

7.8 Generating Linknames

The compiler transforms the names of externally visible identifiers when creating their linknames. The algorithm used depends on the scope within which the identifier is declared. For objects and C functions, an underscore (_) is prefixed to the identifier name. C++ functions are prefixed with an underscore also, but the function name is modified further.

Mangling is the process of embedding a function's signature (the number and types of its parameters) into its name. Mangling occurs only in C++ code. The mangling algorithm used closely follows that described in *The Annotated Reference Manual* (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

For example, the general form of a C++ linkname for a function named func is:

```
_func__Fparmcodes
```

Where parmcodes is a sequence of letters that encodes the parameter types of func.

For this simple C++ source file:

```
int foo(int i){ }   //global C++ function
```

This is the resulting assembly code:

```
_foo__Fi
```

The linkname of foo is _foo__Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided that demangles names into those found in the original C++ source. See Chapter 11, *C++ Name Demangling*, for more information.

7.9 Initializing Static and Global Variables

The ISO C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running. This task is typically done when the program is loaded. Because the loading process is heavily dependent on the specific environment of the target application system, the compiler itself makes no provision for preinitializing variables at run time. It is up to your application to fulfill this requirement.

7.9.1 Initializing Static and Global Variables With the Linker

If your loader does not preinitialize variables, you can use the linker to preinitialize the variables to 0 in the object file. For example, in the linker command file, use a fill value of 0 in the .bss section:

```
SECTIONS
   {
      ...
      .bss: fill = 0x00;
      ...
   }
```

Because the linker writes a complete load image of the zeroed .bss section into the output COFF file, this method can have the unwanted effect of significantly increasing the size of the output file (but not the program).

If you burn your application into ROM, you should explicitly initialize variables that require initialization. The preceding method initializes .bss to 0 only at load time, not at system reset or power up. To make these variables 0 at run time, explicitly define them in your code.

For more information about linker command files and the SECTIONS directive, see the linker description information in the *TMS320C6000 Assembly Language Tools User's Guide.*

7.9.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in section 7.9). For example:

```
const int zero;    /*  may not be initialized to 0  */
```

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .const. For example:

```
const int zero = 0    /*  guaranteed to be 0  */
```

This corresponds to an entry in the .const section:

```
. . .sect ...  .const
_zero
.. .word ...  0
```

This feature is particularly useful for declaring a large table of constants, because neither time nor space is wasted at system startup to initialize the table. Additionally, the linker can be used to place the .const section in ROM.

You can use the DATA_SECTION pragma to put the variable in a section other than .const. For example, the following C code:

```
#pragma DATA_SECTION (var, ".mysect");
   const int zero=0;
```

is compiled into this assembly code:

```
   .sect .mysect
_zero
   .word 0
```

7.10 Changing the ISO C Language Mode

The −pk, −pr, and −ps options let you specify how the C/C++ compiler interprets your source code. You can compile your source code in the following modes:

- ❏ Normal ISO mode
- ❏ K&R C mode
- ❏ Relaxed ISO mode
- ❏ Strict ISO mode

The default is normal ISO mode. Under normal ISO mode, most ISO violations are emitted as errors. Strict ISO violations (those idioms and allowances commonly accepted by C/C++ compilers, although violations with a strict interpretation of ISO), however, are emitted as warnings. Language extensions, even those that conflict with ISO C, are enabled.

For C++ code, ISO mode designates the latest supported working paper. K&R C mode does not apply to C++ code.

7.10.1 Compatibility With K&R C (−pk Option)

The ISO C/C++ language is a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ISO compilers correctly compile and run without modification.

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ISO C and the first edition's C standard (the first edition is referred to in this manual as K&R C).

To simplify the process of compiling existing C programs with the C6000 ISO C/C++ compiler, the compiler has a K&R option (−pk) that modifies some semantic rules of the language for compatibility with older code. In general, the −pk option relaxes requirements that are stricter for ISO C than for K&R C. The −pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, −pk simply liberalizes the ISO rules without revoking any of the features.

The specific differences between the ISO version of C and the K&R version of C are as follows:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider signed type. Under K&R C, the result type was an unsigned version of the wider type; under ISO, the result type is a signed version of the wider type. This affects operations that perform differently when applied to signed or unsigned operands; namely, comparisons, division (and mod), and right shift:

```
unsigned short u;
int i;
if (u < i) ...   /* SIGNED comparison, unless -pk used */
```

- ISO prohibits combining two pointers to different types in an operation. In most K&R compilers, this situation produces only a warning. Such cases are still diagnosed when −pk is used, but with less severity:

```
int *p;
char *q = p;     /* error without -pk, warning with -pk */
```

- External declarations with no type or storage class (only an identifier) are illegal in ISO but legal in K&R:

```
a;               /* illegal unless -pk used */
```

- ISO interprets file scope definitions that have no initializers as *tentative definitions.* In a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object and usually an error. For example:

```
int a;
int a;           /* illegal if -pk used, OK if not */
```

  Under ISO, the result of these two definitions is a single definition for the object a. For most K&R compilers, this sequence is illegal, because int a is defined twice.

- ISO prohibits, but K&R allows objects with external linkage to be redeclared as static:

```
extern int a;
static int a;    /* illegal unless -pk used */
```

- Unrecognized escape sequences in string and character constants are explicitly illegal under ISO but ignored under K&R:

```
char c = '\q';   /* same as 'q' if -pk used, error */
                 /* if not */
```

- ISO specifies that bit fields must be of type int or unsigned. With −pk, bit fields can be legally defined with any integral type. For example:

```
struct s
{
 short f : 2;    /* illegal unless -pk used */
};
```

- K&R syntax allows a trailing comma in enumerator lists:

```
enum { a, b, c, };   /* illegal unless -pk used */
```

- K&R syntax allows trailing tokens on preprocessor directives:

```
#endif NAME          /* illegal unless -pk used */
```

7.10.2 Enabling Strict ISO Mode and Relaxed ISO Mode (−ps and −pr Options)

Use the −ps option when you want to compile under strict ISO mode. In this mode, error messages are provided when non-ISO features are used, and language extensions that could invalidate a strictly conforming program are disabled. Examples of such extensions are the inline and asm keywords.

Use the −pr option when you want the compiler to ignore strict ISO violations rather than emit a warning (as occurs in normal ISO mode) or an error message (as occurs in strict ISO mode). In relaxed ISO mode, the compiler accepts extensions to the ISO C standard, even when they conflict with ISO C.

7.10.3 Enabling Embedded C++ Mode (−pe Option)

The compiler supports the compilation of embedded C++. In this mode, some features of C++ are removed that are of less value or too expensive to support in an embedded system. Embedded C++ omits these C++ features:

- Templates
- Exception handling
- Run-time type information
- The new cast syntax
- The keyword mutable
- Multiple inheritance
- Virtual inheritance

Under the standard definition of embedded C++, namespaces and using-declarations are not supported. The C6000 compiler nevertheless allows these features under embedded C++ because the C++ runtime support library makes use of them. Furthermore, these features impose no runtime penalty.

Chapter 8

Run-Time Environment

This chapter describes the TMS320C6000 C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

Topic **Page**

8.1 Memory Model

The C6000 compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 32-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces.

For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

8.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*. The sections are allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about sections and allocating them, see the introductory COFF information in the *TMS320C6000 Assembly Language Tools User's Guide*.

The C6000 compiler creates the following sections:

- **Initialized sections** contain data or executable code. The C/C++ compiler creates the following initialized sections:
  - The **.cinit section** contains tables for initializing variables and constants.
  - The **.const section** contains string literals, floating-point constants, and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - The **.switch section** contains jump tables for large switch statements.
  - The **.text section** contains all the executable code.

- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run-time to create and store variables. The compiler creates the following uninitialized sections:
  - The **.bss section** reserves space for global and static variables. When you specify the −c linker option, at program startup, the C boot routine copies data out of the .cinit section (which can be in ROM) and stores it in the .bss section. The compiler defines the global symbol $bss and assigns $bss the value of the starting address of the .bss section.
  - The **.far section** reserves space for global and static variables that are declared far.
  - The **.stack section** allocates memory for the system stack. This memory passes arguments to functions and allocates local variables.
  - The **.sysmem section** reserves space for dynamic memory allocation. The reserved space is used by the malloc, calloc, and realloc functions. If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.

Note: Use Only Code in Program Memory

With the exception of .text, the initialized and uninitialized sections cannot be allocated into internal program memory.

The assembler creates the default sections .text, .bss, and .data. The C/C++ compiler, however, does not use the .data section. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see sections 7.7.1, *The CODE_SECTION Pragma*, on page 7-19 and 7.7.4, *The DATA_SECTION Pragma*, on page 7-22).

8.1.2 C/C++ System Stack

The C/C++ compiler uses a stack to:

- ❏ Save function return addresses
- ❏ Allocate local variables
- ❏ Pass arguments to functions
- ❏ Save temporary results

The run-time stack grows from the high addresses to the low addresses. The compiler uses the B15 register to manage this stack. B15 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 0x400 (1024) bytes. You can change the stack size at link time by using the −stack option with the linker command. For more information on the −stack option, see section 5.2, *Linker Option*s, on page 5-5.

At system initialization, SP is set to the first 8-byte aligned address before the end (highest numerical address) of the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time.

The C/C++ environment automatically decrements SP (register B15) at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is incremented at the exit of the function to restore the stack to its state before the function was entered. If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the state it had before the function was entered. For more information about stack and stack pointer, see section 8.4, *Function Structure and Calling Conventions*, on page 8-19.

Note: Stack Overflow

The compiler provides no means to check for stack overflow during compilation or at run-time. Place the beginning of the .stack section in the first address after an unmapped memory space so stack overflow will cause a simulator fault. This makes this problem easy to detect. Be sure to allow enough space for the stack to grow.

8.1.3 Dynamic Memory Allocation

Dynamic memory allocation is not a standard part of the C language. The run-time-support library supplied with the C6000 compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run-time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the −heap *size* option with the linker command. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 0x400 bytes. For more information on the −heap option, see section 5.2, *Linker Options,* on page 5-6.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers) and the memory pool is in a separate section (.sysmem); therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of defining them as global or static. For example, instead of a definition such as:

```
struct big table[100];
```

use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

8.1.4 Initialization of Variables

The C/C++ compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .cinit section are stored in ROM. At system initialization time, the C/C++ boot routine copies data from these tables (in ROM) to the initialized variables in .bss (RAM).

In situations where a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at run-time. You can specify this to the linker by using the −cr linker option. For more information, see section 8.8, *System Initialization*, on page 8-51.

8.1.5 Memory Models

The compiler supports two memory models that affect how the .bss section is allocated into memory. Neither model restricts the size of the .text or .cinit sections.

- **The small memory model**, which is the default, requires that the entire .bss section fit within 32K bytes (32 768 bytes) of memory. This means that the total space for all static and global data in the program must be less than 32K bytes. The compiler sets the data-page pointer register (DP, which is B14) during run-time initialization to point to the beginning of the .bss section. Then the compiler can access all objects in .bss (global and static variables and constant tables) with direct addressing without modifying the DP.

- **The large memory model** does not restrict the size of the .bss section; unlimited space is available for static and global data. However, when the compiler accesses any global or static object that is not stored in .bss, it must first load the object's address into a register before a global data item is accessed. This task produces two extra assembly instructions.

  For example, the following compiler-generated assembly language uses the MVKL and MVKH instructions to move the global variable _x into the A0 register, then loads the B0 register using a pointer to A0:

  ```
  MVKL   _x, A0
  MVKH   _x, A0
  LDW    *A0, B0
  ```

  To use the large memory model, invoke the compiler with the −ml*n* option. For more information on the −ml*n* option, see section 7.4.4.4, *Large Model Option (−ml)*, on page 7-13.

For more information on the storage allocation of global and static variables, see section 7.4.4, *The near and far Keywords*, on page 7-11.

8.1.6 Position Independent Data

Near global and static data are stored in the .bss section. All near data for a program must fit within 32K bytes of memory. This limit comes from the addressing mode used to access near data, which is limited to a 15-bit unsigned offset from DP (B14) the data page pointer.

For some applications, it may be desirable to have multiple data pages with separate instances of near data. For example, a multi-channel application may have multiple copies of the same program running with different data pages. The functionality is supported by the C6000 compiler's memory model, and is referred to as position independent data.

Position independent data means that all near data accesses are relative to the data page (DP) pointer, allowing for the DP to be changed at run-time. There are three areas where position independent data is implemented by the compiler:

1) Near direct memory access

```
STW  B4,*DP(_a)

.global _a
.bss    _a,4,4
```

All near direct accesses are relative to the DP.

2) Near indirect memory access

```
MVK (_a – $bss),A0
ADD DP,A0,A0
```

The expression (_a − $bss) calculates the offset of the symbol _a from the start of the .bss section. The compiler defines the global $bss in generated assembly code. The value of $bss is the starting address of the .bss section.

3) Initialized near pointers

The .cinit record for an initialized near pointer value is stored as an offset from the beginning of the .bss section. During the autoinitialization of global variables, the data page pointer is added to these offsets. (See section 8.8.3, *Initialization Tables*, on page 8-53.)

8.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

8.2.1 Data Type Storage

Table 8−1 lists register and memory storage for various data types:

Table 8−1. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	Bits 0−7 of register	8 bits aligned to 8-bit boundary
unsigned char	Bits 0−7 of register	8 bits aligned to 8-bit boundary
short	Bits 0−15 of register	16 bits aligned to 16-bit boundary
unsigned short	Bits 0−15 of register	16 bits aligned to 16-bit boundary
int	Entire register	32 bits aligned to 32-bit boundary
unsigned int	Entire register	32 bits aligned to 32-bit boundary
enum	Entire register	32 bits aligned to 32-bit boundary
float	Entire register	32 bits aligned to 32-bit boundary
long	Bits 0−39 of even/odd register pair	64 bits aligned to 64-bit boundary
unsigned long	Bits 0−39 of even/odd register pair	64 bits aligned to 64-bit boundary
long long	Even/odd register pair	64 bits aligned to 64-bit boundary
unsigned long long	Even/odd register pair	64 bits aligned to 64-bit boundary
double	Even/odd register pair	64 bits aligned to 64-bit boundary
long double	Even/odd register pair	64 bits aligned to 64-bit boundary
struct	Members are stored as their individual types require.	Multiple of 8 bits aligned to boundary of largest member type; members are stored and aligned as their individual types require.
array	Members are stored as their individual types require.	Members are stored as their individual types require; aligned to 64-bit boundary for C64x; aligned to 32-bit boundary for all types 32 bits and smaller, and to 64-bit boundary for all types larger than 32 bits for C62x and C67x. All arrays inside a structure are aligned according to the type of each element in the array.

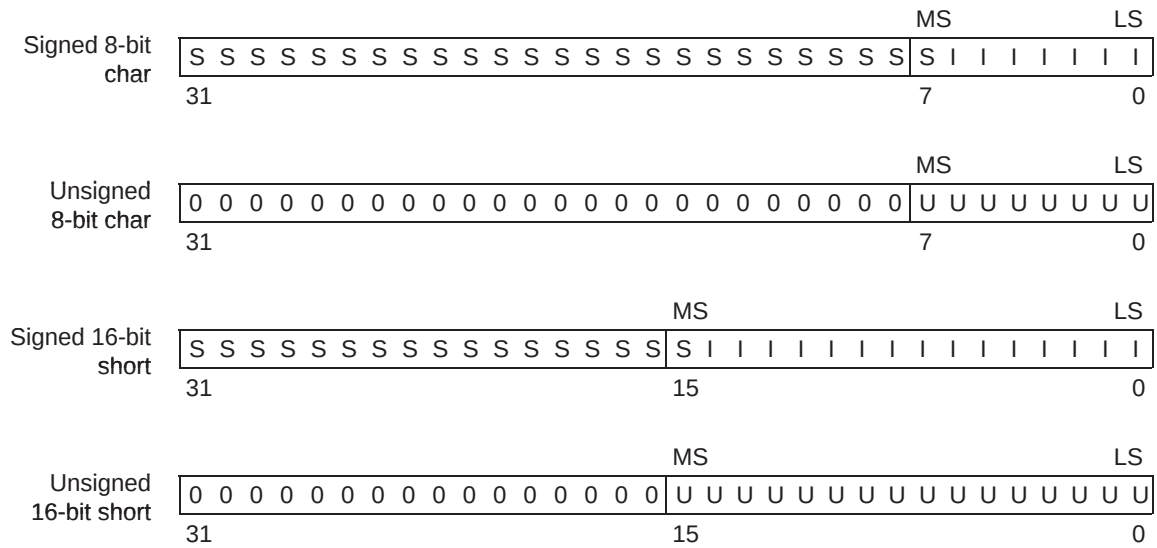
Table 8−1. Data Representation in Registers and Memory (Continued)

Data Type	Register Storage	Memory Storage
pointer to data member	Bits 0−31 of register	32 bits aligned to 32-bit boundary
pointer to member function	Components stored as their individual types require	64 bits aligned to 32-bit boundary

8.2.1.1 char and short Data Types (signed and unsigned)

The char and unsigned char data types are stored in memory as a single byte and are loaded to and stored from bits 0−7 of a register (see Figure 8−1). Objects defined as short or unsigned short are stored in memory as two bytes and are loaded to and stored from bits 0−15 of a register (see Figure 8−1). In big-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 8−15 of the register and moving the second byte of memory to bits 0−7. In little-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0−7 of the register and moving the second byte of memory to bits 8−15.

Figure 8−1. Char and Short Data Storage Format

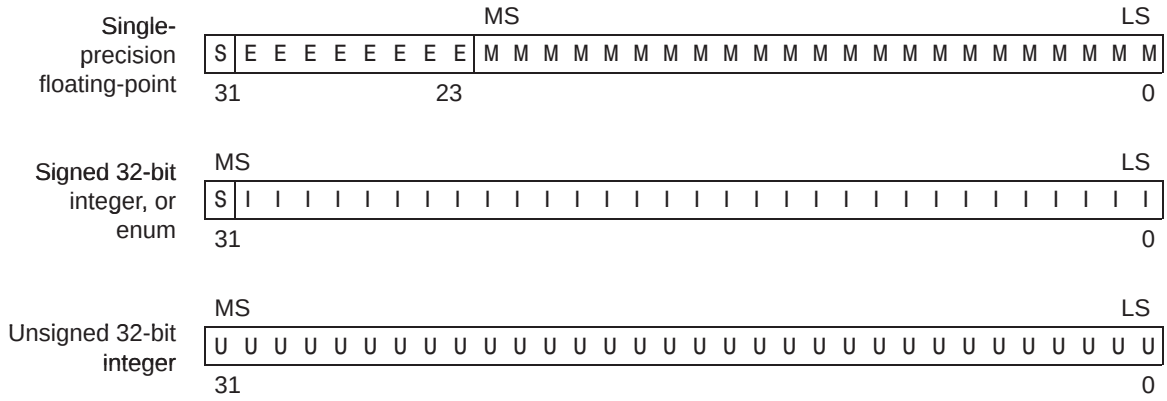


Legend: S = sign
I = signed integer
U = unsigned integer
MS = most significant
LS = least significant

8.2.1.2 enum, float, and int Data Types (signed and unsigned)

The int, unsigned int, enum, and float data types are stored in memory as 32-bit objects (see Figure 8−2). Objects of these types are loaded to and stored from bits 0−31 of a register. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24−31 of the register, moving the second byte of memory to bits 16−23, moving the third byte to bits 8−15, and moving the fourth byte to bits 0−7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0−7 of the register, moving the second byte to bits 8−15, moving the third byte to bits 16−23, and moving the fourth byte to bits 24−31.

Figure 8−2. 32-Bit Data Storage Format



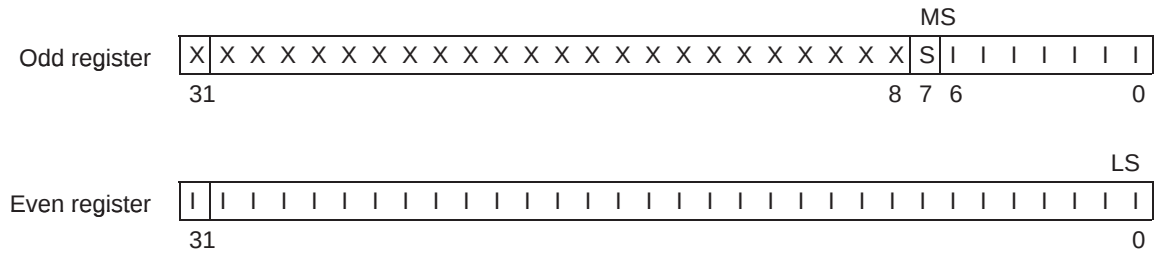
Legend: S = sign; M = mantissa; U = unsigned integer; E = exponent; I = signed integer; MS = most significant; LS = least significant

8.2.1.3 long Data Types (signed and unsigned)

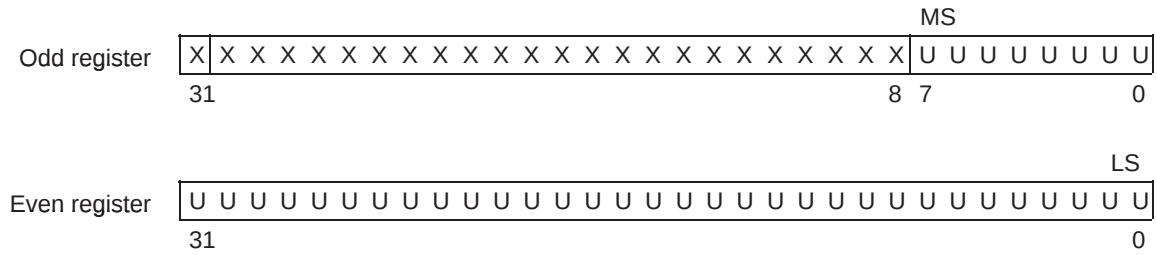
Long and unsigned long data types are stored in an odd/even pair of registers (see Figure 8−3) and are always referenced as a pair in the format of odd register:even register (for example, A1:A0). In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the highest byte of the odd register but is ignored.

Figure 8−3. 40-Bit Data Storage Format

(a) Signed 40-bit long



(b) Unsigned 40-bit long



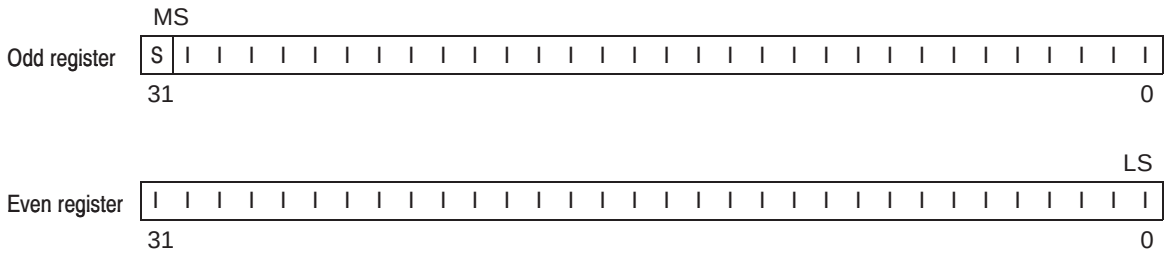
Legend: S = sign I = signed integer MS = most significant
U = unsigned integer X = unused LS = least significant

8.2.1.4 long long Data Types (signed and unsigned)

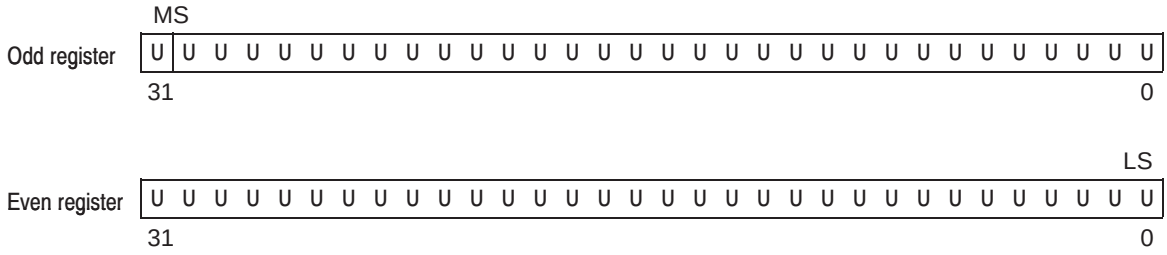
Long long and unsigned long long data types are stored in an odd/even pair of registers (see Figure 8−4) and are always referenced as a pair in the format of odd register:even register (for example, A1:A0). In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the highest byte of the odd register.

Figure 8−4. 64-Bit Data Storage Format

(a) Signed 64-bit long



(b) Unsigned 64-bit long

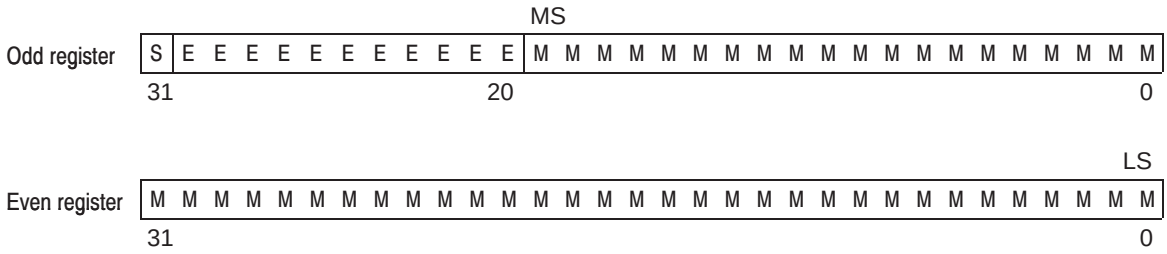


Legend: S = sign I = signed integer MS = most significant
U = unsigned integer X = unused LS = least significant

8.2.1.5 double and long double Data Types

Double and long double data types are stored in an odd/even pair of registers (see Figure 8−5) and can only exist in a register in one format: as a pair in the format of odd register:even register (for example, A1:A0). The odd memory word contains the sign bit, exponent, and the most significant part of the mantissa. The even memory word contains the least significant part of the mantissa. In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register. In little-endian mode, if code is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, if code is loaded from location 0, then the byte at 0 is the highest byte of the odd register.

Figure 8−5. Double-Precision Floating-Point Data Storage Format



Legend: S = sign
M = mantissa
E = exponent
MS = most significant
LS = least significant

8.2.1.6 Structures and Arrays

A nested structure is aligned to a boundary required by the largest type it contains. For example, if the largest type in a nested structure is of type short, then the nested structure is aligned to a 2-byte boundary. If the largest type in a nested structure is of type long, unsigned long, double, or long double, then the nested structure is aligned to an 8-byte boundary.

Structures always reserve memory in multiples of the size of the largest element type. For example, if a structure contains an int, unsigned int, or float, a multiple of 4 bytes of storage is reserved in memory. Members of structures are stored in the same manner as if they were individual objects.

Arrays are aligned on an 8-byte boundary for C64x, and either a 4-byte (for all element types of 32 bits or smaller) or an 8-byte boundary for C62x or C67x. Elements of arrays are stored in the same manner as if they were individual objects.

8.2.1.7 Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (32 bit) integral type. Its value is the byte offset to the data member in the class, plus 1. The zero value is reserved to represent the NULL pointer to the data member.

8.2.1.8 Pointer to Member Function Types

Pointer to member function objects are stored as a structure with three members:

```
struct {
   short int d;
   short int i;
   union {
      void (*f) ();
      int 0;

   }
};
```

The parameter d is the offset to be added to the beginning of the class object for this pointer. The parameter i is the index into the virtual function table, offset by 1. The index enables the NULL pointer to be represented. Its value is −1 if the function is nonvirtual. The parameter f is the pointer to the member function if it is nonvirtual, when i is 0. The 0 is the offset to the virtual function pointer within the class object.

8.2.2 Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 32 bits, but they never span a 4-byte boundary.

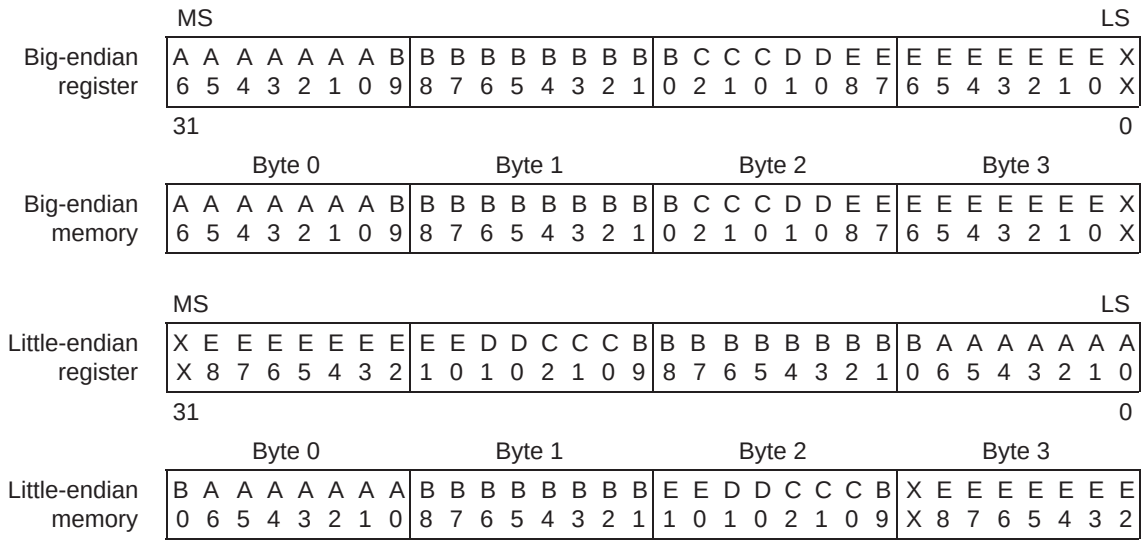
For big-endian mode, bit fields are packed into registers from most significant bit (MSB) to least significant bit (LSB) in the order in which they are defined Bit fields are packed in memory from most significant byte (MSbyte) to least significant byte (LSbyte). For little-endian mode, bit fields are packed into registers from the LSB to the MSB in the order in which they are defined, and packed in memory from LSbyte to MSbyte (see Figure 8−6).

Figure 8−6 illustrates bit field packing, using the following bit field definitions:

```
struct{
   int A:7
   int B:10
   int C:3
   int D:2
   int E:9
}x;
```

A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc. Again, storage of bit fields in memory is done with a byte-by-byte, rather than bit-by-bit, transfer.

Figure 8−6. Bit Field Packing in Big-Endian and Little-Endian Formats



Legend: X = not used
MS = most significant
LS = least significant

8.2.3 Character String Constants

In C, a character string constant is used in one of the following ways:

- To initialize an array of characters. For example:

  ```
  char s[] = ”abc”;
  ```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see section 8.8, *System Initialization*, on page 8-51.

- In an expression. For example:

  ```
  strcpy (s, ”abc”);
  ```

  When a string is used in an expression, the string itself is defined in the .const section with the .string assembler directive, along with a unique label that points to the string; the terminating 0 byte is included. For example, the following lines define the string abc, and the terminating 0 byte (the label SL5 points to the string):

  ```
        .sect ”.const”
  SL5:  .string ”abc”,0
  ```

  String labels have the form SL*n*, where *n* is a number assigned by the compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

  The label SL*n* represents the address of the string constant. The compiler uses this label to reference the string expression.

  Because strings are stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

  ```
  const char   *a = ”abc”
  a[1] = ’x’;         /* Incorrect!   */
  ```

8.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 8−2 summarizes how the compiler uses the TMS320C6000 registers.

The registers in Table 8−2 are available to the compiler for allocation to register variables and temporary expression results. If the compiler cannot allocate a register of a required type, spilling occurs. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

Objects of type double, long, long long, or long double are allocated into an odd/even register pair and are always referenced as a register pair (for example, A1:A0). The odd register contains the sign bit, the exponent, and the most significant part of the mantissa. The even register contains the least significant part of the mantissa. The A4 register is used with A5 for passing the first argument if the first argument is a double, long, long long, or long double. The same is true for B4 and B5 for the second parameter, and so on. For more information about argument-passing registers and return registers, see section 8.4, *Function Structure and Calling Conventions*.

Table 8−2. Register Usage

Register	Function Preserved By	Special Uses	Register	Function Preserved By	Special Uses
A0	Parent	−−	B0	Parent	−−
A1	Parent	−−	B1	Parent	−−
A2	Parent	−−	B2	Parent	−−
A3	Parent	Structure register (pointer to a returned structure)	B3	Parent	Return register (address to return to)
A4	Parent	Argument 1 or return value	B4	Parent	Argument 2
A5	Parent	Argument 1 or return value with A4 for doubles, longs and long longs	B5	Parent	Argument 2 with B4 for doubles, longs and long longs
A6	Parent	Argument 3	B6	Parent	Argument 4
A7	Parent	Argument 3 with A6 for doubles, longs, and long longs	B7	Parent	Argument 4 with B6 for doubles, longs, and long longs
A8	Parent	Argument 5	B8	Parent	Argument 6
A9	Parent	Argument 5 with A8 for doubles, longs, and long longs	B9	Parent	Argument 6 with B8 for doubles, longs, and long longs
A10	Child	Argument 7	B10	Child	Argument 8
A11	Child	Argument 7 with A10 for doubles, longs, and long longs	B11	Child	Argument 8 with B10 for doubles, longs, and long longs
A12	Child	Argument 9	B12	Child	Argument 10
A13	Child	Argument 9 with A12 for doubles, longs, and long longs	B13	Child	Argument 10 with B12 for doubles, longs, and long longs
A14	Child	−−	B14	Child	Data page pointer (DP)
A15	Child	Frame pointer (FP)	B15	Child	Stack pointer (SP)
A16−A31	Parent	C64x only	B16−B31	Parent	C64x only

8.4 Function Structure and Calling Conventions

The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

8.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function).

1) Arguments passed to a function are placed in registers or on the stack.

   If arguments are passed to a function, up to the first ten arguments are placed in registers A4, B4, A6, B6, A8, B8, A10, B10, A12, and B12. If longs, long longs, doubles, or long doubles are passed, they are placed in register pairs A5:A4, B5:B4, A7:A6, and so on.

   Any remaining arguments are placed on the stack (that is, the stack pointer points to the next free location; SP + *offset* points to the eleventh argument, and so on). Arguments placed on the stack must be aligned to a value appropriate for their size. An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int. An argument that is a float is passed as double if it has no prototype declared.

   A structure argument is passed as the address of the structure. It is up to the called function to make a local copy.

   For a function declared with an ellipsis indicating that it is called with varying numbers of arguments, the convention is slightly modified. The last explicitly declared argument is passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments.

   Figure 8−7 shows the register argument conventions.

Figure 8−7. Register Argument Conventions

```
int func1(int a,  int b,    int c);
A4        A4      B4        A6

int func2(int a,  float b,  int *c,  struct A d,  float e,  int f,  int g);
A4        A4      B4        A6       B6           A8        B8      A10

int func3(int a,  double b, float c, long double d);
A4        A4      B5:B4     A6       B7:B6

/* NOTE: The following function has a variable number of arguments */

int vararg(int a, int b,    int c,   int d,        ...);
A4        A4      B4        A6       stack        ...

struct A func4(   int y);
A3                A4
```

2) The calling function must save registers A0 to A9 and B0 to B9 (and A16 to A31 and B16 to B31 for C64x), if their values are needed after the call, by pushing the values onto the stack.

3) The caller (parent) calls the function (child).

4) Upon returning, the caller reclaims any stack space needed for arguments by adding to the stack pointer. This step is needed only in assembly programs that were not compiled from C/C++ code. This is because the C/C++ compiler allocates the stack space needed for all calls at the beginning of the function and deallocates the space at the end of the function.

8.4.2 How a Called Function Responds

A called function (child function) must perform the following tasks:

1) The called function (child) allocates enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function and may include the allocation of the frame pointer (FP).

   The frame pointer is used to read arguments from the stack and to handle register spilling instructions. If any arguments are placed on the stack or if the frame size exceeds 128K bytes, the frame pointer (A15) is allocated in the following manner:

   a) The old A15 is saved on the stack.

   b) The new frame pointer is set to the current SP (B15).

c) The frame is allocated by decrementing SP by a constant.

d) Neither A15 (FP) nor B15 (SP) is decremented anywhere else within this function.

If the above conditions are not met, the frame pointer (A15) is not allocated. In this situation, the frame is allocated by subtracting a constant from register B15 (SP). Register B15 (SP) is not decremented anywhere else within this function.

2) If the called function calls any other functions, the return address must be saved on the stack. Otherwise, it is left in the return register (B3) and is overwritten by the next function call.

3) If the called function modifies any registers numbered A10 to A15 or B10 to B15, it must save them, either in other registers or on the stack. The called function can modify any other registers without saving them.

4) If the called function expects a structure argument, it receives a pointer to the structure instead. If writes are made to the structure from within the called function, space for a local copy of the structure must be allocated on the stack and the local structure must be copied from the passed pointer to the structure. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.

   You must be careful to declare functions properly that accept structure arguments, both at the point where they are called (so that the structure argument is passed as an address) and at the point where they are declared (so the function knows to copy the structure to a local copy).

5) The called function executes the code for the function.

6) If the called function returns any integer, pointer, or float type, the return value is placed in the A4 register. If the function returns a double, long double, long, or long long type, the value is placed in the A5:A4 register pair.

   If the function returns a structure, the caller allocates space for the structure and passes the address of the return space to the called function in A3. To return a structure, the called function copies the structure to the memory block pointed to by the extra argument.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can actually make the call as f(&s, x). The function f then copies the return structure directly into s, performing the assignment automatically.

If the caller does not use the return structure value, an address value of 0 can be passed as the first argument. This directs the called function not to copy the return structure.

You must be careful to declare functions properly that return structures, both at the point where they are called (so that the extra argument is passed) and at the point where they are declared (so the function knows to copy the result).

7) Any register numbered A10 to A15 or B10 to B15 that was saved in step 3 is restored.

8) If A15 was used as a frame pointer (FP), the old value of A15 is restored from the stack. The space allocated for the function in step 1 is reclaimed at the end of the function by adding a constant to register B15 (SP).

9) The function returns by jumping to the value of the return register (B3) or the saved value of the return register.

8.4.3 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through register A15 (FP) or through register B15 (SP), one of which points to the top of the stack. Since the stack grows toward smaller addresses, the local and argument data for a function are accessed with a positive offset from FP or SP. Local variables, temporary storage, and the area reserved for stack arguments to functions called by this function are accessed with offsets smaller than the constant subtracted from FP or SP at the beginning of the function.

Stack arguments passed to this function are accessed with offsets greater than or equal to the constant subtracted from register FP or SP at the beginning of the function. The compiler attempts to keep register arguments in their original registers if optimization is used or if they are defined with the register keyword. Otherwise, the arguments are copied to the stack to free those registers for further allocation.

For information on whether FP or SP is used to access local variables, temporary storage, and stack arguments, see section 8.4.2, *How a Called Function Responds*, on page 8-20. For more information on the C/C++ System stack, see section 8.1.2, *C/C++ System Stack*, on page 8-4

8.5 Interfacing C and C++ With Assembly Language

The following are ways to use assembly language with C/C++ code:

- ❑ Use separate modules of assembled code and link them with compiled C/C++ modules (see section 8.5.1).
- ❑ Use intrinsics in C/C++ source to directly call an assembly language statement (see section 8.5.2 on page 8-26).
- ❑ Use inline assembly language embedded directly in the C/C++ source (see section 8.5.8 on page 8-43).
- ❑ Use assembly language variables and constants in C/C++ source (see section 8.5.9 on page 8-44).

8.5.1 Using Assembly Language Modules With C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in section 8.4, *Function Structure and Calling Conventions*, on page 8-19 and the register conventions defined in section 8.3, *Register Conventions*, on page 8-17. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- ❑ All functions, whether they are written in C/C++ or assembly language, must follow the register conventions outlined in section 8.3, *Register Conventions*, on page 8-17.
- ❑ You must preserve registers A10 to A15, B3, and B10 to B15, and you may need to preserve A3. If you use the stack normally, you do not need to explicitly preserve the stack. In other words, you are free to use the stack inside a function as long as you pop everything you pushed before your function exits. You can use all other registers freely without preserving their contents.
- ❑ Interrupt routines must save *all* the registers they use. For more information, see section 8.6, *Interrupt Handling*, on page 8-46.

- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in section 8.4.1, *How a Function Makes a Call*, on page 8-19.

  Remember that only A10 to A15 and B10 to B15 are preserved by the C/C++ compiler. C/C++ functions can alter any other registers, save any other registers whose contents need to be preserved by pushing them onto the stack before the function is called, and restore them after the function returns.

- Functions must return values correctly according to their C/C++ declarations. Integers and 32-bit floating-point (float) values are returned in A4. Doubles, long doubles, longs, and long longs are returned in A5:A4. Structures are returned by copying them to the address in A3.

- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C/C++ startup routine assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.

- The compiler assigns linknames to all external objects. Thus, when you are writing assembly language code, you must use the same linknames as those assigned by the compiler. For identifiers that are to be used only in an assembly language module or modules, the identifier should not begin with an underscore (_). See section 7.8, *Generating Linknames*, on page 7-33, for more information.

- Any object or function declared in assembly language that is accessed or called from C/C++ must be declared with the .def or .global directive in the assembly language modifier. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.

Example 8−1 illustrates a C++ function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 8−1. Calling an Assembly Language Function From C/C++

(a) C program

```
extern "C" {
extern int asmfunc(int a);/* declare external as function*/
int gvar = 4;             /* define global variable       */
}

void main()
{
   int i = 5;

   i = asmfunc(i);      /* call function normally        */
```

(b) Assembly language program

```
   .global _asmfunc
   .global _gvar
_asmfunc:
   LDW     *+b14(_gvar),A3
   NOP     4
   ADD     a3,a4,a3
   STW     a3,*b14(_gvar)
   MV      a3,a4
   B       b3
   NOP     5
```

In the C++ program in Example 8−1, the extern declaration of asmfunc is optional because the return type is int. Like C/C++ functions, you need to declare assembly functions only if they return noninteger values or pass noninteger parameters.

Note: SP Semantics

The stack pointer must always be 8-byte aligned. This is automatically performed by the C compiler and system initialization code in the run-time support libraries. Any hand assembly code that calls a function defined in C or linear assembly source should also reserve a multiple of 8 bytes on the stack.

Note: Stack Allocation

Even though the compiler guarantees a doubleword alignment of the stack and the stack pointer (SP) points to the next free location in the stack space, there is only enough guaranteed room to store one 32-bit word at that location. The called function must allocate space to store the doubleword.

8.5.2 Using Intrinsics to Access Assembly Language Statements

The C6000 compiler recognizes a number of intrinsic operators. Intrinsics allow you to express the meaning of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Intrinsics are used like functions; you can use C/C++ variables with these intrinsics, just as you would with any normal function.

The intrinsics are specified with a leading underscore, and are accessed by calling them as you do a function. For example:

```
int x1, x2, y;
y = _sadd(x1, x2);
```

The intrinsics listed in Table 8−3 are included for all C6000 devices. They correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

Note: Intrinsic Instructions in C versus Assembly Language

In some instances, an intrinsic's exact corresponding assembly language instruction may not be used by the compiler. When this is the case, the meaning of the program does not change.

See Table 8−4 on page 8-31 for the listing of C64x-specific intrinsics. See Table 8−5, on page 8-35, for the listing of C67x-specific intrinsics.

Table 8−3. TMS320C6000 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs(**int *src***);** int **_labs(**long *src***);**	**ABS**	Returns the saturated absolute value of src
int **_add2(**int *src1***,** int *src2***);**	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add does not affect the upper half add.
ushort & **_amem2(**void **ptr***);**	**LDHU** **STHU**	Allows aligned loads and stores of 2 bytes to memory†
const ushort & **_amem2_const(**const void **ptr***);**	**LDHU**	Allows aligned loads of 2 bytes from memory†
uint & **_amem4(**void **ptr***);**	**LDW** **STW**	Allows aligned loads and stores of 4 bytes to memory†
const uint & **_amem4_const(**const void **ptr***);**	**LDW**	Allows aligned loads of 4 bytes from memory†
double & **_amemd8(**void **ptr***);**	**LDW/LDW** **STW/STW**	Allows aligned loads and stores of 8 bytes to memory†‡ For C64x _amemd corresponds to different assembly instructions than when used with other C6000 devices; see Table 8−4 for specifics.
const double & **_amemd8_const(**const void **ptr***);**	**LDDW**	Allows aligned loads of 8 bytes from memory†‡
uint **_clr(**uint *src2,* uint *csta*, uint *cstb***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by csta and cstb, respectively.
uint **_clrr(**uint *src2*, int *src1***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by the lower 10 bits of src1.
ulong **_dtol(**double *src***);**		Reinterprets double register pair src as an unsigned long register pair

† See the *TMS320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−3. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_ext(**int *src2,* uint *csta*, uint *cstb***);**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; csta and cstb are the shift left and shift right amounts, respectively.
int **_extr(**int *src2*, int *src1***)**	**EXT**	Extracts the specified field in src2, sign-extended to 32 bits. The extract is performed by a shift left followed by a signed shift right; the shift left and shift right amounts are specified by the lower 10 bits of src1.
uint **_extu(**uint *src2,* uint *csta*, uint *cstb***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right;csta and cstb are the shift left and shift right amounts, respectively.
uint **_extur(**uint *src2*, int *src1***);**	**EXTU**	Extracts the specified field in src2, zero-extended to 32 bits. The extract is performed by a shift left followed by a unsigned shift right;the shift left and shift right amounts are specified by the lower 10 bits of src1.
uint **_ftoi(**float *src***);**		Reinterprets the bits in the float as an unsigned. For example: _ftoi (1.0) == 1065353216U
uint **_hi(**double *src***);**		Returns the high (odd) register of a double register pair
double **_itod(**uint *src2*, uint *src1***)**		Builds a new double register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register
float **_itof(**uint *src***);**		Reinterprets the bits in the unsigned as a float. For example: _itof (0x3f800000)==1.0
long long **_itoll(**uint *src2*, uint *src1***)**		Builds a new long long register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register

† See the *TMS320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−3. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
uint **_lo(**double *src***);**		Returns the low (even) register of a double register pair
uint **_lmbd(**uint s*rc1,* uint *src2***);**	**LMBD**	Searches for a leftmost 1 or 0 of src2 determined by the LSB of src1. Returns the number of bits up to the bit change.
double **_ltod(**long *src***);**		Reinterprets long register pair src as a double register pair
int **_mpy(**int *src1***,** int *src2***);**	**MPY**	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.
int **_mpyus(**uint *src1***,** int *src2***);**	**MPYUS**	
int **_mpysu(**int *src1***,** uint *src2***);**	**MPYSU**	
uint **_mpyu(**uint *src1***,** uint *src2***);**	**MPYU**	
int **_mpyh(**int *src1***,** int *src2***);**	**MPYH**	Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.
int **_mpyhus(**uint *src1***,** int *src2***);**	**MPYHUS**	
int **_mpyhsu(**int *src1***,** uint *src2***);**	**MPYHSU**	
uint **_mpyhu(**uint *src1***,** uint *src2***);**	**MPYHU**	
int **_mpyhl(**int *src1*, int *src2***);**	**MPYHL**	Multiplies the 16 MSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.
int **_mpyhuls(**uint *src1*, int *src2***);**	**MPYHULS**	
int **_mpyhslu(**int *src1*, uint *src2***);**	**MPYHSLU**	
uint **_mpyhlu(**uint *src1*, uint *src2***);**	**MPYHLU**	
int **_mpyhl(**int *src1***,** int *src2***);**	**MPYLH**	Multiplies the 16 LSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.
int **_mpyluhs(**uint *src1*, int *src2***);**	**MPYLUHS**	
int **_mpylshu(**int *src1*, uint *src2***);**	**MPYLSHU**	
uint **_mpylhu(**uint *src1*, uint *src2***);**	**MPYLHU**	
void **_nassert(**int**);**		Generates no code. Tells the optimizer that the expression declared with the assert function is true; this gives a hint to the optimizer as to what optimizations might be valid.
uint **_norm(**int *src2***);** uint **_lnorm(**long *src2***);**	**NORM**	Returns the number of bits up to the first nonredundant sign bit of src2

† See the *TMS320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−3. TMS320C6000 C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int _**sadd(**int *src1,* int *src2***);** long _**lsadd(**int *src1,* long *src2***);**	**SADD**	Adds src1 to src2 and saturates the result. Returns the result
int _**sat(**long *src2***);**	**SAT**	Converts a 40-bit long to a 32-bit signed int and saturates if necessary
uint _**set(**uint *src2,* uint *csta*, uint *cstb***);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by csta and cstb, respectively.
unit **_setr(**unit *src2*, int *src1***);**	**SET**	Sets the specified field in src2 to all 1s and returns the src2 value. The beginning and ending bits of the field to be set are specified by the lower ten bits of src1.
int _**smpy (**int *src1*, int *sr2***);** int _**smpyh (**int *src1*, int *sr2***);** int _**smpyhl (**int *src1*, int *sr2***);** int _**smpylh (**int *src1*, int *sr2***);**	**SMPY** **SMPYH** **SMPYHL** **SMPYLH**	Multiplies src1 by src2, left shifts the result by 1, and returns the result. If the result is 0x8000 0000, saturates the result to 0x7FFF FFFF
int _**sshl (**int *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result
int _**ssub (**int *src1,* int *src2***);** long _**lssub (**int *src1,* long *src2***);**	**SSUB**	Subtracts src2 from src1, saturates the result, and returns the result
uint _**subc (**uint *src1,* uint *src2***);**	**SUBC**	Conditional subtract divide step
int _**sub2 (**int *src1*, int *src2***);**	**SUB2**	Subtracts the upper and lower halves of src2 from the upper and lower halves of src1, and returns the result. Borrowing in the lower half subtract does not affect the upper half subtract.

† See the *TMS320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

The intrinsics listed in Table 8−4 are included only for C64x devices. The intrinsics shown correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

See Table 8−3 on page 8-27 for the listing of generic C6000 intrinsics. See Table 8−5 on page 8-35 for the listing of C67x-specific intrinsics.

Table 8−4. TMS320C64x C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs2(**int *src***);**	**ABS2**	Calculates the absolute value for each 16-bit value
int **_add4(**int *src1***,** int *src2***);**	**ADD4**	Performs 2s-complement addition to pairs of packed 8-bit numbers
long long & **_amem8(**void **ptr***);*	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory.
const long long & **_amem8_const(**const void **ptr***);*	**LDDW**	Allows aligned loads of 8 bytes from memory.‡
double & **_amemd8(**void **ptr***);*	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory†‡ For C64x _amemd corresponds to different assembly instructions than when used with other C6000 devices; see Table 8−3.
const double & **_amemd8_const(**const void **ptr***);*	**LDDW**	Allows aligned loads of 8 bytes from memory†‡
int **_avg2**(int *src1***,** int *src2***);**	**AVG2**	Calculates the average for each pair of signed 16-bit values
uint **_avgu4(**uint**,** uint**);**	**AVGU4**	Calculates the average for each pair of signed 8-bit values
uint **_bitc4(**uint *src***);**	**BITC4**	For each of the 8-bit quantities in src, the number of 1 bits is written to the corresponding position in the return value
uint **_bitr(**uint *src***);**	**BITR**	Reverses the order of the bits
int **_cmpeq2(**int *src1***,** int *src2***);**	**CMPEQ2**	Performs equality comparisons on each pair of 16-bit values. Equality results are packed into the two least-significant bits of the return value.
int **_cmpeq4(**int *src1***,** int *src2***);**	**CMPEQ4**	Performs equality comparisons on each pair of 8-bit values. Equality results are packed into the four least-significant bits of the return value.
int **_cmpgt2(**int *src1***,** int *src2***);**	**CMPGT2**	Compares each pair of signed 16-bit values. Results are packed into the two least-significant bits of the return value.

† See the *Tms320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−4. TMS320C64x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
uint _**cmpgtu4(**uint *src1***,** uint *src2***);**	**CMPGTU4**	Compares each pair of 8-bit values. Results are packed into the four least-significant bits of the return value.
uint _**deal(**uint *src***);**	**DEAL**	The odd and even bits of src are extracted into two separate 16-bit values.
int _**dotp2(**int *src1***,** int *src2***);** double _**ldotp2(**int *src1***,** int *src2***);**	**DOTP2** **LDOTP2**	The product of the signed lower 16-bit values of src1 and src2 is added to the product of the signed upper 16-bit values of src1 and src2. The _lo and _hi intrinsics are needed to access each half of the 64-bit integer result.
int _**dotpn2(**int *src1***,** int *src2***);**	**DOTPN2**	The product of the signed lower 16-bit values of src1 and src2 is subtracted from the product of the signed upper 16-bit values of src1 and src2.
int _**dotpnrsu2(**int *src1***,** uint *src2***);**	**DOTPNRSU2**	The product of the lower unsigned 16-bit values in src1 and src2 is subtracted from the product of the signed upper 16-bit values of src1 and src2. 2^15 is added and the result is sign shifted right by 16.
int _**dotprsu2(**int *src1***,** uint *src2***);**	**DOTPRSU2**	The product of the first signed pair of 16-bit values is added to the product of the unsigned second pair of 16-bit values. 2^15 is added and the result is sign shifted by 16.
int _**dotprsu4(**int *src1***,** uint *src2***);** uint _**dotpu4(**uint *src1***,** uint *src2***);**	**DOTPRSU4** **DOTPU4**	For each pair of 8-bit values in src1 and src2, the 8-bit value from src1 is multiplied with the 8-bit value from src2. The four products are summed together.
int _**gmpy4(**int *src1*, int *src2***);**	**GMPY4**	Performs the galois field multiply on four values in src1 with four parallel values in src2. The four products are packed into the return value.

int _**max2 (**int *src1***,** int *src2***);**
int _**min2(**int *src1***,** int *src2*);
uint _**maxu4(**uint *src1*, uint *src2*);
uint _**minu4(**uint *src1*, uint *src2***);**
	MAX2
	MIN2
	MAX4
	MINU4
		Places the larger/smaller of each pair of values in the corresponding position in the return value. Values can be 16-bit signed or 8-bit unsigned.

† See the *Tms320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−4. TMS320C64x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
ushort & **_mem2(**void * *ptr***);**	**LDB/LDB** **STB/STB**	Allows unaligned loads and stores of 2 bytes to memory†
const ushort & **_mem2_const(**const void * *ptr***);**	**LDB/LDB**	Allows unaligned loads of 2 bytes to memory†
uint & **_mem4(**void * *ptr***);**	**LDNW** **STNW**	Allows unaligned loads and stores of 4 bytes to memory†
const uint & **_mem4_const(**const void * *ptr***);**	**LDNW**	Allows unaligned loads of 4 bytes from memory†
long long & **_mem8(**void * *ptr***);**	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory†
const long long & **_mem8_const(**const void * *ptr***);**	**LDNDW**	Allows unaligned loads of 8 bytes from memory†
double & **_memd8(**void * *ptr***)**	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory†‡
const double & **_memd8_const(**const void * *ptr***)**	**LDNDW**	Allows unaligned loads of 8 bytes from memory†‡
double **_mpy2(**int *src1***,** int *src2***);**	**MPY2**	Returns the products of the lower and higher 16-bit values in src1 and src2
double **_mpyhi(**int *src1***,** int *src2***);** double **_mpyli(**int *src1***,** int *src2***);**	**MPYHI** **MPYLI**	Produces a 16 by 32 multiply. The result is placed into the lower 48 bits of the returned double. Can use the upper or lower 16 bits of src1.
int **_mpyhir(**int *src1***,** int *src2***);** int **_mpylir(**int *src1***,** int *src2***);**	**MPYHIR** **MPYLIR**	Produces a signed 16 by 32 multiply. The result is shifted right by 15 bits. Can use the upper or lower 16 bits of src1.
double **_mpysu4 (**int *src1***,** uint *src2*); double **_mpyu4 (**uint *src1***,** uint *src2***);**	**MPYSU4** **MPYU4**	For each 8-bit quantity in src1 and src2, performs an 8-bit by 8-bit multiply. The four 16-bit results are packed into a double. The results can be signed or unsigned.
int **_mvd (**int *src2***);**	**MVD**	Moves the data from src2 to the return value over four cycles using the multiplier pipeline
uint **_pack2 (**uint *src1***,** uint *src2***);** uint **_packh2 (**uint *src1***,** uint *src2***);**	**PACK2** **PACKH2**	The lower/upper halfwords of src1 and src2 are placed in the return value.

† See the *Tms320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−4. TMS320C64x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
uint _**packh4 (**uint *src1***,** uint *src2***);** uint _**packl4 (**uint *src1***,** uint *src2***);**	**PACKH4** **PACKL4**	Packs alternate bytes into return value. Can pack high or low bytes.
uint _**packhl2 (**uint *src1***,** uint *src2***);** uint _**packlh2 (**uint *src1***,** uint *src2***);**	**PACKHL2** **PACKLH2**	The upper/lower halfword of src1 is placed in the upper halfword the return value. The lower/upper halfword of src2 is placed in the lower halfword the return value.
uint _**rotl (**uint *src1***,** uint *src2***);**	**ROTL**	Rotates src2 to the left by the amount in src1
int _**sadd2(**int *src1,* int *src2***);** int _**saddus2(**uint *src1,* int *src2***);**	**SADD2** **SADDUS2**	Performs saturated addition between pairs of 16-bit values in src1 and src2. Values for src1 can be signed or unsigned.
uint _**saddu4(**uint *src1***,** uint *src2***);**	**SADDU4**	Performs saturated addition between pairs of 8-bit unsigned values in src1 and src2.
uint _**shfl (**uint *src2***);**	**SHFL**	The lower 16 bits of src2 are placed in the even bit positions, and the upper 16 bits of src are placed in the odd bit positions.
uint _**shlmb (**uint *src1***,** uint *src2***);** uint _**shrmb (**uint *src1***,** uint *src2***);**	**SHLMB** **SHRMB**	Shifts src2 left/right by one byte, and the most/least significant byte of src1 is merged into the least/most significant byte position.
int _**shr2 (**int *src1*, uint *src2***);** uint **shru2 (**uint *src1***,** uint *src2***);**	**SHR2** **SHRU2**	For each 16-bit quantity in src2, the quantity is arithmetically or logically shifted right by src1 number of bits. src2 can contain signed or unsigned values
double _**smpy2 (**int *src1*, int *sr2***);**	**SMPY2**	Performs 16-bit multiplication between pairs of signed packed 16-bit values, with an additional 1 bit left-shift and saturate into a double result.
int _**spack2 (**int *src1*, int *sr2***);**	**SPACK2**	Two signed 32-bit values are saturated to 16-bit values and packed into the return value
uint _**spacku4 (**int *src1*, int *sr2***);**	**SPACKU4**	Four signed 16-bit values are saturated to 8-bit values and packed into the return value
int _**sshvl (**int *src2*, int *src1***);** int _**sshvr (**int *src2*, int *src1***);**	**SSHVL** **SSHVR**	Shifts src2 to the left/right src1 bits. Saturates the result if the shifted value is greater than MAX_INT or less than MIN_INT.
int _**sub4 (**int *src1*, int *src2***);**	**SUB4**	Performs 2s-complement subtraction between pairs of packed 8-bit values

† See the *Tms320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

Table 8−4. TMS320C64x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int _**subabs4 (**int *src1*, int *src2***);**	**SUBABS4**	Calculates the absolute value of the differences for each pair of packed 8-bit values
uint _**swap4 (**uint *src***);**	**SWAP4**	Exchanges pairs of bytes (an endian swap) within each 16-bit value
uint _**unpkhu4 (**uint *src***);**	**UNPKHU4**	Unpacks the two high unsigned 8-bit values into unsigned packed 16-bit values
uint _**unpklu4 (**uint *src***);**	**UNPKLU4**	Unpacks the two low unsigned 8-bit values into unsigned packed 16-bit values
uint _**xpnd2 (**uint *src***);**	**XPND2**	Bits 1 and 0 of src are replicated to the upper and lower halfwords of the result, respectively.
uint _**xpnd4 (**uint *src***);**	**XPND4**	Bits 3 and 0 of src are replicated to bytes 3 through 0 of the result.

† See the *Tms320C6000 Programmer's Guide* for more information.
‡ See section 8.5.3, *Using Unaligned Data and 64-Bit Values*, for details on manipulating 8-byte data quantities.

The intrinsics listed in Table 8−5 are included only for C67x devices. The intrinsics shown correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

See Table 8−3 on page 8-27 for the listing of generic C6000 intrinsics. See Table 8−4 on page 8-31 for the listing of C64x-specific intrinsics.

Table 8−5. TMS320C67x C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_dpint(**double *src***);**	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register
double _**fabs(**double *src***);** float _**fabsf(**float *src***);**	**ABSDP** **ABSSP**	Returns absolute value of src
double _**mpyid (**int *src1***,** int *src2***);**	**MPYID**	Produces a signed integer multiply. The result is placed in a register pair.
double **_rcpdp(**double *src***);**	**RCPDP**	Computes the approximate 64-bit double reciprocal
float **_rcpsp(**float *src***);**	**RCPSP**	Computes the approximate 32-bit float reciprocal

Table 8−5. TMS320C67x C/C++ Compiler Intrinsics (Continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
double **_rsqrdp(**double *src***);**	**RSQRDP**	Computes the approximate 64-bit double square root reciprocal
float **_rsqrsp(**float *src***);**	**RSQRSP**	Computes the approximate 32-bit float square root reciprocal
int **_spint (**float**);**	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register

8.5.3 Using Unaligned Data and 64-Bit Values

The C64x family has support for unaligned loads and stores of 64-bit and 32-bit values via the use of the _mem8, _memd8, and _mem4 intrinsics. The _lo and _hi intrinsics are useful for extracting the two 32-bit portions from a 64-bit double. Example 8−2 shows the usage of the _lo, _hi, _mem8, and _memd8 intrinsics.

Example 8−2. Using the _lo and _hi Intrinsics

```
void load_longlong_unaligned(void *a, int *high, int *low)
{
  double d = _memd8(a);

  *high = _hi(d);
  *low  = _lo(d);
}

void store_longlong_unaligned(void *a, int high, int low)
{
  double d  = _itod(high, low);
  _mem8d(a) = d;
}
```

Example 8−3. Using the _lo and _hi Intrinsics With long long Integers

```
void alt_load_longlong_unaligned(void *a, int *high, int *low)
{
  long long p = _mem8(a);

  *high = p >> 32;
  *low  = (unsigned int) p;
}

void alt_store_longlong_unaligned(void *a, int high, int low)
{
  long long p = _itoll(high, low);
  _mem8(a)    = p;
}
```

8.5.4 Using MUST_ITERATE and _nassert to Enable SIMD and Expand Compiler Knowledge of Loops

Through the use of MUST_ITERATE and _nassert, you can guarantee that a loop executes a certain number of times.

This example tells the compiler that the loop is guaranteed to run exactly 10 times:

```
#pragma MUST_ITERATE(10,10);
for (i = 0; i < trip_count; i++) { ...
```

MUST_ITERATE can also be used to specify a range for the trip count as well as a factor of the trip count. For example:

```
#pragma MUST_ITERATE(8,48,8);
for (i = 0; i < trip; i++) { ...
```

This example tells the compiler that the loop executes between 8 and 48 times and that the trip variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The compiler can now use all this information to generate the best loop possible by unrolling better even when the −m*i*n option is used to specify that interrupts do occur every *n* cycles.

The *TMS320C6000 Programmer's Guide* states that one of the ways to refine C/C++ code is to use word accesses to operate on 16-bit data stored in the high and low parts of a 32-bit register. Examples using casts to int pointers are shown with the use of intrinsics to use certain instructions like _mpyh. This can be automated by using the _nassert(); intrinsic to specify that 16-bit short arrays are aligned on a 32-bit (word) boundary.

The following two examples generate the same assembly code:

❑ **Example 1**

```
int dot_product(short *x, short *y, short z)
{
    int *w_x = (int *)x;
    int *w_y = (int *)y;
    int sum1 = 0, sum2 = 0, i;
    for (i = 0; i < z/2; i++)
    {
        sum1 += _mpy(w_x[i], w_y[i]);
        sum2 += _mpyh(w_x[i], w_y[i]);
    }
    return (sum1 + sum2);
}
```

❑ **Example 2**

```
int dot_product(short *x, short *y, short z)
{
    int sum = 0, i;

    _nassert (((int)(x) & 0x3) == 0);
    _nassert (((int)(y) & 0x3) == 0);
    #pragma MUST_ITERATE(20, , 4);
    for (i = 0; i < z; i++) sum += x[i] * y[i];
    return sum;
}
```

Note: C++ Syntax for _nassert

In C++ code, _nassert is part of the standard namespace. Thus, the correct syntax is std::_nassert().

8.5.5 Methods to Align Data

In the following code, the _nassert tells the compiler, for every invocation of f(), that ptr is aligned to an 8-byte boundary. Such an assertion often leads to the compiler producing code which operates on multiple data values with a single instruction, also known as SIMD (single instruction multiple data) optimization.

```
void f(short *ptr)
{
   _nassert((int) ptr % 8 == 0)

   ; a loop operating on data accessed by ptr
}
```

The following subsections describe methods you can use to ensure the data referenced by ptr is aligned. You have to employ one of these methods at every place in your code where f() is called.

8.5.5.1 Base Address of an Array

An argument such as ptr is most commonly passed the base address of an array, for example:

```
short buffer[100];

...

f(buffer);
```

When compiling for C64x devices, such an array is automatically aligned to an 8-byte boundary. When compiling for C62x or C67x, such an array is automatically aligned to 4-byte boundary, or, if the base type requires it, an 8-byte boundary. This is true whether the array is global, static, or local. This automatic alignment is all that is required to achieve SIMD optimization on those respective devices. You still need to include the _nassert because, in the general case, the compiler cannot guarantee that ptr holds the address of a properly aligned array.

If you always pass the base address of an array to pointers like ptr, then you can use the following macro to reflect that fact.

```
#if   defined(_TMS320C6400)
   #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 8 == 0)
#elif defined(_TMS320C6200) || defined(_TMS320C6700)
   #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 4 == 0)
#else
   #define ALIGNED_ARRAY(ptr) /* empty */
#endif

void f(short *ptr)
{
   ALIGNED_ARRAY(ptr);

   ; a loop operating on data accessed by ptr
}
```

The macro will work regardless of which C6x device you build for, or if you port the code to another target.

8.5.5.2 Offset from the Base of an Array

A more rare case is to pass the address of an offset from an array, for example:

```
f(&buffer[3]);
```

This code passes an unaligned address to ptr, thus violating the presumption coded in the _nassert(). There is no direct remedy for this case. Avoid this practice whenever possible.

8.5.5.3 Dynamic Memory Allocation

Ordinary dynamic memory allocation does not guarantee that the address of the buffer is aligned, for example:

```
buffer = malloc(100 * sizeof(short));
```

You should use memalign() with an alignment of 8 instead, for example:

```
buffer = memalign(8, 100 * sizeof(short));
```

If you are using BIOS memory allocation routines, be sure to pass the alignment factor as the last argument using the syntax that follows:

buffer = **MEM_alloc(***segid***,** 100 * sizeof(short), **8);**

See the *TMS320C6000 DSP/BIOS Help* for more information about BIOS memory allocation routines and the *segid* parameter in particular.

8.5.5.4 Member of a Structure or Class

Arrays which are members of a structure or a class are aligned only as the base type of the array requires. The automatic alignment described in section 8.5.5.1, *Base Address of an Array*, does not occur.

Example 8−4. An Array in a Structure

```
struct s
{
   ...
   short buf1[50];
   ...
} g;

...

f(g.buf1);
```

Example 8−5. An Array in a Class

```
class c
{
   public :
      short buf1[50];
      void mfunc(void);
   ...
};

void c::mfunc()
{
   f(buf1);
   ...
}
```

The most straightforward way to align an array in a structure or class is to declare, right before the array, a scalar that requires the desired alignment. So, if you want 8-byte alignment, use a long or double. If you want 4-byte alignment, use an int or float. For example:

```
struct s
{
   long  not_used;    /* 8-byte aligned      */
   short buffer[50];  /* also 8-byte aligned */
   ...
};
```

If you want to declare several arrays contiguously, and maintain a given alignment, you can do so by keeping the array size, measured in bytes, an even multiple of the desired alignment. For example:

```
struct s
{
   long  not_used; /* 8-byte aligned      */
   short buf1[50]; /* also 8-byte aligned */
   short buf2[50]; /* 4-byte aligned      */
   ...
};
```

Because the size of buf1 is 50 * 2-bytes per short = 100 bytes, and 100 is an even multiple of 4, not 8, buf2 is only aligned on a 4-byte boundary. Padding buf1 out to 52 elements makes buf2 8-byte aligned.

Within a structure or class, there is no way to enforce an array alignment greater than 8. For the purposes of SIMD optimization, this is not necessary.

Note: Alignment With Program-Level Optimization

In most cases program-level optimization (see section 3.6 on page 3-20) entails compiling all of your source files with a single invocation of the compiler, while using the −pm −o3 options. This allows the compiler to see all of your source code at once, thus enabling optimizations that are rarely applied otherwise. Among these optimizations is seeing that, for instance, all of the calls to the function f() are passing the base address of an array to ptr, and thus ptr is always correctly aligned for SIMD optimization. In such a case, the _nassert() is not required. The compiler automatically determines that ptr must be aligned, and produces the optimized SIMD instructions.

8.5.6 SAT Bit Side Effects

The saturated intrinsic operations define the SAT bit if saturation occurs. The SAT bit can be set and cleared from C/C++ code by accessing the control status register (CSR). The compiler uses the following steps for generating code that accesses the SAT bit:

1) The SAT bit becomes undefined by a function call or a function return. This means that the SAT bit in the CSR is valid and can be read in C/C++ code until a function call or until a function returns.

2) If the code in a function accesses the CSR, then the compiler assumes that the SAT bit is live across the function, which means:

   - The SAT bit is maintained by the code that disables interrupts around software pipelined loops.
   - Saturated instructions cannot be speculatively executed.

3) If an interrupt service routine modifies the SAT bit, then the routine should be written to save and restore the CSR.

8.5.7 IRP and AMR Conventions

There are certain assumptions that the compiler makes about the IRP and AMR control registers. The assumptions should be enforced in all programs and are as follows:

1) The AMR must be set to 0 upon calling or returning from a function. A function does not have to save and restore the AMR, but must ensure that the AMR is 0 before returning.

2) The AMR must be set to 0 when interrupts are enabled, or the SAVE_AMR and STORE_AMR macros should be used in all interrupts (see section 8.6.2).

3) The IRP can be safely modified only when interrupts are disabled.

4) The IRP's value must be saved and restored if you use the IRP as a temporary register.

8.5.8 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see section 7.6, *The asm Statement*, on page 7-17.

The asm statement is useful for inserting comments in the compiler output. Simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

Note: Using the asm Statement

Keep the following in mind when using the asm statement:

- Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.
- Avoid inserting jumps or labels into C/C++ code because they can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
- Do not change the value of a C/C++ variable when using an asm statement. This is because the compiler does not verify such statements. They are inserted as is into the assembly code, and potentially can cause problems if you are not sure of their effect.
- Do not use the asm statement to insert assembler directives that change the assembly environment.
- Avoid creating assembly macros in C code and compiling with the −g (debug) option. The C environment's debug information and the assembly macro expansion are not compatible.

8.5.9 Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables or constants defined in assembly language. There are several methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a constant.

8.5.9.1 Accessing Assembly Language Global Variables

Accessing uninitialized variables from the .bss section or a section named with .usect is straightforward:

1) Use the .bss or .usect directive to define the variable.

2) When you use .usect, the variable is defined in a section other than .bss and therefore must be declared far in C.

3) Use the .def or .global directive to make the definition external.

4) Precede the name with an underscore in assembly language.

5) In C/C++, declare the variable as *extern* and access it normally.

Example 8−6 shows how you can access a variable defined in .bss.

Example 8−6. Accessing an Assembly Language Variable From C

(a) C program

```
extern int var1;        /* External variable       */
far extern int var2;    /* External variable       */
var1 = 1;               /* Use the variable        */
var2 = 1;               /* Use the variable        */
```

(b) Assembly language program

```
* Note the use of underscores in the following lines

       .bss    _var1,4,4    ; Define the variable
       .global var1         ; Declare it as external

_var2  .usect  "mysect",4,4 ; Define the variable
       .global _var2        ; Declare it as external
```

8.5.9.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set, .def, and .global directives, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

For normal variables defined in C/C++ or assembly language, the symbol table contains the *address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the compiler attempts to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C/C++ is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 8−7.

Example 8−7. Accessing an Assembly Language Constant From C

(a) C program

```
extern int table_size; /*external ref */
#define TABLE_SIZE ((int) (&table_size))
       .            /* use cast to hide address-of */
       .
       .
for (i=0; i<TABLE_SIZE; ++i)
                    /* use like normal symbol */
```

(b) Assembly language program

```
_table_size   .set   10000        ; define the constant
              .global _table_size ; make it global
```

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 8−7, int is used. You can reference linker-defined symbols in a similar manner.

8.6 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

8.6.1 Saving Registers During Interrupts

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. The compiler handles register preservation if the interrupt service routine is written in C/C++.

8.6.2 Using C/C++ Interrupt Routines

A C/C++ interrupt routine is like any other C/C++ function in that it can have local variables and register variables; however, it should be declared with no arguments and should return void. C/C++ interrupt routines can allocate up to 32K on the stack for local variables. For example:

```
interrupt void example (void)
{
...
}
```

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler attempts to define are saved and restored. However, if a C/C++ interrupt routine *does* call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the routine saves all usable registers if any other functions are called. Interrupts branch to the interrupt return pointer (IRP). Do not call interrupt handling functions directly.

Interrupts can be handled *directly* with C/C++ functions by using the interrupt pragma or the interrupt keyword. For more information, see section 7.7.13, *The INTERRUPT Pragma*, on page 7-27, and section 7.4.3, *The interrupt Keyword*, on page 7-10.

You are responsible for handling the AMR control register and the SAT bit in the CSR correctly inside an interrupt. By default, the compiler does not do anything extra to save/restore the AMR and the SAT bit. Macros for handling the SAT bit and the AMR register are included in the c6x.h header file.

For example, you are using circular addressing in some hand assembly code (that is, the AMR does not equal 0). This hand assembly code can be interrupted into a C code interrupt service routine. The C code interrupt service routine assumes that the AMR is set to 0. You need to define a local unsigned int temporary variable and call the SAVE_AMR and RESTORE_AMR macros at the beginning and end of your C interrupt service routine to correctly save/restore the AMR inside the C interrupt service routine.

Example 8−8. AMR and SAT Handling

```
#include <c6x.h>

interrupt void interrupt_func()
{
   unsigned int temp_amr;
   /* define other local variables used inside interrupt */

   /* save the AMR to a temp location and set it to 0 */
   SAVE_AMR(temp_amr);

   /* code and function calls for interrupt service routine */
   ...

   /* restore the AMR for you hand assembly code before exiting
*/
   RESTORE_AMR(temp_amr);
}
```

If you need to save/restore the SAT bit (i.e. you were performing saturated arithmetic when interrupted into the C interrupt service routine which may also perform some saturated arithmetic) in your C interrupt service routine, it can be done in a similar way as the above example using the SAVE_SAT and RESTORE_SAT macros.

8.6.3 Using Assembly Language Interrupt Routines

You can handle interrupts with assembly language code as long as you follow the same register conventions the compiler does. Like all assembly functions, interrupt routines can use the stack, access global C/C++ variables, and call C/C++ functions normally. When calling C/C++ functions, be sure that any registers listed in Table 8−2 on page 8-18 are saved, because the C/C++ function can modify them.

8.7 Run-Time-Support Arithmetic Routines

The run-time-support library contains a number of assembly language functions that provide arithmetic routines for C/C++ math operations that the C6000 instruction set does not provide, such as integer division, integer remainder, and floating-point operations.

These routines follow the standard C/C++ calling sequence. The compiler automatically adds these routines when appropriate; they are not intended to be called directly by your programs.

The source code for these functions is in the source library rts.src. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of the math functions. Be sure, however, that you follow the calling conventions and register-saving rules outlined in this chapter. Table 8−6 summarizes the run-time-support functions used for arithmetic.

Table 8−6. Summary of Run-Time-Support Arithmetic Functions

	Type	Function	Description
	float	_cvtdf (double)	Convert double to float
	int	_fixdi (double)	Convert double to signed integer
	long	_fixdli (double)	Convert double to long
	long long	_fixdlli (double)	Convert double to long long
	uint	_fixdu (double)	Convert double to unsigned integer
	ulong	_fixdul (double)	Convert double to unsigned long
	ulong long	_fixdull (double)	Convert double to unsigned long long
	double	_cvtfd (float)	Convert float to double
	int	_fixfi (float)	Convert float to signed integer
	long	_fixfli (float)	Convert float to long
	long long	_fixflli (float)	Convert float to long long
	uint	_fixfu (float)	Convert float to unsigned integer
	ulong	_fixful (float)	Convert float to unsigned long
	ulong long	_fixfull (float)	Convert float to unsigned long long

Table 8−6. Summary of Run-Time-Support Arithmetic Functions

	Type	Function	Description
	double	_fltid (int)	Convert signed integer to double
	float	_fltif (int)	Convert signed integer to float
	double	_fltud (uint)	Convert unsigned integer to double
	float	_fltuf (uint)	Convert unsigned integer to float
	double	_fltlid (long)	Convert signed long to double
	float	_fltlif (long)	Convert signed long to float
	double	_fltuld (ulong)	Convert unsigned long to double
	float	_fltulf (ulong)	Convert unsigned long to float
	double	_fltllid (long long)	Convert signed long long to double
	float	_fltllif (long long)	Convert signed long long to float
	double	_fltulld (ulong long)	Convert unsigned long long to double
	float	_fltullf (ulong long)	Convert unsigned long long to float
	double	_absd (double)	Double absolute value
	float	_absf (float)	Float absolute value
	long	_labs (long)	Long absolute value
	long long	_llabs (long long)	Long long absolute value
	double	_negd (double)	Double negate value
	float	_negf (float)	Float negate value
	long long	_negll (long)	Long long negate value
	long long	_llshl (long long)	Long long shift left
	long long	_llshr (long long)	Long long shift right
	ulong long	_llshru (ulong long)	Unsigned long long shift right
	double	_addd (double, double)	Double addition
	double	_cmpd (double, double)	Double comparison
	double	_divd (double, double)	Double division

Table 8−6. Summary of Run-Time-Support Arithmetic Functions

	Type	Function	Description
	double	_mpyd (double, double)	Double multiplication
	double	_subd (double, double)	Double subtraction
	float	_addf (float, float)	Float addition
	float	_cmpf (float, float)	Float comparison
	float	_divf (float, float)	Float division
	float	_mpyf (float, float)	Float multiplication
	float	_subf (float, float)	Float subtraction
	int	_divi (int, int)	Signed integer division
	int	_remi (int, int)	Signed integer remainder
	uint	_divu (uint, uint)	Unsigned integer division
	uint	_remu (uint, uint)	Unsigned integer remainder
	long	_divli (long, long)	Signed long division
	long	_remli (long, long)	Signed long remainder
	ulong	_divul (ulong, ulong)	Unsigned long division
	ulong	_remul (ulong, ulong)	Unsigned long remainder
	long long	_divlli (long long, long long)	Signed long long division
	long long	_remlli (long long, long long)	Signed long long remainder
	ulong long	_mpyll(ulong long, ulong long)	Unsigned long long multiplication
	ulong long	_divull (ulong long, ulong long)	Unsigned long long division
	ulong long	_remull (ulong long, ulong long)	Unsigned long long remainder

8.8 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00. The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c.

To begin running the system, the c_int00 function can be branched to or called, but it is usually vectored to by reset hardware. You must link the c_int00 function with the other object modules. This occurs automatically when you use the −c or −cr linker option and include a standard run-time-support library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output module to the symbol c_int00. This does not, however, set the hardware to automatically vector to c_int00 at reset (see the *TMS320C6000 CPU and Instruction Set Reference Guide*).

The c_int00 function performs the following tasks to initialize the environment:

1) It defines a section called .stack for the system stack and sets up the initial stack pointers.

2) It initializes global variables by copying the data from the initialization tables in the .cinit section to the storage allocated for the variables in the .bss section. If you are initializing variables at load time (−cr option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see section 8.8.1, *Automatic Initialization of Variables*.

3) It calls the function main to run the C/C++ program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

See section 9.1 on page 9-2 for a list of the standard run-time-support libraries that are shipped with the C6000 code generation tools.

8.8.1 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and intializing the variables with the data is called autoinitialization.

The compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

Note: Initializing Variables

In ISO C, global and static variables that are not explicitly initialized are set to 0 before program execution. The C6000 C/C++ compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

The easiest method is to have the stand-alone simulator using the −b option clear the .bss section before the program starts running. Another method is to set a fill value of 0 in the linker control map for the .bss section.

You cannot use these methods with code that is burned into ROM.

Global variables are either autoinitialized at run-time or at load time. For information, see sections 8.8.4, *Autoinitialization of Variables at Run-Time*, on page 8-56, and 8.8.5, *Initialization of Variables at Load Time*, on page 8-57. Also, see section 7.9, *Initializing Static and Global Variables*,on page 7-34.

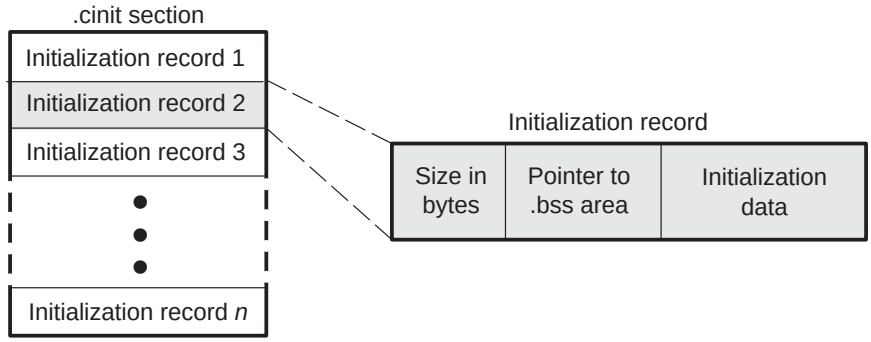
8.8.2 Global Constructors

All global C++ variables that have constructors must have their constructor called before main (). The compiler builds a table of global constructor addresses that must be called, in order, before main () in a section called .pinit. The linker combines the .pinit section form each input file to form a single table in the .pinit section. The boot routine uses this table to execute the constructors.

8.8.3 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has a record in the .cinit section. Figure 8−8 shows the format of the .cinit section and the initialization records.

Figure 8−8. Format of Initialization Records in the .cinit Section



The fields of an initialization record contain the following information:

- The first field of an initialization record is the size (in bytes) of the initialization data.
- The second field contains the starting address of the area within the .bss section where the initialization data must be copied.
- The third field contains the data that is copied into the .bss section to initialize the variable.

Each variable that must be autoinitialized has an initialization record in the .cinit section.

Example 8−9 (a) shows initialized global variables defined in C. Example 8−9 (b) shows the corresponding initialization table. The section .cinit:c is a subsection in the .cinit section that contains all scalar data. The subsection is handled as one record during initialization, which minimizes the overall size of the .cinit section.

Example 8−9. Initialization Table

(a) Initialized variables defined in C

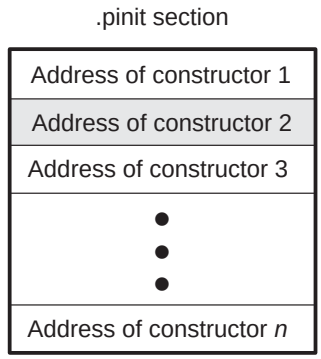
```
int x;
short i = 23;
int *p = &x;
int a[5] = {1,2,3,4,5};
```

(b) Initialized information for variables defined in (a)

```
        .global _x
        .bss    _x,4,4

        .sect   ".cinit:c"
        .align  8
        .field          (CIR - $) - 8, 32
        .field          _i+0,32
        .field         23,16                    ; _i @ 0

        .sect   ".text"
        .global _i
_i:     .usect  ".bss:c",2,2

        .sect   ".cinit:c"
        .align  4
        .field         _x,32                    ; _p @ 0

        .sect   ".text"
        .global _p
_p:     .usect  ".bss:c",4,4

        .sect   ".cinit"
        .align  8
        .field          IR_1,32
        .field          _a+0,32
        .field         1,32                     ; _a[0] @ 0
        .field         2,32                     ; _a[1] @ 32
        .field         3,32                     ; _a[2] @ 64
        .field         4,32                     ; _a[3] @ 96
        .field         5,32                     ; _a[4] @ 128
IR_1:   .set    20

        .sect   ".text"
        .global _a
        .bss    _a,20,4
;***************************************************************
;* MARK THE END OF THE SCALAR INIT RECORD IN CINIT:C            *
;***************************************************************

CIR:    .sect   ".cinit:c"
```

The .cinit section must contain only initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other purpose.

Figure 8−9. Format of Initialization Records in the .pinit Section



When you use the −c or −cr option, the linker combines the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0 and marks the end of the initialization tables.

Likewise, the −c or −cr linker option causes the linker to combine all of the .pinit sections from all C/C++ modules and append a null word to the end of the composite .pinit section. The boot routine knows the end of the global constructor table when it encounters a null constructor address.

The const-qualified variables are initialized differently; see section 7.4.1, *The const Keyword*, on page 7-7.

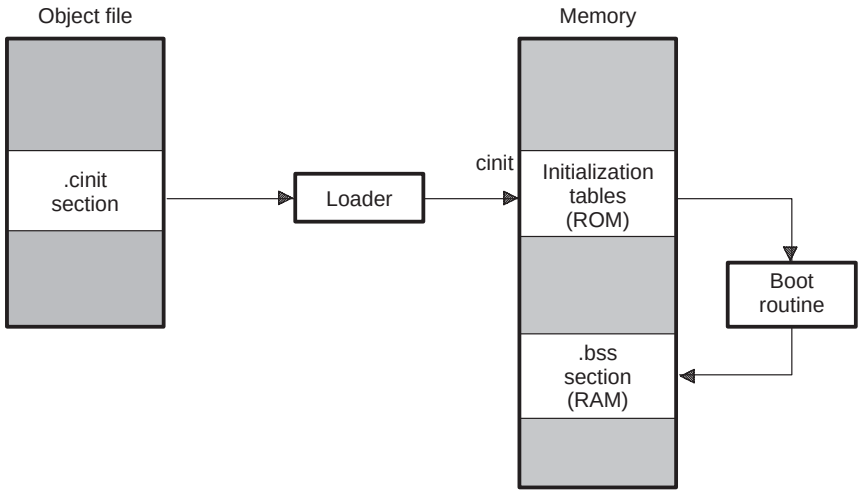
8.8.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the −c option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 8−10 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 8−10. Autoinitialization at Run Time



8.8.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the −cr option.

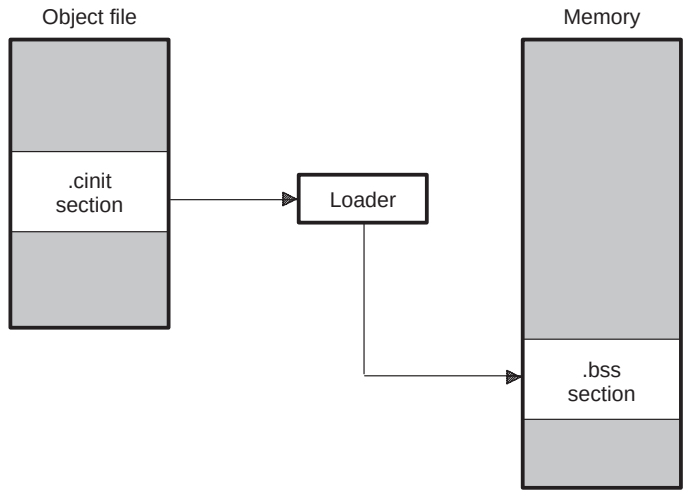
When you use the −cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to −1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- ❑ Detect the presence of the .cinit section in the object file
- ❑ Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- ❑ Understand the format of the initialization tables

Figure 8−11 illustrates the initialization of variables at load time.

Figure 8−11. Initialization at Load Time



Chapter 9

Run-Time-Support Functions

Some of the tasks that a C/C++ program performs (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C/C++ language itself. However, the ISO C standard defines a set of run-time-support functions that perform these tasks. The TMS320C6000 C/C++ compiler implements the complete ISO standard library except for those facilities that handle exception conditions and locale issues (properties that depend on local language, nationality, or culture). Using the ISO standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ISO-specified functions, the TMS320C6000 run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests.

A library build utility is provided with the code generation tools that lets you create customized run-time-support libraries. The use of this utility is covered in Chapter 10, *Library-Build Utility*.

Topic **Page**

9.1 Libraries

The following libraries are included with the TMS320C6000 C/C++ compiler:

- rts6200.lib, rts6400.lib, and rts6700.lib—run-time-support object libraries for use with little-endian C/C++ code
- rts6200e.lib, rts6400e.lib, and rts6700e.lib—run-time-support object libraries for use with big-endian C/C++ code
- rts.src—run-time-support source library. The run-time-support object libraries are built from the C, C++, and assembly source contained in the rts.src library.

The run-time-support libraries do not contain functions involving signals and locale issues. They do contain the following:

- ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Intrinsic arithmetic routines
- System startup routine, _c_int00
- Functions and macros that allow C/C++ to access specific instructions

You can control how the run-time-support functions are called in terms of near or far calls with the −mr option. For more information, see section 7.4.4.3, *Controlling How Run-Time-Support Functions Are Called (−mr Option)*, on page 7-12.

9.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved.

You should specify libraries *last* on the linker command line because the linker searches a library for unresolved references when it encounters the library on the command line. You can also use the −x linker option to force repeated searches of each library until the linker can resolve no more references.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *TMS320C6000 Assembly Language Tools User's Guide*.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

9.1.2 Modifying a Library Function

You can inspect or modify library functions by using the archiver to extract the appropriate source file or files from the source libraries. For example, the following command extracts two source files:

```
ar6x x rts.src atoi.c strcpy.c
```

To modify a function, extract the source as in the previous example. Make the required changes to the code, recompile, and reinstall the new object file or files into the library. For example:

```
cl6x -options atoi.c strcpy.c           ;recompile
ar6x r rts6200.lib atoi.obj strcpy.obj  ;rebuild library
```

You can also build a new library this way, rather than rebuilding into rts6200.lib. For more information about the archiver, see the *TMS320C6000 Assembly Language Tools User's Guide*.

9.1.3 Building a Library With Different Options

You can create a new library from rts.src by using the library-build utility mk6x. For example, use this command to build an optimized run-time-support library:

```
mk6x --u -O2 -x rts.src -l rts.lib
```

The `--u` option tells the mk6x utility to use the header files in the current directory, rather than extracting them from the source archive. The use of the −O2 option does not affect compatibility with code compiled without this option. For more information on the library build utility, see Chapter 10, *Library-Build Utility*.

9.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O. The capability to perform I/O on the host gives you more options when debugging and testing code.

To use the I/O functions, include the header file stdio.h, or cstdio for C++ code, for each module that references a C I/O function.

For example, given the following C program in a file named main.c:

```
#include <stdio.h>

main()
{
   FILE *fid;

   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);

   printf("Hello again, world\n");
}
```

Issuing the following compiler command compiles, links, and creates the file main.out from the 'C6200 run-time-support little-endian library:

```
cl6x main.c -z -heap 400 -l rts6200.lib -o main.out
```

Executing main.out results in

```
Hello, world
```

being output to a file and

```
Hello again, world
```

being output to your host's stdout window.

With properly written device drivers, the library also offers facilities to perform I/O on a user-specified device.

Note: C I/O Buffer Failure

If there is not enough space on the heap for a C I/O buffer, buffered operations on the file will fail. If a call to printf() mysteriously fails, this may be the reason. Check the size of the heap. To set the heap size, use the −heap option when linking (see page 5-6).

9.2.1 Overview of Low-Level I/O Implementation

The code that implements I/O is logically divided into layers: high level, low level, and device level.

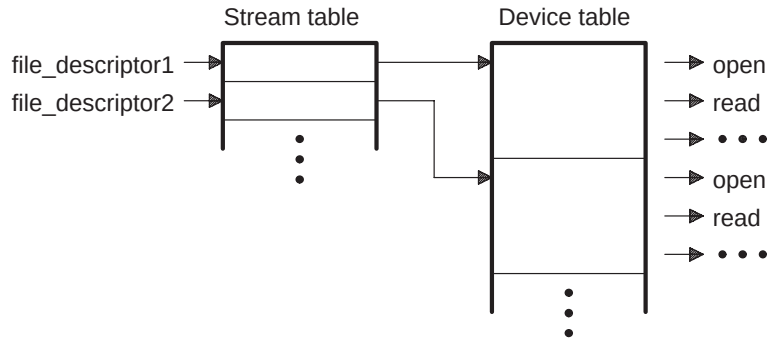
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level routines.

The low-level routines are comprised of basic I/O functions: open, read, write, close, lseek, rename, and unlink. These low-level routines provide the interface between the high-level functions and the device-level drivers that actually perform the I/O command on the specified device.

The low-level functions also define and maintain a stream table that associates a file descriptor with a device. The stream table interacts with the device table to ensure that an I/O command performed on a stream executes the correct device-level routine.

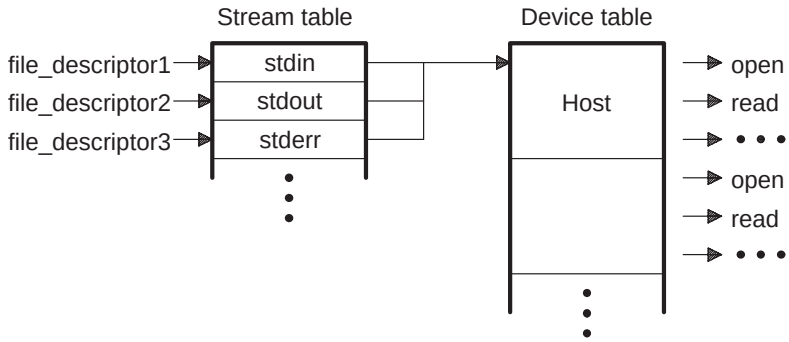
The data structures interact as shown in Figure 9−1.

Figure 9−1. Interaction of Data Structures in I/O Functions



The first three streams in the stream table are predefined to be stdin, stdout, and stderr and they point to the host device and associated device drivers.

Figure 9−2. The First Three Streams in the Stream Table



At the next level are the user-definable device-level drivers. They map directly to the low-level I/O functions. The run-time-support library includes the device drivers necessary to perform I/O on the host on which the debugger is running.

The specifications for writing device-level routines to interface with the low-level routines follow. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

add_device — *Add Device to Device Table*

Syntax for C

```
#include <file.h>

int add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               fpos_t (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

Defined in

lowlev.c in rts.src

Description

The add_device function adds a device record to the device table allowing that device to be used for input/output from C. The first entry in the device table is predefined to be the host device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen() with a string of the format *devicename*:*filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in stdio.h.
- The dopen, dclose, dread, dwrite, dlseek, dunlink, drename specifiers are function pointers to the device drivers that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in section 9.2.1, *Overview of Low-Level I/O Implementation*, on page 9-5. The device drivers for the host that the TMS320C6000 debugger is run on are included in the C I/O library.

Return Value

The function returns one of the following values:

0 if successful

−1 if fails

Example This example does the following:

- ❏ Adds the device *mydevice* to the device table
- ❏ Opens a file named *test* on that device and associates it with the file **fid*
- ❏ Writes the string *Hello, world* into the file
- ❏ Closes the file

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{

   FILE *fid;
   add_device(”mydevice”, _MSA, my_open, my_close, my_read, my_write, my_lseek,
                                my_unlink, my_rename);

   fid = fopen(”mydevice:test”,”w”);

   fprintf(fid,”Hello, world\n”);

   fclose(fid);
}
```

close — *Close File or Device For I/O*

Syntax for C

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

int close(int file_descriptor);
```

Syntax for C++

```
#include <cstdio>
#include <file.h>

int std::close(int file_descriptor);
```

Description

The close function closes the device or file associated with *file_descriptor*.

The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened device or file.

Return Value

The return value is one of the following:

0	if successful
−1	if fails

lseek — *Set File Position Indicator*

Syntax for C

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

long lseek(int file_descriptor, long offset, int origin);
```

Syntax for C++

```
#include <cstdio>
#include <file.h>

long std::lseek(int file_descriptor, long offset, int origin);
```

Description

The lseek function sets the file position indicator for the given file to *origin* + *offset*. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the stream number assigned by the low-level routines that the device-level driver must associate with the opened file or device.
- The *offset* indicates the relative offset from the *origin* in characters.
- The *origin* is used to indicate which of the base locations the *offset* is measured from. The *origin* must be a value returned by one of the following macros:

  SEEK_SET (0x0000) Beginning of file
  SEEK_CUR (0x0001) Current value of the file position indicator
  SEEK_END (0x0002) End of file

Return Value

The return function is one of the following:

#	new value of the file-position indicator if successful
EOF	if fails

open

Open File or Device For I/O

Syntax for C

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

int open(const char *path, unsigned flags, int file_descriptor);
```

Syntax for C++

```
#include <cstdio>
#include <file.h>

int std::open(const char *path, unsigned flags, int file_descriptor);
```

Description

The open function opens the device or file specified by *path* and prepares it for I/O.

- The *path* is the filename of the file to be opened, including path information.
- The *flags* are attributes that specify how the device or file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY  (0x0000)  /* open for reading */
O_WRONLY  (0x0001)  /* open for writing */
O_RDWR    (0x0002)  /* open for read & write */
O_APPEND  (0x0008)  /* append on each write */
O_CREAT   (0x0200)  /* open with file create */
O_TRUNC   (0x0400)  /* open with truncation */
O_BINARY  (0x8000)  /* open in binary mode */
```

  These parameters can be ignored in some cases, depending on how data is interpreted by the device. However, the high-level I/O calls look at how the file was opened in an fopen statement and prevent certain actions, depending on the open attributes.

- The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened file or device.

  The next available file_descriptor (in order from 3 to 20) is assigned to each new device opened. You can use the finddevice() function to return the device structure and use this pointer to search the _stream array for the same pointer. The file_descriptor number is the other member of the _stream array.

Return Value

The function returns one of the following values:

≠−1 if successful
−1 if fails

read — *Read Characters From Buffer*

Syntax for C

#include <stdio.h>
#include <file.h>

int read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

Syntax for C++

#include <cstdio>
#include <file.h>

int std::read(int *file_descriptor*, **char** **buffer*, **unsigned** *count*);

Description

The read function reads the number of characters specified by *count* to the *buffer* from the device or file associated with *file_descriptor*.

- The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened file or device.
- The *buffer* is the location of the buffer where the read characters are placed.
- The *count* is the number of characters to read from the device or file.

Return Value

The function returns one of the following values:

0	if EOF was encountered before the read was complete
#	number of characters read in every other instance
−1	if fails

rename — *Rename File*

Syntax for C

#include <stdio.h>
#include <file.h>

int rename(const char * *old_name*, **const char *** *new_name*);

Syntax for C++

#include <cstdio>
#include <file.h>

int std::rename(const char * *old_name*, **const char *** *new_name*);

Description

The rename function changes the name of a file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Return Value

The function returns one of the following values:

0	if successful
Non-0	if not successful

unlink — *Delete File*

Syntax for C

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

int unlink(const char *path);
```

Syntax for C++

```
#include <cstdio>
#include <file.h>

int std::unlink(const char *path);
```

Description

The unlink function deletes the file specified by *path*.

The *path* is the filename of the file to be opened, including path information.

Return Value

The function returns one of the following values:

0	if successful
−1	if fails

write — *Write Characters to Buffer*

Syntax for C

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

int write(int file_descriptor, const char *buffer, unsigned count);
```

Syntax for C++

```
#include <cstdio>
#include <file.h>

int write(int file_descriptor, const char *buffer, unsigned count);
```

Description

The write function writes the number of characters specified by *count* from the *buffer* to the device or file associated with *file_descriptor*.

- The *file_descriptor* is the stream number assigned by the low-level routines. It is associated with the opened file or device.
- The *buffer* is the location of the buffer where the write characters are placed.
- The *count* is the number of characters to write to the device or file.

Return Value

The function returns one of the following values:

#	number of characters written if successful
−1	if fails

9.2.2 Adding a Device for C I/O

The low-level functions provide facilities that allow you to add and use a device for I/O at run time. The procedure for using these facilities is:

1) Define the device-level functions as described in section 9.2.1, *Overview of Low-Level I/O Implementation,* on page 9-5.

> **Note: Use Unique Function Names**
>
> The function names open, close, read, and so on, are used by the low-level routines. Use other names for the device-level functions that you write.

2) Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio. The structure representing a device is also defined in stdio.h/cstdio and is composed of the following fields:

name	String for device name
flags	Flags that specify whether the device supports multiple streams or not
function pointers	Pointers to the device-level functions: ❑ CLOSE ❑ LSEEK ❑ OPEN ❑ READ ❑ RENAME ❑ WRITE ❑ UNLINK

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function on page 9-7.

3) Once the device is added, call fopen() to open a stream and associate it with that device. Use *devicename*:*filename* as the first argument to fopen().

The following program illustrates adding and using a device for C I/O:

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int  my_open(const char *path, unsigned flags, int fno);
extern int  my_close(int fno);
extern int  my_read(int fno, char *buffer, unsigned count);
extern int  my_write(int fno, const char *buffer, unsigned count);
extern long my_lseek(int fno, long offset, int origin);
extern int  my_unlink(const char *path);
extern int  my_rename(const char *old_name, char *new_name);

main()
{
   FILE *fid;

   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,

                                my_unlink, my_rename);

   fid = fopen("mydevice:test","w");

   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

9.3 Header Files

Each run-time-support function is declared in a *header file*. Each header file declares the following:

- ❏ A set of related functions (or macros)
- ❏ Any types that you need to use the functions
- ❏ Any macros that you need to use the functions

These are the header files that declare the ISO C run-time-support functions:

assert.h	inttypes.h	setjmp.h	stdio.h
ctype.h	iso646.h	stdarg.h	stdlib.h
errno.h	limits.h	stddef.h	string.h
float.h	math.h	stdint.h	time.h

In addition to the ISO C header files, the following C++ header files are included:

cassert	climits	cstdio	new
cctype	cmath	cstdlib	stdexcept
cerrno	csetjmp	cstring	typeinfo
cfloat	cstdarg	ctime	
ciso646	cstddef	exception	

Furthermore, the following header files are included for the additional functions we provide:

c6x.h	cpy_tbl.h	file.h	gsm.h	linkage.h

To use a run-time-support function, you must first use the #include preprocessor directive to include the header file that declares the function. For example, in C the isdigit function is declared by the ctype.h header. Before you can use the isdigit function, you must first include ctype.h:

```
#include <ctype.h>
   .
   .
   .
   val = isdigit(num);
```

You can include headers in any order. You must, however, include a header before you reference any of the functions or objects that it declares.

Sections 9.3.1, *Diagnostic Messages (assert.h/cassert)*, on page 9-17 through 9.3.21, *Run-Time Type Information (typeinfo)*, on page 9-28 describe the header files that are included with the C6000 C/C++ compiler. Section 9.4, *Summary of Run-Time-Support Functions and Macros*, on page 9-29 lists the functions that these headers declare.

9.3.1 Diagnostic Messages (assert.h/cassert)

The assert.h/cassert header defines the assert macro, which inserts diagnostic failure messages into programs at run time. The assert macro tests a run-time expression.

- If the expression is true (nonzero), the program continues running.
- If the expression is false, the macro outputs a message that contains the expression, the source file name, and the line number of the statement that contains the expression; then, the program terminates (using the abort function).

The assert.h/cassert header refers to another macro named NDEBUG (assert.h/cassert does not define NDEBUG). If you have defined NDEBUG as a macro name when you include assert.h/cassert, assert is turned off and does nothing. If NDEBUG is *not* defined, assert is enabled.

The assert.h/cassert header refers to another macro named NASSERT (assert.h/cassert does not define NASSERT). If you have defined NASSERT as a macro name when you include assert.h/cassert, assert acts like _nassert. The _nassert intrinsic generates no code and tells the compiler that the expression declared with assert is true. This gives a hint to the compiler as to what optimizations might be valid. If NASSERT is *not* defined, assert is enabled normally.

The _nassert intrinsic can also be used to guarantee that a pointer has a certain alignment. For more information, see section 8.5.4, *Using _nassert to Enable SIMD and Expand Compiler Knowledge of Loops*, on page 8-37.

The assert function is listed in Table 9−3 (a) on page 9-30.

9.3.2 Character-Typing and Conversion (ctype.h/cctype)

The ctype.h/cctype header declares functions that test type of characters and converts them.

The character-typing functions test a character to determine whether it is a letter, a printing character, a hexadecimal digit, etc. These functions return a value of *true* (a nonzero value) or *false* (0). Character-typing functions have names in the form is*xxx* (for example, *isdigit*).

The character-conversion functions convert characters to lowercase, uppercase, or ASCII, and return the converted character. Character-conversion functions have names in the form to*xxx* (for example, *toupper*).

The ctype.h/cctype header also contains macro definitions that perform these same operations. The macros run faster than the corresponding functions. Use the function version if an argument is passed that has side effects. The typing macros expand to a lookup operation in an array of flags (this array is defined in ctype.c). The macros have the same name as the corresponding functions, but each macro is prefixed with an underscore (for example, *_isdigit*).

The character typing and conversion functions are listed in Table 9−3 (b) page 9-30.

9.3.3 Error Reporting (errno.h/cerrno)

The errno.h/cerrno header declares the errno variable. The errno variable indicates errors in library functions. Errors can occur in a math function if invalid parameter values are passed to the function or if the function returns a result that is outside the defined range for the type of the result. When this happens, a variable named errno is set to the value of one of the following macros:

- EDOM for domain errors (invalid parameter)
- ERANGE for range errors (invalid result)
- ENOENT for path errors (path does not exist)
- EFPOS for seek errors (file position error)

C code that calls a math function can read the value of errno to check for error conditions. The errno variable is declared in errno.h/cerrno and defined in errno.c.

9.3.4 Low-Level Input/Output Functions (file.h)

The file.h header declares the low-level I/O functions used to implement input and output operations.

How to implement I/O for the C6000 is described in section 9.2, *The C/IO Functions*, on page 9-4.

9.3.5 Fast Macros/Static Inline Functions (gsm.h)

The gsm.h header file contains fast macros, and static inline function definitions to define the basic ETSI math operations of a GSM vocoder.

9.3.6 Limits (float.h/cfloat and limits.h/climits)

The float.h/cfloat and limits.h/climits headers define macros that expand to useful limits and parameters of the TMS320C6000's numeric representations. Table 9−1 and Table 9−2 list these macros and their limits.

Table 9−1. Macros That Supply Integer Type Range Limits (limits.h/climits)

Macro	Value	Description
CHAR_BIT	8	Number of bits in type char
SCHAR_MIN	−128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	−32 768	Minimum value for a short int
SHRT_MAX	32 767	Maximum value for a short int
USHRT_MAX	65 535	Maximum value for an unsigned short int
INT_MIN	(−INT_MAX − 1)	Minimum value for an int
INT_MAX	2 147 483 647	Maximum value for an int
UINT_MAX	4 294 967 295	Maximum value for an unsigned int
LONG_MIN	(−LONG_MAX − 1)	Minimum value for a long int
LONG_MAX	549 755 813 887	Maximum value for a long int
ULONG_MAX	1 099 511 627 775	Maximum value for an unsigned long int
LLONG_MIN	(−LLONG_MAX − 1)	Minimum value for a long long int
LLONG_MAX	9 223 372 036 854 775 807	Maximum value for a long long int
ULLONG_MAX	18 446 744 073 709 551 615	Maximum value for an unsigned long long int

Note: Negative values in this table are defined as expressions in the actual header file so that their type is correct.

Table 9−2. Macros That Supply Floating-Point Range Limits (float.h/cfloat)

Macro	Value	Description
FLT_RADIX	2	Base or radix of exponent representation
FLT_ROUNDS	1	Rounding mode for floating-point addition
FLT_DIG DBL_DIG LDBL_DIG	6 15 15	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 53 53	Number of base FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	−125 −1021 −1021	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 1024 1024	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e−07 2.22044605e−16 2.22044605e−16	Minimum positive float, double, or long double number *x* such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.17549435e−38 2.22507386e−308 2.22507386e−308	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 1.79769313e+308 1.79769313e+308	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	−37 −307 −307	Minimum negative integers such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 308 308	Maximum positive integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles

Legend: FLT_ applies to type float.
DBL_ applies to type double.
LDBL_ applies to type long double.

Note: The precision of some of the values in this table has been reduced for readability. Refer to the float.h/cfloat header file supplied with the compiler for the full precision carried by the processor.

9.3.7 Format Conversion of Integer Types (inttypes.h)

The stdint.h header declares sets of integer types of specified widths and defines corresponding sets of macros. The inttypes.h header contains stdint.h and also provides a set of integer types with definitions that are consistent across machines and independent of operating systems and other implementation idiosyncrasies. The inttypes.h header declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers.

Through typedef, inttypes.h defines integer types of various sizes. You are free to typedef integer types as standard C integer types or as the types provided in inttypes.h. Consistent use of the inttypes.h header greatly increases the portability of your program across platforms.

The header declares three types:

- The *imaxdiv_t* type, a structure type of the type of the value returned by the imaxdiv function
- The *intmax_t* type, an integer type large enough to represent any value of any signed integer type
- The *uintmax_t* type, an integer type large enough to represent any value of any unsigned integer type

The header declares several macros and functions:

- For each size type available on the architecture and provided in stdint.h, there are several fprintf and fscanf macros. For example, three fprintf macros for signed integers are PRId32, PRIdLEAST32, and PRIdFAST32. An example use of these macros is:

```
printf("The largest integer value is %020"
      PRIxMAX "\n", i);
```

- The *imaxabs* function that computes the absolute value of an integer of type intmax_t.
- The *strtoimax* and *strtoumax* functions, which are equivalent to the strtol, strtoll, strtoul, and strtoull functions. The initial portion of the string is converted to intmax_t and uintmax_t, respectively.

For detailed information on the inttypes.h header, see the *ISO/IEC 9899:1999, International Standard − Programming Languages − C (The C Standard*.

9.3.8 Alternative Spellings (iso646.h/ciso646)

The iso646.h/ciso646 header defines the following eleven macros that expand to the corresponding tokens:

Macro	Token	Macro	Token
and	&&	not_eq	!=
and_eq	&=	or	||
bitand	&	or_eq	|=
bitor	|	xor	^
compl	~	xor_eq	^=
not	!		

9.3.9 Function Calls as near or far (linkage.h)

The linkage.h header declares macros that determine how code and data in the run-time support library is accessed.. Depending on the value of the _FAR_RTS macro, the _CODE_ACCESS macro is set to force calls to run-time-support functions to be either user default, near or far. The _FAR_RTS macro is set according to the use of the −mr compiler option.

The _DATA_ACCESS macro is set to always be far. The _IDECL macro determines how inline functions are declared.

All header files that define functions or data declare #include <linkage.h>. Functions are modified with _CODE_ACCESS, for example:

```
extern _CODE_ACCESS void    exit(int _status);
```

Data is modified with _DATA_ACCESS, for example:

```
extern _DATA_ACCESS unsigned char _ctypes_[];
```

9.3.10 Floating-Point Math (math.h/cmath)

The math.h/cmath header declares several trigonometric, exponential, and hyperbolic math functions. These functions are listed in Table 9−3 (c) on page 9-31. The math functions expect arguments either of type double or of type float and return values either of type double or of type float, respectively. Except where indicated, all trigonometric functions use angles expressed in radians.

The math.h/cmath header also defines one macro named HUGE_VAL. The math functions use this macro to represent out-of-range values. When a function produces a floating-point return value that is too large to represent, it returns HUGE_VAL instead.

The math.h/cmath header includes enhanced math functions that are available when you define the _TI_ENHANCED_MATH_H symbol in your source file. When you define the _TI_ENHANCED_MATH_H symbol, the HUGE_VALF symbol is made visible. HUGE_VALF is the float counterpart to HUGE_VAL.

For all math.h/cmath functions, domain and range errors are handled by setting errno to EDOM or ERANGE, as appropriate. The function input/outputs are rounded to the nearest legal value.

9.3.11 Nonlocal Jumps (**setjmp.h/csetjmp**)

The setjmp.h/csetjmp header defines a type and a macro and declares a function for bypassing the normal function call and return discipline. These include:

- The *jmp_buf* type is an array type suitable for holding the information needed to restore a calling environment.
- The *setjmp* macro saves its calling environment in its jmp_buf argument for later use by the longjmp function.
- The *longjmp* function uses its jmp_buf argument to restore the program environment.

The nonlocal jmp macro and function are listed in Table 9−3 (d) on page 9-34.

9.3.12 Variable Arguments (**stdarg.h/cstdarg**)

Some functions can have a variable number of arguments whose types can differ. Such functions are called *variable-argument functions*. The stdarg.h/cstdarg header declares macros and a type that help you to use variable-argument functions.

- The macros are *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments can vary each time a function is called.
- The type *va_list* is a pointer type that can hold information for va_start, va_end, and va_arg.

A variable-argument function can use the macros declared by stdarg.h/cstdarg to step through its argument list at run time when the function knows the number and types of arguments actually passed to it. You must ensure that a call to a variable-argument function has visibility to a prototype for the function in order for the arguments to be handled correctly. The variable argument functions are listed in Table 9−3 (e) page 9-34.

9.3.13 Standard Definitions (sttdef.h/cstddef)

The stddef.h/cstddef header defines these types and macros:

- The *ptrdiff_t* type is a signed integer type that is the data type resulting from the subtraction of two pointers.
- The *size_t* type is an unsigned integer type that is the data type of the *sizeof* operator.
- The *NULL* macro expands to a null pointer constant (0).
- The *offsetof(type, identifier)* macro expands to an integer that has type *size_t*. The result is the value of an offset in bytes to a structure member (identifier) from the beginning of its structure (type).

These types and macros are used by several of the run-time-support functions.

9.3.14 Integer Types (stdint.h)

The stdint.h header declares sets of integer types of specified widths and defines corresponding sets of macros. It also defines macros that specify limits of integer types that correspond to types defined in other standard headers. Types are defined in these categories:

- Integer types with certain exact widths of the signed form int*N*_t and of the unsigned form uint*N*_t
- Integer types with at least certain specified widths of the signed form int_least*N*_t and of the unsigned form uint_least*N*_t
- Fastest integer types with at least certain specified widths of the signed form int_fast*N*_t and of the unsigned form uint_fast*N*_t
- Signed, *intprt_t*, and unsigned, *uintptr_t*, integer types large enough to hold a pointer value
- Signed, *intmax_t*, and unsigned, *uintmax_t*, integer types large enough to represent any value of any integer type

For each signed type provided by stdint.h there is a macro that specifies the minimum or maximum limit. Each macro name corresponds to a similar type name described above.

The INT*N*_C(*value*) macro expands to a signed integer constant with the specified value and type int_least*N*_t. The unsigned UINT*N*_C(*value*) macro expands to an unsigned integer constant with the specified value and type uint_least*N*_t.

This example shows a macro defined in stdint.h that uses the smallest integer that can hold at least 16 bits:

```
       typedef    uint_least_16 id_number;
extern id_number  lookup_user(char *uname);
```

For detailed information on the stdint.h header, see the *ISO/IEC 9899:1999, International Standard − Programming Languages − C (The C Standard*.

9.3.15 Input/Output Functions (stdio.h/cstdio)

The stdio.h/cstdio header defines seven macros, two types, a structure, and a number of functions. The types and structure are:

- The *size_t* type is an unsigned integer type that is the data type of the *sizeof* operator. Originally defined in sttdef.h/cstddef.
- The *fpos_tI* type is a signed integer type that can uniquely specify every position within a file.
- The *FILE* type is a structure type that records all the information necessary to control a stream.

The macros are:

- The *NULL* macro expands to a null pointer constant(0). Originally defined in sttdef.h/cstddef. It is not redefined if it was already defined.
- The *BUFSIZ* macro expands to the size of the buffer that setbuf() uses.
- The *EOF*macro is the end-of-file marker.
- The *FOPEN_MAX* macro expands to the largest number of files that can be open at one time.
- The *FILENAME_MAX* macro expands to the length of the longest file name in characters.
- The *L_tmpnam* macro expands to the longest filename string that tmpnam() can generate.
- The *SEEK_CUR*, *SEEK_SET*, and *SEEK_END* macros expand to indicate the position (current, start-of-file, or end-of-file, respectively) in a file.
- The *TMP_MAX* macro expands to the maximum number of unique filenames that tmpnam() can generate.
- The *stderr*, *stdin*, *stdout* macros are pointers to the standard error, input, and output files, respectively.

The input/output functions are listed in Table 9−3 (f) on page 9-34.

9.3.16 General Utilities (stdlib.h/cstdlib)

The stdlib.h/cstdlib header defines a macro and types and declares functions. The macro is named RAND_MAX, and it returns the largest value returned by the rand() function. The types are:

- The *div_t* type is a structure type that is the type of the value returned by the div function.
- The *ldiv_t* type is a structure type that is the type of the value returned by the ldiv function

The functions are:

- String conversion functions convert strings to numeric representations
- Searching and sorting functions search and sort arrays
- Sequence-generation functions generate a pseudo-random sequence and choose a starting point for a sequence
- Program-exit functions terminate your program normally or abnormally
- Integer-arithmetic that is not provided as a standard part of the C language

The general utility functions are listed in Table 9−3 (g) on page 9-37.

9.3.17 String Functions (string.h/cstring)

The string.h/cstring header declares standard functions that perform the following tasks with character arrays (strings):

- Move or copy entire strings or portions of strings
- Concatenate strings
- Compare strings
- Search strings for characters or other strings
- Find the length of a string

In C, all character strings are terminated with a 0 (null) character. The string functions named str*xxx* all operate according to this convention. Additional functions that are also declared in string.h/cstring perform corresponding operations on arbitrary sequences of bytes (data objects), where a 0 value does not terminate the object. These functions are named mem*xxx*.

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result. The string functions are listed in Table 9−3 (h) on page 9-38.

9.3.18 Time Functions (time.h/ctime)

The time.h/ctime header defines one macro and several types, and declares functions that manipulate dates and times. Times are represented in the following ways:

- As an arithmetic value of type *time_t*. When expressed in this way, a time is represented as a number of seconds since 12:00 AM January 1, 1900. The time_t type is a synonym for the type unsigned long.

- As a structure of type *struct_tm*. This structure contains members for expressing time as a combination of years, months, days, hours, minutes, and seconds. A time represented like this is called broken-down time. The structure has the following members.

```
int   tm_sec;     /* seconds after the minute (0–59) */
int   tm_min;     /* minutes after the hour (0–59)   */
int   tm_hour;    /* hours after midnight (0–23)     */
int   tm_mday;    /* day of the month (1–31)         */
int   tm_mon;     /* months since January (0–11)     */
int   tm_year;    /* years since 1900 (0 and up)     */
int   tm_wday;    /* days since Saturday (0–6)       */
int   tm_yday;    /* days since January 1 (0–365)    */
int   tm_isdst;   /* daylight savings time flag      */
```

A time, whether represented as a time_t or a struct tm, can be expressed from different points of reference:

- Calendar time represents the current Gregorian date and time.
- Local time is the calendar time expressed for a specific time zone.

The time functions and macros are listed in Table 9−3 (i) on page 9-40.

You can adjust local time for local or seasonal variations. Obviously, local time depends on the time zone. The time.h/ctime header defines a structure type called tmzone and a variable of this type called _tz. You can change the time zone by modifying this structure, either at run time or by editing tmzone.c and changing the initialization. The default time zone is CST (Central Standard Time), U.S.A.

The basis for all the time.h/ctime functions are the system functions of clock and time. Time provides the current time (in time_t format), and clock provides the system time (in arbitrary units). You can divide the value returned by clock by the macro CLOCKS_PER_SEC to convert it to seconds. Since these functions and the CLOCKS_PER_SEC macro are system specific, only stubs are provided in the library. To use the other time functions, you must supply custom versions of these functions.

Note: Writing Your Own Clock Function

The clock function works with the stand-alone simulator (load6x). Used in the load6x environment, clock() returns a cycle accurate count. The clock function returns −1 when used with the HLL debugger.

A host-specific clock function can be written. You must also define the CLOCKS_PER_SEC macro according to the units of your clock so that the value returned by clock()—number of clock ticks—can be divided by CLOCKS_PER_SEC to produce a value in seconds.

9.3.19 Exception Handling (exception and stdexcept)

Exception handling is not supported. The exception and stdexcept include files, which are for C++ only, are empty.

9.3.20 Dynamic Memory Management (new)

The new header, which is for C++ only, defines functions for new, new[ ], delete, delete[ ], and their placement versions.

The type new_handler and the function set_new_handler( ) are also provided to support error recovery during memory allocation.

9.3.21 Run-Time Type Information (typeinfo)

The typeinfo header, which is for C++ only, defines the type_info structure, which is used to represent C++ type information at run time.

9.4 Summary of Run-Time-Support Functions and Macros

Table 9−3 summarizes the run-time-support header files (in alphabetical order) provided with the TMS320C6000 ANSI/ISO C/C++ compiler. Most of the functions described are per the ISO standard and behave exactly as described in the standard.

The functions and macros listed in Table 9−3 are described in detail in section 9.5, *Description of Run-Time-Support Functions and Macros* on page 9-41. For a complete description of a function or macro, see the indicated page.

A superscripted number is used in the following descriptions to show exponents. For example, x^y is the equivalent of x to the power y.

Table 9−3. Summary of Run-Time-Support Functions and Macros

(a) Error message macro (assert.h/cassert)

Macro	Description	Page
void **assert**(int expr);	Inserts diagnostic messages into programs	9-46

(b) Character typing and conversion functions (ctype.h/cctype)

Function	Description	Page
int **isalnum**(int c);	Tests c to see if it is an alphanumeric-ASCII character	9-69
int **isalpha**(int c);	Tests c to see if it is an alphabetic-ASCII character	9-69
int **isascii**(int c);	Tests c to see if it is an ASCII character	9-69
int **iscntrl**(int c);	Tests c to see if it is a control character	9-69
int **isdigit**(int c);	Tests c to see if it is a numeric character	9-69
int **isgraph**(int c);	Tests c to see if it is any printing character except a space	9-69
int **islower**(int c);	Tests c to see if it is a lowercase alphabetic ASCII charcter	9-69
int **isprint**(int c);	Tests c to see if it is a printable ASCII character (including a space)	9-69
int **ispunct**(int c);	Tests c to see if it is an ASCII punctuation character	9-69
int **isspace**(int c);	Tests c to see if it is an ASCII space bar, tab (horizontal or vertical), carriage return, form feed, or new line character	9-69
int **isupper**(int c);	Tests c to see if it is an uppercase ASCII alphabetic character	9-69
int **isxdigit**(int c);	Tests c to see if it is a hexadecimal digit	9-69
int **toascii**(int c);	Masks c into a legal ASCII value	9-107
int **tolower**(int char c);	Converts c to lowercase if it is uppercase	9-108
int **toupper**(int char c);	Converts c to uppercase if it is lowercase	9-108

Note: Functions in ctype.h/cctype are expanded inline except when the −pi option is used.

(c) Floating-point math functions (math.h/cmath)

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	9-42
float **acosf**(float x);	Returns the arc cosine of x	9-42
double **acosh**(double x);	Returns the hyperbolic arc cosine of x †	9-42
float **acoshf**(float x);	Returns the hyperbolic arc cosine of x †	9-42
double **acot**(double x);	Returns the arc cotangent of x †	9-43
double **acot2**(double x, double y);	Returns the arc cotangent of x/y †	9-43
float **acot2f**(float x, float y);	Returns the arc cotangent of x/y †	9-43
float **acotf**(float x);	Returns the arc cotangent of x †	9-43
double **acoth**(double x);	Returns the hyperbolic arc cotangent of x †	9-44
float **acothf**(float x);	Returns the hyperbolic arc cotangent of x †	9-44
double **asin**(double x);	Returns the arc sine of x	9-45
float **asinf**(float x);	Returns the arc sine of x	9-45
double **asinh**(double x);	Returns the hyperbolic arc sine of x †	9-45
float **asinhf**(float x);	Returns the hyperbolic arc sine of x †	9-45
double **atan**(double x);	Returns the arc tangent of x	9-47
double **atan2**(double y, double x);	Returns the arc tangent of y/x	9-47
float **atan2f**(float y, float x);	Returns the arc tangent of y/x	9-47
float **atanf**(float x);	Returns the arc tangent of x	9-47
double **atanh**(double x);	Returns the hyperbolic arc tangent of x †	9-48
float **atanhf**(float x);	Returns the hyperbolic arc tangent of x †	9-48
double **ceil**(double x);	Returns the smallest integer ≥ x; expands inline except when −pi is used	9-51
float **ceilf**(float x);	Returns the smallest integer ≥ x; expands inline except when −pi is used	9-51
double **cos**(double x);	Returns the cosine of x	9-53
float **cosf**(float x);	Returns the cosine of x	9-53
double **cosh**(double x);	Returns the hyperbolic cosine of x	9-53
float **coshf**(float x);	Returns the hyperbolic cosine of x	9-53
double **cot**(double x);	Returns the cotangent of x †	9-54

† Enhanced math function. See section 9.3.10 on page 9-22 for information on accessing this function.

(c) Floating-point math functions (math.h/cmath)(Continued)

Function	Description	Page
float **cotf**(float x);	Returns the cotangent of x †	9-54
double **coth**(double x);	Returns the hyperbolic cotangent of x †	9-54
float **cothf**(float x);	Returns the hyperbolic cotangent of x †	9-54
double **exp**(double x);	Returns e^x	9-57
double **exp10**(double x);	Returns 10.0^x †	9-58
float **exp10f**(float x);	Returns 10.0^x †	9-57
double **exp2**(double x);	Returns 2.0^x †	9-58
float **exp2f**(float x);	Returns 2.0^x †	9-58
float **expf**(float x);	Returns e^x	9-57
double **fabs**(double x);	Returns the absolute value of x	9-59
float **fabsf**(float x);	Returns the absolute value of x	9-59
double **floor**(double x);	Returns the largest integer $\leq$ x; expands inline except when −pi is used	9-61
float **floorf**(float x);	Returns the largest integer $\leq$ x; expands inline except when −pi is used	9-61
double **fmod**(double x, double y);	Returns the exact floating-point remainder of x/y	9-62
float **fmodf**(float x, float y);	Returns the exact floating-point remainder of x/y	9-62
double **frexp**(double value, int *exp);	Returns f and exp such that $.5 \leq \|f\| < 1$ and value is equal to $f \times 2^{exp}$	9-65
float **frexpf**(float value, int *exp);	Returns f and exp such that $.5 \leq \|f\| < 1$ and value is equal to $f \times 2^{exp}$	9-65
double **ldexp**(double x, int exp);	Returns $x \times 2^{exp}$	9-71
float **ldexpf**(float x, int exp);	Returns $x \times 2^{exp}$	9-71
double **log**(double x);	Returns the natural logarithm of x	9-72
double **log10**(double x);	Returns the base-10 logarithm of x	9-72
float **log10f**(float x);	Returns the base-10 logarithm of x	9-72
double **log2**(double x);	Returns the base-2 logarithm of x †	9-72
float **log2f**(float x);	Returns the base-2 logarithm of x †	9-72
float **logf**(float x);	Returns the natural logarithm of x	9-72

† Enhanced math function. See section 9.3.10 on page 9-22 for information on accessing this function.

(c) Floating-point math functions (math.h/cmath)(Continued)

Function	Description	Page
double **modf**(double value, double *ip);	Breaks value into a signed integer and a signed fraction	9-80
float **modff**(float value, float *ip);	Breaks value into a signed integer and a signed fraction	9-80
double **pow**(double x, double y);	Returns x^y	9-81
float **powf**(float x, float y);	Returns x^y	9-81
double **powi**(double x, int y);	Returns x^i †	9-81
float **powif**(float x, int y);	Returns x^i †	9-81
double **round**(double x);	Returns x rounded to the nearest integer †	9-86
float **roundf**(float x);	Returns x rounded to the nearest integer †	9-86
double **rsqrt**(double x);	Returns the reciprocal square root of x †	9-87
float **rsqrtf**(float x);	Returns the reciprocal square root of x †	9-87
double **sin**(double x);	Returns the sine of x	9-90
float **sinf**(float x);	Returns the sine of x	9-90
double **sinh**(double x);	Returns the hyperbolic sine of x	9-90
float **sinhf**(float x);	Returns the hyperbolic sine of x	9-90
double **sqrt**(double x);	Returns the nonnegative square root of x	9-91
float **sqrtf**(float x);	Returns the nonnegative square root of x	9-91
double **tan**(double x);	Returns the tangent of x	9-105
float **tanf**(float x);	Returns the tangent of x	9-105
double **tanh**(double x);	Returns the hyperbolic tangent of x	9-106
float **tanhf**(float x);	Returns the hyperbolic tangent of x	9-106
double **trunc**(double x);	Returns x truncated toward 0 †	9-108
float **truncf**(float x);	Returns x truncated toward 0 †	9-108

† Enhanced math function. See section 9.3.10 on page 9-22 for information on accessing this function.

(d) Nonlocal jumps macro and function (setjmp.h/csetjmp)

Function or Macro	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for use by longjmp; this is a macro	9-88
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved environment	9-88

(e) Variable argument macros (stdarg.h/cstdarg)

Macro	Description	Page
type **va_arg**(va_list, type);	Accesses the next argument of type type in a variable-argument list	9-109
void **va_end**(va_list);	Resets the calling mechanism after using va_arg	9-109
void **va_start**(va_list, parmN);	Initializes ap to point to the first operand in the variable-argument list	9-109

(f) C I/O functions (stdio.h/cstdio)

Function	Description	Page
int **add_device**(char *name, unsigned flags, int (*dopen)(), int (*dclose)(), int (*dread)(), int (*dwrite)(), fpos_t (*dlseek)(), int (*dunlink)(), int (*drename)());	Adds a device record to the device table	9-7
void **clearerr**(FILE *_fp);	Clears the EOF and error indicators for the stream that _fp points to	9-52
int **fclose**(FILE *_fp);	Flushes the stream that _fp points to and closes the file associated with that stream	9-59
int **feof**(FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	9-59
int **ferror**(FILE *_fp);	Tests the error indicator for the stream that _fp points to	9-60
int **fflush**(register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	9-60
int **fgetc**(register FILE *_fp);	Reads the next character in the stream that _fp points to	9-60
int **fgetpos**(FILE *_fp, fpos_t *pos);	Stores the object that pos points to to the current value of the file position indicator for the stream that _fp points to	9-60
char ***fgets**(char *_ptr, register int _size, register FILE *_fp);	Reads the next _size minus 1 characters from the stream that _fp points to into array _ptr	9-61

(f) C I/O functions (stdio.h/cstdio) (Continued)

Function	Description	Page
FILE ***fopen**(const char *_fname, const char *_mode);	Opens the file that _fname points to; _mode points to a string describing how to open the file	9-62
int **fprintf**(FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	9-63
int **fputc**(int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	9-63
int **fputs**(const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	9-63
size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);	Reads from the stream pointed to by _fp and stores the input to the array pointed to by _ptr	9-64
FILE ***freopen**(const char *_fname, const char *_mode, register FILE *_fp);	Opens the file that _fname points to using the stream that _fp points to; _mode points to a string describing how to open the file	9-65
int **fscanf**(FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	9-66
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	9-66
int **fsetpos**(FILE *_fp, const fpos_t *_pos);	Sets the file position indicator for the stream that _fp points to to _pos. The pointer _pos must be a value from fgetpos() on the same stream.	9-66
long **ftell**(FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	9-67
size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);	Writes a block of data from the memory pointed to by _ptr to the stream that _fp points to	9-67
int **getc**(FILE *_fp);	Reads the next character in the stream that _fp points to	9-67
int **getchar**(void);	A macro that calls fgetc() and supplies stdin as the argument	9-68
char ***gets**(char *_ptr);	Performs the same function as fgets() using stdin as the input stream	9-68
void **perror**(const char *_s);	Maps the error number in _s to a string and prints the error message	9-80
int **printf**(const char *_format, ...);	Performs the same function as fprintf but uses stdout as its output stream	9-82
int **putc**(int _x, FILE *_fp);	A macro that performs like FPU()	9-82

(f) C I/O functions (stdio.h/cstdio) (Continued)

Function	Description	Page
int **putchar**(int _x);	A macro that calls fputc() and uses stout as the output stream	9-82
int **puts**(const char *_ptr);	Writes the string pointed to by _ptr to stdout	9-83
int **remove**(const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	9-85
int **rename**(const char *_old, const char *_new);	Causes the file with the name pointed to by _old to be known by the name pointed to by _new	9-85
void **rewind**(register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	9-86
int **scanf**(const char *_fmt, ...);	Performs the same function as fscanf() but reads input from stdin	9-87
void **setbuf**(register FILE *_fp, char *_buf);	Returns no value. setbuf() is a restricted version of setvbuf() and defines and associates a buffer with a stream	9-87
int **setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	9-89
int **sprintf**(char *_string, const char *_format, ...);	Performs the same function as fprintf() but writes to the array that _string points to	9-91
int **sscanf** (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf() but reads from the string that _str points to	9-91
FILE ***tmpfile**(void);	Creates a temporary file	9-107
char ***tmpnam**(char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	9-107
int **ungetc**(int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	9-109
int **vfprintf**(FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf() but replaces the argument list with _ap	9-110
int **vprintf**(const char *_format, va_list _ap);	Performs the same function as printf() but replaces the argument list with _ap	9-110
int **vsprintf**(char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf() but replaces the argument list with _ap	9-111

(g) General functions (stdlib.h/cstdlib)

Function	Description	Page
void **abort**(void);	Terminates a program abnormally	9-41
int **abs**(int i);	Returns the absolute value of val; expands inline	9-41
int **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at program termination	9-48
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline except when −pi is used	9-49
int **atoi**(const char *st);	Converts a string to an integer	9-49
long **atol**(const char *st);	Converts a string to a long integer value; expands inline except when −pi is used	9-49
long long **atoll**(const char *st);	Converts a string to a long long integer value; expands inline except when −pi is used	9-49
void ***bsearch**(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	Searches through an array of nmemb objects for the object that key points to	9-50
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	9-51
div_t **div**(int numer, int denom);	Divides numer by denom producing a quotient and a remainder	9-56
void **exit**(int status);	Terminates a program normally	9-57
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	9-64
char ***getenv**(const char *_string)	Returns the environment information for the variable associated with _string	9-68
long **labs**(long i);	Returns the absolute value of i; expands inline	9-41
long long **llabs**(long long i);	Returns the absolute value of i; expands inline	9-41
ldiv_t **ldiv**(long numer, long denom);	Divides numer by denom	9-56
lldiv_t **lldiv**(long long numer, long long denom);	Divides numer by denom	9-56
int **lltoa**(long long val, char *buffer);	Converts val to the equivalent string	9-74
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	9-74
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	9-74
void ***memalign**(size_t alignment, size_t size);	Allocates memory for an object of size bytes aligned to an alignment byte boundary	9-75

(g) General functions (stdlib.h/cstdlib)(Continued)

Function	Description	Page
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc	9-78
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));	Sorts an array of nmemb members; base points to the first member of the unsorted array, and size specifies the size of each member	9-83
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	9-84
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	9-84
void **srand**(unsigned int seed);	Resets the random number generator	9-84
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	9-103
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	9-103
long long **strtoll**(const char *st, char **endptr, int base);	Converts a string to a long long integer	9-103
unsigned long **strtoul**(const char *st, char **endptr**,** int base**);**	Converts a string to an unsigned long integer	9-103
unsigned long long **strtoull**(const char *st, char **endptr**,** int base**);**	Converts a string to an unsigned long long integer	9-103

(h) String functions (*string.h/cstring*)

Function	Description	Page
void ***memchr**(const void *cs, int c, size_t n);	Finds the first occurrence of c in the first n characters of cs; expands inline except when −pi is used	9-75
int **memcmp**(const void *cs, const void *ct, size_t n);	Compares the first n characters of cs to ct; expands inline except when −pi is used	9-76
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s1 to s2	9-76
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s1 to s2	9-77
void ***memset**(void *mem, int ch, size_t length);	Copies the value of ch into the first length characters of mem; expands inline except when −pi is used	9-77
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	9-92
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if −x is used	9-93

(h) String functions (string.h/cstring)(Continued)

Function	Description	Page
int **strcmp**(register const char *string1, register const char *s2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2. Expands inline if −x is used.	9-93
int **strcoll**(const char *string1, const char *string2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; 0 if string1 is equal to string2; >0 if string1 is greater than string2.	9-93
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline except when −pi is used	9-94
size_t **strcspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string that is made up entirely of characters that are not in chs	9-95
char ***strerror**(int errno);	Maps the error number in errno to an error message string	9-95
size_t **strlen**(const char *string);	Returns the length of a string	9-97
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	9-98
int **strncmp**(const char *string1, const char *string2, size_t n);	Compares up to n characters in two strings; expands inline except when −pi is used	9-99
char ***strncpy**(register char *dest, register const char *src, register size_t n);	Copies up to n characters from src to dest; expands inline except when −pi is used	9-100
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	9-101
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline except when −pi is used	9-101
size_t **strspn**(register const char *string, const char *chs);	Returns the length of the initial segment of string, which is entirely made up of characters from chs	9-102
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	9-102
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	9-104
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	Transforms n characters from from, to to	9-105

(i) Time-support functions (time.h/ctime)

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	9-44
clock_t **clock**(void);	Determines the processor time used	9-52
char ***ctime**(const time_t *timer);	Converts calendar time to local time	9-55
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times	9-55
struct tm ***gmtime**(const time_t *timer);	Converts local time to Greenwich Mean Time	9-69
struct tm ***localtime**(const time_t *timer);	Converts time_t value to broken down time	9-71
time_t **mktime**(struct tm *tptr);	Converts broken down time to a time_t value	9-79
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	9-96
time_t **time**(time_t *timer);	Returns the current calendar time	9-106

9.5 Description of Run-Time-Support Functions and Macros

This section describes the run-time-support functions and macros. For each function or macro, the syntax is given in both C and C++. Because the functions and macros originated from C header files, however, program examples are shown in C code only. The same program in C++ code would differ in that the types and functions declared in the header file are introduced into the std namespace.

abort — *Abort*

Syntax for C

#include <stdlib.h>

void **abort**(void);

Syntax for C++

#include <cstdlib>

void **std::abort**(void);

Defined in

exit.c in rts.src

Description

The abort function terminates the program.

Example

```
void abort(void)
{
   exit(EXIT_FAILURE);
}
```

See the exit function on page 9-57.

abs/labs/llabs — *Absolute Value*

Syntax for C

#include <stdlib.h>

int **abs**(int i);
long **labs**(long i);
long long **llabs**(long long i);

Syntax for C++

#include <cstdlib>

int **std::abs**(int i);
long **std::labs**(long i);
long long **std::llabs**(long long i);

Defined in

abs.c in rts.src

Description

The C/C++ compiler supports three functions that return the absolute value of an integer:

- ❏ The abs function returns the absolute value of an integer i.
- ❏ The labs function returns the absolute value of a long i.
- ❏ The llabs function returns the absolute value of a long long i.

acos/acosf — *Arc Cosine*

Syntax for C

#include <math.h>

double **acos**(double x);
float **acosf**(float x);

Syntax for C++

#include <cmath>

double **std::acos**(double x);
float **std::acosf**(float x);

Defined in

acos.c and acosf.c in rts.src

Description

The acos and acosf functions return the arc cosine of a floating-point argument x, which must be in the range [−1,1]. The return value is an angle in the range [0,π] radians.

Example

```
double 3Pi_Over_2;

3Pi_Over_2 = acos(-1.0)  /* Pi  */
           + acos( 0.0)  /* Pi/2 */
           + acos( 1.0); /* 0.0  */
```

acosh/acoshf — *Hyperbolic Arc Cosine*

Syntax for C

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **acosh**(double x);
float **acoshf**(float x);

Syntax for C++

#define _TI_ENHANCED_MATH_H 1
#include <cmath>

double **std::acosh**(double x);
float **std::acoshf**(float x);

Defined in

acosh.c and acoshf.c in rts.src

Description

The acosh and acoshf functions return the hyperbolic arc cosine of a floating-point argument x, which must be in the range [1, infinity]. The return value is ≥ 0.0.

acot/acotf — *Polar Arc Cotangent*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include  <math.h>

double acot(double x);
float acotf(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include  <cmath>

double std::acot(double x);
float std::acotf(float x);
```

Defined in

acot.c and acotf.c in rts.src

Description

The acot and acotf functions return the arc cotangent of a floating-point argument x. The return value is an angle in the range [0, π/2] radians.

Example

```
double realval, radians;

realval = 0.0;
radians = acotf(realval);      /* return value = Pi/2 */
```

acot2/acot2f — *Cartesian Arc Cotangent*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include  <math.h>

double acot2(double x, double y);
float acot2f(float x, float y);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include  <cmath>

double std::acot2(double x, double y);
float std::acot2f(float x, float y);
```

Defined in

acot2.c and acot2f.c in rts.src

Description

The acot2 and acot2f functions return the inverse cotangent of x/y. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [−π, π] radians.

acoth/acothf — *Hyperbolic Arc Cotangent*

Syntax for C

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **acoth**(double x);
float **acothf**(float x);

Syntax for C++

#define _TI_ENHANCED_MATH_H 1
#include <cmath>

double **std::acoth**(double x);
float **std::acothf**(float x);

Defined in

acoth.c and acothf.c in rts.src

Description

The acothf function returns the hyperbolic arc cotangent of a floating-point argument x. The magnitude of x must be ≥ 0.

asctime — *Convert Internal Time to String*

Syntax for C

#include <time.h>

char ***asctime**(const struct tm *timeptr);

Syntax for C++

#include <ctime>

char ***std::asctime**(const struct tm *timeptr);

Defined in

asctime.c in rts.src

Description

The asctime function converts a broken-down time into a string with the following form:

```
Mon Jan 11 11:18:36 1988 \n\0
```

The function returns a pointer to the converted string.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

asin/asinf — *Arc Sine*

Syntax for C

```
#include   <math.h>

double asin(double x);
float asinf(float x);
```

Syntax for C++

```
#include   <cmath>

double std::asin(double x);
float std::asinf(float x);
```

Defined in

asin.c and asinf.c in rts.src

Description

The asin and asinf functions return the arc sine of a floating-point argument x, which must be in the range [−1, 1]. The return value is an angle in the range [−π/2, π/2] radians.

Example

```
double realval, radians;
realval = 1.0;
radians = asin(realval);  /* asin returns π/2 */
```

asinh/asinhf — *Hyperbolic Arc Sine*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double asinh(double x);
float asinhf(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::asinh(double x);
float std::asinhf(float x);
```

Defined in

asinh.c and asinhf.c in rts.src

Description

The asinh and asinhf functions return the hyperbolic arc sine of a floating-point number x. A range error occurs if the magnitude of the argument is too large.

assert — *Insert Diagnostic Information Macro*

Syntax for C

#include <assert.h>

void **assert**(int expr);

Syntax for C++

#include <cassert>

void **std::assert**(int expr);

Defined in

assert.h/cassert as macro

Description

The assert macro tests an expression; depending upon the value of the expression, assert either issues a message and aborts execution or continues execution. This macro is useful for debugging.

- If expr is false, the assert macro writes information about the call that failed to the standard output device and aborts execution.
- If expr is true, the assert macro does nothing.

The header file that defines the assert macro refers to another macro, NDEBUG. If you have defined NDEBUG as a macro name when the assert.h header is included in the source file, the assert macro is defined as:

```
#define assert(ignore)
```

The header file that defines the assert macro refers to another macro, NASSERT. If you have defined NASSERT as a macro name when the assert.h header is included in the source file, the assert macro behaves as if it is a call to the _nassert intrinsic.

Example

In this example, an integer i is divided by another integer j. Since dividing by 0 is an illegal operation, the example uses the assert macro to test j before the division. If j = = 0, assert issues a message and aborts the program.

```
int   i, j;
assert(j);
q = i/j;
```

atan/atanf — *Polar Arc Tangent*

Syntax for C

#include <math.h>

double **atan**(double x);
float **atanf**(float x);

Syntax for C++

#include <cmath>

double **std::atan**(double x);
float **std::atanf**(float x);

Defined in

atan.c and atanf.c in rts.src

Description

The atan and atanf functions return the arc tangent of a floating-point argument x. The return value is an angle in the range [−π/2, π/2] radians.

Example

```
double realval, radians;

realval = 0.0;
radians = atan(realval);      /* radians = 0.0 */
```

atan2/atan2f — *Cartesian Arc Tangent*

Syntax for C

#include <math.h>

double **atan2**(double y, double x);
float **atan2f**(float y, float x);

Syntax for C++

#include <cmath>

double **std::atan2**(double y, double x);
float **std::atan2f**(float y, float x);

Defined in

atan2.c and atan2f.c in rts.src

Description

The atan2 and atan2f functions return the inverse tangent of y/x. The function uses the signs of the arguments to determine the quadrant of the return value. Both arguments cannot be 0. The return value is an angle in the range [−π, π] radians.

Example

```
double rvalu = 0.0, rvalv = 1.0, radians;

radians = atan2(rvalu, rvalv);    /* radians = 0.0 */
```

atanh/atanhf — *Hyperbolic Arc Tangent*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double atanh(double y, double x);
float atanhf(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::atanh(double y, double x);
float std::atanhf(float x);
```

Defined in

atanh.c and atanhf.c in rts.src

Description

The atanh and atanhf functions return the hyperbolic arc tangent of a floating-point argument x. The return value is in the range [−1.0, 1.0].

atexit — *Register Function Called by Exit()*

Syntax for C

```
#include   <stdlib.h>

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

Syntax for C++

```
#include   <cstdlib>

int std::atexit(void (*fun)(void));
```

Defined in

exit.c in rts.src

Description

The atexit function registers the function that is pointed to by *fun*, to be called without arguments at normal program termination. Up to 32 functions can be registered.

When the program exits through a call to the exit function, the functions that were registered are called without arguments in reverse order of their registration.

atof/atoi/atol/ atoll

Convert String to Number

Syntax for C

```
#include   <stdlib.h>

double atof(const char *st);
int atoi(const char *st);
long atol(const char *st);
long long atoll(const char *st);
```

Syntax for C++

```
#include   <cstdlib>

double std::atof(const char *st);
int std::atoi(const char *st);
long std::atol(const char *st);
long long std::atoll(char *st);
```

Defined in

atof.c, atoi.c, atol.c, and atoll.c in rts.src

Description

These functions convert strings to numeric representations:

- The atof function converts a string into a floating-point value. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- The atoi function converts a string into an integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

- The atol function converts a string into a long integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

- The atoll function converts a string into a long long integer. Argument st points to the string; the string must have the following format:

  [*space*] [*sign*] *digits*

The *space* is indicated by a space (character), a horizontal or vertical tab, a carriage return, a form feed, or a new line. Following the *space* is an optional *sign*, and the *digits* that represent the integer portion of the number. In the atof stream, the fractional part of the number follows, then the exponent, including an optional *sign*.

The first character that cannot be part of the number terminates the string.

The functions do not handle any overflow resulting from the conversion.

bsearch — *Array Search*

Syntax for C

#include <stdlib.h>

void ***bsearch**(const void *key, const void *base, size_t nmemb, size_t size**,**
int (*compar)(const void *, const void *));

Syntax for C++

#include <cstdlib>

void ***std::bsearch**(const void *key, const void *base, size_t nmemb,
size_t size**,** int (*compar)(const void *, const void *));

Defined in

bsearch.c in rts.src

Description

The bsearch function searches through an array of nmemb objects for a member that matches the object that key points to. Argument base points to the first member in the array; size specifies the size (in bytes) of each member.

The contents of the array must be in ascending order. If a match is found, the function returns a pointer to the matching member of the array; if no match is found, the function returns a null pointer (0).

Argument compar points to a function that compares key to the array elements. The comparison function should be declared as:

```
int cmp(const void *ptr1, const void *ptr2);
```

The cmp function compares the objects that ptr1 and ptr2 point to and returns one of the following values:

< 0	if *ptr1 is less than *ptr2
0	if *ptr1 is equal to *ptr2
> 0	if *ptr1 is greater than *ptr2

Example

```
int list[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

 int intcmp(const void *ptr1, const void *ptr2)
{
   return *(int*)ptr1 − *(int*)ptr2;
}
```

calloc — *Allocate and Clear Memory*

Syntax for C

#include <stdlib.h>

void ***calloc**(size_t num, size_t size);

Syntax for C++

#include <cstdlib>

void ***std::calloc**(size_t num, size_t size);

Defined in

memory.c in rts.src

Description

The calloc function allocates size bytes (size is an unsigned integer or size_t) for each of num objects and returns a pointer to the space. The function initializes the allocated memory to all 0s. If it cannot allocate the memory (that is, if it runs out of memory), it returns a null pointer (0).

The memory that calloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the −heap option and specifying the desired size of the heap (in bytes) directly after the option. (See section 8.1.3, *Dynamic Memory Allocation*, on page 8-5.)

Example

This example uses the calloc routine to allocate and clear 20 bytes.

```
prt = calloc (10,2)  ;   /*Allocate and clear 20 bytes */
```

ceil/ceilf — *Ceiling*

Syntax for C

#include <math.h>

double **ceil**(double x);
float **ceilf**(float x);

Syntax for C++

#include <cmath>

double **std::ceil**(double x);
float **std::ceilf**(float x);

Defined in

ceil.c and ceilf.c in rts.src

Description

The ceil and ceilf functions return a floating-point number that represents the smallest integer greater than or equal to x.

Example

```
extern float ceil();
float answer

answer = ceilf(3.1415);   /* answer = 4.0  */
answer = ceilf(-3.5);     /* answer = -3.0 */
```

clearerr — *Clear EOF and Error Indicators*

Syntax for C

#include <stdio.h>

void **clearerr**(FILE *_fp);

Syntax for C++

#include <cstdio>

void **std::clearerr**(FILE *_fp);

Defined in

clearerr.c in rts.src

Description

The clearerr functions clears the EOF and error indicators for the stream that _fp points to.

clock — *Processor Time*

Syntax for C

#include <time.h>

clock_t **clock**(void);

Syntax for C++

#include <ctime>

clock_t **std::clock**(void);

Defined in

clock.c in rts.src

Description

The clock function determines the amount of processor time used. It returns an approximation of the processor time used by a program since the program began running. The time in seconds is the return value divided by the value of the macro CLOCKS_PER_SEC.

If the processor time is not available or cannot be represented, the clock function returns the value of [(clock_t) −1].

Note: Writing Your Own Clock Function

The clock function works with the stand-alone simulator (load6x). Used in the load6x environment, clock() returns a cycle accurate count. The clock function returns −1 when used with the HLL debugger.

A host-specific clock function can be written. You must also define the CLOCKS_PER_SEC macro according to the units of your clock so that the value returned by clock() (number of clock ticks) can be divided by CLOCKS_PER_SEC to produce a value in seconds.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

cos/cosf — *Cosine*

Syntax for C

#include <math.h>

double **cos**(double x);
float **cosf**(float x);

Syntax for C++

#include <cmath>

double **std::cos**(double x);
float **std::cosf**(float x);

Defined in

cos.c and cosf.c in rts.src

Description

The cos and cosf functions return the cosine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude might produce a result with little or no significance.

Example

```
double radians, cval;
radians = 0.0;
cval = cos(radians);  /* cval = 0.0 */
```

cosh/coshf — *Hyperbolic Cosine*

Syntax for C

#include <math.h>

double **cosh**(double x);
float **coshf**(float x);

Syntax for C++

#include <cmath>

double **std::cosh**(double x);
float **std::coshf**(float x);

Defined in

cosh.c and coshf.c in rts.src

Description

The cosh and coshf functions return the hyperbolic cosine of a floating-point number x. A range error occurs (errno is set to the value of EDOM) if the magnitude of the argument is too large. These functions are equivalent to $(e^x + e^{-x}) / 2$, but are computationally faster and more accurate.

Example

```
double x, y;

x = 0.0;
y = cosh(x); /* return value = 1.0 */
```

cot/cotf — *Polar Cotangent*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double cot(double x);
float cotf(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::cot(double x);
float std::cotf(float x);
```

Defined in

cot.c and cotf.c in rts.src

Description

The cot and cotf functions return the cotangent of a floating-point argument x, which must not equal 0.0. When x is 0.0, errno is set to the value of EDOM and the function returns the most positive number.

coth/cothf — *Hyperbolic Cotangent*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double coth(double x);
float cothf(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::coth(double x);
float std::cothf(float x);
```

Defined in

coth.c and cothf.c in rts.src

Description

The coth and cothf functions return the hyperbolic cotangent of a floating-point argument x. The magnitude of the return value is ≥ 1.0.

ctime — *Calendar Time*

Syntax for C

#include <time.h>

char ***ctime**(const time_t *timer);

Syntax for C++

#include <ctime>

char ***std::ctime**(const time_t *timer);

Defined in

ctime.c in rts.src

Description

The ctime function converts a calendar time (pointed to by timer) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

The function returns the pointer returned by the asctime function.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

difftime — *Time Difference*

Syntax for C

#include <time.h>

double **difftime**(time_t time1, time_t time0);

Syntax for C++

#include <ctime>

double **std::difftime**(time_t time1, time_t time0);

Defined in

difftime.c in rts.src

Description

The difftime function calculates the difference between two calendar times, time1 minus time0. The return value expresses seconds.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

div/ldiv/lldiv *Division*

Syntax for C

#include <stdlib.h>

div_t **div**(int numer, int denom);
ldiv_t **ldiv**(long numer, long denom);
lldiv_t **lldiv**(long long numer, long long denom);

Syntax for C++

#include <cstdlib>

div_t **std::div**(int numer, int denom);
ldiv_t **std::ldiv**(long numer, long denom);
lldiv_t **std::lldiv**(long long numer, long long denom);

Defined in

div.c in rts.src

Description

These functions support integer division by returning numer (numerator) divided by denom (denominator). You can use these functions to determine both the quotient and the remainder in a single operation.

- The div function performs integer division. The input arguments are integers; the function returns the quotient and the remainder in a structure of type div_t. The structure is defined as follows:

```
typedef struct
{
   int  quot;            /*  quotient */
   int  rem;             /* remainder */
} div_t;
```

- The ldiv function performs long integer division. The input arguments are long integers; the function returns the quotient and the remainder in a structure of type ldiv_t. The structure is defined as follows:

```
typedef struct
{
   long  quot;           /*  quotient */
   long  rem;            /* remainder  */
} ldiv_t;
```

- The lldiv function performs long long integer division. The input arguments are long long integers; the function returns the quotient and the remainder in a structure of type lldiv_t. The structure is defined as follows:

```
typedef struct
{
   long long  quot;      /*  quotient */
   long long  rem;       /* remainder */
} lldiv_t;
```

The sign of the quotient is negative if either but not both of the operands is negative. The sign of the remainder is the same as the sign of the dividend.

exit — *Normal Termination*

Syntax for C

#include <stdlib.h>

void **exit**(int status);

Syntax for C++

#include <cstdlib>

void **std::exit**(int status);

Defined in

exit.c in rts.src

Description

The exit function terminates a program normally. All functions registered by the atexit function are called in reverse order of their registration. The exit function can accept EXIT_FAILURE as a value. (See the abort function on page 9-41).

You can modify the exit function to perform application-specific shut-down tasks. The unmodified function simply runs in an infinite loop until the system is reset.

The exit function cannot return to its caller.

exp/expf — *Exponential*

Syntax for C

#include <math.h>

double **exp**(double x);
float **expf**(float x);

Syntax for C++

#include <cmath>

double **std::exp**(double x);
float **std::expf**(float x);

Defined in

exp.c and expf.c in rts.src

Description

The exp and expf functions return the exponential function of real number x. The return value is the number e raised to the power x. A range error occurs if the magnitude of x is too large.

Example

```
double x, y;

x = 2.0;
y = exp(x);    /* y = approx 7.38 (e*e, e is 2.17828)... */
```

exp10/exp10f — *Exponential*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double exp10(double x);
float exp10f(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::exp10(double x);
float std::exp10f(float x);
```

Defined in

exp10.c and exp10f.c in rts.src

Description

The exp10 and exp10f functions return 10 raised to the power x, where x is a real number. A range error occurs if the magnitude of x is too large.

exp2/exp2f — *Exponential*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double exp2(double x);
float exp2f(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::exp2(double x);
float std::exp2f(float x);
```

Defined in

exp2.c and exp2f.c in rts.src

Description

The exp2 and exp2f functions return 2 to the power x, where x is a real number. A range error occurs if the magnitude of x is too large.

fabs/fabsf — *Absolute Value*

Syntax for C

#include <math.h>

double **fabs**(double x);
float **fabsf**(float x);

Syntax for C++

#include <cmath>

double **std::fabs**(double x);
float **std::fabsf**(float x);

Defined in

fabs.c in rts.src

Description

The fabs and fabsf functions return the absolute value of a floating-point number x.

Example

```
double x, y;

x = –57.5;
y = fabs(x);        /* return value = +57.5 */
```

fclose — *Close File*

Syntax for C

#include <stdio.h>

int **fclose**(FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::fclose**(FILE *_fp);

Defined in

fclose.c in rts.src

Description

The fclose function flushes the stream that _fp points to and closes the file associated with that stream.

feof — *Test EOF Indicator*

Syntax for C

#include <stdio.h>

int **feof**(FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::feof**(FILE *_fp);

Defined in

feof.c in rts.src

Description

The feof function tests the EOF indicator for the stream pointed to by _fp.

ferror — *Test Error Indicator*

Syntax for C

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

Syntax for C++

```
#include   <cstdio>
int std::ferror(FILE *_fp);
```

Defined in ferror.c in rts.src

Description The ferror function tests the error indicator for the stream pointed to by _fp.

fflush — *Flush I/O Buffer*

Syntax for C

```
#include   <stdio.h>
int fflush(register FILE *_fp);
```

Syntax for C++

```
#include   <cstdio>
int std::fflush(register FILE *_fp);
```

Defined in fflush.c in rts.src

Description The fflush function flushes the I/O buffer for the stream pointed to by _fp.

fgetc — *Read Next Character*

Syntax for C

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

Syntax for C++

```
#include   <cstdio>
int std::fgetc(register FILE *_fp);
```

Defined in fgetc.c in rts.src

Description The fgetc function reads the next character in the stream pointed to by _fp.

fgetpos — *Store Object*

Syntax for C

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

Syntax for C++

```
#include   <cstdio>
int std::fgetpos(FILE *_fp, fpos_t *pos);
```

Defined in fgetpos.c in rts.src

Description The fgetpos function stores the object pointed to by pos to the current value of the file position indicator for the stream pointed to by _fp.

fgets — *Read Next Characters*

Syntax for C

#include <stdio.h>

char ***fgets**(char *_ptr, register int _size, register FILE *_fp);

Syntax for C++

#include <cstdio>

char ***std::fgets**(char *_ptr, register int _size, register FILE *_fp);

Defined in

fgets.c in rts.src

Description

The fgets function reads the specified number of characters from the stream pointed to by _fp. The characters are placed in the array named by _ptr. The number of characters read is _size −1.

floor/floorf — *Floor*

Syntax for C

#include <math.h>

double **floor**(double x);
float **floorf**(float x);

Syntax for C++

#include <cmath>

double **std::floor**(double x);
float **std::floorf**(float x);

Defined in

floor.c and floorf.c in rts.src

Description

The floor and floorf functions return a floating-point number that represents the largest integer less than or equal to x.

Example

```
double answer;

answer = floor(3.1415);      /* answer =  3.0 */
answer = floor(-3.5);        /* answer = -4.0 */
```

fmod/fmodf — *Floating-Point Remainder*

Syntax for C

#include <math.h>

double **fmod**(double x, double y);
float **fmodf**(float x, float y);

Syntax for C++

#include <cmath>

double **std::fmod**(double x, double y);
float **std::fmodf**(float x, float y);

Defined in

fmod.c and fmodf.c in rts.src

Description

The fmod and fmodf functions return the exact floating-point remainder of x divided by y. If y == 0, the function returns 0.

The functions are equivalent mathematically to x − trunc(x / y) × y, but not to the C expression written the same way. For example, fmod(x, 3.0) is 0.0, 1.0, or 2.0 for any small integer x > 0.0. When x is large enough that x / y can no longer be expressed exactly, fmod(x, 3.0) continues to yield correct answers, while the C expression returns 0.0 for all values of x.

Example

```
double x, y, r;

x = 11.0;
y = 5.0;
r = fmod(x, y);        /* fmod returns 1.0 */
```

fopen — *Open File*

Syntax for C

#include <stdio.h>

FILE ***fopen**(const char *_fname, const char *_mode);

Syntax for C++

#include <cstdio>

FILE ***std::fopen**(const char *_fname, const char *_mode);

Defined in

fopen.c in rts.src

Description

The fopen function opens the file that _fname points to. The string pointed to by _mode describes how to open the file.

fprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int **fprintf**(FILE *_fp, const char *_format, ...);

Syntax for C++

#include <cstdio>

int **std::fprintf**(FILE *_fp, const char *_format, ...);

Defined in

fprint.c in rts.src

Description

The fprintf function writes to the stream pointed to by _fp. The string pointed to by _format describes how to write the stream.

fputc — *Write Character*

Syntax for C

#include <stdio.h>

int **fputc**(int _c, register FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::fputc**(int _c, register FILE *_fp);

Defined in

fputc.c in rts.src

Description

The fputc function writes a character to the stream pointed to by _fp.

fputs — *Write String*

Syntax for C

#include <stdio.h>

int **fputs**(const char *_ptr, register FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::fputs**(const char *_ptr, register FILE *_fp);

Defined in

fputs.c in rts.src

Description

The fputs function writes the string pointed to by _ptr to the stream pointed to by _fp.

fread — *Read Stream*

Syntax for C

#include <stdio.h>

size_t **fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);

Syntax for C++

#include <cstdio>

size_t **std::fread**(void *_ptr, size_t _size, size_t _count, FILE *_fp);

Defined in

fread.c in rts.src

Description

The fread function reads from the stream pointed to by _fp. The input is stored in the array pointed to by _ptr. The number of objects read is _count. The size of the objects is _size.

free — *Deallocate Memory*

Syntax for C

#include <stdlib.h>

void **free**(void *packet);

Syntax for C++

#include <cstdlib>

void **free**(void *packet);

Defined in

memory.c in rts.src

Description

The free function deallocates memory space (pointed to by packet) that was previously allocated by a malloc, calloc, or realloc call. This makes the memory space available again. If you attempt to free unallocated space, the function takes no action and returns. For more information, see section 8.1.3, *Dynamic Memory Allocation*, on page 8-5.

Example

This example allocates ten bytes and frees them.

```
char *x;
x = malloc(10);       /*  allocate 10 bytes   */
free(x);              /*  free 10 bytes       */
```

freopen — *Open File*

Syntax for C

#include <stdio.h>

FILE ***freopen**(const char *_fname, const char *_mode, register FILE *_fp);

Syntax for C++

#include <cstdio>

FILE ***std::freopen**(const char *_fname, const char *_mode,
register FILE *_fp);

Defined in

fopen.c in rts.src

Description

The freopen function opens the file pointed to by _fname, and associates with it the stream pointed to by _fp. The string pointed to by _mode describes how to open the file.

frexp/frexpf — *Fraction and Exponent*

Syntax for C

#include <math.h>

double **frexp**(double value**,** int *exp);
float **frexpf**(float value**,** int *exp);

Syntax for C++

#include <cmath>

double **std::frexp**(double value**,** int *exp);
float **std::frexpf**(float value**,** int *exp);

Defined in

frexp.c and frexpf.c in rts.src

Description

The frexp and frexpf functions break a floating-point number into a normalized fraction (f) and the integer power of 2. These functions return f and exp such that $0.5 \leq |f| < 1.0$ and value $== f \times 2^{exp}$. The power is stored in the int pointed to by exp. If value is 0, both parts of the result are 0.

Example

```
double fraction;
int exp;

fraction = frexp(3.0, &exp);
/* after execution, fraction is .75 and exp is 2 */
```

fscanf — *Read Stream*

Syntax for C

```
#include   <stdio.h>

int fscanf(FILE *_fp, const char *_fmt, ...);
```

Syntax for C++

```
#include   <cstdio>

int std::fscanf(FILE *_fp, const char *_fmt, ...);
```

Defined in fscanf.c in rts.src

Description The fscanf function reads from the stream pointed to by _fp. The string pointed to by _fmt describes how to read the stream.

fseek — *Set File Position Indicator*

Syntax for C

```
#include   <stdio.h>

int fseek(register FILE *_fp, long _offset, int _ptrname);
```

Syntax for C++

```
#include   <cstdio>

int std::fseek(register FILE *_fp, long _offset, int _ptrname);
```

Defined in fseek.c in rts.src

Description The fseek function sets the file position indicator for the stream pointed to by _fp. The position is specified by _ptrname. For a binary file, use _offset to position the indicator from _ptrname. For a text file, offset must be 0.

fsetpos — *Set File Position Indicator*

Syntax for C

```
#include   <stdio.h>

int fsetpos(FILE *_fp, const fpos_t *_pos);
```

Defined in fsetpos.c in rts.src

Description The fsetpos function sets the file position indicator for the stream pointed to by _fp to _pos. The pointer _pos must be a value from fgetpos() on the same stream.

ftell — *Get Current File Position Indicator*

Syntax for C

#include <stdio.h>

long **ftell**(FILE *_fp);

Syntax for C++

#include <cstdio>

long **std::ftell**(FILE *_fp);

Defined in

ftell.c in rts.src

Description

The ftell function gets the current value of the file position indicator for the stream pointed to by _fp.

fwrite — *Write Block of Data*

Syntax for C

#include <stdio.h>

size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE ***_fp*)**;

Syntax for C++

#include <cstdio>

size_t **std::fwrite**(const void *_ptr, size_t _size, size_t _count,
register FILE ***_fp*)**;

Defined in

fwrite.c in rts.src

Description

The fwrite function writes a block of data from the memory pointed to by _ptr to the stream that _fp points to.

getc — *Read Next Character*

Syntax for C

#include <stdio.h>

int **getc**(FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::getc**(FILE *_fp);

Defined in

fgetc.c in rts.src

Description

The getc function reads the next character in the file pointed to by _fp.

getchar — *Read Next Character From Standard Input*

Syntax for C

#include <stdio.h>

int **getchar**(void);

Syntax for C++

#include <cstdio>

int **std::getchar**(void);

Defined in fgetc.c in rts.src

Description The getchar function reads the next character from the standard input device.

getenv — *Get Environment Information*

Syntax for C

#include <stdlib.h>

char ***getenv**(const char *_string);

Syntax for C++

#include <cstdlib>

char ***std::getenv**(const char *_string);

Defined in trgdrv.c in rts.src

Description The getenv function returns the environment information for the variable associated with _string.

gets — *Read Next From Standard Input*

Syntax for C

#include <stdio.h>

char ***gets**(char *_ptr);

Syntax for C++

#include <cstdio>

char ***std::gets**(char *_ptr);

Defined in fgets.c in rts.src

Description The gets function reads an input line from the standard input device. The characters are placed in the array named by _ptr. Use the function fgets() instead of gets when possible.

gmtime — *Greenwich Mean Time*

Syntax for C

#include <time.h>

struct tm ***gmtime**(const time_t *timer);

Syntax for C++

#include <ctime>

struct tm ***std::gmtime**(const time_t *timer);

Defined in

gmtime.c in rts.src

Description

The gmtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as Greenwich Mean Time.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

isxxx — *Character Typing*

Syntax for C

#include <ctype.h>

int **isalnum**(int c);
int **isalpha**(int c);
int **isascii**(int c);
int **iscntrl**(int c);
int **isdigit**(int c);
int **isgraph**(int c);
int **islower**(int c);
int **isprint**(int c);
int **ispunct**(int c);
int **isspace**(int c);
int **isupper**(int c);
int **isxdigit**(int c);

Syntax for C

#include <cctype>

int **std::isalnum**(int c);
int **std::isalpha**(int c);
int **std::isascii**(int c);
int **std::iscntrl**(int c);
int **std::isdigit**(int c);
int **std::isgraph**(int c);
int **std::islower**(int c);
int **std::isprint**(int c);
int **std::ispunct**(int c);
int **std::isspace**(int c);
int **std::isupper**(int c);
int **std::isxdigit**(int c);

Defined in

is*xxx*.c and ctype.c in rts.src
Also defined in ctype.h/cctype as macros

Description

These functions test a single argument, c, to see if it is a particular type of character — alphabetic, alphanumeric, numeric, ASCII, etc. If the test is true, the function returns a nonzero value; if the test is false, the function returns 0. The character typing functions include:

isalnum	Identifies alphanumeric ASCII characters (tests for any character for which isalpha or isdigit is true)
isalpha	Identifies alphabetic ASCII characters (tests for any character for which islower or isupper is true)
isascii	Identifies ASCII characters (any character 0−127)
iscntrl	Identifies control characters (ASCII characters 0−31 and 127)
isdigit	Identifies numeric characters between 0 and 9 (inclusive)
isgraph	Identifies any nonspace character
islower	Identifies lowercase alphabetic ASCII characters
isprint	Identifies printable ASCII characters, including spaces (ASCII characters 32−126)
ispunct	Identifies ASCII punctuation characters
isspace	Identifies ASCII tab (horizontal or vertical), space bar, carriage return, form feed, and new line characters
isupper	Identifies uppercase ASCII alphabetic characters
isxdigit	Identifies hexadecimal digits (0−9, a−f, A−F)

The C/C++ compiler also supports a set of macros that perform these same functions. The macros have the same names as the functions but are prefixed with an underscore; for example, _isascii is the macro equivalent of the isascii function. In general, the macros execute more efficiently than the functions.

labs/llabs

See abs/labs/llabs on page 9-41.

ldexp/ldexpf

Multiply by a Power of 2

Syntax for C

#include <math.h>

double **ldexp**(double x**,** int exp);
float **ldexpf**(float x**,** int exp);

Syntax for C++

#include <cmath>

double **std::ldexp**(double x**,** int exp);
float **std::ldexpf**(float x**,** int exp);

Defined in

ldexp.c and ldexpf.c in rts.src

Description

The ldexp and ldexpf functions multiply a floating-point number x by 2^{exp} and return $(x \times 2)^{exp}$. The *exp* can be a negative or a positive value. A range error occurs if the result is too large.

Example

```
double result;

result = ldexp(1.5, 5);          /* result is 48.0 */
result = ldexp(6.0, -3);         /* result is 0.75 */
```

ldiv/lldiv

See div/ldiv/lldiv on page 9-56.

localtime

Local Time

Syntax for C

#include <time.h>

struct tm ***localtime**(const time_t *timer);

Syntax for C++

#include <ctime>

struct tm ***std::localtime**(const time_t *timer);

Defined in

localtime.c in rts.src

Description

The localtime function converts a calendar time (pointed to by timer) into a broken-down time, which is expressed as local time. The function returns a pointer to the converted time.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

log/logf — *Natural Logarithm*

Syntax for C

#include <math.h>

double **log**(double x);
float **logf**(float x);

Syntax for C++

#include <cmath>

double **std::log**(double x);
float **std::logf**(float x);

Defined in

log.c and logf.c in rts.src

Description

The log and logf functions return the natural logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 2.718282;
y = logf(x);        /* y = approx 1.0 */
```

log10/log10f — *Common Logarithm*

Syntax for C

#include <math.h>

double **log10**(double x);
float **log10f**(float x);

Syntax for C++

#include <cmath>

double **std::log10**(double x);
float **std::log10f**(float x);

Defined in

log10.c and log10f.c in rts.src

Description

The log10 and log10f functions return the base-10 logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 10.0;
y = log10f(x);         /* y = approx 1.0 */
```

log2/log2f — *Base-2 Logarithm*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double log2(double x);
float log2f(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::log2(double x);
float std::log2f(float x);
```

Defined in

log2.c and log2f.c in rts.src

Description

The log2 and log2f functions return the base-2 logarithm of a real number x. A domain error occurs if x is negative; a range error occurs if x is 0.

Example

```
float x, y;

x = 2.0;
y = log2f(x);          /* y = approx 1.0 */
```

longjmp

See setjmp/longjmp on page 9-88.

lltoa — *Convert Long Long Integer to ASCII*

Syntax for C

no prototype provided

```
int lltoa(long long val, char *buffer);
```

Syntax for C++

no prototype provided

```
int std::lltoa(long long val, char *buffer);
```

Defined in

lltoa.c in rts.src

Description

The lltoa function is a nonstandard (non-ISO) function and is provided for compatibility. The standard equivalent is sprintf. The function is not prototyped in rts.src. The lltoa function converts a long long integer n to an equivalent ASCII string and writes it into the buffer. If the input number val is negative, a leading minus sign is output. The lltoa function returns the number of characters placed in the buffer.

ltoa — *Convert Long Integer to ASCII*

Syntax for C

no prototype provided

int **ltoa**(long val, char *buffer);

Syntax for C++

no prototype provided

int **std::ltoa**(long val, char *buffer);

Defined in

ltoa.c in rts.src

Description

The ltoa function is a nonstandard (non-ISO) function and is provided for compatibility. The standard equivalent is sprintf. The function is not prototyped in rts.src. The ltoa function converts a long integer n to an equivalent ASCII string and writes it into the buffer. If the input number val is negative, a leading minus sign is output. The ltoa function returns the number of characters placed in the buffer.

malloc — *Allocate Memory*

Syntax for C

#include <stdlib.h>

void ***malloc**(size_t size);

Syntax for C++

#include <stdlib.h>

void ***std::malloc**(size_t size);

Defined in

memory.c in rts.src

Description

The malloc function allocates space for an object of size bytes and returns a pointer to the space. If malloc cannot allocate the packet (that is, if it runs out of memory), it returns a null pointer (0). This function does not modify the memory it allocates.

The memory that malloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the −heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 8.1.3, *Dynamic Memory Allocation*, on page 8-5.

memalign — *Align Heap*

Syntax for C

#include <stdlib.h>

void ***memalign**(size_t alignment**,** size_t _size);

Syntax for C++

#include <stdlib.h>

void ***std::memalign**(size_t alignment**,** size_t _size);

Defined in

memory.c in rts.src

Description

The memalign function performs like the ANS/ISO standard malloc function, except that it returns a pointer to a block of memory that is aligned to an *alignment* byte boundary. Thus if _size is 128, and alignment is 16, memalign returns a pointer to a 128-byte block of memory aligned on a 16-byte boundary.

memchr — *Find First Occurrence of Byte*

Syntax for C

#include <string.h>

void ***memchr**(const void *cs, int c, size_t n);

Syntax for C++

#include <cstring>

void ***std::memchr**(const void *cs, int c, size_t n);

Defined in

memchr.c in rts.src

Description

The memchr function finds the first occurrence of c in the first n characters of the object that cs points to. If the character is found, memchr returns a pointer to the located character; otherwise, it returns a null pointer (0).

The memchr function is similar to strchr, except that the object that memchr searches can contain values of 0 and c can be 0.

memcmp — *Memory Compare*

Syntax for C

#include <string.h>

int **memcmp**(const void *cs, const void *ct, size_t n);

Syntax for C++

#include <cstring>

int **std::memcmp**(const void *cs, const void *ct, size_t n);

Defined in

memcmp.c in rts.src

Description

The memcmp function compares the first n characters of the object that ct points to with the object that cs points to. The function returns one of the following values:

< 0	if *cs is less than *ct
0	if *cs is equal to *ct
> 0	if *cs is greater than *ct

The memcmp function is similar to strncmp, except that the objects that memcmp compares can contain values of 0.

memcpy — *Memory Block Copy — Nonoverlapping*

Syntax for C

#include <string.h>

void ***memcpy**(void *s1, const void *s2, register size_t n);

Syntax for C++

#include <cstring>

void ***std::memcpy**(void *s1, const void *s2, register size_t n);

Defined in

memcpy.c in rts.src

Description

The memcpy function copies n characters from the object that s2 points to into the object that s1 points to. If you attempt to copy characters of overlapping objects, the function's behavior is undefined. The function returns the value of s1.

The memcpy function is similar to strncpy, except that the objects that memcpy copies can contain values of 0.

memmove — *Memory Block Copy — Overlapping*

Syntax for C

#include <string.h>

void ***memmove**(void *s1, const void *s2, size_t n);

Syntax for C++

#include <cstring>

void ***std::memmove**(void *s1, const void *s2, size_t n);

Defined in

memmove.c in rts.src

Description

The memmove function moves n characters from the object that s2 points to into the object that s1 points to; the function returns the value of s1. The memmove function correctly copies characters between overlapping objects.

memset — *Duplicate Value in Memory*

Syntax for C

#include <string.h>

void ***memset**(void *mem, int ch, size_t length);

Syntax for C++

#include <cstring>

void ***std::memset**(void *mem, int ch, size_t length);

Defined in

memset.c in rts.src

Description

The memset function copies the value of ch into the first length characters of the object that mem points to. The function returns the value of mem.

minit — *Reset Dynamic Memory Pool*

Syntax for C

no prototype provided

void **minit**(void);

Syntax for C++

no prototype provided

void **std::minit**(void);

Defined in

memory.c in rts.src

Description

The minit function resets all the space that was previously allocated by calls to the malloc, calloc, or realloc functions.

The memory that minit uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the −heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, refer to section 8.1.3, *Dynamic Memory Allocation*, on page 8-5.

Note: No Previously Allocated Objects Are Available After minit

Calling the minit function makes *all* the memory space in the heap available again. Any objects that you allocated previously will be lost; do not try to access them.

mktime — *Convert to Calendar Time*

Syntax for C

#include <time.h>

time_t **mktime**(register struct tm *tptr);

Syntax for C++

#include <ctime>

time_t **std::mktime**(register struct tm *tptr);

Defined in

mktime.c in rts.src

Description

The mktime function converts a broken-down time, expressed as local time, into proper calendar time. The tptr argument points to a structure that holds the broken-down time.

The function ignores the original values of tm_wday and tm_yday and does not restrict the other values in the structure. After successful completion of time conversions, tm_wday and tm_yday are set appropriately and the other components in the structure have values within the restricted ranges. The final value of tm_mday is not sent until tm_mon and tm_year are determined.

The return value is encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value −1.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

Example

This example determines the day of the week that July 4, 2001, falls on.

```
#include <time.h>
static const char *const wday[] = {
            "Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday" };

struct tm time_str;

time_str.tm_year  = 2001 - 1900;
time_str.tm_mon   = 7;
time_str.tm_mday  = 4;
time_str.tm_hour  = 0;
time_str.tm_min   = 0;
time_str.tm_sec   = 1;
time_str.tm_isdst = 1;

mktime(&time_str);

/* After calling this function, time_str.tm_wday    */
/*    contains the day of the week for July 4, 2001 */
```

modf/modff — *Signed Integer and Fraction*

Syntax for C

#include <math.h>

double **modf**(double value**,** double *ip);
float **modff**(float value**,** float *ip);

Syntax for C++

#include <cmath>

double **std::modf**(double value**,** double *ip);
float **std::modff**(float value**,** float *ip);

Defined in

modf.c and modff.c in rts.src

Description

The modf and modff functions break a value into a signed integer and a signed fraction. Each of the two parts has the same sign as the input argument. The function returns the fractional part of value and stores the integer as a double at the object pointed to by iptr.

Example

```
double value, ipart, fpart;

value = -10.125;

fpart = modf(value, &ipart);

/* After execution, ipart contains -10.0, */
/* and fpart contains -.125.               */
```

perror — *Map Error Number*

Syntax for C

#include <stdio.h>

void **perror**(const char *_s);

Syntax for C

#include <cstdio>

void **std::perror**(const char *_s);

Defined in

perror.c in rts.src

Description

The perror function maps the error number in _s to a string and prints the error message.

pow/powf — *Raise to a Power*

Syntax for C

```
#include   <math.h>

double pow(double x, double y);
float powf(float x, float y);
```

Syntax for C++

```
#include   <cmath>

double std::pow(double x, double y);
float std::powf(float x, float y);
```

Defined in

pow.c and powf.c in rts.src

Description

The pow and powf functions return x raised to the power y. These pow functions are equivalent mathematically to exp(y × log(x)) but are faster and more accurate. A domain error occurs if x = 0 and y ≤ 0, or if x is negative and y is not an integer. A range error occurs if the result is too large to represent.

Example

```
double x, y, z;

x = 2.0;
y = 3.0;
x = pow(x, y);          /* return value = 8.0 */
```

powi/powif — *Raise to an Integer Power*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double powi(double x, int y);
float powif(float x, int y);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::powi(double x, int y);
float std::powif(float x, int y);
```

Defined in

powi.c and powif.c in rts.src

Description

The powi and powif functions return x^i. These powi functions are equivalent mathematically to pow(x, (double) i), but are faster and have similar accuracy. A domain error occurs if x = 0 and i ≤ 0, or if x is negative and i is not an integer. A range error occurs if the result is too large to represent.

printf — *Write to Standard Output*

Syntax for C

#include <stdio.h>

int **printf**(const char *_format, ...);

Syntax for C++

#include <cstdio>

int **std::printf**(const char *_format, ...);

Defined in printf.c in rts.src

Description The printf function writes to the standard output device. The string pointed to by _format describes how to write the stream.

putc — *Write Character*

Syntax for C

#include <stdio.h>

int **putc**(int _x, FILE *_fp);

Syntax for C++

#include <cstdio>

int **std::putc**(int _x, FILE *_fp);

Defined in fputc.c in rts.src

Description The putc function writes a character to the stream pointed to by _fp.

putchar — *Write Character to Standard Output*

Syntax for C

#include <stdlib.h>

int **putchar**(int _x);

Syntax for C++

#include <cstdlib>

int **std::putchar**(int _x);

Defined in fputc.c in rts.src

Description The putchar function writes a character to the standard output device.

puts — *Write to Standard Output*

Syntax for C

#include <stdlib.h>

int **puts**(const char *_ptr);

Syntax for C++

#include <cstdlib>

int **std::puts**(const char *_ptr);

Defined in

fputs.c in rts.src

Description

The puts function writes the string pointed to by _ptr to the standard output device.

qsort — *Array Sort*

Syntax for C

#include <stdlib.h>

void **qsort**(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));

Syntax for C++

#include <cstdlib>

void **std::qsort**(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));

Defined in

qsort.c in rts.src

Description

The qsort function sorts an array of nmemb members. Argument base points to the first member of the unsorted array; argument size specifies the size of each member.

This function sorts the array in ascending order.

Argument compar points to a function that compares key to the array elements. Declare the comparison function as:

```
int  cmp(const void *ptr1, const void *ptr2)
```

The cmp function compares the objects that ptr1 and ptr2 point to and returns one of the following values:

< 0	if *ptr1 is less than *ptr2
0	if *ptr1 is equal to *ptr2
> 0	if *ptr1 is greater than *ptr2

Example

```
int list[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

 int intcmp(const void *ptr1, const void *ptr2)
 {
     return *(int*)ptr1 - *(int*)ptr2;
 }
```

rand/srand — *Random Integer*

Syntax for C

#include <stdlib.h>

int **rand**(void);
void **srand**(unsigned int seed);

Syntax for C++

#include <cstdlib>

int **std::rand**(void);
void **std::srand**(unsigned int seed);

Defined in

rand.c in rts.src

Description

These functions work together to provide pseudorandom sequence generation:

- The rand function returns pseudorandom integers in the range 0–RAND_MAX.
- The srand function sets the value of seed so that a subsequent call to the rand function produces a new sequence of pseudorandom numbers. The srand function does not return a value.

If you call rand before calling srand, rand generates the same sequence it would produce if you first called srand with a seed value of 1. If you call srand with the same seed value, rand generates the same sequence of numbers.

realloc — *Change Heap Size*

Syntax for C

#include <stdlib.h>

void ***realloc**(void *packet, size_t size);

Syntax for C++

#include <cstdlib>

void ***std::realloc**(void *packet, size_t size);

Defined in

memory.c in rts.src

Description

The realloc function changes the size of the allocated memory pointed to by packet to the size specified in bytes by size. The contents of the memory space (up to the lesser of the old and new sizes) is not changed.

- If packet is 0, realloc behaves like malloc.
- If packet points to unallocated space, realloc takes no action and returns 0.
- If the space cannot be allocated, the original memory space is not changed and realloc returns 0.
- If size = = 0 and packet is not null, realloc frees the space that packet points to.

If the entire object must be moved to allocate more space, realloc returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, the function returns a null pointer (0).

The memory that calloc uses is in a special memory pool or heap. The constant _ _SYSMEM_SIZE defines the size of the heap as 2K bytes. You can change this amount at link time by invoking the linker with the −heap option and specifying the desired size of the heap (in bytes) directly after the option. For more information, see section 8.1.3, *Dynamic Memory Allocation*, on page 8-5.

remove — *Remove File*

Syntax for C

```
#include   <stdlib.h>
int remove(const char *_file);
```

Syntax for C++

```
#include   <cstdlib>
int std::remove(const char *_file);
```

Defined in remove.c in rts.src

Description The remove function makes the file pointed to by _file no longer available by that name.

rename — *Rename File*

Syntax for C

```
#include   <stdlib.h>
int rename(const char *old_name, const char *new_name);
```

Syntax for C++

```
#include   <cstdlib>
int std::rename(const char *old_name, const char *new_name);
```

Defined in lowlev.c in rts.src

Description The rename function renames the file pointed to by old_name. The new name is pointed to by new_name.

rewind — *Position File Position Indicator to Beginning of File*

Syntax for C

#include <stdlib.h>

int **rewind**(register FILE *_fp);

Syntax for C++

#include <cstdlib>

int **std::rewind**(register FILE *_fp);

Defined in

rewind.c in rts.src

Description

The rewind function sets the file position indicator for the stream pointed to by _fp to the beginning of the file.

round/roundf — *Round to Nearest Integer*

Syntax for C

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **std::round**(double x);
float **std::roundf**(float x);

Syntax for C++

#define _TI_ENHANCED_MATH_H 1
#include <cmath>

double **round**(double x);
float **roundf**(float x);

Defined in

round.c and roundf.c in rts.src

Description

The round and roundf functions return a floating-point number equal to x rounded to the nearest integer. When x is an equal distance from two integers, the even value is returned.

Example

```
float x, y, u, v, r, s, o, p;

x = 2.65;
y = roundf(x);          /* y = 3  */

u = -5.28
v = roundf(u);          /* v = -5 */

r = 3.5
s = roundf(s);          /* s = 4 */

o = 6.5
p = roundf(o);          /* p = 6.0 */
```

rsqrt/rsqrtf — *Reciprocal Square Root*

Syntax for C

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double rsqrt(double x);
float rsqrtf(float x);
```

Syntax for C++

```
#define _TI_ENHANCED_MATH_H 1
#include   <cmath>

double std::rsqrt(double x);
float std::rsqrtf(float x);
```

Defined in rsqrt.c and rsqrtf.c in rts.src

Description The rsqrt and rsqrtf functions return the reciprocal square root of a real number x. The rsqrt(x) function is equivalent mathematically to 1.0 / sqrt(x), but is much faster and has similar accuracy. A domain error occurs if the argument is negative.

scanf — *Read Stream From Standard Input*

Syntax for C

```
#include   <stdlib.h>

int scanf(const char *_fmt, ...);
```

Syntax for C++

```
#include   <cstdlib>

int std::scanf(const char *_fmt, ...);
```

Defined in fscanf.c in rts.src

Description The scanf function reads from the stream from the standard input device. The string pointed to by _fmt describes how to read the stream.

setbuf — *Specify Buffer for Stream*

Syntax for C

```
#include   <stdlib.h>

void setbuf(register FILE *_fp, char *_buf);
```

Syntax for C++

```
#include   <cstdlib>

void std::setbuf(register FILE *_fp, char *_buf);
```

Defined in setbuf.c in rts.src

Description The setbuf function specifies the buffer used by the stream pointed to by _fp. If _buf is set to null, buffering is turned off. No value is returned.

setjmp/longjmp *Nonlocal Jumps*

Syntax for C

#include <setjmp.h>

int **setjmp**(jmp_buf env)
void **longjmp**(jmp_buf env, int _val)

Syntax for C++

#include <csetjmp>

int **std::setjmp**(jmp_buf env)
void **std::longjmp**(jmp_buf env, int _val)

Defined in

setjmp.asm in rts.src

Description

The setjmp.h header defines a type and a macro and declares a function for bypassing the normal function call and return discipline:

- The **jmp_buf** type is an array type suitable for holding the information needed to restore a calling environment.
- The **setjmp** macro saves its calling environment in the jmp_buf argument for later use by the longjmp function.

  If the return is from a direct invocation, the setjmp macro returns the value 0. If the return is from a call to the longjmp function, the setjmp macro returns a nonzero value.
- The **longjmp** function restores the environment that was saved in the jmp_buf argument by the most recent invocation of the setjmp macro. If the setjmp macro was not invoked or if it terminated execution irregularly, the behavior of longjmp is undefined.

  After longjmp is completed, the program execution continues as if the corresponding invocation of setjmp had just returned the value specified by _val. The longjmp function does not cause setjmp to return a value of 0, even if _val is 0. If _val is 0, the setjmp macro returns the value 1.

Example

These functions are typically used to effect an immediate return from a deeply nested function call:

```
#include <setjmp.h>

jmp_buf env;

main()
{
    int errcode;

    if ((errcode = setjmp(env)) == 0)
        nest1();
    else
        switch (errcode)
    . . .
}
    . . .
nest42()
{
    if (input() == ERRCODE42)
    /* return to setjmp call in main */
        longjmp (env, ERRCODE42);
    . . .
}
```

setvbuf — *Define and Associate Buffer With Stream*

Syntax for C

#include <stdio.h>

int **setvbuf**(register FILE *_fp, register char *_buf, register int _type,
register size_t _size);

Syntax for C++

#include <cstdio>

int **std::setvbuf**(register FILE *_fp, register char *_buf, register int _type,
register size_t _size);

Defined in

setvbuf.c in rts.src

Description

The setvbuf function defines and associates the buffer used by the stream pointed to by _fp. If _buf is set to null, a buffer is allocated. If _buf names a buffer, that buffer is used for the stream. The _size specifies the size of the buffer. The _type specifies the type of buffering as follows:

_IOFBF	Full buffering occurs
_IOLBF	Line buffering occurs
_IONBF	No buffering occurs

sin/sinf — *Sine*

Syntax for C

#include <math.h>

double **sin**(double x);
float **sinf**(float x);

Syntax for C++

#include <cmath>

double **std::sin**(double x);
float **std::sinf**(float x);

Defined in

sin.c and sinf.c in rts.src

Description

The sin and sinf functions return the sine of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double radian, sval;     /* sin returns sval        */

radian = 3.1415927;
sval = sin(radian);      /* sin returns approx -1.0 */
```

sinh/sinhf — *Hyperbolic Sine*

Syntax for C

#include <math.h>

double **sinh**(double x);
float **sinhf**(float x);

Syntax for C++

#include <cmath>

double **std::sinh**(double x);
float **std::sinhf**(float x);

Defined in

sinh.c and sinhf.c in rts.src

Description

The sinh and sinhf functions return the hyperbolic sine of a floating-point number x. A range error occurs if the magnitude of the argument is too large. These functions are equivalent to $(e^x - e^{-x}) / 2$, but are computationally faster and more accurate.

Example

```
double x, y;

x = 0.0;
y = sinh(x);       /* y = 0.0 */
```

sprintf — *Write Stream*

Syntax for C

#include <stdio.h>

int **sprintf**(char *_string, const char *_format, ...);

Syntax for C++

#include <cstdio>

int **std::sprintf**(char *_string, const char *_format, ...);

Defined in

sprintf.c in rts.src

Description

The sprintf function writes to the array pointed to by _string. The string pointed to by _format describes how to write the stream.

sqrt/sqrtf — *Square Root*

Syntax for C

#include <math.h>

double **sqrt**(double x);
float **sqrtf**(float x);

Syntax for C++

#include <cmath>

double **std::sqrt**(double x);
float **std::sqrtf**(float x);

Defined in

sqrt.c and sqrtf.c in rts.src

Description

The sqrt function returns the nonnegative square root of a real number x. A domain error occurs if the argument is negative.

Example

```
double x, y;

x = 100.0;
y = sqrt(x);       /* return value = 10.0 */
```

srand

See rand/srand on page 9-84.

sscanf — *Read Stream*

Syntax for C

#include <stdio.h>

int **sscanf**(const char *_str, const char *_fmt, ...);

Syntax for C++

#include <cstdio>

int **std::sscanf**(const char *_str, const char *_fmt, ...);

Defined in

sscanf.c in rts.src

Description

The sscanf function reads from the string pointed to by str. The string pointed to by _format describes how to read the stream.

strcat — *Concatenate Strings*

Syntax for C

#include <string.h>

char ***strcat**(char *string1, const char *string2);

Syntax for C++

#include <cstring>

char ***std::strcat**(char *string1, const char *string2);

Defined in

strcat.c in rts.src

Description

The strcat function appends a copy of string2 (including a terminating null character) to the end of string1. The initial character of string2 overwrites the null character that originally terminated string1. The function returns the value of string1. String1 must be large enough to contain the entire string.

Example

In the following example, the character strings pointed to by *a, *b, and *c are assigned to point to the strings shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
.
.
.
/* a --> "The quick black fox\0"                          */
/* b --> " jumps over \0"                                 */
/* c --> "the lazy dog.\0"                                */

strcat (a,b);

/* a --> "The quick black fox jumps over \0"              */
/* b --> " jumps over \0"                                 */
/* c --> "the lazy dog.\0" */

strcat (a,c);

/*a --> "The quick black fox jumps over the lazy dog.\0"*/
/* b --> " jumps over \0"                                 */
/* c --> "the lazy dog.\0"                                */
```

strchr — *Find First Occurrence of a Character*

Syntax for C

#include <string.h>

char ***strchr**(const char *string, int c);

Syntax for C++

#include <cstring>

char ***std::strchr**(const char *string, int c);

Defined in

strchr.c in rts.src

Description

The strchr function finds the first occurrence of c in string. If strchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

Example

```
char *a = "When zz comes home, the search is on for zs.";
char *b;
char the_z = 'z';

b = strchr(a,the_z);
```

After this example, *b points to the first z in zz.

strcmp/strcoll — *String Compare*

Syntax for C

#include <string.h>

int **strcmp**(const char *string1, register const char *string2);
int **strcoll**(const char *string1, const char *string2);

Syntax for C++

#include <cstring>

int **std::strcmp**(const char *string1, register const char *string2);
int **std::strcoll**(const char *string1, const char *string2);

Defined in

strcmp.c and strcoll.c in rts.src

Description

The strcmp and strcoll functions compare string2 with string1. The functions are equivalent; both functions are supported to provide compatibility with ISO C.

The functions return one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why ask why";

if (strcmp(stra, strb) > 0)
   {
      /*    statements here execute        */
   }
if (strcoll(stra, strc) == 0)
   {
      /* statements here execute also      */
   }
```

strcpy *String Copy*

Syntax for C

#include <string.h>

char ***strcpy***(register char *dest, register const char *src);

Syntax for C++

#include <cstring>

char ***std::strcpy***(register char *dest, register const char *src);

Defined in

strcpy.c in rts.src

Description

The strcpy function copies src (including a terminating null character) into dest. If you attempt to copy strings that overlap, the function's behavior is undefined. The function returns a pointer to dest.

Example

In the following example, the strings pointed to by *a and *b are two separate and distinct memory locations. In the comments, the notation \0 represents the null character:

```
char a[] = "The quick black fox";
char b[] = " jumps over ";

/* a --> "The quick black fox\0"   */
/* b --> " jumps over \0"          */

strcpy(a,b);

/* a --> " jumps over \0"          */
/* b --> " jumps over \0"          */
```

strcspn — *Find Number of Unmatching Characters*

Syntax for C

#include <string.h>

size_t **strcspn**(register const char *string, const char *chs);

Syntax for C++

#include <cstring.h>

size_t **std::strcspn**(register const char *string, const char *chs);

Defined in

strcspn.c in rts.src

Description

The strcspn function returns the length of the initial segment of string, which is made up entirely of characters that are not in chs. If the first character in string is in chs, the function returns 0.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strcspn(stra,strb);    /* length = 0 */
length = strcspn(stra,strc);    /* length = 9 */
```

strerror — *String Error*

Syntax for C

#include <string.h>

char ***strerror**(int errno);

Syntax for C++

#include <cstring>

char ***std::strerror**(int errno);

Defined in

strerror.c in rts.src

Description

The strerror function returns the string “string error.” This function is supplied to provide ISO compatibility.

strftime *Format Time*

Syntax for C

#include <time.h>

size_t ***strftime**(char *out, size_t maxsize, const char *format,
const struct tm *time);

Syntax for C++

#include <ctime>

size_t ***std::strftime**(char *out, size_t maxsize, const char *format,
const struct tm *time);

Defined in

strftime.c in rts.src

Description

The strftime function formats a time (pointed to by time) according to a format string and returns the formatted time in the string out. Up to maxsize characters can be written to out. The format parameter is a string of characters that tells the strftime function how to format the time; the following list shows the valid characters and describes what each character expands to.

Character	Expands to
%a	The abbreviated *weekday* name (Mon, Tue, . . .)
%A	The full *weekday* name
%b	The abbreviated *month* name (Jan, Feb, . . .)
%B	The locale's full *month* name
%c	The *date* and *time* representation
%d	The *day* of the month as a decimal number (0−31)
%H	The *hour* (24-hour clock) as a decimal number (00−23)
%I	The *hour* (12-hour clock) as a decimal number (01−12)
%j	The *day* of the year as a decimal number (001−366)
%m	The *month* as a decimal number (01−12)
%M	The *minute* as a decimal number (00−59)
%p	The locale's equivalency of either *a.m.* or *p.m.*
%S	The *seconds* as a decimal number (00−59)
%U	The *week* number of the year (Sunday is the first day of the week) as a decimal number (00−52)
%x	The *date* representation
%X	The *time* representation
%y	The *year* without century as a decimal number (00−99)

Character	Expands to
%Y	The *year* with century as a decimal number
%Z	The *time zone* name, or by no characters if no time zone exists

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

strlen — *Find String Length*

Syntax for C

#include <string.h>

size_t **strlen**(const char *string);

Syntax for C++

#include <cstring.h>

size_t **std::strlen**(const char *string);

Defined in

strlen.c in rts.src

Description

The strlen function returns the length of string. In C, a character string is terminated by the first byte with a value of 0 (a null character). The returned result does not include the terminating null character.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strlen(stra);       /* length = 13 */
length = strlen(strb);       /* length = 26 */
length = strlen(strc);       /* length = 7  */
```

strncat — *Concatenate Strings*

Syntax for C

#include <string.h>

char ***strncat***(char *dest, const char *src**,** size_t n);

Syntax for C++

#include <cstring>

char ***strncat***(char *dest, const char *src**,** size_t n);

Defined in

strncat.c in rts.src

Description

The strncat function appends up to n characters of src (including a terminating null character) to dest. The initial character of src overwrites the null character that originally terminated dest; strncat appends a null character to the result. The function returns the value of dest.

Example

In the following example, the character strings pointed to by *a, *b, and *c were assigned the values shown in the comments. In the comments, the notation \0 represents the null character:

```
char *a, *b, *c;
size_t size = 13;
.
.
.
/* a--> "I do not like them,\0"                          */;
/* b--> " Sam I am, \0"                                  */;
/* c--> "I do not like green eggs and ham\0"             */;

strncat (a,b,size);

/* a--> "I do not like them, Sam I am, \0"               */;
/* b--> " Sam I am, \0"                                  */;
/* c--> "I do not like green eggs and ham\0"             */;

strncat (a,c,size);

/* a--> "I do not like them, Sam I am, I do not like\0"  */;
/* b--> " Sam I am, \0"                                  */;
/* c--> "I do not like green eggs and ham\0"             */;
```

strncmp — *Compare Strings*

Syntax for C

#include <string.h>

int **strncmp**(const char *string1, const char *string2, size_t n);

Syntax for C++

#include <cstring>

int **std::strncmp**(const char *string1, const char *string2, size_t n);

Defined in

strncmp.c in rts.src

Description

The strncmp function compares up to n characters of string2 with string1. The function returns one of the following values:

< 0	if *string1 is less than *string2
0	if *string1 is equal to *string2
> 0	if *string1 is greater than *string2

Example

```
char *stra = "why ask why";
char *strb = "just do it";
char *strc = "why not?";
size_t size = 4;

if (strcmp(stra, strb, size) > 0)
   {
      /* statements here execute       */
   }
if (strcomp(stra, strc, size) == 0)
   {
      /* statements here execute also  */
   }
```

strncpy — *String Copy*

Syntax for C

#include <string.h>

char ***strncpy***(register char *dest, register const char *src**,**
register size_t n);

Syntax for C++

#include <cstring>

char ***strncpy***(register char *dest, register const char *src**,**
register size_t n);

Defined in

strncpy.c in rts.src

Description

The strncpy function copies up to n characters from src into dest. If src is n characters long or longer, the null character that terminates src is not copied. If you attempt to copy characters from overlapping strings, the function's behavior is undefined. If src is shorter than n characters, strncpy appends null characters to dest so that dest contains n characters. The function returns the value of dest.

Example

Note that strb contains a leading space to make it five characters long. Also note that the first five characters of strc are an *I*, a space, the word *am*, and another space, so that after the second execution of strncpy, stra begins with the phrase *I am* followed by two spaces. In the comments, the notation \0 represents the null character.

```
char stra[100] = "she is the one mother warned you of";
char strb[100] = " he is";
char strc[100] = "I am the one father warned you of";
char strd[100] = "oops";
int length = 5;

strncpy (stra,strb,length);

/* stra--> " he is the one mother warned you of\0" */;
/* strb--> " he is\0"                              */;
/* strc--> "I am the one father warned you of\0"   */;
/* strd--> "oops\0"                                */;

strncpy (stra,strc,length);

/* stra--> "I am  the one mother warned you of\0"  */;
/* strb--> " he is\0"                              */;
/* strc--> "I am the one father warned you of\0"   */;
/* strd--> "oops\0"                                */;

strncpy (stra,strd,length);

/* stra--> "oops\0"                                */;
/* strb--> " he is\0"                              */;
/* strc--> "I am the one father warned you of\0"   */;
/* strd--> "oops\0"                                */;
```

strpbrk — *Find Any Matching Character*

Syntax for C

#include <string.h>

char ***std::strpbrk**(const char *string, const char *chs);

Syntax for C++

#include <cstring>

char ***std::strpbrk**(const char *string, const char *chs);

Defined in

strpbrk.c in rts.src

Description

The strpbrk function locates the first occurrence in string of *any* character in chs. If strpbrk finds a matching character, it returns a pointer to that character; otherwise, it returns a null pointer (0).

Example

```
char *stra = "it was not me";
char *strb = "wave";
char *a;

a = strpbrk (stra,strb);
```

After this example, *a points to the w in was.

strrchr — *Find Last Occurrence of a Character*

Syntax for C

#include <string.h>

char ***strrchr**(const char *string**,** int c);

Syntax for C++

#include <cstring>

char ***std::strrchr**(const char *string**,** int c);

Defined in

strrchr.c in rts.src

Description

The strrchr function finds the last occurrence of c in string. If strrchr finds the character, it returns a pointer to the character; otherwise, it returns a null pointer (0).

Example

```
char *a = "When zz comes home, the search is on for zs";
char *b;
char the_z = 'z';
```

After this example, *b points to the z in zs near the end of the string.

strspn — *Find Number of Matching Characters*

Syntax for C

#include <string.h>

size_t **strspn**(register const char *string, const char *chs);

Syntax for C++

#include <cstring>

size_t **std::strspn**(register const char *string, const char *chs);

Defined in

strspn.c in rts.src

Description

The strspn function returns the length of the initial segment of string, which is entirely made up of characters in chs. If the first character of string is not in chs, the strspn function returns 0.

Example

```
char *stra = "who is there?";
char *strb = "abcdefghijklmnopqrstuvwxyz";
char *strc = "abcdefg";
size_t length;

length = strspn(stra,strb);     /* length = 3 */
length = strspn(stra,strc);     /* length = 0 */
```

strstr — *Find Matching String*

Syntax for C

#include <string.h>

char ***strstr**(register const char *string1, const char *string2);

Syntax for C++

#include <cstring>

char ***std::strstr**(register const char *string1, const char *string2);

Defined in

strstr.c in rts.src

Description

The strstr function finds the first occurrence of string2 in string1 (excluding the terminating null character). If strstr finds the matching string, it returns a pointer to the located string; if it does not find the string, it returns a null pointer. If string2 points to a string with length 0, strstr returns string1.

Example

```
char *stra = "so what do you want for nothing?";
char *strb = "what";
char *ptr;

ptr = strstr(stra,strb);
```

The pointer *ptr now points to the w in what in the first string.

strtod/strtol/ strtoll/strtoul/ strtoull

String to Number

Syntax for C

```
#include   <stdlib.h>

double strtod(const char *st, char **endptr);
long strtol(const char *st, char **endptr, int base);
long long strtoll(const char *st, char **endptr, int base);
unsigned long strtoul(const char *st, char **endptr, int base);
unsigned long long strtoull(const char *st, char **endptr, int base);
```

Syntax for C++

```
#include   <cstdlib>

double std::strtod(const char *st, char **endptr);
long std::strtol(const char *st, char **endptr, int base);
long long std::strtoll(const char *st, char **endptr, int base);
unsigned long std::strtoul(const char *st, char **endptr, int base);
unsigned long long std::strtoull(const char *st, char **endptr, int base);
```

Defined in

strtod.c, strtol.c, strtoll.c, strtoul.c, and strtoull.c in rts.src

Description

These functions convert ASCII strings to numeric values. For each function, argument st points to the original string. Argument endptr points to a pointer; the functions set this pointer to point to the first character after the converted string.The functions that convert to integers also have a third argument, base, which tells the function what base to interpret the string in.

- The strtod function converts a string to a floating-point value. The string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

  The function returns the converted string; if the original string is empty or does not have the correct format, the function returns a 0. If the converted string would cause an overflow, the function returns ±HUGE_VAL; if the converted string would cause an underflow, the function returns 0. If the converted string overflows or underflows, errno is set to the value of ERANGE.

- The strtol function converts a string to a long integer. The string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- The strtoll function converts a string to a long long integer. The string must have the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- The strtoul function converts a string to an unsigned long integer. Specify the string in the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

- The strtoull function converts a string to an unsigned long long integer. Specify the string in the following format:

  [*space*] [*sign*] *digits* [.*digits*] [e|E [*sign*] *integer*]

The space is indicated by a horizontal or vertical tab, space bar, carriage return, form feed, or new line. Following the space is an optional sign and digits that represent the integer portion of the number. The fractional part of the number follows, then the exponent, including an optional sign.

The first unrecognized character terminates the string. The pointer that endptr points to is set to point to this character.

strtok — *Break String into Token*

Syntax for C

#include <string.h>

char ***std::strtok**(char *str1, const char *str2);

Syntax for C++

#include <cstring>

char ***std::strtok**(char *str1, const char *str2);

Defined in

strtok.c in rts.src

Description

Successive calls to the strtok function break str1 into a series of tokens, each delimited by a character from str2. Each call returns a pointer to the next token.

Example

After the first invocation of strtok in the example below, the pointer stra points to the string excuse\0; because strtok has inserted a null character where the first space used to be. In the comments, the notation \0 represents the null character.

```
char stra[] = "excuse me while I kiss the sky";
char *ptr;

ptr = strtok (stra," "); /* ptr --> "excuse\0" */
ptr = strtok (0," ");    /* ptr --> "me\0"     */
ptr = strtok (0," ");    /* ptr --> "while\0"  */
```

strxfrm — *Convert Characters*

Syntax for C

#include <string.h>

size_t **strxfrm**(register char *to, register const char *from, register size_t n);

Syntax for C++

#include <cstring>

size_t **std::strxfrm**(register char *to, register const char *from,
register size_t n);

Defined in

strxfrm.c in rts.src

Description

The strxfrm function converts n characters pointed to by from into the n characters pointed to by to.

tan/tanf — *Tangent*

Syntax for C

#include <math.h>

double **tan**(double x);
float **tanf**(float x);

Syntax for C++

#include <cmath>

double **std::tan**(double x);
float **std::tanf**(float x);

Defined in

tan.c and tanf.c in rts.src

Description

The tan and tanf functions return the tangent of a floating-point number x. The angle x is expressed in radians. An argument with a large magnitude can produce a result with little or no significance.

Example

```
double x, y;

x = 3.1415927/4.0;
y = tan(x);              /* y = approx 1.0 */
```

tanh/tanhf — *Hyperbolic Tangent*

Syntax for C

#include <math.h>

double **tanh**(double x);
float **tanhf**(float x);

Syntax for C++

#include <cmath>

double **std::tanh**(double x);
float **std::tanhf**(float x);

Defined in

tanh.c and tanhf.c in rts.src

Description

The tanh and tanhf functions return the hyperbolic tangent of a floating-point number x.

Example

```
double x, y;

x = 0.0;
y = tanh(x);        /* return value = 0.0 */
```

time — *Time*

Syntax for C

#include <time.h>

time_t **time**(time_t *timer);

Syntax for C++

#include <ctime>

time_t **std::time**(time_t *timer);

Defined in

time.c in rts.src

Description

The time function determines the current calendar time, represented in seconds. If the calendar time is not available, the function returns −1. If timer is not a null pointer, the function also assigns the return value to the object that timer points to.

For more information about the functions and types that the time.h/ctime header declares and defines, see section 9.3.18, *Time Functions (time.h/ctime)*, on page 9-27.

Note: The time Function Is Target-System Specific

The time function is target-system specific, so you must write your own time function.

tmpfile — *Create Temporary File*

Syntax for C

```
#include   <stdlib.h>

FILE *tmpfile(void);
```

Syntax for C++

```
#include   <cstdlib>

FILE *std::tmpfile(void);
```

Defined in tmpfile.c in rts.src

Description The tmpfile function creates a temporary file.

tmpnam — *Generate Valid Filename*

Syntax for C

```
#include   <stdlib.h>

char *tmpnam(char *_s);
```

Syntax for C++

```
#include   <cstdlib>

char *std::tmpnam(char *_s);
```

Defined in tmpnam.c in rts.src

Description The tmpnam function generates a string that is a valid filename.

toascii — *Convert to ASCII*

Syntax for C

```
#include   <ctype.h>

int toascii(int c);
```

Syntax for C++

```
#include   <cctype>

int std::toascii(int c);
```

Defined in toascii.c in rts.src

Description The toascii function ensures that c is a valid ASCII character by masking the lower seven bits. There is also an equivalent macro call _toascii.

tolower/toupper — *Convert Case*

Syntax for C

#include <ctype.h>

int **tolower**(int c);
int **toupper**(int c);

Syntax for C++

#include <cctype>

int **std::tolower**(int c);
int **std::toupper**(int c);

Defined in

tolower.c and toupper.c in rts.src

Description

These functions convert the case of a single alphabetic character c into uppercase or lowercase:

- The tolower function converts an uppercase argument to lowercase. If c is already in lowercase, tolower returns it unchanged.
- The toupper function converts a lowercase argument to uppercase. If c is already in uppercase, toupper returns it unchanged.

The functions have macro equivalents named _tolower and _toupper.

trunc/truncf — *Truncate Toward 0*

Syntax for C

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **trunc**(double x);
float **truncf**(float x);

Syntax for C++

#define _TI_ENHANCED_MATH_H 1
#include <cmath>

double **std::trunc**(double x);
float **std::truncf**(float x);

Defined in

trunc.c and truncf.c in rts.src

Description

The trunc and truncf functions return a floating-point number equal to the nearest integer to x in the direction of 0.

Example

```
float x, y, u, v;

x = 2.35;
y = truncf(x);          /* y = 2 */

u = -5.65;
v = truncf(v);          /* v = -5 */
```

ungetc

Write Character to Stream

Syntax for C

```
#include   <stdlib.h>

int ungetc(int _c, register FILE *_fp);
```

Syntax for C++

```
#include   <cstdlib>

int std::ungetc(int _c, register FILE *_fp);
```

Defined in

ungetc.c in rts.src

Description

The ungetc function writes the character _c to the stream pointed to by _fp.

va_arg/va_end/ va_start

Variable-Argument Macros

Syntax for C

```
#include   <stdarg.h>
typedef    char *va_list;
           type va_arg(va_list, _type);
           void va_end(va_list);
           void va_start(va_list, parmN);
```

Syntax for C++

```
#include   <cstdarg>
typedef    char *std::va_list;
           type std::va_arg(va_list, _type);
           void std::va_end(va_list);
           void std::va_start(va_list, parmN);
```

Defined in

stdarg.h in rts.src

Description

Some functions are called with a varying number of arguments that have varying types. Such a function, called a *variable-argument function*, can use the following macros to step through its argument list at run time. The _ap parameter points to an argument in the variable-argument list.

- The va_start macro initializes _ap to point to the first argument in an argument list for the variable-argument function. The parmN parameter points to the right-most parameter in the fixed, declared list.
- The va_arg macro returns the value of the next argument in a call to a variable-argument function. Each time you call va_arg, it modifies _ap so that successive arguments for the variable-argument function can be returned by successive calls to va_arg (va_arg modifies _ap to point to the next argument in the list). The type parameter is a type name; it is the type of the current argument in the list.
- The va_end macro resets the stack environment after va_start and va_arg are used.

Note that you must call va_start to initialize _ap before calling va_arg or va_end.

Example

```
int    printf (char *fmt...)
       va_list ap;
       va_start(ap, fmt);
       .
       .
       .
       i = va_arg(ap, int);     /* Get next arg, an integer */
       s = va_arg(ap, char *);  /* Get next arg, a string   */
       l = va_arg(ap, long);    /* Get next arg, a long     */
       .
       .
       .
       va_end(ap);              /* Reset                    */
}
```

vfprintf — *Write to Stream*

Syntax for C

#include <stdio.h>

int **vfprintf**(FILE *_fp, const char *_format, va_list _ap);

Syntax for C++

#include <cstdio>

int **std::vfprintf**(FILE *_fp, const char *_format, va_list _ap);

Defined in

vfprintf.c in rts.src

Description

The vfprintf function writes to the stream pointed to by _fp. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

vprintf — *Write to Standard Output*

Syntax for C

#include <stdio.h>

int **vprintf**(const char *_format, va_list _ap);

Syntax for C++

#include <cstdio>

int **std::vprintf**(const char *_format, va_list _ap);

Defined in

vprintf.c in rts.src

Description

The vprintf function writes to the standard output device. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

vsprintf — *Write Stream*

Syntax for C

```
#include   <stdio.h>

int vsprintf(char *_string, const char *_format, va_list _ap);
```

Syntax for C++

```
#include   <cstdio>

int std::vsprintf(char *_string, const char *_format, va_list _ap);
```

Defined in

vsprintf.c in rts.src

Description

The vsprintf function writes to the array pointed to by _string. The string pointed to by _format describes how to write the stream. The argument list is given by _ap.

Chapter 10

Library-Build Utility

When using the C/C++ compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Since it would be cumbersome to include all possible combinations in individual run-time-support libraries, this package includes the source archive, rts.src, which contains all run-time-support functions.

You can build your own run-time-support libraries by using the mk6x utility described in this chapter and the archiver described in the *TMS320C6000 Assembly Language Tools User's Guide*.

Topic **Page**

10.1 Standard Run-Time-Support Libraries

The run-time-support libraries that are shipped with the C6000 code generation tools are built with the following command strings:

Command	Comment
`mk6x –o –ml2 –-RTS rts.src –l rts6200.lib`	base, C6200
`mk6x –o –ml2 –-RTS –me rts.src –l rts6200e.lib`	base, C6200, big endian
`mk6x –o –ml2 –-RTS –mv6400 rts.src –l rts6400.lib`	base, C6400
`mk6x –o –ml2 –-RTS –mv6400 –me rts.src –l rts6400e.lib`	base, C6400, big endian
`mk6x –o –ml2 –-RTS –mv6700 rts.src –l rts6700.lib`	base, C6700
`mk6x –o –ml2 –-RTS –mv6700 –me rts.src –l rts6700e.lib`	base, C6700 big endian

The base option set for every library is:

- ❏ Optimization level 2 (−o or −o2 option)
- ❏ Global structures and arrays accessed as far data; function calls are far calls (−ml2 option)
- ❏ Enables compiling of a C++ run-time-support library based on TI proprietary source code contained in rts.src (−−RTS option).

10.2 Invoking the Library-Build Utility

The syntax for invoking the library-build utility is:

mk6x [*options*] *src_arch1* [**−l***obj.lib1*] [*src_arch2* [**−l***obj.lib2*]] ...

mk6x Command that invokes the utility.

options Options affect how the library-build utility treats your files. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 10.2 and 10.4.)

src_arch The name of a source archive file. For each source archive named, mk6x builds an object library according to the run-time model specified by the command-line options.

−l*obj.lib* The optional object library name. If you do not specify a name for the library, mk6x uses the name of the source archive and appends a *.lib* suffix. For each source archive file specified, a corresponding object library file is created. You cannot build an object library from multiple source archive files.

The mk6x utility runs the compiler program on each source file in the archive to compile and/or assemble it. Then, the utility collects all the object files into the object library. All the tools must be in your PATH environment variable. The utility ignores the environment variables C6X_C_OPTION, C_OPTION, C6X_C_DIR and C_DIR.

10.3 Library-Build Utility Options

Most of the options that are included on the command line correspond directly to options of the same name used with the compiler, assembler, linker, and compiler. The following options apply only to the library-build utility.

−−c Extracts C source files contained in the source archive from the library and leaves them in the current directory after the utility completes execution.

−−h Uses header files contained in the source archive and leaves them in the current directory after the utility completes execution. Use this option to install the run-time-support header files from the rts.src archive that is shipped with the tools.

−−k Overwrites files. By default, the utility aborts any time it attempts to create an object file when an object file of the same name already exists in the current directory, regardless of whether you specified the name or the utility derived it.

−−q Suppresses header information (quiet).

−−RTS Uses default options to compile proprietary C++ source code into a run-time-support library. This option is required if you want to build your own version of the standard C6000 run-time-support libraries.

−−u Does not use the header files contained in the source archive when building the object library. If the desired headers are already in the current directory, there is no reason to reinstall them. This option gives you flexibility in modifying run-time-support functions to suit your application.

−−v Prints progress information to the screen during execution of the utility. Normally, the utility operates silently (no screen messages).

10.4 Options Summary

The other options you can use with the library-build utility correspond directly to the options used with the compiler and assembler. Table 10−1 lists these options. These options are described in detail on the indicated page below.

Table 10−1. Summary of Options and Their Effects

(a) Options that control the compiler

Option	Effect	Page
−D*name*[**=***def*]	Predefines *name*	2-15
−g	Enables symbolic debugging	2-15
−U*name*	Undefines *name*	2-17

(b) Options that are machine-specific

Option	Effect	Page
−ma	Assumes aliased variables	3-25
−mb	Compiles C6400 code compatible with array alignment restrictions of version 4.0 tools or C6200/C6700 object code	2-45
−mc	Prevents reordering of associative floating-point operations	3-28
−me	Produces object code in big-endian format	2-16
−mh*n*	Allows speculative execution	3-14
−mi*n*	Specifies an interrupt threshold value	2-43
−ml*n*	Changes near and far assumptions on four levels (−ml0, −ml1, −ml2, and −ml3)	2-16
−mo	Turns on function subsections	5-13
−ms*n*	Controls code size on four levels (−ms0, −ms1, −ms2, and −ms3)	3-17
−mt	Indicates that specific aliasing techniques are *not* used	3-26
−mu	Turns off software pipelining	3-5
−mv*n*	Selects target version	2-17

Table 10−1. Summary of Options and Their Effects (Continued)

(c) Options that control the parser

Option	Effect	Page
−pi	Disables definition-controlled inlining (but −o3 optimizations still perform automatic inlining)	2-40
−pk	Makes code K&R compatible	7-36
−pr	Enables relaxed mode; ignores strict ISO violations	7-38
−ps	Enables strict ISO mode (for C/C++, not K&R C)	7-38

(d) Parser options that control diagnostics

Option	Effect	Page
−pdr	Issues remarks (nonserious warnings)	2-33
−pdv	Provides verbose diagnostics that display the original source with line wrap	2-34
−pdw	Suppresses warning diagnostics (errors are still issued)	2-34

(e) Options that control the optimization level

Option	Effect	Page
−O0	Compiles with register optimization	3-2
−O1	Compiles with −o0 optimization + local optimization	3-2
−O2 (or −O)	Compiles with −o1 optimization + global optimization	3-3
−O3	Compiles with −o2 optimization + file optimization. Note that mk6x automatically sets −oI0 and −op0.	3-3

(f) Option that controls the assembler

Option	Effect	Page
−as	Keeps labels as symbols	2-24

(g) Options that change the default file extensions

Option	Effect	Page
−ea[.]*new extension*	Sets default extension for assembly files	2-21
−eo[.]*new extension*	Sets default extension for object files	2-21

Chapter 11

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's signature in its link-level name. The process of encoding the signature into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files or linker output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tells you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

Topic **Page**

11.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

dem6x [*options*][*filenames*]

dem6x	Command that invokes the C++ name demangler.
options	Options affect how the name demangler behaves. Options can appear anywhere on the command line. (Options are discussed in section 11.2.)
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, and the linker map file. If no filenames are specified on the command line, dem6x uses standard in.

By default, the C++ name demangler outputs to standard out. You can use the −o file option if you want to output to a file.

11.2 C++ Name Demangler Options

The following options apply only to the C++ name demangler:

−h	Prints a help screen that provides an online summary of the C++ name demangler options
−o *file*	Outputs to the given file rather than to standard out
−u	Specifies that external names do not have a C++ prefix
−v	Enables verbose mode (outputs a banner)

11.3 Sample Usage of the C++ Name Demangler

Example 11−1 shows a sample C++ program and the resulting assembly that is output by the C6000 compiler. In Example 11−1 (b) the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

Example 11−1. Name Mangling

(a) C Code for calories_in_a_banana

```
class banana {

public:

    int calories(void);

    banana();

    ~banana();

};

int calories_in_a_banana(void)
{
    banana x;

    return x.calories();
}
```

(b) Assembly Output for calories_in_a_banana

```
_calories_in_a_banana__Fv:
;** --------------------------------------------------------------------------*
           CALL    .S1     ___ct__6bananaFv  ; |10|
           STW     .D2T2   B3,*SP--(16)      ; |9|
           MVKL    .S2     RL0,B3            ; |10|
           MVKH    .S2     RL0,B3            ; |10|
           ADD     .S1X    8,SP,A4           ; |10|
           NOP             1
RL0:       ; CALL OCCURS                     ; |10|
           CALL    .S1     _calories__6bananaFv ; |12|
           MVKL    .S2     RL1,B3            ; |12|
           ADD     .S1X    8,SP,A4           ; |12|
           MVKH    .S2     RL1,B3            ; |12|
           NOP             2
RL1:       ; CALL OCCURS                     ; |12|
           CALL    .S1     ___dt__6bananaFv  ; |13|
           STW     .D2T1   A4,*+SP(4)        ; |12|
           ADD     .S1X    8,SP,A4           ; |13|
           MVKL    .S2     RL2,B3            ; |13|
           MVK     .S2     0x2,B4            ; |13|
           MVKH    .S2     RL2,B3            ; |13|
RL2:       ; CALL OCCURS                     ; |13|
           LDW     .D2T1   *+SP(4),A4        ; |12|
           LDW     .D2T2   *++SP(16),B3      ; |13|
           NOP             4
           RET     .S2     B3                ; |13|
           NOP             5
           ; BRANCH OCCURS                   ; |13|
```

Executing the C++ name demangler demangles all names that it believes to be mangled. If you enter:

```
% dem6x calories_in_a_banana.asm
```

the result is shown in Example 11−2. The linknames in Example 11−1 (b) `___ct__6bananaFv`, `_calories__6bananaFv`, and `___dt__6bananaFv` are demangled.

Example 11−2. Result After Running the C++ Name Demangler

```
_calories_in_a_banana():
           CALL    .S1     banana::banana()  ; |10|
           STW     .D2T2   B3,*SP--(16)      ; |9|
           MVKL    .S2     RL0,B3            ; |10|
           MVKH    .S2     RL0,B3            ; |10|
           ADD     .S1X    8,SP,A4           ; |10|
           NOP             1
RL0:       ; CALL OCCURS                     ; |10|
           CALL    .S1     banana::_calories() ; |12|
           MVKL    .S2     RL1,B3            ; |12|
           MVKH    .S2     RL1,B3            ; |12|
           ADD     .S1X    8,SP,A4           ; |12|
           NOP             2
RL1:       ; CALL OCCURS                     ; |12|
           CALL    .S1     banana::~banana()  ; |13|
           STW     .D2T1   A4,*+SP(4)        ; |12|
           ADD     .S1X    8,SP,A4           ; |13|
           MVKL    .S2     RL2,B3            ; |13|
           MVK     .S2     0x2,B4            ; |13|
           MVKH    .S2     RL2,B3            ; |13|
RL2:       ; CALL OCCURS                     ; |13|
           LDW     .D2T1   *+SP(4),A4        ; |12|
           LDW     .D2T2   *++SP(16),B3      ; |13|
           NOP             4
           RET     .S2     B3                ; |13|
           NOP             5
           ; BRANCH OCCURS                   ; |13|
```

Appendix A

Glossary

A

ANSI: *See American National Standards Institute.*

alias disambiguation: A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

aliasing: The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

allocation: A process in which the linker calculates the final memory addresses of output sections.

American National Standards Institute(ANSI): An organization that establishes standards voluntarily followed by industries.

archive library: A collection of individual files grouped into a single file by the archiver.

archiver: A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assembly optimizer: A software program that optimizes linear assembly code, which is assembly code that has not been register-allocated or scheduled. The assembly optimizer is automatically invoked with the compiler program, cl6x, when one of the input files has a .sa extension.

assignment statement: A statement that initializes a variable with a value.

autoinitialization: The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time: An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke the linker with the −c option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

B

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

block: A set of statements that are grouped together within braces and treated as an entity.

.bss section: One of the default COFF sections. You use the .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte: Per ANSI C, the smallest addressable unit that can hold a character.

C

C/C++ compiler: A software program that translates C source statements into assembly language source statements. A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

C/C++ optimizer: See *optimizer*

code generator: A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF: *See common object file format*.

command file: A file that contains linker or hex conversion utility options and names input files for the linker or hex conversion utility.

comment: A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

common object file format(COFF): A system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

compiler:

constant: A type whose value cannot change.

cross-reference listing: An output file created by the assembler that lists the symbols it defined, what line they were defined on, which lines referenced them, and their final values.

D

.data section: One of the default COFF sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

direct call: A function call where one function calls another using the function's name.

directives: Special-purpose commands that control the actions and functions of a software tool.

disambiguation: See *alias disambiguation*

dynamic memory allocation: A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at run time. This is accomplished by defining a large memory pool (heap) and using the functions to allocate memory from the heap.

E

emulator: A hardware development system that duplicates the TMS320C6000 operation.

entry point: A point in target memory where execution starts.

environment variable: A system symbol that you define and assign to a string. Environmental variables are often included in batch files, for example, .cshrc.

epilog: The portion of code in a function that restores the stack and returns. See also *pipelined-loop epilog*

executable module: A linked object file that can be executed in a target system.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined or declared in a different program module.

F

file-level optimization: A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program-level optimization, where the compiler uses information that it has about the entire program to optimize your code).

function inlining: The process of inserting code for a function at the point of call. This saves the overhead of a function call and allows the optimizer to optimize the function in the context of the surrounding code.

G

global symbol: A symbol that is either defined in the current module and accessed in another or accessed in the current module but defined in another.

H

hex conversion utility: A utility that converts COFF object files into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer.

I

indirect call: A function call where one function calls another function by giving the address of the called function.

initialization at load time: An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke the linker with the −cr option. This method initializes variables at load time instead of run time.

initialized section: A COFF section that contains executable code or data. An initialized section can be built with the .data, .text, or .sect directive.

integrated preprocessor: A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.

interlist feature: A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.

intrinsics: Operators that are used like functions and produce assembly language code that would otherwise be inexpressible in C, or would take greater time and effort to code.

ISO: International Organization for Standardization. A worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

K

kernel: The body of a software-pipelined loop between the pipelined-loop prolog and the pipelined-loop epilog.

K&R C: Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ISO C compilers should correctly compile and run without modification.

L

label: A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

linear assembly: Assembly code that has not been register-allocated or scheduled, which is used as input for the assembly optimizer. Linear assembly files have a .sa extension.

linker: A software program that combines object files to form an object module that can be allocated into system memory and executed by the device.

listing file: An output file created by the assembler that lists source statements, their line numbers, and their effects on the section program counter (SPC).

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

live in: A value that is defined before a procedure and used as an input to that procedure.

live out: A value that is defined within a procedure and used as an output from that procedure.

loader: A device that places an executable module into system memory.

loop unrolling: An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the efficiency of your code.

M

macro: A user-defined routine that can be used as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro.

macro expansion: The process of inserting source statements into your code in place of a macro call.

map file: An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

memory map: A map of target system memory space that is partitioned into functional blocks.

N

name mangling: A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

O

object file: An assembled or linked file that contains machine-language object code.

object library: An archive library made up of individual object files.

operand: An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

optimizer: A software tool that improves the execution speed and reduces the size of C programs. See also *assembly optimizer*

options: Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

output module: A linked, executable object file that is downloaded and executed on a target system.

output section: A final, allocated section in a linked, executable module.

P

parser: A software tool that reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file used as input for the optimizer or code generator.

partitioning: The process of assigning a data path to each instruction.

pipelined-loop epilog: The portion of code that drains a pipeline in a software-pipelined loop. See also *epilog*

pipelined-loop prolog: The portion of code that primes the pipeline in a software-pipelined loop. See also *prolog*

pop: An operation that retrieves a data object from a stack.

pragma: A preprocessor directive that provides directions to the compiler about how to treat a particular statement.

preprocessor: A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

program-level optimization: An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

prolog: The portion of code in a function that sets up the stack. See also *pipelined-loop prolog*

push: An operation that places a data object on a stack for temporary storage.

R

redundant loops: Two versions of the same loop, where one is a software-pipelined loop and the other is an unpipelined loop. Redundant loops are generated when the TMS320C6000 tools cannot guarantee that the trip count is large enough to pipeline a loop for maximum performance.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

run-time environment: The run time parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

run-time-support functions: Standard ISO functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

run-time-support library: A library file, rts.src, that contains the source for the run time-support functions.

S

section: A relocatable block of code or data that will ultimately be contiguous with other sections in the memory map.

section header: A portion of a COFF object file that contains information about a section in the file. Each section has its own header. The header points to the section's starting address, contains the section's size, etc.

software pipelining: A technique used by the C/C++ optimizer and the assembly optimizer to schedule instructions from a loop so that multiple iterations of the loop execute in parallel.

source file: A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.

stand-alone preprocessor: A software tool that expands macros, #include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.

stand-alone simulator: A software tool that loads and runs an executable COFF .out file. When used with the C I/O libraries, the stand-alone simulator supports all C I/O functions with standard output to the screen.

static variable: A variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.

storage class: An entry in the symbol table that indicates how to access a symbol.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

symbolic debugging: The ability of a software tool to retain symbolic information that can be used by a debugging tool such as a simulator or an emulator.

T

target system: The system on which the object code you have developed is executed.

.text section: One of the default COFF sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

trigraph sequence: A 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.

trip count: The number of times that a loop executes before it terminates.

U

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.

unsigned value: A value that is treated as a nonnegative number, regardless of its actual sign.

V

variable: A symbol representing a quantity that can assume any of a set of values.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z