TMS320C6000 Optimizing C Compiler User's Guide

Literature Number: SPRU187E
February 1999

PRINTED WITH SOY INK™

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

Preface

Read This First

About This Manual

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

- Compiler
- Assembly optimizer
- Standalone simulator
- Library-build utility

The TMS320C6000 C compiler accepts American National Standards Institute (ANSI) standard C source code and produces assembly language source code for the TMS320C6000 device. This user's guide discusses the characteristics of the 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 ANSI C standard. You can use the Kernighan and Ritchie (hereafter referred to as K&R) book as a supplement to this manual.

Before you use the information about the C compiler in this user's guide, you should install the 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:

lnk6x {**–c** | **–cr**} *filenames* [**–o** *name.out*] **–l** *libraryname*

- The TMS320C6200 core is referred to as TMS320C62x and ’C62x. . The TMS320C6700 core is referred to as TMS32067x and ’C67x. TMS320C6000 and ’C6000 can refer to either ’C62x or ’C67x.

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.

TMS320C6x C Source Debugger User’s Guide (literature number SPRU188) tells you how to invoke the ’C6x simulator and emulator versions of the C source debugger interface. This book discusses various aspects of the debugger, including command entry, code execution, data management, breakpoints, profiling, and analysis.

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

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6000 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 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:

American National Standard for Information Systems—Programming Language C X3.159-1989, American National Standards Institute (ANSI standard for C)

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

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

C: A Reference Manual, by Harbison, Samuel P. , Steele, Guy L. (contributor), Prentice Hall Computer Books.

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.

320 Hotline On-line is a trademark of Texas Instruments Incorporated.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580	
TI Literature Response Center U.S.A.	(800) 477-8924	
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285	
U.S. Technical Training Organization	(972) 644-5580	
DSP Hotline		Email: dsph@ti.com

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/pub/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32

Email: epic@ti.com

Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68	
English	+33 1 30 70 11 65	
Francais	+33 1 30 70 11 64	
Italiano	+33 1 30 70 11 67	
EPIC Modem BBS	+33 1 30 70 11 99	
European Factory Repair	+33 4 93 22 25 40	
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: dsph@ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Notes

Chapter 1

Introduction

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

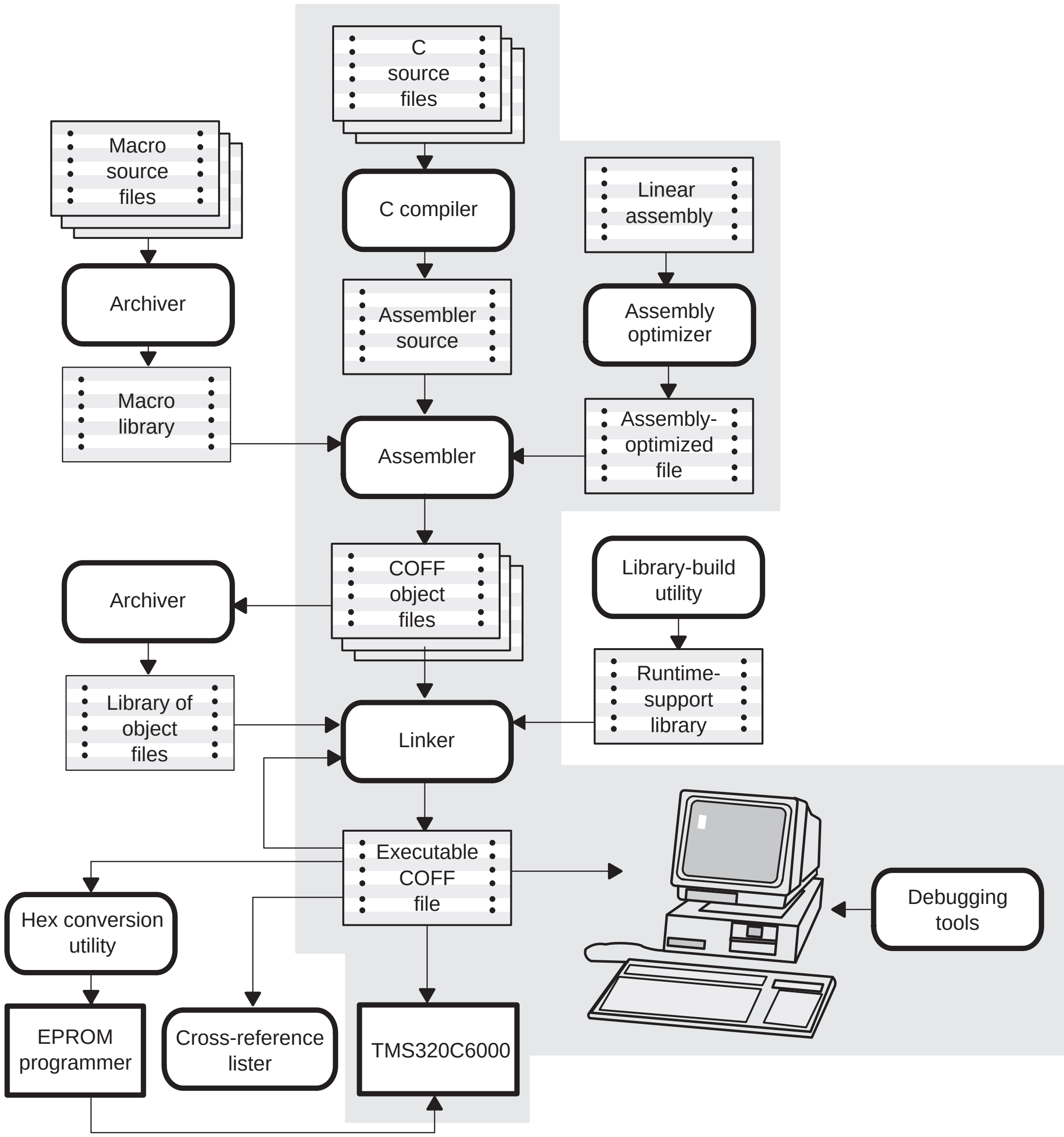
This chapter provides an overview of these tools and introduces the features of the optimizing C compiler. The assembly optimizer is discussed in Chapter 4. The assembler and linker are discussed in detail in the *TMS320C6000 Assembly Language Tools User's Guide*.

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 compiler** accepts C source code and produces ’C6000 assembly language source code. A **shell program**, an **optimizer**, and an **interlist utility** are parts of the compiler:

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

  See Chapter 2, *Using the C Compiler*, for information about how to invoke the C compiler, the optimizer, and the interlist utility using the shell 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 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 are provided as source code in rts.src. The object code for the runtime-support functions is compiled for little-endian mode in rts6201.lib and rts6701.lib, and big-endian mode in rts6201e.lib and rts6701e.lib.

  The **runtime-support libraries** contain the ANSI standard runtime-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the ’C6000 compiler. See Chapter 8, *Runtime Environment.*

- The ’C6000 debugger accepts executable COFF files as input, but most EPROM programmers do not. 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

  For information about these debugging tools, see the *TMS320C6000 C Source Debugger User’s Guide*.

1.2 C Compiler Overview

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

1.2.1 ANSI Standard

The following features pertain to ANSI standards:

❑ **ANSI-standard C**

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

❑ **ANSI-standard runtime support**

The compiler tools come with a complete runtime library. All library functions conform to the ANSI 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. For more information, see Chapter 8, *Runtime 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 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 code and data objects into specific memory areas. COFF also supports source-level debugging.

❑ **Code to initialize data into ROM**

For stand-alone embedded applications, the compiler enables you to link all code and initialization data into ROM, allowing C code to run from reset.

1.2.3 Compiler Interface

The following features pertain to interfacing with the compiler:

- **Compiler shell program**

  The compiler tools include a shell 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 Shell 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, *Runtime Environment*.

1.2.4 Compiler Operation

The following features pertain to the operation of the compiler:

- **Integrated preprocessor**

  The 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 2-23.

- **Optimization**

  The compiler uses a sophisticated optimization pass that employs several advanced techniques for generating efficient, compact code from C source. General optimizations can be applied to any C code, and ’C6000-specific optimizations take advantage of the features specific to the ’C6000 architecture. For more information about the 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 utility**

  The compiler tools include a utility that interlists your original 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 statement. For more information, see section 2.12, *Using the Interlist Utility,* on page 2-42.

- **Library-build utility**

  The library-build utility (mk6x) lets you custom-build object libraries from source for any combination of runtime 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*.

Chapter 2

Using the C Compiler

Translating your source program into code that the ’C6000 can execute is a multistep process. You must compile, assemble, and link your source files to create an executable object file. The ’C6000 compiler tools contain a special shell program, cl6x, that enables you to execute all of these steps with one command. This chapter provides a complete description of how to use the shell program to compile, assemble, and link your programs.

This chapter also describes the preprocessor, inline function expansion features, and interlist utility:

Topic **Page**

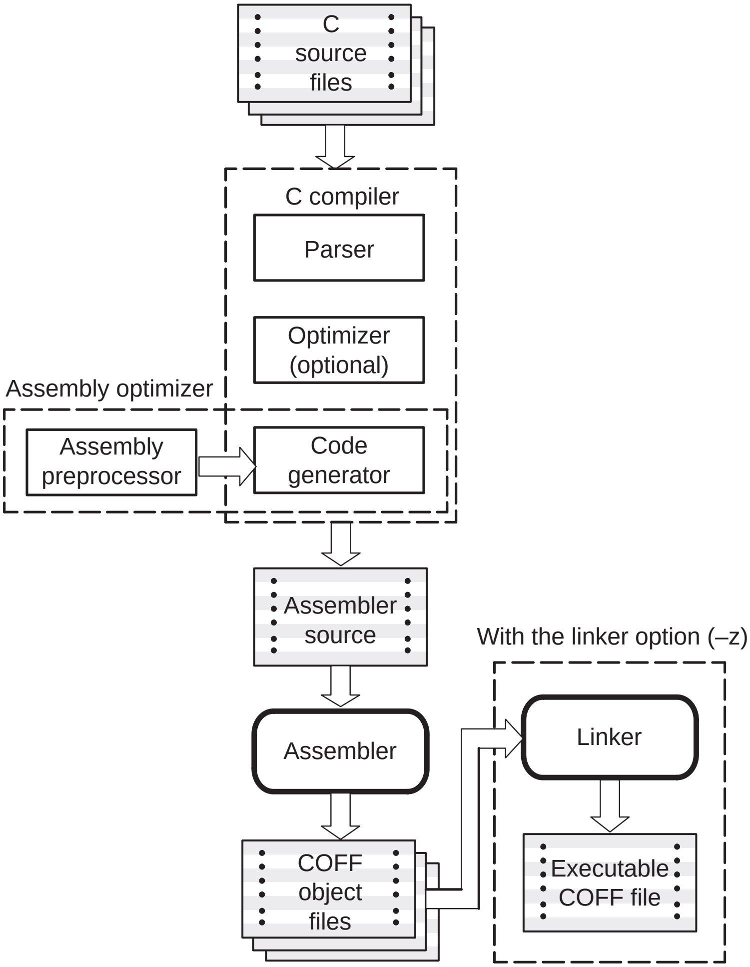
2.1 About the Shell Program

The compiler shell program (cl6x) lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the following:

- The **compiler**, which includes the parser, optimizer, and code generator, accepts C source code and produces ’C6000 assembly language source code.
- The **assembler** generates a COFF object file.
- The **linker** links your files to create an executable object file. The linker is optional with the shell. You can compile and assemble various files with the shell and link them later. See Chapter 5, *Linking C Code,* for information about linking the files in a separate step.

By default, the shell compiles and assembles files; however, you can also link the files using the –z shell option. Figure 2–1 illustrates the path the shell takes with and without using the linker.

Figure 2–1. The Shell Program Overview



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

2.2 Invoking the C Compiler Shell

To invoke the compiler shell, 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 shell processes input files. The options are listed in Table 2–1 on page 2-7.
filenames	One or more C source files, assembly language source files, linear assembly files, or object files
–z	Option that invokes the linker. See Chapter 5, *Linking 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 –z option and its associated information (linker options and object files) must follow all filenames and compiler options on the command line. You can specify all other options (except linker options) and filenames in any order on the command line. 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 suppress progress messages (–q), you enter:

```
cl6x –q symtab file seek.asm find.sa
```

As cl6x encounters each source file, it prints the C filenames in square brackets ([ ]), assembly language filenames in angle brackets (< >), and linear assembly files in braces ({ }). This example uses the –q option to suppress the additional progress information that cl6x produces. Entering this command produces these messages:

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

The normal progress information consists of a banner for each compiler pass and the names of functions as they are processed. The example below shows the output from compiling a single file (symtab) *without* the –q option:

```
% cl6x symtab
[symtab]
TMS320C6x ANSI C Compiler     Version xx
Copyright (c) 1996–1999 Texas Instruments Incorporated
   ”symtab.c”      ==>    symtab
TMS320C6x ANSI C Codegen      Version xx
Copyright (c) 1996–1999 Texas Instruments Incorporated
   ”symtab.c”:     ==>    symtab
TMS320C6x COFF Assembler      Version xx
Copyright (c) 1996–1999 Texas Instruments Incorporated
 PASS 1
 PASS 2

No Errors, No Warnings
```

2.3 Changing the Compiler's Behavior With Options

Options control the operation of both the shell 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 either single letters or 2-letter pairs.
- Options are *not* case sensitive.
- Options are preceded by a hyphen.
- Single-letter options without parameters can be combined. For example, –sgq is equivalent to –s –g –q.
- The following 2-letter pair options that have the same first letter can be combined. For example, –pe, –pf, and –pk can be combined as –pefk.
- Options that have parameters, such as –u*name* and –i*directory*, cannot be combined. They must be specified separately.
- Options with parameters can have a space between the option and parameter or be right next to each other.
- Files and options can occur in any order except the –z option. The –z option must follow all other compiler options and precede any linker options.

You can define default options for the shell by using the C_OPTION environment variable. For a detailed description of the C_OPTION environment variable, see section 2.4.1, *Setting Default Shell Options (C_OPTION and C6X_C_OPTION)*, on page 2-21.

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. Shell Options Summary

(a) Options that control the compiler shell

Option	Effect	Page
–@*filename*	Interprets contents of a file as an extension to the command line	2-15
–c	Disables linking (negates –z)	2-15, 5-5
–d*name*[=*def*]	Predefines *name*	2-15
–g	Enables symbolic debugging	2-15
–i*directory*	Defines #include search path	2-15, 2-25
–k	Keeps the assembly language (.asm) file	2-15
–n	Compiles or assembly optimizes only	2-16
–q	Suppresses progress messages (quiet)	2-16
–qq	Suppresses all messages (super quiet)	2-16
–s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	2-16
–ss	Interlists optimizer comments with C source and assembly statements	2-17, 3-26
–u*name*	Undefines *name*	2-17
–z	Enables linking	2-17

Table 2–1. Shell Options Summary (Continued)

(b) Options that change the default file extensions

Option	Effect	Page
–ea[.]*extension*	Sets a default extension for assembly source files	2-18
–el[.]*extension*	Sets a default extension for assembly optimizer source files	2-18
–eo[.]*extension*	Sets a default extension for object files	2-18

(c) Options that specify files

Option	Effect	Page
–fa*filename*	Changes how assembler source files are identified	2-18
–fc*filename*	Changes how C source files are identified	2-18
–fl*filename*	Changes how assembly optimizer source files are identified	2-18
–fo*filename*	Changes how object code is identified	2-18

(d) Options that specify directories

Option	Effect	Page
–fb*directory*	Specifies an absolute listing file directory	2-19
–ff*directory*	Specifies an assembly listing file and cross-reference listing file directory	2-19
–fr*directory*	Specifies an object file directory	2-19
–fs*directory*	Specifies an assembly file directory	2-19
–ft*directory*	Specifies a temporary file directory	2-19

Table 2–1. Shell Options Summary (Continued)

(e) Options that are machine-specific

Option	Effect	Page
–ma	Indicates that a specific aliasing technique is used	3-21
–me	Produces object code in big-endian format.	2-16
–mg	Allows you to profile optimized code	3-30
–mh*n*	Allows speculative execution	3-10
–mi*n*	Specifies an interrupt threshold value	2-41
–ml*n*	Changes near and far assumptions on four levels (–ml0, –ml1, –and ml2, and –ml3)	2-16
–mr*n*	Make calls to runtime-support functions near (–mr0) or far (–mr0)	7-10
–ms*n*	Controls code size on three levels (–ms0, –ms1, and –ms2)	3-14
–mt	Indicates that specific aliasing techniques are *not* used	3-22, 4-55
–mu	Turns off software pipelining	3-5
–mv*n*	Selects target version	3-12
–mw	Embed software pipelined loop information in the .asm file	3-5
–mz	NEW	

Table 2–1. Shell Options Summary (Continued)

(f) Options that control the parser

Option	Effect	Page
–pi	Disables definition-controlled inlining (but –o3 optimizations still perform automatic inlining)	2-36
–pk	Allows K&R compatibility	7-23
–pl	Generates a raw listing file	2-33
–pm	Combines source files to perform program-level optimization	3-17
–pr	Enables relaxed mode; ignores strict ANSI violations	7-25
–ps	Enables strict ANSI mode	7-25
–px	Generates a cross-reference listing file	2-32

(g) Parser options that control preprocessing

Option	Effect	Page
–ppa	Continues compilation after preprocessing	2-26
–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	2-26
–ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility	2-27
–ppf	Generates a preprocessing output file	2-27
–ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	2-27
–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	2-26
–ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension	2-26

Table 2–1. Shell Options Summary (Continued)

(h) 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.)	2-29
–pden	Displays a diagnostic's identifiers along with its text	2-29
–pdf *outfile*	Writes diagnostics to *outfile* rather than standard error	2-29
–pdr	Issues remarks (nonserious warnings)	2-29
–pds *num*	Suppresses the diagnostic identified by *num*	2-30
–pdse *num*	Categorizes the diagnostic identified by *num* as an error	2-30
–pdsr *num*	Categorizes the diagnostic identified by *num* as a remark	2-30
–pdsw *num*	Categorizes the diagnostic identified by *num* as a warning	2-30
–pdv	Provides verbose diagnostics that display the original source with line-wrap	2-30
–pdw	Suppresses warning diagnostics (errors are still issued)	2-30

Table 2–1. Shell Options Summary (Continued)

(i) 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	3-2
–o3	Uses –o2 optimizations and optimizes the file	3-3
–oi*size*	Sets automatic inlining size (–o3 only)	3-25
–ol0 or –oL0	Informs the optimizer that your file alters a standard library function	3-15
–ol1 or –oL1	Informs the optimizer that your file declares a standard library function	3-15
–ol2 or –oL2	Informs the optimizer that your file does not declare or alter library functions. Overrides the –ol0 and –ol1 options (default).	3-15
–on0	Disables the optimization information file	3-16
–on1	Produces an optimization information file	3-16
–on2	Produces a verbose optimization information file	3-16
–op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	3-17
–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	3-17
–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)	3-17
–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	3-17
–os	Interlists optimizer comments with assembly statements	3-26

Table 2–1. Shell Options Summary (Continued)

(j) Options that control the definition-controlled inline function expansion

Option	Effect	Page
–x0	Disables intrinsic operators, the inline keyword, and automatic inlining	2-36
–x1	Disables the inline keyowrd and automatic inlining	2-36
–x2 or –x	Defines the symbol _INLINE and invokes the optimizer with –o2	2-36

(k) Options that control the assembler

Option	Effect	Page
–aa	Enables absolute listing	2-20
–ad*name*	Sets the *name* symbol.	
–ahc*filename*	Copies the specified file for the assembly module	2-20
–ahi*filename*	Includes the specified file for the assembly module	2-20
–al	Generates an assembly listing file	
–as	Puts labels in the symbol table	2-20
–au*name*	Undefines the predefined constant *name*	
–ax	Generates the cross-reference file	2-20

Table 2–1. Shell Options Summary (Continued)

(l) Options that control the linker

Options	Effect	Page
–a	Generates absolute executable output	5-6
–ar	Generates relocatable executable output	5-6
–b	Disables merge of symbolic debugging information.	5-6
–c	Autoinitializes variables at runtime	5-2, 8-35
–cr	Initializes variables at loadtime	5-2, 8-35
–e *global_symbol*	Defines entry point	5-6
–f *fill_value*	Defines fill value	5-6
–g *global_symbol*	Keeps a *global_symbol* global (overrides –h)	5-6
–h	Makes global symbols static	5-6
–heap *size*	Sets heap size (bytes)	5-6
–i *directory*	Defines library search path	5-6
–l *libraryname*	Supplies library or command filename	5-2
–m *filename*	Names the map file	5-6
–n	Ignores all fill specifications in MEMORY directives	5-7
–o *name.out*	Names the output file	5-2
–q	Suppresses progress messages (quiet)	5-7
–r	Generates relocatable nonexecutable output	5-7
–s	Strips symbol table information and line number entries from the output module	5-7
–stack *size*	Sets stack size (bytes)	5-6
–u *symbol*	Undefines 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 /*.

–c Suppresses the linker and overrides the –z option, which specifies linking. Use this option when you have –z specified in the C_OPTION environment variable and you do not want to link. For more information, see section 5.3, *Disabling the Linker (–c Shell Option)*, on page 5-5.

–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.

–g Generates symbolic debugging directives that are used by the 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 3.10.1, *Debugging Optimized Code (–g and –o Options)*, on page 3-29).

–i*directory* Adds *directory* to the list of directories that the compiler searches for #include files. You can use this option a maximum of 32 times to define several directories; be sure to separate –i options with spaces. 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 2-25.

–k Retains the assembly language output from the compiler or assembly optimizer. Normally, the shell 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 ± 1024 words away from the call site. The linker issues an error message when these situations occur. See section 7.3.4, *The near and far Keywords*, on page 7-9, and section 8.1.5, *Memory Models*, on page 8-6, for more information.

–mv Selects the target CPU version (For more information about the –mv option, see page 3-12.)

–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.

–qq Suppresses all output except error messages

–s Invokes the interlist utility, which interweaves optimizer comments *or* 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 source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C statement. The –s option implies the –k option.

–ss Invokes the interlist utility, which interweaves original C source with compiler-generated assembly language. If the optimizer is invoked (–o*n* option), this option might reorganize your code substantially. For more information, see section 2.12, *Using the Interlist Utility,* on page 2-42.

–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 as an Individual Program*, on page 5-2.

2.3.2 Specifying Filenames

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

Extension	File Type
.c or none (.c is assumed)	C source
.sa	Linear assembly
.asm, .abs, or .s* (extension begins with s)	Assembly source
.obj	Object

Files without extensions are assumed to be C source files. The conventions for filename extensions allow you to compile C files and optimize and assemble assembly files with a single command.

For information about how you can alter the way that the shell interprets individual filenames, see section 2.3.3 on page 2-18. For information about how you can alter the way that the shell interprets and names the extensions of assembly source and object files, see section 2.3.5 on page 2-19.

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 C files in a directory, enter the following:

```
cl6x *.c
```

2.3.3 Changing How the Shell Program Interprets Filenames (–fa, –fc, –fl, and –fo Options)

You can use options to change how the shell interprets your filenames. If the extensions that you use are different from those recognized by the shell, you can use the –fa, –fc, –fl, and –fo 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

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.

2.3.4 Changing How the Shell Program Interprets and Names Extensions (–ea, –el, and –eo Options)

You can use options to change how the shell 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
–el[.] *new extension*	for an assembly optimizer file
–eo[**.**] *new extension*	for an object file

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
```

2.3.5 Specifying Directories

By default, the shell program places the object, assembly, and temporary files that it creates into the current directory. If you want the shell 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. The –ft option overrides the TMP environment variable. (For more information, see section 2.4.2, *Specifying a Temporary File Directory (C6x_TMP and TMP)*, on page 2-22.) To specify a temporary directory, type the directory’s pathname on the command line after the –ft option:

```
cl6x –ft c:\temp
```

2.3.6 Options That Control the Assembler

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

–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.

–ad*name* **–d***name* [=*value*] sets the *name* symbol. This is equivalent to inserting *name* **.set** [*value*] at the beginning of the assembly file. If *value* is omitted, the symbol is set to 1.

–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.

–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*, which 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.*

2.4 Changing the Compiler’s Behavior With Environment Variables

You can define environment variables that set certain software tool parameters you normally use. An *environment variable* is a special system symbol that you define and associate to a string in your system initialization file. The compiler uses this symbol to find or obtain certain types of information.

When you use environment variables, default values are set, making each individual invocation of the compiler simpler because these parameters are automatically specified. When you invoke a tool, you can use command-line options to override many of the defaults that are set with environment variables.

2.4.1 Setting Default Shell Options (C_OPTION and C6X_C_OPTION)

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

Setting the default options with the C_OPTION environment variable is useful when you want to run the shell consecutive times with the same set of options and/or input files. After the shell 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 C_OPTION the environment variable. Select the command for your operating system:

Operating System	Enter
UNIX with C shell	**setenv C_OPTION ”***$option_1$* [*$option_2$* . . .]**”**
UNIX with Bourne or Korn 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 source interlisting (the –s option), and link (the –z option) for Windows, set up the C_OPTION environment variable as follows:

```
set C_OPTION=–qs –z
```

In the following examples, each time you run the compiler shell, 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 shell command line. If you have set –z in the environment variable and want to compile only, use the –c option of the shell. 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 shell options, see section 2.3, *Changing the Compiler's Behavior With Options*, on page 2-6. For more information about linker options, see section 5.4, *Linker Options*, on page 5-6.

2.4.2 Specifying a Temporary File Directory (C6X_TMP and TMP)

The compiler shell program creates intermediate files as it processes your program. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files by using the C6X_TMP or TMP environment variable.

The shell looks for the C6X_TMP environment variable before it looks for the TMP environment variable. Using the C6X_TMP or TMP environment variables allows use of a RAM disk or other file systems. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides. This is useful for protected directories.

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

Operating System	Enter
UNIX with C shell	**setenv TMP "***pathname***"**
UNIX with Bourne or Korn shell	**TMP="***pathname***"** **export TMP**
Windows	**set TMP=***pathname*

Note: For UNIX workstations, be sure to enclose the directory name within quotes.

For example, to set up a directory named temp for intermediate files on your hard drive for Windows, enter:

```
set TMP=c:\temp
```

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 compiler includes standard 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 2–2.

Table 2–2. Predefined Macro Names

Macro Name	Description
_TMS320C6000	Always defined
_TMS320C6200	Defined if target is fixed-point
_TMS320C6700	Defined if target is floating-point
_LITTLE_ENDIAN	Defined if little-endian mode is selected (the –me option is not used); otherwise, it is undefined
_BIG_ENDIAN	Defined if big-endian mode is selected (the –me option is used); otherwise, it is undefined
_LARGE_MODEL	Defined if large-model mode is selected (the –ml option is used); otherwise, it is undefined
_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

† Specified by the ANSI standard

Table 2–2. Predefined Macro Names (Continued)

Macro Name	Description
_ _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*
_INLINE	Expands to 1 under the –x or –x2 option; undefined otherwise
_ _STDC_ _†	Defined to indicate that compiler conforms to ANSI C Standard. See section 7.1, *Characteristics of TMS320C6000 C*, on page 7-2, for exceptions to ANSI C conformance.

† Specified by the ANSI standard

You can use the names listed in Table 2–2 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 C_DIR or C6X_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 C_DIR or C6X_DIR environment variable

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* ...]

You can use up to 32 –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_DIR environment variable.

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

```
setenv C_DIR ”/usr/include;/usr/ucb”
```

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 Directing Preprocessed Output to a File (–ppf *outfile* Option)

The –ppf *outfile* option writes preprocessed output to *outfile* rather than to a file with the same name as the source file but with a .pp extension. Use this option with any of the other preprocessing options except –ppa.

2.5.8 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.

2.5.9 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 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 shell 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 shell 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.

2.6.1 Controlling Diagnostics

The C compiler provides diagnostic options that allow you to modify how the parser interprets your code. You can use these options to 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 2-28 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 shell 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 2–3.

Table 2–3. 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 diagnostic identifiers as defined in Table 2–4.

Table 2–4. 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 2–4 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 2-28.

2.10 Using Inline Function Expansion

When an inline function is called, the 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.

Inline function expansion is performed in one of the following ways:

- Intrinsic operators are expanded by default.
- Automatic inline function expansion is performed on small functions that are invoked by the optimizer with the –o3 option. For more information about automatic inline function expansion, see section 3.8 on page 3-25.
- Definition-controlled inline expansion is performed when you invoke the compiler with optimization (–x option) and the compiler encounters the inline keyword in code.

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline expands code size, and inlining a function that is called in a number of places increases code size. 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, try compiling with the –oi0 option and note the difference in code size.

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. You can stop the inlining by invoking the compiler with the –x0 option.

For details about intrinsics, and a list of the intrinsics, see section 8.5.2, *Using Intrinsics to Access Assembly Language Statements*, on page 8-24.

2.10.2 Controlling Inline Function Expansion (–x Option)

The –x option controls the definition of the _INLINE preprocessor symbol and some types of inline function expansion. There are three levels of expansion:

–x0 Disables the expansion of intrinsic operator functions, definition-controlled inlining with the inline keyword, and the automatic inline function expansions described in section 3.8 on page 3-25.

–x1 Disables definition-controlled inlining with the inline keyword and the automatic inline function expansions described in section 3.8 on page 3-25.

–x2 or **–x** Creates the _INLINE preprocessor symbol and assigns it the value 1, and invokes the optimizer is at level 2 (–o2), thereby enabling definition-controlled inline expansion.

2.10.3 Using the inline Keyword and –o3 Optimization

Definition-controlled inline function expansion is performed when you invoke the compiler with optimization and the compiler encounters the inline keyword in code. 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. Inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this). You can control this type of function inlining with the inline keyword.

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 will perform inline expansion of functions declared with the inline keyword, and can automatically inline small functions.

For a function to be eligible for inlining:

- The function must be declared with the inline keyword, or
- The optimizer must be invoked using the –o3 switch, and
  - The function is very small (controlled by the –oi switch), and
  - The function is declared before it is called

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 # pragmas
- has too large of a stack (too many local variables)

2.10.3.1 Disabling the inline Keyword

When you want to compile code containing the inline keyword without definition-controlled inlining, use the –pi option. When you use the –pi option with –o3 optimizations, automatic inlining is still performed.

2.10.3.2 Declaring a Function as Inline Within a Module

By declaring a function as inline within a module (with the inline keyword), you can specify that the function is inlined within that module. A global symbol for the function is created (code is generated), and the function is inlined only within the module where it is declared as inline. The global symbol can be called by other modules if they do not contain a static inline declaration for the function.

Functions declared as inline are expanded when the optimizer is invoked. Using the –x2 option automatically invokes the optimizer at the default level (–o2).

Use this syntax to declare a function as inline within a module:

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

2.10.3.3 Declaring a Function as Static Inline

Declaring a function as static inline in a header file specifies that the function is inlined in any module that includes the header. This names the function and specifies to expand the function inline, but no code is generated for the function declaration itself. A function declared in this way can be placed in header files and included by all source modules of the program.

Use this syntax to declare a function as static inline:

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

2.10.4 The _INLINE Preprocessor Symbol

The _INLINE preprocessor symbol is defined (and set to 1) if you invoke the parser (or compiler shell utility) with the –x2 (or –x) option. It allows you to write code so that it runs whether or not the optimizer is used. It is used by standard header files included with the compiler to control the declaration of standard C runtime functions.

Example 2–1 on page 2-39 illustrates how the runtime-support library uses the _INLINE preprocessor symbol.

The _INLINE preprocessor symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE preprocessor symbol conditionally defines _ _INLINE so that strlen is declared as static inline only if the _INLINE preprocessor symbol is defined.

If the rest of the modules are compiled with inlining enabled and the string.h header is included, all references to strlen are inlined and the linker does not have to use the strlen in the runtime-support library to resolve any references. Otherwise, the runtime-support library code resolves the references to strlen, and function calls are generated.

Use the _INLINE preprocessor symbol in your header files in the same way that the function libraries use it so that your programs run, regardless of whether inlining is selected for any or all of the modules in your program.

Functions declared as inline are expanded whenever the optimizer is invoked at any level. Functions declared as inline and controlled by the _INLINE preprocessor symbol, such as the runtime-library functions, are expanded whenever the optimizer is invoked and the _INLINE preprocessor symbol is equal to 1. When you declare an inline function in a library, it is recommended that you use the _INLINE preprocessor symbol to control its declaration. If you fail to control the expansion using _INLINE and subsequently compile *without* the optimizer, the call to the function is unresolved.

In Example 2–1, there are two definitions of the strlen function. The first, in the header file, is an inline definition. Note that this definition is enabled and the prototype is declared as static inline only if _INLINE is true; that is, the module including this header is compiled with the –x option.

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 2–1. How the Runtime-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
```

Example 2–1. How the Runtime-Support Library Uses the _INLINE Preprocessor Symbol (Continued)

(b) strlen.c

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

#include <string.h>

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

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

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 threshold defaults to infinity and the compiler assumes that the code is never interrupted.

Interrupts are only 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 will take to execute a loop. If it can determine that the maximum number of cycles is less than the threshold value, then the compiler will disable interrupts around the software pipelined loop. Otherwise, the compiler makes the loop interruptible, 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 7.6.7, *The FUNC_INTERRUPT_THRESHOLD Pragma*, on page 7-18 or the *TMS320C6000 Programmer’s Guide* for more information.

2.12 Using the Interlist Utility

The compiler tools include a utility that interlists C source statements into the assembly language output of the compiler. The interlist utility enables you to inspect the assembly code generated for each C statement. The interlist utility 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 utility is to use the –ss option. To compile and run the interlist utility on a program called function.c, enter:

cl6x –ss function

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

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

Example 2–2 shows a typical interlisted assembly file.

Example 2–2. 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
           MVK     .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVK     .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 utility with the optimizer, see section 3.9, *Using the Interlist Utility With the Optimizer*, on page 3-26.

Chapter 3

Optimizing Your Code

The compiler tools include an optimization program that improves the execution speed and reduces the size of C programs by performing such tasks as simplifying loops, software pipelining, rearranging statements and expressions, and allocating variables into registers.

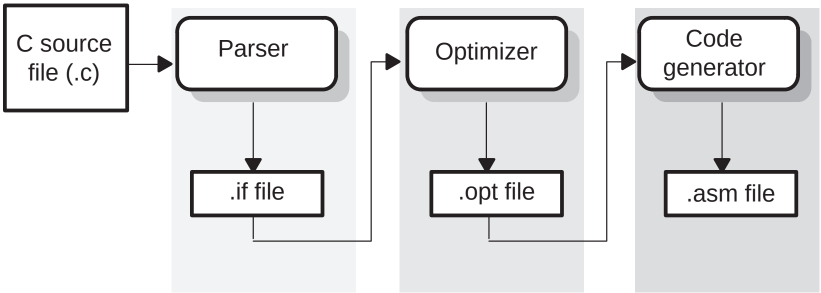
This chapter describes how to invoke the optimizer and describes which optimizations are performed when you use it. This chapter also describes how you can use the interlist utility with the optimizer and how you can profile or debug optimized code.

Topic **Page**

3.1 Using the C Compiler Optimizer

The optimizer runs as a separate pass between the parser and the code generator. Figure 3–1 illustrates the execution flow of the compiler with stand-alone optimization.

Figure 3–1. Compiling a C Program With the Optimizer



The easiest way to invoke the optimizer is to use the cl6x shell 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 3.4, *Using the –o3 Option*, on page 3-15 for more information.

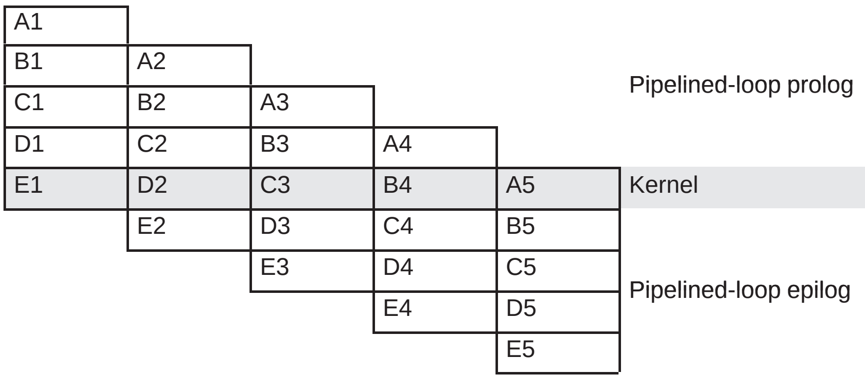
The levels of optimization described above are performed by the stand-alone optimization pass. The code generator performs several additional optimizations, particularly ’C6000-specific optimizations; it does so regardless of whether or not you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

3.2 Software Pipelining

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. When you use the –o2 and –o3 options, the compiler attempts to software pipeline your code with information that it gathers from your program.

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 *TMS320C62x/C67x Programmer’s Guide*.

3.2.1 Turn Off Software Pipelining (–mu Option)

By default, the compiler attempts to software pipeline your loops. You might not want your loops to be software-pipelined for the following reasons:

- To help you debug your loops in C and in linear assembly. Software-pipelined loops are sometimes difficult to debug because the code is not presented serially.
- To save code size. Although software pipelining can greatly improve the efficiency of your code, a pipelined loop usually requires more code size than an unpipelined loop.

This option affects both compiled C code and assembly optimized code.

3.2.2 Software Pipelining Information (–mw Option)

The –mw option embeds software pipelined loop information in the .asm file. This information is used to optimize 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.

Example 3–1. Software Pipelining Information

```
  Loop label: LOOP
  Known Minimum Trip Count         : 8
Known Max Trip Count Factor      : 1
Loop Carried Dependency Bound(^) : 0
Unpartitioned Resource Bound     : 10
Partitioned Resource Bound(*)    : 10
Resource Partition:
                          A–side   B–side
.L units                     6        4
.S units                     3        6
.D units                     8        8
.M units                     3        9
.X cross paths               7        7
.T address paths             8        8
Long read  paths             4        4
Long write paths             0        0
Logical  ops (.LS)           0        0     (.L or .S unit)
Addition ops (.LSD)         11       12     (.L or .S or .D unit)
Bound(.L .S .LS)             5       15
Bound(.L .S .D .LS .LSD)    10*      10*

Searching for software pipeline schedule at ...
   ii = 10 Register is live too long
              |72| –> |74|
              |73| –> |75|
   ii = 11 Cannot allocate machine registers
              Regs Live Always   : 1/5 (A/B-side)
              Max Regs Live      : 14/19
              Max Cond Regs Live : 1/0
   ii = 12 Cannot allocate machine registers
              Regs Live Always   : 1/5 (A/B-side)
              Max Regs Live      : 15/17
              Max Cond Regs Live : 1/0
   ii = 13 Schedule found with 3 iterations in parallel
Done
Speculative load threshold : 48
```

The terms defined below appear in the software pipelining information. For more information on each term, see the *TMS320C62x/C67x 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 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 .LSD) = ceil((.L + .S + .D + .LS + .LSD) / 3)

- **Speculative load threshold.** The number of bytes that are read if speculative execution is enabled. Use this value with the –mh option to eliminate loop epilogs and save code size.

3.2.2.1 Searching for Software Pipeline Messages

The following messages can appear when the compiler or assembly optimizer is searching for a software pipeline:

- **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.

- **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.

- **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.

- **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.

- **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 Cond 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.

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

3.2.2.2 Loop Disqualified for Software Pipelining Messages

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

- **Unknown trip counter variable**. The compiler was unable to identify a trip counter that is a downcounter.
- **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.
- **Unknown trip count**. The minimum trip count is unknown and it is required to software pipeline the loop.
- **Cannot identify trip counter**. The loop trip counter could not be identified or was used incorrectly in the loop body.
- **Too many instructions**. There are too many instructions in the loop to software pipeline.
- **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.

3.2.3 Removing Epilogs

Normally, software pipelined loops require an epilog to complete execution. However, using a technique called *speculative execution*, it is possible to remove many epilogs, significantly reducing code size and improving loop preformance.

3.2.3.1 Speculative Execution

An instruction is *speculatively executed* if it is executed before it is known whether the result will be used. If the result is unused, it is discarded. To speculatively execute an instruction, the compiler moves it past a branch. This has the same effect as moving a statement out of the body of an if statement.

An instruction can be legally speculatively executed only if the discarded result does not change the program semantics. In a speculatively executed instruction, not only is the result discarded, but also the previous value in that location. For instance, an instruction which writes to memory cannot be speculatively executed, because some other instruction might read that location.

Load instructions might be able to be speculatively executed, but it is possible that the address register is invalid if the result will not be used. This means that a load instruction might access an address that is past the beginning or end of valid memory, causing a fault.

Instructions which perform saturating arithmetic cannot be speculatively executed if the CSR register (which contains the SAT bit) is read anywhere in the function. See section 8.5.4, *SAT Bit Side Effects*, on page 8-28 for details.

3.2.3.2 Disqualifying Epilog Removal

To eliminate the epilog from a software pipelined loop, the compiler may need to speculatively execute instructions. If, in order to remove the epilog, any of the following are speculatively executed, the epilog will not be removed:

- Store instructions
- Load instructions that would read past the end of an array
- Load instructions that would read from a volatile memory location
- MVC or IDLE instructions
- Saturating instructions
- Instructions that define a register needed after the loop is done.

3.2.3.3 Aggressively Removing Epilogs (–mh Option)

The –mh*n* option can significantly reduce code size by allowing the compiler to aggressively eliminate software pipeline loop epilogs. The –mhn option indicates that load instructions may be issued which read up to *n* bytes past the beginning or end of an buffer. This means you guarantee all of the buffers read are at least *n* bytes away from the borders of valid memory. The threshold value *n* is optional. If the value is not specified, an unlimited number of bytes may be read past the end.

Since load instructions in loops most often advance through buffers of data, speculatively executing such an instruction might advance past the end of the buffer, reading from an address which might be invalid memory. To eliminate a software pipelined loop epilog by using speculative execution, the complier must compute the worst case of a load reaching past the beginning or end of a buffer, so that this does not happen.

For each loop kernel, the compiler calculates how far (both forward and backward) the loop might read past the end of a buffer if the epilog were removed. If the threshold that you specified is less than the calculated buffer distance for a loop, the epilog is not removed. If the threshold is greater than or equal to the calculated buffer distance, the loads can be speculatively executed and the epilog may be removed.

For example, in the following software pipelined loop kernel, if the epilog is removed, seven extra load instructions are performed through *A0++, reading 14 bytes past the end of the buffer pointed to by A0. If –mh14 or higher is used, the epilog can be removed from this loop.

```
LOOP:       ; PIPED LOOP KERNEL
              ADD     .L1     A5,A4,A4      ; |6|
  ||          MPY     .M1X    B4,A3,A5      ;@@ |6|
  || [ B0]    B       .S2     L3            ;@@@@@ |5|
  || [ B0]    SUB     .L2     B0,1,B0       ;@@@@@@ |5|
  ||          LDH     .D1T1   *A0++,A3      ;@@@@@@@ |6|
  ||          LDH     .D2T2   *B5++,B4      ;@@@@@@@ |6|
```

Note: Padding Data Sections

Speculative execution makes it possible for the compiler to generate code that reads past the beginning or end of a data section. Use the linker to pad the beginning and end of data sections to allow for the speculative reads. The threshold argument to the –mh option indicates the size of the required padding on each end.

3.2.4 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. If this option is not used, the compiler will generate ’C62x (fixed point) code.

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 runtime trip count is less than the loop's minimum trip count. The second loop is the software pipelined loop, and it executes when the runtime 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)*/
```

3.3.1 Reduce Code Size (–ms*n* Option)

Redundant loops allow the compiler to choose the most efficient method for code execution; however, this occurs at the expense of code size. If code size is an issue, use the –ms*n* option when you invoke the optimizer with the –o (–o2 or –o3) option. These options affect both compiled C code and assembly optimized code.

Specifying –ms0, –ms1 or –ms2 causes the compiler to increasingly favor code size over performance and could restrict performance oriented optimizations. You should experiment with the –ms*n* option to determine which speed and code size sacrifices best fit your application. In general, use –ms0 on all but the most performance critical code, and use –ms2 on control code. Using –ms*n* (–ms0, –ms1, and –ms2) may invoke other options.

- **–ms0**
  - Disallows redundant loops
- **–ms1**
  - Applies –ms0 optimizations
  - Applies inlining to intrinsic operators
- **–ms2**
  - Applies –ms1 optimizations
  - Disables software pipelining

For more help with reducing code size, see section 3.2.3, *Removing Epilogs*, on page 3-10.

3.4 Using the –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*	3-15
Want to create an optimization information file	–on*n*	3-16
Want to compile multiple source files	–pm	3-17

3.4.1 Controlling File-Level Optimization (–ol*n* Option)

When you invoke the optimizer 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

3.4.2 Creating an Optimization Information File (–on*n* Option)

When you invoke the optimizer 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

3.5 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, 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 3.4.2, *Creating an Optimization Information File (–on*n *Option)*, on page 3-16 for more information.

3.5.1 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	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	Reverts to –op0
–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 3.5.2, *Optimization Considerations When Mixing C and Assembly*, on page 3-19 for information about these situations.

3.5.2 Optimization Considerations When Mixing 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 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 functions, the –pm option optimizes out those C functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see section 7.6.6, *The FUNC_EXT_CALLED Pragma*, on page 7-18) 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 3.5.1, *Controlling Program-Level Optimization*, on page 3-17).

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 source code that calls assembly functions. Those assembly functions do not call any C functions or modify any C variables.

Solution Compile with –pm –o3 –op2 to tell the compiler that outside functions do not call C functions or modify C variables. See section 3.5.1 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 may call other C functions or modify C variables.

Situation Your application consists of C source code that calls assembly functions. The assembly language functions do not call C functions, but they modify 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 3.5.1 on page 3-17 for information about the –op*n* option.

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

Solution Add the volatile keyword to the 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 removes 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.

3.6 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 optimizer 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 optimizer behaves conservatively.

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

3.6.1 Use the –ma Option When Certain Aliases are Used

The optimizer 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;
}
```

3.6.2 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 indicates that your code does not use the aliasing technique described in section 3.6.1. 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 ANSI 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).

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.

3.6.3 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 4-26 and 4-56.

3.7 Use Caution With asm Statements in Optimized Code

You must be extremely careful when using asm (inline assembly) statements in optimized code. The optimizer 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 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 environment or access 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.

3.8 Automatic Inline Expansion (–oi Option)

The optimizer automatically inlines small functions when it is invoked with the –o3 option. A command-line option, –oi*size*, specifies the size of the functions inlined. When you use –oi, specify the *size* limit for the largest function to be inlined. You can use the –oi*size* option in the following ways:

- If you set the *size* parameter to 0 (–oi0), all size-controlled inlining is disabled.
- If you set the *size* parameter to a nonzero integer, the compiler inlines functions based on *size*. The optimizer 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 optimizer 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 –oi*size* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the –oi*size* option, the optimizer inlines very small functions. The –x option controls the inlining of functions declared as inline (see section 2.10.3.2 on page 2-37).

3.9 Using the Interlist Utility With the Optimizer

You control the output of the interlist utility when running the optimizer (the –o*n* option) with the –os and –ss options.

- The –os option interlists optimizer comments with assembly source statements.
- The –ss and –os options together interlist the optimizer comments and the original C source with the assembly code.

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

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

Example 3–2 shows the function from Example 2–2 on page 2-42 compiled with the optimizer (–o2) and the –os option. Note that the assembly file contains optimizer comments interlisted with assembly code.

Example 3–2. The Function From Example 2–2 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
           MVK     .S1     SL1+0,A0

           MVKH    .S1     SL1+0,A0
||         MVK     .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 the optimizer, the optimizer inserts its comments and the interlist utility runs between the code generator and the assembler, merging the original C source into the assembly file.

Example 3–3 shows the function from Example 2–2 on page 2-42 compiled with the optimizer (–o2) and the –ss and –os options. Note that the assembly file contains optimizer comments and C source interlisted with assembly code.

Example 3–3. The Function From Example 2–2 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
           MVK     .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVK     .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
```

3.10 Debugging and Profiling Optimized Code

Debugging and profiling fully optimized code is not recommended, because the optimizer'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. To remedy this problem, 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 it.

3.10.1 Debugging Optimized Code (–g and –o Options)

To debug optimized code, use the –g and –o options. The –g option generates symbolic debugging directives that are used by the C source-level debugger, but it disables many code generator optimizations. When you use the –o option (which invokes the optimizer) with the –g 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. Refer to section 3.2.1 on page 3-5 for more information.

3.10.2 Profiling Optimized Code (–mg, –g, and –o Options)

To profile optimized code, use the –mg option with the –g and –o options. The –mg option allows you to profile optimized code by turning on the maximum amount of optimization that is compatible with profiling. When you combine the –g option and the –o option with the –mg option, all of the line directives are removed except for the first one and the last one. The first line directive indicates the end of the prolog and the last line directive indicates the beginning of the epilog. The shaded area indicates the area between the line directives, which is the body of the function:

```
_main:
           STW     .D2     B3,*SP--                Prolog
   .line 1
           B       .S1     _initialize
           NOP             3
           MVK     .S2     RL0,B3
           MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
           B       .S1     _compute
           NOP             3
           MVK     .S2     RL1,B3                  Body of the
           MVKH    .S2     RL1,B3                  function
RL1:       ; CALL OCCURS
           B       .S1     _cleanup
           NOP             3
           MVK     .S2     RL2,B3
           MVKH    .S2     RL2,B3
RL2:       ; CALL OCCURS
   .line 6
           LDW     .D2     *++SP,B3
           NOP             4
           B       .S2     B3                      Epilog
           NOP             5
           ; BRANCH OCCURS
   .endfunc  8,000080000h,4
```

3.11 What Kind of Optimization Is Being Performed?

The TMS320C6000 C compiler uses a variety of optimization techniques to improve the execution speed of your C programs and to reduce their size. Optimization occurs at various levels throughout the compiler.

Most of the optimizations described here are performed by the separate optimizer pass that you enable and control with the –o compiler options (see section 3.1 on page 3-2). However, the code generator performs some optimizations, which you cannot selectively enable or disable.

Following are the optimizations performed by the compiler. These optimizations improve any C code:

Optimization	Page
Cost-based register allocation	3-32
Alias disambiguation	3-34
Branch optimizations and control-flow simplification	3-34
Data flow optimizations ❑ Copy propagation ❑ Common subexpression elimination ❑ Redundant assignment elimination	3-37
Expression simplification	3-37
Inline expansion of runtime-support library functions	3-38
Induction variable optimizations and strength reduction	3-39
Loop-invariant code motion	3-40
Loop rotation	3-40
Register variables	3-40
Register tracking/targeting	3-40

3.11.1 Cost-Based Register Allocation

The optimizer, when 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
```

3.11.2 Alias Disambiguation

C programs generally use many pointer variables. Frequently, compilers are unable to determine whether or not two or more lvalues (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.

3.11.3 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|
```

3.11.4 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 optimizer 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 3-35 and Example 3–6 on page 3-38).

- **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 optimizer removes these dead assignments (see Example 3–6).

3.11.5 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
```

3.11.6 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
```

3.11.7 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 3-32).

3.11.8 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.

3.11.9 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.

3.11.10 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 3-32).

3.11.11 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 3-41).

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
```

Chapter 4

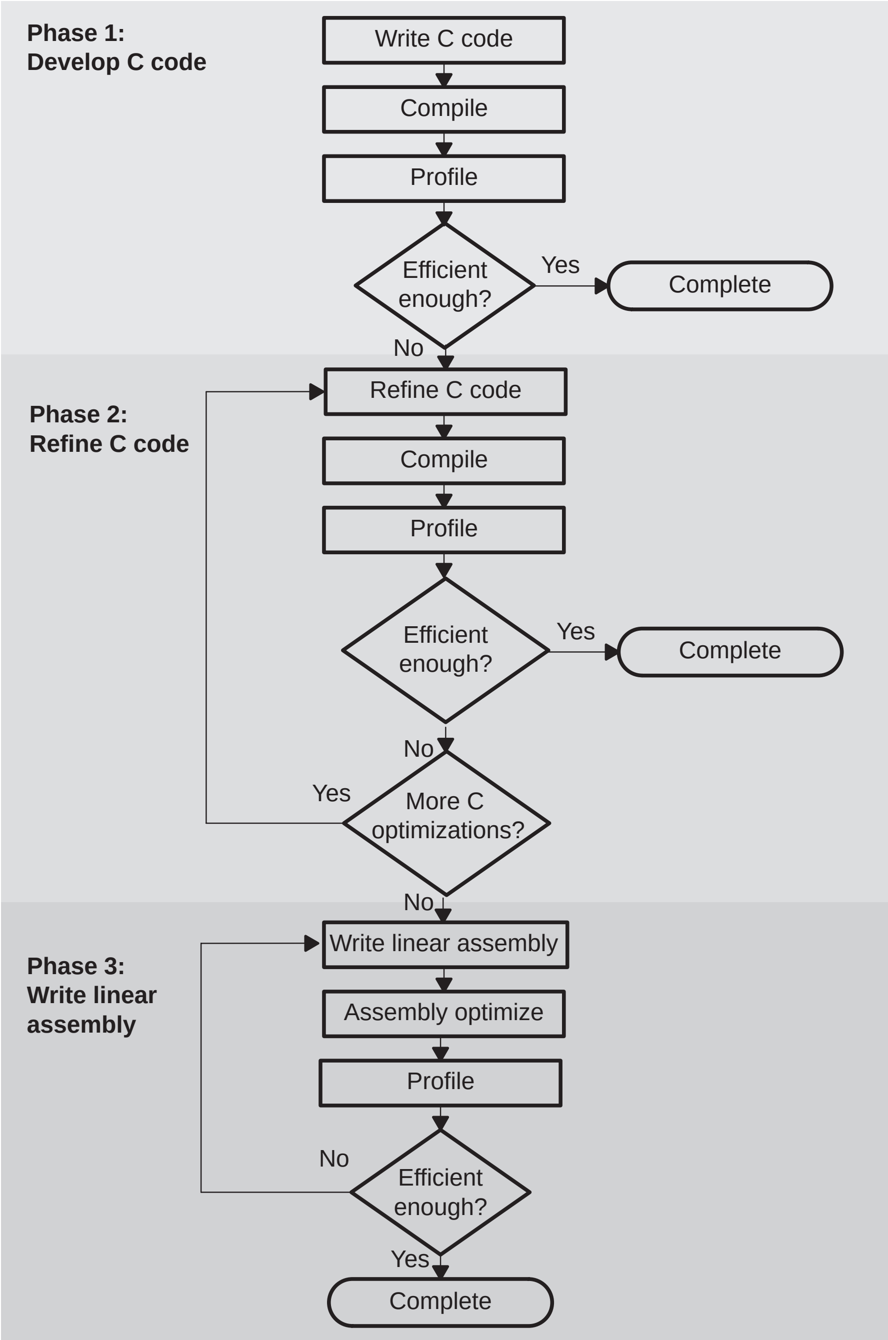
Using the Assembly Optimizer

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 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 6-7) to identify any inefficient areas in your C code. To improve the performance of your code, proceed to phase 2.

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

  In phase 2, use the intrinsics and shell options that are described in this book to improve your 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 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 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 which functional unit to use.

  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 code after you have used all of the 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 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 shell program (cl6x). The assembly optimizer is automatically invoked by the shell program if one of your input files has a *.sa* extension. You can specify C source files along with your linear assembly files. For more information about the shell 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 include information about instruction latencies or register usage. 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 4-17.

- **Options that affect what the assembly optimizer does**

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

Option	Effect	Page
–el	Changes the default extension for assembly optimizer source files	2-18
–fl	Changes how assembly optimizer source files are identified	2-18
–k	Keeps the assembly language (.asm) file	2-15
–mh*n*	Allows speculative execution	3-10
–mi*n*	Specifies an interrupt threshold value	2-41
–ms*n*	Controls code size on three levels (–ms0, –ms1, and –ms2)	3-14
–mt	Presumes no memory aliasing	3-23
–mu	Turns off software pipelining	3-5
–mv*n*	Select target version	3-12
–mw	Provides software pipelining feedback	3-5
–n	Compiles or assembly optimizes only (does not assemble)	2-16
–q	Suppresses progress messages	2-16

- **TMS320C62x 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 specify which unit should be used.

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 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. See section 4.3.2 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 @ characters to the comments to specify what iteration of the loop an instruction is on in the software pipeline. See section 4.3.3, *Using Linear Assembly Source Comments*, on page 4-14 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 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 unit specifier enables you to specify the functional unit.

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 Functional Unit Specification for Linear Assembly

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 use the unit specifier field in linear assembly:

- 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).

If you do not specify the functional unit, the assembly optimizer selects the functional unit based on the mnemonic field.

You can use the –mw shell option to display the functional unit allocation summary for a software pipelined loop. See section 3.2.2, *Software Pipelining Information (–mw Option)*, on page 3-5 for more information.

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

The following examples show how specifying functional units can be helpful in the linear assembly code.

Example 4–1 is refined C code for computing a dot product.

Example 4–1. C Code for Computing a Dot Product

```
int dotp(short a[], short b[])
{
    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 sum0 + sum1;
}
```

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

Example 4–2. Linear Assembly Code for Computing a Dot Product

```
_dotp:  .cproc          a_0, b_0

        .reg            a_4, b_4, cnt, tmp
        .reg            prod1, prod2, prod3, prod4
        .reg            valA, valB, sum0, sum1, sum

        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], valA    ; load a[0–1]
        LDW             *b_0++[2], valB    ; load b[0–1]
        MPY             valA, valB, prod1  ; a[0] * b[0]
        MPYH            valA, valB, prod2  ; a[1] * b[1]
        ADD             prod1, prod2, tmp  ; sum0 += (a[0] * b[0]) +
        ADD             tmp, sum0, sum0    ;         (a[1] * a[1])

        LDW             *a_4++[2], valA    ; load a[2–3]
        LDW             *b_4++[2], valB    ; load b[2–3]
        MPY             valA, valB, prod3  ; a[2] * b[2]
        MPYH            valA, valB, prod4  ; a[3] * b[3]
        ADD             prod3, prod4, tmp  ; sum1 += (a[0] * b[0]) +
        ADD             tmp, sum1, sum1    ;         (a[1] * a[1])

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

        ADD             sum0, sum1, sum    ; compute final result

        .return         sum

        .endproc
```

The assembly optimizer generates the software-pipeline kernel shown in Example 4–3 for the hand-coded program in Example 4–2.

Example 4–3. Software-Pipeline Kernel for Computing a Dot Product With Example 4–2

```
---------------------------------------------------------------------------*
loop:        ; PIPED LOOP KERNEL

           MV      .L2X    A0,B4        ; |1|
|| [ B0]   B       .S1     loop         ;@ |32|  if (!0) goto loop
||         MV      .L1X    B1,A7        ;@ |1|
||         LDW     .D2T2   *B9++(8),B5 ;@@ |24|  load a[2–3]
||         LDW     .D1T1   *A6++(8),A4 ;@@ |25|  load b[2–3]

           ADD     .L1X    B7,A0,A0     ; |28|  sum1 += (a[0] * b[0]) +
||         ADD     .L2     B4,B5,B4     ; |21|  sum0 += (a[0] * b[0]) +
||         MPYH    .M2X    A4,B1,B5     ;@ |20|  a[1] * b[1]
||         MPY     .M1     A4,A7,A0     ;@ |19|  a[0] * b[0]
||         LDW     .D2T2   *B6++(8),B1 ;@@@ |18|  load b[0–1]

           ADD     .L1     A0,A3,A3     ; |29|          (a[1] * a[1])
||         ADD     .L2     B4,B8,B8     ; |22|          (a[1] * a[1])
||         MPY     .M2X    B5,A4,B7     ;@ |26|  a[2] * b[2]
||         MPYH    .M1X    B5,A4,A0     ;@ |27|  a[3] * b[3]
|| [ B0]   SUB     .S2     B0,0x4,B0    ;@@ |31|  cnt –= 4
||         LDW     .D1T1   *A5++(8),A4 ;@@@ |17|  load a[0–1]
```

The kernel displayed in Example 4–3 is not the best possible kernel to use. This kernel cannot be scheduled in two cycles because the cross path (indicated by the X appended to the functional unit specifier) is repeated too many times. If you use the –mw option, the assembly optimizer automatically embeds a comment in the scheduled assembly indicating this, so that you do not have to analyze the output by hand. Example 4–4 shows the cross paths in the software pipeline information generated by the –mw option.

Example 4–4. Software Pipeline Information for Example 4–2

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : loop
;*      Known Minimum Trip Count         : 25
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 0
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 3
;*      Resource Partition:
;*                                 A–side   B–side
;*      .L units                      0        0
;*      .S units                      1        0
;*      .D units                      2        2
;*      .M units                      2        2
;*      .X cross paths                3*       3*
;*      .T address paths              2        2
;*      Long read paths               0        0
;*      Long write paths              0        0
;*      Logical  ops (.LS)            2        1     (.L or .S unit)
;*      Addition ops (.LSD)           1        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)              2        1
;*      Bound(.L .S .D .LS .LSD)      2        2
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 3  Schedule found with 4 iterations in parallel
;*      Done
;*
;*      Speculative Load Threshold : 48
;*
;*----------------------------------------------------------------------------*
```

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. In Example 4–3, the compiler partitioned two ADD instructions to sides requiring cross paths before the multiply instructions that needed the cross paths were partitioned.

You can partition enough instructions by hand to force optimal partitioning by using functional unit specifiers. If you use functional unit specifiers to force the MPYs to the sides you want them to be on, the compiler has more information about where the subsequent ADDs should go (rather, more information about where the symbolic registers involved must go). Example 4–5 shows the assembly code from after functional unit specifiers are added.

Example 4–5. Code From Example 4–2 With Functional Unit Specifiers Added

```
_dotp:  .cproc          a_0, b_0

        .reg            a_4, b_4, cnt, tmp
        .reg            prod1, prod2, prod3, prod4
        .reg            valA, valB, sum0, sum1, sum

        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], valA    ; load a[0-1]
        LDW             *b_0++[2], valB    ; load b[0-1]
        MPY     .M1     valA, valB, prod1  ; a[0] * b[0]
        MPYH    .M1     valA, valB, prod2  ; a[1] * b[1]
        ADD             prod1, prod2, tmp  ; sum0 += (a[0] * b[0]) +
        ADD             tmp, sum0, sum0    ;         (a[1] * a[1])

        LDW             *a_4++[2], valA    ; load a[2-3]
        LDW             *b_4++[2], valB    ; load b[2-3]
        MPY     .M2     valA, valB, prod3  ; a[2] * b[2]
        MPYH    .M2     valA, valB, prod4  ; a[3] * b[3]
        ADD             prod3, prod4, tmp  ; sum1 += (a[0] * b[0]) +
        ADD             tmp, sum1, sum1    ;         (a[1] * a[1])

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

        ADD             sum0, sum1, sum    ; compute final result

        .return         sum

        .endproc
```

The resulting kernel from Example 4–5 is shown in Example 4–6.

Example 4–6. Software-Pipeline Kernel for Computing a Dot Product With Example 4–5

```
;** --------------------------------------------------------------------------*
loop:        ; PIPED LOOP KERNEL

           ADD     .L1     A4,A3,A3     ; |21|  sum0 += (a[0] * b[0]) +
||         ADD     .L2     B8,B9,B9     ; |29|          (a[1] * a[1])
||         MPYH    .M1X    B5,A8,A3     ;@ |20|  a[1] * b[1]
|| [ B0]   B       .S1     loop         ;@@ |32|  if (!0) goto loop
||         MPY     .M2X    A5,B4,B6     ;@@ |26|  a[2] * b[2]
||         LDW     .D2T2   *B7++(16),B5 ;@@@@ |17|  load a[0-1]
||         LDW     .D1T1   *A7++(16),A8 ;@@@@ |18|  load b[0-1]

           ADD     .L1     A3,A0,A0     ; |22|          (a[1] * a[1])
||         ADD     .L2     B6,B8,B8     ;@ |28|  sum1 += (a[0] * b[0]) +
||         MPY     .M1X    B5,A8,A4     ;@@ |19|  a[0] * b[0]
||         MPYH    .M2X    A5,B4,B8     ;@@ |27|  a[3] * b[3]
|| [ B0]   SUB     .S2     B0,0x4,B0    ;@@@ |31|  cnt -= 4
||         LDW     .D1T1   *A6++(16),A5 ;@@@@@ |24|  load a[2-3]
||         LDW     .D2T2   *B1++(16),B4 ;@@@@@ |25|  load b[2-3]
```

4.3.3 Using Linear Assembly Source Comments

A comment in linear assembly can begin in any column and extends to the end of the source line. A comment can contain any ASCII character, including blanks. Comments 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.

The assembly optimizer attaches comments on instructions from the input linear assembly to the output file. It attaches @ (iteration delta) characters to the comments to specify the iteration of the loop that an instruction is on in the software pipeline. Zero @ characters represents the first iteration, one @ character represents the second iteration, and so on.

Example 4–7 shows code for a function called Lmac that contains comments. Example 4–8 shows the assembly optimizer output for Example 4–7.

Example 4–7. 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     .1      *a4++, t0       ; t0 = a[i]
        LDH     .2      *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:sh1

        .endproc
```

Example 4–8. Lmac Function's Assembly Optimizer Output Showing Loop Iterations, Pipelined-Loop Prolog and Kernel

```
;* BB -----------------------------------------------------
L2:         ; PIPE LOOP PROLOG

         LDH   .D1  *A4++,A3      ; t0 = a[i]
||       LDH   .D2  *B4++,B5      ; t1 = b[i]

   [ B0] ADD   .L2  –1,B0,B0      ; ––i
||       LDH   .D1  *A4++,A3      ;@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@ t1 = b[i]

   [ B0] B     .S2  L3            ; if (i) goto loop
|| [ B0] ADD   .L2  –1,B0,B0      ;@ ––i
||       LDH   .D1  *A4++,A3      ;@@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@@ t1 = b[i]

   [ B0] B     .S2  L3            ;@ if (i) goto loop
|| [ B0] ADD   .L2  –1,B0,B0      ;@@ ––i
||       LDH   .D1  *A4++,A3      ;@@@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@@@ t1 = b[i]

   [ B0] B     .S2  L3            ;@@ if (i) goto loop
|| [ B0] ADD   .L2  –1,B0,B0      ;@@@ ––i
||       LDH   .D1  *A4++,A3      ;@@@@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@@@@ t1 = b[i]

         MPY   .M1X A3,B5,A5      ; prod = t0 * t1
|| [ B0] B     .S2  L3            ;@@@ if (i) goto loop
|| [ B0] ADD   .L2  –1,B0,B0      ;@@@@ ––i
||       LDH   .D1  *A4++,A3      ;@@@@@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@@@@@ t1 = b[i]

         MPY   .M1X A3,B5,A5      ;@ prod = t0 * t1
|| [ B0] B     .S2  L3            ;@@@@ if (i) goto loop
|| [ B0] ADD   .L2  –1,B0,B0      ;@@@@@ ––i
||       LDH   .D1  *A4++,A3      ;@@@@@@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@@@@@@ t1 = b[i]

;* BB -----------------------------------------------------
L3:         ; PIPE LOOP KERNEL

         ADD   .L1  A5,A1:A0,A1:A0 ; sum += prod
||       MPY   .M1X A3,B5,A5      ;@@ prod = t0 * t1
|| [ B0] B     .S2  L3            ;@@@@@ if (i) goto loop
|| [ B0] ADD   .L2  –1,B0,B0      ;@@@@@@ ––i
||       LDH   .D1  *A4++,A3      ;@@@@@@@ t0 = a[i]
||       LDH   .D2  *B4++,B5      ;@@@@@@@ t1 = b[i]
```

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	4-18
label **.cproc** [$variable_1$ [**,** $variable_2$**,** …]]	Start a C callable procedure	Must use with .endproc	4-20
.endproc	End a C callable procedure	Must use with .cproc	4-20
.endproc [$register_1$ [**,** $register_2$**,** …]]	End a procedure	Must use with .proc; cannot use variables in the register parameter	4-30
.mdep [*symbol1*], [*symbol2*]	Indicates a memory dependence	Valid only within procedures	4-26
.mptr {*register*| *symbol*}, *base* [**+** *offset*] [**,** *stride*]	Avoid memory bank conflicts	Valid only within procedures; can use variables in the register parameter	4-27
.no_mdep	No memory aliases in the function	Valid only within procedures	4-29
label **.proc** [$register_1$ [**,** $register_2$**,** …]]	Start a procedure	Must use with .endproc; cannot use variables in the register parameter	4-30
.reg $variable_1$ [**,** $variable_2$**,**…]	Declare variables	Valid only within procedures	4-34
.return [*argument*]	Return value to procedure	Valid only within .cproc procedures	4-39
.reserve [$register_1$ [**,** $register_2$**,** …]]	Reserve register use		4-38
label **.trip** *min*	Specify trip count value	Valid only within procedures	4-40

Syntax

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

Description

Use the **.call** directive to call a function. Optionally, you may 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 compiler. For information, see section 8.3, *Register Conventions*, on page 8-15, and section 8.4, *Function Structure and Calling Conventions*, on page 8-17. 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 .def directive.
arguments	(Optional) Symbolic/machine registers passed as an argument. The arguments are passed in this order and may not 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 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 overide the calling convention. After returning from the *compute* function with the returned result in A4, a MV instruction transfers the result to A6.

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 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 callable function. The assembly optimizer performs some operations automatically in a .cproc region in order to make the function conform to the C calling conventions and to 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 specifed with the .reg directive (see page 4-34) 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 8-15 and section 8.4, *Function Structure and Calling Conventions*, on page 8-17.

Use the optional *variable* to represent function parameters. The variable entries are very similar to parameters declared in a 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 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.

- **Symbolic names**. If you specify a symbolic name, then the assembly optimizer ensures that either the symbolic name is allocated to the appropriate argument passing register or the argument passing register is copied to the register allocated for the symbolic name. For example, the first argument in a 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 'C67xx. 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 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 calling conventions is passed in B8 or B9:B8 for longs.

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 4-39 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 4-33 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     .S2X    cword,mask,cond       ; cond = codeword & mask
 [cond] MVK     .S2     1,cond                ; !(!(cond))
        CMPEQ   .L2     theta,cond,if         ; (theta == !(!(cond)))
        LDH     .D1     *a++,ai               ; a[i]
 [if]   ADD     .L1     sum,ai,sum            ; sum += a[i]
 [!if]  SUB     .D1     sum,ai,sum            ; sum –= a[i]
        SHL     .S1     mask,1,mask           ; mask = mask << 1
 [cntr] ADD     .L2     –1,cntr,cntr          ; decrement counter
 [cntr] B       .S1     LOOP                  ; for LOOP

        .return sum

        .endproc
```

This is the output from the assembly optimizer:

Runtime check to determine which version of the loop to use

Unpipelined loop body

```
;******************************************************************************
;* FUNCTION NAME: _if_then                                                    *
;*                                                                            *
;*   Regs Modified     : A0,A3,A4,A5,B0,B1,B2,B4,B5,B6,B7                     *
;*   Regs Used         : A0,A3,A4,A5,A6,B0,B1,B2,B3,B4,B5,B6,B7               *
;******************************************************************************
_if_then:
;** --------------------------------------------------------------------------*
;          .reg    cond, if, ai, sum, cntr
           MVK     .S2     0x20,B1      ; cntr = 32
           CMPGTU  .L2     B1,3,B0
   [ B0]   B       .S1     L4
           NOP             2

           ZERO    .L1     A5           ; sum = 0
||         MV      .S1     A6,A0
||         MV      .L2     B4,B5
|| [ B0]   MVC     .S2     CSR,B6
|| [ B0]   MV      .D2     B6,B4

   [ B0]   AND     .L2     –2,B6,B7

   [ B0]   MVC     .S2     B7,CSR
|| [ B0]   SUB     .L2     B1,3,B1

           ; BRANCH OCCURS
;** --------------------------------------------------------------------------*
LOOP:
   [ B1]   ADD     .L2     0xffffffff,B1,B1 ; decrement counter

   [ B1]   B       .S1     LOOP         ; for LOOP
||         LDH     .D1     *A4++,A3     ; a[i]

           NOP             1
           AND     .S2X    B5,A0,B0     ; cond = codeword & mask
   [ B0]   MVK     .S2     0x1,B0       ; !(!(cond))
           CMPEQ   .L2     B6,B0,B0     ; (theta == !(!(cond)))

   [!B0]   SUB     .D1     A5,A3,A5     ; sum –= a[i]
|| [ B0]   ADD     .L1     A5,A3,A5     ; sum += a[i]
||         SHL     .S1     A0,0x1,A0    ; mask = mask << 1

           ; BRANCH OCCURS
;** --------------------------------------------------------------------------*
           B       .S1     L9
           NOP             5
           ; BRANCH OCCURS
;** --------------------------------------------------------------------------*
```

```
L4:
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop label : LOOP
;*      Loop Carried Dependency Bound : 1
;*      Unpartitioned Resource Bound  : 2
;*      Partitioned Resource Bound(*) : 2
;*      Resource Partition:
;*                                A–side   B–side
;*      .L units                     1        2*
;*      .S units                     1        2*
;*      .D units                     2*       0
;*      .M units                     0        0
;*      .X cross paths               0        1
;*      .T address paths             1        0
;*      Long read  paths             0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          0        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             1        2*
;*      Bound(.L .S .D .LS .LSD)     2*       2*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 4 iterations in parallel
;*      Done
;*----------------------------------------------------------------------------*
L5:        ; PIPED LOOP PROLOG
   [ B1]   ADD     .L2     0xffffffff,B1,B1 ; decrement counter

           LDH     .D1     *A4++,A3     ; a[i]
|| [ B1]   B       .S1     L6           ; for LOOP

   [ B1]   ADD     .L2     0xffffffff,B1,B1 ;@ decrement counter

           AND     .S2X    B5,A0,B2     ; cond = codeword & mask
||         LDH     .D1     *A4++,A3     ;@ a[i]
|| [ B1]   B       .S1     L6           ;@ for LOOP

           SHL     .S1     A0,0x1,A0    ; mask = mask << 1
|| [ B2]   MVK     .S2     0x1,B2       ; !(!(cond))
|| [ B1]   ADD     .L2     0xffffffff,B1,B1 ;@@ decrement counter

;** --------------------------------------------------------------------------*
L6:        ; PIPED LOOP KERNEL

           CMPEQ   .L2     B4,B2,B0     ; (theta == !(!(cond)))
||         AND     .S2X    B5,A0,B2     ;@ cond = codeword & mask
||         LDH     .D1     *A4++,A3     ;@@ a[i]
|| [ B1]   B       .S1     L6           ;@@ for LOOP

   [ B0]   ADD     .L1     A5,A3,A5     ; sum += a[i]
|| [!B0]   SUB     .D1     A5,A3,A5     ; sum –= a[i]
||         SHL     .S1     A0,0x1,A0    ;@ mask = mask << 1
|| [ B2]   MVK     .S2     0x1,B2       ;@ !(!(cond))
|| [ B1]   ADD     .L2     0xffffffff,B1,B1 ;@@@ decrement counter
```

Software-pipelining information produced by the –mw option

Pipelined loop body

```
;** --------------------------------------------------------------------------*
L7:        ; PIPED LOOP EPILOG

           CMPEQ   .L2     B4,B2,B0      ;@ (theta == !(!(cond)))
||         AND     .S2X    B5,A0,B2      ;@@ cond = codeword & mask
||         LDH     .D1     *A4++,A3      ;@@@ a[i]

   [ B0]   ADD     .L1     A5,A3,A5      ;@ sum += a[i]
|| [!B0]   SUB     .D1     A5,A3,A5      ;@ sum -= a[i]
||         SHL     .S1     A0,0x1,A0     ;@@ mask = mask << 1
|| [ B2]   MVK     .S2     0x1,B2        ;@@ !(!(cond))

           CMPEQ   .L2     B4,B2,B0      ;@@ (theta == !(!(cond)))
||         AND     .S2X    B5,A0,B2      ;@@@ cond = codeword & mask

   [ B0]   ADD     .L1     A5,A3,A5      ;@@ sum += a[i]
|| [!B0]   SUB     .D1     A5,A3,A5      ;@@ sum -= a[i]
||         SHL     .S1     A0,0x1,A0     ;@@@ mask = mask << 1
|| [ B2]   MVK     .S2     0x1,B2        ;@@@ !(!(cond))

           CMPEQ   .L2     B4,B2,B0      ;@@@ (theta == !(!(cond)))

   [ B0]   ADD     .L1     A5,A3,A5      ;@@@ sum += a[i]
|| [!B0]   SUB     .D1     A5,A3,A5      ;@@@ sum -= a[i]

;** --------------------------------------------------------------------------*
           MVC     .S2     B6,CSR
;** --------------------------------------------------------------------------*
L9:
           B       .S2     B3
           NOP             4
           MV      .L1     A5,A4
           ; BRANCH OCCURS


;          .endproc
```

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”
```

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 4-45.

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
```

This is the output from the assembly optimizer:

```
;******************************************************************************
;* FUNCTION NAME: _blkcp                                                      *
;*                                                                            *
;*   Regs Modified     : A0,A3,A4,B0                                          *
;*   Regs Used         : A0,A3,A4,B0,B3                                       *
;******************************************************************************
_blkcp:
;** --------------------------------------------------------------------------*
; _blkcp:   .cproc i
;           .reg   ptr1, ptr2, tmp1, tmp2
          MVK     .S1     0x0,A3            ; |5|  ptr1 = address 0

          MVK     .S1     0x8,A0            ; |6|  ptr2 = address 8
||        MV      .L2X    A4,B0             ; |1|

; loop:     .trip  50
;           .mptr  ptr1, a+0, 4
;           .mptr  foo, a+8, 4
   [ B0]  ADD     .L2     0xffffffff,B0,B0  ; |17|  i––
;** --------------------------------------------------------------------------*
L1:

   [ B0]  B       .S1     L1                ; |18|  if (!0) goto loop
||        LDW     .D1T1   *A3++,A4          ; |14|  load  *0, bank 0


  [ B0]  ADD     .L2     0xffffffff,B0,B0  ; |17|  i––
          NOP             3
          STW     .D1T1   A4,*A0++          ; |15|  store *8, bank 0
          ; BRANCH OCCURS                   ; |18|
;** --------------------------------------------------------------------------*
          B       .S2     B3
          NOP             5
          ; BRANCH OCCURS

;           .endproc
```

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
```

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. 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 .proc directive, it is assumed that *all* of the registers referenced in the procedure are live in. 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 code is the output from the assembly optimizer. The shaded areas of the example highlight portions of the code that are affected by redundant loops. For information about redundant loops, see section 3.3 page 3-13.

```
;******************************************************************************
;* FUNCTION NAME: move                                                        *
;*                                                                            *
;*   Regs Modified     : A0,A1,A3,A4,B0,B1,B4,B5,B6                           *
;*   Regs Used         : A0,A1,A3,A4,B0,B1,B4,B5,B6                           *
;******************************************************************************
move:
;** --------------------------------------------------------------------------*
;
; move .proc  A4, B4, B0
;      .no_mdep

           ZERO    .L1     A0
||         ZERO    .L2     B5

           CMPEQ   .L2X    B5,A0,B1
||         MVK     .S2     0x18,B6
||         MV      .L1X    B0,A3

   [ B1]   CMPGTU  .L1X    A3,B6,A1
   [!B1]   CMPGTU  .L1X    B5,A0,A1
   [ A1]   B       .S1     L5
   [ A1]   SUBAW   .D2     B0,6,B0      ; |8|
           NOP             4
           ; BRANCH OCCURS
;** --------------------------------------------------------------------------*
           ADD     .L2     0xfffffffc,B0,B0 ; |7|
;** --------------------------------------------------------------------------*
L3:

   [ B0]   B       .S1     L3           ; |8|
||         LDW     .D2T2   *B4++,B1     ; |5|

   [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; |7|
           NOP             3
           STW     .D1T2   B1,*A4++     ; |6|
           ; BRANCH OCCURS ; |8|
;** --------------------------------------------------------------------------*
           B       .S1     L8           ; |8|
           NOP             5
           ; BRANCH OCCURS ; |8|
;** --------------------------------------------------------------------------*
```

```
L5:        ; PIPED LOOP PROLOG

   [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; |7|
||         LDW     .D2T2   *B4++,B1     ; |5|

   [ B0]   B       .S2     loop         ;  ^|8|
|| [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; @|7|
||         LDW     .D2T2   *B4++,B1     ; @|5|

   [ B0]   B       .S2     loop         ; @ ^|8|
|| [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; @@|7|
||         LDW     .D2T2   *B4++,B1     ; @@|5|

   [ B0]   B       .S2     loop         ; @@ ^|8|
|| [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; @@@|7|
||         LDW     .D2T2   *B4++,B1     ; @@@|5|

   [ B0]   B       .S2     loop         ; @@@ ^|8|
|| [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; @@@@|7|
||         LDW     .D2T2   *B4++,B1     ; @@@@|5|

           MV      .L1X    B1,A0        ; |5|
|| [ B0]   B       .S2     loop         ; @@@@ ^|8|
|| [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; @@@@@|7|
||         LDW     .D2T2   *B4++,B1     ; @@@@@|5|
;** --------------------------------------------------------------------------*
loop:        ; PIPED LOOP KERNEL
           STW     .D1T1   A0,*A4++     ; |6|
||         MV      .L1X    B1,A0        ; @|5|              Pipelined loop body
|| [ B0]   B       .S2     loop         ; @@@@@ ^|8|
|| [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; @@@@@@|7|
||         LDW     .D2T2   *B4++,B1     ; @@@@@@|5|
;** --------------------------------------------------------------------------*
L7:        ; PIPED LOOP EPILOG
           STW     .D1T1   A0,*A4++     ; @|6|
||         MV      .L1X    B1,A0        ; @@|5|

           STW     .D1T1   A0,*A4++     ; @@|6|
||         MV      .L1X    B1,A0        ; @@@|5|

           STW     .D1T1   A0,*A4++     ; @@@|6|
||         MV      .L1X    B1,A0        ; @@@@|5|

           STW     .D1T1   A0,*A4++     ; @@@@|6|
||         MV      .L1X    B1,A0        ; @@@@@|5|

           STW     .D1T1   A0,*A4++     ; @@@@@|6|
||         MV      .L1X    B1,A0        ; @@@@@@|5|

           STW     .D1T1   A0,*A4++     ; @@@@@@|6|
;** --------------------------------------------------------------------------*
L8:

;      .endproc
```

The following types of instructions are not allowed in .proc or .cproc (see page 4-20 and 4-30) 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:
```

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.

Objects of type long, double, or long double are allocated into an even/odd register pair and are always referenced as a register pair (for example, A1:A0). A symbolic register that is used as a register in a register pair must be defined as a register pair with the .reg directive. For example:

```
.reg ahi:alo

ADD  a0,ahi:alo,ahi:alo
```

Example 1

This example uses the same code as the block move example on page 4-30 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 the output of the following example differs from the output of the .proc example on page 4-30: symbolic registers declared with .reg are allocated as machine registers.

```
;******************************************************************************
;* FUNCTION NAME: move                                                        *
;*                                                                            *
;*   Regs Modified     : A0,A1,A3,A4,A5,B0,B1,B4,B5,B6                        *
;*   Regs Used         : A0,A1,A3,A4,A5,A6,B0,B1,B3,B4,B5,B6                  *
;******************************************************************************
move:
;** --------------------------------------------------------------------------*
;
; move .cproc dst, src, cnt
;      .no_mdep
;      .reg   tmp1, tmp2

           ZERO    .L1     A3
||         ZERO    .L2     B5

           CMPEQ   .L2X    B5,A3,B1
||         MVK     .S2     0x18,B6
||         MV      .L1     A6,A5         ; |1|

   [ B1]   CMPGTU  .L1X    A5,B6,A1
   [!B1]   CMPGTU  .L1X    B5,A3,A1
   [ A1]   B       .S1     L5
           MV      .L1X    B4,A0         ; |1|
           MV      .L2X    A6,B0         ; |1|
   [ A1]   SUBAW   .D2     B0,6,B0       ; |9|
   [ A1]   MV      .L2X    A4,B4         ; |9|
           NOP             1
           ; BRANCH OCCURS
;** --------------------------------------------------------------------------*
           ADD     .L2     0xfffffffc,B0,B0 ; |8|
;** --------------------------------------------------------------------------*
L3:

   [ B0]   B       .S1     L3            ; |9|
||         LDW     .D1T1   *A0++,A3      ; |6|

   [ B0]   ADD     .L2     0xfffffffc,B0,B0 ; |8|
           NOP             3
           STW     .D1T1   A3,*A4++      ; |7|
           ; BRANCH OCCURS ; |9|
;** --------------------------------------------------------------------------*
           B       .S1     L8            ; |9|
           NOP             5
           ; BRANCH OCCURS ; |9|
;** --------------------------------------------------------------------------*
```

```
L5:         ; PIPED LOOP PROLOG
   [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; |8|
||         LDW     .D1T1   *A0++,A3     ; |6|

   [ B0]   B       .S1     loop         ;  ^|9|
|| [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; @|8|
||         LDW     .D1T1   *A0++,A3     ; @|6|

   [ B0]   B       .S1     loop         ; @ ^|9|
|| [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; @@|8|
||         LDW     .D1T1   *A0++,A3     ; @@|6|

   [ B0]   B       .S1     loop         ; @@ ^|9|
|| [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; @@@|8|
||         LDW     .D1T1   *A0++,A3     ; @@@|6|

   [ B0]   B       .S1     loop         ; @@@ ^|9|
|| [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; @@@@|8|
||         LDW     .D1T1   *A0++,A3     ; @@@@|6|

           MV      .L2X    A3,B5        ; |1|
|| [ B0]   B       .S1     loop         ; @@@@ ^|9|
|| [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; @@@@@|8|
||         LDW     .D1T1   *A0++,A3     ; @@@@@|6|
;** --------------------------------------------------------------------------*
loop:        ; PIPED LOOP KERNEL
           STW     .D2T2   B5,*B4++     ; |7|
||         MV      .L2X    A3,B5        ; @|1|
|| [ B0]   B       .S1     loop         ; @@@@@ ^|9|
|| [ B0]   ADD     .S2     0xfffffffc,B0,B0 ; @@@@@@|8|
||         LDW     .D1T1   *A0++,A3     ; @@@@@@|6|
;** --------------------------------------------------------------------------*
L7:         ; PIPED LOOP EPILOG
           STW     .D2T2   B5,*B4++     ; @|7|
||         MV      .L2X    A3,B5        ; @@|1|

           STW     .D2T2   B5,*B4++     ; @@|7|
||         MV      .L2X    A3,B5        ; @@@|1|

           STW     .D2T2   B5,*B4++     ; @@@|7|
||         MV      .L2X    A3,B5        ; @@@@|1|

           STW     .D2T2   B5,*B4++     ; @@@@|7|
||         MV      .L2X    A3,B5        ; @@@@@|1|

           STW     .D2T2   B5,*B4++     ; @@@@@|7|
||         MV      .L2X    A3,B5        ; @@@@@@|1|

           STW     .D2T2   B5,*B4++     ; @@@@@@|7|
;** --------------------------------------------------------------------------*
L8:
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
;      .endproc
```

Pipelined loop body

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
```

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 A5, even though it is in the .reserve list, since A5 was explicitly defined in the ADD instruction:

```
   .cproc
   .reserve  a5
   .reg      tmp1

   ....
   ADD       a4, b4, a5
   ....

   .endproc
```

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 symbol tmp:

```
   .cproc ...
   .reserve a1,a2,b0,b1,b2
   .reg tmp

   ....
[tmp] ....
   ....

   .endproc
```

Syntax

.return [*argument*]

Description

The **.return** directive function is equivalent to the return statement in C code. It places the optional argument in the appropriate register for a return value as per the C calling conventions (see section 8.4 on page 8-17). Also, .return branches to the .cproc region pipelined-loop epilog.

The optional *argument* can have the folowing meanings:

- Zero arguments implies a .cproc region that has no return value, similar to a void function in C code.
- An argument implies a .cproc region that has a 32-bit return value, similar to an int function in C code.
- A register pair of the format hi:lo implies a .cproc region that has a 40-bit return value, or a 64-bit type double for ’C67xx, similar to a long function in 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
```

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 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 2-41 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 3-13.

Example 1

The .trip directive states that the loop executes somewhere between 16 and 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
```

The .sa file was compiled with –k –o2 –mh –mi40 specified. The –mi40 option says that interrupts occur at no fewer than every 40 cycles, and that the loop must be interruptible. The tools generate a 6-cycle loop to execute one iteration of the loop.

Here is the resulting assembly code:

```
FP      .set    A15
DP      .set    B14
SP      .set    B15
        .global $bss

        .sect   ".text"

;******************************************************************************
;* FUNCTION NAME: w_vecsum                                                    *
;*                                                                            *
;*   Regs Modified     : A0,A1,A2,A3,A4,A5,B4,B5,B6,B7                        *
;*   Regs Used         : A0,A1,A2,A3,A4,A5,A6,A8,B3,B4,B5,B6,B7               *
;******************************************************************************
w_vecsum:
;** --------------------------------------------------------------------------*
; w_vecsum:   .cproc  ptr_a, ptr_b, ptr_c, weight, cnt
;             .reg    ai, bi, prod, scaled_prod, ci
;             .no_mdep
           MV      .L1     A4,A5            ;            Pipelined loop body

           MVK     .S1     0x1,A2           ; init prolog collapse predicate
||         MV      .D1     A8,A1            ;
||         MV      .L1X    B6,A0            ;
||         MV      .L2X    A6,B5            ;

;** --------------------------------------------------------------------------*
C10:    ; PIPED LOOP PROLOG
; loop:       .trip 16, 48
;** --------------------------------------------------------------------------*
loop:    ; PIPED LOOP KERNEL
   [ A1]   B       .S1     loop             ; |13|
           MPY     .M1     A0,A3,A4         ; |8|
           LDH     .D1T1   *A5++,A3         ; @|6|
           SHR     .S1     A4,0xf,A4        ; |9|

           ADD     .L2X    A4,B6,B7         ; |10|
||         LDH     .D2T2   *B4++,B6         ; @|7|

   [ A2]   SUB     .S1     A2,1,A2          ;
|| [!A2]   STH     .D2T2   B7,*B5++         ; |11|
|| [ A1]   SUB     .L1     A1,0x1,A1        ; @|12|

;** --------------------------------------------------------------------------*
C11:    ; PIPED LOOP EPILOG
;** --------------------------------------------------------------------------*
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS


;             .endproc
```

Example 2

The .trip directive states that the loop will execute either 16, 24, 32, 40, or 48 times because the factor parameter is specified. When using the same options as specified in Example 1, the compiler knows that it can unroll the loop several times without affecting the results. This leads to a 7-cycle loop that executes four iterations; over a 3x speedup in terms of performance without the optional factor parameter.

```
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
```

Here is the resulting assembly code:

```
FP      .set    A15
DP      .set    B14
SP      .set    B15
        .global $bss

        .sect   ".text"
;******************************************************************************
;* FUNCTION NAME: w_vecsum                                                    *
;*                                                                            *
;*   Regs Modified     : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,B0,B4,B5,  *
;*                           B6,B7,B8,B9,SP                                   *
;*   Regs Used         : A0,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,B0,B3,B4,  *
;*                           B5,B6,B7,B8,B9,SP                                *
;******************************************************************************
w_vecsum:
;** --------------------------------------------------------------------------*
; w_vecsum:   .cproc  ptr_a, ptr_b, ptr_c, weight, cnt
;             .reg    ai, bi, prod, scaled_prod, ci
;             .no_mdep
           STW     .D2T1   A12,*SP--(16)     ; |1|

           STW     .D2T1   A10,*+SP(8)       ; |1|
||         ADD     .L1X    2,B4,A5           ;
||         MV      .D1     A4,A9             ;
||         MVK     .S1     0x1,A2            ; init prolog collapse predicate

           MV      .L2     B4,B6             ;
||         MV      .L1X    B6,A10            ;
||         STW     .D2T1   A11,*+SP(12)      ; |1|
||         ADD     .S1     4,A8,A1           ;
||         ADD     .D1     2,A6,A0           ;
||         MV      .S2X    A6,B8             ;
```

```
;** ---------------------------------------------------------------------------*
C10:    ; PIPED LOOP PROLOG
; loop:       .trip 16, 48, 8
;** ---------------------------------------------------------------------------*
loop:    ; PIPED LOOP KERNEL

   [ A1]   SUB     .L1     A1,0x4,A1          ; |12|
|| [!A2]   LDH     .D2T2   *–B6(2),B7         ; |7|
||         MPY     .M1     A10,A6,A3          ; |8|

   [ A1]   B       .S1     loop               ; |13|
||         MPY     .M2X    A10,B0,B4          ; |8|
||         LDH     .D1T1   *A9++(8),A12       ; @|6|

           MV      .L2     B9,B5              ; Inserted to split a long life
||         SHR     .S1     A4,0xf,A4          ; |9|
||         MPY     .M1     A10,A8,A8          ; |8|
||         LDH     .D2T2   *B6++(8),B9        ; @|7|
||         LDH     .D1T1   *–A9(6),A6         ; @|6|

           MV      .L1     A11,A4             ; Inserted to split a long life
||         SHR     .S1     A3,0xf,A3          ; |9|
||         ADD     .L2X    A4,B5,B5           ; |10|
||         SHR     .S2     B4,0xf,B4          ; |9|
||         LDH     .D2T1   *–B6(4),A11        ; @|7|
||         LDH     .D1T2   *–A9(4),B0         ; @|6|

           ADD     .L1     A3,A7,A3           ; |10|        Pipelined loop body
|| [!A2]   STH     .D2T2   B5,*B8++(8)        ; |11|
||         SHR     .S1     A8,0xf,A4          ; |9|
||         ADD     .L2X    B4,A4,B4           ; |10|
||         LDH     .D1T1   *–A9(2),A8         ; @|6|

   [!A2]   STH     .D1T1   A3,*A0++(8)        ; |11|
|| [!A2]   STH     .D2T2   B4,*–B8(4)         ; |11|
||         ADD     .L2X    A4,B7,B4           ; |10|

   [ A2]   SUB     .L1     A2,1,A2            ;
|| [!A2]   STH     .D2T2   B4,*–B8(2)         ; |11|
||         LDH     .D1T1   *A5++(8),A7        ; @|7|
||         MPY     .M1     A10,A12,A4         ; @|8|
;** ---------------------------------------------------------------------------*
C11:    ; PIPED LOOP EPILOG
;** ---------------------------------------------------------------------------*
           LDW     .D2T1   *+SP(8),A10

           B       .S2     B3
||         LDW     .D2T1   *+SP(12),A11

           LDW     .D2T1   *++SP(16),A12
           NOP             4
           ; BRANCH OCCURS


;             .endproc
```

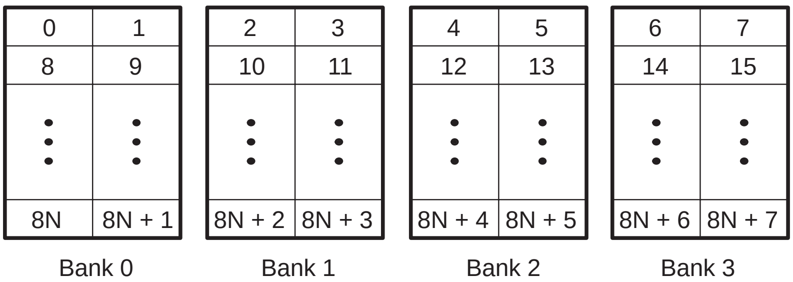
4.5 Avoiding Memory Bank Conflicts With the Assembly Optimizer

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.

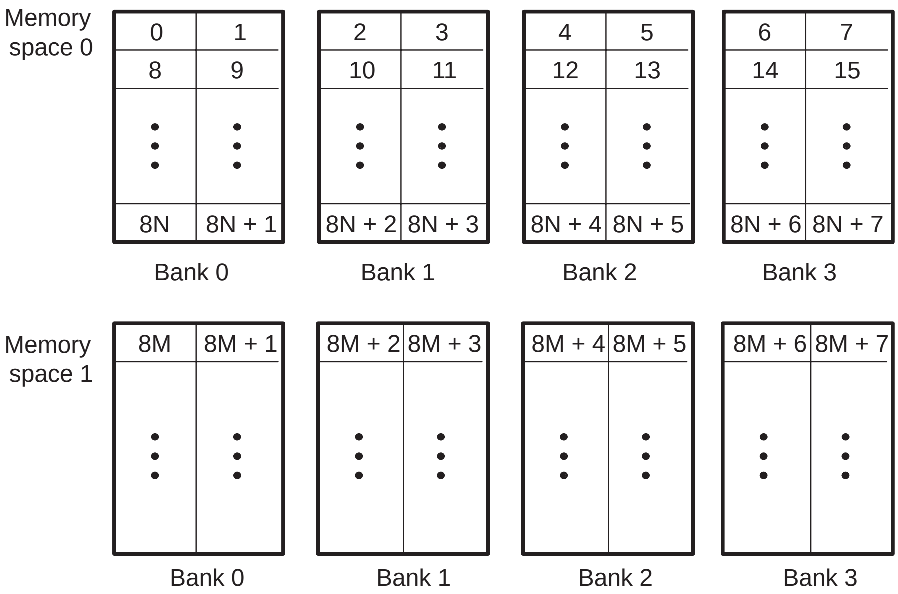
Because each bank is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle while the second value is read from memory. Two memory operations per cycle are allowed 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 ’C67x). 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 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 4-27 for information about the .mptr directive.

Example 4–9 shows loads and stores extracted from a loop that is being software pipelined.

Example 4–9. 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     .D1     *Ain––[2],Ain12            ; IN(k–i) & IN(k–i+1)
        LDW     .D2     *Bin––[2],Bin23            ; IN(k–i–2) & IN(k–i–1)
        LDW     .D1     *Ain––[2],Ain34            ; IN(k–i–4) & IN(k–i–3)
        LDW     .D2     *Bin––[2],Bin56            ; IN(k–i–6) & IN(k–i–5)

        LDW     .D2     *Bco++[2],Bco12            ; COEF(i) & COEF(i+1)
        LDW     .D1     *Aco++[2],Aco23            ; COEF(i+2) & COEF(i+3)
        LDW     .D2     *Bco++[2],Bin34            ; COEF(i+4) & COEF(i+5)
        LDW     .D1     *Aco++[2],Ain56            ; COEF(i+6) & COEF(i+7)

        STH     .D1     Assum,*Aout++[2]           ; *oPtr++ = (r >> 15)
        STH     .D2     Bssum,*Bout++[2]           ; *oPtr++ = (i >> 15)
```

4.5.2 A Dot Product Example That Avoids Memory Bank Conflicts

The C code in Example 4–10 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 4–11 implement the dotp loop kernel. Example 4–12 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 4–10. 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 4–11. 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 4–12. Dot Product Software-Pipelined Kernel

```
L3:        ; PIPE LOOP KERNEL

         ADD   .L2  B4,B6,B6     ; sum0 += a[0] * b[0]
||       ADD   .L1  A5,A0,A0     ; sum1 += a[1] * b[1]
||       MPY   .M2X A3,B5,B4     ;@@ a[0] * b[0]
||       MPYH  .M1X A3,B5,A5     ;@@ a[1] * b[1]
|| [ B0] B     .S1  L3           ;@@@@@ if (!i) goto loop
|| [ B0] ADD   .S2  –1,B0,B0     ;@@@@@@ i––
||       LDW   .D1  *A4++,A3     ;@@@@@@@ load a[0–1]  bank0
||       LDW   .D2  *B4++,B5     ;@@@@@@@ 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 4–13 shows the unrolled loop kernel.

Example 4–13. Dot Product From Example 4–11 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,A4,a_4
        ADD     4,B4,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 50
        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 4–14. Unrolled Dot Product Kernel From Example 4–12

```
L3:         ; PIPE LOOP KERNEL

         ADD   .L2  B6,B9,B9     ; sum0 += a[0] * b[0]
||       ADD   .L1  A6,A0,A0     ; sum1 += a[1] * b[1]
||       MPY   .M2X B5,A4,B6     ;@ a[0] * b[0]
||       MPYH  .M1X B5,A4,A6     ;@ a[1] * b[1]
|| [ B0] B     .S1  L3           ;@@ if (!0) goto loop
||       LDW   .D1  *A3++(8),A4  ;@@@ load a[2–3]  bankx+2
||       LDW   .D2  *B4++(8),B5  ;@@@@ load a[0–1]  bankx

         ADD   .L2  B6,B9,B9     ; sum0 += a[2] * b[2]
||       ADD   .L1  A6,A0,A0     ; sum1 += a[3] * b[3]
||       MPY   .M2X A4,B8,B6     ;@ a[2] * b[2]
||       MPYH  .M1X A4,B8,A6     ;@ a[3] * b[3]
|| [ B0] ADD   .S2  –1,B0,B0     ;@@@ i––
||       LDW   .D2  *B7++(8),B8  ;@@@@ load b[2–3]  banky+2
||       LDW   .D1  *A5++(8),A4  ;@@@@ load b[0–1]  banky
```

Without the .mptr directives in Example 4–13, 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 4–14 can never have a memory bank conflict.

In Example 4–11, 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 4–15, the .mptr directive is used to specify which memory accesses conflict and which never conflict.

Example 4–15. 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 3.6.2, page 3-22.

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 4–16. 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 interation n.

Example 4–17 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 4–17. 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 4–18.

Example 4–18. 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 Code

The 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 by using cl6x. 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 code, including the runtime-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 as an Individual Program

This section shows how to invoke the linker in a separate step after you have compiled and assembled your programs. This is the general syntax for linking C programs in a separate step:

lnk6x {**–c**|**–cr**} *filenames* [*-options*] [**–o** *name.out*] **–l** *libraryname* [*lnk.cmd*]

lnk6x The command that invokes the linker.

–c | –cr Options that tell the linker to use special conventions defined by the C environment. When you use lnk6x, you must use –c or –cr. The –c option uses automatic variable initialization at runtime; 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. Options can appear anywhere on the command line or in a linker command file. (Options are discussed in section 5.4)

–o *name.out* The –o option names the output file.

–l *libraryname* (lowercase L) Identifies the appropriate archive library containing C runtime-support and floating-point math functions. (The –l option tells the linker that a file is an archive library.) If you are linking C code, you must use a runtime-support library. You can use the libraries included with the compiler, or you can create your own runtime-support library. If you have specified a runtime-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. For example, you can link a C program consisting of modules prog1, prog2, and prog3 (the output file is named prog.out), enter:

```
lnk6x –c prog1 prog2 prog3 –o prog.out –l rts6201.lib
```

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 more information, see the *TMS320C6000 Assembly Language Tools User's Guide*.

5.2 Invoking the Linker With the Compiler Shell (–z Option)

The options and parameters discussed in this section apply to both methods of linking; however, when you link while compiling, the linker options must follow the –z option (see section 2.2, *Invoking the C Compiler Shell,* on page 2-4).

By default, the compiler does not run the linker. However, if you use the –z option, a program is compiled, assembled, and linked in one step. When using –z to enable linking, remember that:

- The –z option divides the command line into compiler options (the options before –z) and linker options (the options following –z).
- The –z option must follow all source files and other compiler options on the command line or be specified with the C_OPTION environment variable.

All arguments that follow –z on the command line are passed on to the linker. These arguments can be linker command files, additional object files, linker options, or libraries. For example, to compile and link all the .c files in a directory, enter:

```
cl6x –sq *.c –z c.cmd –o prog.out –l rts6201.lib
```

First, all of the files in the current directory that have a .c extension are compiled using the –s (interlist C and assembly code) and –q (run in quiet mode) options. Second, the linker links the resulting object files by using the c.cmd command file. The –o option names the output file, and the –l option names the runtime-support library.

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 environment variable

5.3 Disabling the Linker (–c Shell Option)

You can override the –z option by using the –c shell option. The –c option is especially helpful if you specify the –z option in the 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 shell option. By default, the compiler uses the –c linker option when you use the –z option. This tells the linker to use C linking conventions (autoinitialization of variables at runtime). If you want to initialize variables at load time, use the –cr linker option following the –z option.

5.4 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.
–ar	Produces a relocatable, executable object module
–b	Disables merge of symbolic debugging information
–c	Autoinitializes variables at runtime. See section 8.8.3 on page 8-40, for more information.
–cr	Initializes variables at load time. See section 8.8.4 on page 8-41, 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
–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 eight –i options.
–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.

–n	Ignores all fill specifications in memory directives. Use this option in the development stage of a project to avoid generating large .out files, which can result from using memory directive fill specifications.
–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)
–r	Retains relocation entries in the output module
–s	Strips symbol table information and line number entries from the output module.
–stack *size*	Sets the C system stack size to *size* bytes and defines a global symbol that specifies the stack size. The default is 1K bytes.
–u *symbol*	Places the unresolved external symbol *symbol* into the output module's symbol table
–w	Displays a message when an undefined output section is created
–x	Forces rereading of libraries. Resolves back references

For more information on linker options, see the *Linker Description* chapter in the *TMS320C6000 Assembly Language Tools User's Guide.*

5.5 Controlling the Linking Process

Regardless of the method you choose for invoking the linker, special requirements apply when linking C programs. You must:

- ❑ Include the compiler's runtime-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.*

5.5.1 Linking With Runtime-Support Libraries

You must link all C programs with a runtime-support library. The library contains standard C functions as well as functions used by the compiler to manage the C environment. You must use the –l linker option to specify which 'C6000 runtime-support library to use. The –l option also tells the linker to look at the –i options and then the C_DIR environment variable to find an archive path or object file. To use the –l linker option, type on the command line:

lnk6x {**–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 linker 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.

The 'C6000 libraries are rts6201.lib and rts6701.lib, for use with little-endian code, and rts6201e.lib and rts6701e.lib, for use with big-endian code.

You must link all C programs with an object module called *boot.obj*. When a C program begins running, it must execute boot.obj first. The boot.obj file contains code and data to initialize the runtime environment; the linker automatically extracts boot.obj and links it when you use –c or –cr and include rts6201.lib or rts6201e.lib, and either rts6701.lib or rts6701e.lib in the link.

Note: The _c_int00 Symbol

One important function contained in the runtime 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.

The boot.obj module contains code and data for initializing the runtime environment. The module performs the following tasks:

1) Sets up the stack
2) Processes the runtime initialization table and autoinitializes global variables (when using the –c option)
3) Calls main
4) Calls exit when main returns

Chapter 9 describes additional runtime-support functions that are included in the library. These functions include ANSI C standard runtime support.

5.5.2 Specifying the Type of Initialization

The C compiler produces data tables for initializing global variables. Section 8.8.2, *Initialization Tables*, on page 8-37 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 *runtime*. Use the –c linker option (see section 8.8.3, *Autoinitialization of Variables at Runtime*, on page 8-40).
- Global variables are initialized at *load time*. Use the –cr linker option (see section 8.8.4, *Initialization of Variables at Load time*, on page 8-41).

When you link a 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 5.2, *Invoking the Linker With the Compiler Shell*, on page 5-4). 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 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 runtime-support library.

- The *.cinit* output section is padded with a termination record so that the loader (load time initialization) or the boot routine (runtime initialization) knows when to stop reading the initialization tables.

- When using initializing 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 runtime.

  - 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.

5.5.3 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
.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 8-3 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*.

5.5.4 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 runtime.
–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, rts6201.lib, for input.

To link the program, use the following syntax:

lnk6x *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 rts6201.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
    .tables     >       BMEM
    .data       >       BMEM
    .stack      >       BMEM
    .bss        >       BMEM
    .sysmem     >       BMEM
    .cinit      >       BMEM
    .const      >       BMEM
    .cio        >       BMEM
    .far        >       BMEM
}
```

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 is useful for quick simulation of small pieces of code; specifically, to gather cycle count information. It is faster for iterative code changes than using the TMS320C6000 debugger.

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 both ’C62xx and ’C67xx files. No options are needed to specify either a floating-point or fixed-point .out file. The stand-alone simulator determines the version (’C62xx or ’C67xx) 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) 1989–1999 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 clock.out invoked with the –a option

```
load6x clock.out
TMS320C6x Standalone Simulator Version X.X
Copyright (c) 1989–1999 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 memory hierarchy modelling which counts data memory bank conflicts, external memory access stalls, and model program cache on both ’C62x and ’C67x models. See section 6.3, *Selecting Memory Hierarchy Modelling (–a Option)*, on page 6-6 for more information.

–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 the –mg profiling option 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 6-7 for more information.

–h Prints the list of available options for the stand-alone simulator.

–o *xxx* Sets overall timeout to *xxx* minutes. The stand-alone simulator aborts if the loaded program is not finished after *xxx* minutes.

–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.

–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 chaper 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.

–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 Selecting Memory Hierarchy Modeling (–a Option)

The stand-alone simulator does not by default count memory bank conflicts, external memory access stalls, or model program cache on both ’C62x and ’C67x models. To enable these for more accurate chip simulation, use the –a option.

When the –a option is used and the –q option is not used, the load6x banner shows that the memory hierarchy modelling option has been selected. The clock cycles provided by load6x may not be accurate in all cases involving external memory accesses, but can be considered as indicative of the silicon behavior. Enabling memory modeling causes the stand-alone simulator (load6x) to run much slower.

The amount of time required to perform external memory accesses is determined by the values in the EMIF registers. By default, the stand-alone simulator uses best case (i.e. fastest) values for all external memory spaces. The default EMIF values are:

- CE0 Space Control Register = 0x00000040 (32-bit wide SBSRAM)
- CE1 Space Control Register = 0x00000020 (32-bit wide asynchronous interface)
- CE2 Space Control Register = 0x00000040 (32-bit wide SBSRAM)
- CE3 Space Control Register = 0x00000040 (32-bit wide SBSRAM)

These EMIF values can be changed from within a .out file code to select different external memories and to modify access time values that more accurately reflect your system requirements. In a C code application, it is recommended that this be done as early as possible to ensure the most accurate simulation possible.

Note: Cannot Change Memory Type

Code running in external memory cannot change the memory type of the external memory space it is executing in.

Note: Fast Validation

Not using the –a option on a ’C62x/C67x .out file enables the fast version of the stand-alone simulator. This allows for very fast validation of an application without the use of hardware.

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 -mg profiling option for profiling to work on the stand-alone simulator (see section 3.10.2, *Profiling Optimized Code (–mg, –g, and –o Options)*, on page 3-30.) The profile results resemble the results given by the profiler in the TI simulator debugger. The profile results are stored in a 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 6–2 runs three different versions of the dot product routines and prints out the result of each routine.

Example 6–2. Profiling Dot Product Routines

```
load6x –q –g t.out
val = 11480
val = 11480
val = 11480

<t.vaa>
Program Name:   /c6xcode/t.out
Start Address:  0000554c main, at line 32, "/c6xcode/t.c"
Stop Address:   000073a0        exit
Run Cycles:     10543
Profile Cycles: 10543
BP Hits:        18



*****************************************************************************
 Area Name                 Count  Inclusive   Incl–Max  Exclusive
Excl–Max
CF dot_prod1()                  1         60         60
60        60
CF dot_prod2()                  1         55         55
55        55
CF dot_prod3()                  1         35         35
35        35
CF main()                       1      10537      10537
134       134
```

Example 6–2. Profiling Dot Product Routines (Continued)

```
****************************************************************************
 Area Name                Count
CF dot_prod1()                  1  25%
=====================================
CF dot_prod2()                  1  25%
=====================================
CF dot_prod3()                  1  25%
=====================================
CF main()                       1  25%
=====================================

****************************************************************************
 Area Name            Inclusive
CF main()                   10537  99%
=====================================
CF dot_prod1()                 60  <1%
CF dot_prod2()                 55  <1%
CF dot_prod3()                 35  <1%

****************************************************************************
 Area Name             Incl–Max
CF main()                   10537  99%
=====================================
CF dot_prod1()                 60  <1%
CF dot_prod2()                 55  <1%
CF dot_prod3()                 35  <1%

****************************************************************************
 Area Name            Exclusive
CF main()                     134   1%
=====================================
CF dot_prod1()                 60  <1% ================
CF dot_prod2()                 55  <1% ===============
CF dot_prod3()                 35  <1% =========

****************************************************************************
 Area Name             Excl–Max
CF main()                     134   1%
=====================================
CF dot_prod1()                 60  <1% ================
CF dot_prod2()                 55  <1% ===============
CF dot_prod3()                 35  <1% =========

****************************************************************************
 Area Name              Address
CF dot_prod1()           000052c0
CF dot_prod2()           000053a4
CF dot_prod3()           00005444
CF main()                0000554c
```

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 simualtor 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 4-46.

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 6–3 shows an example of the C code for accomplishing this.

Example 6–3. 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 6–3, 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 6–4. Stand-Alone Simulator Results After Compiling and Linking Example 6–3

```
TMS320C6x Standalone Simulator   Version x.xx
Copyright (c) 1989–1997 Texas Instruments Incorporated
Interrupt to abort . . .
Hello, world
Time = 3338 cycles
NORMAL COMPLETION: 11692 cycles
```

Chapter 7

TMS320C6000 C Language Implementation

The TMS320C6000 C compiler supports the C language standard that was developed by a committee of the American National Standards Institute (ANSI) to standardize the C programming language.

ANSI C supersedes the de facto C standard that is described in the first edition of *The C Programming Language* by Kernighan and Ritchie. The ANSI 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 ANSI standard. ANSI C encompasses many of the language extensions provided by current C compilers and formalizes many previously unspecified characteristics of the language.

Topic **Page**

7.1 Characteristics of TMS320C6000 C

The ANSI standard identifies certain features of the C language that are affected by characteristics of the target processor, runtime environment, or host environment. For efficiency or practicality, these characteristics can differ among standard compilers. This section describes how these characteristics are implemented for the ’C6000 C compiler.

The following list identifies all such cases and describes the behavior of the ’C6000 C compiler in each case. Each description also includes a reference to more information. Many of the references are to the formal ANSI standard or to the second edition of *The C Programming Language* by Kernighan and Ritchie (K&R).

7.1.1 Identifiers and Constants

- The first 100 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.
(ANSI 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.
(ANSI 2.2.1, K&R A12.1)

- Hex or octal escape sequences in character or string constants may have values up to 32 bits. (ANSI 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’` (ANSI 3.1.3.4, K&R A2.5.2)

7.1.2 Data Types

- For information about the representation of data types, see section 7.2 on page 7-5. (ANSI 3.1.2.5, K&R A4.2)

- The type size_t, which is the result of the *sizeof* operator, is unsigned int.
(ANSI 3.3.3.4, K&R A7.4.8)

- The type ptrdiff_t, which is the result of pointer subtraction, is int.
(ANSI 3.3.6, K&R A7.7)

7.1.3 Conversions

- Float-to-integer conversions truncate toward 0. (ANSI 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted. (ANSI 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
  ```
  (ANSI 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. (ANSI 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 7.4, *Register Variables*, on page 7-12. (ANSI 3.5.1, K&R A2.1)
- Structure members are packed into words. (ANSI 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 8-13. (ANSI 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 7.3.3 on page 7-8.

7.1.6 Preprocessor

- The preprocessor ignores any unsupported #pragma directive.
  (ANSI 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
  - NMI_INTERRUPT
  - STRUCT_ALIGN

  For more information on pragmas, see section 7.6 on page 7-14.

7.2 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 9-16.

Table 7–1. TMS320C6000 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
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	32 bits	Binary	0	0xFFFFFFFF

† Figures are minimum precision.

7.3 Keywords

The ’C6000 C compiler supports the standard const, register, and volatile keywords. In addition, the ’C6000 C compiler extends the C language through the support of the cregister, interrupt, near, and far keywords.

7.3.1 The const Keyword

The TMS320C6000 C compiler supports the ANSI 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 is auto (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};
```

7.3.2 The cregister Keyword

The ’C6000 compiler extends the 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	(’C67x only) FP ADD control register
FAUCR	(’C67x only) FP AUX control register
FMCR	(’C67x only) FP MULT control 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);
}
```

7.3.3 The interrupt Keyword

The ’C6000 compiler extends the 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 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 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.

7.3.4 The near and far Keywords

The ’C6000 C compiler extends the 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. Two storage class modifiers cannot be used together in a single declaration. For example:

```
far static int x;
static near int x;
static int far x;
far int foo();
static far int foo();
```

7.3.4.1 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:

```
mvk   _address,a1
mvkh  _address,a1
ldw   *a1,a0
```

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. This ensures access to the variable, since the variable might not be in the .bss section. For details, see section 7.6.4, DATA_SECTION pragma, on page 7-17.

7.3.4.2 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.

```
mvk   _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.

7.3.4.3 Controlling How Runtime-Support Functions Are Called (–mr Option)

The –mr*n* option controls how runtime-support functions are called:

–mr0 Runtime-support data and calls are near

–ml1 Runtime-support data and calls are far

By default, runtime-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 runtime-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 runtime-support functions to be far, regardless of the setting of the –ml option.

The –mr option only addresses how runtime-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.

7.3.4.4 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.

7.3.5 The volatile Keyword

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the 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.

7.4 Register Variables

The TMS320C6000 C compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the optimizer.

- **Compiling with the optimizer**

  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 the optimizer**

  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 8-15.

7.5 The asm Statement

The TMS320C6000 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 language—the *asm* statement. The asm statement provides access to hardware features that 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 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 Environment With asm Statements

Be careful not to disrupt the C environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into 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 the optimizer with asm statements. Although the optimizer cannot remove asm statements, it can significantly rearrange the code order near them and cause undesired results.

7.6 Pragma Directives

Pragma directives tell the compiler's preprocessor how to treat functions. The 'C6000 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
- NMI_INTERRUPT
- STRUCT_ALIGN

Some of these pragmas use the arguments *func* and *symbol*. With the exception of the DATA_MEM_BANK pragma, these arguments must have file scope; that is, you cannot define or declare them inside the body of a function. You must specify the pragma 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 follow these rules, the compiler issues a warning.

7.6.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in a section named *section name*. The syntax of the pragma is:

#pragma CODE_SECTION (*symbol*, *"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 7–2 demonstrates the use of the CODE_SECTION pragma.

Example 7–2. Using the CODE_SECTION Pragma

(a) C source file

```
#pragma CODE_SECTION(fn, ”my_sect”)

int fn(int x)
{
   return c;
}
```

(b) Assembly source file

```
    .file  ”CODEN.c”
    .sect  ”my_sect”
    .global   _fn
    .sym   _fn,_fn,36,2,0
    .func  3
```

7.6.2 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 is:

#pragma DATA_ALIGN (*symbol***,** *constant***);**

7.6.3 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. The value of *constant* can be 0–3 for 'C62xx (for data memory banks 0, 1, 2, 3 on the current 'C62xx parts) or 0–7 for 'C67xx (for data banks 0–7 on the current 'C67xx parts). See Figure 4–1 on page 4-45 for a graphic representation of memory banks.

The syntax of this pragma is:

#pragma DATA_MEM_BANK (*symbol***,** *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 ‘C62xx, the code in Example 7–3 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 2 or a. 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 6 or e.

Example 7–3. Using the DATA_MEM_BANK Pragma

```
#pragma DATA_MEM_BANK (x, 2);
short x[100];

#pragma DATA_MEM_BANK (z, 3);
#pragma DATA_SECTION (z, ”.z_sect”);
short z[100];

void main()
{
    #pragma DATA_MEM_BANK (y, 1);
    short y[100];
    ...
}
```

7.6.4 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section named *section name*. The syntax of the pragma is:

#pragma DATA_SECTION (*symbol*, "*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.

Example 7–4 demonstrates the use of the DATA_SECTION pragma.

Example 7–4. Using the DATA_SECTION Pragma

(a) C source file

```
#pragma DATA_SECTION(bufferB, ”my_sect”)
char bufferA[512];
char bufferB[512];
```

(b) Assembly source file

```
        .global _bufferA
        .bss    _bufferA,512,4
        .global _bufferB
_bufferB:       .usect  ”my_sect”,512,4
```

7.6.5 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.

The pragma must appear before any declaration or reference to the function that you want to keep. The syntax of the pragma is:

#pragma FUNC_CANNOT_INLINE (*func***);**

The argument *func* is the name of the C function that cannot be inlined. For more information, see section 2.10, *Using Inline Function Expansion*, on page 2-35.

7.6.6 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 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 functions call. These functions act as entry points into C.

The pragma must appear before any declaration or reference to the function that you want to keep. The syntax of the pragma is:

#pragma FUNC_EXT_CALLED (*func***);**

The argument *func* is the name of the C function that you do not want removed.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See section 3.5.2, *Optimization Considerations When Mixing C and Assembly*, on page 3-19.

7.6.7 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 2-41). 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 is:

#pragma FUNC_INTERRUPT_THRESHOLD (*func*, *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.

7.6.8 The FUNC_IS_PURE Pragma

The FUNC_IS_PURE pragma specifies to the optimizer that the named function has no side effects. This allows the optimizer 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 is:

#pragma FUNC_IS_PURE (*func***);**

The argument *func* is the name of a C function.

7.6.9 The FUNC_IS_SYSTEM Pragma

The FUNC_IS_SYSTEM pragma specifies to the optimizer that the named function has the behavior defined by the ANSI 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 is:

#pragma FUNC_IS_SYSTEM (*func***);**

The argument *func* is the name of the C function to treat as an ANSI standard function.

7.6.10 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the optimizer 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 is:

#pragma FUNC_NEVER_RETURNS (*func***);**

The argument *func* is the name of the C function that does not return.

7.6.11 The FUNC_NO_GLOBAL_ASG Pragma

The FUNC_NO_GLOBAL_ASG pragma specifies to the optimizer 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 is:

#pragma FUNC_NO_GLOBAL_ASG (*func***);**

The argument *func* is the name of the C function that makes no assignments.

7.6.12 The FUNC_NO_IND_ASG Pragma

The FUNC_NO_IND_ASG pragma specifies to the optimizer 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 is:

#pragma FUNC_NO_IND_ASG (*func***);**

The argument *func* is the name of the C function that makes no assignments.

7.6.13 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. The argument *func* is the name of a function. The pragma syntax is:

#pragma INTERRUPT (*func***);**

The code for *func* 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.

7.6.14 The NMI_INTERRUPT Pragma

The NMI_INTERRUPT pragma enables you to handle non-maskable interrupts directly with C code. The argument *func* is the name of a function. The pragma syntax is:

#pragma NMI_INTERRUPT (*func***);**

The code generated for `func` 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 (the `func` argument) does not need to conform to a naming convention.

7.6.15 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 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, it's base type must be either a structure tag or a union tag.

Since ANSI 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 Initializing Static and Global Variables

The ANSI 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.

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.*

7.8 Changing the ANSI C Language Mode

The –pk, –pr, and –ps options let you specify how the C compiler interprets your source code. You can compile your source code in the following modes:

- Normal ANSI mode
- K&R C mode
- Relaxed ANSI mode
- Strict ANSI mode

The default is normal ANSI mode. Under normal ANSI mode, most ANSI violations are emitted as errors. Strict ANSI violations (those idioms and allowances commonly accepted by C compilers, although violations with a strict interpretation of ANSI), however, are emitted as warnings. Language extensions, even those that conflict with ANSI C, are enabled.

7.8.1 Compatibility With K&R C (–pk Option)

The ANSI 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-ANSI 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 ANSI 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 ANSI 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 ANSI 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 ANSI rules without revoking any of the features.

The specific differences between the ANSI 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 ANSI, 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 */
  ```

- ANSI 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 */
  ```

  Even without –pk, a violation of this rule is a code-E (recoverable) error. You can use –pe, which converts code-E errors to warnings, as an alternative to –pk.

- External declarations with no type or storage class (only an identifier) are illegal in ANSI but legal in K&R:

  ```
  a;                  /* illegal unless –pk used */
  ```

- ANSI 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 ANSI, 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.

- ANSI 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 ANSI but ignored under K&R:

  ```
  char c = '\q';      /* same as 'q' if –pk used, error */
                      /* if not */
  ```

- ANSI 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 */
  };
  ```

  The 'C6000 C compiler operates on bit fields defined as unsigned ints. Signed int bit field definitions are prohibited.

- 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 */
  ```

7.8.2 Enabling Strict ANSI Mode and Relaxed ANSI Mode (–ps and –pr Options)

Use the –ps option when you want to compile under strict ANSI mode. In this mode, error messages are provided when non-ANSI 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 ANSI violations rather than emit a warning (as occurs in normal ANSI mode) or an error message (as occurs in strict ANSI mode). In relaxed ANSI mode, the compiler accepts extensions to the ANSI C standard, even when they conflict with ANSI C.

Chapter 8

Runtime Environment

This chapter describes the TMS320C6000 C runtime environment. To ensure successful execution of C programs, it is critical that all runtime 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 code.

Topic **Page**

8.1 Memory Model

The C 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 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 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 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 runtime 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 program does not use these functions, the compiler does not create the .sysmem section.

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 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 7.6.1, *The CODE_SECTION Pragma*, on page 7-14 and 7.6.4, *The DATA_SECTION Pragma*, on page 7-17).

8.1.2 C System Stack

The C compiler uses a stack to:

- ❑ Save function return addresses
- ❑ Allocate local variables
- ❑ Pass arguments to functions
- ❑ Save temporary results

The runtime 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 5.4, *Linker Option*s, on page 5-6.

At system initialization, SP is set to a designated address for the top of the stack. This address is the first location past the end 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.

At system initialization, SP is set to the first 8-byte aligned address before the end (highest numerical address) of the .stack section. This address is the first location past the end 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 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 programs, be sure to restore the stack pointer to the state it had before the function was entered. (For more information about using the stack pointer, see section 8.3, *Register Conventions*, on page 8-15; for more information about the stack, see section 8.4, *Function Structure and Calling Conventions*, on page 8-17.)

Note: Stack Overflow

The compiler provides no means to check for stack overflow during compilation or at runtime. 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 runtime.

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 5.4, *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 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 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 8-35.

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 runtime 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 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 MVK and MVKH instructions to move the global variable _x into the A0 register, then loads the B0 register using a pointer to A0:

  ```
  MVK    _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 7.3.4.4, *Large Model Option (–ml)*, on page 7-11.

For more information on the storage allocation of global and static variables, see section 7.3.4, *The near and far Keywords*, on page 7-9.

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 ’C6x compilers 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 runtime. 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 8.8.2, *Initialization Tables*, on page 8-37.)

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:

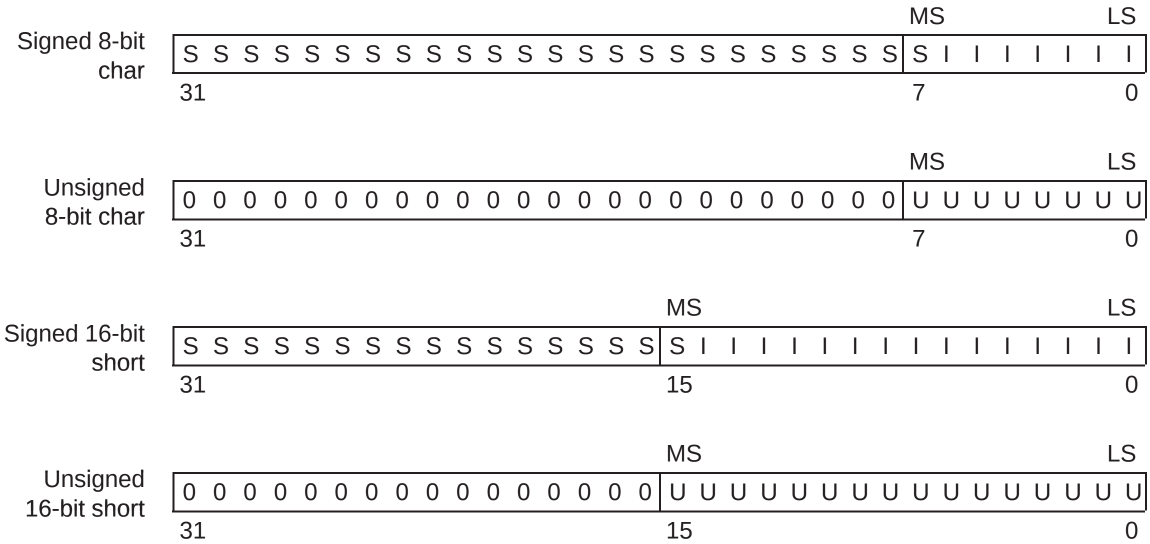
Table 8–1. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	Bits 0–7 of register	8 bits
unsigned char	Bits 0–7 of register	8 bits
short	Bits 0–15 of register	16 bits
unsigned short	Bits 0–15 of register	16 bits
int	Entire register	32 bits
unsigned int	Entire register	32 bits
enum	Entire register	32 bits
float	Entire register	32 bits
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
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 8-bit boundary; members are stored as their individual types require.
array	Members are stored as their individual types require.	Members are stored as their individual types require, 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

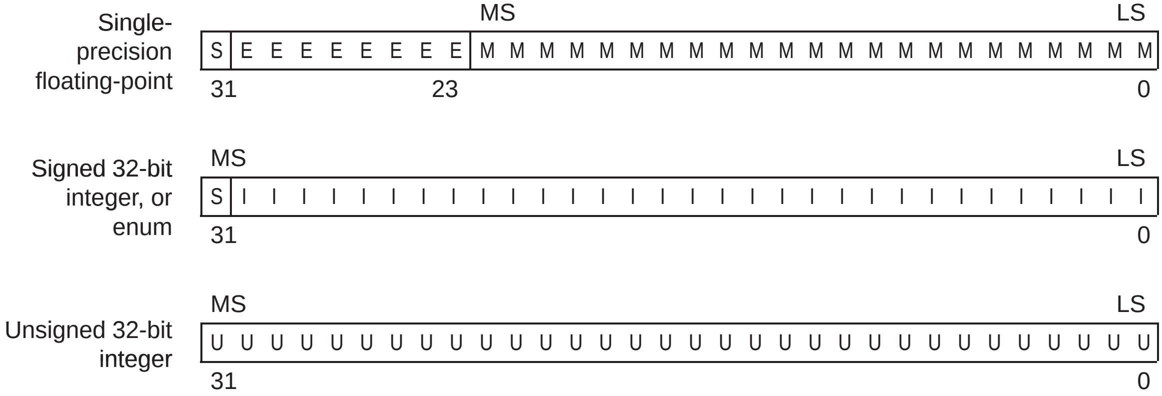


Legend: S = sign
I = signed integer
U = unsigned integer
MS = most significant
LS = least significant

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 0–32 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



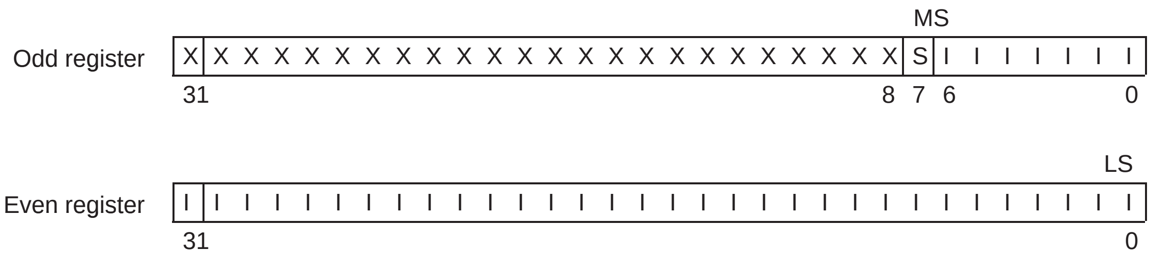
Legend: S = sign
M = mantissa
U = unsigned integer
E = exponent
I = signed integer
MS = most significant
LS = least significant

8.2.1.3 long Data Types (signed and unsigned)

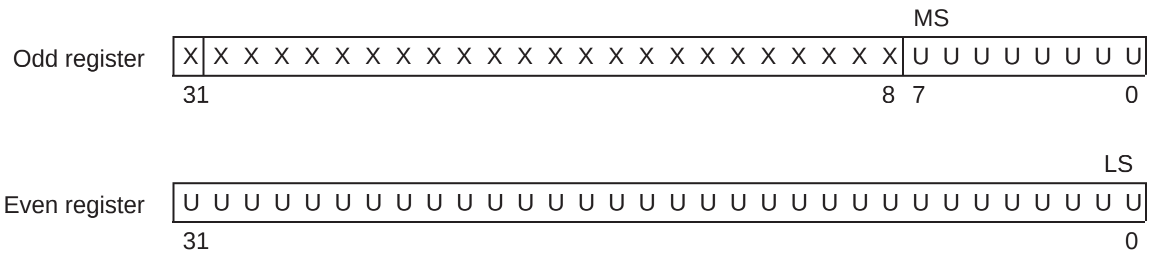
Long and unsigned long data types are stored in an odd/even pair of registers (see Figure 8–3) and are always referenced as a pair in the format of odd register:even register (for example, A1:A0). In little-endian mode, the lower address is loaded into the even register and the higher address is loaded 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

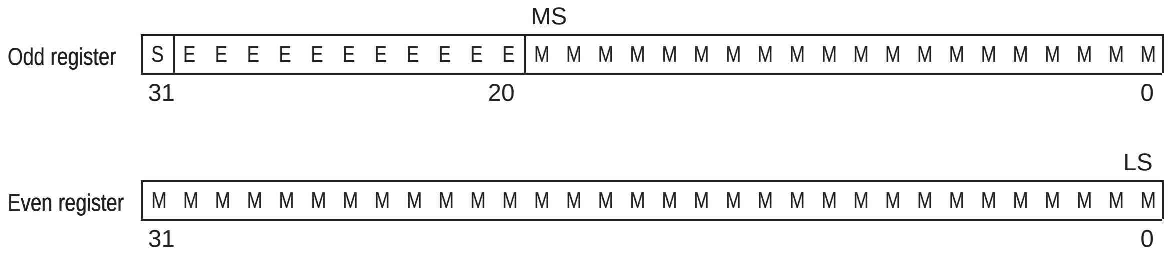


Legend: S = sign; U = unsigned integer; I = signed integer; X = unused; MS = most significant; LS = least significant

8.2.1.4 double and long double Data Types

Double and long double data types are stored in an odd/even pair of registers (see Figure 8–4) 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 8–4. Double-Precision Floating-Point Data Storage Format



Legend: S = sign
M = mantissa
E = exponent
MS = most significant
LS = least significant

8.2.1.5 Structures and Arrays

A nested structure is aligned on a 4-byte boundary only if it does not contain a double or a long double. Top level structures and nested structures containing a long, unsigned long, double or long double are aligned on an 8-byte boundary. Structures always reserve a multiple of four bytes of storage in memory. However, when a structure contains a double or a long double type, the structure reserves a multiple of eight bytes. Members of structures are stored in the same manner as if they were individual objects.

Arrays are aligned on a boundary required by their element types. Elements of arrays are stored in the same manner as if they were individual objects.

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 8–5).

Figure 8–5 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 8–5. Bit Field Packing in Big-Endian and Little-Endian Formats

	MS (bit 31)			LS (bit 0)
Big-endian register	A6 A5 A4 A3 A2 A1 A0 B9	B8 B7 B6 B5 B4 B3 B2 B1	B0 C2 C1 C0 D1 D0 E8 E7	E6 E5 E4 E3 E2 E1 E0 XX

	Byte 0	Byte 1	Byte 2	Byte 3
Big-endian memory	A6 A5 A4 A3 A2 A1 A0 B9	B8 B7 B6 B5 B4 B3 B2 B1	B0 C2 C1 C0 D1 D0 E8 E7	E6 E5 E4 E3 E2 E1 E0 XX

	MS (bit 31)			LS (bit 0)
Little-endian register	XX E8 E7 E6 E5 E4 E3 E2	E1 E0 D1 D0 C2 C1 C0 B9	B8 B7 B6 B5 B4 B3 B2 B1	B0 A6 A5 A4 A3 A2 A1 A0

	Byte 0	Byte 1	Byte 2	Byte 3
Little-endian memory	B0 A6 A5 A4 A3 A2 A1 A0	B8 B7 B6 B5 B4 B3 B2 B1	E1 E0 D1 D0 C2 C1 C0 B9	XX E8 E7 E6 E5 E4 E3 E2

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 8-35.

- 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 environment. If you plan to interface an assembly language routine to a 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.

8.3.1 Register Variables and Register Allocation

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, 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, 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 and longs	B5	Parent	Argument 2 with B4 for doubles and longs
A6	Parent	Argument 3	B6	Parent	Argument 4
A7	Parent	Argument 3 with A6 for doubles and longs	B7	Parent	Argument 4 with B6 for doubles and longs
A8	Parent	Argument 5	B8	Parent	Argument 6
A9	Parent	Argument 5 with A8 for doubles and longs	B9	Parent	Argument 6 with B8 for doubles and longs
A10	Child	Argument 7	B10	Child	Argument 8
A11	Child	Argument 7 with A10 for doubles and longs	B11	Child	Argument 8 with B10 for doubles and longs
A12	Child	Argument 9	B12	Child	Argument 10
A13	Child	Argument 9 with A12 for doubles and longs	B13	Child	Argument 10 with B12 for doubles and longs
A14	Child	—	B14	Child	Data page pointer (DP)
A15	Child	Frame pointer (FP)	B15	Child	Stack pointer (SP)

8.4 Function Structure and Calling Conventions

The C compiler imposes a strict set of rules on function calls. Except for special runtime support functions, any function that calls or is called by a C function must follow these rules. Failure to adhere to these rules can disrupt the 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, 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 8–6 shows the register argument conventions.

Figure 8–6. 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

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, 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 code. This is because the 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 or long double 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 the optimizer 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 8-18.

8.5 Interfacing C With Assembly Language

The following are ways to use assembly language with C code:

- ❑ Use separate modules of assembled code and link them with compiled C modules (see section 8.5.1).
- ❑ Use intrinsics in C source to directly call an assembly language statement (see section 8.5.2 on page 8-24).
- ❑ Use inline assembly language embedded directly in the C source (see section 8.5.5 on page 8-29).
- ❑ Use assembly language variables and constants in C source (see section 8.5.6 on page 8-30).

8.5.1 Using Assembly Language Modules With C Code

Interfacing 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 8-17 and the register conventions defined in section 8.3, *Register Conventions*, on page 8-15. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions.

Follow these guidelines to interface assembly language and C:

- ❑ All functions, whether they are written in C or assembly language, must follow the register conventions outlined in section 8.3, *Register Conventions*, on page 8-15.
- ❑ 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 8-32.

- When you call a 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 8-17.

  Remember that only A10 to A15 and B10 to B15 are preserved by the C compiler. 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 declarations. Integers and 32-bit floating-point (float) values are returned in A4. Doubles and long doubles 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 startup routine in boot.c 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 adds an underscore (_) to the beginning of all identifiers (that is, labels). In assembly language modules, you must use an underscore prefix for all objects that are to be accessible from C. For example, a C object named x is called _x in assembly language. Identifiers that are used only in assembly language modules can use any name that does not begin with a leading underscore without conflicting with a C identifier.

- Any object or function declared in assembly language that is accessed or called from C must be declared with the .def or .global directive in the assembler. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C function or object from assembly language, declare the C object with .ref or .global. 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

(a) C program

```
extern int asmfunc(); /* declare external asm function */
int gvar = 4;         /* define global variable        */

main()
{
   int i;
   i = 1;
   i = asmfunc(i);    /* call function normally        */
}
```

(b) Assembly language program

```
   .global   _gvar        ; declare external variables
   .global   _asmfunc     ; declare external function
_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 functions, you need to declare assembly functions only if they return noninteger values or pass noninteger parameters.

8.5.2 Using Intrinsics to Access Assembly Language Statements

The 'C6000 C compiler recognizes a number of intrinsic operators. Intrinsics are used like functions and produce assembly language statements that would otherwise be inexpressible in C. You can use 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 correspond to the indicated 'C6000 assembly language instructions. See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

Table 8–3. TMS320C6000 C Compiler Intrinsics

C Compiler Intrinsic	Assembly Instruction	Description	Device†
int **_abs(**int *src2***);** int**_labs(**long *src2***);**	**ABS**	Returns the saturated absolute value of src2	
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.	
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.	
int **_dpint(**double *src***);**	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register	'C67x

† Instructions not specified with a device apply to all 'C6000 devices.

Table 8–3. TMS320C6000 C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device†
int **_ext(**uint *src2,* uint *csta*, int *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 unsigneds, 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	

† Instructions not specified with a device apply to all ’C6000 devices.

Table 8–3. TMS320C6000 C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device†
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.	
int **_mpy(**int *src1*, int *src2***);** int **_mpyus(**uint *src1*, int *src2***);** int **_mpysu(**int *src1*, uint *src2***);** uint **_mpyu(**uint *src1*, uint *src2***);**	**MPY** **MPYUS** **MPYSU** **MPYU**	Multiplies the 16 LSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyh(**int *src1*, int *src2***);** int **_mpyhus(**uint *src1*, int *src2***);** int **_mpyhsu(**int *src1*, uint *src2***);** uint **_mpyhu(**uint *src1*, uint *src2***);**	**MPYH** **MPYHUS** **MPYHSU** **MPYHU**	Multiplies the 16 MSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpyhl(**int *src1*, int *src2***);** int **_mpyhuls(**uint *src1*, int *src2***);** int **_mpyhslu(**int *src1*, uint *src2***);** uint **_mpyhlu(**uint *src1*, uint *src2***);**	**MPYHL** **MPYHULS** **MPYHSLU** **MPYHLU**	Multiplies the 16 MSBs of src1 by the 16 LSBs of src2 and returns the result. Values can be signed or unsigned.	
int **_mpylh(**int *src1*, int *src2***);** int **_mpyluhs(**uint *src1*, int *src2***);** int **_mpylshu(**int *src1*, uint *src2***);** uint **_mpylhu(**uint *src1*, uint *src2***);**	**MPYLH** **MPYLUHS** **MPYLSHU** **MPYLHU**	Multiplies the 16 LSBs of src1 by the 16 MSBs of src2 and returns the result. Values can be signed or unsigned.	
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	
double **_rcpdp(**double *src***);**	**RCPDP**	Computes the approximate 64-bit double reciprocal	’C67x
float **_rcpsp(**float *src***);**	**RCPSP**	Computes the approximate 32-bit float reciprocal	’C67x
double **_rsqrdp(**double *src***);**	**RSQRDP**	Computes the approximate 64-bit double square root reciprocal	’C67x

† Instructions not specified with a device apply to all ’C6000 devices.

Table 8–3. TMS320C6000 C Compiler Intrinsics (Continued)

C Compiler Intrinsic	Assembly Instruction	Description	Device†
float **_rsqrsp(**float *src***);**	**RSQRSP**	Computes the approximate 32-bit float square root reciprocal	’C67x
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	
uint **_sshl(**uint *src2*, uint *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result	
int **_spint(**float**);**	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register	’C67x
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.	

† Instructions not specified with a device apply to all ’C6000 devices.

8.5.3 Using _nassert to Expand Compiler Knowledge of Loops

The _nassert intrinsic has been expanded to allow other types of information. You can now 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:

```
_nassert (trip_count == 10);
for (i = 0; i < trip_count; i++) { ...
```

_nassert(); can also be used to specify a range for the trip count as well as a factor of the trip count. For example:

```
_nassert ((trip >= 8) && (trip <= 48) && ((trip % 8) ==
0));
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 –mi*n* option is used to specify that interrupts do occur every *n* cycles.

8.5.4 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 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 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.

8.5.5 Using Inline Assembly Language

Within a 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 7.5, *The asm Statement*, on page 7-13.

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 environment. The compiler does not check or analyze the inserted instructions.
- Avoid inserting jumps or labels into 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 variable when using an asm statement.
- Do not use the asm statement to insert assembler directives that change the assembly environment.

8.5.6 Accessing Assembly Language Variables From C

It is sometimes useful for a 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.

8.5.6.1 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, declare the variable as *extern* and access it normally.

Example 8–2 shows how you can access a variable defined in .bss.

Example 8–2. 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
```

8.5.6.2 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 only with the use of special operators.

For normal variables defined in 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 is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 8–3.

Example 8–3. 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 8–3, 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 code without disrupting the C environment. When the 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 environment and are easily incorporated with asm statements or calling an assembly language function.

8.6.1 Saving Registers During Interrupts

When 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.

8.6.2 Using C Interrupt Routines

A C interrupt routine is like any other 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 interrupt routines can allocate up to 32K on the stack for local variables. For example:

```
interrupt void example (void)
{
...
}
```

If a 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 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 functions by using the interrupt pragma or the interrupt keyword. For more information, see section 7.6.13, *The INTERRUPT Pragma*, on page 7-20, and section 7.3.3, *The interrupt Keyword*, on page 7-8.

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 variables, and call C functions normally. When calling C functions, be sure that any registers listed in Table 8–2 on page 8-16 are saved, because the C function can modify them.

8.7 Runtime-Support Arithmetic Routines

The runtime-support library contains a number of assembly language functions that provide arithmetic routines for 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 calling sequence. You can call them directly from C, but the compiler automatically adds them when appropriate.

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 8–4 summarizes the runtime-support functions used for arithmetic.

Table 8–4. Summary of Runtime-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
uint	_fixdu (double)	Convert double to unsigned integer
ulong	_fixdul (double)	Convert double to unsigned long
double	_cvtfd (float)	Convert float to double
int	_fixfi (float)	Convert float to signed integer
long	_fixfli (float)	Convert float to long
uint	_fixfu (float)	Convert float to unsigned integer
ulong	_fixful (float)	Convert float to unsigned long
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
float	_fltlif (long)	Convert signed long to float

Table 8–4. Summary of Runtime-Support Arithmetic Functions (Continued)

Type	Function	Description
double	_fltlid (long)	Convert signed long to double
double	_fltuld (ulong)	Convert unsigned long to double
float	_fltulf (ulong)	Convert unsigned long to float
double	_absd (double)	Double absolute value
double	_negd (double)	Double negative value
float	_absf (float)	Float absolute value
float	_negf (float)	Float negative value
double	_addd (double, double)	Double addition
double	_cmpd (double, double)	Double comparison
double	_divd (double, double)	Double division
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

8.8 System Initialization

Before you can run a C program, you must create the C runtime environment. The C boot routine performs this task using a function called c_int00. The runtime-support source library, rts.src, contains the source to this routine in a module named boot.asm.

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 rts6201.lib as one of the linker input files.

When 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 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 environment.

8.8.1 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a 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 standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The 'C6000 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.

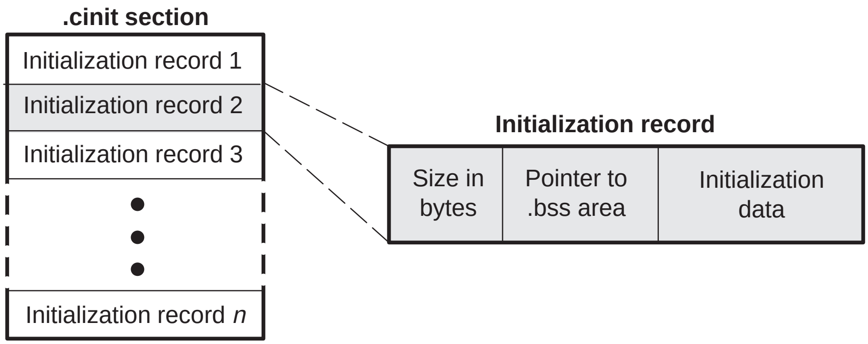
You cannot use these methods with code that is burned into ROM.

Global variables are either autoinitialized at runtime or at load time. For information, see sections 8.8.3, *Autoinitialization of Variables at Runtime*, on page 8-40, and 8.8.4, *Initialization of Variables at Load Time*, on page 8-41. Also, see section 7.7, *Initializing Static and Global Variables*,on page 7-22.

8.8.2 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 8–7 shows the format of the .cinit section and the initialization records.

Figure 8–7. Format of Initialization Records in the .cinit Section



An initialization record contains the following information:

- The first field of an initialization record is the size (in bytes) of the initialization data. If the the size is negative, then the data is DP address patch data (described below).
- 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.

If the first field is negative, then the record represents a list of addresses that need to be patched by adding the value of the data page pointer (DP). This is only required for autoinitialized pointers to near data. The DP address patch autoinitialization record has the following fields:

- A negative size in bytes of the list of addresses
- A list of addresses to be patched

Each variable that is autoinitialized with the address of a near variable will be in the DP address patch list. Example 8–4 (a) shows initialized global variables defined in C. Example 8–4 (b) shows the corresponding initialization table.

The ”.cinit:c” is a subsection in the .cinit section that contains all scalar data. The sub–section is handeled as one record during initialization, which minimizes the overall size of the .cinit section.

Example 8–4. 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          0x17,16                  ; _i @ 0

        .sect   ”.text”
        .global _i
_i:     .usect  ”.bss:c”,2,2

        .sect   ”.cinit:c”
        .align  4
        .field          _x–$bss,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          0x1,32                  ; _a[0] @ 0
        .field          0x2,32                  ; _a[1] @ 32
        .field          0x3,32                  ; _a[2] @ 64
        .field          0x4,32                  ; _a[3] @ 96
        .field          0x5,32                  ; _a[4] @ 128
IR_1:   .set    20

        .sect   ”.text”
        .global _a
        .bss    _a,20,4
```

(b) Initialized information for variables defined in (a) (Continued)

```
;*****************************************************************************
;* MARK THE END OF THE SCALAR INIT RECORD IN CINIT:C                         *
;*****************************************************************************
CIR:    .sect   ".cinit:c"
;*****************************************************************************
;* ADDRESS RELOCATIONS – SCALARS                                              *
;*****************************************************************************
        .sect   ".cinit:c"
        .align  8
        .field  -4,32
        .align  4
        .field          _p,32
```

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.

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.

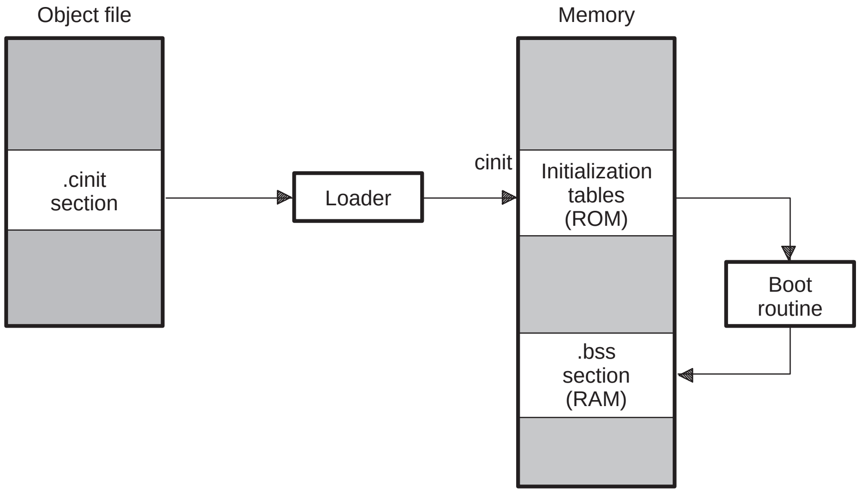
8.8.3 Autoinitialization of Variables at Runtime

Autoinitializing variables at runtime 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 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 8–8 illustrates autoinitialization at runtime. Use this method in any system where your application runs from code burned into ROM.

Figure 8–8. Autoinitialization at Run time



8.8.4 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.

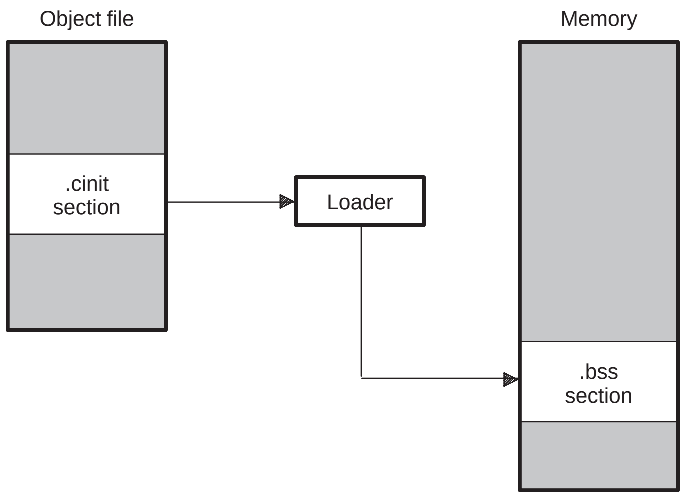
When you use the –cr linker option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to –1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no runtime 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 8–9 illustrates the initialization of variables at load time.

Figure 8–9. Initialization at Load Time



Chapter 9

Runtime-Support Functions

Some of the tasks that a C program performs (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C language itself. However, the ANSI C standard defines a set of runtime-support functions that perform these tasks. The TMS320C6000 C compiler implements the complete ANSI standard library except for those facilities that handle exception conditions and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI-specified functions, the TMS320C6000 runtime-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 runtime-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 compiler:

- rts6201.lib and rts6701.lib—runtime-support object libraries for use with little-endian code, and rts6201e.lib rts6701e.lib—runtime-support object libraries for use with big-endian code.

  The rts6201.lib, rts6701.lib, rts6201e.lib, and rts6701e.lib libraries do not contain functions involving signals and locale issues. They do contain the following:

  - ANSI 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 to access specific instructions

- rts.src—runtime-support source library. The runtime-support object libraries are built from the C and assembly source contained in the rts.src library.

You can control how the runtime-support functions are called in terms of near or far calls with the –mr option. For more information, see section 7.3.4.3, *Controlling How Runtime-Support Functions Are Called (–mr Option)*, on page 7-10.

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 runtime-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*.

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:

```
cl6x –options atoi.c strcpy.c           ;recompile
ar6x r rts6201.lib  atoi.obj strcpy.obj  ;rebuild library
```

You can also build a new library this way, rather than rebuilding into rts6201.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 runtime-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 optimizer (–o2) and inline function expansion (–x) options does not affect compatibility with code compiled without these options. 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 (using the debugger). For example, printf statements executed in a program appear in the debugger command window. When used in conjunction with the debugging tools, 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 for each module that references a C I/O function.

For example, given the following 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 shell command compiles, links, and creates the file main.out:

```
cl6x main.c –z –heap 400 –l rts6201.lib –o main.out
```

Executing main.out under the debugger on a SPARC host accomplishes the following:

1) Opens the file *myfile* in the directory where the debugger was invoked
2) Prints the string *Hello, world* into that file
3) Closes the file
4) Prints the string *Hello again, world* in the debugger command window

With properly written device drivers, the library also offers facilities to perform I/O on a user-specified device.

Note:

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.

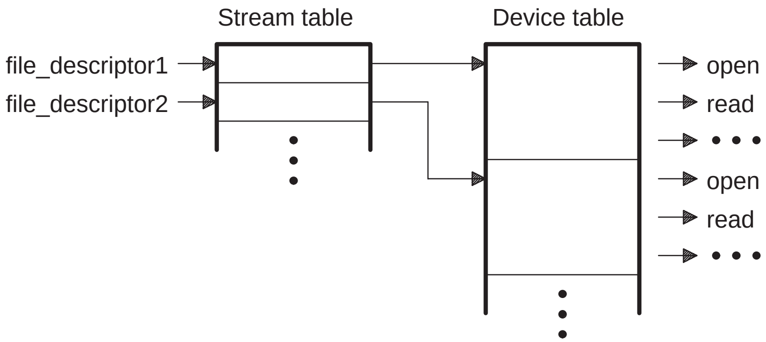
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level routines.

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.

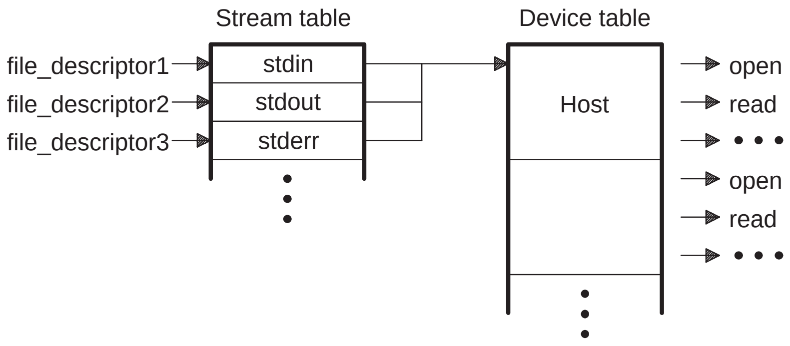
The data structures interact as shown in Figure 9–1.

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 runtime-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.

close — *Close File or Device For I/O*

Syntax

#include <stdio.h>
#include <file.h>

int 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 not successful

lseek — *Set File Position Indicator*

Syntax

#include <stdio.h>
#include <file.h>

long 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 not successful

open — *Open File or Device For I/O*

Syntax

#include <stdio.h>
#include <file.h>

int open(char ****path*, unsigned** *flags***, int** *mode***);**

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   (0x0100)  /* open with file create */
  O_TRUNC   (0x0200)  /* 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. Note, however, that 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 *mode* is required but ignored.

Return Value

The function returns one of the following values:

#	stream number assigned by the low-level routines that the device-level driver associates with the opened file or device if successful
< 0	if not successful

read — *Read Characters From Buffer*

Syntax

#include <stdio.h>
#include <file.h>

int 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 not successful

rename — *Rename File*

Syntax

#include <stdio.h>
#include <file.h>

int rename(char ****old_name*, **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

#include <stdio.h>
#include <file.h>

int unlink(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 not successful

write — *Write Characters to Buffer*

Syntax

#include <stdio.h>
#include <file.h>

int write(int *file_descriptor***, 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 not successful

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 (pages 9–7 to 9–10), 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. The structure representing a device is also defined in stdio.h 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 9-45.

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(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 long 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);
}
```

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 runtime-support functions:

```
assert.h     float.h      setjmp.h     stdlib.h
ctype.h      gsm.h        stdarg.h     string.h
errno.h      limits.h     stdef.h      time.h
file.h       math.h       stdio.h
```

In order to use a runtime-support function, you must first use the #include preprocessor directive to include the header file that declares the function. For example, 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)*, on page 9-14 through 9.3.15, *Time Functions (time.h)*, on page 9-22 describe the header files that are included with the ’C6000 C compiler. Section 9.5, *Summary of Runtime-Support Functions and Macros*, on page 9-30 lists the functions that these headers declare.

9.3.1 Diagnostic Messages (assert.h)

The assert.h 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 header refers to another macro named NDEBUG (assert.h does not define NDEBUG). If you have defined NDEBUG as a macro name when you include assert.h, assert is turned off and does nothing. If NDEBUG is *not* defined, assert is enabled.

The assert.h header refers to another macro named NASSERT (assert.h does not define NASSERT). If you have defined NASSERT as a macro name when you include assert.h, assert acts like _nassert. The _nassert intrinsic generates no code and tells the optimizer that the expression declared with assert is true. This gives a hint to the optimizer 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 tht a loop will execute a certain number of times. For more information, see section 8.5.3, *Using _nassert to Enable SIMD and Expand Compiler Knowledge of Loops*, on page 8-28.

The assert function is listed in Table 9–3 (a) on page 9-31.

9.3.2 Character-Typing and Conversion (ctype.h)

The ctype.h 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 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 9-31.

9.3.3 Error Reporting (errno.h)

The errno.h 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 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 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 operations of a GSM vocoder.

9.3.6 Limits (float.h and limits.h)

The float.h and limits.h 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)

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

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)

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 header file supplied with the compiler for the full precision carried by the processor.

9.3.7 Function Calls as near or far (linkage.h)

The linkage.h header declares two macros. 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 shell 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_[];
```

9.3.8 Floating-Point Math (math.h)

The math.h header declares several trigonometric, exponential, and hyperbolic math functions. These functions are listed in Table 9–3 (c) on page 9-32. 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 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 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 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.

9.3.9 Nonlocal Jumps (setjmp.h)

The setjmp.h header defines a type and a macro and declares a function for bypassing the normal function call and return discipline. These include:

- *jmp_buf*, an array type suitable for holding the information needed to restore a calling environment
- *setjmp*, a macro that saves its calling environment in its jmp_buf argument for later use by the longjmp function
- *longjmp*, a function that 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 9-35.

9.3.10 Variable Arguments (stdarg.h)

Some functions can have a variable number of arguments whose types can differ. Such functions are called *variable-argument functions*. The stdarg.h 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 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 9-35.

9.3.11 Standard Definitions (stddef.h)

The stddef.h header defines types and macros. The types are:

- *ptrdiff_t*, a signed integer type that is the data type resulting from the subtraction of two pointers
- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator

The macros are:

- *NULL*, a macro that expands to a null pointer constant(0)
- *offsetof(type, identifier)*, a macro that 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.

9.3.12 Input/Output Functions (stdio.h)

The stdio.h header defines types and macros and declares functions. The types are:

- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator. Originally defined in stddef.h
- *fpos_t*, an unsigned integer type that can uniquely specify every position within a file
- *FILE*, a structure type to record all the information necessary to control a stream

The macros are:

- *NULL*, a macro that expands to a null pointer constant(0). Originally defined in stddef.h. It is not redefined if it was already defined.
- *BUFSIZ*, a macro that expands to the size of the buffer that setbuf() uses
- *EOF*, the end-of-file marker
- *FOPEN_MAX*, a macro that expands to the largest number of files that can be open at one time
- *FILENAME_MAX*, a macro that expands to the length of the longest file name in characters
- *L_tmpnam*, a macro that expands to the longest filename string that tmpnam() can generate

- *SEEK_CUR*, *SEEK_SET*, and *SEEK_END*, macros that expand to indicate the position (current, start-of-file, or end-of-file, respectively) in a file
- *TMP_MAX*, a macro that expands to the maximum number of unique filenames that tmpnam() can generate
- *stderr*, *stdin*, *stdout*, pointers to the standard error, input, and output files, respectively

The input/output functions are listed in Table 9–3 (f) on page 9-35.

9.3.13 General Utilities (stdlib.h)

The stdlib.h 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:

- *div_t*, a structure type that is the type of the value returned by the div function
- *ldiv_t*, a structure type that is the type of the value returned by the ldiv function

The functions are:

- String conversion functions that convert strings to numeric representations
- Searching and sorting functions that search and sort arrays
- Sequence-generation functions that generate a pseudo-random sequence and choose a starting point for a sequence
- Program-exit functions that 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 9-38.

9.3.14 String Functions (string.h)

The string.h 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 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 9-39.

9.3.15 Time Functions (time.h)

The time.h 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 9-41.

You can adjust local time for local or seasonal variations. Obviously, local time depends on the time zone. The time.h 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 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.

9.4 Saving On-Chip Memory by Placing Runtime-Support Off-Chip

One of many techniques you might use to save valuable on-chip space is to place the code and data needed by the runtime-support functions in off-chip memory.

Placing the runtime-support in off-chip memory has the advantage of saving valuable on-chip space. However, it comes at a cost. The runtime-support functions will run much slower. Depending on your application, this may or may not be acceptable. It is also possible your application doesn't use the runtime-support library much, and placing the runtime-support off-chip saves very little on-chip memory.

The following terms are used in this section:

- Normal runtime-support functions. Ordinary runtime-support functions. Example: strcpy.
- Internal runtime-support functions that implement atomic C operations like divide or floating point math functions on the 'C62xx. Example: _divu performs 32-bit unsigned divide.
- near calls are function calls performed with a ordinary PC-relative branch instruction. The destination of such branches must be within 1048576 (0x100000) words of the branch. Such calls use one instruction word and one cycle.
- far calls are functions calls performed by loading the address of the function into a register and then branching to the address in the register. There is no limit on the range of the call. Such calls use three instruction words and three cycles.

For information on using the –mr shell option to control near and far function calls, see section 7.3.4.3, *Controlling How Runtime-Support Functions Are Called (–mr Option)*, on page 7-10.

9.4.1 Must #include Header File

When you call a runtime-support function, you must include the header file which corresponds to that function. For instance, when you call memcmp, you must declare #include <string.h>. If you do not include the header, the memcmp call looks like a normal user call to the compiler, and the effect of using –mr1 does not occur.

See section 9.3, *Header Files*, on page 9-13 for a list of header files and more detailed information of each 'C6000 header file.

9.4.2 Runtime-Support Data

Most runtime-support functions do not have any data of their own. Data is typically passed as arguments or through pointers. However, a few functions do have their own data. All of the isxxx character recognition functions defined in ctype.h refer to a global table. And many of the floating-point math functions have their own constant look-up tables. All runtime-support data is now defined to be far data, i.e. accessed without regard to where it is in memory. This does not necessarily mean this data is in off-chip memory.

9.4.3 How to Link When Runtime-Support Functions Are Off-Chip

You get runtime-support code and data in off-chip memory through the linking process. Example 9–1 shows a sample linker command file for linking when the runtime-support functions are in off-chip memory. Using this linker command file, your sections are built and allocated normally.

The .cinit section is built normally as well. It is important to not allocate the runtime-support .cinit sections separately as is done with the other sections. All of the .cinit sections must be combined together into one section for autoinitialization of global variables to work properly.

The .stack, .sysmem, and .cio sections are entirely created from within the runtime-support functions. So, you do not need any special syntax to build and allocate these sections separately from sections you define. Typically, you place the .stack (system stack) and .sysmem (dynamic memory heap) sections in on-chip memory for performance reasons. The .cio section is a buffer used by printf and related functions. You can typically afford slower performance of such I/O functions, so it is placed in off-chip memory.

The .rtstext section collects all the .text, or code, sections from runtime-support and allocates them to external memory name EXT0. If needed, replace the library name rts6201.lib with the library you normally use, perhaps rts6701.lib. The –l option is required, and no space is allowed between the –l and the name of the library. The choice of EXT0 is arbitrary. Use the memory range which makes the most sense in your application.

The .bss section combines all of the undefined data sections together. Undefined sections reserve memory without any initialization of the contents of that memory. You use the .bss and.usect assembler directives to create undefined data sections.

The .rtsdata section combines all of the defined data sections together. Defined data sections both reserve and initialize the contents of a section. You use the .sect assembler directive to create defined sections.

Example 9–1. Runtime-Support Linker Command File

```
/****************************************************************************/
/* farlnk.cmd – Link command file which puts RTS off-chip                   */
/****************************************************************************/
–c
–heap  0x2000
–stack 0x4000
/* Memory Map 1 – the default */
MEMORY
{
        PMEM:   o = 00000000h   l = 00010000h
        EXT0:   o = 00400000h   l = 01000000h
        EXT1:   o = 01400000h   l = 00400000h
        EXT2:   o = 02000000h   l = 01000000h
        EXT2:   o = 03000000h   l = 01000000h
        BMEM:   o = 80000000h   l = 00010000h
}
SECTIONS
{
    /*----------------------------------------------------------------------*/
    /* Sections of user code and data                                       */
    /*----------------------------------------------------------------------*/
    .text       >       PMEM
    .bss        >       BMEM
    .const      >       BMEM
    .data       >       BMEM
    .switch     >       BMEM
    .far        >       EXT2
    /*----------------------------------------------------------------------*/
    /* All of .cinit, including from RTS, must be collected together in     */
    /* one step.                                                            */
    /*----------------------------------------------------------------------*/
    .cinit      >       BMEM
    /*----------------------------------------------------------------------*/
    /* Sections defined only in RTS.                                        */
    /*----------------------------------------------------------------------*/
    .stack      >       BMEM
    .sysmem     >       BMEM
    .cio        >       EXT0

    /*----------------------------------------------------------------------*/
    /* RTS code – placed off chip                                           */
    /*----------------------------------------------------------------------*/
    .rtstext   { –lrts6201.lib(.text)   } > EXT0
    /*----------------------------------------------------------------------*/
    /* RTS data – undefined sections – placed off chip                      */
    /*----------------------------------------------------------------------*/
    .rtsbss    { –lrts6201.lib(.bss)
                 –lrts6201.lib(.far)    } > EXT0
    /*----------------------------------------------------------------------*/
    /* RTS data – defined sections – placed off chip                        */
    /*----------------------------------------------------------------------*/
    .rtsdata   { –lrts6201.lib(.const)
                 –lrts6201.lib(.switch) } > EXT0
}
```

It is necessary to build and allocate the undefined data sections separately from the defined data sections. When a defined data section is combined with an undefined data section, the resulting output section is a defined data section. This forces the linker to fill the range of memory corresponding to the undefined section with a value, typically the default value of 0. This has the undesirable effect of making your resulting .out file much larger.

You may get a linker warning such as the following:

```
>> farlnk.cmd, line 65: warning: rts6201.lib(.switch) not found
```

This message simply means that none of the runtime-support functions needed by your application define a .switch section. Simply delete the corresponding –l option entry in the linker command file to avoid the message. If your application changes such that you later do include an runtime-support function with a .switch section, it will be linked next to the .switch sections from your code. This is fine, except it is taking up that valuable on-chip memory. So, you may want to check for this situation occasionally by looking at the linker map file you create with the linker –m option.

9.4.4 Example Compiler Invocation When Runtime-Support Is Off-Chip

A typical build could look like ...

```
cl6x –mr1 <options> <C files> –z –o app.out –m app.map farlnk.cmd
```

In this one step you both compile all the C files and link them together. The ’C6000 executable image file is named app.out and the linker map file is named app.map.

9.4.5 Linker Error Messages When Calls Don't Reach

When you try to call a function which, due to how you linked your application, is too far away from a call site to be reached with the normal PC-relative branch instruction, you will see a linker error message such as:

```
>> PC–relative displacement overflow. Located in file.obj,
section .text, SPC offset 000000bc
```

The message means that in the named object file, in that particular section, there is a PC-relative branch instruction which is trying to reach a call destination that is too far away. The SPC offset is the section program counter (SPC) offset within that section where the branch occurs. For C code, the section name is always .text. If this happens to you when you are linking C code, follow these steps to find the problem:

1) Recompile the C source file as you did before but include –s –al.

   ```
   cl6x <other options> –s –al file.c
   ```

   This gives you C interlisted in the assembly output and creates an assembler listing file with the .lst extension.

2) Edit the resulting .lst file, in this case file.lst.

3) Each line in the assembly listing has several fields. The field you are interested in here is the second one, the section program counter (SPC) field. Find the line with the same SPC field as the SPC offset given in the linker error message, such as:

```
245 000000bc 0FFFEC10!           B       .S1     _atoi            ; |56|
```

   In this case, the call to the function atoi is too far away from the location where this code is linked.

   It is possible that use of the –s option will cause instructions to move around some and thus the instruction at the given SPC offset is not a branch. The branch nearest to that instruction is the likely culprit. Or, you can rebuild the whole application with –s –al and relink to see the new SPC offset of the error.

To fix the problem, your choices are:

- Use the –mr1 option to force the call to atoi, and all other runtime-support functions, to be far
- Compile with –ml1 or higher to force all calls to be far
- Rewrite your linker command file (looking at a map file usually helps) so that all the calls to atoi are close (within 0x100000 words) to where atoi is linked.

9.4.6 Changing Runtime-Support Data to near

If for some reason you do not want accesses of runtime-support data to use the far access method, take these steps:

1) Edit the linkage.h header file, and change the

   ```
   #define _DATA_ACCESS far
   ```

   macro to

   ```
   #define _DATA_ACCESS near
   ```

   to force all access of runtime-support data to use near access, or change it to

   ```
   #define _DATA_ACCESS
   ```

   if you want runtime-support data access to use the same method used when accessing ordinary user data.

2) Replace the linkage.h entry in the source library using the library-build utility:

   ```
   ar6x –r rts.src linkage.h
   ```

3) Rename or delete the object library you use when linking.

4) Rebuild the object library you use with the library-build command as given in

You have to perform this process each time you install an update of the code generation toolset.

9.5 Summary of Runtime-Support Functions and Macros

Table 9–3 summarizes the runtime-support header files (in alphabetical order) provided with the TMS320C6000 ANSI C compiler. Most of the functions described are per the ANSI standard and behave exactly as described in the standard.

The functions and macros listed in Table 9–3 are described in detail in section 9.6, *Description of Run time-Support Functions and Macros* on page 9-42. 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 Runtime-Support Functions and Macros

(a) Error message macro (assert.h)

Macro	Description	Page
void **assert**(int expr);	Inserts diagnostic messages into programs	9-48

(b) Character typing and conversion functions (ctype.h)

Function	Description	Page
int **isalnum**(int c);	Tests c to see if it is an alphanumeric-ASCII character	9-66
int **isalpha**(int c);	Tests c to see if it is an alphabetic-ASCII character	9-66
int **isascii**(int c);	Tests c to see if it is an ASCII character	9-66
int **iscntrl**(int c);	Tests c to see if it is a control character	9-66
int **isdigit**(int c);	Tests c to see if it is a numeric character	9-66
int **isgraph**(int c);	Tests c to see if it is any printing character except a space	9-66
int **islower**(int c);	Tests c to see if it is a lowercase alphabetic ASCII charcter	9-66
int **isprint**(int c);	Tests c to see if it is a printable ASCII character (including a space)	9-66
int **ispunct**(int c);	Tests c to see if it is an ASCII punctuation character	9-66
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	9-66
int **isupper**(int c);	Tests c to see if it is an uppercase ASCII alphabetic character	9-66
int **isxdigit**(int c);	Tests c to see if it is a hexadecimal digit	9-66
char **toascii**(int c);	Masks c into a legal ASCII value	9-96
char **tolower**(int char c);	Converts c to lowercase if it is uppercase	9-96
char **toupper**(int char c);	Converts c to uppercase if it is lowercase	9-96

Note: Functions in ctype.h are expanded inline if the –x option is used.

(c) Floating-point math functions (math.h)

Function	Description	Page
double **acos**(double x);	Returns the arc cosine of x	9-43
float **acosf**(float x);	Returns the arc cosine of x	9-43
double **acosh**(double x);	Returns the hyperbolic arc cosine of x †	9-43
float **acoshf**(float x);	Returns the hyperbolic arc cosine of x †	9-43
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 †	9-44
float **acot2f**(float x, float y);	Returns the arc cotangent of x/y †	9-44
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	9-47
float **asinf**(float x);	Returns the arc sine of x	9-47
double **asinh**(double x);	Returns the hyperbolic arc sine of x †	9-47
float **asinhf**(float x);	Returns the hyperbolic arc sine of x †	9-47
double **atan**(double x);	Returns the arc tangent of x	9-48
double **atan2**(double y, double x);	Returns the arc tangent of y/x	9-49
float **atan2f**(float y, float x);	Returns the arc tangent of y/x	9-49
float **atanf**(float x);	Returns the arc tangent of x	9-48
double **atanh**(double x);	Returns the hyperbolic arc tangent of x †	9-49
float **atanhf**(float x);	Returns the hyperbolic arc tangent of x †	9-49
double **ceil**(double x);	Returns the smallest integer ≥ x; expands inline if –x is used	9-52
float **ceilf**(float x);	Returns the smallest integer ≥ x; expands inline if –x is used	9-52
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	9-54
float **coshf**(float x);	Returns the hyperbolic cosine of x	9-54
double **cot**(double x);	Returns the cotangent of x †	9-54

† Enhanced math function. See section 9.3.8 on page 9-18 for information on accessing this function.

(c) Floating-point math functions (math.h) (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 †	9-57
float **exp10f**(float x);	Returns 10.0^x †	9-57
double **exp2**(double x);	Returns 2.0^x †	9-57
float **exp2f**(float x);	Returns 2.0^x †	9-57
float **expf**(float x);	Returns e^x	9-57
double **fabs**(double x);	Returns the absolute value of x	9-58
float **fabsf**(float x);	Returns the absolute value of x	9-58
double **floor**(double x);	Returns the largest integer $\leq$ x; expands inline if –x is used	9-60
float **floorf**(float x);	Returns the largest integer $\leq$ x; expands inline if –x is used	9-60
double **fmod**(double x, double y);	Returns the exact floating-point remainder of x/y	9-60
float **fmodf**(float x, float y);	Returns the exact floating-point remainder of x/y	9-60
double **frexp**(double value, int *exp);	Returns f and exp such that $.5 \leq \|f\| < 1$ and value is equal to $f \times 2^{exp}$	9-63
float **frexpf**(float value, int *exp);	Returns f and exp such that $.5 \leq \|f\| < 1$ and value is equal to $f \times 2^{exp}$	9-63
double **ldexp**(double x, int exp);	Returns $x \times 2^{exp}$	9-67
float **ldexpf**(float x, int exp);	Returns $x \times 2^{exp}$	9-67
double **log**(double x);	Returns the natural logarithm of x	9-68
double **log10**(double x);	Returns the base-10 logarithm of x	9-68
float **log10f**(float x);	Returns the base-10 logarithm of x	9-68
double **log2**(double x);	Returns the base-2 logarithm of x †	9-68
float **log2f**(float x);	Returns the base-2 logarithm of x †	9-68
float **logf**(float x);	Returns the natural logarithm of x	9-68

† Enhanced math function. See section 9.3.8 on page 9-18 for information on accessing this function.

(c) Floating-point math functions (math.h) (Continued)

Function	Description	Page
double **modf**(double value, double *ip);	Breaks value into a signed integer and a signed fraction	9-73
float **modff**(float value, float *ip);	Breaks value into a signed integer and a signed fraction	9-73
double **pow**(double x, double y);	Returns x^y	9-73
float **powf**(float x, float y);	Returns x^y	9-73
double **powi**(double x, int y);	Returns x^i †	9-74
float **powif**(float x, int y);	Returns x^i †	9-74
double **round**(double x);	Returns x rounded to the nearest integer †	9-78
float **roundf**(float x);	Returns x rounded to the nearest integer †	9-78
double **rsqrt**(double x);	Returns the reciprocal square root of x †	9-78
float **rsqrtf**(float x);	Returns the reciprocal square root of x †	9-78
double **sin**(double x);	Returns the sine of x	9-81
float **sinf**(float x);	Returns the sine of x	9-81
double **sinh**(double x);	Returns the hyperbolic sine of x	9-82
float **sinhf**(float x);	Returns the hyperbolic sine of x	9-82
double **sqrt**(double x);	Returns the nonnegative square root of x	9-82
float **sqrtf**(float x);	Returns the nonnegative square root of x	9-82
double **tan**(double x);	Returns the tangent of x	9-94
float **tanf**(float x);	Returns the tangent of x	9-94
double **tanh**(double x);	Returns the hyperbolic tangent of x	9-95
float **tanhf**(float x);	Returns the hyperbolic tangent of x	9-95
double **trunc**(double x);	Returns x truncated toward 0 †	9-97
float **truncf**(float x);	Returns x truncated toward 0 †	9-97

† Enhanced math function. See section 9.3.8 on page 9-18 for information on accessing this function.

(d) Nonlocal jumps macro and function (setjmp.h)

Function or Macro	Description	Page
int **setjmp**(jmp_buf env);	Saves calling environment for use by longjmp; this is a macro	9-80
void **longjmp**(jmp_buf env, int _val);	Uses jmp_buf argument to restore a previously saved environment	9-80

(e) Variable argument macros (stdarg.h)

Macro	Description	Page
type **va_arg**(va_list, type);	Accesses the next argument of type type in a variable-argument list	9-98
void **va_end**(va_list);	Resets the calling mechanism after using va_arg	9-98
void **va_start**(va_list, parmN);	Initializes ap to point to the first operand in the variable-argument list	9-98

(f) C I/O functions (stdio.h)

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	9-45
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	9-58
int **feof**(FILE *_fp);	Tests the EOF indicator for the stream that _fp points to	9-58
int **ferror**(FILE *_fp);	Tests the error indicator for the stream that _fp points to	9-58
int **fflush**(register FILE *_fp);	Flushes the I/O buffer for the stream that _fp points to	9-59
int **fgetc**(register FILE *_fp);	Reads the next character in the stream that _fp points to	9-59
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	9-59
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	9-59

(f) C I/O functions (stdio.h) (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	9-61
int **fprintf**(FILE *_fp, const char *_format, ...);	Writes to the stream that _fp points to	9-61
int **fputc**(int _c, register FILE *_fp);	Writes a single character, _c, to the stream that _fp points to	9-61
int **fputs**(const char *_ptr, register FILE *_fp);	Writes the string pointed to by _ptr to the stream pointed to by _fp	9-61
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	9-62
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	9-62
int **fscanf**(FILE *_fp, const char *_fmt, ...);	Reads formatted input from the stream that _fp points to	9-63
int **fseek**(register FILE *_fp, long _offset, int _ptrname);	Sets the file position indicator for the stream that _fp points to	9-63
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.	9-64
long **ftell**(FILE *_fp);	Obtains the current value of the file position indicator for the stream that _fp points to	9-64
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	9-64
int **getc**(FILE *_fp);	Reads the next character in the stream that _fp points to	9-64
int **getchar**(void);	A macro that calls fgetc() and supplies stdin as the argument	9-65
char ***gets**(char *_ptr);	Performs the same function as fgets() using stdin as the input stream	9-65
void **perror**(const char *_s);	Maps the error number in _s to a string and prints the error message	9-73
int **printf**(const char *_format, ...);	Performs the same function as fprintf but uses stdout as its output stream	9-74
int **putc**(int _x, FILE *_fp);	A macro that performs like fputc()	9-74
int **putchar**(int _x);	A macro that calls fputc() and uses stdout as the output stream	9-74

(f) C I/O functions (stdio.h) (Continued)

Function	Description	Page
int **puts**(const char *_ptr);	Writes the string pointed to by _ptr to stdout	9-75
int **remove**(const char *_file);	Causes the file with the name pointed to by _file to be no longer available by that name	9-77
int **rename**(const char *_old_name, const char *_new_name);	Causes the file with the name pointed to by _old_name to be known by the name pointed to by _new_name	9-77
void **rewind**(register FILE *_fp);	Sets the file position indicator for the stream pointed to by _fp to the beginning of the file	9-77
int **scanf**(const char *_fmt, ...);	Performs the same function as fscanf() but reads input from stdin	9-79
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	9-79
int **setvbuf**(register FILE *_fp, register char *_buf, register int _type, register size_t _size);	Defines and associates a buffer with a stream	9-81
int **sprintf**(char *_string, const char *_format, ...);	Performs the same function as fprintf() but writes to the array that _string points to	9-82
int **sscanf** (const char *_str, const char *_fmt, ...);	Performs the same function as fscanf() but reads from the string that _str points to	9-83
FILE ***tmpfile**(void);	Creates a temporary file	9-95
char ***tmpnam**(char *_s);	Generates a string that is a valid filename (that is, the filename is not already being used)	9-96
int **ungetc**(int _c, register FILE *_fp);	Pushes the character specified by _c back into the input stream pointed to by _fp	9-97
int **vfprintf**(FILE *_fp, const char *_format, va_list _ap);	Performs the same function as fprintf() but replaces the argument list with _ap	9-99
int **vprintf**(const char *_format, va_list _ap);	Performs the same function as printf() but replaces the argument list with _ap	9-99
int **vsprintf**(char *_string, const char *_format, va_list _ap);	Performs the same function as sprintf() but replaces the argument list with _ap	9-99

(g) General functions (stdlib.h)

Function	Description	Page
void **abort**(void);	Terminates a program abnormally	9-42
int **abs**(int i);	Returns the absolute value of val; expands inline unless –x0 is used	9-42
int **atexit**(void (*fun)(void));	Registers the function pointed to by fun, called without arguments at program termination	9-49
double **atof**(const char *st);	Converts a string to a floating-point value; expands inline if –x is used	9-50
int **atoi**(register const char *st);	Converts a string to an integer	9-50
long **atol**(register const char *st);	Converts a string to a long integer value; expands inline if –x is used	9-50
void ***bsearch**(register const void *key, register 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	9-51
void ***calloc**(size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes	9-51
div_t **div**(register int numer, register int denom);	Divides numer by denom producing a quotient and a remainder	9-56
void **exit**(int status);	Terminates a program normally	9-56
void **free**(void *packet);	Deallocates memory space allocated by malloc, calloc, or realloc	9-62
char ***getenv**(const char *_string)	Returns the environment information for the variable associated with _string	9-65
long **labs**(long i);	Returns the absolute value of i; expands inline unless –x0 is used	9-42
ldiv_t **ldiv**(register long numer, register long denom);	Divides numer by denom	9-56
int **ltoa**(long val, char *buffer);	Converts val to the equivalent string	9-69
void ***malloc**(size_t size);	Allocates memory for an object of size bytes	9-69
void ***memalign**(size_t alignment, size_t size);	Allocates memory for an object of size bytes aligned to an alignment byte boundary	9-69
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc	9-71
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());	Sorts an array of nmemb members; base points to the first member of the unsorted array, and size specifies the size of each member	9-75

(g) General functions (stdlib.h)(Continued)

Function	Description	Page
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX	9-76
void ***realloc**(void *packet, size_t size);	Changes the size of an allocated memory space	9-76
void **srand**(unsigned int seed);	Resets the random number generator	9-76
double **strtod**(const char *st, char **endptr);	Converts a string to a floating-point value	9-93
long **strtol**(const char *st, char **endptr, int base);	Converts a string to a long integer	9-93
unsigned long **strtoul**(const char *st, char **endptr, int base);	Converts a string to an unsigned long integer	9-93

(h) String functions (string.h)

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 if –x is used	9-70
int **memcmp**(const void *cs, const void *ct, size_t n);	Compares the first n characters of cs to ct; expands inline if –x is used	9-70
void ***memcpy**(void *s1, const void *s2, register size_t n);	Copies n characters from s1 to s2	9-70
void ***memmove**(void *s1, const void *s2, size_t n);	Moves n characters from s1 to s2	9-71
void ***memset**(void *mem, register int ch, register size_t length);	Copies the value of ch into the first length characters of mem; expands inline if –x is used	9-71
char ***strcat**(char *string1, const char *string2);	Appends string2 to the end of string1	9-83
char ***strchr**(const char *string, int c);	Finds the first occurrence of character c in s; expands inline if –x is used	9-84
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.	9-84
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.	9-84
char ***strcpy**(register char *dest, register const char *src);	Copies string src into dest; expands inline if –x is used	9-85

(h) String functions (string.h)(Continued)

Function	Description	Page
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	9-86
char ***strerror**(int errno);	Maps the error number in errno to an error message string	9-86
size_t **strlen**(const char *string);	Returns the length of a string	9-88
char ***strncat**(char *dest, const char *src, register size_t n);	Appends up to n characters from src to dest	9-88
int **strncmp**(const char *string1, const char *string2, size_t n);	Compares up to n characters in two strings; expands inline if –x is used	9-89
char ***strncpy**(register char *dest, register const char *src, register size_t n);	Copies up to n characters from src to dest; expands inline if –x is used	9-90
char ***strpbrk**(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs	9-91
char ***strrchr**(const char *string, int c);	Finds the last occurrence of character c in string; expands inline if –x is used	9-91
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	9-92
char ***strstr**(register const char *string1, const char *string2);	Finds the first occurrence of string2 in string1	9-92
char ***strtok**(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2	9-94
size_t **strxfrm**(register char *to, register const char *from, register size_t n);	Transforms n characters from from, to to	9-94

(i) Time-support functions (time.h)

Function	Description	Page
char ***asctime**(const struct tm *timeptr);	Converts a time to a string	9-46
clock_t **clock**(void);	Determines the processor time used	9-53
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	9-65
struct tm *__localtime__(const time_t *timer);	Converts time_t value to broken down time	9-67
time_t **mktime**(register struct tm *tptr);	Converts broken down time to a time_t value	9-72
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *time);	Formats a time into a character string	9-87
time_t **time**(time_t *timer);	Returns the current calendar time	9-95

9.6 Description of Runtime-Support Functions and Macros

This section describes the runtime-support functions and macros. 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.

abort — *Abort*

Syntax

#include <stdlib.h>

void **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 9-56.

abs/labs — *Absolute Value*

Syntax

#include   <stdlib.h>

int **abs**(int i);
long **labs**(long i);

Defined in

abs.c in src

Description

The C compiler supports two 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.

acos/acosf

Arc Cosine

Syntax

#include <math.h>

double **acos**(double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **acosh**(double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **acot**(double x);
float **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

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double acot2(double x, double y);
float 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

```
#define _TI_ENHANCED_MATH_H 1
#include   <math.h>

double acoth(double x);
float 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.

add_device — *Add Device to Device Table*

Syntax

```
#include <stdio.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 *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 associate 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);
}
```

asctime — *Convert Internal Time to String*

Syntax #include <time.h>

char ***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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

asin/asinf

Arc Sine

Syntax

#include <math.h>

double **asin**(double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **asinh**(double x);
float **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

#include <assert.h>

void **assert**(int expr);

Defined in

assert.h 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)
```

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

#include <math.h>

double **atan**(double x);
float **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

#include <math.h>

double **atan2**(double y, double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **atanh**(double y, double x);
float **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

#include <stdlib.h>

int **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 — *Convert String to Number*

Syntax

#include <stdlib.h>

double **atof**(const char *st);
int **atoi**(register const char *st);
long **atol**(register const char *st);

Defined in

atof.c, atoi.c, and atol.c in rts.src

Description

Three 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 *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. 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

```
#include <stdlib.h>

void *bsearch(register const void *key, register 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

```
#include <stdlib.h>

void *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 #include <math.h>

double **ceil**(double x);
float **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 #include <stdio.h>

void **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

#include <time.h>

clock_t **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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

cos/cosf — *Cosine*

Syntax

#include <math.h>

double **cos**(double x);
float **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

#include <math.h>

double **cosh**(double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **cot**(double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **coth**(double x);
float **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

#include <time.h>

char ***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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

difftime

Time Difference

Syntax

#include <time.h>

double **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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

div/ldiv — *Division*

Syntax #include <stdlib.h>

div_t **div**(register int numer, register int denom);
ldiv_t **ldiv**(register long numer, register long denom);

Defined in div.c in rts.src

Description Two 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 int  quot;    /*  quotient    */
   long int  rem;     /* remainder    */
} ldiv_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 #include <stdlib.h>

void **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 9-42).

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

#include <math.h>

double **exp**(double x);
float **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^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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **exp10**(double x);
float **exp10f**(float x);

Defined in

exp10.c and exp10f.c in rts.src

Description

The exp10 and exp10f functions return 10^x, where x is a real number. A range error occurs if the magnitude of x is too large.

exp2/exp2f

Exponential

Syntax

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **exp2**(double x);
float **exp2f**(float x);

Defined in

exp2.c and exp2f.c in rts.src

Description

The exp2 and exp2f functions return 2^x, where x is a real number. A range error occurs if the magnitude of x is too large.

fabs/fabsf *Absolute Value*

Syntax #include <math.h>

double **fabs**(double x);
float **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 #include <stdio.h>

int **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 #include <stdio.h>

int **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 #include <stdio.h>

int **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

#include <stdio.h>

int **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

#include <stdio.h>

int **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

#include <stdio.h>

int **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

#include <stdio.h>

char ***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

#include <math.h>

double **floor**(double x);
float **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

#include <math.h>

double **fmod**(double x, double y);
float **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 #include <stdio.h>

FILE ***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 #include <stdio.h>

int **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 #include <stdio.h>

int **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 #include <stdio.h>

int **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 #include <stdio.h>

size_t **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 #include <stdlib.h>

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 #include <stdio.h>

FILE ***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

#include <math.h>

double **frexp**(double value**,** int *exp);
float **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

#include <stdio.h>

int **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

#include <stdio.h>

int **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

#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

#include <stdio.h>

long **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

#include <stdio.h>

size_t **fwrite**(const void *_ptr, size_t _size, size_t _count, register FILE *_fp);

Defined in

fwrite.c in rtd.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

#include <stdio.h>

int **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

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

int **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

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

char ***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

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

char ***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

```
#include <time.h>
```

struct tm ***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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

isxxx — *Character Typing*

Syntax

```
#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);

Defined in

is*xxx*.c and ctype.c in rts.src
Also defined in ctype.h 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 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

See abs/labs on page 9-42.

ldexp/ldexpf

Multiply by a Power of 2

Syntax

#include <math.h>

double **ldexp**(double x**,** int exp);
float **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

See div/ldiv on page 9-56.

localtime

Local Time

Syntax

#include <time.h>

struct tm ***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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

log/logf

Natural Logarithm

Syntax

#include <math.h>

double **log**(double x);
float **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

#include <math.h>

double **log10**(double x);
float **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **log2**(double x);
float **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 9-80.

ltoa

Convert Long Integer to ASCII

Syntax

no prototype provided

int **ltoa**(long val, char *buffer);

Defined in

ltoa.c in rts.src

Description

The ltoa function is a nonstandard (non-ANSI) 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

#include <stdlib.h>

void ***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

#include <stdlib.h>

void ***memalign**(size_t alignment**,** size_t _size);

Defined in

memory.c in rts.src

Description

The memalign function performs like the ANSI 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

#include <string.h>

void ***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

#include <string.h>

int **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

#include <string.h>

void ***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

#include <string.h>

void ***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

#include <string.h>

void ***memset**(void *mem, register int ch, register 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

no prototype provided

void **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

#include <time.h>

time_t **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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

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

#include <math.h>

double **modf**(double value**,** double *ip);
float **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

#include <stdio.h>

void **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

#include <math.h>

double **pow**(double x, double y);
float **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 $\times$ log(x)) but are faster and more accurate. A domain error occurs if $x = 0$ and $y \leq 0$, 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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **powi**(double x, int y);
float **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 $i \leq 0$, 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

#include <stdio.h>

int **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

#include <stdio.h>

int **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

#include <stdlib.h>

int **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

#include <stdlib.h>

int **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

#include <stdlib.h>

void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) ());

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

#include <stdlib.h>

int **rand**(void);
void **srand**(unsigned int seed);

Defined in rand.c in rts.src

Description

Two 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

#include <stdlib.h>

void ***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

#include <stdlib.h>

int **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

#include <stdlib.h>

int **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

#include <stdlib.h>

int **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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **rsqrt**(double x);
float **rsqrtf**(float x);

Defined in

rsqrt.c and rsqrtf.c in rst.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

#include <stdlib.h>

int **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

#include <stdlib.h>

void **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

#include <setjmp.h>

int **setjmp**(jmp_buf env)
void **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

#include <stdlib.h>

int **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

#include <math.h>

double **sin**(double x);
float **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

#include <math.h>

double **sinh**(double x);
float **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

#include <stdlib.h>

int **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

#include <math.h>

double **sqrt**(double x);
float **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 9-76.

sscanf

Read Stream

Syntax

#include <stdlib.h>

int **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

#include <string.h>

char ***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

#include <string.h>

char ***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

#include <string.h>

int **strcmp**(const char *string1, register const char *string2);
int **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 ANSI 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

#include <string.h>

char ***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

#include <string.h>

size_t **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

#include <string.h>

char ***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 ANSI compatibility.

strftime — *Format Time*

Syntax

```
#include <time.h>

size_t *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)
%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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

strlen — *Find String Length*

Syntax

#include <string.h>

size_t **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

#include <string.h>

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

#include <string.h>

int **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

#include <string.h>

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

#include <string.h>

char ***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

#include <string.h>

char ***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

#include <string.h>

size_t **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

#include <string.h>

char ***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/strtoul

String to Number

Syntax

#include <stdlib.h>

double **strtod**(const char *st**,** char **endptr);
long **strtol**(const char *st**,** char **endptr, int base);
unsigned long **strtoul**(const char *st, char **endptr, int base);

Defined in

strtod.c,
strtol.c, and
strtoul.c in rts.src

Description

Three 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 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 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

#include <string.h>

char ***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

#include <string.h>

size_t **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

#include <math.h>

double **tan**(double x);
float **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

#include <math.h>

double **tanh**(double x);
float **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

#include <time.h>

time_t **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 header declares and defines, see section 9.3.15, *Time Functions (time.h)*, on page 9-22.

> **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

#include <stdlib.h>

FILE ***tmpfile**(void);

Defined in tmpfile.c in rts.src

Description The tmpfile function creates a temporary file.

tmpnam — *Generate Valid Filename*

Syntax

#include <stdlib.h>

char ***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

#include <ctype.h>

char **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

#include <ctype.h>

char **tolower**(int char c);
char **toupper**(int char c);

Defined in

tolower.c in rts.src
toupper.c in rts.src

Description Two 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

#define _TI_ENHANCED_MATH_H 1
#include <math.h>

double **trunc**(double x);
float **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

#include <stdlib.h>

int **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

#include <stdarg.h>

typedef char ***va_list**;
type **va_arg**(va_list**,** _type);
void **va_end**(va_list);
void **va_start**(va_list**,** parmN);

Defined in stdarg.h

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 runtime. 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

#include <stdlib.h>

int **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

#include <stdlib.h>

int **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

#include <stdlib.h>

int **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.

Library-Build Utility

When using the 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 runtime-support libraries, this package includes the source archive, rts.src, which contains all runtime-support functions.

You can build your own runtime-support libraries by using the mk6x utility described in this chapter and the archiver described in the *TMS320C6000 Assembly Language Tools User's Guide*.

The runtime-support libraries that are shipped with the 'C6000 code generation tools are built as follows:

Command	Comment
`mk6x –o –ml rts.src –l rts6201.lib`	base, 'C6201
`mk6x –o –ml –me rts.src –l rts6201e.lib`	base, 'C6201, big endian
`mk6x –o –ml –mv6700 rts.src –l rts6701.lib`	base, 'C6701
`mk6x –o –ml –mv6701 –me rts.src –l rts6701e.lib`	base, 'C6701, big endian

The base option set for every library is optimization level 2 (–o2 option) and global structures and arrays accessed as far data (–ml option).

Topic **Page**

10.1 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 10.3.)

src_arch The name of a source archive file. For each source archive named, mk6x builds an object library according to the runtime 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 shell 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 TMP, C_OPTION, and C_DIR.

10.2 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 shell. 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 runtime-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).

−−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 runtime-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).

10.3 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/shell

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	3-21
–me	Produces object code in big-endian format	2-16
–mg	Allows you to profile optimized code	3-30
–mh*n*	Allows speculative execution	3-10
–mi*n*	Specifies an interrupt threshold value	2-41
–ml*n*	Changes near and far assumptions on four levels (–ml0, –ml1, –ml2, and –ml3)	2-16
–mr*n*	Makes calls to runtime-support functions near (–mr0) or far (–mr1)	2-16
–ms*n*	Controls code size on three levels (–ms0, –ms1, –ms2, and –ms2)	3-14
–mt	Indicates that specific aliasing techniques are *not* used	3-22
–mu	Turns off software pipelining	3-5
–mv*n*	Selects target version	3-12
–mw	Embeds software pipelined loop information in the .asm file	3-5

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)	2-36
–pk	Makes code K&R compatible	7-23
–pr	Enables relaxed mode; ignores strict ANSI violations	7-25
–ps	Enables strict ANSI mode (for C, not K&R C)	7-25

(d) Parser options that control diagnostics

Option	Effect	Page
–pdr	Issues remarks (nonserious warnings)	2-29
–pdv	Provides verbose diagnostics that display the original source with line wrap	2-30
–pdw	Suppresses warning diagnostics (errors are still issued)	2-30

(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	3-2
–o3	Compiles with –o2 optimization + file optimization. Note that mk6x automatically sets –ol0 and –op0.	3-2

(f) Options that control the definition-controlled inline function expansion

Option	Effect	Page
–x0	Disables intrinsic function inlining, the inline keyword, and automatic inlining	2-36
–x1	Disables the inline keyword and automatic inlining	2-36
–x2 (or –x)	Defines _INLINE and invokes optimizer (at –o2 if not specified differently)	2-36

Table 10–1. Summary of Options and Their Effects (Continued)

(g) Option that controls the assembler

Option	Effect	Page
–as	Keeps labels as symbols	2-20

(h) Options that change the default file extensions

Option	Effect	Page
–ea[.]*new extension*	Sets default extension for assembly files	2-18
–eo[.]*new extension*	Sets default extension for object files	2-18

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 shell 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: A sequence of eight adjacent bits operated upon as a unit.

C

C compiler: A software program that translates C source statements into assembly language source statements.

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 configure according to a standard developed by AT&T. These files are relocatable in memory space.

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 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 preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.

interlist utility: A utility that inserts as comments your original C source statements into the assembly language output from the assembler. The C statements are inserted next to the equivalent assembly instructions.

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-ANSI 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.

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.

runtime 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.

runtime-support functions: Standard ANSI functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

runtime-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.

shell program: A utility that lets you compile, assemble, and optionally link in one step. The shell runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

software pipelining: A technique used by the 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 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

Q

R

S

T