TMS320C6000 Optimizing Compiler v 6.1

User's Guide

Literature Number: SPRU187O
May 2008

Contents

List of Figures

List of Tables

Read This First

About This Manual

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

- Compiler
- Assembly optimizer
- Library-build utility
- C++ name demangler
- Object file display utility

The C/C++ compiler accepts C and C++ code conforming to the International Organization for Standardization (ISO) standards for these languages. The compiler supports the 1989 version of the C language and the 1998 version of the C++ language.

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

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`. 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, cruel world\n";
}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in the **bold typeface**, do not enter the brackets themselves. The following is an example of a command that has an optional parameter:

  cl6x [*options*] [*filenames*] [**--run_linker** [*link_options*] [*object files*]]

- 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 --rom_model or --ram_model option:

```
cl6x --run_linker   {--rom_model | --ram_model} filenames [--output_file=name.out]
          --library= libraryname
```

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If it is a required parameter, it is shown starting against the left margin of the box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in column 1.

```
symbol .usect "section name", size in bytes[, alignment]
```

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. This syntax is shown as [, ..., *parameter*].

```
cl6x --run_linker   {--rom_model | --ram_model} filenames [--output_file=name.out]
          --library= libraryname
```

- The TMS320C6200 core is referred to as C6200. The TMS320C6400 core is referred to as C6400. The TMS320C6700 core is referred to as C6700. TMS320C6000 and C6000 can refer to either C6200, C6400, C6400+, C6700, C6700+, or C6740.

Related Documentation

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

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

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

DWARF Debugging Information Format Version 3, DWARF Debugging Information Format Workgroup, Free Standards Group, 2005 (http://dwarfstd.org)

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

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

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

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

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

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

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

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

Tool Interface Standards (TIS) DWARF Debugging Information Format Specification Version 2.0, TIS Committee, 1995

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):

SPRU186 — *TMS320C6000 Assembly Language Tools v 6.1 User's Guide.* 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 TMS320C6000 platform of devices (including the C64x+ and C67x+ generations).

SPRU190 — *TMS320C6000 DSP Peripherals Overview Reference Guide.* Provides an overview and briefly describes the peripherals available on the TMS320C6000 family of digital signal processors (DSPs).

SPRU198 — *TMS320C6000 Programmer's Guide.* Reference for programming the TMS320C6000 digital signal processors (DSPs). Before you use this manual, you should install your code generation and debugging tools. Includes a brief description of the C6000 DSP architecture and code development flow, includes C code examples and discusses optimization methods for the C code, describes the structure of assembly code and includes examples and discusses optimizations for the assembly code, and describes programming considerations for the C64x DSP.

SPRU197 — *TMS320C6000 Technical Brief.* Provides an introduction to the TMS320C62x and TMS320C67x digital signal processors (DSPs) of the TMS320C6000 DSP family. Describes the CPU architecture, peripherals, development tools and third-party support for the C62x and C67x DSPs.

SPRU423 — *TMS320 DSP/BIOS User's Guide.* DSP/BIOS gives developers of mainstream applications on Texas Instruments TMS320 digital signal processors (DSPs) the ability to develop embedded real-time software. DSP/BIOS provides a small firmware real-time library and easy-to-use tools for real-time tracing and analysis.

SPRU731 — *TMS320C62x DSP CPU and Instruction Set Reference Guide.* Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C62x digital signal processors (DSPs) of the TMS320C6000 DSP family. The C62x DSP generation comprises fixed-point devices in the C6000 DSP platform.

SPRU732 — *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.* Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C64x and TMS320C64x+ digital signal processors (DSPs) of the TMS320C6000 DSP family. The C64x/C64x+ DSP generation comprises fixed-point devices in the C6000 DSP platform. The C64x+ DSP is an enhancement of the C64x DSP with added functionality and an expanded instruction set.

SPRU733 — *TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide.* Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C67x and TMS320C67x+ digital signal processors (DSPs) of the TMS320C6000 DSP platform. The C67x/C67x+ DSP generation comprises floating-point devices in the C6000 DSP platform. The C67x+ DSP is an enhancement of the C67x DSP with added functionality and an expanded instruction set.

Trademarks

Windows is a registered trademark of Microsoft Corporation.

UNIX is a registered trademark of licensed exclusively through X/Open Company Limited.

Introduction to the Software Development Tools

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

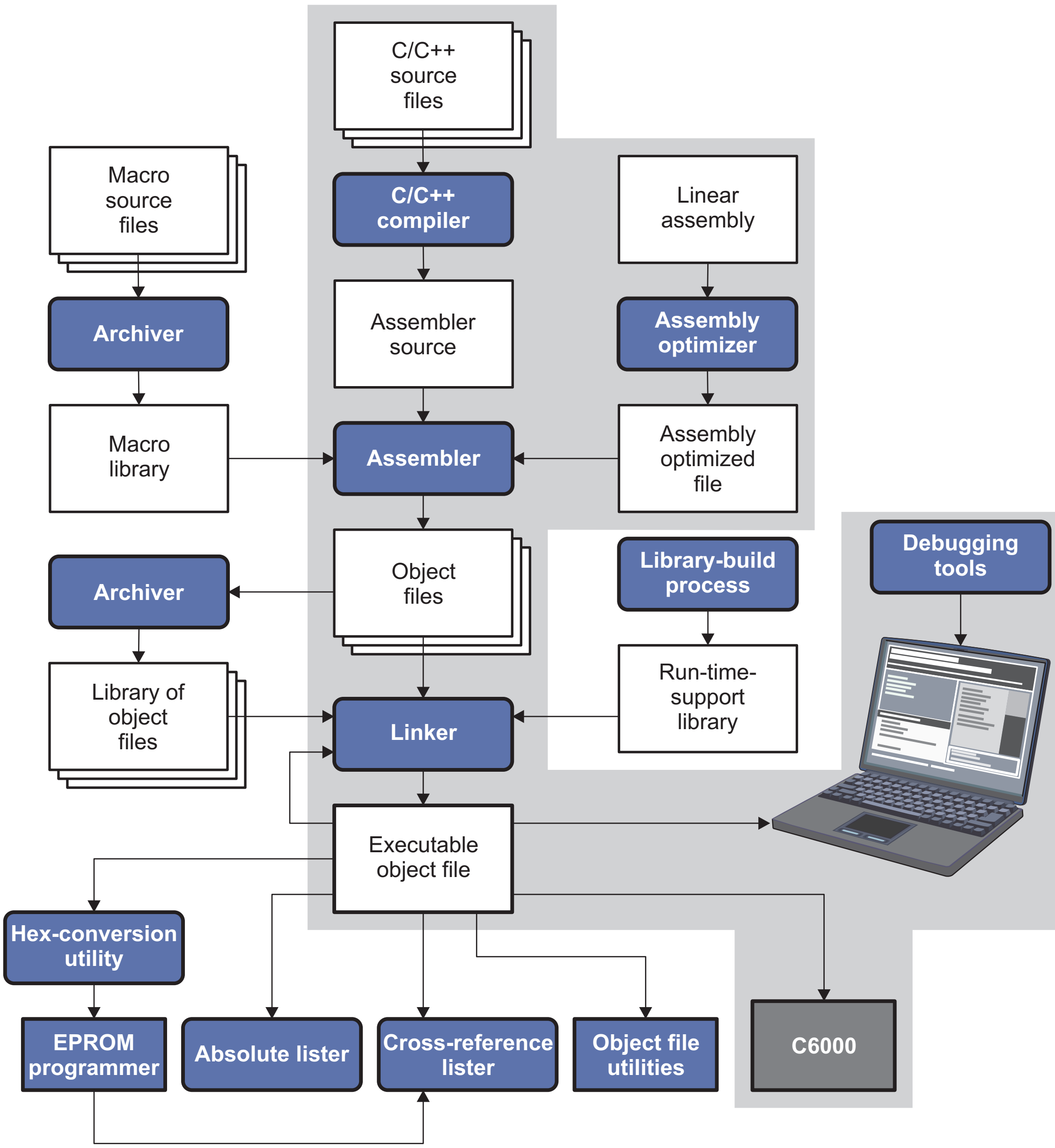
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembly optimizer is discussed in Chapter 4. The assembler and link step are discussed in detail in the *TMS320C6000 Assembly Language Tools User's Guide*.

Topic **Page**

1.1 Software Development Tools Overview

Figure 1-1 illustrates the 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.
- The **compiler** accepts C/C++ source code and produces C6000 assembly language source code. See Chapter 2.
- The **assembler** translates assembly language source files into machine language object files. 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 object files and object libraries as input. See Chapter 5. The *TMS320C6000 Assembly Language Tools User's Guide* provides 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 process** to build your own customized run-time-support library. See Section 8.5. Standard run-time-support library functions for C and C++ are provided in the self-contained rtssrc.zip file.

  The **run-time-support libraries** contain the standard ISO run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler. See Chapter 8.
- The **hex conversion utility** converts an object file into other object formats. 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 and describes all supported formats.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. The *TMS320C6000 Assembly Language Tools User's Guide* explains how to use the absolute lister.
- 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 **C++ name demangler** is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 1-1, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file. See Chapter 9.
- The main product of this development process is a module that can be executed in a **TMS320C6000** device.

1.2 C/C++ Compiler Overview

The following subsections describe the key features of the compiler.

1.2.1 ANSI/ISO Standard

The following features pertain to ISO standards:

- **ISO-standard C**

  The C/C++ compiler fully conforms to the ISO C standard as defined by the ISO specification and described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The ISO C standard supercedes and is the same as the ANSI C standard.
- **ISO-standard C++**

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

C++. For a description of *unsupported* C++ features, see Section 6.2.

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

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

1.2.2 Output Files

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

- **COFF object files**

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

- **EPROM programmer data files**

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

1.2.3 Compiler Interface

The following features pertain to interfacing with the compiler:

- **Compiler program**

  The compiler tools include a compiler program that you use to compile, assembly optimize, assemble, and link programs in a single step. For more information, see Section 2.1.

- **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 7.

1.2.4 Utilities

The following features pertain to the compiler utilities:

- **Library-build process**

  The library-build process lets you custom-build object libraries from source for any combination of run-time models. For more information, see Section 8.5.

- **C++ name demangler**

  The C++ name demangler (dem6x) is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code. For more information, see Chapter 9.

Using the C/C++ Compiler

The compiler translates your source program into code that the TMS320C6000 can execute. Source code must be compiled, assembled, and linked to create an executable object file. All of these steps are executed at once by using the compiler.

2.1 About the Compiler

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

- The **compiler** accepts C/C++ source code and assembly code, and produces object code.

  You can compile C, C++, and assembly files in a single command. The compiler uses the filename extensions to distinguish between different file types. See Section 2.3.8 for more information.
- The **linker** combines object files to create an executable object file. The linker is optional, so you can compile and assemble many modules independently and link them later. See Chapter 5 for information about linking the files.

By default, the compiler does not perform the linker. You can invoke the linker by using the --run_linker compiler option.

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

2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

cl6x [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]]

cl6x	Command that runs the compiler and the assembler.
options	Options that affect the way the compiler processes input files. The options are listed in Table 2-1 through Table 2-12.
filenames	One or more C/C++ source files, assembly language source files, linear assembly files, or object files.
--run_linker	Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 5 for more information.
link_options	Options that control the linking process.
object files	Name of the additional object files for the linking process.

The arguments to the compiler are of three types:

- Compiler options
- Link options
- Filenames

The --run_linker option indicates linking is to be performed. If the --run_linker option is used, any compiler options must precede the --run_linker option, and all other link options must follow the --run_linker option.

Source code filenames must be placed before the --run_linker option. Additional object file filenames can be placed after the --run_linker option.

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

```
cl6x symtab.c file.c seek.asm find.sa --run_linker --library=lnk.cmd
       --library=rts6200.lib --output_file=myprogram.out
```

2.3 Changing the Compiler's Behavior With Options

Options control the operation of the compiler. 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.

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

The following apply to the compiler options:

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

You can define default options for the compiler by using the C6X_C_OPTION environment variable. For a detailed description of the environment variable, see Section 2.4.1.

Table 2-1 through Table 2-12 summarize all options (including link options). Use the references in the tables for more complete descriptions of the options.

Table 2-1. Options That Control the Compiler

Option	Alias	Effect	Section
--c_src_interlist	-ss	Interlists C source and assembly statements	Section 2.14 Section 3.13
--cmd_file=*filename*	-@	Interprets contents of a file as an extension to the command line. Multiple -@ instances can be used.	Section 2.3.1
--compile_only	-c	Disables linking (negates --run_linker)	Section 2.3.1 Section 5.1.3
--compiler_revision		Prints out the compiler release revision and exits	–
--define=*name*[=*def*]	-D	Predefines *name*	Section 2.3.1
--gen_func_subsections	-mo	Puts each function in a separate subsection in the object file	Section 5.3.2
--help	-h	Help	Section 2.3.1
--include_path=*directory*	-I	Defines #include search path	Section 2.3.1 Section 2.6.2.1
--keep_asm	-k	Keeps the assembly language (.asm) file	Section 2.3.1
--preinclude=*filename*		Includes *filename* at the beginning of compilation	Section 2.3.1
--quiet	-q	Suppresses progress messages (quiet)	Section 2.3.1
--run_linker	-z	Enables linking	Section 2.3.1
--skip_assembler	-n	Compiles or assembly optimizes only	Section 2.3.1
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	Section 2.3.1
--undefine=*name*	-U	Undefines *name*	Section 2.3.1
--verbose	-v	Displays a banner and function progress information	Section 2.3.1
--tool_version	-version	Displays version number for each tool	–

Table 2-2. Options That Control Symbolic Debugging and Profiling

Option	Alias	Effect	Section
--profile:breakpt		Enables breakpoint-based profiling	Section 2.3.4 Section 3.14.2
--profle:power		Enables power profiling	Section 2.3.4 Section 3.14.2
--symdebug:coff		Enables symbolic debugging using the alternate STABS debugging format	Section 2.3.4 Section 3.14.1
--symdebug:dwarf	-g	Enables symbolic debugging	Section 2.3.4 Section 3.14.1
--symdebug:dwarf_version=2|3		Specifies the DWARF format version	Section 2.3.4
--symdebug:none		Disables all symbolic debugging	Section 2.3.4
--symdebug:profile_coff		Enables profiling using the alternate STABS debugging format	Section 2.3.4
--symdebug:skeletal		Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	Section 2.3.4

Table 2-3. Options That Change the Default File Extensions

Option	Alias	Effect	Section
--ap_extension=[.]*extension*	-el	Sets a default extension for linear assembly source files	Section 2.3.8
--asm_extension=[.]*extension*	-ea	Sets a default extension for assembly source files	Section 2.3.8
--c_extension=[.]*extension*	-ec	Sets a default extension for C source files	Section 2.3.8
--cpp_extension=[.]*extension*	-ep	Sets a default extension for C++ source files	Section 2.3.8
--listing_extension=[.]*extension*	-es	Sets a default extension for listing files	Section 2.3.8
--obj_extension=[.]*extension*	-eo	Sets a default extension for object files	Section 2.3.8

Table 2-4. Options That Specify Files

Option	Alias	Effect	Section
--ap_file=*filename*	-fl	Identifies *filename* as a linear assembly source file regardless of its extension. By default, the compiler and assembly optimizer treat .sa files as linear assembly source files.	Section 2.3.6
--asm_file=*filename*	-fa	Identifies *filename* as an assembly source file regardless of its extension. By default, the compiler and assembler treat .asm files as assembly source files.	Section 2.3.6
--c_file=*filename*	-fc	Identifies *filename* as a C source file regardless of its extension. By default, the compiler treats .c files as C source files.	Section 2.3.6
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 2.3.6
--cpp_file=*filename*	-fp	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc and .cxx files as a C++ files.	Section 2.3.6
--obj_file=*filename*	-fo	Identifies *filename* as an object code file regardless of its extension. By default, the compiler and linker treat .obj files as object code files.	Section 2.3.6

Table 2-5. Options That Specify Directories

Option	Alias	Effect	Section
--abs_directory=*directory*	-fb	Specifies an absolute listing file directory	Section 2.3.9
--asm_directory=*directory*	-fs	Specifies an assembly file directory	Section 2.3.9
--list_directory=*directory*	-ff	Specifies an assembly listing file and cross-reference listing file directory	Section 2.3.9
--obj_directory=*directory*	-fr	Specifies an object file directory	Section 2.3.9
--temp_directory=*directory*	-ft	Specifies a temporary file directory	Section 2.3.9

Table 2-6. Options That Are Machine-Specific

Option	Alias	Effect	Section
--aliased_variables	-ma	Indicates that a specific aliasing technique is used	Section 3.9.1
--big_endian	-me	Produces object code in big-endian format	Section 2.13
--consultant		Generates compiler Consultant Advice	Section 2.3.2
--debug_software_pipeline	-mw	Produce verbose software pipelining report	Section 3.2.2
--disable_software_pipelining	-mu	Turns off software pipelining	Section 3.2.1
--dprel		Specifies that all non-const data is addressed using DP-relative addressing	Section 7.1.5.2
--entry_hook[=*name*]		Enables entry hooks	Section 2.15
--exit_hook[=*name*]		Enables exit hooks	Section 2.15
--fp_not_associative	-mc	Prevents reordering of associative floating-point operations	Section 3.10
--gen_pic	-mpic	Generates position-independent code for call returns	Section 2.3.2
--gen_profile_info		Generates instrumentation code to collect profile information.	Section 3.8.1.3
--interrupt_threshold	-mi	Specifies an interrupt threshold value	Section 2.12
--mem_model:const=*type*		Allows const objects to be made far independently of the --mem_model:data option	Section 7.1.5.3
--mem_model:data=*type*		Determines data access model	Section 7.1.5.1
--no_bad_aliases	-mt	Allows certain assumptions about aliasing and loops	Section 3.9.2 Section 4.6.2
--opt_for_space=*n*	-ms=*n*	Controls code size on four levels (0, 1, 2, and 3)	Section 3.5
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions	Section 2.3.2
--silicon_version=*n*	-mv=*n*	Selects target version	Section 2.3.3
--speculate_loads=*n*	-mh=*n*	Specifies speculative load byte count threshold. Allows speculative execution of loads with bounded address ranges.	Section 3.2.3.1
--speculate_unknown_loads		Allows speculative execution of loads with unbounded addresses	Section 2.3.2
--target_compatiblity_6200	-mb	Enables C62xx compatibility with C6400 code	Section 2.13
--use_const_for_alias_analysis	-ox	Uses const to disambiguate pointers	Section 2.3.2
--use_profile_info=*file1*[, *file2*,...]		Specifies the profile information file(s)	Section 3.8.1.3

Table 2-7. Options That Control Parsing

Option	Alias	Effect	Section
--create_pch=*filename*		Creates a precompiled header file with the name specified	Section 2.5
--embedded_cpp	-pe	Enables embedded C++ mode	Section 6.11.3
--exceptions		Enables C++ exception handling	Section 2.3.1
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode	Section 2.3.1
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic	Section 2.3.1
--gcc		Enables support for GCC extensions	Section 6.12
--gen_asp_raw	-pl	Generates a raw listing file	Section 2.10
--gen_acp_xref	-px	Generates a cross-reference listing file	Section 2.9
--kr_compatible	-pk	Allows K&R compatibility	Section 6.11.1
--no_inlining	-pi	Disables definition-controlled inlining (but --opt_level=3 (or -O3) optimizations still perform automatic inlining)	Section 2.11
--pch		Creates or uses precompiled header files	Section 2.5
--pch_dir=*directory*		Specifies the path where the precompiled header file resides	Section 2.5
--pch_verbose		Displays a message for each precompiled header file that is considered but not used	Section 2.5
--program_level_compile	-pm	Combines source files to perform program-level optimization	Section 3.7
--relaxed_ansi	-pr	Enables relaxed mode; ignores strict ISO violations	Section 6.11.2
--rtti	-rtti	Enables run time type information (RTTI)	–
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic	Section 2.3.1
--static_template_instantiation		Instantiate all template entities with internal linkage	–
--strict_ansi	-ps	Enables strict ISO mode (for C/C++, not K&R C)	Section 6.11.2
--use_pch=*filename*		Specifies the precompiled header file to use for this compilation	Section 2.5

Table 2-8. Parser Options That Control Preprocessing

Option	Alias	Effect	Section
--preproc_dependency[=*filename*]	-ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility	Section 2.6.7
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	Section 2.6.8
--preproc_macros		Performs preprocessing only. Writes list of predefined and user-defined macros to a file with the same name as the input but with a .pp extension.	Section 2.6.9
--preproc_only	-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension.	Section 2.6.3
--preproc_with_comments	-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.	Section 2.6.5
--preproc_with_compile	-ppa	Continues compilation after preprocessing	Section 2.6.4
--preproc_with_line	-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.	Section 2.6.6

Table 2-9. Parser Options That Control Diagnostics

Option	Alias	Effect	Section
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error	Section 2.7.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark	Section 2.7.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*	Section 2.7.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning	Section 2.7.1
--display_error_number=*num*	-pden	Displays a diagnostic's identifiers along with its text	Section 2.7.1
--issue_remarks	-pdr	Issues remarks (nonserious warnings)	Section 2.7.1
--no_warnings	-pdw	Suppresses warning diagnostics (errors are still issued)	Section 2.7.1
--set_error_limit=*num*	-pdel	Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.)	Section 2.7.1
--verbose_diagnostics	-pdv	Provides verbose diagnostics that display the original source with line-wrap	Section 2.7.1
--write_diagnostics_file[(1)]	-pdf	Generates a diagnostics information file	Section 2.7.1

(1) Parser only option.

Table 2-10. Options That Control Optimization[(1)]

Option	Alias	Effect	Section
--auto_inline=[*size*]	-oi	Sets automatic inlining size (--opt_level=3 only). If *size* is not specified, the default is 1.	Section 3.12
--call_assumptions=0	-op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	Section 3.7.1
--call_assumptions=1	-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	Section 3.7.1
--call_assumptions=2	-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)	Section 3.7.1
--call_assumptions=3	-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	Section 3.7.1
--gen_opt_info=0	-on0	Disables the optimization information file	Section 3.6.2
--gen_opt_info=1	-on1	Produces an optimization information file	Section 3.6.2
--gen_opt_info=2	-on2	Produces a verbose optimization information file	Section 3.6.2
--opt_level=0	-O0	Optimizes register usage	Section 3.1
--opt_level=1	-O1	Uses -O0 optimizations and optimizes locally	Section 3.1
--opt_level=2	-O2 or -O	Uses -O1 optimizations and optimizes globally	Section 3.1
--opt_level=3	-O3	Uses -O2 optimizations and optimizes the file	Section 3.1 Section 3.6
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements	Section 3.13
--single_inline		Inlines functions that are only called once	
--std_lib_func_defined	-ol1 or -oL1	Informs the optimizer that your file declares a standard library function	Section 3.6.1
--std_lib_func_not_defined	-ol2 or -oL2	Informs the optimizer that your file does not declare or alter library functions. Overrides the -ol0 and -ol1 options (default).	Section 3.6.1
--std_lib_func_redefined	-ol0 or -oL0	Informs the optimizer that your file alters a standard library function	Section 3.6.1

(1) **Note:** Machine-specific options (see Table 2-6) can also affect optimization.

Table 2-11. Options That Control the Assembler

Option	Alias	Effect	Section
--absolute_listing	-aa	Enables absolute listing	Section 2.3.10
--asm_define=*name*[=*def*]	-ad	Sets the *name* symbol	Section 2.3.10
--asm_dependency	-apd	Performs preprocessing; lists only assembly dependencies	Section 2.3.10
--asm_includes	-api	Performs preprocessing; lists only included #include files	Section 2.3.10
--asm_listing	-al	Generates an assembly listing file	Section 2.3.10
--asm_undefine=*name*	-au	Undefines the predefined constant *name*	Section 2.3.10
--copy_file=*filename*	-ahc	Copies the specified file for the assembly module	Section 2.3.10
--cross_reference	-ax	Generates the cross-reference file	Section 2.3.10
--include_file=*filename*	-ahi	Includes the specified file for the assembly module	Section 2.3.10
--machine_regs		Displays reg operands as machine registers in assembly code	Section 2.3.10
--no_compress		Prevents compression on C6400+ and C6740	Section 2.3.10
--no_reload_errors		Turns off all reload-related loop buffer error messages for C6400+ and C6740	Section 2.3.10
--output_all_syms	-as	Puts labels in the symbol table	Section 2.3.10
--syms_ignore_case	-ac	Makes case insignificant in assembly source files	Section 2.3.10

Table 2-12. Options That Control the Linker

Option	Alias	Description	Section
--absolute_exe	-a	Generates absolute executable output	Section 5.2
-ar		Generates relocatable, executable output	Section 5.2
--arg_size=*size*	--args	Allocates memory to be used by the loader to pass arguments	Section 5.2
--compress_dwarf		Aggressively reduces the size of DWARF information from input object files	Section 5.2
--define=*name*[=*val*]		Predefines *name* as a preprocessor macro.	Section 5.2
--diag_error=*num*		Categorizes the diagnostic identified by *num* as an error	Section 2.7.1
--diag_remark=*num*		Categorizes the diagnostic identified by *num* as a remark	Section 2.7.1
--diag_suppress=*num*		Suppresses the diagnostic identified by *num*	Section 2.7.1
--diag_warning=*num*		Categorizes the diagnostic identified by *num* as a warning	Section 2.7.1
--disable_auto_rts		Disables the automatic selection of a run-time-support library	Section 5.2
--disable_clink	-j	Disables conditional linking of COFF object modules	Section 5.2
--disable_pp		Disables preprocessing for link command files	Section 5.2
--display_error_number=*num*		Displays a diagnostic's identifiers along with its text	Section 2.7.1
--entry_point=*global_symbol*	-e	Defines an entry point	Section 5.2
--fill_value=*value*	-f	Sets default fill value	Section 5.2
--generate_dead_funcs_list= *filename*		Writes a list of the dead functions that were removed by the linker to *filename*.	Section 5.2
--heap_size=*size*	-heap	Sets heap size (bytes)	Section 5.2
--issue_remarks		Issues remarks (nonserious warnings)	Section 2.7.1
--library=*libraryname*	-l	Supplies library or command filename	Section 5.2
--linker_help	-help	Displays usage information	Section 5.2
--make_global=*global_symbol*	-g	Keeps a *global_symbol* global (overrides -h)	Section 5.2
--make_static	-h	Makes all global symbols static	Section 5.2
--map_file=*filename*	-m	Names the map file	Section 5.2
--mapfile_contents=*filter*[, *filter*]		Controls the information that appears in the map file	Section 5.2
--no_demangle		Disables demangling of symbol names in diagnostics	Section 5.2
--no_sym_merge	-b	Disables merge of COFF symbolic debugging information	Section 5.2
--no_sym_table	-s	Strips symbol table information and line number entries from the output module	Section 5.2
--no_warnings		Suppresses warning diagnostics (errors are still issued)	Section 2.7.1

Table 2-12. Options That Control the Linker (continued)

Option	Alias	Description	Section
--output_file=*filename*	-o	Names the output file.	Section 5.2
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol	Section 5.2
--ram_model	-cr	Initializes variables at load time	Section 5.2 Section 7.8.5
--relocatable	-r	Produces nonexecutable, relocatable output	Section 5.2
--reread_libs	-x	Forces rereading of libraries	Section 5.2
--rom_model	-c	Autoinitializes variables at run time	Section 5.2 Section 7.8.5
--run_abs	-abs	Produces an absolute listing file	Section 5.2
--scan_libraries		Scans all libraries for duplicate symbol definitions	Section 5.2
--search_path=*directory*	`-I`	Defines library search path	Section 5.2
--set_error_limit=*num*		Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.)	Section 2.7.1
--stack_size=*size*	-stack	Sets stack size (bytes)	Section 5.2
--strict_compatibility		Performs more conservative and rigorous compatibility checking of input object files	Section 5.2
--symbol_map=*refname*=*defname*		Enables symbol mapping	Section 5.2
--trampolines		Generates trampoline code sections	Section 5.2
--undef_sym	-u	Creates unresolved external symbol	Section 5.2
--undefine=*name*[=*val*]		Removes the preprocessor macro *name*	Section 5.2
--verbose_diagnostics		Provides verbose diagnostics that display the original source with line-wrap	Section 2.7.1
--xml_link_info		Generates an XML information file	Section 5.2

2.3.1 Frequently Used Options

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

--c_src_interlist

Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. The interlisted C statements may appear to be out of sequence. You can use the interlist feature with the optimizer by combining the --optimizer_interlist and --c_src_interlist options. See Section 3.13. The --c_src_interlist option can have a negative performance and/or code size impact.

--cmd_file=*filename*

Appends the contents of a file to the option set. 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 */. To specify options, surround hyphens with quotation marks. For example, "--"quiet.

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

```
cl6x --cmd_file=file1 --cmd_file=file2 file3
```

--compile_only

Suppresses the linker and overrides the --run_linker option, which specifies linking. The --compile_only option's short form is -c. Use this option when you have --run_linker specified in the C6X_C_OPTION environment variable and you do not want to link. See Section 5.1.3.

--define_name=*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. The --define_name option's short form is -D.

If you want to define a quoted string and keep the quotation marks, do one of the following:

- For Windows®, use --define_name=name="\"string def\"". For example, --define_name=car="\"sedan\""
- For UNIX®, use --define_name=name='"string def"'. For example, --define_name=car='"sedan"'
- For Code Composer Studio, enter the definition in a file and include that file with the --cmd_file option.

--exceptions

Enables support of C++ exception handling. The compiler will generate code to handle try/catch/throw statements in C++ code. See Section 6.5.

--fp_mode={relaxed|strict}

Supports relaxed floating-point mode. In this mode, if the result of a double-precision floating-point expression is assigned to a single-precision floating-point or an integer, the computations in the expression are converted to single-precision computations. Any double-precision constants in the expression are also converted to single-precision if they can be correctly represented as single-precision constants. This behavior does not conform with ISO; but it results in faster code, with some loss in accuracy. In the following example, where N is a number, iN=integer variable, fN=float variable, dN=double variable:

```
il = f1 + f2 * 5.0 -> +, * are float, 5.0 is converted to 5.0f
il = d1 + d2 * d3 -> +, are float
f1 = f2 + f3 * 1.1; -> +, are float, 1.1 is converted to 1
```

To enable relaxed floating-point mode use the --fp_mode=relaxed option, which also sets --fp_reassoc=on. To disable relaxed floating-point mode use the --fp_mode=strict option, which also sets --fp_reassoc=off. The default behavior is --fp_mode=strict.

If --strict_ansi is specified, --fp_mode=strict is set automatically. You can enable the relaxed floating-point mode with strict ansi mode by specifying --fp_mode=relaxed after --strict_ansi.

--fp_reassoc={on|off}

Enables or disables the reassociation of floating-point arithmetic. If --fp_mode=relaxed is specified, --fp_reassoc=on is set automatically. If --strict_ansi is set, --fp_reassoc=off is set since reassociation of floating-point arithmetic is an ANSI violation.

--help

Displays the syntax for invoking the compiler and lists available options. If the --help option is followed by another option or phrase detailed information about the option or phrase is displayed. For example, to see information about debugging options use --help debug.

--include_path=*directory*

Adds *directory* to the list of directories that the compiler searches for #include files. The --include_path option's short form is -I. You can use this option several times to define several directories; be sure to separate the --include_path options with spaces. If you do not specify a directory name, the preprocessor ignores the --include_path option. See Section 2.6.2.1.

--keep_asm	Retains the assembly language output from the compiler or assembly optimizer. Normally, the compiler deletes the output assembly language file after assembly is complete. The --keep_asm option's short form is -k.
--preinclude=*filename*	Includes the source code of *filename* at the beginning of the compilation. This can be used to establish standard macro definitions. The filename is searched for in the directories on the include search list. The files are processed in the order in which they were specified.
--quiet	Suppresses banners and progress information from all the tools. Only source filenames and error messages are output. The --quiet option's short form is -q.
--run_linker	Runs the linker on the specified object files. The --run_linker option and its parameters follow all other options on the command line. All arguments that follow --run_linker are passed to the linker. The --run_linker option's short form is -z. See Section 5.1.
--sat_reassoc={on|off}	Enables or disables the reassociation of saturating arithmetic.
--skip_assembler	Compiles only. The specified source files are compiled but not assembled or linked. The --skip_assembler option's short form is -n. This option overrides --run_linker. The output is assembly language output from the compiler.
--src_interlist	Invokes the interlist feature, which interweaves optimizer comments *or* C/C++ source with assembly source. If the optimizer is invoked (--opt_level=*n* option), optimizer comments are interlisted with the assembly language output of the compiler, which may rearrange code significantly. If the optimizer is not invoked, C/C++ source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C/C++ statement. The --src_interlist option implies the --keep_asm option. The --src_interlist option's short form is -s.
--tool_version	Prints the version number for each tool in the compiler. No compiling occurs.
--undefine_name=*name*	Undefines the predefined constant *name*. This option overrides any --define_name options for the specified constant. The --undefine_name option's short form is -U.
--verbose	Displays progress information and toolset version while compiling. Resets the --quiet option.

2.3.2 Machine-Specific Options

These options are specific to the TMS302C6000 toolset. Please see the referenced sections for more information.

--aliased_variables	Indicates that a specific aliasing technique is used with optimizations. See Section 3.9.1.
--big_endian	Produces code in big-endian format. By default, little-endian code is produced.
--consultant	Generates compile-time loop information through the Compiler Consultant Advice tool. See the *TMS320C6000 Code Composer Studio Online Help* for more information about the Compiler Consultant Advice tool.
--debug_software_pipeline	Produces verbose software pipelining report. See Section 3.2.2.
--disable_software_pipelining	Turns off software pipelining. See Section 3.2.1.
--dprel	Specifies all non-const data, including far data, is addressed using DP-relative addressing. See Section 7.1.5.2
--entry_hook[=*name*]	Enables entry hooks. The hook function is called by the optional *name*. Otherwise, the default entry hook function is named __entry_hook. See Section 2.15.
--exit_hook[=*name*]	Enables exit hooks. The hook function is called by the optional *name*. Otherwise, the default exit hook function is named __exit_hook. See Section 2.15.
--fp_not_associative	Compiler does not reorder floating-point operations. See Section 3.10.
--gen_profile_info	Compiler adds instrumentation code to collect profile information. See Section 3.8.1.3.
--interrupt_threshold=*n*	Specifies an interrupt threshold value *n* that sets the maximum cycles the compiler can disable interrupts. See Section 2.12.
--mem_model:const=*type*	Allows const objects to be made far independently of the --mem_model:data option. The *type* can be data, far, or far_aggregates. See Section 7.1.5.3
--mem_model:data=*type*	Specifies data access model as *type* far, far_aggregates, or near. Default is far_aggregates. See Section 7.1.5.1.
--gen_pic	Generates position-independent code for call returns.
--no_bad_aliases	Allows the compiler to make certain assumptions about aliasing and loops. See Section 3.9.2.
--opt_for_space	Adjusts the compiler priorities between performance and code size. The --opt_for_space=0, --opt_for_space=1, --opt_for_space=2 and --opt_for_space=3 options increasingly favor code size over performance. See Section 3.5.
--remove_hooks_when_inlining	Removes entry/exit hooks for functions that are auto-inlined by the optimizer.
--silicon_version=*num*	Selects the target CPU version. See Section 2.3.3.
--speculate_loads=*n*	Specifies speculative load byte count threshold. Allows speculative execution of loads with bounded addresses. See Section 3.2.3.1.
--speculate_unknown_loads	Allows speculative execution of loads with bounded addresses.

--target_compatibility_6200	Compiles C6400 code that is compatible with array alignment restrictions of version 4.0 tools or C6200/C6700 object code. See Section 2.13
--use_const_for_alias_analysis	Uses const to disambiguate pointers.
--use_profile_info	Specifies the profile information files to use when performing feedback directed optimization. See Section 3.8.1.3.

2.3.3 Selecting Target CPU Version (--silicon_version 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 --silicon_version=6701 or --silicon_version=6412. Alternatively, you can also specify the family of the part, for example, --silicon_version=6400 or --silicon_version=6700. If this option is not used, the compiler generates code for the C6200 parts. If the --silicon_version option is not specified, the code generated runs on all C6000 parts; however, the compiler does not take advantage of target-specific instructions or alignment.

2.3.4 Symbolic Debugging and Profiling Options

The following options are used to select symbolic debugging or profiling:

--profile:breakpt	Disables optimizations that would cause incorrect behavior when using a breakpoint-based profiler.
--profile:power	Enables power profiling support by inserting NOPs into the frame code. These NOPs can then be instrumented by the power profiling tooling to track the power usage of functions. If the power profiling tool is not used, this option increases the cycle count of each function because of the NOPs. The --profile:power option also disables optimizations that cannot be handled by the power-profiler.
--symdebug:coff	Enables symbolic debugging using the alternate STABS debugging format. This may be necessary to allow debugging with older debuggers or custom tools, which do not read the DWARF format. STABS format is not supported for C6400+.
--symdebug:dwarf	Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The --symdebug:dwarf option's short form is -g. The --symdebug_dwarf option disables many code generator optimizations, because they disrupt the debugger. You can use the --symdebug_dwarf option with the --opt_level (aliased as -O) option to maximize the amount of optimization that is compatible with debugging (see Section 3.14.1). For more information on the DWARF debug format, see *The DWARF Debugging Standard*.
--symdebug:dwarf_version={2|3}	Specifies the DWARF debugging format version (2 or 3) to be generated when --symdebug:dwarf or --symdebug:skeletal is specified. For more information on TI extensions to the DWARF language, see *The Impact of DWARF on TI Object Files* (SPRAAB5).
--symdebug:none	Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.

--symdebug:profile_coff	Adds the necessary debug directives to the object file which are needed by the profiler to allow function level profiling with minimal impact on optimization (when used). Using --symdebug:coff may hinder some optimizations to ensure that debug ability is maintained, while this option will not hinder optimization. You can set breakpoints and profile on function-level boundaries in Code Composer Studio, but you cannot single-step through code as with full debug ability.
--symdebug:skeletal	Generates as much symbolic debugging information as possible without hindering optimization. Generally, this consists of global-scope information only. This option reflects the default behavior of the compiler.

See Section 2.3.11 for a list of deprecated symbolic debugging options.

2.3.5 Specifying Filenames

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

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

Note: Case Sensitivity in Filename Extensions

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

For information about how you can alter the way that the compiler interprets individual filenames, see Section 2.3.6. For information about how you can alter the way that the compiler interprets and names the extensions of assembly source and object files, see Section 2.3.9.

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

```
cl6x *.cpp
```

Note: No Default Extension for Source Files is Assumed

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

2.3.6 Changing How the Compiler Interprets Filenames

You can use options to change how the compiler interprets your filenames. If the extensions that you use are different from those recognized by the compiler, you can use the filename 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:

--ap_file=*filename*	for a linear assembly file
--asm_file=*filename*	for an assembly language source file
--c_file=*filename*	for a C source file
--cpp_file=*filename*	for a C++ source file
--obj_file=*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 --asm_file and --c_file options to force the correct interpretation:

```
cl6x --c_file=file.s --asm_file=assy
```

You cannot use the filename options with wildcard specifications.

2.3.7 Changing How the Compiler Processes C Files

The --cpp_default option causes the compiler to process C files as C++ files. By default, the compiler treats files with a .c extension as C files. See Section 2.3.8 for more information about filename extension conventions.

2.3.8 Changing How the Compiler Interprets and Names Extensions

You can use options to change how the compiler program interprets filename extensions and names the extensions of the files that it creates. The filename extension 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:

--ap_extension=*new extension*	for a linear assembly source file
--asm_extension=*new extension*	for an assembly language file
--c_extension=*new extension*	for a C source file
--cpp_extension=*new extension*	for a C++ source file
--listing_extension=*new extension*	sets default extension for listing files
--obj_extension=*new extension*	for an object file

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

```
cl6x --asm_extension=.rrr --obj_extension=.o fit.rrr
```

The period (.) in the extension is optional. You can also write the example above as:

```
cl6x --asm_extension=rrr --obj_extension=o fit.rrr
```

2.3.9 Specifying Directories

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

--abs_directory=*directory* Specifies the destination directory for absolute listing files. The default is to use the same directory as the object file directory. For example:

```
cl6x --abs_directory=d:\abso_list
```

--asm_directory=*directory* Specifies a directory for assembly files. For example:

```
cl6x --asm_directory=d:\assembly
```

--list_directory=*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. For example:

```
cl6x --list_directory=d:\listing
```

--obj_directory=*directory* Specifies a directory for object files. For example:

```
cl6x --obj_directory=d:\object
```

--temp_directory=*directory* Specifies a directory for temporary intermediate files. For example:

```
cl6x --temp_directory=c:\temp
```

2.3.10 Assembler Options

Following are assembler options that you can use with the compiler. For more information, see the *TMS320C6000 Assembly Language Tools User's Guide.*

--absolute_listing Generates a listing with absolute addresses rather than section-relative offsets.

--asm_define=*name*[=*def*] Predefines the constant *name* for the assembler; produces a .set directive for a constant or a .arg directive for a string. If the optional [=*def*] is omitted, the *name* is set to 1. If you want to define a quoted string and keep the quotation marks, do one of the following:

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

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

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

--asm_listing Produces an assembly listing file.

--asm_undefine=*name* Undefines the predefined constant *name*. This option overrides any --asm_define options for the specified constant.

--copy_file=*filename*	Copies the specified file for the assembly module; acts like a .copy directive. The file is inserted before source file statements. The copied file appears in the assembly listing files.
--cross_reference	Produces a symbolic cross-reference in the listing file.
--include_file=*filename*	Includes the specified file for the assembly module; acts like a .include directive. The file is included before source file statements. The included file does not appear in the assembly listing files.
--machine_regs	Displays reg operands as machine registers in the assembly file for debugging purposes.
--no_compress	Prevents compression in the assembler. For C6400+ and C6740, compression is the changing of 32-bit instructions to 16-bit instructions, where possible/profitable.
--no_reload_errors	Turns off all reload-related loop buffer error messages in assembly code for C6400+ and C6740.
--output_all_syms	Puts labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.
--syms_ignore_case	Makes letter case insignificant in the assembly language source files. For example, --syms_ignore_case makes the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (this is the default).

2.3.11 Deprecated Options

Several compiler options have been deprecated. The compiler continues to accept these options, but they are not recommended for use. Future releases of the tools will not support these options. Table 2-13 lists the deprecated options and the options that have replaced them.

Table 2-13. Compiler Backwards-Compatibility Options Summary

Old Option	Effect	New Option
-gp	Allows function-level profiling of optimized code	--symdebug:dwarf or -g
-gt	Enables symbolic debugging using the alternate STABS debugging format	--symdebug:coff
-gw	Enables symbolic debugging using the DWARF debugging format	--symdebug:dwarf or -g
-ml	Changes near and far assumptions on four levels	--mem_model:data=*near*, *far*, or *far_aggregate*
-mr	Makes calls to run-time-support functions near or far	

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

Since C6400+ and C6740 produce only DWARF debug information, the -gp, -gt/--symdebug:coff, and --symdebug:profile_coff options are not supported for C6400+ and C6740.

2.4 Controlling the Compiler Through Environment Variables

An environment variable is a system symbol that you define and assign a string to. Setting environment variables is useful when you want to run the compiler repeatedly without re-entering options, input filenames, or pathnames.

Note: C_OPTION and C_DIR

The C_OPTION and C_DIR environment variables are deprecated. Use the device-specific environment variables instead.

2.4.1 Setting Default Compiler Options (C6X_C_OPTION)

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

Setting the default options with these environment variables is useful when you want to run the compiler consecutive times with the same set of options and/or input files. After the compiler reads the command line and the input filenames, it looks for the C6X_C_OPTION environment variable and processes it.

The table below shows how to set the C6X_C_OPTION environment variable. Select the command for your operating system:

Operating System	Enter
UNIX (Bourne shell)	**C6X_C_OPTION="***option*$_1$[*option*$_2$ . . .]"**; export C6X_C_OPTION**
Windows	**set C6X_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 --quiet option), enable C/C++ source interlisting (the --src_interlist option), and link (the --run_linker option) for Windows, set up the C6X_C_OPTION environment variable as follows:

```
set C6X_C_OPTION=--quiet --src_interlist --run_linker
```

In the following examples, each time you run the compiler, it runs the linker. Any options following --run_linker on the command line or in C6X_C_OPTION are passed to the linker. Thus, you can use the C6X_C_OPTION environment variable to specify default compiler and linker options and then specify additional compiler and linker options on the command line. If you have set --run_linker in the environment variable and want to compile only, use the compiler --compile_only option. These additional examples assume C6X_C_OPTION is set as shown above:

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

For details on compiler options, see Section 2.3. For details on linker options, see Section 5.2.

2.4.2 Naming an Alternate Directory (C6X_C_DIR)

The linker uses the C6X_C_DIR environment variable to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

Operating System	Enter
UNIX (Bourne shell)	**C6X_C_DIR="** *pathname$_1$* **;** *pathname$_2$* **;...";** **export C6X_C_DIR**
Windows	**set C6X_C_DIR=***pathname$_1$***;***pathname$_2$***;...**

The *pathnames* are directories that contain input files. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.
- Spaces or tabs at the beginning or end of a path are ignored. For example, the space before and after the semicolon in the following is ignored:

  ```
  set C6X_C_DIR=c:\path\one\to\tools ; c:\path\two\to\tools
  ```
- Spaces and tabs are allowed within paths to accommodate Windows directories that contain spaces. For example, the pathnames in the following are valid:

  ```
  set C6X_C_DIR=c:\first path\to\tools;d:\second path\to\tools
  ```

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
UNIX (Bourne shell)	`unset C6X_C_DIR`
Windows	`set C6X_C_DIR=`

2.5 Precompiled Header Support

Precompiled header files may reduce the compile time for applications whose source files share a common set of headers, or a single file which has a large set of header files. Using precompiled headers, some recompilation is avoided thus saving compilation time.

There are two ways to use precompiled header files. One is the automatic precompiled header file processing and the other is called the manual precompiled header file processing.

2.5.1 Automatic Precompiled Header

The option to turn on automatic precompiled header processing is: --pch. Under this option, the compile step takes a snapshot of all the code prior to the header stop point, and dump it out to a file with suffix .pch. This snapshot does not have to be recompiled in the future compilations of this file or compilations of files with the same header files.

The stop point typically is the first token in the primary source file that does not belong to a preprocessing directive. For example, in the following the stopping point is before int i:

```
#include "x.h"
#include "y.h"
int i
```

Carefully organizing the include directives across multiple files so that their header files maximize common usage can increase the compile time savings when using precompiled headers.

A precompiled header file is produced only if the header stop point and the code prior to it meet certain requirements.

2.5.2 Manual Precompiled Header

You can manually control the creation and use of precompiled headers by using several command line options. You specify a precompiled header file with a specific filename as follows:

--create_pch=*filename*

The --use_pch=*filename* option specifies that the indicated precompiled header file should be used for this compilation. If this precompiled header file is invalid, if its prefix does not match the prefix for the current primary source file for example, a warning is issued and the header file is not used.

If --create_pch=*filename* or --use_pch=*filename* is used with --pch_dir, the indicated filename, which can be a path name, is tacked on to the directory name, unless the filename is an absolute path name.

The --create_pch, --use_pch, and --pch options cannot be used together. If more than one of these options is specified, only the last one is applied. In manual mode, the header stop points are determined in the same way as in automatic mode. The precompiled header file applicability is determined in the same manner.

2.5.3 Additional Precompiled Header Options

The --pch_verbose option displays a message for each precompiled header file that is considered but not used. The --pch_dir=*pathname* option specifies the path where the precompiled header file resides.

2.6 Controlling the Preprocessor

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

- Macro definitions and expansions
- #include files
- Conditional compilation
- Various 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.

2.6.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 2-14.

Table 2-14. C6000 Predefined Macro Names

Macro Name	Description
_BIG_ENDIAN	Defined if big-endian mode is selected (the --big_endian option is used); otherwise, it is undefined
__DATE__ [(1)]	Expands to the compilation date in the form *mmm dd yyyy*
__FILE__ [(1)]	Expands to the current source filename
_INLINE	Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise. Regardless of any optimization, always undefined when --no_inlining is used.
__LINE__ [(1)]	Expands to the current line number
_LITTLE_ENDIAN	Defined if little-endian mode is selected (the --big_endian option is not used); otherwise, it is undefined

(1) Specified by the ISO standard

Table 2-14. C6000 Predefined Macro Names (continued)

Macro Name	Description
__TI_COMPILER_VERSION__	Defined to a 7-digit integer that takes the 3-digit release version number X.Y.Z and generates an integer XXXYYYZZZ where each portion X, Y and Z is expanded to three digits and concatenated together. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.
__TIME__ [(1)]	Expands to the compilation time in the form *hh:mm:ss*
_TMS320C6X	Always defined
_TMS320C6200	Defined if target is C6200
_TMS320C6400	Defined if target is C6400, C6400+, or C6740
_TMS320C6400_PLUS	Defined if target is C6400+ or C6740
_TMS320C6700	Defined if target is C6700, C6700+, or C6740
_TMS320C6700_PLUS	Defined if target is C6700+ or C6740
_TMS320C6740	Defined if target is C6740
__STDC__ [(1)]	Defined to indicate that compiler conforms to ISO C Standard. See Section 6.1 for exceptions to ISO C conformance.

You can use the names listed in Table 2-14 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");
```

2.6.2 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 of the file that contains the #include directive and in the directories of any files that contain that file.
  2. Directories named with the --include_path option.
  3. Directories set with the C6X_C_DIR environment variable.
- If you enclose the filename in angle brackets (< >), the compiler searches for the file in the following directories in this order:
  1. Directories named with the --include_path option.
  2. Directories set with the C6X_C_DIR environment variable.

See Section 2.6.2.1 for information on using the --include_path option. See Section 2.4.2 for more information on input file directories.

2.6.2.1 Changing the #include File Search Path (--include_path Option)

The --include_path option names an alternate directory that contains #include files. The --include_path option's short form is -I. The format of the --include_path option is:

--include_path=*directory1* [**--include_path=***directory2* ...]

There is no limit to the number of --include_path options per invocation of the compiler; each --include_path 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 --include_path 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	/tools/files/alt.h
Windows	c:\tools\files\alt.h

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

Operating System	Enter
UNIX	`cl6x --include_path=tools/files source.c`
Windows	`cl6x --include_path=c:\tools\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 --include_path options and the C6X_C_DIR environment variable.

For example, if you set up C6X_C_DIR with the following command:

```
C6X_C_DIR "/usr/include;/usr/ucb"; export C_DIR
```

or invoke the compiler with the following command:

```
cl6x --include_path=/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
```

2.6.3 Generating a Preprocessed Listing File (--preproc_only Option)

The --preproc_only 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.

2.6.4 Continuing Compilation After Preprocessing (--preproc_with_compile Option)

If you are preprocessing, the preprocessor performs preprocessing only; it does not compile your source code. To override this feature and continue to compile after your source code is preprocessed, use the --preproc_with_compile option along with the other preprocessing options. For example, use --preproc_with_compile with --preproc_only to perform preprocessing, write preprocessed output to a file with a .pp extension, and compile your source code.

2.6.5 Generating a Preprocessed Listing File With Comments (--preproc_with_comments Option)

The --preproc_with_comments 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 --preproc_with_comments option instead of the --preproc_only option if you want to keep the comments.

2.6.6 Generating a Preprocessed Listing File With Line-Control Information (--preproc_with_line Option)

By default, the preprocessed output file contains no preprocessor directives. To include the #line directives, use the --preproc_with_line option. The --preproc_with_line option performs preprocessing only and writes preprocessed output with line-control information (#line directives) to a file named as the source file but with a .pp extension.

2.6.7 Generating Preprocessed Output for a Make Utility (--preproc_dependency Option)

The --preproc_dependency option performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension.

2.6.8 Generating a List of Files Included With the #include Directive (--preproc_includes Option)

The --preproc_includes option performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension.

2.6.9 Generating a List of Macros in a File (--preproc_macros Option)

The --preproc_macros option generates a list of all predefined and user-defined macros. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension. Predefined macros are listed first and indicated by the comment /* Predefined */. User-defined macros are listed next and indicated by the source filename.

2.7 Understanding Diagnostic Messages

One of the compiler's primary functions is to report diagnostics for the source program. The new linker also reports diagnostics. When the compiler or linker 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 diagnostic message severity (severity category descriptions follow)
diagnostic message	The text that describes the problem

Diagnostic messages have an associated severity, as follows:

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

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

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

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

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

Long messages are wrapped to additional lines, when necessary.

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

```
"Test_name.c", line 7: error #64-D: declaration does not declare anything
    struct {};
      ^
"Test_name.c", line 9: error #77: this declaration has no storage class or type specifier
    xxxxx;
    ^
```

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

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

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

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

```
"test.c", line 7: error: "A::A()" is inaccessible
   B x;
   ^
          detected during implicit generation of "B::B()" at line 7
```

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

2.7.1 Controlling Diagnostics

The C/C++ compiler provides diagnostic options to modify how the parser interprets your code. These options are used by the linker to control linker-generated diagnostics. The diagnostic options must be specified before the --run_linker option.

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

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

--diag_suppress=*num* — Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_suppress=*num* to suppress the diagnostic. You can only suppress discretionary diagnostics.

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

--display_error_number — 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 (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). 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 2.7.

--issue_remarks — Issues remarks (nonserious warnings), which are suppressed by default.

--no_warnings — Suppresses warning diagnostics (errors are still issued).

--set_error_limit=*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.)

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

--write_diagnostics_file — Produces a diagnostics information file with the same source file name with an *.err* extension. (The --write_diagnostics_file option is not supported by the linker.)

2.7.2 How You Can Use Diagnostic Suppression Options

The following example demonstrates how you can control diagnostic messages issued by the compiler. You control the linker diagnostic messages in a similar manner.

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

If you invoke the compiler with the --quiet 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 --display_error_number 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 --diag_remark 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.

2.8 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.

2.9 Generating Cross-Reference Listing Information (--gen_acp_xref Option)

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

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

sym-id name X filename line number column number

sym-id	An integer uniquely assigned to each identifier	
name	The identifier name	
X	One of the following values:	
	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	

2.10 Generating a Raw Listing File (--gen_acp_raw Option)

The --gen_acp_raw 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 --preproc_only, --preproc_with_comment, --preproc_with_line, and --preproc_dependency 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 an *.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 2-15.

Table 2-15. 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: 1 = entry into an include file 2 = exit from an include file

The --gen_acp_raw option also includes diagnostic identifiers as defined in Table 2-16.

Table 2-16. 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 2-16 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 2.7.

2.11 Using Inline Function Expansion

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

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

There are several types of inline function expansion:

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

---

Note: Function Inlining Can Greatly Increase Code Size

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. If your code size seems too large, see Section 3.5.

2.11.1 Inlining Intrinsic Operators

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

For details about intrinsics, and a list of the intrinsics, see Section 7.5.4.

2.11.2 Automatic Inlining

When optimizing with the --opt_level=3 or --opt_level=2 option (aliased as -O3 or -O2), the compiler automatically inlines certain functions. For more information, see Section 3.12.

2.11.3 Unguarded Definition-Controlled Inlining

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

You must invoke the optimizer with any --opt_level option (--opt_level=0, --opt_level=1, --opt_level=2, or --opt_level=3) to turn on definition-controlled inlining. Automatic inlining is also turned on when using --opt_level=3.

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

Example 2-1 shows usage of the inline keyword, where the function call is replaced by the code in the called function.

Example 2-1. Using the Inline Keyword

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

2.11.4 Guarded Inlining and the _INLINE Preprocessor Symbol

When declaring a function in a header file as static inline, you must follow additional procedures to avoid a potential code size increase when inlining is turned off with --no_inlining or the optimizer is not run.

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

- Prototype a static inline version of the function. Then, prototype an alternative, nonstatic, externally-linked version of the function. Conditionally preprocess these two prototypes with the _INLINE preprocessor symbol, as shown in Example 2-2.
- Create an identical version of the function definition in a .c or .cpp file, as shown in Example 2-3.

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

The second definition (see Example 2-3) for the library, ensures that the callable version of strlen exists when inlining is disabled. Since this is not an inline function, the _INLINE preprocessor symbol is undefined (#undef) before string.h is included to generate a noninline version of strlen's prototype.

Example 2-2. Header File string.h

```
/*****************************************************************************/
/* string.h  vx.xx   (Excerpted)                                             */
/* Copyright (c) 1993-1999 Texas Instruments Incorporated                    */
/*****************************************************************************/
#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 2-3. Library Definition File strlen.c

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

#include <string.h>
{
_CODE_ACCESS size_t strlen(cont char * string)
   size_t      n = (size_t)-1;
   const char *s = string - 1;

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

2.11.5 Inlining Restrictions

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

At a given call site, a function may be disqualified from inlining if it:

- Is not defined in the current compilation unit
- Never returns
- Is recursive
- Has a FUNC_CANNOT_INLINE pragma
- Has a variable length argument list
- Has a different number of arguments than the call site
- Has an argument whose type is incompatible with the corresponding call site argument
- Has a structure or union parameter
- Contains a volatile local variable or argument
- Is not declared inline and contains an asm() statement that is not a comment
- Is not declared inline and it is main()
- Is not declared inline and it is an interrupt function
- Is not declared inline and returns void but its return value is needed.
- Is not declared inline and will require too much stack space for local array or structure variables.

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

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

The --interrupt_threshold=*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 6.8.8 or the *TMS320C6000 Programmer's Guide* for more information.

RTS Library Files Are Not Built With the --interrupt_threshold Option

Note: The run-time-support library files provided with the compiler are not built with the interrupt flexibility option. Please refer to the readme file to see how the run-time-support library files were built for your release. See Section 8.5 to build your own run-time-support library files with the interrupt flexibility option.

Special Cases With the --interrupt_threshold Option

Note: The --interrupt_threshold=0 option generates the same code to disable interrupts around software-pipelined loops as when the --interrupt_threshold option is not used.

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

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

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

If you are linking C6400/C6400+/C6740 with C6200/6700 code or older C6400 code, you may need to take steps to ensure compatibility. The following lists the potential alignment conflicts and possible solutions.

Potential alignment conflicts occur when:

- Linking new C6400/C6400+/C6740 code with any C6400 code already compiled with the 4.0 tools.
- Linking new C6400/C6400+/C6740 code with code already compiled with any version of the tools for the C6200 or C6700 family.

Solutions (pick one):

- Recompile the entire application with the --silicon_version=6400 switch. This solution, if possible, is recommended because it can lead to better performance.
- Compile the new code with the --target_compatibility_6200 option. The --target_compatibility_6200 option changes the alignment of top-level arrays to 4 bytes when the --silicon_version=6400 or --silicon_version=6400+ option is used.

2.14 Using Interlist

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

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

```
cl6x --c_src_interlist function
```

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

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

Using the --c_src_interlist option can cause performance and/or code size degradation.

Example 2-4 shows a typical interlisted assembly file.

For more information about using the interlist feature with the optimizer, see Section 3.13.

Example 2-4. An Interlisted Assembly Language File

```
_main:

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

2.15 Enabling Entry Hook and Exit Hook Functions

An entry hook is a routine that is called upon entry to each function in the program. An exit hook is a routine that is called upon exit of each function. Applications for hooks include debugging, trace, profiling, and stack overflow checking.

Entry and exit hooks are enabled using the following options:

--entry_hook[=*name*] Enables entry hooks. If specified, the hook function is called *name*. Otherwise, the default entry hook function name is __entry_hook.

--entry_param{=name|address|none} Specify the parameters to the hook function. The name parameter specifies that the name of the calling function is passed to the hook function as an argument. In this case the signature for the hook function is: void hook(const char *name);
The address parameter specifies that the address of the calling function is passed to the hook function. In this case the signature for the hook function is: void hook(void (*addr)());
The none parameter specifies that the hook is called with no parameters. This is the default. In this case the signature for the hook function is: void hook(void);

--exit_hook[=*name*] Enables exit hooks. If specified, the hook function is called *name*. Otherwise, the default exit hook function name is __exit_hook.

--exit_param{=name|address|none}	Specify the parameters to the hook function. The name parameter specifies that the name of the calling function is passed to the hook function as an argument. In this case the signature for the hook function is: void hook(const char *name); The address parameter specifies that the address of the calling function is passed to the hook function. In this case the signature for the hook function is: void hook(void (*addr)()); The none parameter specifies that the hook is called with no parameters. This is the default. In this case the signature for the hook function is: void hook(void);

The presence of the hook options creates an implicit declaration of the hook function with the given signature. If a declaration or definition of the hook function appears in the compilation unit compiled with the options, it must agree with the signatures listed above.

In C++, the hooks are declared extern "C". Thus you can define them in C (or assembly) without being concerned with name mangling.

Hooks can be declared inline, in which case the compiler tries to inline them using the same criteria as other inline functions.

Entry hooks and exit hooks are independent. You can enable one but not the other, or both. The same function can be used as both the entry and exit hook.

You must take care to avoid recursive calls to hook functions. The hook function should not call any function which itself has hook calls inserted. To help prevent this, hooks are not generated for inline functions, or for the hook functions themselves.

See Section 6.8.17 for information about the NO_HOOKS pragma.

The --remove_hooks_when_inlining option removes entry/exit hooks for functions that are auto-inlined by the optimizer.

Chapter 3
SPRU187O – May 2008

Optimizing Your Code

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

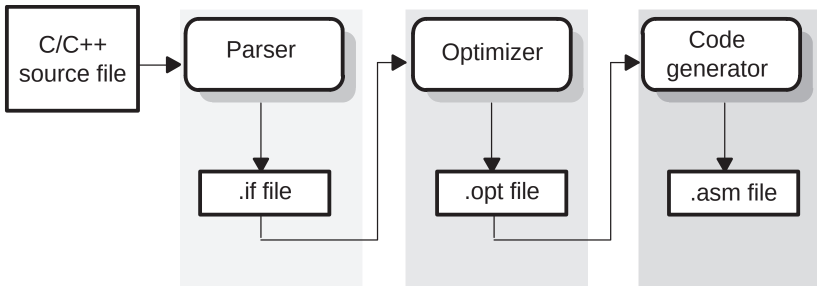
This chapter describes how to invoke different levels of optimization and describes which optimizations are performed at each level. This chapter also describes how you can use the Interlist feature when performing optimization and how you can profile or debug optimized code.

3.1 Invoking Optimization

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

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

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



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

- **--opt_level=0** or **-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
- **--opt_level=1** or **-O1**

  Performs all --opt_level=0 (-O0) optimizations, plus:
  - Performs local copy/constant propagation
  - Removes unused assignments
  - Eliminates local common expressions
- **--opt_level=2** or **-O2**

  Performs all --opt_level=1 (-O1) optimizations, plus:
  - Performs software pipelining (see Section 3.2)
  - 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 --opt_level=2 (or -O2) as the default if you use --opt_level (-O) without an optimization level.

- **--opt_level=3** or **-O3**

  Performs all --opt_level=2 (or -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; the called functions attributes 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 --opt_level=3 (or -O3), see Section 3.6 and Section 3.7 for more information.

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

Do Not Lower the Optimization Level to Control Code Size

Note: To reduce code size, do not lower the level of optimization. Instead, use the --opt_for_space option to control the code size/performance tradeoff. Higher optimization levels (--opt_level or -O) combined with high --opt_for_space levels result in the smallest code size. For more information, see Section 3.5.

The --opt_level=*n* (-O*n*) Option Applies to the Assembly Optimizer

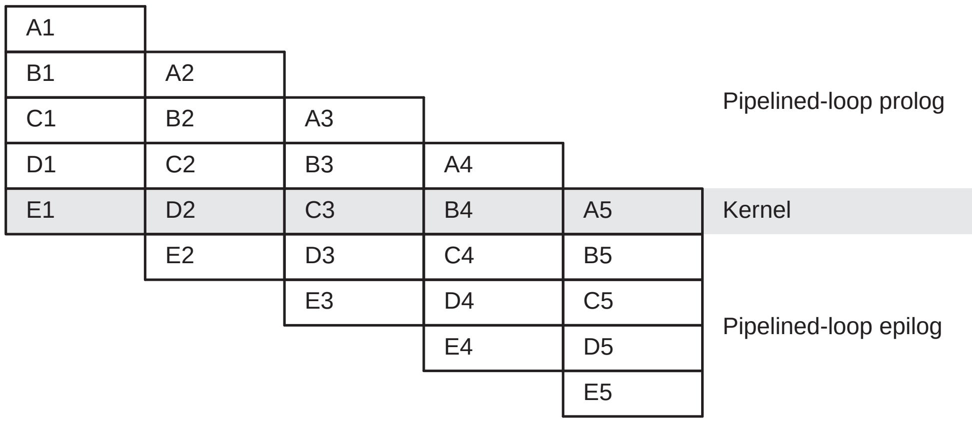
Note: The --opt_level=*n* (-O*n*) option should also be used with the assembly optimizer. Although the assembly optimizer does not perform all the optimizations described here, key optimizations such as software pipelining and loop unrolling require the --opt_level (-O) option.

3.2 Optimizing Software Pipelining

Software pipelining schedules instructions from a loop so that multiple iterations of the loop execute in parallel. At optimization levels --opt_level=2 (or -O2) and --opt_level=3 (or -O3), the compiler usually attempts to software pipeline your loops. The --opt_for_space option also affects the compiler's decision to attempt to software pipeline loops. In general, code size and performance are better when you use the --opt_level=2 or --opt_level=3 options. (See Section 3.1.)

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



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

For more information about software pipelining, see the *TMS320C6000 Programmer's Guide*.

3.2.1 Turn Off Software Pipelining (--disable_software_pipelining Option)

At optimization levels --opt_level=2 (or -O2) and -O3, the compiler attempts to software pipeline your loops. You might not want your loops to be software pipelined for debugging reasons. Software-pipelined loops are sometimes difficult to debug because the code is not presented serially. The --disable_software_pipelining option affects both compiled C/C++ code and assembly optimized code.

Software Pipelining May Increase Code Size

Note: To reduce code size, use the --opt_for_space option, rather than the --disable_software_pipelining option. These code size options may disable software pipelining, and enable other code size reduction optimizations.

3.2.2 Software Pipelining Information

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

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

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

More Details on Software Pipelining Information

Note: Refer to the *TMS320C6000 Programmer’s Guide* for details on the information and messages that can appear in the Software Pipelining Information comment block before each loop.

Example 3-1. Software Pipelining Information

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

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

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

- **Loop unroll factor.** The number of times the loop was unrolled specifically to increase performance based on the resource bound constraint in a software pipelined loop.
- **Known minimum trip count.** The minimum number of times the loop will be executed.
- **Known maximum trip count.** The maximum number of times the loop will be executed.
- **Known max trip count factor.** Factor that would always evenly divide the loops trip count. This information can be used to possibly unroll the loop.
- **Loop label.** The label you specified for the loop in the linear assembly input file. This field is not present for C/C++ code.
- **Loop carried dependency bound.** The distance of the largest loop carry path. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration. Instructions that are part of the loop carry bound are marked with the ^ symbol.
- **Initiation interval** (ii). The number of cycles between the initiation of successive iterations of the loop. The smaller the initiation interval, the fewer cycles it takes to execute a loop.
- **Resource bound.** The most used resource constrains the minimum initiation interval. 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).** 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).** The resource bound value as determined by the number of instructions that use the .D, .L, and .S units. It is calculated with the following formula:

  Bound(.L .S .D .LS .SLED) = ceil((.L + .S + .D + .LS + .LSD) / 3)
- **Minimum required memory pad.** The number of bytes that are read if speculative execution is enabled. See Section 3.2.3 for more information.

3.2.2.1 Loop Disqualified for Software Pipelining Messages

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

- **Bad loop structure**. This error is very rare and can stem from the following:
  - An asm statement inserted in the C code inner loop
  - Parallel instructions being used as input to the Linear Assembly Optimizer
  - Complex control flow such as GOTO statements and breaks
- **Loop contains a call**. Sometimes the compiler may not be able to inline a function call that is in a loop. Because the compiler could not inline the function call, the loop could not be software pipelined.
- **Too many instructions**. There are too many instructions in the loop to software pipeline.
- **Software pipelining disabled**. Software pipelining has been disabled by a command-line option, such as when using the --disable_software_pipelining option, not using the --opt_level=2 (or -O2) or --opt_level=3 (or -O3) option, or using the --opt_for_space=2 or --opt_for_space=3 option.
- **Uninitialized trip counter**. The trip counter may not have been set to an initial value.
- **Suppressed to prevent code expansion**. Software pipelining may be suppressed because of the --opt_for_space=1 option. When the --opt_for_space=1 option is used, software pipelining is disabled in less promising cases to reduce code size. To enable pipelining, use --opt_for_space=0 or omit the --opt_for_space option altogether.
- **Loop carried dependency bound too large**. If the loop has complex loop control, try --speculate_loads according to the recommendations in Section 3.2.3.2.
- **Cannot identify trip counter**. The loop trip counter could not be identified or was used incorrectly in the loop body.

3.2.2.2 Pipeline Failure Messages

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

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

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

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

  – **Regs Live Always.** The number of values that must be assigned a register for the duration of the whole loop body. This means that these values must always be allocated registers for any given schedule found for the loop.
  – **Max Regs Live.** Maximum number of values live at any given cycle in the loop that must be allocated to a register. This indicates the maximum number of registers required by the schedule found.
  – **Max Cond Regs Live.** Maximum number of registers live at any given cycle in the loop kernel that must be allocated to a condition register.
- **Cycle count too high. Never profitable**. With the schedule that the compiler found for the loop, it is more efficient to use a non-software-pipelined version.
- **Did not find schedule**. The compiler was unable to find a schedule for the software pipeline at the given ii (iteration interval). You should simplify the loop and/or eliminate loop carried dependencies.
- **Iterations in parallel > minimum or maximum trip count**. A software pipeline schedule was found, but the schedule has more iterations in parallel than the minimum or maximum loop trip count. You must enable redundant loops or communicate the trip information.
- **Speculative threshold exceeded**. It would be necessary to speculatively load beyond the threshold currently specified by the --speculate_loads option. You must increase the --speculate_loads threshold as recommended in the software-pipeline feedback located in the assembly file.
- **Register is live too long**. A register must have a value that exists (is live) for more than ii cycles. You may insert MV instructions to split register lifetimes that are too long.

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

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

  This means that the instruction that defines the register value is on line 10 and the instruction that uses the register value is on line 17 in the .sa file.
- **Too many predicates live on one side**. The C6000 has predicate, or conditional, registers available for use with conditional instructions. There are five predicate registers on the C6200 and C6700, and six predicate registers on the C6400, C6400+, and C6700+. There are two or three on the A side and three on the B side. Sometimes the particular partition and schedule combination requires more than these available registers.
- **Schedule found with N iterations in parallel**. (This is not a failure message.) A software pipeline schedule was found with N iterations executing in parallel.
- **Too many reads of one register**. The same register can be read a maximum of four times per cycle with the C6200 or C6700 core. The C6400 core can read the same register any number of times per cycle.
- **Trip variable used in loop - Cannot adjust trip count**. The loop trip counter has a use in the loop other than as a loop trip counter.

3.2.2.3 Register Usage Table Generated by the --debug_software_pipeline Option

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

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

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

Each column represents one register on the TMS320C6000. The registers are labeled in the first three rows of the register usage table and should be read columnwise.

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

An example of the register usage table follows:

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

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

3.2.3 Collapsing Prologs and Epilogs for Improved Performance and Code Size

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

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

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

The prolog and epilog of a software-pipelined loop consists of up to *p*-1 stages of length *ii*, where *p* is the number of iterations that are executed in parallel during the steady state and *ii* is the cycle time for the pipelined loop body. During prolog and epilog collapsing the compiler tries to collapse as many stages as possible. However, over-collapsing can have a negative performance impact. Thus, by default, the compiler attempts to collapse as many stages as possible without sacrificing performance. When the --opt_for_space=0 or --opt_for_space=1 options are invoked, the compiler increasingly favors code size over performance.

3.2.3.1 Speculative Execution

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

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

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

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

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

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

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

3.2.3.2 Selecting the Best Threshold Value

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

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

This information shows up in the comment block as follows:

```
;*    Minimum required memory pad : 5 bytes
;*    Minimum threshold value     : --speculate_loads=7
;*
;*    For further improvement on this loop, try option --speculate_loads=14
```

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

However, you need to consider whether a larger threshold value would facilitate additional collapsing. This information is also provided, if applicable. For example, in the above comment block, a threshold value of 14 might facilitate further collapsing.

3.3 Redundant Loops

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

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

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

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

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

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

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

You may be able to help the compiler avoid producing redundant loops with the use of --program_level_compile --opt_level=3 (see Section 3.7) or the use of the MUST_ITERATE pragma (see Section 6.8.15).

Note: Turning Off Redundant Loops

Specifying any --opt_for_space option turns off redundant loops.

3.4 Utilizing the Loop Buffer Using SPLOOP on C6400+ and C6740

The C6400+ and C6740 ISA has a loop buffer which improves performance and reduces code size for software pipelined loops. The loop buffer provides the following benefits:

- Code size. A single iteration of the loop is stored in program memory.
- Interrupt latency. Loops executing out of the loop buffer are interruptible.
- Improves performance for loops with unknown trip counts and eliminates redundant loops.
- Reduces or eliminates the need for speculated loads.
- Reduces power usage.

You can tell that the compiler is using the loop buffer when you find SPLOOP(D/W) at the beginning of a software pipelined loop followed by an SPKERNEL at the end. Refer to the *TMS320C6400/C6400+ CPU and Instruction Set Reference Guide* for information on SPLOOP.

When the --opt_for_space option is not used, the compiler will not use the loop buffer if it can find a faster software pipelined loop without it. When using the --opt_for_space option, the compiler will use the loop buffer when it can.

The compiler does not generate code for the loop buffer (SPLOOP/D/W) when any of the following occur:

- ii (initiation interval) > 14 cycles
- Dynamic length (of a single iteration) > 48 cycles
- The optimizer completely unrolls the loop
- Code contains elements that disqualify normal software pipelining (call in loop, complex control code in loop, etc.). See the *TMS320C6000 Programmer's Guide* for more information.

3.5 Reducing Code Size (--opt_for_space (or -ms) Option)

When using the --opt_level=*n* option (or -O*n*), you are telling the compiler to optimize your code. The higher the value of *n*, the more effort the compiler invests in optimizing your code. However, you might still need to tell the compiler what your optimization priorities are. By default, when --opt_level=2 or -opt_level=3 is specified, the compiler optimizes primarily for performance. (Under lower optimization levels, the priorities are compilation time and debugging ease.) You can adjust the priorities between performance and code size by using the code size flag --opt_for_space=*n*. The --opt_for_space=0, --opt_for_space=1, --opt_for_space=2 and --opt_for_space=3 options increasingly favor code size over performance.

When you specify --silicon_version=6400+ in conjunction with the --opt_for_space option, the code will be tailored for compression. That is, more instructions are tailored so they will more likely be converted from 32-bit to 16-bit instructions when assembled.

It is recommended that a code size flag not be used with the most performance-critical code. Using --opt_for_space=0 or --opt_for_space=1 is recommended for all but the most performance-critical code. Using --opt_for_space=2 or --opt_for_space=3 is recommended for seldom-executed code. Either --opt_for_space=2 or --opt_for_space=3 should be used if you need minimum code size. It is generally recommended that the code size flags be combined with --opt_level=2 or --opt_level=3.

Disabling Code-Size Optimizations or Reducing the Optimization Level

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

The --opt_for_space Option is Equivalent to --opt_for_space=0

Note: If you use --opt_for_space with no code size level number specified, the option level defaults to --opt_for_space=0.

3.6 Performing File-Level Optimization (--opt_level=3 option)

The --opt_level=3 option (aliased as the -O3 option) instructs the compiler to perform file-level optimization. You can use the --opt_level=3 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 --opt_level=3 to perform the indicated optimization:

Table 3-1. Options That You Can Use With --opt_level=3

If You ...	Use this Option	See
Have files that redeclare standard library functions	--std_lib_func_defined --std_lib_func_redefined	Section 3.6.1
Want to create an optimization information file	--gen_opt_level=*n*	Section 3.6.2
Want to compile multiple source files	--program_level_compile	Section 3.7

Do Not Lower the Optimization Level to Control Code Size

Note: When trying to reduce code size, do not lower the level of optimization, as you might see an increase in code size. Instead, use the --opt_for_space option to control the code.

3.6.1 *Controlling File-Level Optimization (--std_lib_func_def Options)*

When you invoke the compiler with the --opt_level=3 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. Use Table 3-2 to select the appropriate file-level optimization option.

Table 3-2. Selecting a File-Level Optimization Option

If Your Source File...	Use this Option
Declares a function with the same name as a standard library function	--std_lib_func_redefined
Contains but does not alter functions declared in the standard library	--std_lib_func_defined
Does not alter standard library functions, but you used the --std_lib_func_redefined or --std_lib_func_defined option in a command file or an environment variable. The --std_lib_func_not_defined option restores the default behavior of the optimizer.	--std_lib_func_not_defined

3.6.2 *Creating an Optimization Information File (--gen_opt_info Option)*

When you invoke the compiler with the --opt_level=3 option, you can use the --gen_opt_info option to create an optimization information file that you can read. The number following the option 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 option.

Table 3-3. Selecting a Level for the --gen_opt_info Option

If you...	Use this option
Do not want to produce an information file, but you used the --gen_opt_level=1 or --gen_opt_level=2 option in a command file or an environment variable. The --gen_opt_level=0 option restores the default behavior of the optimizer.	--gen_opt_level=0
Want to produce an optimization information file	--gen_opt_level=1
Want to produce a verbose optimization information file	--gen_opt_level=2

3.7 Performing Program-Level Optimization (--program_level_compile and --opt_level=3 options)

You can specify program-level optimization by using the --program_level_compile option with the --opt_level=3 option (aliased as -O3). With program-level optimization, all of your source files are compiled into one intermediate file called a *module*. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

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

To see which program-level optimizations the compiler is applying, use the --gen_opt_level=2 option to generate an information file. See Section 3.6.2 for more information.

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

Note: Compiling Files With the --program_level_compile and --keep_asm Options

If you compile all files with the --program_level_compile and --keep_asm options, the compiler produces only one .asm file, not one for each corresponding source file.

3.7.1 Controlling Program-Level Optimization (--call_assumptions Option)

You can control program-level optimization, which you invoke with --program_level_compile --opt_level=3, by using the --call_assumptions option. Specifically, the --call_assumptions option indicates if functions in other modules can call a module's external functions or modify a module's external variables. The number following --call_assumptions indicates the level you set for the module that you are allowing to be called or modified. The --opt_level=3 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 --call_assumptions option.

Table 3-4. Selecting a Level for the --call_assumptions Option

If Your Module ...	Use this Option
Has functions that are called from other modules and global variables that are modified in other modules	--call_assumptions=0
Does not have functions that are called by other modules but has global variables that are modified in other modules	--call_assumptions=1
Does not have functions that are called by other modules or global variables that are modified in other modules	--call_assumptions=2
Has functions that are called from other modules but does not have global variables that are modified in other modules	--call_assumptions=3

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

Table 3-5. Special Considerations When Using the --call_assumptions Option

If Your Option is...	Under these Conditions...	Then the --call_assumptions Level...
Not specified	The --opt_level=3 optimization level was specified	Defaults to --call_assumptions=2
Not specified	The compiler sees calls to outside functions under the --opt_level=3 optimization level	Reverts to --call_assumptions=0
Not specified	Main is not defined	Reverts to --call_assumptions=0
--call_assumptions=1 or --call_assumptions=2	No function has main defined as an entry point and functions are not identified by the FUNC_EXT_CALLED pragma	Reverts to --call_assumptions=0
--call_assumptions=1 or --call_assumptions=2	No interrupt function is defined	Reverts to --call_assumptions=0
--call_assumptions=1 or --call_assumptions=2	Functions are identified by the FUNC_EXT_CALLED pragma	Remains --call_assumptions=1 or --call_assumptions=2
--call_assumptions=3	Any condition	Remains --call_assumptions=3

In some situations when you use --program_level_compile and --opt_level=3, you *must* use a --call_assumptions option or the FUNC_EXT_CALLED pragma. See Section 3.7.2 for information about these situations.

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

If you have any assembly functions in your program, you need to exercise caution when using the --program_level_compile option. The compiler recognizes only the C/C++ source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C/C++ functions, the --program_level_compile option optimizes out those C/C++ functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see Section 6.8.7) 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 --call_assumptions=*n* option with the --program_level_compile and --opt_level=3 options (see Section 3.7.1).

In general, you achieve the best results through judicious use of the FUNC_EXT_CALLED pragma in combination with --program_level_compile --opt_level=3 and --call_assumptions=1 or --call_assumptions=2.

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

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

Solution — Compile with --program_level_compile --opt_level=3 --call_assumptions=2 to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables. See Section 3.7.1 for information about the --call_assumptions=2 option.

If you compile with the --program_level_compile --opt_level=3 options only, the compiler reverts from the default optimization level (--call_assumptions=2) to --call_assumptions=0. The compiler uses --call_assumptions=0, because it presumes that the calls to the assembly language functions that have a definition in C/C++ may call other C/C++ functions or modify C/C++ variables.

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

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

- Compile with --program_level_compile --opt_level=3 --call_assumptions=1.
- Add the volatile keyword to those variables that may be modified by the assembly functions and compile with --program_level_compile --opt_level=3 --call_assumptions=2.

See Section 3.7.1 for information about the --call_assumptions=*n* option.

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

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

- You achieve the best optimization by applying the FUNC_EXT_CALLED pragma to all of the entry-point functions called from the assembly language interrupts, and then compiling with --program_level_compile --opt_level=3 --call_assumptions=2. *Be sure that you use the pragma with all of the entry-point functions.* If you do not, the compiler might remove the entry-point functions that are not preceded by the FUNC_EXT_CALLED pragma.
- Compile with --program_level_compile --opt_level=3 --call_assumptions=3. Because you do not use the FUNC_EXT_CALLED pragma, you must use the --call_assumptions=3 option, which is less aggressive than the --call_assumptions=2 option, and your optimization may not be as effective.

Keep in mind that if you use --program_level_compile --opt_level=3 without additional options, the compiler removes the C functions that the assembly functions call. Use the FUNC_EXT_CALLED pragma to keep these functions.

3.8 Using Feedback Directed Optimization

Feedback directed optimization provides a method for finding frequently executed paths in an application using compiler-based instrumentation. This information is fed back to the compiler and is used to perform optimizations. This information is also used to provide you with information about application behavior.

3.8.1 Feedback Directed Optimization

Feedback directed optimization uses run-time feedback to identify and optimize frequently executed program paths. Feedback directed optimization is a two-phase process.

3.8.1.1 Phase 1: Collect Program Profile Information

In this phase the compiler is invoked with the option --gen_profile_info, which instructs the compiler to add instrumentation code to collect profile information. The compiler inserts a minimal amount of instrumentation code to determine control flow frequencies. Memory is allocated to store counter information.

The instrumented application program is executed on the target using representative input data sets. The input data sets should correlate closely with the way the program is expected to be used in the end product environment. When the program completes, a run-time-support function writes the collected information into a profile data file called a PDAT file. Multiple executions of the program using different input data sets can be performed and in such cases, the run-time-support function appends the collected information into the PDAT file. The resulting PDAT file is post-processed using a tool called the Profile Data Decoder or pdd6x. The pdd6x tool consolidates multiple data sets and formats the data into a feedback file (PRF file, see Section 3.8.2) for consumption by phase 2 of feedback directed optimization.

3.8.1.2 Phase 2: Use Application Profile Information for Optimization

In this phase, the compiler is invoked with the --use_profile_info=*file.prf* option, which reads the specified PRF file generated in phase 1. In phase 2, optimization decisions are made using the data generated during phase 1. The profile feedback file is used to guide program optimization. The compiler optimizes frequently executed program paths more aggressively.

The compiler uses data in the profile feedback file to guide certain optimizations of frequently executed program paths.

3.8.1.3 Generating and Using Profile Information

There are two options that control feedback directed optimization:

--gen_profile_info tells the compiler to add instrumentation code to collect profile information. When the program executes the run-time-support exit() function, the profile data is written to a PDAT file. If the environment variable TI_PROFDATA on the host is set, the data is written into the specified file name. Otherwise, it uses the default filename: pprofout.pdat. The full pathname of the PDAT file (including the directory name) can be specified using the TI_PROFDATA host environment variable.

By default, the RTS profile data output routine uses the C I/O mechanism to write data to the PDAT file. You can install a device handler for the PPHNDL device that enables you to re-direct the profile data to a custom device driver routine.

Feedback directed optimization requires you to turn on at least skeletal debug information when using the --gen_profile_info option. This enables the compiler to output debug information that allows pdd6x to correlate compiled functions and their associated profile data.

--use_profile_info specifies the profile information file(s) to use for performing phase 2 of feedback directed optimization. More than one profile information file can be specified on the command line; the compiler uses all input data from multiple information files. The syntax for the option is:

--use_profile_info==*file1*[, *file2*, ..., *filen*]

If no filename is specified, the compiler looks for a file named pprofout.prf in the directory where the compiler in invoked.

3.8.1.4 Example Use of Feedback Directed Optimization

These steps illustrate the creation and use of feedback directed optimization.

1. Generate profile information. (Skeletal debug is on by default.)

```
cl6x -mv6400+ --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
     --library=lnk.cmd --library=rts64plus.lib
```

2. Execute the application.

   The execution of the application creates a PDAT file named pprofout.pdat in the current (host) directory. The application can be run on a simulator or on target hardware connected to a host machine.

3. Process the profile data.

   After running the application with multiple data-sets, run pdd6x on the PDAT files to create a profile information (PRF) file to be used with --use_profile_info.

```
ppd6x -e foo.out -o pprofout.prf pprofout.pdat
```

4. Re-compile using the profile feedback file. Skeletal debug is not required.

```
cl6x -mv6400+ --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker
     --output_file=foo.out --library=lnk.cmd --library=rts64plus.lib
```

3.8.1.5 The .ppdata Section

The profile information collected in phase 1 is stored in the *.ppdata* section, which must be allocated into target memory. The .ppdata section contains profiler counters for all functions compiled with --gen_profile_info. The default lnk.cmd file in code generation tools version 6.1 and later has directives to place the .ppdata section in data memory. If the link command file has no section directive for allocating .ppdata section, the link step places the .ppdata section in a writable memory range.

The .ppdata section must be allocated memory in multiples of 32 bytes. Please refer to the linker command file in the distribution for example usage.

3.8.1.6 Feedback Directed Optimization and Code Size Tune

Feedback directed optimization is different from the Code Size Tune feature in Code Composer Studio (CCS). The code size tune feature uses CCS profiling to select specific compilation options for each function in order to minimize code size while still maintaining a specific performance point. Code size tune is coarse-grain, since it is selecting an option set for the whole function. Feedback directed optimization selects different optimization goals along specific regions within a function.

3.8.1.7 Instrumented Program Execution Overhead

During profile collection, the execution time of the application may increase. The amount of increase depends on the size of the application and the number of files in the application compiled for profiling.

The profiling counters increase the code and data size of the application. Consider using the --opt_for_space (-ms) code size options when using profiling to mitigate the code size increase. This has no effect on the accuracy of the profile data being collected. Since profiling only counts execution frequency and not cycle counts, code size optimization flags do not affect profiler measurements.

3.8.1.8 Invalid Profile Data

When recompiling with --use_profile_info, the profile information is invalid in the following cases:

- The source file name changed between the generation of profile information (gen-profile) and the use of the profile information (use-profile).
- The source code was modified since gen-profile. In this case, profile information is invalid for the modified functions.
- Certain compiler options used with gen-profile are different from those with used with use-profile. In particular, options that affect parser behavior could invalidate profile data during use-profile. In general, using different optimization options during use-profile should not affect the validity of profile data.

3.8.2 Profile Data Decoder

The code generation tools include a new tool called the profile data decoder or pdd6x, which is used for post processing profile data (PDAT) files. The pdd6x tool generates a profile feedback (PRF) file. See Section 3.8.1 for a discussion on where pdd6x fits in the profiling flow. The pdd6x tool is invoked with this syntax:

pdd6x -e *exec.out* **-o** *application.prf filename***.pdat**

-a	Computes the average of the data values in the data sets instead of accumulating data values
-e *exec.out*	Specifies *exec.out* is the name of the application executable.
-o *application.prf*	Specifies *application.prf* is the formatted profile feedback file that is used as the argument to --use_profile_info during recompilation. If no output file is specified, the default output filename is pprofout.prf.
filename **.pdat**	Is the name of the profile data file generated by the run-time-support function. This is the default name and it can be overridden by using the host environment variable TI_PROFDATA.

The run-time-support function and pdd6x append to their respective output files and do not overwrite them. This enables collection of data sets from multiple runs of the application.

Profile Data Decoder Requirements

Note: Your application must be compiled with at least skeletal (dwarf) debug support to enable feedback directed optimization. When compiling for feedback directed optimization, the pdd6x tool relies on basic debug information about each function in generating the formatted .prf file.

The pprofout.pdat file generated by the run-time support is a raw data file of a fixed format understood only by pdd6x. You should not modify this file in any way.

3.8.3 Code Coverage

The information collected during feedback directed optimization can be used for generating code coverage reports. As with feedback directed optimization, the program must be compiled with the --gen_profile_info option.

Code coverage conveys the execution count of each line of source code in the file being compiled, using data collected during profiling.

3.8.3.1 Phase1: Collect Program Profile Information

In this phase the compiler is invoked with the option --gen_profile_info, which instructs the compiler to add instrumentation code to collect profile information. The compiler inserts a minimal amount of instrumentation code to determine control flow frequencies. Memory is allocated to store counter information.

The instrumented application program is executed on the target using representative input data sets. The input data sets should correlate closely with the way the program is expected to be used in the end product environment. When the program completes, a run-time-support function writes the collected information into a profile data file called a PDAT file. Multiple executions of the program using different input data sets can be performed and in such cases, the run-time-support function appends the collected information into the PDAT file. The resulting PDAT file is post-processed using a tool called the Profile Data Decoder or pdd6x. The pdd6x tool consolidates multiple data sets and formats the data into a feedback file (PRF file, see Section 3.8.2) for consumption by phase 2 of feedback directed optimization.

3.8.3.2 Phase 2: Generate Code Coverage Reports

In this phase, the compiler is invoked with the --use_profile_info=*file.prf* option, which indicates that the compiler should read the specified PRF file generated in phase 1. The application must also be compiled with either the --codecov or --onlycodecov option; the compiler generates a code-coverage info file. The --codecov option directs the compiler to continue compilation after generating code-coverage information, while the --onlycodecov option stops the compiler after generating code-coverage data. For example:

```
cl6x --opt_level=2 --use_profile_info=pprofout.prf --onlycodecov foo.c
```

You can specify two environment variables to control the destination of the code-coverage information file.

- The TI_COVDIR environment variable specifies the directory where the code-coverage file should be generated. The default is the directory where the compiler is invoked.
- The TI_COVDATA environment variable specifies the name of the code-coverage data file generated by the compiler. the default is *filename*.csv where filename is the base-name of the file being compiled. For example, if foo.c is being compiled, the default code-coverage data file name is foo.csv.

If the code-coverage data file already exists, the compiler appends the new dataset at the end of the file.

Code-coverage data is a comma-separated list of data items that can be conveniently handled by data-processing tools and scripting languages. The following is the format of code-coverage data:

"*filename-with-full-path*","*funcname*",*line#*,*column#*,*exec-frequency*,"*comments*"

"*filename-with-full-path*"	Full pathname of the file corresponding to the entry
"*funcname*"	Name of the function
line#	Line number of the source line corresponding to frequency data
column#	Column number of the source line
exec-frequency	Execution frequency of the line
"*comments*"	Intermediate-level representation of the source-code generated by the parser

The full filename, function name, and comments appear within quotation marks ("). For example:

```
"/some_dir/zlib/c64p/deflate.c","_deflateInit2_",216,5,1,"( strm->zalloc )"
```

Other tools, such as a spreadsheet program, can be used to format and view the code coverage data.

3.8.4 Feedback Directed Optimization API

There are two user interfaces to the profiler mechanism. You can start and stop profiling in your application by using the following run-time-support calls.

- TI_start_pprof_collection()
  This interface informs the run-time support that you wish to start profiling collection from this point on and causes the run-time support to clear all profiling counters in the application (that is, discard old counter values).
- TI_stop_pprof_collection()
  This interface directs the run-time support to stop profiling collection and output profiling data into the output file (into the default file or one specified by the TI_PROFDATA host environment variable). The run-time support also disables any further output of profile data into the output file during exit(), unless you call TI_start_pprof_collection() again.

3.8.5 Feedback Directed Optimization Summary

Options

--gen_profile_info	Adds instrumentation to the compiled code. Execution of the code results in profile data being emitted to a PDAT file.
--use_profile_info=*file.prf*	Uses profile information for optimization and/or generating code coverage information.
--codecov	Generates a code coverage information file and continues with profile-based compilation. Must be used with --use_profile_info.
--onlycodecov	Generates only a code coverage information file. Must be used with --use_profile_info.

Host Environment Variables

TI_PROFDATA	Writes profile data into the specified file
TI_COVDIR	Creates code coverage files in the specified directory
TI_COVDATA	Writes code coverage data into the specified file

API

TI_start_pprof_collection()	Clears the profile counters to file
TI_stop_pprof_collection()	Writes out all profile counters to file
PPHDNL	Device driver handle for low-level C I/O based driver for writing out profile data from a target program.

Files Created

*.pdat	Profile data file, which is created by executing an instrumented program and used as input to the profile data decoder
*.prf	Profiling feedback file, which is created by the profile data decoder and used as input to the re-compilation step

3.9 Indicating Whether Certain Aliasing Techniques Are Used

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

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

3.9.1 Use the --aliased_variables Option When Certain Aliases are Used

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

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

If you use aliases like this in your code, you must use the --aliased_variables option when you are optimizing your code. For example, if your code is similar to this, use the --aliased_variables 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;
}
```

3.9.2 Use the --no_bad_aliases Option to Indicate That These Techniques Are Not Used

The --no_bad_aliases 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 --no_bad_aliases option also specifies that loop-invariant counter increments and decrements are non-zero. Loop invariant means the value of an expression does not change within the loop.

- The --no_bad_aliases option indicates that your code does not use the aliasing technique described in Section 3.9.1. If your code uses that technique, do *not* use the --no_bad_aliases option. You must compile with the --aliased_variables option.

  Do *not* use the --aliased_variables option with the --no_bad_aliases option. If you do, the --no_bad_aliases option overrides the --aliased_variables option.

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

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

    p[2] = 5;
}
```

- The --no_bad_aliases 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 --no_bad_aliases option:

```
g(int j)
{

    int a[20];

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

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

- The --no_bad_aliases 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 --no_bad_aliases 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 --no_bad_aliases option because both of its indices are within the bounds (0..19).

- The --no_bad_aliases option indicates that loop-invariant counter increments and decrements of loop counters are non-zero. Loop invariant means a value of an expression does not change within the loop.

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

3.9.3 Using the --no_bad_aliases Option With the Assembly Optimizer

The --no_bad_aliases 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 dependencies you point out with the .mdep directive. For more information about the .mdep directive, see the .mdep topic and Section 4.6.4.

3.10 Prevent Reordering of Associative Floating-Point Operations

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

3.11 Use Caution With asm Statements in Optimized Code

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

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

3.12 Automatic Inline Expansion (--auto_inline Option)

When optimizing with the --opt_level=3 option or --opt_level=2 option (aliased as -O3 and -O2, respectively), the compiler automatically inlines small functions. A command-line option, --auto_inline=*size*, specifies the size threshold for automatic inlining. This option controls only the inlining of functions that are not explicitly declared as inline.

When the --auto_inline option is not used, the compiler sets the size limit based on the optimization level and the optimization goal (performance versus code size). If the -auto_inline size parameter is set to 0, automatic inline expansion is disabled. If the --auto_inline size parameter is set to a non-zero integer, the compiler automatically inlines any function smaller than *size*. (This is a change from previous releases, which inlined functions for which the product of the function size and the number of calls to it was less than *size*. The new scheme is simpler, but will usually lead to more inlining for a given value of *size*.)

The compiler measures the size of a function in arbitrary units; however the optimizer information file (created with the --gen_opt_info=1 or --gen_opt_info=2 option) reports the size of each function in the same units that the --auto_inline option uses. When --auto_inline is used, the compiler does not attempt to prevent inlining that causes excessive growth in compile time or size; use with care.

When --auto_inline option is not used, the decision to inline a function at a particular call-site is based on an algorithm that attempts to optimize benefit and cost. The compiler inlines eligible functions at call-sites until a limit on size or compilation time is reached.

When deciding what to inline, the compiler collects all eligible call-sites in the module being compiled and sorts them by the estimated benefit over cost. Functions declared static inline are ordered first, then leaf functions, then all others eligible. Functions that are too big are not included.

Inlining behavior varies, depending on which compile-time options are specified:

- The code size limit is smaller when compiling for code size rather than performance. The --auto_inline option overrides this size limit.
- At --opt_level=3, the compiler auto-inlines aggressively if compiling for performance.
- At --opt_level=2, the compiler only automatically inlines small functions.

Some Functions Cannot Be Inlined

Note: For a call-site to be considered for inlining, it must be legal to inline the function and inlining must not be disabled in some way. See the inlining restrictions in Section 2.11.5.

Optimization Level 3 or 2 and Inlining

Note: In order to turn on automatic inlining, you must use the --opt_level=3 option or --opt_level=2 option. At --opt_level=2, only small functions are auto-inlined. If you desire the --opt_level=3 or 2 optimizations, but not automatic inlining, use --auto_inline=0 with the --opt_level=3 or 2 option.

Inlining and Code Size

Note: Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. To prevent increases in code size because of inlining, use the --auto_inline=0 and --no_inlining options. These options, used together, cause the compiler to inline intrinsics only.

3.13 Using the Interlist Feature With Optimization

You control the output of the interlist feature when compiling with optimization (the --opt_level=*n* or -O*n* option) with the --optimizer_interlist and --c_src_interlist options.

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

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

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

Example 3-2 shows a function that has been compiled with optimization (--opt_level=2) and the --optimizer_interlist option. The assembly file contains compiler comments interlisted with assembly code.

Note: Impact on Performance and Code Size

The --c_src_interlist option can have a negative effect on performance and code size.

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

Example 3-3 shows the function from Example 3-2 compiled with the optimization (--opt_level=2) and the --c_src_interlist and --optimizer_interlist options. The assembly file contains compiler comments and C source interlisted with assembly code.

Example 3-2. The Function From Example 2-4 Compiled with the --opt_level=2 and --optimizer_interlist Options

```
_main:
;** 5    -----------------------    printf("Hello, world\n");
;** 6    -----------------------    return 0;
           STW      .D2      B3,*SP--(12)
    .line    3
           B        .S1      _printf
           NOP               2
           MVKL     .S1      SL1+0,A0
           MVKH     .S1      SL1+0,A0
||         MVKL     .S2      RL0,B3
           STW      .D2      A0,*+SP(4)
||         MVKH     .S2      RL0,B3
RL0:       ; CALL OCCURS
    .line    4
           ZERO     .L1      A4
    .line    5
           LDW      .D2      *++SP(12),B3
           NOP               4
           B        .S2      B3
           NOP               5
           ; BRANCH OCCURS
```

Example 3-3. The Function From Example 2-4 Compiled with the --opt_level=2, --optimizer_interlist, and --c_src_interlist Options

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

3.14 Debugging and Profiling Optimized Code

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

3.14.1 Debugging Optimized Code (--symdebug:dwarf, --symdebug:coff, and --opt_level Options)

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

If you are having trouble debugging loops in your code, you can use the --disable_software_pipelining option to turn off software pipelining. See Section 3.2.1 for more information.

Note: Symbolic Debugging Options Affect Performance and Code Size

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

C6400+ and C6740 Support Only DWARF Debugging

Note: Since C6400+ and C6740 produce only DWARF debug information, the --symdebug:coff option is not supported when compiling with -mv6400 or -mv6740.

3.14.2 Profiling Optimized Code

To profile optimized code, use optimization (--opt_level=0 through --opt_level=3) without any debug option. By default, the compiler generates a minimal amount of debug information without affecting optimizations, code size, or performance.

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

If you have a power profiler, use the --profile:power option with the --opt_level option. The --profile:power option produces instrument code for the power profiler.

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

Note: Profile Points

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

3.15 What Kind of Optimization Is Being Performed?

The TMS320C6000 C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/C++ programs and to reduce their size. See Section 2.11 for more information.

Following are some of the optimizations performed by the compiler:

Optimization	See
Cost-based register allocation	Section 3.15.1
Alias disambiguation	Section 3.15.1
Branch optimizations and control-flow simplification	Section 3.15.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 3.15.4
Expression simplification	Section 3.15.5
Inline expansion of functions	Section 3.15.6
Induction variable optimizations and strength reduction	Section 3.15.7
Loop-invariant code motion	Section 3.15.8
Loop rotation	Section 3.15.9
Instruction scheduling	Section 3.15.10

C6000-Specific Optimization	See
Register variables	Section 3.15.11
Register tracking/targeting	Section 3.15.12
Software pipelining	Section 3.15.13

3.15.1 Cost-Based Register Allocation

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

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

3.15.2 Alias Disambiguation

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

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

3.15.3 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 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.

3.15.4 Data Flow Optimizations

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

- **Copy propagation.** Following an assignment to a variable, the compiler replaces references to the variable with its value. The value can be another variable, a constant, or a common subexpression. This can result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable.
- **Common subexpression elimination.** When two or more expressions produce the same value, the compiler computes the value once, saves it, and reuses it.
- **Redundant assignment elimination.** Often, copy propagation and common subexpression elimination optimizations result in unnecessary assignments to variables (variables with no subsequent reference before another assignment or before the end of the function). The compiler removes these dead assignments.

3.15.5 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.

3.15.6 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.

3.15.7 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.

3.15.8 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.

3.15.9 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.

3.15.10 Instruction Scheduling

The compiler performs instruction scheduling, which is the rearranging of machine instructions in such a way that improves performance while maintaining the semantics of the original order. Instruction scheduling is used to improve instruction parallelism and hide pipeline latencies. It can also be used to reduce code size.

3.15.11 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.

3.15.12 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.

3.15.13 Software Pipelining

Software pipelining is a technique use to schedule from a loop so that multiple iterations of a loop execute in parallel. See Section 3.2 for more information.

Chapter 4

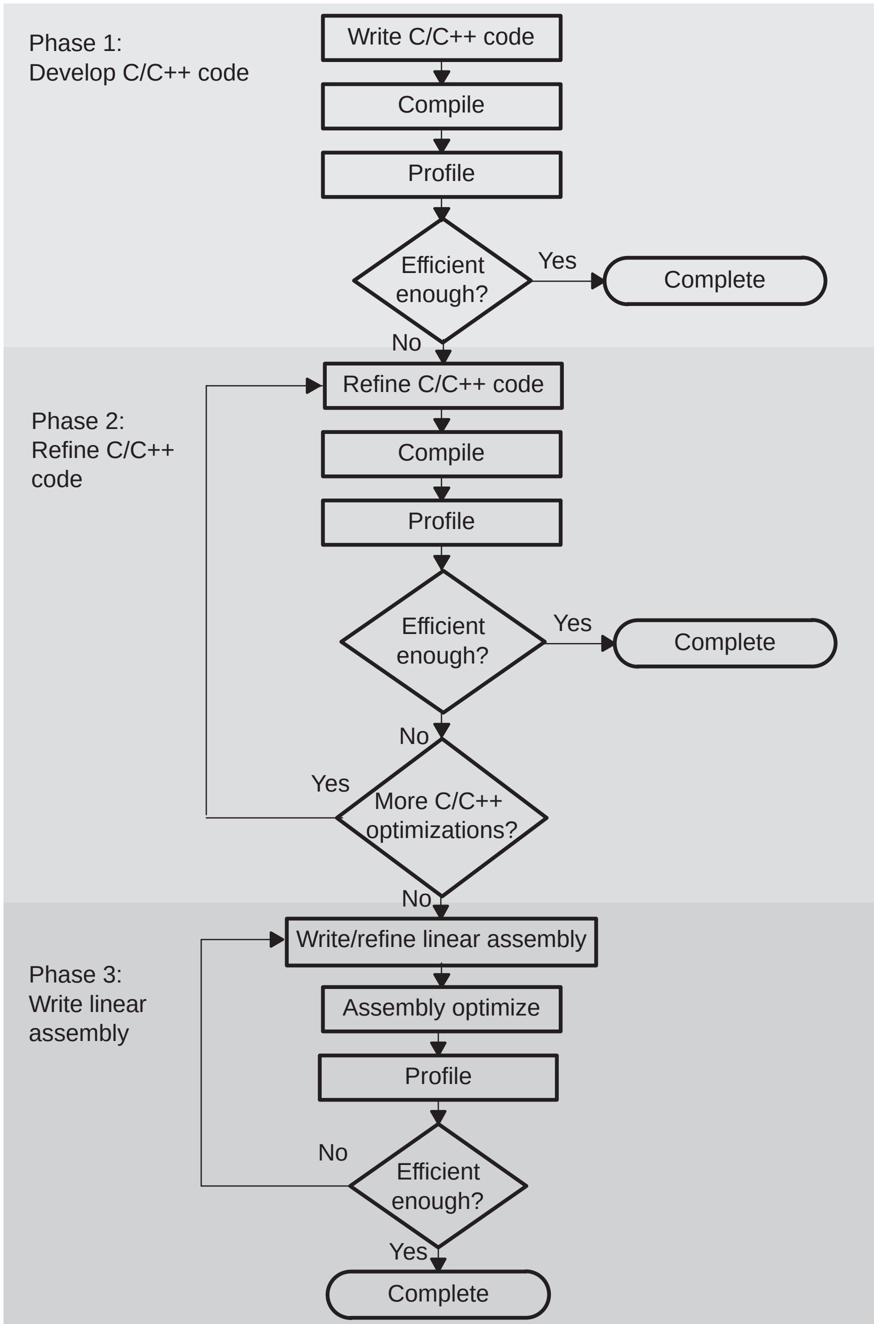
SPRU187O–May 2008

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.

4.1 Code Development Flow to Increase Performance

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



There are three phases of code development for the C6000:

- **Phase 1: write in C**

  You can develop your C/C++ code for phase 1 without any knowledge of the C6000. Use a simulator after compiling with the --opt_level=3 option without any --debug option to identify any inefficient areas in your C/C++ code. See Section 3.14 for more information about debugging and profiling optimized code. To improve the performance of your code, proceed to phase 2.
- **Phase 2: refine your C/C++ code**

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

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

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

4.2 About the Assembly Optimizer

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

The assembly optimizer performs several tasks including the following:

- Optionally, partitions instructions and/or registers
- Schedules instructions to maximize performance using the instruction-level parallelism of the C6000
- Ensures that the instructions conform to the C6000 latency requirements
- Optionally, allocates registers for your source code

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

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

4.3 What You Need to Know to Write Linear Assembly

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

- **Assembly optimizer directives**

  Your linear assembly file can be a combination of linear assembly code segments 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.

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

  The compiler options in Table 4-1 affect the behavior of the assembly optimizer.

Table 4-1. Options That Affect the Assembly Optimizer

Option	Effect	See
--ap_extension	Changes the default extension for assembly optimizer source files	Section 2.3.8
--ap_file	Changes how assembly optimizer source files are identified	Section 2.3.6
--disable_software_pipelining	Turns off software pipelining	Section 3.2.1
--debug_software_pipeline	Generates verbose software pipelining information	Section 3.2.2
--interrupt_threshold=*n*	Specifies an interrupt threshold value	Section 2.12
--keep_asm	Keeps the assembly language (.asm) file	Section 2.3.1
--no_bad_aliases	Presumes no memory aliasing	Section 3.9.3
--opt_for_space=*n*	Controls code size on four levels (*n*=0, 1, 2, or 3)	Section 3.5
--opt_level=*n*	Increases level of optimization (*n*=0, 1, 2, or 3)	Section 3.1
--quiet	Suppresses progress messages	Section 2.3.1
--silicon_version=*n*	Select target version	Section 2.3.1
--skip_assembler	Compiles or assembly optimizes only (does not assemble)	Section 2.3.1
--speculate_loads=*n*	Allows speculative execution of loads with bounded address ranges	Section 3.2.3

- **TMS320C6000 instructions**

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

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

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

Do Not Use Scheduled Assembly Code as Source

Note: 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.

If the compiler cannot make your instructions linear (non-parallel), it produces an error message. The compiler assumes instructions occur in the order the instructions appear in the file. Scheduled code is illegal (even non-parallel scheduled code). Scheduled code may not be detected by the compiler but the resulting output may not be what you intended.

- **Linear assembly source statement syntax**
  The linear assembly source programs consist of source statements that can contain assembly optimizer directives, assembly language instructions, and comments. See Section 4.3.1 for more information on the elements of a source statement.
- **Specifying registers or register sides**
  Registers can be assigned explicitly to user symbols. Alternatively, symbols can be assigned to the A-side or B-side leaving the compiler to do the actual register allocation. See Section 4.3.2 for information on specifying registers.
- **Specifying the functional unit**
  The functional unit specifier is optional in linear assembly code. Data path information is respected; unit information is ignored.
- **Source comments**
  The assembly optimizer attaches the comments on instructions from the input linear assembly to the output file. It attaches the 2-tuple <x, y> to the comments to specify which iteration and cycle of the loop an instruction is on in the software pipeline. The zero-based number x represents the iteration the instruction is on during the first execution of the kernel. The zero-based number y represents the cycle the instruction is scheduled on within a single iteration of the loop. See Section 4.3.4, 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[:]	Labels are optional for all assembly language instructions and for most (but not all) assembly optimizer directives. When used, a label must begin in column 1 of a source statement. A label can be followed by a colon.
[*register* **]**	Square brackets ([ ]) enclose conditional instructions. The machine-instruction mnemonic is executed based on the value of the register within the brackets; valid register names are A0 for C6400 and C6400+ and C6740 only, A1, A2, B0, B1, B2, or symbolic.
mnemonic	The mnemonic is a machine-instruction (such as ADDK, MVKH, B) or assembly optimizer directive (such as .proc, .trip)
unit specifier	The optional unit specifier enables you to specify the functional unit operand. Only the specified unit side is used; other specifications are ignored. The preferred method is specifying register sides.
operand list	The operand list is not required for all instructions or directives. The operands can be symbols, constants, or expressions and must be separated by commas.
comment	Comments are optional. Comments that begin in column 1 must begin with a semicolon or an asterisk; comments that begin in any other column must begin with a semicolon.

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

Follow these guidelines in writing linear assembly code:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Tab characters are interpreted as blanks. You must separate the operand list from the preceding field with a blank.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;) but comments that begin in any other column *must* begin with a semicolon.

- If you set up a conditional instruction, the register must be surrounded by square brackets.
- A mnemonic cannot begin in column 1 or it is interpreted as a label.

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

4.3.2 Register Specification for Linear Assembly

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

It is recommended that you do not initially partition the linear assembly source code by hand. This allows the compiler more freedom to partition and optimize your code. If the compiler does not find an optimal partition in a software pipelined loop, then you can partition enough instructions by hand to force optimal partitioning by partitioning 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.

Registers can be directly partitioned through two directives. The **.rega** directive is used to constrain a symbolic name to A-side registers. The **.regb** directive is used to constrain a symbolic name to B-side registers. See the .rega/.regb topic for further details on these directives. The **.reg** directive allows you to use descriptive names for values that are stored in registers. See the .reg topic for further details and examples of the .reg directive.

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

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

```
_dotp: .cproc a_0, b_0

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

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

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

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

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

       ADD    sum0, sum1, sum    ; compute final result

       .return sum
       .endproc
```

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

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

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

    int sum, I;

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

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

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

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

4.3.3 Functional Unit Specification for Linear Assembly

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

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

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

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

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

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

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

For more information on functional units refer to the *TMS320C6000 CPU and Instruction Set Reference Guide*.

4.3.4 Using Linear Assembly Source Comments

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

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

Example 4-3. Lmac Function Code Showing Comments

```
Lmac:   .cproc   A4,B4

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

        MVK     100,i
        ZERO    sh
        ZERO    sl

loop:   .trip   100

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

        .return sh:sl

        .endproc
```

4.3.5 Assembly File Retains Your Symbolic Register Names

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

A .map directive (see the .map topic) at the beginning of an assembly function associates the symbolic name with the actual register. In other words, the symbolic name becomes an alias for the actual register. The .map directive can be used in assembly and linear assembly code.

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

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

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

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

4.4 Assembly Optimizer Directives

Assembly optimizer directives supply data for and control the assembly optimization process. The assembly optimizer optimizes linear assembly code that is contained within procedures; that is, code within the .proc and .endproc directives or within the .cproc and .endproc directives. If you do not use .cproc/.proc 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 4-2 summarizes the assembly optimizer directives. It provides the syntax for each directive, a description of each directive, and any restrictions that you should keep in mind. See the specific directive topic for more detail.

In Table 4-2 and the detailed directive topics, the following terms for parameters are used:

argument— Symbolic variable name or machine register

memref— Symbol used for a memory reference (not a register)

register— Machine (hardware) register

symbol— Symbolic user name or symbolic register name

variable— Symbolic variable name or machine register

Table 4-2. Assembly Optimizer Directives Summary

Syntax	Description	Restrictions
.call [*ret_reg* =] *func_name* (*argument*$_1$, *argument*$_2$, ...)	Calls a function	Valid only within procedures
.circ *symbol*$_1$/*register*$_1$[,*symbol*$_2$/*register*$_2$]	Declares circular addressing	Must manually insert setup/teardown code for circular addressing. Valid only within procedures
label **.cproc** [*argument*$_1$ [, *argument*$_2$, …]]	Start a C/C++ callable procedure	Must use with .endproc
.endproc	End a C/C++ callable procedure	Must use with .cproc
.endproc [*variable*$_1$ [, *variable*$_2$,…]]	End a procedure	Must use with .proc
.map *symbol*$_1$/*register*$_1$[,*symbol*$_2$/*register*$_2$]	Assigns a symbol to a register	Must use an actual machine register
.mdep [*memref*$_1$[, *memref*$_2$]]	Indicates a memory dependence	Valid only within procedures
.mptr {*variable*|*memref*}, *base* [+ *offset*] [, *stride*]	Avoid memory bank conflicts	Valid only within procedures
.no_mdep	No memory aliases in the function	Valid only within procedures
.pref *symbol*/*register*$_1$[/*register*$_2$/...]	Assigns a symbol to a register in a set	Must use actual machine registers
*label***.proc** [*variable*$_1$ [, *variable*$_2$, …]]	Start a procedure	Must use with .endproc
.reg *symbol*$_1$[, *symbol*$_2$,…]	Declare variables	Valid only within procedures
.rega *symbol*$_1$[, *symbol*$_2$,…]	Partition symbol to A-side register	Valid only within procedures
.regb *symbol*$_1$[, *symbol*$_2$,…]	Partition symbol to B-side register	Valid only within procedures
.reserve [*register*$_1$ [,*register*$_2$,…]]	Prevents the compiler from allocating a register	Valid only within procedures
.return [*argument*]	Return a value to a procedure	Valid only within .cproc procedures
*label***.trip** *min*	Specify trip count value	Valid only within procedures
.volatile *memref*$_1$[,*memref*$_2$,…]	Designate memory reference volatile	Use --interrupt_threshold=1 if reference may be modified during an interrupt

.call — *Calls a Function*

Syntax

.call [*ret_reg*=] *func_name* **(**[*argument*$_1$, *argument*$_2$,...]**)**

Description

Use the **.call** directive to call a function. Optionally, you can specify a register that is assigned the result of the call. The register can be a symbolic or machine register. The .call directive adheres to the same register and function calling conventions as the C/C++ compiler. For information, see Section 7.3 and Section 7.4. There is no support for alternative register or function calling conventions.

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

Following is a description of the .call directive parameters:

ret_reg	(Optional) Symbolic/machine register that is assigned the result of the call. If not specified, the assembly optimizer presumes the call overwrites the registers A5 and A4 with a result.
func_name	The name of the function to call, or the name of the symbolic/ machine register for indirect calls. A register pair is not allowed. The label of the called function must be defined in the file. If the code for the function is not in the file, the label must be defined with the .global or .ref directive (refer to the *TMS320C6000 Assembly Language Tools User's Guide* for details). If you are calling a C/C++ function, you must use the appropriate linkname of that function. See Section 6.9 for more information.
arguments	(Optional) Symbolic/machine registers passed as an argument. The arguments are passed in this order and cannot be a constant, memory reference, or other expression.

By default, the compiler generates near calls and the linker utilizes trampolines if the near call will not reach its destination. To force a far call, you must explicitly load the address of the function into a register, and then issue an indirect call. For example:

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

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

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

For example:

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

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

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

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

.call  minimal( )             ; no arguments

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

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

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

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

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

Example Here is a complete .call example:

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

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

.circ — *Declare Circular Registers*

Syntax **.circ** *symbol*$_1$/*register*$_1$ [**,** *symbol*$_2$/*register*$_2$, ...]

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

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

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

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

The .circ directive is equivalent to using .map with a circular declaration.

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

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

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

Syntax

label **.cproc** [*argument*$_1$ [**,** *argument*$_2$**,** …]]
.endproc

Description

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

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

These operations include the following:

- When you use save-on-entry registers (A10 to A15 and B10 to B15), the assembly optimizer saves the registers on the stack and restores their original values at the end of the procedure.
- If the compiler cannot allocate machine registers to symbolic register names specified with the .reg directive (see the .reg topic) 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 7.3 and Section 7.4.

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

- **Machine-register names**. If you specify a machine-register name, its position in the argument list must correspond to the argument passing conventions for C (see Section 7.4). For example, the C/C++ compiler passes the first argument to a function in register A4. This means that the first argument in a .cproc directive must be A4 or a symbolic name. Up to ten arguments can be used with the .cproc directive.
- **Variable names**.If you specify a variable name, then the assembly optimizer ensures that either the variable name is allocated to the appropriate argument passing register or the argument passing register is copied to the register allocated for the variable name. For example, the first argument in a C/C++ call is passed in register A4, so if you specify the following .cproc directive:

  ```
  frame   .cproc arg1
  ```

  The assembly optimizer either allocates arg1 to A4, or arg1 is allocated to a different register (such as B7) and an MV A4, B7 is automatically generated.
- **Register pairs**. A register pair is specified as arghi:arglo and represents a 40-bit argument or a 64-bit type double argument for C6700. For example, the .cproc defined as follows:

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

  corresponds to a C function declared as:

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

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

If you are calling a procedure from C++ source, you must use the appropriate linkname for the procedure label. Otherwise, you can force C naming conventions by using the extern C declaration. See Section 6.9 and Section 7.5 for more information.

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

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 Section 4.4.1 for a list of instruction types that cannot appear in a .cproc region.

Example

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

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

        .reg    cond, if, ai, sum, cntr

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

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

        .return sum

        .endproc
```

.map — *Assign a Variable to a Register*

Syntax

.map *symbol*$_1$**/***register*$_1$[, *symbol*$_2$**/***register*$_2$, ...]

Description

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

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

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

Example

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

```
    .map x/A6, y/B7
    MV   x, y             ; equivalent to MV A6, B7
```

.mdep — *Indicates a Memory Dependence*

Syntax

.mdep *memref*$_1$**,** *memref*$_2$

Description

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

Following is a description of the .mdep directive parameters:

memref	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, refer to 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      ;memory reference "ld1"
;other code ...
    STW   outp2, *p2++{st1}  ;memory reference "st1"
```

.mptr — *Avoid Memory Bank Conflicts*

Syntax

.mptr {*variable* | *memref*}, *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.

Following are descriptions of the .mptr directive parameters:

variable|memref	The name of the register symbol or memory reference used to identify a load or store involved in a dependence.
base	A symbolic address 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 *symbol* or *memref* 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 *symbolic addresses* used for base symbol names are in a name space separate from all other labels. This means that a symbolic register or assembly label can have the same name as a memory bank base name. For example:

```
     .mptr   Darray,Darray
```

Example

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

```
_blkcp:    .cproc I

           .reg   ptr1, ptr2, tmp1, tmp2

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

loop:      .trip  50

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

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

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

           .endproc
```

.no_mdep — *No Memory Aliases in the Function*

Syntax

.no_mdep

Description

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

Example

Here is an example in which .no_mdep is used.

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

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

Syntax

.pref *symbol* **/** *register*$_1$[/*register*$_2$...]

Description

The **.pref** directive communicates a preference to assign a variable to one of a list of registers. The preference is used only in the .cproc or .proc region the .pref directive is declared in and is valid only until the end of the region.

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

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

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

Example

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

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

.proc/.endproc — *Define a Procedure*

Syntax

label **.proc** [*variable*$_1$ [**,** *variable*$_2$**,** …]]
.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 unscheduled assembly instructions that you want optimized by the compiler. 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 *variable* parameter in the .proc directive to indicate which registers are live in, and use the optional register parameter of the .endproc directive to indicate which registers are live out for each procedure. The *variable* can be an actual register or a symbolic name. For example:

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

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

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

See Section 4.4.1 for a list of instruction types that cannot appear in a .proc region.

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
```

.reg — *Declare Symbolic Registers*

Syntax

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

Description

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

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

Declaring register pairs explicitly is optional. Doing so is only necessary if the registers should be allocated as a pair, but they are not used that way. Here is an example of declaring a register pair:

```
        .reg    A7:A6
```

Example 1

This example uses the same code as the block move example shown for .proc/.endproc but the .reg directive is used:

```
move    .cproc dst, src, cnt

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

        .endproc
```

Notice how this example differs from the .proc example: symbolic registers declared with .reg are allocated as machine registers.

Example 2

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++, top
        MV      top, B5
        .endproc
        MV top, B6    ; WRONG: top is invalid outside of the procedure
```

.rega/.regb — *Partition Registers Directly*

Syntax

.rega *symbol*$_1$[**,** *symbol*$_2$**,** …]

.regb *symbol*$_1$[**,** *symbol*$_2$**,** …]

Description

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

```
.REGA y
.REGB u, v, w
MV    x, y
LDW  *u, v:w
```

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

When a symbol is declared with the .rega or .regb directive, it is not necessary to declare that symbol with the .reg directive.

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

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

.reserve — *Reserve a Register*

Syntax

.reserve [*register*$_1$ [**,** *register*$_2$**,** …]]

Description

The **.reserve** directive prevents the assembly optimizer from using the specified *register* in a .proc or .cproc region.

If a .reserved register is explicitly assigned in a .proc or .cproc region, then the assembly optimizer can also use that register. For example, the variable tmp1 can be allocated to register A7, even though it is in the .reserve list, since A7 was explicitly defined in the ADD instruction:

```
.cproc
.reserve  a7
.reg      tmp1
....
ADD       a6, b4, a7
....
.endproc
```

Reserving Registers A4 and A5

Note: When inside of a .cproc region that contains a .call statement, A4 and A5 cannot be specified in a .reserve statement. The calling convention mandates that A4 and A5 are used as the return registers for a .call statement.

Example 1

The .reserve in this example guarantees that the assembly optimizer does not use A10 to A13 or B10 to B13 for the variables tmp1 to tmp5:

```
test .proc    a4, b4
     .reg     tmp1, tmp2, tmp3, tmp4, tmp5
     .reserve a10, a11, a12, a13, b10, b11, b12, b13
     .....
     .endproc a4
```

Example 2

The assembly optimizer may generate less efficient code if the available register pool is overly restricted. In addition, it is possible that the available register pool is constrained such that allocation is not possible and an error message is generated. For example, the following code generates an error since all of the conditional registers have been reserved, but a conditional register is required for the variable tmp:

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

.return — *Return a Value to a C callable Procedure*

Syntax

.return [*argument*]

Description

The **.return** directive function is equivalent to the return statement in C/C++ code. It places the optional argument in the appropriate register for a return value as per the C/C++ calling conventions (see Section 7.4).

The optional *argument* can have the following meanings:

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

Arguments to the .return directive can be either symbolic register names or machine-register names.

All return statements in a .cproc region must be consistent in the type of the return value. It is not legal to mix a .return arg with a .return hi:lo in the same .cproc region.

The .return directive is unconditional. To perform a conditional .return, simply use a conditional branch around a .return. The assembly optimizer removes the branch and generates the appropriate conditional code. For example, to return if condition cc is true, code the return as:

```
[!cc]  B   around
       .return
around:
```

Example

This example uses a symbolic register, tmp, and a machine-register, A5, as .return arguments:

```
     .cproc ...
     .reg tmp
     ...
     .return tmp    = legal symbolic name
     ...
     .return a5     = legal actual name
```

.trip *Specify Trip Count Values*

Syntax *label* **.trip** *minimum value* [**,***maximum value*[**,** *factor*]]

Description The **.trip** directive specifies the value of the trip count. The *trip count* indicates how many times a loop iterates. The .trip directive is valid within procedures only. Following are descriptions of the .trip directive parameters:

label	The label represents the beginning of the loop. This is a required parameter.
minimum value	The minimum number of times that the loop can iterate. This is a required parameter. The default is 1.
maximum value	The maximum number of times that the loop can iterate. The maximum value is an optional parameter.
factor	The factor used, along with *minimum value* and *maximum value*, to determine the number of times that the loop can iterate. In the following example, the loop executes some multiple of 8, between 8 and 48, times: `loop: .trip 8, 48, 8` A *factor* of 2 states that your loop always executes an even number of times allowing the compiler to unroll once; this can result in a performance increase. The factor is optional when the maximum value is specified.

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.

You are not required to specify a .trip directive with every loop; however, you should use .trip if you know that a loop iterates some number of times. This generally means that redundant loops are not generated (unless the minimum value is really small) saving code size and execution time.

If you know that a loop always executes the same number of times whenever it is called, define maximum value (where maximum value equals minimum value) as well. The compiler may now be able to unroll your loop thereby increasing performance.

When you are compiling with the interrupt flexibility option (--interrupt_threshold=*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 --interrupt_threshold option. See Section 2.12 for more information.

Example

The .trip directive states that the loop will execute 16, 24, 32, 40 or 48 times when the w_vecsum routine is called.

```
w_vecsum:    .cproc  ptr_a, ptr_b, ptr_c, weight, cnt
             .reg    ai, bi, prod, scaled_prod, ci
             .no_mdep

loop:        .trip 16, 48, 8
             ldh     *ptr_a++, ai
             ldh     *ptr_b++, bi
             mpy     weight, ai, prod
             shr     prod, 15, scaled_prod
             add     scaled_prod, bi, ci
             sth     ci, *ptr_c++
  [cnt]      sub     cnt, 1, cnt
  [cnt]      b       loop
             .endproc
```

.volatile *Declare Memory References as Volatile*

Syntax

.volatile *memref*$_1$[, *memref*$_2$, ...]

Description

The **.volatile** directive allows you to designate memory references as volatile. Volatile loads and stores are not deleted. Volatile loads and stores are not reordered with respect to other volatile loads and stores.

If the .volatile directive references a memory location that may be modified during an interrupt, compile with the --interrupt_threshold=1 option to ensure all code referencing the volatile memory location can be interrupted.

Example

The st and ld memory references are designated as volatile.

```
.volatile st, ld

STW  W, *X{st}          ; volatile store
STW  U, *V
LDW *Y{ld}, Z           ; volatile load
```

4.4.1 Instructions That Are Not Allowed in Procedures

These types of instructions are not allowed in .cproc or .proc topic regions:

- The stack pointer (register B15) can be read, but it cannot be written to. Instructions that write to 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 changes the stack pointer register.
- Indirect branches are not allowed in a .proc or .cproc region so that the .proc or .cproc region exit protocols cannot be bypassed. Here is an example of an indirect branch:

```
B  B4    <=  illegal
```

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

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

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

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

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

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

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

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

```
.if
.cproc
.endif
.endproc

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

- The following assembly instructions cannot be used from linear assembly:
  – EFI
  – SPLOOP, SPLOOPD and SPLOOPW and all other loop-buffer related instructions
  – C6700+ instructions
  – ADDKSP and DP-relative addressing

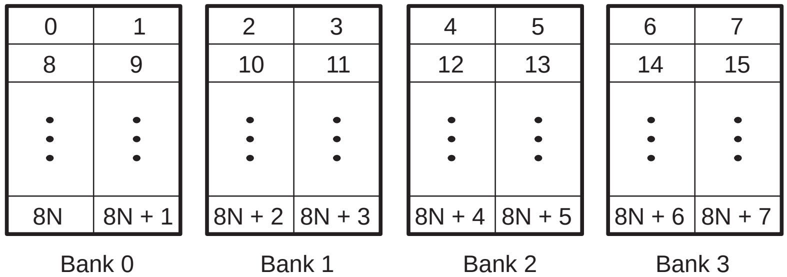
4.5 Avoiding Memory Bank Conflicts With the Assembly Optimizer

The internal memory of the C6000 family varies from device to device. See the appropriate device data sheet to determine the memory spaces in your particular device. This section discusses how to write code to avoid memory bank conflicts.

Most C6000 devices use an interleaved memory bank scheme, as shown in Figure 4-1. Each number in the diagram represents a byte address. A load byte (LDB) instruction from address 0 loads byte 0 in bank 0. A load halfword (LDH) from address 0 loads the halfword value in bytes 0 and 1, which are also in bank 0. A load word (LDW) from address 0 loads bytes 0 through 3 in banks 0 and 1.

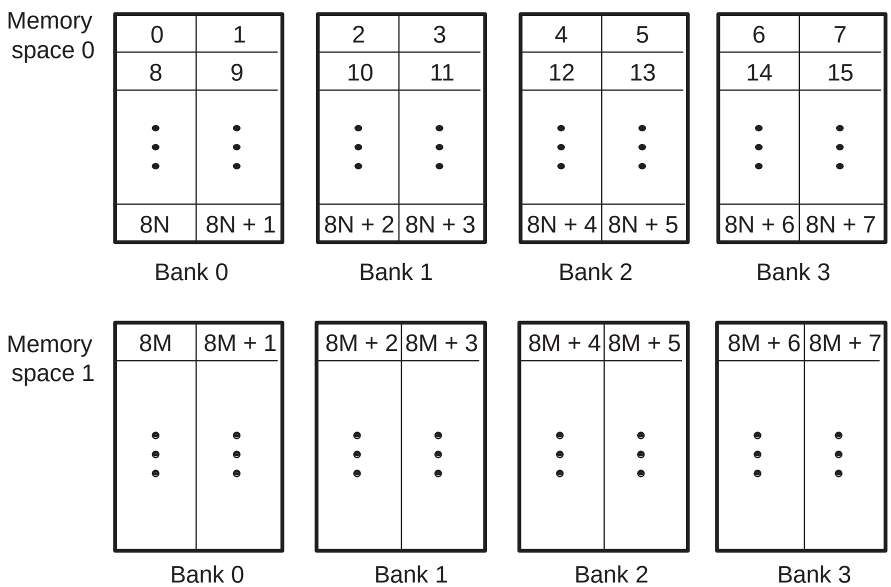
Because each bank is single-ported memory, only one access to each bank is allowed per cycle. Two accesses to a single bank in a given cycle result in a memory stall that halts all pipeline operation for one cycle while the second value is read from memory. Two memory operations per cycle are allowed without any stall, as long as they do not access the same bank.

Figure 4-1. 4-Bank Interleaved Memory



For devices that have more than one memory space (Figure 4-2), an access to bank 0 in one memory space does not interfere with an access to bank 0 in another memory space, and no pipeline stall occurs.

Figure 4-2. 4-Bank Interleaved Memory With Two Memory Spaces



4.5.1 Preventing Memory Bank Conflicts

The assembly optimizer uses the assumptions that memory operations do not have bank conflicts. If it determines that two memory operations have a bank conflict on any loop iteration it does *not* schedule the operations in parallel. The assembly optimizer checks for memory bank conflicts only for those loops that it is trying to software pipeline.

The information required for memory bank analysis indicates a base, an offset, a stride, a width, and an iteration delta. The width is implicitly determined by the type of memory access (byte, halfword, word, or double word for the C6400 and C6700). The iteration delta is determined by the assembly optimizer as it constructs the schedule for the software pipeline. The base, offset, and stride are supplied by the load and store instructions and/or by the .mptr directive.

An LD(B/BU)(H/HU)(W) or ST(B/H/W) operation in linear assembly can have memory bank information associated with it implicitly, by using the .mptr directive. The .mptr directive associates a register with the information that allows the assembly optimizer to determine automatically whether two memory operations have a bank conflict. If the assembly optimizer determines that two memory operations have a memory bank conflict, then it does not schedule them in parallel within a software pipelined loop. The syntax is:

.mptr *variable* **,** *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 the .mptr topic for more information.

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

Example 4-4. Load and Store Instructions That Specify Memory Bank Information

```
        .mptr   Ain,IN,-16
        .mptr   Bin,IN-4,-16

        .mptr   Aco,COEF,16
        .mptr   Bco,COEF+4,16

        .mptr   Aout,optr+0,4
        .mptr   Bout,optr+2,4

        LDW         *Ain--[2],Ain12          ; IN(k-I) & IN(k-I+1)
        LDW         *Bin--[2],Bin23          ; IN(k-I-2) & IN(k-I-1)
        LDW         *Ain--[2],Ain34          ; IN(k-I-4) & IN(k-I-3)
        LDW         *Bin--[2],Bin56          ; IN(k-I-6) & IN(k-I-5)

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

        STH         Assum,*Aout++[2]         ; *oPtr++ = ® >> 15)
        STH         Bssum,*Bout++[2]         ; *oPtr++ = (I >> 15)
```

4.5.2 A Dot Product Example That Avoids Memory Bank Conflicts

The C code in Example 4-5 implements a dot product function. The inner loop is unrolled once to take advantage of the C6000's ability to operate on two 16-bit data items in a single 32-bit register. LDW instructions are used to load two consecutive short values. The linear assembly instructions in Example 4-6 implement the dotp loop kernel. Example 4-7 shows the loop kernel determined by the assembly optimizer.

For this loop kernel, there are two restrictions associated with the arrays a[ ] and b[ ]:

- Because LDW is being used, the arrays must be aligned to start on word boundaries.
- To avoid a memory bank conflict, one array must start in bank 0 and the other array in bank 2. If they start in the same bank, then a memory bank conflict occurs every cycle and the loop computes a result every two cycles instead of every cycle, due to a memory bank stall. For example:

  Bank conflict:

  ```
     MVK   0, A0
  || MVK   8, B0
     LDW   *A0, A1
  || LDW   *B0, B1
  ```

  No bank conflict:

  ```
     MVK   0, A0
  || MVK   4, B0
     LDW   *A0, A1
  || LDW   *B0, B1
  ```

Example 4-5. C Code for Dot Product

```
int dot(short a[], short b[])
{
    int sum0 = 0, sum1 = 0, sum, I;

    for (I = 0; I < 100/2; I+= 2)
    {
        sum0 += a[i] * b[i];
        sum1 += a[i + 1] * b[i + 1];
    }
    return sum0 + sum1;
}
```

Example 4-6. Linear Assembly for Dot Product

```
_dot:  .cproc  a, b
        .reg   sum0, sum1, I
        .reg    val1, val2, prod1, prod2

        MVK     50,i ; I = 100/2
        ZERO    sum0 ; multiply result = 0
        ZERO    sum1 ; multiply result = 0

loop:   .trip 50
        LDW     *a++,val1        ; load a[0-1]  bank0
        LDW     *b++,val2        ; load b[0-1]  bank2
        MPY     val1,val2,prod1  ; a[0] * b[0]
        MPYH    val1,val2,prod2  ; a[1] * b[1]
        ADD     prod1,sum0,sum0  ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1  ; sum1 += a[1] * b[1]

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

        ADD     sum0,sum1,A4     ; compute final result
        .return A4
        .endproc
```

Example 4-7. Dot Product Software-Pipelined Kernel

```
L2:     ; PIPED LOOP KERNEL

          ADD    .L2     B7,B4,B4            ; |14| <0,7>  sum0 += a[0]*b[0]
||        ADD    .L1     A5,A0,A0            ; |15| <0,7>  sum1 += a[1]*b[1]
||        MPY    .M2X    B6,A4,B7            ; |12| <2,5>  a[0] * b[0]
||        MPYH   .M1X    B6,A4,A5            ; |13| <2,5>  a[1] * b[1]
|| [ B0]  B      .S1     L2                  ; |18| <5,2>  if (!I) goto loop
|| [ B0]  ADD    .S2     0xffffffff,B0,B0    ; |17| <6,1>  I--
||        LDW    .D2T2   *B5++,B6            ; |10| <7,0>  load a[0-1] bank0
||        LDW    .D1T1   *A3++,A4            ; |11| <7,0>  load b[0-1] bank2
```

It is not always possible to control fully how arrays and other memory objects are aligned. This is especially true when a pointer is passed into a function and that pointer may have different alignments each time the function is called. A solution to this problem is to write a dot product routine that cannot have memory hits. This would eliminate the need for the arrays to use different memory banks.

If the dot product loop kernel is unrolled once, then four LDW instructions execute in the loop kernel. Assuming that nothing is known about the bank alignment of arrays a and b (except that they are word aligned), the only safe assumptions that can be made about the array accesses are that a[0-1] cannot conflict with a[2-3] and that b[0-1] cannot conflict with b[2-3]. Example 4-8 shows the unrolled loop kernel.

Example 4-8. Dot Product From Example 4-6 Unrolled to Prevent Memory Bank Conflicts

```
_dotp2: .cproc   a_0, b_0
        .reg    a_4, b_4, sum0, sum1, I
        .reg    val1, val2, prod1, prod2

        ADD     4,a_0,a_4
        ADD     4,b_0,b_4
        MVK     25,i     ; I = 100/4
        ZERO    sum0     ; multiply result = 0
        ZERO    sum1     ; multiply result = 0

        .mptr   a_0,a+0,8
        .mptr   a_4,a+4,8
        .mptr   b_0,b+0,8
        .mptr   b_4,b+4,8


loop:   .trip   25

        LDW     *a_0++[2],val1  ; load a[0-1] bankx
        LDW     *b_0++[2],val2  ; load b[0-1] banky
        MPY     val1,val2,prod1 ; a[0] * b[0]
        MPYH    val1,val2,prod2 ; a[1] * b[1]
        ADD     prod1,sum0,sum0 ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1 ; sum1 += a[1] * b[1]

        LDW     *a_4++[2],val1  ; load a[2-3] bankx+2
        LDW     *b_4++[2],val2  ; load b[2-3] banky+2
        MPY     val1,val2,prod1 ; a[2] * b[2]
        MPYH    val1,val2,prod2 ; a[3] * b[3]
        ADD     prod1,sum0,sum0 ; sum0 += a[2] * b[2]
        ADD     prod2,sum1,sum1 ; sum1 += a[3] * b[3]

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

         ADD     sum0,sum1,A4    ; compute final result
        .return A4
        .endproc
```

The goal is to find a software pipeline in which the following instructions are in parallel:

```
    LDW *a0++[2],val1 ; load a[0-1]  bankx
||  LDW *a2++[2],val2 ; load a[2-3]  bankx+2
    LDW *b0++[2],val1 ; load b[0-1]  banky
||  LDW *b2++[2],val2 ; load b[2-3]  banky+2
```

Example 4-9. Unrolled Dot Product Kernel From Example 4-7

```
L2:      ; PIPED LOOP KERNEL

   [ B1]  SUB    .S2     B1,1,B1        ; <0,8>
||        ADD    .L2     B9,B5,B9       ; |21| <0,8>  ^  sum0 += a[0] * b[0]
||        ADD    .L1     A6,A0,A0       ; |22| <0,8>  ^  sum1 += a[1] * b[1]
||        MPY    .M2X    B8,A4,B9       ; |19| <1,6>  a[0] * b[0]
||        MPYH   .M1X    B8,A4,A6       ; |20| <1,6>  a[1] * b[1]
|| [ B0]  B      .S1     L2             ; |32| <2,4>  if (!I) goto loop
|| [ B1]  LDW    .D1T1   *A3++(8),A4    ; |24| <3,2>  load a[2-3] bankx+2
|| [ A1]  LDW    .D2T2   *B6++(8),B8    ; |17| <4,0>  load a[0-1] bankx

   [ A1]  SUB    .S1     A1,1,A1        ; <0,9>
||        ADD    .L2     B5,B9,B5       ; |28| <0,9>  ^  sum0 += a[2] * b[2]
||        ADD    .L1     A6,A0,A0       ; |29| <0,9>  ^  sum1 += a[3] * b[3]
||        MPY    .M2X    A4,B7,B5       ; |26| <1,7>  a[2] * b[2]
||        MPYH   .M1X    A4,B7,A6       ; |27| <1,7>  a[3] * b[3]
|| [ B0]  ADD    .S2     -1,B0,B0       ; |31| <3,3>  I--
|| [ A1]  LDW    .D2T2   *B4++(8),B7    ; |25| <4,1>  load b[2-3] banky+2
|| [ A1]  LDW    .D1T1   *A5++(8),A4    ; |18| <4,1>  load b[0-1] banky
```

Without the .mptr directives in Example 4-8, the loads of a[0-1] and b[0-1] are scheduled in parallel, and the loads of a[2-3] and b[2-3] might be scheduled in parallel. This results in a 50% chance that a memory conflict will occur on every cycle. However, the loop kernel shown in Example 4-9 can never have a memory bank conflict.

In Example 4-6, if .mptr directives had been used to specify that a and b point to different bases, then the assembly optimizer would never find a schedule for a 1-cycle loop kernel, because there would always be a memory bank conflict. However, it would find a schedule for a 2-cycle loop kernel.

4.5.3 Memory Bank Conflicts for Indexed Pointers

When determining memory bank conflicts for indexed memory accesses, it is sometimes necessary to specify that a pair of memory accesses always conflict, or that they never conflict. This can be accomplished by using the .mptr directive with a stride of 0.

A stride of 0 indicates that there is a constant relation between the memory accesses regardless of the iteration delta. Essentially, only the base, offset, and width are used by the assembly optimizer to determine a memory bank conflict. Recall that the stride is optional and defaults to 0.

In Example 4-10, the .mptr directive is used to specify which memory accesses conflict and which never conflict.

Example 4-10. Using .mptr for Indexed Pointers

```
      .mptr a,RS
      .mptr b,RS
      .mptr c,XY
      .mptr d,XY+2
      LDW     *a++[i0a],A0  ; a and b always conflict with each other
      LDW     *b++[i0b],B0  ;
      STH     A1,*c++[i1a]    ; c and d never conflict with each other
      STH     B2,*d++[i1b]    ;
```

4.5.4 Memory Bank Conflict Algorithm

The assembly optimizer uses the following process to determine if two memory access instructions might have a memory bank conflict:

1. If either access does not have memory bank information, then they do not conflict.
2. If both accesses do not have the same base, then they conflict.
3. The offset, stride, access width, and iteration delta are used to determine if a memory bank conflict will occur. The assembly optimizer uses a straightforward analysis of the access patterns and determines if they ever access the same relative bank. The stride and offset values are always expressed in bytes.

   The iteration delta is the difference in the loop iterations of the memory references being scheduled in the software pipeline. For example, given three instructions A, B, C and a software pipeline with a single-cycle kernel, then A and C have an iteration delta of 2:

   ```
   A
   B   A
   C   B   A
       C   B
           C
   ```

4.6 Memory Alias Disambiguation

Memory aliasing occurs when two instructions can access the same memory location. Such memory references are called ambiguous. Memory alias disambiguation is the process of determining when such ambiguity is not possible. When you cannot determine whether two memory references are ambiguous, you presume they are ambiguous. This is the same as saying the two instructions have a memory dependence between them.

Dependencies between instructions constrain the instruction schedule, including the software pipeline schedule. In general, the fewer the Dependencies, 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 memory references are aliased, unless it can prove otherwise.

Because alias analysis is very limited in the assembly optimizer, this presumption is often overly conservative. In such cases, the extra instruction Dependencies, 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 --no_bad_aliases Option to Handle Memory References

In the assembly optimizer, the --no_bad_aliases option means no memory references ever depend on each other. The --no_bad_aliases option does not mean the same thing to the C/C++ compiler. The C/C++ compiler interprets the --no_bad_aliases switch to indicate several specific cases of memory aliasing are guaranteed not to occur. For more information about using the --no_bad_aliases option, see Section 3.9.2.

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 Dependencies occur within that function.

Memory Dependency Exception

Note: For both of these methods, --no_bad_aliases and .no_mdep, the assembly optimizer recognizes any memory Dependencies you point out with the .mdep directive.

4.6.4 Using the .mdep Directive to Identify Specific Memory Dependencies

You can use the .mdep directive to identify specific memory Dependencies 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:

{ *memref* **}**

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

Example 4-11. Annotating a Memory Reference

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

The directive to indicate a specific memory dependence in the previous example is as follows:

```
.mdep  ld1,  st1
```

This means that whenever ld1 accesses memory at location X, some later time in code execution, st1 may also access location X. This is equivalent to adding a dependence between these two instructions. In terms of the software pipeline, these two instructions must remain in the same order. The ld1 reference must always occur before the st1 reference; the instructions cannot even be scheduled in parallel.

It is important to note the directional sense of the directive from ld1 to st1. The opposite, from st1 to ld1, is not implied. In terms of the software pipeline, while every ld1 must occur before every st1, it is still legal to schedule the ld1 from iteration n+1 before the st1 from iteration n.

Example 4-12 is a picture of the software pipeline with the instructions from two different iterations in different columns. In the actual instruction sequence, instructions on the same horizontal line are in parallel.

Example 4-12. Software Pipeline Using .mdep ld1, st1

```
iteration n               iteration n+1
-------------             -------------
LDW { ld1 }

...                       LDW { ld1 }

STW { st1 }               ...

                          STW { st1 }
```

If that schedule does not work because the iteration n st1 might write a value the iteration n+1 ld1 should read, then you must note a dependence relationship from st1 to ld1.

```
.mdep   st1, ld1
```

Both directives together force the software pipeline shown in Example 4-13.

Example 4-13. Software Pipeline Using .mdep st1, ld1 and .mdep ld1, st1

```
iteration n               iteration n+1
-------------             -------------
LDW { ld1 }

...

STW { st1 }

                          LDW { ld1 }

                          ...

                          STW { st1 }
```

Indexed addressing, *+base[index], is a good example of an addressing mode where you typically do not know anything about the relative sequence of the memory accesses, except they sometimes access the same location. To correctly model this case, you need to note the dependence relation in both directions, and you need to use both directives.

```
.mdep   ld1, st1
.mdep   st1, ld1
```

4.6.5 Memory Alias Examples

Following are memory alias examples that use the .mdep and .no_mdep directives.

- **Example 1**

  The .mdep r1, r2 directive declares that LDW must be before STW. In this case, src and dst might point to the same array.

```
fn:     .cproc      dst, src, cnt
        .reg        tmp
        .no_mdep
        .mdep       r1, r2

        LDW         *src{r1}, tmp
        STW         cnt, *dst{r2}

        .return     tmp
        .endproc
```

- **Example 2**

  Here, .mdep r2, r1 indicates that STW must occur before LDW. Since STW is after LDW in the code, the dependence relation is across loop iterations. The STW instruction writes a value that may be read by the LDW instruction on the next iteration. In this case, a 6-cycle recurrence is created.

```
fn:     .cproc          dst, src, cnt
        .reg        tmp
        .no_mdep
        .mdep       r2, r1

LOOP:   .trip       100
        LDW         *src++{r1}, tmp
        STW         tmp, *dst++{r2}
 [cnt]  SUB         cnt, 1, cnt
 [cnt]  B           LOOP

        .endproc
```

Memory Dependence/Bank Conflict

Note: Do not confuse memory alias disambiguation with the handling of memory bank conflicts. These 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.

Volatile References

Note: For volatile references, use .volatile rather than .mdep.

Chapter 5
SPRU187O–May 2008

Linking C/C++ Code

The C/C++ compiler and assembly language tools provide two methods for linking your programs:

- You can compile individual modules and link them together. This method is especially useful when you have multiple source files.
- You can compile and link in one step. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/C++ code, including the run-time-support libraries, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see the *TMS320C6000 Assembly Language Tools User's Guide*.

Topic **Page**

5.1 Invoking the Linker Through the Compiler (-z Option)

This section explains how to invoke the linker after you have compiled and assembled your programs: as a separate step or as part of the compile step.

5.1.1 Invoking the Linker Separately

This is the general syntax for linking C/C++ programs as a separate step:

cl6x --run_linker {**--rom_model | --ram_model**} *filenames* [*options*]
[**--output_file=***name.out*] **--library=***library* [*lnk.cmd*]

cl6x --run_linker	The command that invokes the linker.
--rom_model | --ram_model	Options that tell the linker to use special conventions defined by the C/C++ environment. When you use cl6x --run_linker, you must use **--rom_model** or **--ram_model**. The --rom_model option uses automatic variable initialization at run time; the --ram_model 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 --output_file option to name the output file.
options	Options affect how the linker handles your object files. Linker options can only appear after the **--run_linker** option on the command line, but otherwise may be in any order. (Options are discussed in Section 5.2.)
--output_file= *name.out*	Names the output file.
--library= *library*	Identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions, or linker command files. If you are linking C/C++ code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter. The --library option's short form is -l.
lnk.cmd	Contains options, filenames, directives, or commands for the linker.

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For information, see the *TMS320C6000 Assembly Language Tools User's Guide*.

You can link a C/C++ program consisting of modules prog1.obj, prog2.obj, and prog3.obj, with an executable filename of prog.out with the command:

```
cl6x --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts6200.lib
```

5.1.2 Invoking the Linker as Part of the Compile Step

This is the general syntax for linking C/C++ programs as part of the compile step:

cl6x *filenames* [*options*] **--run_linker** {**--rom_model | --ram_model**} *filenames* [*options*] [**--output_file=***name.out*] **--library=***library* [*lnk.cmd*]

The **--run_linker** option divides the command line into the compiler options (the options before --run_linker) and the linker options (the options following --run_linker). The --run_linker option must follow all source files and compiler options on the command line.

All arguments that follow --run_linker on the command line are passed to the linker. These arguments can be linker command files, additional object files, linker options, or libraries. These arguments are the same as described in Section 5.1.1.

All arguments that precede --run_linker on the command line are compiler arguments. These arguments can be C/C++ source files, assembly files, linear assembly files, or compiler options. These arguments are described in Section 2.2.

You can compile and link a C/C++ program consisting of modules prog1.c, prog2.c, and prog3.c, with an executable filename of prog.out with the command:

```
cl6x prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --library=rts6200.lib
```

Note: Order of Processing Arguments in the Linker

The order in which the linker processes arguments is important. The compiler passes arguments to the linker in the following order:

1. Object filenames from the command line
2. Arguments following the --run_linker option on the command line
3. Arguments following the --run_linker option from the C6X_C_OPTION environment variable

5.1.3 Disabling the Linker (--compile_only Compiler Option)

You can override the --run_linker option by using the --compile_only compiler option. The -run_linker option's short form is -z and the --compile_only option's short form is -c.

The --compile_only option is especially helpful if you specify the --run_linker option in the C6X_C_OPTION environment variable and want to selectively disable linking with the --compile_only option on the command line.

5.2 Linker Options

All command-line input following the --run_linker option (aliased as -z) is passed to the linker as parameters and options. Following are the options that control the linker, along with detailed descriptions of their effects.

--absolute_exe	Produces an absolute, executable module. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe is specified.
-ar	Produces a relocatable, executable object module. The output module contains the special linker symbols, an optional header, and all symbol references. The relocation information is retained.
--arg_size=*size*	Allocates memory to be used by the loader to pass arguments from the command line of the loader to the program. The linker allocates *size* bytes in an uninitialized .args section. The __c_args__ symbol contains the address of the .args section.
--compress_dwarf	Aggressively reduces the size of DWARF information from input object files
--define=*name*[=*val*]	Predefines *name* as a preprocessor macro. This option is distinct from the compiler --define option.
--diag_error=*num*	Categorizes the diagnostic identified by *num* as an error. See Section 2.7.1 for details.
--diag_remark=*num*	Categorizes the diagnostic identified by *num* as a remark. See Section 2.7.1 for details.
--diag_suppress=*num*	Suppresses the diagnostic identified by *num*. See Section 2.7.1 for details.
--diag_warning=*num*	Categorizes the diagnostic identified by *num* as a warning. See Section 2.7.1 for details.
--disable_auto_rts	Disables the automatic selection of a run-time-support library. See Section 5.4.1.2 for more information.
--disable_clink	Disables conditional linking that has been set up with the assembler .clink directive for COFF object files. By default, all sections are unconditionally linked.
--disable_pp	Disables preprocessing for command files. By default, the linker now preprocesses link command files using a standard C preprocessor.
--display_error_number=*num*	Displays a diagnostic's identifiers along with its text. See Section 5.4.1.2 for more information.
--entry_point=*global_symbol*	Defines a *global_symbol* that specifies the primary entry point for the output module
--fill_value=*value*	Sets the default fill value for null areas within output sections; *value* is a 32-bit constant
--generate_dead_funcs_list	Writes a list of the dead functions that were removed by the linker to file fname for object files
--heap_size=*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.
--issue_remarks	Issues remarks (nonserious warnings). See Section 5.4.1.2 for more information.

--library= *libraryname*	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. The --library option's short form is -l.
--linker_help	Produces a help listing displaying syntax and available options
--make_global=*global_symbol*	Defines *global_symbol* as global even if the global symbol has been made static with the --make_static option
--make_static	Makes all global symbols static; global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.
--map_file=*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.
--mapfile_contents=*filter*[,*filter*]	Controls the information that appears in the map file. Enter --mapfile_contents=help on the command line to produce a listing of available options.
--no_demangle	Disables demangling of symbol names in diagnostics
--no_sym_merge	Disables merge of symbolic debugging information in COFF object files. The linker keeps the duplicate entries of symbolic debugging information commonly generated when a C program is compiled for debugging. (Deprecated option; use the strip utility described in the *TMS320C6000 Assembly Language Tools User's Guide*.
--no_sym_table	Creates a smaller output section by stripping symbol table information and line number entries from the output module.
--no_warnings	Suppresses warning diagnostics (errors are still issued). See Section 5.4.1.2 for more information.
--output_file=*filename*	Names the executable output module. The *filename* must follow operating system conventions. If the --output_file option is not used, the default filename is a.out.
--priority	Satisfies each unresolved reference by the first library that contains a definition for that symbol
--ram_model	Initializes variables at load time. See Section 7.8.5 for more information.
--relocatable	Retains relocation entries in the output module.
--reread_libs	Forces rereading of libraries. The linker continues to reread libraries until no more references can be resolved.
--rom_model	Autoinitializes variables at run time. See Section 7.8.4 for more information.
--run_abs	Produces an absolute listing file.
--scan_libraries	Scans all libraries during a link to look for duplicate symbol definitions to those symbols that are actually included in the link.
--set_error_limit=*num*	Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.) See Section 5.4.1.2 for more information.
--stack_size=*size*	Sets the C/C++ system stack size to *size* bytes and defines a global symbol that specifies the stack size. The default is 1K bytes.

--strict_compatibility	Performs more conservative and rigorous compatibility checking of input object files.
--symbol_map=*refname*=*defname*	Enables symbol mapping, which allows a symbol reference to be resolved by a symbol with a different name.
--trampolines	Generates a trampoline code section for each call that is linked out-of-range of its called destination. The trampoline code section contains a sequence of instructions that performs a transparent long branch to the original called address. Each calling instruction that is out-of-range from the called function is redirected to the trampoline.
--undef_sym=*symbol*	Places the unresolved external symbol *symbol* into the output module's symbol table. This forces the linker to search a library and include the member that defines the symbol.
--undefine=*name*	Removes the preprocessor macro *name*. This option is distinct from the compiler --undefine option.
--verbose_diagnostics	Provides verbose diagnostics that display the original source with line-wrap. See Section 5.4.1.2 for more information.
--xml_link_info=*file*	Generates an XML link information file. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file.

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

5.3 Linker Code Optimizations

These options are used to further optimize your code.

5.3.1 Generate List of Dead Functions (--generate_dead_funcs_list Option)

In order to facilitate the removal of unused code, the linker generates a feedback file containing a list of functions that are never referenced. The feedback file must be used the next time you compile the source files. The syntax for the --generate_dead_funcs_list option is:

--generate_dead_funcs_list= *filename*

If *filename* is not specified, a default filename of dead_funcs.txt is used.

Proper creation and use of the feedback file entails the following steps:

1. Compile all source files using the --gen_func_subsections compiler option. For example:
   ```
   cl6x file1.c file2.c --gen_func_subsections
   ```
2. During the linker, use the --generate_dead_funcs_list option to generate the feedback file based on the generated object files. For example:
   ```
   cl6x --run_linker file1.obj file2.obj
    --generate_dead_funcs_list=feedback.txt
   ```
   Alternatively, you can combine steps 1 and 2 into one step. When you do this, you are not required to specify --gen_func_subsections when compiling the source files as this is done for you automatically. For example:
   ```
   cl6x file1.c file2.c --run_linker --generate_dead_funcs_list=feedback.txt
   ```
3. Once you have the feedback file, rebuild the source. Give the feedback file to the compiler using the --use_dead_funcs_list option. This option forces each dead function listed in the file into its own subsection. For example:
   ```
   cl6x file1.c file2.c --use_dead_funcs_list=feedback.txt
   ```
4. Invoke the linker with the newly built object files. The linker removes the subsections. For example:
   ```
   cl6x --run_linker file1.obj file2.obj
   ```
   Alternatively, you can combine steps 3 and 4 into one step. For example:
   ```
   cl6x file1.c file2.c --use_dead_funcs_list=feedback.txt --run_linker
   ```

Note: Dead Functions Feedback

The feedback file generated with the -gen_dead_funcs_list option is version controlled. It must be generated by the linker in order to be processed correctly by the compiler.

5.3.2 Generating Function Subsections (--gen_func_subsections Compiler Option)

When the linker places code into an executable file, it allocates all the functions in a single source file as a group. This means that if any function in a file needs to be linked into an executable, then all the functions in the file are linked in. This can be undesirable if a file contains many functions and only a few are required for an executable.

This situation may exist in libraries where a single file contains multiple functions, but the application only needs a subset of those functions. An example is a library .obj file that contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. By default, both the signed and unsigned routines are linked in since they exist in the same .obj file.

The --gen_func_subsections compiler option remedies this problem by placing each function in a file in its own subsection. Thus, only the functions that are referenced in the application are linked into the final executable. This can result in an overall code size reduction.

However, be aware that using the --gen_func_subsections compiler option can result in overall code size growth if all or nearly all functions are being referenced. This is because any section containing code must be aligned to a 32-byte boundary to support the C6000 branching mechanism. When the --gen_func_subsections option is not used, all functions in a source file are usually placed in a common section which is aligned. When --gen_func_subsections is used, each function defined in a source file is placed in a unique section. Each of the unique sections requires alignment. If all the functions in the file are required for linking, code size may increase due to the additional alignment padding for the individual subsections.

Thus, the --gen_func_subsections compiler option is advantageous for use with libraries where normally only a limited number of the functions in a file are used in any one executable.

The alternative to the --gen_func_subsections option is to place each function in its own source file.

In addition to placing each function in a separate subsection, the compiler also annotates that subsection with a conditional linking directive, .clink. This directive marks the section as a candidate to be removed if it is not referenced by any other section in the program. The compiler does not place a .clink directive in a subsection for a trap or interrupt function, as these may be needed by a program even though there is no symbolic reference to them anywhere in the program.

If a section that has been marked for conditional linking is never referenced by any other section in the program, that section is removed from the program. Conditional linking is disabled when performing a partial link or when relocation information is kept with the output of the link. Conditional linking can also be disabled with the --disable_clink link option.

5.4 Controlling the Linking Process

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

- Include the compiler's run-time-support library
- Specify the type of initialization
- Determine how you want to allocate your program into memory

This section discusses how these factors are controlled and provides an example of the standard default linker command file.

For more information about how to operate the linker, see the linker description in the *TMS320C6000 Assembly Language Tools User's Guide.*

5.4.1 Including the Run-Time-Support Library

You must include a run-time-support library in the linker process. The following sections describe two methods for including the run-time-support library.

5.4.1.1 Manual Run-Time-Support Library Selection

You must link all C/C++ programs with a run-time-support library. The library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. You must use the --library linker option to specify which C6000 run-time-support library to use. The --library option also tells the linker to look at the --search_path options and then the C6X_C_DIR environment variable to find an archive path or object file. To use the --library linker option, type on the command line:

cl6x --run_linker {**--rom_model | --ram_model**} *filenames* **--library=***libraryname*

Generally, you should specify the run-time-support 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 --reread_libs option to force the linker to reread all libraries until references are resolved. Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

By default, if a library introduces an unresolved reference and multiple libraries have a definition for it, then the definition from the same library that introduced the unresolved reference is used. Use the --priority option if you want the linker to use the definition from the first library on the command line that contains the definition.

5.4.1.2 Automatic Run-Time-Support Library Selection

If the --rom_model or --ram_model option is specified during the linker and the entry point for the program (normally c_int00) is not resolved by any specified object file or library, the linker attempts to automatically include the best compatible run-time-support library for your program. The chosen run-time-support library is linked in as if it was specified with the --library option last on the command line. Alternatively, you can always force the linker to choose an appropriate run-time-support library by specifying “libc.a” as an argument to the --library option, or when specifying the run-time-support library name explicitly in a linker command file.

The automatic selection of a run-time-support library can be disabled with the --disable_auto_rts option.

If the --issue_remarks option is specified before the --run_linker option during the linker, a remark is generated indicating which run-time support library was linked in. If a different run-time-support library is desired, you must specify the name of the desired run-time-support library using the --library option and in your linker command files when necessary.

For example:

```
cl6x --silicon_version=6400+ --issue_remarks main.c --run_linker --rom_model

<Linking>

remark: linking in "libc.a"

remark: linking in "rts64plus.lib" in place of "libc.a"
```

5.4.2 Run-Time Initialization

You must link all C/C++ programs with code to initialize and execute the program, called a *bootstrap* routine, also known as the *boot.obj* object module. When a C/C++ program begins running, it must execute boot.obj first. The boot.obj module contains code and data to initialize the run-time environment; the link step automatically extracts boot.obj and links it when you use --rom_model and include the appropriate run-time-support library in the link.

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

1. Sets up the stack and configuration registers
2. Processes the *.cinit* run-time initialization table and autoinitializes global variables (when using the --rom_model option)
3. Calls all global constructors (*.pinit*)
4. Calls main
5. Calls exit when main returns

A sample bootstrap routine is _c_int00, provided in boot.obj in the run-time support object libraries. The *entry point* is usually set to the starting address of the bootstrap routine.

Note: The _c_int00 Symbol

If you use the --ram_model or --rom_model link option, _c_int00 is automatically defined as the entry point for the program.

5.4.3 Global Object Constructors

Global C++ variables that have constructors and destructors require their constructors to be called during program initialization and their destructors to be called during program termination. The C++ compiler produces a table of constructors to be called at startup.

The constructors are invoked in the order that they occur in the table.

Global constructors are called after initialization of other global variables and before main( ) is called. Global destructors are invoked during exit( ), similar to functions registered through atexit( ).

Section 7.8.3 discusses the format of the global constructor table.

5.4.4 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 7.8.3 discusses the format of these initialization tables. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the --rom_model linker option (see Section 7.8.4).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 7.8.5).

When you link a C/C++ program, you must use either the --rom_model or --ram_model option. These options tell the linker to select initialization at run time or load time.

When you compile and link programs, the --rom_model option is the default. If used, the --rom_model option must follow the --run_linker option (see Section 5.1). The following list outlines the linking conventions used with --rom_model or --ram_model:

- The symbol _c_int00 is defined as the program entry point; it identifies the beginning of the C/C++ boot routine in boot.obj. When you use --rom_model or --ram_model, _c_int00 is automatically referenced, ensuring that boot.obj is automatically linked in from the run-time-support library.
- The initialization output section is padded with a termination record so that the loader (load-time initialization) or the boot routine (run-time initialization) knows when to stop reading the initialization tables.
- When initializing at load time (the --ram_model option), the following occur:
  - The linker sets the initialization table symbol to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag is set in the initialization table section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the initialization table into memory. The linker does not allocate space in memory for the initialization table.
- When autoinitializing at run time (--rom_model option), the linker defines the initialization table symbol as the starting address of the initialization table. The boot routine uses this symbol as the starting point for autoinitialization.

5.4.5 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 5-1 summarizes the initialized sections. Table 5-2 summarizes the uninitialized sections.

Table 5-1. Initialized Sections Created by the Compiler

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and contain string literals
.pinit	Table of constructors to be called at startup
.switch	Jump tables for large switch statements
.text	Executable code and constants

Table 5-2. Uninitialized Sections Created by the Compiler

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 7.1.1 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 *TMS320C6000 Assembly Language Tools User's Guide*.

5.4.6 A Sample Linker Command File

Example 5-1 shows a typical link step command file that links a C program. The command file in this example is named lnk.cmd and lists several link step options:

--rom_model	Tells the link step to use autoinitialization at run time.
--heap_size	Tells the link step to set the C heap size at 0x2000 bytes.
--stack_size	Tells the link step to set the stack size to 0x0100 bytes.
--library	Tells the link step to use an archive library file, rts6200.lib, for input.

To link the program, use the following syntax:

cl6x --run_linker *object_file(s)* **--output_file=***outfile* **--map_file=***mapfile* **lnk.cmd**

The MEMORY and possibly the SECTIONS directives, might require modification to work with your system. See the *C6000 Assembly Language Tools User's Guide* for more information on these directives.

Example 5-1. Sample Link Step Command File

```
--rom_model
--heap_size=0x2000
--stack_size=0x0100
--library=rts6200.lib

MEMORY
{
    VECS:   o = 0x00000000     l = 0x000000400 /* reset & interrupt vectors     */
    PMEM:   o = 0x00000400     l = 0x00000FC00 /* intended for initialization   */
    BMEM:   o = 0x80000000     l = 0x000010000 /* .bss, .sysmem, .stack, .cinit */
}

SECTIONS
{
    vectors     >       VECS
    .text       >       PMEM
    .data       >       BMEM
    .stack      >       BMEM
    .bss        >       BMEM
    .sysmem     >       BMEM
    .cinit      >       BMEM
    .const      >       BMEM
    .cio        >       BMEM
    .far        >       BMEM
}
```

TMS320C6000 C/C++ Language Implementation

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

The C++ language supported by the C6000 is defined by the ANSI/ISO/IEC 14882-1998 standard with certain exceptions.

Topic **Page**

6.1 Characteristics of TMS320C6000 C

The compiler supports the C language as defined by ISO 9899, which is equivalent to American National Standard for Information Systems-Programming Language C X3.159-1989 (C89). The compiler does not support C99.

Unsupported features of the C library are:

- The run-time library has minimal support for wide and multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h>, but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.

6.2 Characteristics of TMS320C6000 C++

The C6000 compiler supports C++ as defined in the ANSI/ISO/IEC 14882:1998 standard, including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 6.5.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

The *exceptions* to the standard are as follows:

- The library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described above in the C library.
- If the definition of an inline function contains a static variable, and it appears in multiple compilation units (usually because it's a member function of a class defined in a header file), the compiler generates multiple copies of the static variable rather than resolving them to a single definition. The compiler emits a warning (#1369) in such cases.
- Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- Template parameters are not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.

6.3 Data Types

Table 6-1 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.

Table 6-1. TMS320C6000 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	8 bits	ASCII	-128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	40 bits	2s complement	-549 755 813 888	549 755 813 887
unsigned long	40 bits	Binary	0	1 099 511 627 775
long long, signed long long	64 bits	2s complement	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum	32 bits	2s complement	-2 147 483 648	2 147 483 647
float	32 bits	IEEE 32-bit	1.175 494e-38 (1)	3.40 282 346e+38
double	64 bits	IEEE 64-bit	2.22 507 385e-308 (1)	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e-308 (1)	1.79 769 313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

(1) Figures are minimum precision.

6.4 Keywords

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

6.4.1 The const Keyword

The TMS320C6000 C/C++ compiler supports the ANSI/ISO standard keyword *const*. This keyword gives you greater optimization and control over allocation of storage for certain data objects. You can apply the const qualifier to the definition of any variable or array to ensure that its value is not altered.

If you define an object as far const, the .const section allocates storage for the object. The const data storage allocation rule has two exceptions:

- If the keyword volatile is also specified in the definition of an object (for example, volatile const int x). Volatile keywords are assumed to be allocated to RAM. (The program does not modify a const volatile object, but something external to the program might.)
- If the object has automatic storage (allocated on the stack).

In both cases, the storage for the object is the same as if the const keyword were not used.

The placement of the const keyword within a definition is important. For example, the first statement below defines a constant pointer p to a variable int. The second statement defines a variable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
far const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

6.4.2 The cregister Keyword

The compiler extends the C/C++ language by adding the cregister keyword to allow high level language access to control registers.

When you use the cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers for the C6000 (see Table 6-2). 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 6-2. Valid Control Registers

Register	Description
AMR	Addressing mode register
CSR	Control status register
DESR	(C6700+ only) dMAX event status register
DETR	(C6700+ only) dMAX event trigger register
DNUM	(C6400+ and C6740 only) DSP core number register
ECR	(C6400+ and C6740 only) Exception clear register
EFR	(C6400+ and C6740 only) Exception flag register
FADCR	(C6700 and C6700+ only) Floating-point adder configuration register
FAUCR	(C6700 and C6700+ only) Floating-point auxiliary configuration register
FMCR	(C6700 and C6700+ only) Floating-point multiplier configuration register
GFPGFR	(C6400 only) Galois field polynomial generator function register
GPLYA	(C6400+ and C6740 only) GMPY A-side polynomial register
CPLYB	(C6400+ and C6740 only) GMPY B-side polynomial register
ICR	Interrupt clear register

Table 6-2. Valid Control Registers (continued)

Register	Description
IER	Interrupt enable register
IERR	(C6400+ and C6740 only) Internal exception report register
IFR	Interrupt flag register
ILC	(C6400+ and C6740 only) Inner loop count register
IRP	Interrupt return pointer
ISR	Interrupt set register
ISTP	Interrupt service table pointer
ITSR	(C6400+ and C6740 only) Interrupt task state register
NRP	Nonmaskable interrupt return pointer
NTSR	(C6400+ and C6740 only) NMI/exception task state register
REP	(C6400+ and C6740 only) Restricted entry point address register
RILC	(C6400+ and C6740 only) Reload inner loop count register
SSR	(C6400+ and C6740 only) Saturation status register
TSCH	(C6400+ and C6740 only) Time-stamp counter (high 32) register
TSCL	(C6400+ and C6740 only) Time-stamp counter (low 32) register
TSR	(C6400+ and C6740 only) Task state register

The cregister keyword can be used only 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 6-2, you must declare each register as follows. The c6x.h include file defines all the control registers through this syntax:

extern cregister volatile unsigned int *register* **;**

Once you have declared the register, you can use the register name directly. IFR is read only. See the*TMS320C62x DSP CPU and Instruction Set Reference Guide, TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide, or TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide* for detailed information on the control registers.

See Example 6-1 for an example that declares and uses control registers.

Example 6-1. 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);
}
```

6.4.3 The interrupt Keyword

The compiler extends the C/C++ language by adding the interrupt keyword, which specifies that a function is treated as an interrupt function.

Functions that handle interrupts follow special register-saving rules and a special return sequence. When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

You can only use the interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables. For example:

```
interrupt void int_handler()
{
    unsigned int flags;
    ...
}
```

The name c_int00 is the C/C++ entry point. This name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main. Because it has no caller, c_int00 does not save any registers.

Use the alternate keyword, __interrupt, if you are writing code for strict ANSI/ISO mode (using the --strict_ansi compiler option).

HWI Objects and the interrupt Keyword

Note: The interrupt keyword must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause catastrophic results.

6.4.4 The near and far Keywords

The C6000 C/C++ compiler extends the C/C++ language with the near and far keywords to specify how global and static variables are accessed and how functions are called.

Syntactically, the near and far keywords are treated as storage class modifiers. They can appear before, after, or in between the storage class specifiers and types. With the exception of near and far, two storage class modifiers cannot be used together in a single declaration. The following examples are legal combinations of near and far with other storage class modifiers:

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

6.4.4.1 near and far Data Objects

Global and static data objects can be accessed in the following two ways:

near keyword The compiler assumes that the data item can be accessed relative to the data page pointer. For example:

```
LDW *+dp(_address),a0
```

far keyword The compiler cannot access the data item via the DP. This can be required if the total amount of program data is larger than the offset allowed (32K) from the DP. For example:

```
MVKL  _address,a1
MVKH  _address,a1
LDW *a1,a0
```

Once a variable has been defined to be far, all external references to this variable in other C files or headers must also contain the far keyword. This is also true of the near keyword. However, you will get compiler or linker errors when the far keyword is not used everywhere. Not using the near keyword everywhere only leads to slower data access times.

If you use the DATA_SECTION pragma, the object is indicated as a far variable, and this cannot be overridden. If you reference this object in another file, then you need to use *extern far* when declaring this object in the other source file. This ensures access to the variable, since the variable might not be in the .bss section. For details, see Section 6.8.4.

Note: Defining Global Variables in Assembly Code

If you also define a global variable in assembly code with the .usect directive (where the variable is not assigned in the .bss section) or you allocate a variable into separate section using a #pragma DATA_SECTION directive; and you want to reference that variable in C code, you must declare the variable as extern far. This ensures the compiler does not try to generate an illegal access of the variable by way of the data page pointer.

When data objects do not have the near or far keyword specified, the compiler will use far accesses to aggregate data and near accesses to non-aggregate data. For more information on the data memory model and ways to control accesses to data, see Section 7.1.5.1.

6.4.4.2 Near and far Function Calls

Function calls can be invoked in one of two ways:

near keyword The compiler assumes that destination of the call is within ± 1 M word of the caller. Here the compiler uses the PC-relative branch instruction.

```
B _func
```

far keyword The compiler is told by you that the call is not within ± 1 M word.

```
MVKL  _func,al
MVKH  _func,al
B _func
```

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.

For more information on function calls, see Section 7.1.6.

6.4.5 The restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that can be applied to pointers, references, and arrays. Its use represents a guarantee by you, the programmer, that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In Example 6-2, the restrict keyword is used to tell the compiler that the function func1 is never called with the pointers a and b pointing to objects that overlap in memory. You are promising that accesses through a and b will never conflict; therefore, a write through one pointer cannot affect a read from any other pointers. The precise semantics of the restrict keyword are described in the 1999 version of the ANSI/ISO C Standard.

Example 6-2. Use of the restrict Type Qualifier With Pointers

```
void func1(int * restrict a, int * restrict b)
{
  /* func1's code here */
}
```

Example 6-3 illustrates using the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

Example 6-3. Use of the restrict Type Qualifier With Arrays

```
void func2(int c[restrict], int d[restrict])
{
  int i;

  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

6.4.6 The volatile Keyword

The compiler analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C/C++ code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section (as opposed to a register). The compiler does not optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

In this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single-memory read. To correct this, define *ctrl as:

```
volatile unsigned int *ctrl;
```

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.

Consider using the --interrupt_threshold=1 option when compiling with volatiles.

6.5 C++ Exception Handling

The compiler supports all the C++ exception handling features as defined by the ANSI/ISO 14882 C++ Standard. More details are discussed in *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

The compiler --exceptions option enables exception handling. The compiler's default is no exception handling support.

For exceptions to work correctly, all C++ files in the application must be compiled with the --exceptions option, regardless of whether exceptions occur in a particular file. Mixing exception-enabled object files and libraries with object files and libraries that do not have exceptions enabled can lead to undefined behavior. Also, when using --exceptions, you need to link with run-time-support libraries whose name contains _eh. These libraries contain functions that implement exception handling.

Using --exceptions causes code size to increaseexceptions option increase.

See Section 8.1 for details on the run-time libraries.

6.6 Register Variables and Parameters

The C/C++ compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the --opt_level (-O) option.

- **Compiling with optimization**
  The compiler ignores any register definitions and allocates registers to variables and temporary values by using an algorithm that makes the most efficient use of registers.
- **Compiling without optimization**
  If you use the register keyword, you can suggest variables as candidates for allocation into registers. The compiler uses the same set of registers for allocating temporary expression results as it uses for allocating register variables.

The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. This limit causes excessive movement of register contents to memory.

Any object with a scalar type (integral, floating point, or pointer) can be defined as a register variable. The register designator is ignored for objects of other types, such as arrays.

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. The compiler copies a register parameter to a register instead of the stack, which speeds access to the parameter within the function.

For more information about register conventions, see Section 7.3.

6.7 The asm Statement

The C/C++ compiler can embed assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language—the *asm* statement. The asm (or __asm) statement provides access to hardware features that C/C++ cannot provide. The asm statement is syntactically like a call to a function named asm, with one string constant argument:

asm(" *assembler text* **");**

The compiler copies the argument string directly into your output file. The assembler text must be enclosed in double quotes. All the usual character string escape codes retain their definitions. For example, you can insert a .byte directive that contains quotes as follows:

```
asm("STR: .byte \"abc\"");
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about the assembly language statements, see the *TMS320C6000 Assembly Language Tools User's Guide*.

The asm statements do not follow the syntactic restrictions of normal C/C++ statements. Each can appear as a statement or a declaration, even outside of blocks. This is useful for inserting directives at the very beginning of a compiled module.

Use the alternate statement __asm("assembler text") if you are writing code for strict ANSI/ISO C mode (using the --strict_ansi option).

Note: Avoid Disrupting the C/C++ Environment With asm Statements

Be careful not to disrupt the C/C++ environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C/C++ code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use optimization with asm statements. Although the compiler cannot remove asm statements, it can significantly rearrange the code order near them and cause undesired results.

6.8 Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. The C6000 C/C++ compiler supports the following pragmas:

- CODE_SECTION
- DATA_ALIGN
- DATA_MEM_BANK
- DATA_SECTION
- FUNC_ALWAYS_INLINE
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_INTERRUPT_THRESHOLD
- FUNC_IS_PURE
- FUNC_IS_SYSTEM
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- INTERRUPT
- MUST_ITERATE
- NMI_INTERRUPT
- NO_HOOKS
- PROB_ITERATE
- STRUCT_ALIGN
- UNROLL

Most of these pragmas apply to functions. Except for the DATA_MEM_BANK pragma, the arguments *func* and *symbol* cannot be defined or declared inside the body of a function. Pragmas that apply to functions must be specified outside the body of a function; and the pragma specification 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.

For the pragmas that apply to functions or symbols, the syntax for the pragmas differs between C and C++. In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. In C++, the name is omitted; the pragma applies to the declaration of the object or function that follows it.

6.8.1 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in a section named *section name*.

The syntax of the pragma in C is:

#pragma CODE_SECTION (*symbol* **, "** *section name* **");**

The syntax of the pragma in C++ is:

#pragma CODE_SECTION (" *section name* **");**

The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

The following examples demonstrate the use of the CODE_SECTION pragma.

Example 6-4. Using the CODE_SECTION Pragma C Source File

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

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

Example 6-5. Generated Assembly Code From Example 6-4

```
    .sect    "my_sect"
    .global  _fn

;******************************************************************************
;* FUNCTION NAME: _fn                                                         *
;*                                                                            *
;*   Regs Modified     : SP                                                   *
;*   Regs Used         : A4,B3,SP                                             *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
_fn:
;** --------------------------------------------------------------------------*
           RET      .S2      B3                 ; |6|
           SUB      .D2      SP,8,SP            ; |4|
           STW      .D2T1    A4,*+SP(4)         ; |4|
           ADD      .S2      8,SP,SP            ; |6|
           NOP               2
           ; BRANCH OCCURS                      ; |6|
```

6.8.2 The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

6.8.3 The DATA_MEM_BANK Pragma

The DATA_MEM_BANK pragma aligns a symbol or variable to a specified C6000 internal data memory bank boundary. The *constant* specifies a specific memory bank to start your variables on. (See Figure 4-1 for a graphic representation of memory banks.) The value of *constant* depends on the C6000 device:

C6200	The C6200 devices contain four memory banks (0, 1, 2, and 3); *constant* can be 0 or 2.
C6400	The C6400 devices contain 8 memory banks; *constant* can be 0, 2, 4, or 6.
C6400+	The C6400+ devices contain 8 memory banks; *constant* can be 0, 2, 4, or 6.
C6700	The C6700 devices contain 8 memory banks; *constant* can be 0, 2, 4, or 6.
C6740	The C6740 devices contain 8 memory banks; *constant* can be 0, 2, 4, or 6.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma DATA_MEM_BANK (*constant* **);**

Both global and local variables can be aligned with the DATA_MEM_BANK pragma. The DATA_MEM_BANK pragma must reside inside the function that contains the local variable being aligned. The *symbol* can also be used as a parameter in the DATA_SECTION pragma.

When optimization is enabled, the tools may or may not use the stack to store the values of local variables.

The DATA_MEM_BANK pragma allows you to align data on any data memory bank that can hold data of the type size of the *symbol*. This is useful if you need to align data in a particular way to avoid memory bank conflicts in your hand-coded assembly code versus padding with zeros and having to account for the padding in your code.

This pragma increases the amount of space used in data memory by a small amount as padding is used to align data onto the correct bank.

For C6200, the code in Example 6-6 guarantees that array x begins at an address ending in 4 or c (in hexadecimal), and that array y begins at an address ending in 4 or c. The alignment for array y affects its stack placement. Array z is placed in the .z_sect section, and begins at an address ending in 0 or 8.

Example 6-6. Using the DATA_MEM_BANK Pragma

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

#pragma DATA_MEM_BANK (z, 0);
#pragma DATA_SECTION (z, ".z_sect");
short z[100];

void main()
{
    #pragma DATA_MEM_BANK (y, 2);
    short y[100];
    ...
}
```

6.8.4 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in a section named *section name*.

The syntax of the pragma in C is:

#pragma DATA_SECTION (*symbol* **, "** *section name* **");**

The syntax of the pragma in C++ is:

#pragma DATA_SECTION (" *section name* **");**

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section. If you allocate a global variable using a DATA_SECTION pragma and you want to reference the variable in C code, you must declare the variable as extern far.

Example 6-7 through Example 6-9 demonstrate the use of the DATA_SECTION pragma.

Example 6-7. Using the DATA_SECTION Pragma C Source File

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

Example 6-8. Using the DATA_SECTION Pragma C++ Source File

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

Example 6-9. Using the DATA_SECTION Pragma Assembly Source File

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

6.8.5 The FUNC_ALWAYS_INLINE Pragma

The FUNC_ALWAYS_INLINE pragma instructs the compiler to always inline the named function. The compiler only inlines the function if it is legal to inline the function and the compiler is invoked with any level of optimization (--opt_level=0).

The pragma must appear before any declaration or reference to the function that you want to inline. In C, the argument *func* is the name of the function that will be inlined. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

#pragma FUNC_ALWAYS_INLINE (*func* **);**

The syntax of the pragma in C++ is:

#pragma FUNC_ALWAYS_INLINE;

Use Caution with the FUNC_ALWAYS_INLINE Pragma

Note: The FUNC_ALWAYS_INLINE pragma overrides the compiler's inlining decisions. Overuse of the pragma could result in increased compilation times or memory usage, potentially enough to consume all available memory and result in compilation tool failures.

6.8.6 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see Section 2.11.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that cannot be inlined. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_CANNOT_INLINE;

6.8.7 The FUNC_EXT_CALLED Pragma

When you use the --program_level_compile option, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C/C++ functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep these C functions or any other functions that these C/C++ functions call. These functions act as entry points into C/C++.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that you do not want removed. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_EXT_CALLED;

Except for _c_int00, which is the name reserved for the system reset interrupt for C/C++programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See Section 3.7.2.

6.8.8 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 --interrupt_threshold option for a single function (see Section 2.12). The FUNC_INTERRUPT_THRESHOLD pragma always overrides the --interrupt_threshold=*n* command line option. A threshold value less than 0 assumes that the function is never interrupted, which is equivalent to an interrupt threshold of infinity.

The syntax of the pragma in C is:

#pragma FUNC_INTERRUPT_THRESHOLD (*func* **,** *threshold* **);**

The syntax of the pragma in C++ is:

#pragma FUNC_INTERRUPT_THRESHOLD (*threshold* **);**

The following examples demonstrate the use of different thresholds:

- The function foo( ) must be interruptible at least every 2,000 cycles:
  ```
  #pragma FUNC_INTERRUPT_THRESHOLD (foo, 2000)
  ```
- The function foo( ) must always be interruptible.
  ```
  #pragma FUNC_INTERRUPT_THRESHOLD (foo, 1)
  ```
- The function foo( ) is never interrupted.
  ```
  #pragma FUNC_INTERRUPT_THRESHOLD (foo, -1)
  ```

6.8.9 The FUNC_IS_PURE Pragma

The FUNC_IS_PURE pragma specifies to the compiler that the named function has no side effects. This allows the compiler to do the following:

- Delete the call to the function if the function's value is not needed
- Delete duplicate functions

The pragma must appear before any declaration or reference to the function. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_IS_PURE;

6.8.10 The FUNC_IS_SYSTEM Pragma

The FUNC_IS_SYSTEM pragma specifies to the compiler that the named function has the behavior defined by the ANSI/ISO standard for a function with that name.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function to treat as an ANSI/ISO standard function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_IS_SYSTEM ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_SYSTEM;
```

6.8.11 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the compiler that the function never returns to its caller.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that does not return. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NEVER_RETURNS ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NEVER_RETURNS;
```

6.8.12 The FUNC_NO_GLOBAL_ASG Pragma

The FUNC_NO_GLOBAL_ASG pragma specifies to the compiler that the function makes no assignments to named global variables and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NO_GLOBAL_ASG ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_GLOBAL_ASG;
```

6.8.13 The FUNC_NO_IND_ASG Pragma

The FUNC_NO_IND_ASG pragma specifies to the compiler that the function makes no assignments through pointers and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_NO_IND_ASG;

6.8.14 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma INTERRUPT;

The code for the function will return via the IRP (interrupt return pointer).

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

HWI Objects and the INTERRUPT Pragma

Note: The INTERRUPT pragma must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause catastrophic results.

6.8.15 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. You guarantee that these properties are always true. Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a specific number of times. Anytime the UNROLL pragma is applied to a loop, MUST_ITERATE should be applied to the same loop. For loops the MUST_ITERATE pragma's third argument, multiple, is the most important and should always be specified.

Furthermore, the MUST_ITERATE pragma should be applied to any other loops as often as possible. This is because the information provided via the pragma (especially the minimum number of iterations) aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). It also helps the compiler reduce code size.

No statements are allowed between the MUST_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and PROB_ITERATE, can appear between the MUST_ITERATE pragma and the loop.

6.8.15.1 The MUST_ITERATE Pragma Syntax

The syntax of the pragma for C and C++ is:

#pragma MUST_ITERATE (*min*, *max*, *multiple***);**

The arguments *min* and *max* are programmer-guaranteed minimum and maximum trip counts. The trip count is the number of times a loop iterates. The trip count of the loop must be evenly divisible by *multiple*. All arguments are optional. For example, if the trip count could be 5 or greater, you can specify the argument list as follows:

```
#pragma MUST_ITERATE(5);
```

However, if the trip count could be any nonzero multiple of 5, the pragma would look like this:

```
#pragma MUST_ITERATE(5, , 5); /* Note the blank field for max */
```

It is sometimes necessary for you to provide min and multiple in order for the compiler to perform unrolling. This is especially the case when the compiler cannot easily determine how many iterations the loop will perform (that is, the loop has a complex exit condition).

When specifying a multiple via the MUST_ITERATE pragma, results of the program are undefined if the trip count is not evenly divisible by multiple. Also, results of the program are undefined if the trip count is less than the minimum or greater than the maximum specified.

If no min is specified, zero is used. If no max is specified, the largest possible number is used. If multiple MUST_ITERATE pragmas are specified for the same loop, the smallest max and largest min are used.

6.8.15.2 Using MUST_ITERATE to Expand Compiler Knowledge of Loops

Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a certain number of times. The example below tells the compiler that the loop is guaranteed to run exactly 10 times:

```
#pragma MUST_ITERATE(10,10);

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

In this example, the compiler attempts to generate a software pipelined loop even without the pragma. However, if MUST_ITERATE is not specified for a loop such as this, the compiler generates code to bypass the loop, to account for the possibility of 0 iterations. With the pragma specification, the compiler knows that the loop iterates at least once and can eliminate the loop-bypassing code.

MUST_ITERATE can specify a range for the trip count as well as a factor of the trip count. For example:

```
#pragma MUST_ITERATE(8, 48, 8);

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

This example tells the compiler that the loop executes between 8 and 48 times and that the trip_count variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The multiple argument allows the compiler to unroll the loop.

You should also consider using MUST_ITERATE for loops with complicated bounds. In the following example:

```
for(i2 = ipos[2]; i2 <; 40; i2 += 5)  { ...
```

The compiler would have to generate a divide function call to determine, at run time, the exact number of iterations performed. The compiler will not do this. In this case, using MUST_ITERATE to specify that the loop always executes eight times allows the compiler to attempt to generate a software pipelined loop:

```
#pragma MUST_ITERATE(8, 8);

for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

6.8.16 The NMI_INTERRUPT Pragma

The NMI_INTERRUPT pragma enables you to handle non-maskable interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma NMI_INTERRUPT;

The code generated for the function will return via the NRP versus the IRP as for a function declared with the interrupt keyword or INTERRUPT pragma.

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (function) does not need to conform to a naming convention.

6.8.17 The NO_HOOKS Pragma

The NO_HOOKS pragma prevents entry and exit hook calls from being generated for a function.

The syntax of the pragma in C is:

#pragma NO_HOOKS(*func* **);**

The syntax of the pragma in C++ is:

#pragma NO_HOOKS;

See Section 2.15 for details on entry and exit hooks.

6.8.18 The PROB_ITERATE Pragma

The PROB_ITERATE pragma specifies to the compiler certain properties of a loop. You assert that these properties are true in the common case. The PROB_ITERATE pragma aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). PROB_ITERATE is useful only when the MUST_ITERATE pragma is not used or the PROB_ITERATE parameters are more constraining than the MUST_ITERATE parameters.

No statements are allowed between the PROB_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and MUST_ITERATE, may appear between the PROB_ITERATE pragma and the loop.

The syntax of the pragma for C and C++ is:

#pragma PROB_ITERATE(*min* **,** *max* **);**

Where min and max are the minimum and maximum trip counts of the loop in the common case. The trip count is the number of times a loop iterates. Both arguments are optional.

For example, PROB_ITERATE could be applied to a loop that executes for eight iterations in the majority of cases (but sometimes may execute more or less than eight iterations):

```
#pragma PROB_ITERATE(8, 8);
```

If only the minimum expected trip count is known (say it is 5), the pragma would look like this:

```
#pragma PROB_ITERATE(5);
```

If only the maximum expected trip count is known (say it is 10), the pragma would look like this:

```
#pragma PROB_ITERATE(, 10);  /* Note the blank field for min */
```

6.8.19 The STRUCT_ALIGN Pragma

The STRUCT_ALIGN pragma is similar to DATA_ALIGN, but it can be applied to a structure, union type, or typedef and is inherited by any symbol created from that type. The STRUCT_ALIGN pragma is supported only in C.

The syntax of the pragma is:

#pragma STRUCT_ALIGN(*type* **,** *constant expression* **);**

This pragma guarantees that the alignment of the named type or the base type of the named typedef is at least equal to that of the expression. (The alignment may be greater as required by the compiler.) The alignment must be a power of 2. The *type* must be a type or a typedef name. If a type, it must be either a structure tag or a union tag. If a typedef, its base type must be either a structure tag or a union tag.

Since ANSI/ISO C declares that a typedef is simply an alias for a type (i.e. a struct) this pragma can be applied to the struct, the typedef of the struct, or any typedef derived from them, and affects all aliases of the base type.

This example aligns any st_tag structure variables on a page boundary:

```
typedef struct st_tag
{
   int   a;
   short b;
} st_typedef;

#pragma STRUCT_ALIGN (st_tag, 128);
#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.

6.8.20 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. The UNROLL pragma is useful for helping the compiler utilize SIMD instructions on the C6400 family. It is also useful in cases where better utilization of software pipeline resources are needed over a non-unrolled loop.

The optimizer must be invoked (use --opt_level=[1|2|3] or -O1, -O2, or -O3) in order for pragma-specified loop unrolling to take place. The compiler has the option of ignoring this pragma.

No statements are allowed between the UNROLL pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as MUST_ITERATE and PROB_ITERATE, can appear between the UNROLL pragma and the loop.

The syntax of the pragma for C and C++ is:

#pragma UNROLL(*n* **);**

If possible, the compiler unrolls the loop so there are *n* copies of the original loop. The compiler only unrolls if it can determine that unrolling by a factor of *n* is safe. In order to increase the chances the loop is unrolled, the compiler needs to know certain properties:

- The loop iterates a multiple of *n* times. This information can be specified to the compiler via the multiple argument in the MUST_ITERATE pragma.
- The smallest possible number of iterations of the loop
- The largest possible number of iterations of the loop

The compiler can sometimes obtain this information itself by analyzing the code. However, sometimes the compiler can be overly conservative in its assumptions and therefore generates more code than is necessary when unrolling. This can also lead to not unrolling at all.

Furthermore, if the mechanism that determines when the loop should exit is complex, the compiler may not be able to determine these properties of the loop. In these cases, you must tell the compiler the properties of the loop by using the MUST_ITERATE pragma.

Specifying #pragma UNROLL(1); asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

If multiple UNROLL pragmas are specified for the same loop, it is undefined which pragma is used, if any.

6.9 Generating Linknames

The compiler transforms the names of externally visible identifiers when creating their linknames. The algorithm used depends on the scope within which the identifier is declared. For objects and C functions, an underscore (_) is prefixed to the identifier name. C++ functions are prefixed with an underscore also, but the function name is modified further.

Mangling is the process of embedding a function's signature (the number and types of its parameters) into its name. Mangling occurs only in C++ code. The mangling algorithm used closely follows that described in *The Annotated Reference Manual* (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

For example, the general form of a C++ linkname for a function named func is:

_func__F *parmcodes*

Where parmcodes is a sequence of letters that encodes the parameter types of func.

For this simple C++ source file:

```
int foo(int i){ }   //global C++ function
```

This is the resulting assembly code:

```
_foo__Fi
```

The linkname of foo is _foo__Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided that demangles names into those found in the original C++ source. See Chapter 9 for more information.

6.10 Initializing Static and Global Variables

The ANSI/ISO C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running. This task is typically done when the program is loaded. Because the loading process is heavily dependent on the specific environment of the target application system, the compiler itself makes no provision for preinitializing variables at run time. It is up to your application to fulfill this requirement.

6.10.1 Initializing Static and Global Variables With the Linker

If your loader does not preinitialize variables, you can use the linker to preinitialize the variables to 0 in the object file. For example, in the linker command file, use a fill value of 0 in the .bss section:

```
SECTIONS
   {
      ...

      .bss: {} = 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.*

6.10.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in Section 6.10). For example:

```
const int zero;       /*  may not be initialized to 0  */
```

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .const. For example:

```
const int zero = 0   /*  guaranteed to be 0  */
```

This corresponds to an entry in the .const section:

```
   .sect   .const
_zero
   .word   0
```

This feature is particularly useful for declaring a large table of constants, because neither time nor space is wasted at system startup to initialize the table. Additionally, the linker can be used to place the .const section in ROM.

You can use the DATA_SECTION pragma to put the variable in a section other than .const. For example, the following C code:

```
#pragma DATA_SECTION (var, ".mysect");
   const int zero=0;
```

is compiled into this assembly code:

```
   .sect   .mysect
_zero
   .word   0
```

6.11 Changing the ANSI/ISO C Language Mode

The --kr_compatible, --relaxed_ansi, and --strict_ansi options let you specify how the C/C++ compiler interprets your source code. You can compile your source code in the following modes:

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

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

K&R C mode does not apply to C++ code.

6.11.1 Compatibility With K&R C (--kr_compatible Option)

The ANSI/ISO C/C++ language is a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ANSI/ISO compilers correctly compile and run without modification.

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ANSI/ISO C and the first edition's C standard (the first edition is referred to in this manual as K&R C).

To simplify the process of compiling existing C programs with the ANSI/ISO C/C++ compiler, the compiler has a K&R option (--kr_compatible) that modifies some semantic rules of the language for compatibility with older code. In general, the --kr_compatible option relaxes requirements that are stricter for ANSI/ISO C than for K&R C. The --kr_compatible option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, --kr_compatible simply liberalizes the ANSI/ISO rules without revoking any of the features.

The specific differences between the ANSI/ISO version of C and the K&R version of C are as follows:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider signed type. Under K&R C, the result type was an unsigned version of the wider type; under ANSI/ISO, the result type is a signed version of the wider type. This affects operations that perform differently when applied to signed or unsigned operands; namely, comparisons, division (and mod), and right shift:

```
unsigned short u;
int i;
if (u < i)          /* SIGNED comparison, unless --kr_compatible used */
```

- ANSI/ISO prohibits combining two pointers to different types in an operation. In most K&R compilers, this situation produces only a warning. Such cases are still diagnosed when --kr_compatible is used, but with less severity:

```
int *p;
char *q = p;        /* error without --kr_compatible, warning with --kr_compatible */
```

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

```
a;                  /* illegal unless --kr_compatible used */
```

- ANSI/ISO interprets file scope definitions that have no initializers as *tentative definitions.* In a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object and usually an error. For example:

```
int a;
int a;              /* illegal if --kr_compatible used, OK if not */
```

  Under ANSI/ISO, the result of these two definitions is a single definition for the object a. For most K&R compilers, this sequence is illegal, because int a is defined twice.

- ANSI/ISO prohibits, but K&R allows objects with external linkage to be redeclared as static:

```
extern int a;
static int a;       /* illegal unless --kr_compatible used */
```

- Unrecognized escape sequences in string and character constants are explicitly illegal under ANSI/ISO but ignored under K&R:

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

- ANSI/ISO specifies that bit fields must be of type int or unsigned. With --kr_compatible, bit fields can be legally defined with any integral type. For example:

```
struct s
{
   short f : 2;     /* illegal unless --kr_compatible used */
};
```

- K&R syntax allows a trailing comma in enumerator lists:

```
enum { a, b, c, }; /* illegal unless --kr_compatible used */
```

- K&R syntax allows trailing tokens on preprocessor directives:

```
#endif NAME         /* illegal unless --kr_compatible used */
```

6.11.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (--strict_ansi and --relaxed_ansi Options)

Use the --strict_ansi option when you want to compile under strict ANSI/ISO mode. In this mode, error messages are provided when non-ANSI/ISO features are used, and language extensions that could invalidate a strictly conforming program are disabled. Examples of such extensions are the inline and asm keywords.

Use the --relaxed_ansi option when you want the compiler to ignore strict ANSI/ISO violations rather than emit a warning (as occurs in normal ANSI/ISO mode) or an error message (as occurs in strict ANSI/ISO mode). In relaxed ANSI/ISO mode, the compiler accepts extensions to the ANSI/ISO C standard, even when they conflict with ANSI/ISO C.

6.11.3 Enabling Embedded C++ Mode (--embedded_cpp Option)

The compiler supports the compilation of embedded C++. In this mode, some features of C++ are removed that are of less value or too expensive to support in an embedded system. When compiling for embedded C++, the compiler generates diagnostics for the use of omitted features.

Embedded C++ is enabled by compiling with the --embedded_cpp option.

Embedded C++ omits these C++ features:

- Templates
- Exception handling
- Run-time type information
- The new cast syntax
- The keyword mutable
- Multiple inheritance
- Virtual inheritance

Under the standard definition of embedded C++, namespaces and using-declarations are not supported. The C6000 compiler nevertheless allows these features under embedded C++ because the C++ run-time-support library makes use of them. Furthermore, these features impose no run-time penalty.

6.12 GNU C Compiler Extensions

The GNU compiler, GCC, provides a number of language features not found in the ANSI standard C. The definition and official examples of these extensions can be found at http://gcc.gnu.org/onlinedocs/gcc-3.4.4/gcc/CExtensions. html#C-Extensions. To enable GNU extension support, use the --gcc compiler option.

The extensions that the TI C compiler supports are listed in Table 6-3.

Table 6-3. GCC Extensions Supported

Extensions	Descriptions
Statement expressions	Putting statements and declarations inside expressions (useful for creating smart 'safe' macros)
Local labels	Labels local to a statement expression
Naming types	Giving a name to the type of an expression
typeof operator	typeof referring to the type of an expression
Generalized lvalues	Using question mark (?) and comma (,) and casts in lvalues
Conditionals	Omitting the middle operand of a ? expression
long long	Double long word integers and long long integers
Hex floats	Hexadecimal floating-point constants
Zero length	Zero-length arrays
Macro varargs	Macros with a variable number of arguments
Subscripting	Any array can be subscripted, even if it is not an lvalue.
Pointer arithmetic	Arithmetic on void pointers and function pointers
Initializers	Nonconstant initializers
Cast constructors	Constructor expressions give structures, unions, or arrays as values
Labeled elements	Labeling elements of initializers
Cast to union	Casting to union type from any member of the union
Case ranges	'Case 1 ... 9' and such

Table 6-3. GCC Extensions Supported (continued)

Extensions	Descriptions
Function attributes	Declaring that functions have no side effects, or that they can never return
Function prototypes	Prototype declarations and old-style definitions
C++ comments	C++ comments are recognized.
Dollar signs	A dollar sign is allowed in identifiers.
Character escapes	The character ESC is represented as \'e'
Alignment	Inquiring about the alignment of a type or variable
Variable attributes	Specifying the attributes of variables
Type attributes	Specifying the attributes of types
Inline	Defining inline functions (as fast as macros)
Assembly labels	Specifying the assembler name to use for a C symbol
Alternate keywords	Header files can use __const__, __asm__, etc
Incomplete enums	enum foo??
Function names	Printable strings which are the name of the current function
Return address	Getting the return or frame address of a function __builtin_return_address __builtin_frame_address
Other built-ins	Other built-in functions include: __builtin_constant_p __builtin_expect

6.12.1 Function Attributes

The GNU extension support provides a number of attributes about functions to help the C compiler's optimization. The TI compiler accepts only three of these attributes. All others are simply ignored. Table 6-4 lists the attributes that are supported.

Table 6-4. TI-Supported GCC Function Attributes

Attributes	Description
deprecated	This function exists but the compiler generates a warning if it is used.
section	Place this function in the specified section.
unused	The function is meant to be possibly not used.

6.12.2 Built-In Functions

TI provides support for only the four built-in functions in Table 6-5.

Table 6-5. TI-Supported GCC Built-In Functions

Function	Description
__builtin_constant_p(*expr*)	Returns true only if *expr* is a constant at compile time.
__builtin_expect(*expr*, CONST)	Returns *expr*. The compiler uses this function to optimize along paths determined by conditional statements such as if-else. While this function can be used anywhere in your code, it only conveys useful information to the compiler if it is the entire predicate of an if statement and CONST is 0 or 1. For example, the following indicates that you expect the predicate "a == 3" to be true most of the time: `if (__builtin_expect(a == 3, 1))`
__builtin_return_address(int *level*)	Returns 0.
__builtin_frame_address(int *level*)	Returns 0.

Chapter 7

SPRU187O–May 2008

Run-Time Environment

This chapter describes the TMS320C6000 C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

Topic **Page**

7.1 Memory Model

The C6000 compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 32-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces.

For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

7.1.1 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 object module information in the *TMS320C6000 Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. The C/C++ compiler creates the following initialized sections:
  - The **.cinit section** contains tables for initializing variables and constants.
  - The **.const section** contains string literals, floating-point constants, and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - The **.pinit section** contains the table for calling global object constructors at run time.
  - The **.switch section** contains jump tables for large switch statements.
  - The **.text section** contains all the executable code.
- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time to create and store variables. The compiler creates the following uninitialized sections:
  - The **.bss section** reserves space for global and static variables. When you specify the --rom_model 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, realloc, and new functions. If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.

Note: Use Only Code in Program Memory

With the exception of .text, the initialized and uninitialized sections cannot be allocated into internal program memory.

The assembler creates the default sections .text, .bss, and .data. The C/C++ compiler, however, does not use the .data section. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 6.8.1 and Section 6.8.4).

7.1.2 C/C++ System Stack

The C/C++ compiler uses a stack to:

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

The run-time stack grows from the high addresses to the low addresses. The compiler uses the B15 register to manage this stack. B15 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 1K bytes. You can change the stack size at link time by using the --stack_size option with the linker command. For more information on the --stack_size option, see Section 5.2.

At system initialization, SP is set to the first 8-byte aligned address before the end (highest numerical address) of the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time.

The C/C++ environment automatically decrements SP 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 the state before the function was entered. If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the same state it was in before the function was entered.

For more information about the stack and stack pointer, see Section 7.4.

Unaligned SP Can Cause Application Crash

Note: The HWI dispatcher uses SP during an interrupt call regardless of SP alignment. Therefore, SP can never be misaligned, even for 1 cycle.

Stack Overflow

Note: The compiler provides no means to check for stack overflow during compilation or at run time. Place the beginning of the .stack section in the first address after an unmapped memory space so stack overflow will cause a simulator fault. This makes this problem easy to detect. Be sure to allow enough space for the stack to grow.

7.1.3 Dynamic Memory Allocation

The run-time-support library supplied with the C6000 compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the --heap_size=*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 1K bytes. For more information on the --heap_size option, see Section 5.2.

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));
```

7.1.4 Initialization of Variables

The C/C++ compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .cinit section are stored in ROM. At system initialization time, the C/C++ boot routine copies data from these tables (in ROM) to the initialized variables in .bss (RAM).

In situations where a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at run time. You can specify this to the linker by using the --ram_model link option. For more information, see Section 7.8.

7.1.5 Data Memory Models

Several options extend the C6x data addressing model.

7.1.5.1 Determining the Data Address Model

As of the 5.1.0 version of the compiler tools, if a near or far keyword is not specified for an object, the compiler generates far accesses to aggregate data and near accesses to all other data. This means that structures, unions, C++ classes, and arrays are not accessed through the data-page (DP) pointer.

Non-aggregate data, by default, is placed in the .bss section and is accessed using relative-offset addressing from the data page pointer (DP, which is B14). DP points to the beginning of the .bss section. Accessing data via the data page pointer is generally faster and uses fewer instructions than the mechanism used for far data accesses.

If you want to use near accesses to aggregate data, you must specify the --mem_model:data=near option, or declare your data with the near keyword.

If you have too much static and extern data to fit within a 15-bit scaled offset from the beginning of the .bss section, you cannot use --mem_model:data=near. The linker will issue an error message if there is a DP-relative data access that will not reach.

The --mem_model:data=*type* option controls how data is accessed:

--mem_model:data=near	Data accesses default to near
--mem_model:data=far	Data accesses default to far
--mem_model:data=far_aggregates	Data accesses to aggregate data default to far, data accesses to non-aggregate data default to near. This is the default behavior.

The --mem_model:data options do not affect the access to objects explicitly declared with the near of far keyword.

By default, all run-time-support data is defined as far.

For more information on near and far accesses to data, see Section 6.4.4.

7.1.5.2 Using DP-Relative Addressing

The default behavior of the compiler is to use DP-relative addressing for near (.bss) data, and absolute addressing for all other (far) data. The --dprel option specifies that all data, including const data and far data, is addressed using DP-relative addressing.

The purpose of the --dprel option is to support a shared object model so multiple applications running simultaneously can share code, btu each have their own copy of the data.

7.1.5.3 Const Objects as Far

The --mem_model:const option allows const objects to be made far independently of the --mem_model:data option. This enables an application with a small amount of non-const data but a large amount of const data to move the const data out of .bss. Also, since consts can be shared, but .bss cannot, it saves memory by moving the const data into .const.

The --mem_model:const=*type* option has the following values:

--mem_model:const=data	Const objects are placed according to the --mem_model:data option. This is the default behavior.
--mem_model:const=far	Const objects default to far independent of the --mem_model:data option.
--mem_model:const=far_aggregates	Const aggregate objects default to far, scalar consts default to near.

Consts that are declared far, either explicitly through the far keyword or implicitly using --mem_model:const are always placed in the .const section.

7.1.6 Trampoline Generation for Function Calls

Beginning with the 5.1.0 release of the compiler tools, the C6000 compiler generates trampolines by default. Trampolines are a method for modifying function calls at link time to reach destinations that would normally be too far away. When a function call is more than +/- 1M instructions away from its destination, the linker will generate an indirect branch (or trampoline) to that destination, and will redirect the function call to point to the trampoline. The end result is that these function calls branch to the trampoline, and then the trampoline branches to the final destination. With trampolines, you no longer need to specify memory model options to generate far calls.

7.1.7 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), which is the data page pointer.

For some applications, it may be desirable to have multiple data pages with separate instances of near data. For example, a multi-channel application may have multiple copies of the same program running with different data pages. The functionality is supported by the C6000 compiler's memory model, and is referred to as position independent data.

Position independent data means that all near data accesses are relative to the data page (DP) pointer, allowing for the DP to be changed at run time. There are three areas where position independent data is implemented by the compiler:

- Near direct memory access

  ```
  STW  B4,*DP(_a)

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

  All near direct accesses are relative to the DP.
- 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.
- 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 7.8.3.)

7.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

7.2.1 Data Type Storage

Table 7-1 lists register and memory storage for various data types:

Table 7-1. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	Bits 0-7 of register	8 bits aligned to 8-bit boundary
unsigned char	Bits 0-7 of register	8 bits aligned to 8-bit boundary
short	Bits 0-15 of register	16 bits aligned to 16-bit boundary
unsigned short	Bits 0-15 of register	16 bits aligned to 16-bit boundary
int	Entire register	32 bits aligned to 32-bit boundary
unsigned int	Entire register	32 bits aligned to 32-bit boundary
enum	Entire register	32 bits aligned to 32-bit boundary
float	Entire register	32 bits aligned to 32-bit boundary
long	Bits 0-39 of even/odd register pair	64 bits aligned to 64-bit boundary
unsigned long	Bits 0-39 of even/odd register pair	64 bits aligned to 64-bit boundary
long long	Even/odd register pair	64 bits aligned to 64-bit boundary
unsigned long long	Even/odd register pair	64 bits aligned to 64-bit boundary
double	Even/odd register pair	64 bits aligned to 64-bit boundary
long double	Even/odd register pair	64 bits aligned to 64-bit boundary
struct	Members are stored as their individual types require.	Multiple of 8 bits aligned to boundary of largest member type; members are stored and aligned as their individual types require.
array	Members are stored as their individual types require.	Members are stored as their individual types require; for C6400 and C6400+, aligned to a 64-bit boundary; for C6200, C6700, and C6700+, aligned to a 32-bit boundary for all types 32 bits and smaller, and to a 64-bit boundary for all types larger than 32 bits. All arrays inside a structure are aligned according to the type of each element in the array.
pointer to data member	Bits 0-31 of register	32 bits aligned to 32-bit boundary
pointer to member function	Components stored as their individual types require	64 bits aligned to 32-bit boundary

7.2.1.1 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 7-1). Objects defined as short or unsigned short are stored in memory as two bytes at a halfword (2 byte) aligned address and they are loaded to and stored from bits 0-15 of a register (see Figure 7-1).

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 7-1. Char and Short Data Storage Format

Signed 8-bit char

																									MS						LS
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I	I	I
31																									7						0

Unsigned 8-bit char

																									MS						LS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U
31																									7						0

Signed 16-bit short

																MS															LS
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																15															0

Unsigned 16-bit short

																MS															LS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																15															0

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

7.2.1.2 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 7-2). Objects of these types are loaded to and stored from bits 0-31 of a register. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15, moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

Figure 7-2. 32-Bit Data Storage Format

Single-precision floating char

									MS																						LS
S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31								23																							0

Signed 32-bit integer, or enum char

MS																															LS
S	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																															0

Unsigned 32-bit integer

MS																															LS
U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																															0

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

7.2.1.3 long Data Types (signed and unsigned)

Long and unsigned long data types are stored in an odd/even pair of registers (see Figure 7-3) 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 7-3. 40-Bit Data Storage Format Signed 40-bit long

Odd register

																									MS						
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	S	I	I	I	I	I	I
31																								8	7	6					0

Even register

																															LS
I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																															0

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

Figure 7-4. Unsigned 40-bit long

Odd register

																									MS						
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	U	U	U	U	U	U	U
31																								8	7						0

Even register

																															LS
U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																															0

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

7.2.1.4 long long Data Types (signed and unsigned)

Long long and unsigned long long data types are stored in an odd/even pair of registers (see Figure 7-5) and are always referenced as a pair in the format of odd register:even register (for example, A1:A0). In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the highest byte of the odd register.

Figure 7-5. 64-Bit Data Storage Format Signed 64-bit long

Odd register

MS																															
S	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																															0

Even register

																															LS
I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																															0

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

Figure 7-6. Unsigned 64-bit long

Odd register

MS																															
U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																															0

Even register

																															LS
U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																															0

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

7.2.1.5 double and long double Data Types

Double and long double data types are stored in an odd/even pair of registers (see Figure 7-7) 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 7-7. Double-Precision Floating-Point Data Storage Format

Odd register

													MS																		
S	E	E	E	E	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31												20																			0

Even register

																															LS
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31																															0

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

7.2.1.6 Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (32 bit) integral type. Its value is the byte offset to the data member in the class, plus 1. The zero value is reserved to represent the NULL pointer to the data member.

7.2.1.7 Pointer to Member Function Types

Pointer to member function objects are stored as a structure with three members, and the layout is equivalent to:

```
struct {
   short int d;
   short int i;
   union {
      void (*f) ();
      int 0; }
   };
```

The parameter d is the offset to be added to the beginning of the class object for this pointer. The parameter I is the index into the virtual function table, offset by 1. The index enables the NULL pointer to be represented. Its value is -1 if the function is nonvirtual. The parameter f is the pointer to the member function if it is nonvirtual, when I is 0. The 0 is the offset to the virtual function pointer within the class object.

7.2.1.8 Structures and Arrays

A nested structure is aligned to a boundary required by the largest type it contains. For example, if the largest type in a nested structure is of type short, then the nested structure is aligned to a 2-byte boundary. If the largest type in a nested structure is of type long, unsigned long, double, or long double, then the nested structure is aligned to an 8-byte boundary.

Structures always reserve memory in multiples of the size of the largest element type. For example, if a structure contains an int, unsigned int, or float, a multiple of 4 bytes of storage is reserved in memory. Members of structures are stored in the same manner as if they were individual objects.

Arrays are aligned on an 8-byte boundary for C6400 and C6400+, and either a 4-byte (for all element types of 32 bits or smaller) or an 8-byte boundary for C6200, C6700, or C6700+. Elements of arrays are stored in the same manner as if they were individual objects.

7.2.2 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.

Figure 7-8 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 7-8. Bit-Field Packing in Big-Endian and Little-Endian Formats

Big-endian register

MS LS

A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	B	C	C	C	D	D	E	E	E	E	E	E	E	E	E	X
6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	2	1	0	1	0	8	7	6	5	4	3	2	1	0	X

31 0

Big-endian memory

Byte 0								Byte 1								Byte 2								Byte 3							
A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	B	C	C	C	D	D	E	E	E	E	E	E	E	E	E	X
6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	2	1	0	1	0	8	7	6	5	4	3	2	1	0	X

Little-endian register

MS LS

X	E	E	E	E	E	E	E	E	E	D	D	C	C	C	B	B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A
X	8	7	6	5	4	3	2	1	0	1	0	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0

31 0

Little-endian memory

Byte 0								Byte 1								Byte 2								Byte 3							
B	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	E	E	D	D	C	C	C	B	X	E	E	E	E	E	E	E
0	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	1	0	1	0	2	1	0	9	X	8	7	6	5	4	3	2

LEGEND: X = not used, MS = most significant, LS = least significant

7.2.3 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 7.8.

- 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! */
```

7.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 7-2 summarizes how the compiler uses the TMS320C6000 registers.

The registers in Table 7-2 are available to the compiler for allocation to register variables and temporary expression results. If the compiler cannot allocate a register of a required type, spilling occurs. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

Objects of type double, long, long long, or long double are allocated into an odd/even register pair and are always referenced as a register pair (for example, A1:A0). The odd register contains the sign bit, the exponent, and the most significant part of the mantissa. The even register contains the least significant part of the mantissa. The A4 register is used with A5 for passing the first argument if the first argument is a double, long, long long, or long double. The same is true for B4 and B5 for the second parameter, and so on. For more information about argument-passing registers and return registers, see Section 7.4.

Table 7-2. Register Usage

Register	Function Preserved By	Special Uses	Register	Function Preserved By	Special Uses
A0	Parent	–	B0	Parent	–
A1	Parent	–	B1	Parent	–
A2	Parent	–	B2	Parent	–
A3	Parent	Structure register (pointer to a returned structure)	B3	Parent	Return register (address to return to)
A4	Parent	Argument 1 or return value	B4	Parent	Argument 2
A5	Parent	Argument 1 or return value with A4 for doubles, longs and long longs	B5	Parent	Argument 2 with B4 for doubles, longs and long longs
A6	Parent	Argument 3	B6	Parent	Argument 4
A7	Parent	Argument 3 with A6 for doubles, longs, and long longs	B7	Parent	Argument 4 with B6 for doubles, longs, and long longs
A8	Parent	Argument 5	B8	Parent	Argument 6
A9	Parent	Argument 5 with A8 for doubles, longs, and long longs	B9	Parent	Argument 6 with B8 for doubles, longs, and long longs
A10	Child	Argument 7	B10	Child	Argument 8
A11	Child	Argument 7 with A10 for doubles, longs, and long longs	B11	Child	Argument 8 with B10 for doubles, longs, and long longs
A12	Child	Argument 9	B12	Child	Argument 10
A13	Child	Argument 9 with A12 for doubles, longs, and long longs	B13	Child	Argument 10 with B12 for doubles, longs, and long longs
A14	Child	–	B14	Child	Data page pointer (DP)
A15	Child	Frame pointer (FP)	B15	Child	Stack pointer (SP)
A16-A31	Parent	C6400, C6400+, and C6700+ only	B16-B31	Parent	C6400, C6400+, and C6700+ only
ILC	Child	C6400+ and C6740 only, loop buffer counter	NRP	Parent	
IRP	Parent		RILC	Child	C6400+ and C6740 only, loop buffer counter

All other control registers are not saved or restored by the compiler.

The compiler assumes that control registers not listed in Table 7-2 that can have an effect on compiled code have default values. For example, the compiler assumes all circular addressing-enabled registers are set for linear addressing (the AMR is used to enable circular addressing). Enabling circular addressing and then calling a C/C++ function without restoring the AMR to a default setting violates the calling convention. You must be certain that control registers which affect compiler-generated code have a default value when calling a C/C++ function from assembly.

Assembly language programmers must be aware that the linker assumes B15 contains the stack pointer. The linker needs to save and restore values on the stack in trampoline code that it generates. If you do not use B15 as the stack pointer in assembly code, you should use the linker option that disables trampolines, --trampolines=off. Otherwise, trampolines could corrupt memory and overwrite register values.

7.4 Function Structure and Calling Conventions

The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

7.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function):

1. Arguments passed to a function are placed in registers or on the stack.

   If arguments are passed to a function, up to the first ten arguments are placed in registers A4, B4, A6, B6, A8, B8, A10, B10, A12, and B12. If longs, long longs, doubles, or long doubles are passed, they are placed in register pairs A5:A4, B5:B4, A7:A6, and so on.

   Any remaining arguments are placed on the stack (that is, the stack pointer points to the next free location; SP + *offset* points to the eleventh argument, and so on). Arguments placed on the stack must be aligned to a value appropriate for their size. An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int. An argument that is a float is passed as double if it has no prototype declared.

   A structure argument is passed as the address of the structure. It is up to the called function to make a local copy.

   For a function declared with an ellipsis indicating that it is called with varying numbers of arguments, the convention is slightly modified. The last explicitly declared argument is passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments.

   Figure 7-9 shows the register argument conventions.
2. The calling function must save registers A0 to A9 and B0 to B9 (and A16 to A31 and B16 to B31 for C6400, C6400+, and C6700+), if their values are needed after the call, by pushing the values onto the stack.
3. The caller (parent) calls the function (child).
4. Upon returning, the caller reclaims any stack space needed for arguments by adding to the stack pointer. This step is needed only in assembly programs that were not compiled from C/C++ code. This is because the C/C++ compiler allocates the stack space needed for all calls at the beginning of the function and deallocates the space at the end of the function.

Figure 7-9. Register Argument Conventions

`int func1(`	`int a,`	`int b,`	`int c);`				
A4	A4	B4	A6				
`int func2(`	`int a,`	`float b,`	`int c,`	`struct A d,`	`float e,`	`int f,`	`int g);`
A4	A4	B4	A6	B6	A8	B8	A10
`int func3(`	`int a,`	`double b,`	`float c,`	`long double d;`			
A4	A4	B5:B4	A6	B7:B6			
`/* NOTE: The following function has a variable number of arguments */`							
`int vararg(int a,`		`int b,`	`int c,`	`int d,`	`..);`		
A4	A4	B4	A6	stack	...		
`struct A func4(`		`int y);`					
A3		A4					

7.4.2 How a Called Function Responds

A called function (child function) must perform the following tasks:

1. The called function (child) allocates enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function and may include the allocation of the frame pointer (FP).

   The frame pointer is used to read arguments from the stack and to handle register spilling instructions. If any arguments are placed on the stack or if the frame size exceeds 128K bytes, the frame pointer (A15) is allocated in the following manner:

   a. The old A15 is saved on the stack.
   b. The new frame pointer is set to the current SP (B15).
   c. The frame is allocated by decrementing SP by a constant.
   d. Neither A15 (FP) nor B15 (SP) is decremented anywhere else within this function.

   If the above conditions are not met, the frame pointer (A15) is not allocated. In this situation, the frame is allocated by subtracting a constant from register B15 (SP). Register B15 (SP) is not decremented anywhere else within this function.
2. If the called function calls any other functions, the return address must be saved on the stack. Otherwise, it is left in the return register (B3) and is overwritten by the next function call.
3. If the called function modifies any registers numbered A10 to A15 or B10 to B15, it must save them, either in other registers or on the stack. The called function can modify any other registers without saving them.
4. If the called function expects a structure argument, it receives a pointer to the structure instead. If writes are made to the structure from within the called function, space for a local copy of the structure must be allocated on the stack and the local structure must be copied from the passed pointer to the structure. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.

   You must be careful to declare functions properly that accept structure arguments, both at the point where they are called (so that the structure argument is passed as an address) and at the point where they are declared (so the function knows to copy the structure to a local copy).
5. The called function executes the code for the function.
6. If the called function returns any integer, pointer, or float type, the return value is placed in the A4 register. If the function returns a double, long double, long, or long long type, the value is placed in the A5:A4 register pair.

If the function returns a structure, the caller allocates space for the structure and passes the address of the return space to the called function in A3. To return a structure, the called function copies the structure to the memory block pointed to by the extra argument.

In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can actually make the call as f(&s, x). The function f then copies the return structure directly into s, performing the assignment automatically.

If the caller does not use the return structure value, an address value of 0 can be passed as the first argument. This directs the called function not to copy the return structure.

You must be careful to declare functions properly that return structures, both at the point where they are called (so that the extra argument is passed) and at the point where they are declared (so the function knows to copy the result).

7. Any register numbered A10 to A15 or B10 to B15 that was saved in Step 1 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.

7.4.3 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through register A15 (FP) or through register B15 (SP), one of which points to the top of the stack. Since the stack grows toward smaller addresses, the local and argument data for a function are accessed with a positive offset from FP or SP. Local variables, temporary storage, and the area reserved for stack arguments to functions called by this function are accessed with offsets smaller than the constant subtracted from FP or SP at the beginning of the function.

Stack arguments passed to this function are accessed with offsets greater than or equal to the constant subtracted from register FP or SP at the beginning of the function. The compiler attempts to keep register arguments in their original registers if optimization is used or if they are defined with the register keyword. Otherwise, the arguments are copied to the stack to free those registers for further allocation.

For information on whether FP or SP is used to access local variables, temporary storage, and stack arguments, see Section 7.4.2. For more information on the C/C++ System stack, see Section 7.1.2.

7.5 Interfacing C and C++ With Assembly Language

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

- Use separate modules of assembled code and link them with compiled C/C++ modules (see Section 7.5.1).
- Use assembly language variables and constants in C/C++ source (see Section 7.5.2).
- Use inline assembly language embedded directly in the C/C++ source (see Section 7.5.3).
- Use intrinsics in C/C++ source to directly call an assembly language statement (see Section 7.5.4).

7.5.1 Using Assembly Language Modules With C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in Section 7.4, and the register conventions defined in Section 7.3. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- All functions, whether they are written in C/C++ or assembly language, must follow the register conventions outlined in Section 7.3.

- 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.
- A10 to A15 and B10 to B15 need to be restored before a function returns, even if any of A10 to A13 and B10 to B13 are being used for passing arguments.
- Interrupt routines must save *all* the registers they use. For more information, see Section 7.6.
- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in Section 7.4.1.

  Remember that only A10 to A15 and B10 to B15 are preserved by the C/C++ compiler. C/C++ functions can alter any other registers, save any other registers whose contents need to be preserved by pushing them onto the stack before the function is called, and restore them after the function returns.
- Functions must return values correctly according to their C/C++ declarations. Integers and 32-bit floating-point (float) values are returned in A4. Doubles, long doubles, longs, and long longs are returned in A5:A4. Structures are returned by copying them to the address in A3.
- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C/C++ startup routine assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.
- The compiler assigns linknames to all external objects. Thus, when you are writing assembly language code, you must use the same linknames as those assigned by the compiler. See Section 6.9 for more information.
- Any object or function declared in assembly language that is accessed or called from C/C++ must be declared with the .def or .global directive in the assembly language modifier. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.
- The SGIE bit of the TSR control register may need to be saved. Please see Section 7.6.1 for more information.
- The compiler assumes that control registers not listed in Table 7-2 that can have an effect on compiled code have default values. For example, the compiler assumes all circular-addressing-enabled registers are set for linear addressing (the AMR is used to enable circular addressing). Enabling circular addressing and then calling a C/C++ function without restoring the AMR to a default setting violates the calling convention. Also, enabling circular addressing and having interrupts enabled violates the calling convention. You must be certain that control registers that affect compiler-generated code have a default value when calling a C/C++ function from assembly.
- Assembly language programmers must be aware that the linker assumes B15 contains the stack pointer. The linker needs to save and restore values on the stack in trampoline code that it generates. If you do not use B15 as the stack pointer in your assembly code, you should use the linker option that disables trampolines, --trampolines=off. Otherwise, trampolines could corrupt memory and overwrite register values.
- Assembly code that utilizes B14 and/or B15 for localized purposes other than the data-page pointer and stack pointer may violate the calling convention. The assembly programmer needs to protect these areas of non-standard use of B14 and B15 by turning off interrupts around this code. Because interrupt handling routines need the stack (and thus assume the stack pointer is in B15) interrupts need to be turned off around this code. Furthermore, because interrupt service routines may access global data and may call other functions which access global data, this special treatment also applies to B14. After the data-page pointer and stack pointer have been restored, interrupts may be turned back on.

Example 7-1 illustrates a C++ function called main, which calls an assembly language function called asmfunc, Example 7-2. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 7-1. Calling an Assembly Language Function From C/C++ C Program

```
extern "C" {
extern int asmfunc(int a); /* declare external as function */
int gvar = 4;              /* define global variable       */
}

void main()
{
   int I = 5;

   I = asmfunc(I);         /* call function normally       */
```

Example 7-2. Assembly Language Program Called by Example 7-1

```
   .global _asmfunc
   .global _gvar
_asmfunc:
   LDW     *+b14(_gvar),A3
   NOP     4
   ADD     a3,a4,a3
   STW     a3,*b14(_gvar)
   MV      a3,a4
   B       b3
   NOP     5
```

In the C++ program in Example 7-1, the extern declaration of asmfunc is optional because the return type is int. Like C/C++ functions, you need to declare assembly functions only if they return noninteger values or pass noninteger parameters.

SP Semantics

Note: The stack pointer must always be 8-byte aligned. This is automatically performed by the C compiler and system initialization code in the run-time-support libraries. Any hand assembly code that has interrupts enabled or calls a function defined in C or linear assembly source should also reserve a multiple of 8 bytes on the stack.

Stack Allocation

Note: Even though the compiler guarantees a doubleword alignment of the stack and the stack pointer (SP) points to the next free location in the stack space, there is only enough guaranteed room to store one 32-bit word at that location. The called function must allocate space to store the doubleword.

7.5.2 Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables or constants defined in assembly language. There are several methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a constant.

7.5.2.1 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. Use the appropriate linkname in assembly language.
5. In C/C++, declare the variable as *extern* and access it normally.

Example 7-4 and Example 7-3 show how you can access a variable defined in .bss.

Example 7-3. Assembly Language Variable 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
```

Example 7-4. C Program to Access Assembly Language From Example 7-3

```
extern int var1;       /* External variable */
extern far int var2;   /* External variable */
var1 = 1;              /* Use the variable  */
var2 = 1;              /* Use the variable  */
```

7.5.2.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set, .def, and .global directives, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

For normal variables defined in C/C++ or assembly language, the symbol table contains the *address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the compiler attempts to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C/C++ is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 7-5 and Example 7-6.

Example 7-5. Accessing an Assembly Language Constant From C

```
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 */
```

Example 7-6. Assembly Language Program for Example 7-5

```
_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 7-5, int is used. You can reference linker-defined symbols in a similar manner.

7.5.3 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 6.7.

The asm statement is useful for inserting comments in the compiler output. Simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

Note: Using the asm Statement

Keep the following in mind when using the asm statement:

- Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.
- Avoid inserting jumps or labels into C/C++ code because they can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
- Do not change the value of a C/C++ variable when using an asm statement. This is because the compiler does not verify such statements. They are inserted as is into the assembly code, and potentially can cause problems if you are not sure of their effect.
- Do not use the asm statement to insert assembler directives that change the assembly environment.
- Avoid creating assembly macros in C code and compiling with the --symdebug:dwarf (or -g) option. The C environment's debug information and the assembly macro expansion are not compatible.

7.5.4 Using Intrinsics to Access Assembly Language Statements

The C6000 compiler recognizes a number of intrinsic operators. Intrinsics allow you to express the meaning of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Intrinsics are used like functions; you can use C/C++ variables with these intrinsics, just as you would with any normal function.

The intrinsics are specified with a leading underscore, and are accessed by calling them as you do a function. For example:

```
int x1, x2, y;
y = _sadd(x1, x2);
```

The intrinsics listed in Table 7-3 are included for all C6000 devices. They correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

Intrinsic Instructions in C Versus Assembly Language

Note: In some instances, an intrinsic's exact corresponding assembly language instruction may not be used by the compiler. When this is the case, the meaning of the program does not change.

See Table 7-4 for the listing of C6400-specific intrinsics. See Table 7-5 for the listing of C6400+- and C6740-specific intrinsics. See Table 7-6 for the listing of C6700-specific intrinsics.

Table 7-3. TMS320C6000 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs (** int *src***);** int **_labs (** long *src***);**	**ABS**	Returns the saturated absolute value of src
int **_add2 (**int *src1*, int *src2***);**	**ADD2**	Adds the upper and lower halves of src1 to the upper and lower halves of src2 and returns the result. Any overflow from the lower half add does not affect the upper half add.
ushort & **_amem2 (**void **ptr***);**	**LDHU** **STHU**	Allows aligned loads and stores of 2 bytes to memory[(1)]
const ushort & **_amem2_const (**const void **ptr***);**	**LDHU**	Allows aligned loads of 2 bytes from memory[(1)]
unsigned & **_amem4 (**void **ptr***);**	**LDW** **STW**	Allows aligned loads and stores of 4 bytes to memory[(1)]
const unsigned & **_amem4_const (**const void **ptr***);**	**LDW**	Allows aligned loads of 4 bytes from memory[(1)]
double & **_amemd8 (**void **ptr***);**	**LDW/LDW** **STW/STW**	Allows aligned loads and stores of 8 bytes to memory[(1)(2)]. For C6400 _amemd corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-4 for specifics.
const double & **_amemd8_const (**const void **ptr***);**	**LDDW**	Allows aligned loads of 8 bytes from memory[(1)(2)]
unsigned **_clr (**unsigned *src2,* unsigned *csta*, unsigned *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.
unsigned **_clrr (**unsigned *src2*, int *src1***);**	**CLR**	Clears the specified field in src2. The beginning and ending bits of the field to be cleared are specified by the lower 10 bits of src1.
ulong **_dtol (**double *src***);**		Reinterprets double register pair src as an unsigned long register pair
int **_ext (**int *src2***,** unsigned *csta***,** unsigned *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.
unsigned **_extu (**unsigned *src2***,** unsigned *csta***,** unsigned *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.
unsigned **_extur (**unsigned *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.
unsigned **_ftoi (**float *src***);**		Reinterprets the bits in the float as an unsigned. For example: _ftoi (1.0) == 1065353216U
unsigned **_hi (**double *src***);**		Returns the high (odd) register of a double register pair
unsigned **_hill (**long long *src***);**		Returns the high (odd) register of a long long register pair
double **_itod (**unsigned *src2***,** unsigned *src1***);**		Builds a new double register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register
float **_itof (**unsigned *src***);**		Reinterprets the bits in the unsigned as a float. For example: _itof (0x3f800000)==1.0
long long **_itoll (**unsigned *src2***,** unsigned *src1***);**		Builds a new long long register pair by reinterpreting two unsigned values, where src2 is the high (odd) register and src1 is the low (even) register
unsigned **_lmbd (**unsigned *src1***,** unsigned *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.
unsigned **_lo (**double *src***);**		Returns the low (even) register of a double register pair
unsigned **_loll (**long long *src***);**		Returns the low (even) register of a long long register pair
double **_ltod (**long *src***);**		Reinterprets long register pair src as a double register pair

(1) See the *TMS320C6000 Programmer's Guide* for more information.
(2) See Section 7.5.6 for details on manipulating 8-byte data quantities.

Table 7-3. TMS320C6000 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_mpy (**int *src1*, int *src2***);** int **_mpyus (**unsigned *src1*, int *src2***);** int **_mpysu (**int *src1*, unsigned *src2***);** unsigned **_mpyu (**unsigned *src1*, unsigned *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 (**unsigned *src1*, int *src2***);** int **_mpyhsu (**int *src1*, unsigned *src2***);** unsigned **_mpyhu (**unsigned *src1*, unsigned *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 (**unsigned *src1*, int *src2***);** int **_mpyhslu (**int *src1*, unsigned *src2***);** unsigned **_mpyhlu (**unsigned *src1*, unsigned *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 (**unsigned *src1*, int *src2***);** int **_mpylshu (**int *src1*, unsigned *src2***);** unsigned **_mpylhu (**unsigned *src1*, unsigned *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.
unsigned **_norm (**int *src2***);** unsigned **_lnorm (**long *src2***);**	**NORM**	Returns the number of bits up to the first nonredundant sign bit of src2
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
unsigned **_set (**unsigned *src2*, unsigned *csta*, unsigned *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.
unsigned **_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 0x80000000, saturates the result to 0x7FFFFFFF
int **_sshl (**int *src2*, unsigned *src1***);**	**SSHL**	Shifts src2 left by the contents of src1, saturates the result to 32 bits, and returns the result
int **_ssub (**int *src1*, int *src2***);** long **_lssub (**int *src1*, long *src2***);**	**SSUB**	Subtracts src2 from src1, saturates the result, and returns the result
unsigned **_subc (**unsigned *src1*, unsigned *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.

The intrinsics listed in Table 7-4 are included only for C6400 devices. The intrinsics shown correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000CPU and Instruction Set Reference Guide* for more information.

See Table 7-3 for the listing of generic C6000 intrinsics. See Table 7-5 for the listing of C6400+- and C6740-specific intrinsics. See Table 7-6 for the listing of C6700-specific intrinsics.

Table 7-4. TMS320C6400 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs2 (**int *src***);**	**ABS2**	Calculates the absolute value for each 16-bit value
int **_add4 (**int *src1***,** int *src2***);**	**ADD4**	Performs 2s-complement addition to pairs of packed 8-bit numbers
long long & **_amem8 (**void **ptr***);**	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory.
const long long & **_amem8_const (**const void **ptr***);**	**LDDW**	Allows aligned loads of 8 bytes from memory.[1]
double & **_amemd8 (**void **ptr***);**	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory[2][1]. For C6400 _amemd corresponds to different assembly instructions than when used with other C6000 devices; see Table 7-3.
const double & **_amemd8_const (**const void **ptr***);**	**LDDW**	Allows aligned loads of 8 bytes from memory[2][1]
int **_avg2** (int *src1***,** int *src2***);**	**AVG2**	Calculates the average for each pair of signed 16-bit values
unsigned **_avgu4 (**unsigned**,** unsigned**);**	**AVGU4**	Calculates the average for each pair of signed 8-bit values
unsigned **_bitc4 (**unsigned *src***);**	**BITC4**	For each of the 8-bit quantities in src, the number of 1 bits is written to the corresponding position in the return value
unsigned **_bitr (**unsigned *src***);**	**BITR**	Reverses the order of the bits
int **_cmpeq2 (**int *src1***,** int *src2***);**	**CMPEQ2**	Performs equality comparisons on each pair of 16-bit values. Equality results are packed into the two least-significant bits of the return value.
int **_cmpeq4 (**int *src1***,** int *src2***);**	**CMPEQ4**	Performs equality comparisons on each pair of 8-bit values. Equality results are packed into the four least-significant bits of the return value.
int **_cmpgt2 (**int *src1***,** int *src2***);**	**CMPGT2**	Compares each pair of signed 16-bit values. Results are packed into the two least-significant bits of the return value.
unsigned **_cmpgtu4 (**unsigned *src1***,** unsigned *src2***);**	**CMPGTU4**	Compares each pair of 8-bit values. Results are packed into the four least-significant bits of the return value.
unsigned **_deal (**unsigned *src***);**	**DEAL**	The odd and even bits of src are extracted into two separate 16-bit values.
int **_dotp2 (**int *src1***,** int *src2***);** double **_ldotp2 (**int *src1***,** int *src2***);**	**DOTP2** **DOTP2**	The product of the signed lower 16-bit values of src1 and src2 is added to the product of the signed upper 16-bit values of src1 and src2.The _lo and _hi intrinsics are needed to access each half of the 64-bit integer result.
int **_dotpn2 (**int *src1***,** int *src2***);**	**DOTPN2**	The product of the signed lower 16-bit values of src1 and src2 is subtracted from the product of the signed upper 16-bit values of src1 and src2.
int **_dotpnrsu2 (**int *src1***,** unsigned *src2***);**	**DOTPNRSU2**	The product of the lower unsigned 16-bit values in src1 and src2 is subtracted from the product of the signed upper 16-bit values of src1 and src2. 2^15 is added and the result is sign shifted right by 16.
int **_dotprsu2 (**int *src1***,** unsigned *src2***);**	**DOTPRSU2**	The product of the first signed pair of 16-bit values is added to the product of the unsigned second pair of 16-bit values. 2^15 is added and the result is sign shifted by 16.
int **_dotpsu4 (**int *src1***,** unsigned *src2***);** unsigned **_dotpu4 (**unsigned *src1***,** unsigned *src2***);**	**DOTPSU4** **DOTPU4**	For each pair of 8-bit values in src1 and src2, the 8-bit value from src1 is multiplied with the 8-bit value from src2. The four products are summed together.
int **_gmpy4 (**int *src1*, int *src2***);**	**GMPY4**	Performs the Galois Field multiply on four values in src1 with four parallel values in src2. The four products are packed into the return value.
int **_max2 (**int *src1***,** int *src2***);** int **_min2 (**int *src1***,** int *src2***);** unsigned **_maxu4 (**unsigned *src1***,** unsigned *src2***);** unsigned **_minu4 (**unsigned *src1***,** unsigned *src2***);**	**MAX2** **MIN2** **MAX4** **MINU4**	Places the larger/smaller of each pair of values in the corresponding position in the return value. Values can be 16-bit signed or 8-bit unsigned.
ushort & **_mem2 (**void * *ptr***);**	**LDB/LDB** **STB/STB**	Allows unaligned loads and stores of 2 bytes to memory[2]
const ushort & **_mem2_const (**const void * *ptr***);**	**LDB/LDB**	Allows unaligned loads of 2 bytes to memory[2]
unsigned & **_mem4 (**void * *ptr***);**	**LDNW** **STNW**	Allows unaligned loads and stores of 4 bytes to memory[2]

(1) See Section 7.5.6 for details on manipulating 8-byte data quantities.
(2) See the *TMS320C6000 Programmer's Guide* for more information.

Table 7-4. TMS320C6400 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
const unsigned & **_mem4_const (**const void * *ptr***);**	**LDNW**	Allows unaligned loads of 4 bytes from memory[(2)]
long long & **_mem8 (**void * *ptr***);**	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory[(2)]
const long long & **_mem8_const (**const void * *ptr***);**	**LDNDW**	Allows unaligned loads of 8 bytes from memory[(2)]
double & **_memd8 (**void * *ptr***);**	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory[(1)(2)]
const double & **_memd8_const (**const void * *ptr***);**	**LDNDW**	Allows unaligned loads of 8 bytes from memory[(1)(2)]
double **_mpy2 (**int *src1***,** int *src2***);** long long **_mpy2ll (**int *src1***,** int *src2***);**	**MPY2**	Returns the products of the lower and higher 16-bit values in src1 and src2
double **_mpyhi (**int *src1***,** int *src2***);** double **_mpyli (**int *src1***,** int *src2***);** long long **_mpyhill (**int *src1***,** int *src2***);** long long **_mpylill (**int *src1***,** int *src2***);**	**MPYHI** **MPYLI**	Produces a 16 by 32 multiply. The result is placed into the lower 48 bits of the return type. Can use the upper or lower 16 bits of src1.
int **_mpyhir (**int *src1***,** int *src2***);** int **_mpylir (**int *src1***,** int *src2***);**	**MPYHIR** **MPYLIR**	Produces a signed 16 by 32 multiply. The result is shifted right by 15 bits. Can use the upper or lower 16 bits of src1.
double **_mpysu4 (**int *src1***,** unsigned *src2***);** double **_mpyu4 (**unsigned *src1***,** unsigned *src2***);** long long **_mpysu4ll (**int *src1***,** unsigned *src2***);** long long **_mpyu4ll (**unsigned *src1***,** unsigned *src2***);**	**MPYSU4** **MPYU4**	For each 8-bit quantity in src1 and src2, performs an 8-bit by 8-bit multiply. The four 16-bit results are packed into a 64-bit result. The results can be signed or unsigned.
int **_mvd (**int *src2***);**	**MVD**	Moves the data from src2 to the return value over four cycles using the multiplier pipeline
unsigned **_pack2 (**unsigned *src1***,** unsigned *src2***);** unsigned **_packh2 (**unsigned *src1***,** unsigned *src2***);**	**PACK2** **PACKH2**	The lower/upper halfwords of src1 and src2 are placed in the return value.
unsigned **_packh4 (**unsigned *src1***,** unsigned *src2***);** unsigned **_packl4 (**unsigned *src1***,** unsigned *src2***);**	**PACKH4** **PACKL4**	Packs alternate bytes into return value. Can pack high or low bytes.
unsigned **_packhl2 (**unsigned *src1***,** unsigned *src2***);** unsigned **_packlh2 (**unsigned *src1***,** unsigned *src2***);**	**PACKHL2** **PACKLH2**	The upper/lower halfword of src1 is placed in the upper halfword the return value. The lower/upper halfword of src2 is placed in the lower halfword the return value.
unsigned **_rotl (**unsigned *src1***,** unsigned *src2***);**	**ROTL**	Rotates src2 to the left by the amount in src1
int **_sadd2 (**int *src1***,** int *src2***);** int **_saddus2 (**unsigned *src1***,** int *src2***);**	**SADD2** **SADDUS2**	Performs saturated addition between pairs of 16-bit values in src1 and src2. Values for src1 can be signed or unsigned.
unsigned **_saddu4 (**unsigned *src1***,** unsigned *src2***);**	**SADDU4**	Performs saturated addition between pairs of 8-bit unsigned values in src1 and src2.
unsigned **_shfl (**unsigned *src2***);**	**SHFL**	The lower 16 bits of src2 are placed in the even bit positions, and the upper 16 bits of src are placed in the odd bit positions.
unsigned **_shlmb (**unsigned *src1***,** unsigned *src2***);** unsigned **_shrmb (**unsigned *src1***,** unsigned *src2***);**	**SHLMB** **SHRMB**	Shifts src2 left/right by one byte, and the most/least significant byte of src1 is merged into the least/most significant byte position.
int **_shr2 (**int *src1***,** unsigned *src2***);** unsigned **shru2 (**unsigned *src1***,** unsigned *src2***);**	**SHR2** **SHRU2**	For each 16-bit quantity in src2, the quantity is arithmetically or logically shifted right by src1 number of bits. src2 can contain signed or unsigned values
double **_smpy2 (**int *src1***,** int *sr2***);** long long **_smpy2ll (**int *src1***,** int *sr2***);**	**SMPY2**	Performs 16-bit multiplication between pairs of signed packed 16-bit values, with an additional 1 bit left-shift and saturate into a 64-bit result.
int **_spack2 (**int *src1***,** int *sr2***);**	**SPACK2**	Two signed 32-bit values are saturated to 16-bit values and packed into the return value
unsigned **_spacku4 (**int *src1***,** int *sr2***);**	**SPACKU4**	Four signed 16-bit values are saturated to 8-bit values and packed into the return value
int **_sshvl (**int *src2***,** int *src1***);** int **_sshvr (**int *src2***,** int *src1***);**	**SSHVL** **SSHVR**	Shifts src2 to the left/right src1 bits. Saturates the result if the shifted value is greater than MAX_INT or less than MIN_INT.
int **_sub4 (**int *src1***,** int *src2***);**	**SUB4**	Performs 2s-complement subtraction between pairs of packed 8-bit values
int **_subabs4 (**int *src1***,** int *src2***);**	**SUBABS4**	Calculates the absolute value of the differences for each pair of packed 8-bit values
unsigned **_swap4 (**unsigned *src***);**	**SWAP4**	Exchanges pairs of bytes (an endian swap) within each 16-bit value
unsigned **_unpkhu4 (**unsigned *src***);**	**UNPKHU4**	Unpacks the two high unsigned 8-bit values into unsigned packed 16-bit values

Table 7-4. TMS320C6400 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
unsigned **_unpklu4 (**unsigned *src***);**	**UNPKLU4**	Unpacks the two low unsigned 8-bit values into unsigned packed 16-bit values
unsigned **_xpnd2 (**unsigned *src***);**	**XPND2**	Bits 1 and 0 of src are replicated to the upper and lower halfwords of the result, respectively.
unsigned **_xpnd4 (**unsigned *src***);**	**XPND4**	Bits 3 and 0 of src are replicated to bytes 3 through 0 of the result.

The intrinsics listed in Table 7-5 are included only for C6400+ and C6740 devices. The intrinsics shown correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

See Table 7-3 for the listing of generic C6000 intrinsics. See Table 7-4 for the general listing of C6400-specific intrinsics. See Table 7-6 for the listing of C6700-specific intrinsics.

Table 7-5. TMS320C6400+ and C6740 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
long long **_addsub (**int *src1***,** int *src2***);**	**ADDSUB**	Performs an addition and subtraction in parallel.
long long **_addsub2 (**int *src1***,** int *src2***);**	**ADDSUB2**	Performs an ADD2 and SUB2 in parallel.
long long **_cmpy (**unsigned *src1***,** unsigned *src2***);** unsigned **_cmpyr (**unsigned *src1***,** unsigned *src2***);** unsigned **_cmpyr1 (**unsigned *src1***,** unsigned *src2***);**	**CMPY** **CMPYR** **CMPYR1**	Performs various complex multiply operations.
long long **_ddotp4 (**unsigned *src1***,** unsigned *src2***);**	**DDOTP4**	Performs two DOTP2 operations simultaneously.
long long **_ddotph2 (**long long *src1***,** unsigned *src2***);** long long **_ddotpl2 (**long long *src1***,** unsigned *src2***);** unsigned **_ddotph2r (**long long *src1***,** unsigned *src2***);** unsigned **_ddotpl2r (**long long *src1***,** unsigned *src2***);**	**DDOTPH2** **DDOTPL2** **DDOTPH2R** **DDOTPL2**	Performs various dual dot-product operations between two pairs of signed, packed 16-bit values.
long long **_dmv (**int *src1***,** int *src2***);**	**DMV**	Places src1 in the 32 LSBs of the long long and src2 in the 32 MSBs of the long long. See also _itoll().
long long **_dpack2 (**unsigned *src1***,** unsigned *src2***);**	**DPACK2**	PACK2 and PACKH2 operations performed in parallel.
long long **_dpackx2 (**unsigned *src1***,** unsigned *src2***);**	**DPACKX2**	PACKLH2 and PACKX2 operations performed in parallel.
unsigned **_gmpy (**unsigned *src1*, unsigned *src2***);**	**GMPY**	Performs the Galois Field multiply.
int **_mpy32 (**int *src1***,** int *src2***);**	**MPY32**	Returns the 32 LSBs of a 32 by 32 multiply.
long long **_mpy32ll (**int *src1***,** int *src2***);** long long **_mpy32su (**int *src1***,** int *src2***);** long long **_mpy32us (**unsigned *src1***,** int *src2***);** long long **_mpy32u (**unsigned *src1***,** unsigned *src2***);**	**MPY32** **MPY32SU** **MPY32US** **MPY32U**	Returns all 64 bits of a 32 by 32 multiply. Values can be signed or unsigned.
long long **_mpy2ir (**int *src1***,** int *src2***);**	**MPY2IR**	Performs two 16 by 32 multiplies. Both results are shifted right by 15 bits to produce a rounded result.
int **_rpack2 (**int *src1***,** int *src2***);**	**RPACK2**	Shifts src1 and src2 left by 1 with saturation. The 16 MSBs of the shifted src1 is placed in the 16 MSBs of the long long. The 16 MSBs of the shifted src2 is placed in the 16 LSBs of the long long.
long long **_saddsub (**unsigned *src1***,** unsigned *src2***);**	**SADDSUB**	Performs a saturated addition and a saturated subtraction in parallel.
long long **_saddsub2 (**unsigned *src1***,** unsigned *src2***);**	**SADDSUB2**	Performs a SADD2 and a SSUB2 in parallel.
long long **_shfl3 (**unsigned *src1***,** unsigned *src2***);**	**SHFL3**	Takes two 16-bit values from src1 and 16 LSBs from src2 to perform a 3-way interleave, creating a 48-bit result.
int **_smpy32 (**int *src1***,** int *src2***);**	**SMPY32**	Returns the 32 MSBs of a 32 by 32 multiply shifted left by 1.
int **_ssub2 (**unsigned *src1*, unsigned *src2***);**	**SSUB2**	Subtracts the upper and lower halves of src2 from the upper and lower halves of src1 and saturates each result.
unsigned **_xormpy (**unsigned *src1***,** unsigned *src2***);**	**XORMPY**	Performs a Galois Field multiply

The intrinsics listed in Table 7-6 are included only for C6700 devices. The intrinsics shown correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

See Table 7-3 for the listing of generic C6000 intrinsics. See Table 7-4 for the listing of C6400-specific intrinsics. See Table 7-5 for the listing of C6400+- and C6740-specific intrinsics.

Table 7-6. TMS320C6700 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_dpint (**double *src***);**	**DPINT**	Converts 64-bit double to 32-bit signed integer, using the rounding mode set by the CSR register
double **_fabs (**double *src***);** float **_fabsf (**float *src***);**	**ABSDP** **ABSSP**	Returns absolute value of src
double **_mpyid (**int *src1***,** int *src2***);**	**MPYID**	Produces a signed integer multiply. The result is placed in a register pair.
double**_mpysp2dp (**float *src1***,** float *src2***);**	**MPYSP2DP**	(C6700+ only) Produces a double-precision floating-point multiply. The result is placed in a register pair.
double**_mpyspdp (**float *src1***,** double *src2***);**	**MPYSPDP**	(C6700+ only) Produces a double-precision floating-point multiply. The result is placed in a register pair.
double **_rcpdp (**double *src***);**	**RCPDP**	Computes the approximate 64-bit double reciprocal
float **_rcpsp (**float *src***);**	**RCPSP**	Computes the approximate 32-bit float reciprocal
double **_rsqrdp (**double *src***);**	**RSQRDP**	Computes the approximate 64-bit double square root reciprocal
float **_rsqrsp (**float *src***);**	**RSQRSP**	Computes the approximate 32-bit float square root reciprocal
int **_spint (**float**);**	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register

7.5.5 Using Intrinsics for Interrupt Control and Atomic Sections

The C/C++ compiler supports three intrinsics for enabling, disabling, and restoring interrupts. The syntaxes are:

unsigned int	_disable_interrupts ( );
unsigned int	_enable_interrupts ( );
void	_restore_interupts (unsigned int);

The _disable_interrupts() and _enable_interrupts( ) intrinsics both return an unsigned int that can be subsequently passed to _restore_interrupts( ) to restore the previous interrupt state. These intrinsics provide a barrier to optimization and are therefore appropriate for implementing a critical (or atomic) section. For example,

```
unsigned int restore_value;

restore_value = _disable_interrupts();
if (sem) sem--;
_restore_interrupts(restore_value);
```

The example code disables interrupts so that the value of sem read for the conditional clause does not change before the modification of sem in the then clause. The intrinsics are barriers to optimization, so the memory reads and writes of sem do not cross the _disable_interrupts or _restore_interrupts locations.

Overwrites CSR

Note: The _restore_interrupts( ) intrinsic overwrites the CSR control register with the value in the argument. Any CSR bits changed since the _disable_interrupts( ) intrinsic or _enable_interrupts( ) intrinsic will be lost.

On C6400+ and C6740, the _restore_interrupts( ) intrinsic does not use the RINT instruction.

7.5.6 Using Unaligned Data and 64-Bit Values

The C6400, C6400+, and C6740 families have support for unaligned loads and stores of 64-bit and 32-bit values via the use of the _mem8, _memd8, and _mem4 intrinsics. The _lo and _hi intrinsics are useful for extracting the two 32-bit portions from a 64-bit double. The _loll and _hill intrinsics are useful for extracting the two 32-bit portions from a 64-bit long long.

For the C6400+ and C6740 intrinsics that use 64-bit types, the equivalent C type is long long. Do not use the C type double or the compiler performs a call to a run-time-support math function to do the floating-point conversion. Here are ways to access 64-bit and 32-bit values:

- To get the upper 32 bits of a long long in C code, use >> 32 or the _hill( ) intrinsic.
- To get the lower 32 bits of a long long in C code, use a cast to int or unsigned, or use the _loll intrinsic.
- To get the upper 32 bits of a double (interpreted as an int), use _hi( ).
- To get the lower 32 bits of a double (interpreted as an int), use _lo( ).
- To create a long long value, use the _itoll(int high32bits, int low32bits) intrinsic.

Example 7-7 and Example 7-8 shows the usage of the _lo, _hi, _mem8, and _memd8 intrinsics.

Example 7-7. Using the _lo, _hi, and _memd8 Intrinsics

```
void load_longlong_unaligned(void *a, int *high, int *low)
{
  double d = _memd8(a);

  *high = _hi(d);
  *low  = _lo(d);
}
```

Example 7-8. Using the _mem8 Intrinsic

```
void alt_load_longlong_unaligned(void *a, int *high, int *low)
{
  long long p = _mem8(a);

  *high = p >> 32;
  *low  = (unsigned int) p;
}
```

7.5.7 Using MUST_ITERATE and _nassert to Enable SIMD and Expand Compiler Knowledge of Loops

Through the use of MUST_ITERATE and _nassert, you can guarantee that a loop executes a certain number of times.

This example tells the compiler that the loop is guaranteed to run exactly 10 times:

```
#pragma MUST_ITERATE(10,10);
for (I = 0; I <; trip_count; I++) { ...
```

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

```
#pragma MUST_ITERATE(8,48,8);
for (I = 0; I <; trip; I++) { ...
```

This example tells the compiler that the loop executes between 8 and 48 times and that the trip variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The compiler can now use all this information to generate the best loop possible by unrolling better even when the --interrupt_threshold*n* option is used to specify that interrupts do occur every *n* cycles.

The *TMS320C6000 Programmer's Guide* states that one of the ways to refine C/C++ code is to use word accesses to operate on 16-bit data stored in the high and low parts of a 32-bit register. Examples using casts to int pointers are shown with the use of intrinsics to use certain instructions like _mpyh. This can be automated by using the _nassert(); intrinsic to specify that 16-bit short arrays are aligned on a 32-bit (word) boundary.

The following two examples generate the same assembly code:

- **Example 1**

```
int dot_product(short *x, short *y, short z)
{
    int *w_x = (int *)x;
    int *w_y = (int *)y;
    int sum1 = 0, sum2 = 0, I;
    for (I = 0; I < z/2; I++)
    {
        sum1 += _mpy(w_x[i], w_y[i]);
        sum2 += _mpyh(w_x[i], w_y[i]);
    }
    return (sum1 + sum2);
}
```

- **Example 2**

```
int dot_product(short *x, short *y, short z)
{
    int sum = 0, I;

    _nassert (((int)(x) & 0x3) == 0);
    _nassert (((int)(y) & 0x3) == 0);
    #pragma MUST_ITERATE(20, , 4);
    for (I = 0; I < z; I++) sum += x[i] * y[i];
    return sum;
}
```

C++ Syntax for _nassert

Note: In C++ code, _nassert is part of the standard namespace. Thus, the correct syntax is std::_nassert().

7.5.8 Methods to Align Data

In the following code, the _nassert tells the compiler, for every invocation of f(), that ptr is aligned to an 8-byte boundary. Such an assertion often leads to the compiler producing code which operates on multiple data values with a single instruction, also known as SIMD (single instruction multiple data) optimization.

```
void f(short *ptr)
{
   _nassert((int) ptr % 8 == 0)

   ; a loop operating on data accessed by ptr
}
```

The following subsections describe methods you can use to ensure the data referenced by ptr is aligned. You have to employ one of these methods at every place in your code where f() is called.

7.5.8.1 Base Address of an Array

An argument such as ptr is most commonly passed the base address of an array, for example:

```
short buffer[100];

...

f(buffer);
```

When compiling for C6400, C6400+, and C6740 devices, such an array is automatically aligned to an 8-byte boundary. When compiling for C6200 or C6700, such an array is automatically aligned to 4-byte boundary, or, if the base type requires it, an 8-byte boundary. This is true whether the array is global, static, or local. This automatic alignment is all that is required to achieve SIMD optimization on those respective devices. You still need to include the _nassert because, in the general case, the compiler cannot guarantee that ptr holds the address of a properly aligned array.

If you always pass the base address of an array to pointers like ptr, then you can use the following macro to reflect that fact.

```
#if   defined(_TMS320C6400)
   #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 8 == 0)
#elif defined(_TMS320C6200) || defined(_TMS320C6700)
   #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 4 == 0)
#else
   #define ALIGNED_ARRAY(ptr) /* empty */
#endif

void f(short *ptr)
{
   ALIGNED_ARRAY(ptr);

   ; a loop operating on data accessed by ptr
}
```

The macro works regardless of which C6x device you build for, or if you port the code to another target.

7.5.8.2 Offset from the Base of an Array

A more rare case is to pass the address of an offset from an array, for example:

```
f(&buffer[3]);
```

This code passes an unaligned address to ptr, thus violating the presumption coded in the _nassert(). There is no direct remedy for this case. Avoid this practice whenever possible.

7.5.8.3 Dynamic Memory Allocation

Ordinary dynamic memory allocation does not guarantee that the address of the buffer is aligned, for example:

```
buffer = calloc(100 * sizeof(short));
```

You should use memalign() with an alignment of 8 instead, for example:

```
buffer = memalign(8, 100 * sizeof(short);
```

If you are using BIOS memory allocation routines, be sure to pass the alignment factor as the last argument using the syntax that follows:

buffer = **MEM_alloc(***segid***,** 100 * sizeof(short), **8);**

See the *TMS320C6000 DSP/BIOS Help* for more information about BIOS memory allocation routines and the *segid* parameter in particular.

7.5.8.4 Member of a Structure or Class

Arrays which are members of a structure or a class are aligned only as the base type of the array requires. The automatic alignment described in Section 7.5.8.1 does not occur.

Example 7-9. An Array in a Structure

```
struct s
{
   ...
   short buf1[50];
   ...
}  g;

 ...

 f(g.buf1);
```

Example 7-10. An Array in a Class

```
class c
{
   public :
      short buf1[50];
      void mfunc(void);
   ...
};

void c::mfunc()
{
   f(buf1);
   ...
}
```

The most straightforward way to align an array in a structure or class is to declare, right before the array, a scalar that requires the desired alignment. So, if you want 8-byte alignment, use a long or double. If you want 4-byte alignment, use an int or float. For example:

```
struct s
{
    long  not_used;         /* 8-byte aligned      */
    short buffer[50];    /* also 8-byte aligned */
   ...
};
```

If you want to declare several arrays contiguously, and maintain a given alignment, you can do so by keeping the array size, measured in bytes, an even multiple of the desired alignment. For example:

```
struct s
{
   long  not_used;    /* 8-byte aligned      */
   short buf1[50];    /* also 8-byte aligned */
   short buf2[50];    /* 4-byte aligned      */
   ...
};
```

Because the size of buf1 is 50 * 2-bytes per short = 100 bytes, and 100 is an even multiple of 4, not 8, buf2 is only aligned on a 4-byte boundary. Padding buf1 out to 52 elements makes buf2 8-byte aligned.

Within a structure or class, there is no way to enforce an array alignment greater than 8. For the purposes of SIMD optimization, this is not necessary.

Alignment With Program-Level Optimization

Note: In most cases program-level optimization (see Section 3.7) entails compiling all of your source files with a single invocation of the compiler, while using the -pm -o3 options. This allows the compiler to see all of your source code at once, thus enabling optimizations that are rarely applied otherwise. Among these optimizations is seeing that, for instance, all of the calls to the function f() are passing the base address of an array to ptr, and thus ptr is always correctly aligned for SIMD optimization. In such a case, the _nassert() is not required. The compiler automatically determines that ptr must be aligned, and produces the optimized SIMD instructions.

7.5.8.5 SAT Bit Side Effects

The saturated intrinsic operations define the SAT bit if saturation occurs. The SAT bit can be set and cleared from C/C++ code by accessing the control status register (CSR). The compiler uses the following steps for generating code that accesses the SAT bit:

1. The SAT bit becomes undefined by a function call or a function return. This means that the SAT bit in the CSR is valid and can be read in C/C++ code until a function call or until a function returns.
2. If the code in a function accesses the CSR, then the compiler assumes that the SAT bit is live across the function, which means:
   - The SAT bit is maintained by the code that disables interrupts around software pipelined loops.
   - Saturated instructions cannot be speculatively executed.
3. If an interrupt service routine modifies the SAT bit, then the routine should be written to save and restore the CSR.

7.5.8.6 IRP and AMR Conventions

There are certain assumptions that the compiler makes about the IRP and AMR control registers. The assumptions should be enforced in all programs and are as follows:

1. The AMR must be set to 0 upon calling or returning from a function. A function does not have to save and restore the AMR, but must ensure that the AMR is 0 before returning.
2. The AMR must be set to 0 when interrupts are enabled, or the SAVE_AMR and STORE_AMR macros should be used in all interrupts (see Section 7.6.3).
3. The IRP can be safely modified only when interrupts are disabled.
4. The IRP's value must be saved and restored if you use the IRP as a temporary register.

7.6 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If the system is initialized by way of a hardware reset, interrupts are disabled. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

7.6.1 Saving the SGIE Bit

When compiling for C6400+ and C6740, the compiler may use the C6400+ and C6740-specific instructions DINT and RINT to disable and restore interrupts around software-pipelined loops. These instructions utilize the CSR control register as well as the SGIE bit in the TSR control register. Therefore, the SGIE bit is considered to be save-on-call. If you have assembly code that calls compiler-generated code, the SGIE bit should be saved (e.g. to the stack) if it is needed later. The SGIE bit should then be restored upon return from compiler generated code.

7.6.2 Saving Registers During Interrupts

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. The compiler handles register preservation if the interrupt service routine is written in C/C++ and declared with the interrupt keyword. For C6400+ and C6740, the compiler will save and restore the ILC and RILC control registers if needed.

7.6.3 Using C/C++ Interrupt Routines

A C/C++ interrupt routine is like any other C/C++ function in that it can have local variables and register variables; however, it should be declared with no arguments and should return void. C/C++ interrupt routines can allocate up to 32K on the stack for local variables. For example:

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

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler attempts to define are saved and restored. However, if a C/C++ interrupt routine *does* call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the routine saves all usable registers if any other functions are called. Interrupts branch to the interrupt return pointer (IRP). Do not call interrupt handling functions directly.

Interrupts can be handled *directly* with C/C++ functions by using the interrupt pragma or the interrupt keyword. For more information, see Section 6.8.14 and Section 6.4.3.

You are responsible for handling the AMR control register and the SAT bit in the CSR correctly inside an interrupt. By default, the compiler does not do anything extra to save/restore the AMR and the SAT bit. Macros for handling the SAT bit and the AMR register are included in the c6x.h header file.

For example, you are using circular addressing in some hand assembly code (that is, the AMR does not equal 0). This hand assembly code can be interrupted into a C code interrupt service routine. The C code interrupt service routine assumes that the AMR is set to 0. You need to define a local unsigned int temporary variable and call the SAVE_AMR and RESTORE_AMR macros at the beginning and end of your C interrupt service routine to correctly save/restore the AMR inside the C interrupt service routine.

Example 7-11. AMR and SAT Handling

```
#include <c6x.h>

interrupt void interrupt_func()
{
   unsigned int temp_amr;
   /* define other local variables used inside interrupt */

   /* save the AMR to a temp location and set it to 0 */
   SAVE_AMR(temp_amr);

   /* code and function calls for interrupt service routine */
   ...

   /* restore the AMR for you hand assembly code before exiting */
   RESTORE_AMR(temp_amr);
}
```

If you need to save/restore the SAT bit (i.e. you were performing saturated arithmetic when interrupted into the C interrupt service routine which may also perform some saturated arithmetic) in your C interrupt service routine, it can be done in a similar way as the above example using the SAVE_SAT and RESTORE_SAT macros.

For C6400+ and C6740, the compiler saves and restores the ILC and RILC control registers if needed.

7.6.4 Using Assembly Language Interrupt Routines

You can handle interrupts with assembly language code as long as you follow the same register conventions the compiler does. Like all assembly functions, interrupt routines can use the stack, access global C/C++ variables, and call C/C++ functions normally. When calling C/C++ functions, be sure that any registers listed in Table 7-2 are saved, because the C/C++ function can modify them.

7.7 Run-Time-Support Arithmetic Routines

The run-time-support library contains a number of assembly language functions that provide arithmetic routines for C/C++ math operations that the C6000 instruction set does not provide, such as integer division, integer remainder, and floating-point operations.

These routines follow the standard C/C++ calling sequence. The compiler automatically adds these routines when appropriate; they are not intended to be called directly by your programs.

The source code for these functions is in the source library rts.src. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of the math functions. Be sure, however, that you follow the calling conventions and register-saving rules outlined in this chapter. Table 7-7 summarizes the run-time-support functions used for arithmetic.

Table 7-7. Summary of Run-Time-Support Arithmetic Functions

Type	Function	Description
float	_cvtdf (double)	Convert double to float
int	_fixdi (double)	Convert double to signed integer
long	_fixdli (double)	Convert double to long
long long	_fixdlli (double)	Convert double to long long
uint	_fixdu (double)	Convert double to unsigned integer
ulong	_fixdul (double)	Convert double to unsigned long
ulong long	_fixdull (double)	Convert double to unsigned long long
double	_cvtfd (float)	Convert float to double
int	_fixfi (float)	Convert float to signed integer
long	_fixfli (float)	Convert float to long
long long	_fixflli (float)	Convert float to long long
uint	_fixfu (float)	Convert float to unsigned integer
ulong	_fixful (float)	Convert float to unsigned long
ulong long	_fixfull (float)	Convert float to unsigned long long
double	_fltid (int)	Convert signed integer to double
float	_fltif (int)	Convert signed integer to float
double	_fltud (uint)	Convert unsigned integer to double
float	_fltuf (uint)	Convert unsigned integer to float
double	_fltlid (long)	Convert signed long to double
float	_fltlif (long)	Convert signed long to float
double	_fltuld (ulong)	Convert unsigned long to double
float	_fltulf (ulong)	Convert unsigned long to float
double	_fltllid (long long)	Convert signed long long to double
float	_fltllif (long long)	Convert signed long long to float
double	_fltulld (ulong long)	Convert unsigned long long to double
float	_fltullf (ulong long)	Convert unsigned long long to float
double	_absd (double)	Double absolute value
float	_absf (float)	Float absolute value
long	_labs (long)	Long absolute value
long long	_llabs (long long)	Long long absolute value

Table 7-7. Summary of Run-Time-Support Arithmetic Functions (continued)

Type	Function	Description
double	_negd (double)	Double negate value
float	_negf (float)	Float negate value
long long	_negll (long)	Long long negate value
long long	_llshl (long long)	Long long shift left
long long	_llshr (long long)	Long long shift right
ulong long	_llshru (ulong long)	Unsigned long long shift right
double	_addd (double, double)	Double addition
double	_cmpd (double, double)	Double comparison
double	_divd (double, double)	Double division
double	_mpyd (double, double)	Double multiplication
double	_subd (double, double)	Double subtraction
float	_addf (float, float)	Float addition
float	_cmpf (float, float)	Float comparison
float	_divf (float, float)	Float division
float	_mpyf (float, float)	Float multiplication
float	_subf (float, float)	Float subtraction
int	_divi (int, int)	Signed integer division
int	_remi (int, int)	Signed integer remainder
uint	_divu (uint, uint)	Unsigned integer division
uint	_remu (uint, uint)	Unsigned integer remainder
long	_divli (long, long)	Signed long division
long	_remli (long, long)	Signed long remainder
ulong	_divul (ulong, ulong)	Unsigned long division
ulong	_remul (ulong, ulong)	Unsigned long remainder
long long	_divlli (long long, long long)	Signed long long division
long long	_remlli (long long, long long)	Signed long long remainder
ulong long	_mpyll(ulong long, ulong long)	Unsigned long long multiplication
ulong long	_divull (ulong long, ulong long)	Unsigned long long division
ulong long	_remull (ulong long, ulong long)	Unsigned long long remainder

7.8 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or 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 --rom_model or --ram_model link option and include a standard run-time-support library as one of the link step input files.

When C/C++ programs are linked, the link step 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. Defines a section called .stack for the system stack and sets up the initial stack pointers
2. Initializes global variables by copying the data from the initialization tables to the storage allocated for the variables in the .bss section. If you are initializing variables at load time (--ram_model option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see Section 7.8.1.
3. Executes the global constructors found in the global constructors table. For more information, see Section 7.8.2.
4. Calls the function main to run the C/C++ program

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

See Section 8.5 for a list of the standard run-time-support libraries that are shipped with the C6000 code generation tools.

7.8.1 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing 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.

Initializing Variables

Note: In ANSI/ISO C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

The easiest method is to set a fill value of 0 in the linker control map for the .bss section.

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

Global variables are either autoinitialized at run time or at load time. For information, see Section 7.8.4 and Section 7.8.5. Also see Section 6.10.

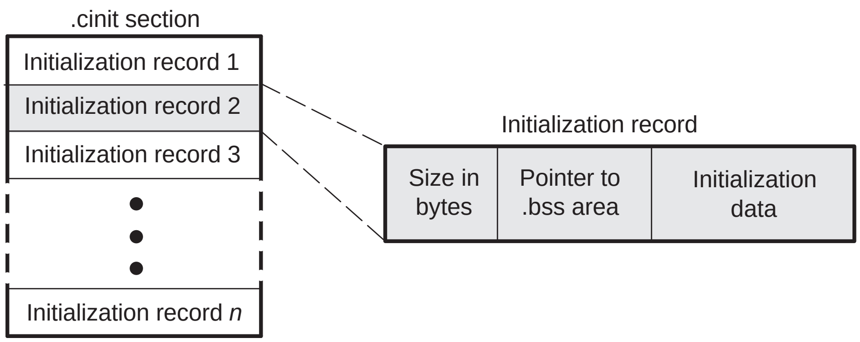
7.8.2 Global Constructors

All global C++ variables that have constructors must have their constructor called before main (). The compiler builds a table of global constructor addresses that must be called, in order, before main () in a section called .pinit. The linker combines the .pinit section form each input file to form a single table in the .pinit section. The boot routine uses this table to execute the constructors.

7.8.3 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has a record in the .cinit section. Figure 7-10 shows the format of the .cinit section and the initialization records.

Figure 7-10. Format of Initialization Records in the .cinit Section



The fields of an initialization record contain the following information:

- The first field of an initialization record contains the size (in bytes) of the initialization data.
- The second field contains the starting address of the area within the .bss section where the initialization data must be copied.
- The third field contains the data that is copied into the .bss section to initialize the variable.

Each variable that must be autoinitialized has an initialization record in the .cinit section.

Example 7-12 shows initialized global variables defined in C. Example 7-13 shows the corresponding initialization table. The section .cinit:c is a subsection in the .cinit section that contains all scalar data. The subsection is handled as one record during initialization, which minimizes the overall size of the .cinit section.

Example 7-12. Initialized Variables Defined in C

```
int x;
short i = 23;
int *p =
int a[5] = {1,2,3,4,5};
```

Example 7-13. Initialized Information for Variables Defined in Example 7-12

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

        .sect   ".cinit:c"
        .align  8
        .field          (CIR - $) - 8, 32
        .field          _I+0,32
        .field         23,16                    ; _I @ 0

        .sect   ".text"
        .global _I
_I:     .usect  ".bss:c",2,2

        .sect   ".cinit:c"
        .align  4
        .field         _x,32                    ; _p @ 0

        .sect   ".text"
        .global _p
_p:     .usect  ".bss:c",4,4
```

Example 7-13. Initialized Information for Variables Defined in Example 7-12 (continued)

```
        .sect   ".cinit"
        .align  8
        .field          IR_1,32
        .field          _a+0,32
        .field         1,32                     ; _a[0] @ 0
        .field         2,32                     ; _a[1] @ 32
        .field         3,32                     ; _a[2] @ 64
        .field         4,32                     ; _a[3] @ 96
        .field         5,32                     ; _a[4] @ 128
IR_1:   .set    20

        .sect   ".text"
        .global _a
        .bss    _a,20,4
;**********************************************************************
;* MARK THE END OF THE SCALAR INIT RECORD IN CINIT:C                  *
;**********************************************************************
CIR:    .sect   ".cinit:c"
```

The .cinit section must contain only initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other purpose.

The table in the .pinit section simply consists of a list of addresses of constructors to be called (see Figure 7-11). The constructors appear in the table after the .cinit initialization.

Figure 7-11. Format of Initialization Records in the .pinit Section

.pinit section
Address of constructor 1
Address of constructor 2
Address of constructor 3
• • •
Address of constructor *n*

When you use the --rom_model or --ram_model option, the linker combines the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0 and marks the end of the initialization tables.

Likewise, the --rom_model or --ram_model link option causes the linker to combine all of the .pinit sections from all C/C++ modules and append a null word to the end of the composite .pinit section. The boot routine knows the end of the global constructor table when it encounters a null constructor address.

The const-qualified variables are initialized differently; see Section 6.4.1.

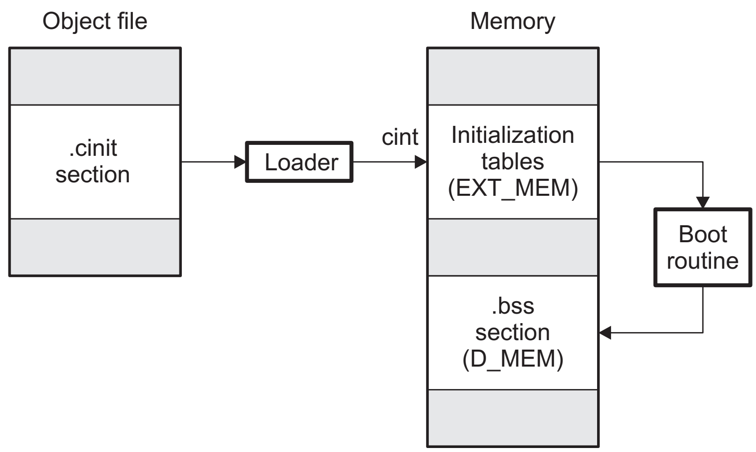
7.8.4 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections, and global variables are initialized at run time. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 7-12 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 7-12. Autoinitialization at Run Time



7.8.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

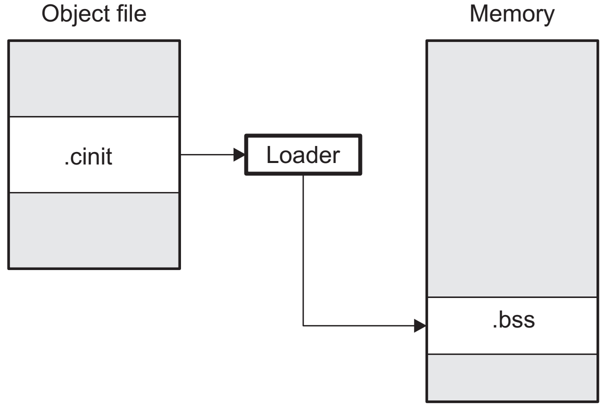
When you use the --ram_model link option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 7-13 illustrates the initialization of variables at load time.

Figure 7-13. Initialization at Load Time



Regardless of the use of the --rom_model or --ram_model options, the .pinit section is always loaded and processed at run time.

Using Run-Time-Support Functions and Building Libraries

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

In addition to the ANSI/ISO-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. These are detailed in Section 8.1 and Section 8.2.

A library-build process is provided with the code generation tools that lets you create customized run-time-support libraries. This process is described in Section 8.5 .

8.1 C and C++ Run-Time Support Libraries

The standard run-time library includes code for both the C and C++ libraries, as well as compiler helper functions and initialization code. The library includes all the features of the C++ Library, including the Standard Template Library (STL), streams, and strings. The following limitations are noted:

- The library has minimal support for wide and multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The C library includes the header file <locale.h> but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale via a call to setlocale() will return NULL.

The C++ library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described in Section 6.1.

The C++ library included with the compiler is licensed from Dinkumware, Ltd. The Dinkumware C++ library is a fully conforming, industry-leading implementation of the standard C++ library.

Table 8-1 summarizes the functionality of the C++ standard library.

Table 8-1. C++ Standard Library Outline

Header	Description	Notes
	C Library API	
<cassert>	Assertions	
<cctype>	Character Classifications	
<cerrno>	Error indicator	
<cfloat>	Floating-point properties	
<ciso646>	Named logical operators	
<climits>	Data type properties	
<clocale>	Locale support	Supports C locale only
<cmath>	Floating-point math functions	
<csetjmp>	Non-local jumps	
<csignal>	Signal and raise	
<cstdarg>	Variadic arguments	
<cstddef>	Standard C definitions	
<cstdio>	C standard I/O	
<cstdlib>	Utility functions	
<cstring>	C character strings	
<ctime>	C time manipulation	
<cwchar>	Wide char functions	Not fully supported
<cwctype>	Wide char classification	Not fully supported
	Standard Template Library	
<algorithm>	Search, sort, etc.	
<complex>	Complex number arithmetic	
<deque>	Double-ended queue	
<functional>	Function objects	

Table 8-1. C++ Standard Library Outline (continued)

Header	Description	Notes
<hash_map>	Map keys to values	Extension
<hash_set>	Map keys to mulivalues	Extension
<iterator>	Iterators for standard containers	
<list>	Linked list	
<map>	Associative array container	
<memory>	Container memory management	
<numeric>	Various numeric functions	
<queue>	Queue container	
<rope>	Null-terminated array	Extension
<set>	General container	
<slist>	Singly-linked list	Extension
<stack>	Stack container	
<utility>	Operators and pairs	
<valarray>	Numeric vectors	
<vector>	One-dimensional array	
I/O Streams		
<fstream>	I/O streams to/from files	
<iomanip>	Manipulate I/O streams	
<ios>	I/O stream base class	
<iosfwd>	Forward declarations of I/O classes	
<iostream>	Standard I/O stream operators	
<istream>	Input stream template	
<ostream>	Output stream template	
<sstream>	I/O streams operations on allocated arrays	
<streambuf>	I/O buffer base class	
<strstream>	I/O streams to/from strings	
Strings		
<string>	C++ style string objects	
Language / Utility		
<bitset>	Array of booleans	
<exception>	Exception handling control	
<limits>	Data type properties	
<locale>	Customizing I/O and other facilities	
<new>	Dynamic memory allocation operators	
<stdexcept>	Exception reporting objects	

TI does not provide documentation that covers the functionality of the C++ library. We suggest referring to one of the following sources:

- *The Standard C++ Library: A Tutorial and Reference*,Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5
- Dinkumware's online reference at http://dinkumware.com/manuals

8.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See Section 5.4.1 for further information.

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 --reread_libs linker option to force repeated searches of each library until the linker can resolve no more references.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *TMS320C6000 Assembly Language Tools User's Guide*.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

8.1.2 Header Files

Set the C6X_C_DIR environment variable to the include directory where the tools are installed. The source for the libraries is included in the rtssrc.zip file. See Section 8.5 for details on rebuilding.

8.1.3 Modifying a Library Function

You can inspect or modify library functions by unzipping the source file (rtssrc.zip), changing the specific function file, and rebuilding the library. When extracted (with any standard unzip tool on windows, linux, or unix), this zip file will recreate the run-time source tree for the run-time library.

8.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O. The capability to perform I/O on the host gives you more options when debugging and testing code.

To use the I/O functions, include the header file stdio.h, or cstdio for C++ code, for each module that references a C I/O function.

For example, given the following C program in a file named main.c:

```
#include <stdio.h>;

main()
{
   FILE *fid;

   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);

   printf("Hello again, world\n");
}
```

Issuing the following compiler command compiles, links, and creates the file main.out from the run-time-support library:

```
cl6x main.c --run_linker --heap_size=400 --library=rts6200.lib --output_file=main.out
```

Executing main.out results in

```
Hello, world
```

being output to a file and

```
Hello again, world
```

being output to your host's stdout window.

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

Note: C I/O Buffer Failure

If there is not enough space on the heap for a C I/O buffer, buffered operations on the file will fail. If a call to printf() mysteriously fails, this may be the reason. Check the size of the heap. To set the heap size, use the --heap_size option when linking (see Section 5.2).

8.2.1 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.

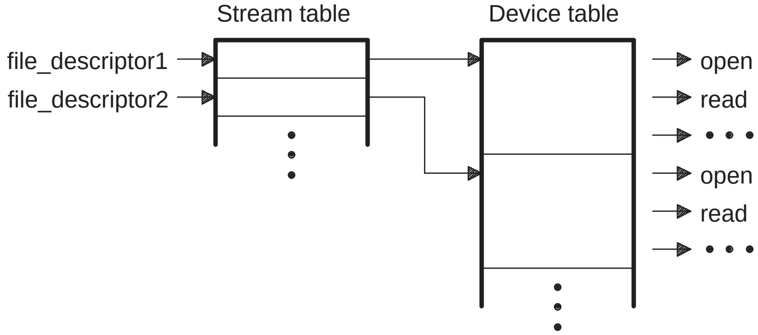
The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These routines map an I/O request to one or more of the I/O commands that are handled by the low-level routines.

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.

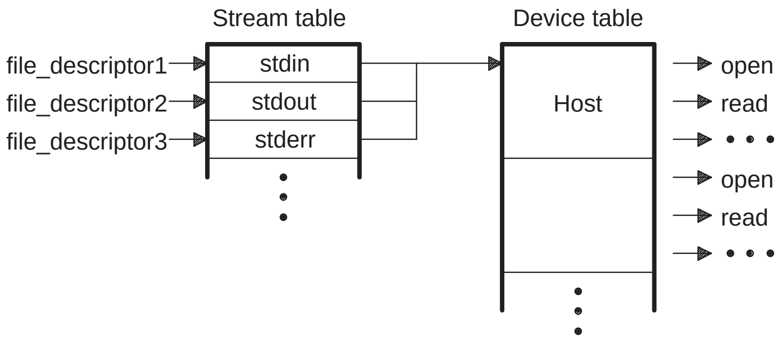
The data structures interact as shown in Figure 8-1.

Figure 8-1. 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 8-2. The First Three Streams in the Stream Table



At the next level are the user-definable device-level drivers. They map directly to the low-level I/O functions. The run-time-support library includes the device drivers necessary to perform I/O on the host on which the debugger is running.

The specifications for writing device-level routines to interface with the low-level routines follow. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

add_device — *Add Device to Device Table*

Syntax for C

#include <file.h>

int add_device(char * *name*,
  unsigned *flags*,
  int (* *dopen* **)( ),**
  int (* *dclose* **)( ),**
  int (* *dread* **)( ),**
  int (* *dwrite* **)( ),**
  fpos_t (* *dlseek* **)( ),**
  int (* *dunlink* **)( ),**
  int (* *drename* **)( ));**

Defined in

lowlev.c in rtssrc.zip

Description

The add_device function adds a device record to the device table allowing that device to be used for input/output from C. The first entry in the device table is predefined to be the host device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen( ) with a string of the format *devicename* **:** *filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in stdio.h.
- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *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 8.2.1. 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

Example 8-1 does the following:

- Adds the device *mydevice* to the device table
- Opens a file named *test* on that device and associates it with the file **fid*
- Writes the string *Hello, world* into the file
- Closes the file

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

Syntax for C

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

int close (int *file_descriptor* **);**

Syntax for C++

#include <cstdio>
#include <file.h>

int std::close (int *file_descriptor* **);**

Description

The close function closes the device or file associated with *file_descriptor*.

The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened device or file.

Return Value

The return value is one of the following:

0	if successful
1	if fails

lseek — *Set File Position Indicator*

Syntax for C

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

long lseek (int*file_descriptor*, **long** *offset*, **int***origin***);**

Syntax for C++

#include <cstdio>
#include <file.h>

long std::lseek (int *file_descriptor* **, long** *offset* **, int** *origin* **);**

Description

The lseek function sets the file position indicator for the given file to *origin* + *offset*. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the stream number assigned by the low-level routines that the device-level driver must associate with the opened file or device.
- The *offset* indicates the relative offset from the *origin* in characters.
- The *origin* is used to indicate which of the base locations the *offset* is measured from. The *origin* must be a value returned by one of the following macros:

  SEEK_SET (0x0000) Beginning of file

  SEEK_CUR (0x0001) Current value of the file position indicator

  SEEK_END (0x0002) End of file

Return Value

The return function is one of the following:

#	new value of the file-position indicator if successful
EOF	if fails

open — *Open File or Device for I/O*

Syntax for C

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

int open (const char * *path* **, unsigned** *flags* **, int** *file_descriptor* **);**

Syntax for C++

#include <cstdio>
#include <file.h>

int std::open (const char * *path* **, unsigned** *flags* **, int** *file_descriptor* **);**

Description

The open function opens the device or file specified by *path* and prepares it for I/O.

- The *path* is the filename of the file to be opened, including path information.
- The *flags* are attributes that specify how the device or file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY   (0x0000)   /* open for reading */
O_WRONLY   (0x0001)   /* open for writing */
O_RDWR     (0x0002)   /* open for read & write */
O_APPEND   (0x0008)   /* append on each write */
O_CREAT    (0x0200)   /* open with file create */
O_TRUNC    (0x0400)   /* open with truncation */
O_BINARY   (0x8000)   /* open in binary mode */
```

  These parameters can be ignored in some cases, depending on how data is interpreted by the device. However, the high-level I/O calls look at how the file was opened in an fopen statement and prevent certain actions, depending on the open attributes.
- The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened file or device.

  The next available file_descriptor (in order from 3 to 20) is assigned to each new device opened. You can use the finddevice() function to return the device structure and use this pointer to search the _stream array for the same pointer. The file_descriptor number is the other member of the _stream array.

Return Value

The function returns one of the following values:

#-1	if successful
-1	if fails

read — *Read Characters from Buffer*

Syntax for C

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

int read (int *file_descriptor* **, char *** *buffer* **, unsigned** *count* **);**

Syntax for C++

```
#include <cstdio>
#include <file.h>
```

int std::read (int *file_descriptor* **, char** **buffer* **, unsigned** *count* **);**

Description

The read function reads the number of characters specified by *count* to the *buffer* from the device or file associated with *file_descriptor*.

- The *file_descriptor* is the stream number assigned by the low-level routines that is associated with the opened file or device.
- The *buffer* is the location of the buffer where the read characters are placed.
- The *count* is the number of characters to read from the device or file.

Return Value

The function returns one of the following values:

0	if EOF was encountered before the read was complete
#	number of characters read in every other instance
-1	if fails

rename — *Rename File*

Syntax for C

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

int rename (const char * *old_name* **, const char *** *new_name* **);**

Syntax for C++

```
#include <cstdio>
#include <file.h>
```

int std::rename (const char * *old_name* **, const char *** *new_name* **);**

Description

The rename function changes the name of a file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Return Value

The function returns one of the following values:

0	if successful
Non-0	if not successful

unlink — *Delete File*

Syntax for C

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

int unlink (const char * *path* **);**

Syntax for C++

#include <cstdio>
#include <file.h>

int std::unlink (const char * *path* **);**

Description

The unlink function deletes the file specified by *path*.

The *path* is the filename of the file to be opened, including path information.

Return Value

The function returns one of the following values:

0	if successful
1	if fails

write — *Write Characters to Buffer*

Syntax for C

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

int write (int *file_descriptor* **, const char *** *buffer* **, unsigned** *count* **);**

Syntax for C++

#include <cstdio>
#include <file.h>

int write (int *file_descriptor* **, const char *** *buffer* **, unsigned** *count* **);**

Description

The write function writes the number of characters specified by *count* from the *buffer* to the device or file associated with *file_descriptor*.

- The *file_descriptor* is the stream number assigned by the low-level routines. It is associated with the opened file or device.
- The *buffer* is the location of the buffer where the write characters are placed.
- The *count* is the number of characters to write to the device or file.

Return Value

The function returns one of the following values:

#	number of characters written if successful
1	if fails

8.2.2 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 8.2.1.

Note: Use Unique Function Names

The function names open, close, read, and so on, are used by the low-level routines. Use other names for the device-level functions that you write.

2. Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio. The structure representing a device is also defined in stdio.h/cstdio and is composed of the following fields:

name String for device name

flags Flags that specify whether the device supports multiple streams or not

function pointers Pointers to the device-level functions:

- CLOSE
- LSEEK
- OPEN
- READ
- RENAME
- WRITE
- UNLINK

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function .

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().

Example 8-1 illustrates adding and using a device for C I/O:

Example 8-1. Program for C I/O Device

```
#include <stdio.h>
/****************************************************************************/
/* Declarations of the user-defined device drivers                          */
/****************************************************************************/
extern int  my_open(const char *path, unsigned flags, int fno);
extern int  my_close(int fno);
extern int  my_read(int fno, char *buffer, unsigned count);
extern int  my_write(int fno, const char *buffer, unsigned count);
extern long my_lseek(int fno, long offset, int origin);
extern int  my_unlink(const char *path);
extern int  my_rename(const char *old_name, char *new_name);
main()
{
   FILE *fid;
   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,
                            my_unlink, my_rename);
   fid = fopen("mydevice:test","w");
   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

8.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions)

The C standard assumes only one thread of execution, with the only exception being extremly narrow support for signal handlers. The issue of reentrancy is avoided by not allowing you to do much of anything in a signal handler. However, BIOS applications have multiple threads which need to modify the same global program state, such as the CIO buffer, so reentrancy is a concern.

Part of the problem of reentrancy remains your responsibility, but the run-time-support environment does provide rudimentary support for multi-threaded reentrancy by providing support for critical sections. This implementation does not protect you from reentrancy issues such as calling run-time-support functions from inside interrupts; this remains your responsibility.

The run-time-support environment provides hooks to install critical section primitives. By default, a single-threaded model is assumed, and the critical section primitives are not employed. In a multi-threaded system such as BIOS, the kernel arranges to install semaphore lock primitive functions in these hooks, which are then called when the run-time-support enters code that needs to be protected by a critical section.

Throughout the run-time-support environment where a global state is accessed, and thus needs to be protected with a critical section, there are calls to the function _lock(). This calls the provided primitive, if installed, and acquires the semaphore before proceeding. Once the critical section is finished, _unlock() is called to to release the semaphore.

Usually BIOS is responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications which do not use the BIOS LCK mechanism.

You should not define the functions _lock() and _unlock() functions directly; instead, the installation functions are called to instruct the run-time-support environment to use these new primitives:

```
void _register_lock  (void (  *lock)());

void _register_unlock(void (*unlock)());
```

The arguments to _register_lock() and _register_unlock() should be functions which take no arguments and return no values, and which implement some sort of global semaphore locking:

```
    extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
    static int sema_depth = 0;
    static void my_lock(void)
    {
        while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
        sema_depth++;
    }
    static void my_unlock(void)
    {
        if (!--sema_depth) ATOMIC_CLEAR(sema);
    }
```

The run-time-support nests calls to _lock(), so the primitives must keep track of the nesting level.

8.4 C6700 FastMath Library

The C6700 FastMath Library provides hand-coded assembly-optimized versions of certain math functions. These implementations are two to three times faster than those found in the standard run-time-support library. However, these functions gain speed improvements at the cost of accuracy in the result.

The C6700 FastMath library contains these files:

- fastmath67x.lib—object library for use with little-endian C/C++ code
- fastmath67xe—object library for use with big-endian C/C++ code
- fastmath67x.h—header file to be included with C/C++ code

To use the C67x FastMath library, specify it before the standard run-time-support library when linking your program. For example:

```
cl6x -mv6700 --run_linker myprogram.obj --library=lnk.cmd --library=fastmath67x.lib --
library=rts6700.lib
```

If you are using Code Composer Studio, include the C6700 FastMath library in your project, and ensure it appears before the standard run-time-support library in the Link Order tab in the Build Options dialog box.

For details, refer to the *TMS320C67x FastRTS Library Programmer's Reference*.

8.5 Library-Build Process

When using the C/C++ compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Because it would be cumbersome to include all possible combinations in individual run-time-support libraries, this package includes a basic run-time-support library, rts6200.lib. Also included are library versions that support various C6000 devices and versions that support C++ exception handling.

You can also build your own run-time-support libraries using the self-contained run-time-support build process, which is found in rtssrc.zip. This process is described in this chapter and the archiver described in the *TMS320C6000 Assembly Language Tools User's Guide*.

8.5.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following support items are required:

- Perl version 5.6 or later available as perl

  Perl is a high-level programming language designed for process, file, and text manipulation. It is:

  - Generally available from http://www.perl.org/get.htm
  - Available from ActiveState.com as ActivePerl for the PC
  - Available as part of the Cygwin package for the PC

  It must be installed and added to PATH so it is available at the command-line prompt as perl. To ensure perl is available, open a Command Prompt window and execute:

  ```
  perl -v
  ```

  No special or additional Perl modules are required beyond the standard perl module distribution.

- GNU-compatible command-line make tool, such as gmake

  More information is available from GNU at http://www.gnu.org/software/ make. This file requires a host C compiler to build. GNU make (gmake) is shipped as part of Code Composer Studio on Windows. GNU make is also included in some Unix support packages for Windows, such as the MKS Toolkit, Cygwin, and Interix. The GNU make used on Windows platforms should explicitly report This program built for Windows32 when the following is executed from the Command Prompt window:

  ```
  gmake -h
  ```

8.5.2 Using the Library-Build Process

Once the perl and gmake tools are available, unzip the rtssrc.zip into a new, empty directory. See the Makefile for additional information on how to customize a library build by modifying the LIBLIST and/or the OPT_XXX macros

Once the desired changes have been made, simply use the following syntax from the command-line while in the rtssrc.zip top level directory to rebuild the selected rtsname library.

gmake *rtsname*

To use custom options to rebuild a library, simply change the list of options for the appropriate base listed in Section 8.5.3 and then rebuild the library. See the tables in Section 2.3 for a summary of available generic and C6000-specific options.

To build an library with a completely different set of options, define a new OPT_XXX base, choose the type of library per Section 8.5.3, and then rebuild the library. Not all library types are supported by all targets. You may need to make changes to targets_rts_cfg.pm to ensure the proper files are included in your custom library.

8.5.3 Library Naming Conventions

The names of the C6000 run-time support libraries have been changed to improve the clarity and uniformity of the names given the large number of libraries that now exist. Library names from prior releases will be deprecated, but still supplied for compatibility.

The run-time support libraries now have the following naming scheme:

rtsDeviceEndian[_eh].lib

Device	The device family of the C6000 architecture that the library was built for. This can be one of the following: 6200, 6400, 64plus, 6700, 67plus.	
endian	Indicates endianness:	
		Little-endian library
	e	Big-endian library
_eh	Indicates the library has exception handling support	

For information on the C6700 FastMath source library, fastmathc67x.src, see Section 8.4.

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's signature in its link-level name. The process of encoding the signature into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files or linker output, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tell you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

Topic **Page**

9.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

dem6x [*options*] [*filenames*]

dem6x	Command that invokes the C++ name demangler.
options	Options affect how the name demangler behaves. Options can appear anywhere on the command line. (Options are discussed in Section 9.2.)
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, and the linker map file. If no filenames are specified on the command line, dem6x uses standard in.

By default, the C++ name demangler outputs to standard out. You can use the -o file option if you want to output to a file.

9.2 C++ Name Demangler Options

The following options apply only to the C++ name demangler:

-h	Prints a help screen that provides an online summary of the C++ name demangler options
-o *file*	Outputs to the given file rather than to standard out
-u	Specifies that external names do not have a C++ prefix
-v	Enables verbose mode (outputs a banner)

9.3 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process. Example 9-1 shows a sample C++ program. Example 9-2 shows the resulting assembly that is output by the compiler. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

Example 9-1. C Code for calories_in_a_banana

```
class banana {

public:

    int calories(void);

    banana();

    ~banana();

};

int calories_in_a_banana(void)
{
    banana x;

    return x.calories();
}
```

Example 9-2. Resulting Assembly for calories_in_a_banana

```
 _calories_in_a_banana__Fv:
;** ----------------------------------------------------------------------*
           CALL     .S1      ___ct__6bananaFv  ; |10|
           STW      .D2T2    B3,*SP--(16)      ; |9|
           MVKL     .S2      RL0,B3            ; |10|
           MVKH     .S2      RL0,B3            ; |10|
           ADD      .S1X     8,SP,A4           ; |10|
           NOP               1
RL0:       ; CALL OCCURS                       ; |10|
           CALL     .S1      _calories__6bananaFv ; |12|
           MVKL     .S2      RL1,B3            ; |12|
           ADD      .S1X     8,SP,A4           ; |12|
           MVKH     .S2      RL1,B3            ; |12|
           NOP               2
RL1:       ; CALL OCCURS                       ; |12|
           CALL     .S1      ___dt__6bananaFv  ; |13|
           STW      .D2T1    A4,*+SP(4)        ; |12|
           ADD      .S1X     8,SP,A4           ; |13|
           MVKL     .S2      RL2,B3            ; |13|
           MVK      .S2      0x2,B4            ; |13|
           MVKH     .S2      RL2,B3            ; |13|
RL2:       ; CALL OCCURS                       ; |13|
           LDW      .D2T1    *+SP(4),A4        ; |12|
           LDW      .D2T2    *++SP(16),B3      ; |13|
           NOP               4
           RET      .S2      B3                ; |13|
           NOP               5
           ; BRANCH OCCURS                     ; |13|
```

Executing the C++ name demangler demangles all names that it believes to be mangled. If you enter:

```
dem6x calories_in_a_banana.asm
```

the result is shown in Example 9-3. The linknames in Example 9-2 ___ct__6bananaFv, _calories__6bananaFv, and ___dt__6bananaFv are demangled.

Example 9-3. Result After Running the C++ Name Demangler

```
calories_in_a_banana():
;** ----------------------------------------------------------------------*
        CALL    .S1 banana::banana()        ; |10|
        STW     .D2T2 B3,*SP--(16)          ; |9|
        MVKL    .S2 RL0,B3                  ; |10|
        MVKH    .S2 RL0,B3                  ; |10|
        ADD     .S1X 8,SP,A4                ; |10|
        NOP 1
RL0:    ; CALL OCCURS                       ; |10|
        CALL    .S1 banana::calories()    ; |12|
        MVKL    .S2 RL1,B3                  ; |12|
        ADD .   S1X 8,SP,A4                 ; |12|
        MVKH    .S2 RL1,B3                  ; |12|
        NOP 2
RL1:    ; CALL OCCURS                       ; |12|
        CALL    .S1 banana::~banana()        ; |13|
        STW     .D2T1 A4,*+SP(4)            ; |12|
        ADD     .S1X 8,SP,A4                ; |13|
        MVKL    .S2 RL2,B3                  ; |13|
        MVK .   S2 0x2,B4                   ; |13|
        MVKH .  S2 RL2,B3                   ; |13|
RL2:    ; CALL OCCURS                       ; |13|
        LDW     .D2T1 *+SP(4),A4            ; |12|
        LDW     .D2T2 *++SP(16),B3          ; |13|
        NOP 4
        RET .S2 B3                          ; |13|
        NOP 5
        ; BRANCH OCCURS                     ; |13|
```

Glossary

absolute lister— A debugging tool that allows you to create assembler listings that contain absolute addresses.

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.

ANSI— American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

archive library— A collection of individual files grouped into a single file by the archiver.

archiver— A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

assembler— A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assembly optimizer— A software program that optimizes linear assembly code, which is assembly code that has not been register-allocated or scheduled. The assembly optimizer is automatically invoked with the compiler program, cl6x, when one of the input files has a .sa extension.

assignment statement— A statement that initializes a variable with a value.

autoinitialization— The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time— An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

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 object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte— Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler— A software program that translates C source statements into assembly language source statements.

code generator—A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF— Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

command file—A file that contains options, filenames, directives, or commands 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.

compiler program— A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

compression— The assembler process of converting 32-bit instructions into 16-bit instructions (C6400+ and C6740 only). Depending on the --opt_for_space level, the compiler selects and tailors certain instructions so that the assembler can convert them to 16-bit instructions. Compression can be turned off with the --no_compress option.

configured memory— Memory that the linker has specified for allocation.

constant— A type whose value cannot change.

cross-reference listing—An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

.data section—One of the default object file 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 (as opposed to assembly language instructions, which control the actions of a device).

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.

ELF— Executable and linking format; a system of object files configured according to the System V Application Binary Interface specification.

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 Windows batch files or UNIX shell scripts such as .cshrc or .profile.

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.

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.

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.

hex conversion utility—A utility that converts object files into one of several standard ASCII hexadecimal formats, suitable for loading into an EPROM programmer.

high-level language debugging—The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

hole— An area between the input sections that compose an output section that contains no code.

indirect call—A function call where one function calls another function by giving the address of the called function.

initialization at load time—An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the --ram_model link option. This method initializes variables at load time instead of run time.

initialized section—A section from an object file that will be linked into an executable module.

input section—A section from an object file that will be linked into an executable module.

integrated preprocessor—A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.

interlist feature—A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.

ISO— International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

kernel— The body of a software-pipelined loop between the pipelined-loop prolog and the pipelined-loop epilog.

K&R C—Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ISO C compilers should correctly compile and run without modification.

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.

line-number entry— An entry in a COFF output module that maps lines of assembly code back to the original C source file that created them.

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 performance of your code.

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.

name mangling—A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

object file—An assembled or linked file that contains machine-language object code.

object library—An archive library made up of individual object files.

object module—A linked, executable object file that can be downloaded and executed on a target system.

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.

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*

pipelining— A technique where a second instruction begins executing before the first instruction has been completed. You can have several instructions in the pipeline, each at a different processing stage.

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.

quiet run— An option that suppresses the normal banner and the progress information.

raw data—Executable code or initialized data in an output section.

redundant loops—Two versions of the same loop, where one is a software-pipelined loop and the other is an unpipelined loop. Redundant loops are generated when the TMS320C6000 tools cannot guarantee that the trip count is large enough to pipeline a loop for maximum performance.

relocation— A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

run-time environment—The run time parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

run-time-support functions—Standard ISO functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

run-time-support library—A library file, rts.src, that contains the source for the run time-support functions.

section— A relocatable block of code or data that ultimately will 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.

sign extend—A process that fills the unused MSBs of a value with the value's sign bit.

simulator— A software development system that simulates TMS320C6000 operation.

software pipelining—A technique used by the C/C++ optimizer and the assembly optimizer to schedule instructions from a loop so that multiple iterations of the loop execute in parallel.

source file—A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.

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.

subsection— A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

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.

target system— The system on which the object code you have developed is executed.

.text section— One of the default object file 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.

unconfigured memory— Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section— A object file 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.

variable— A symbol representing a quantity that can assume any of a set of values.

veneer— A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.

word— A 32-bit addressable location in target memory

J

K

L

N

O

P

Z

IMPORTANT NOTICE

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

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

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

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

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

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Clocks and Timers	www.ti.com/clocks	Digital Control	www.ti.com/digitalcontrol
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

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